Contents

1 Introduction

In the short time that software engineering has existed it had to – and still has to – face problems of a kind that are not entirely comparable to other, older engineering disciplines. I will look at some of the data describing the symptoms of these problems, and then ask what is the cause of each problem. This leads to the significance of computer-aided software engineering (CASE), in its original meaning and not the connotation it has with regard to a particular group of 80's and 90's tools, which is what this thesis is about.

Section 1.1 describes some of the evidence that software engineering is not yet a mature discipline. The question is, of course, why is this so, and what could be done about it. Section 1.2 sheds light on the differences that set software engineering apart from more established engineering disciplines. Section 1.3 discusses CASE technology as one of the important factors in software engineering. Eventually, Sect. 1.4 provides the reader with an overview of the AP1 project. Parts of this chapter were published in [142].

1.1 Software Engineering – Not Entirely Unproblematic

Software engineering is special. Not inherently special, but special with regard to the way it is practiced and its success. One of the peculiarities of software engineering is that a surprisingly high number of big software projects fail or significantly exceed their frame of time and cost. In other engineering disciplines such as civil or mechanical engineering,

failure rates as high as in software engineering would simply be unacceptable.

One of the most shocking studies about software project failure is the CHAOS report from 1994 by the Standish Group [212]. Data about 8380 projects were collected from 365 IT executive managers of US companies of all sizes. According to this study 31% of the surveyed software projects failed, and the average cost overrun of software projects was as high as 189% of the original cost estimate. This would mean that nearly every third project was a failure, and that companies spent almost three times as much money for a project as intially expected.

In hindsight, this report is very controversial and its scientific validity questionable [122]. Barry Boehm [23] argues that the study erroneously equates project cancellation with project failure, and that the 31% cancellation rate is not as bad as it might appear. Good project managers identify and terminate infeasible projects early in order to avoid a waste of resources. However, the 1994 CHAOS report remains one of the most cited empirical results about software project failure, and has had a significant impact on scientific research as well as economical and political decisions. Reports like this one have led to the creation and proliferation of the term "software crisis".

In fact, the term "software crisis" dates back to the late 1960s, and in particular to the "NATO Software Engineering Conference" of 1968 in Garmisch, Germany [160]. This conference is often seen as a major milestone, if not the birth, of software engineering as a discipline. According to Brian Randell [185], who was present at and co-edited both the proceedings of the two NATO conferences, those days at Garmisch were filled with a particular concern and enthusiasm about the necessity to improve software development. The attendants agreed that scientists and policy makers had to be alerted about the serious problems of that domain, about the "software crisis" and the need to establish "software engineering".

At that time, the term "software engineering" was rather unknown, and Randell even describes the fact that the term became part of the conference title as deliberately provocative. Already in these early days the situation of software development was perceived to be highly problematic, with a high failure rate of complex software systems, and large cost and time overruns. Consequently, as Randell puts it, "it was fully accepted that the term software engineering expressed a need rather than a reality".

The situation seems to be better now than depicted in the 1994 CHAOS report. Robert Glass [97] suggests that there is not a software crisis anymore, comparing the results of the 1994 and the 2000 CHAOS studies: the latter study shows a project cancellation rate of 23% (compared to 31% in 1994) and a success rate of 28% (compared to only 16% in 1994). But the remaining 49% "challenged" projects, which have significant problems such as time or cost overrun, are still very frequent and similar to the 53% in 1994.

Molokken and Jorgensen [157] summarize several scientific surveys about software de-

velopment projects and show that most projects, i.e. between 60 and 80%, encounter effort and/or schedule overruns. Acording to the surveys the average cost overrun of a software development project lies between 30 and 40%. This can be seen as an improvement but is still far from being satisfactory. There are many spectalular examples of multi-million US dollar software project failures, some of which are described in [40]. And such failures are not just events of the past, but still happen almost everywhere. Robert Charette [40] estimates that billions of dollars are wasted each year on bad software.

Not only has the industrial world of software engineering attracted criticism. Robert Glass [98] admits that the software industry has problems, but argues that the "software crisis" is to a good part a slander coined by software researchers that are ignorant of the actual practice of software engineering. There is a big gap between software engineering research and actual practice. Furthermore, Glass argues that the "software crisis" has served as an opportunistic means to justify "advocacy research" that would often be irrelevant for real practice.

There are indeed many publications that begin by citing the 1994 CHAOS report, thus invoking the dreadful ghost of "software crisis", and then offering unproven solutions. Software engineering research and practice are indeed not as closely related as they could and should be. Fenton, Pfleeger and Glass [87] criticise that research in software engineering is often on a very intuitive, unscientific level, and emphasizes the importance of empirical validation in order to create real value. Robert Glass [98] even uses the term "software research crisis".

Brian Randell describes in [185] how the second NATO conference 1969 in Rome was very different from the first one. One of the reasons apparently was that some people had a hidden agenda, which brings the aforementioned term of "advocacy research" back into mind. Apparently one of the intentions was to convince funding agencies of the immediate need of a software engineering institute, which failed. After the success of the first conference, the second conference was purportedly more characterized by a fragmentation of its attendees into groups and a communication gap, rather than scientific achievement.

1.2 Is Software Special?

Considering the problems of software engineering described in the last section, many people ask: is software engineering different from other engineering disciplines? Often software is compared with hardware, and people ask: is it possible to make software costs drop as rapidly as hardware costs do? Obviously Moore's law does not describe the evolution of software development adequately, and the question remains what it is that sets software apart.

There are different aspects to this question, and most of them have been discussed in the literature. Some of the differences can be explained by historical reasons, and others are considered to be intrinsic and are not always well-understood. In the following sections I will discuss some of the differences that one should be aware of. Note that I have grouped the arguments into sections, although arguments from different sections are often related. There exist many interrelations in a complex matter like this. Nevertheless, I chose to impose a structure in order to improve the clarity of this discourse.

1.2.1 Professionalism

One of the characteristics indicating the maturity of a discipline is the existence of an established profession. Most traditional engineering disciplines such as civil or agricultural engineering have developed along the path of one or more professions, long before they were given a scientific foundation. Naturally, we have to ask ourselves how far software engineering has matured with regard to its professional infrastructure. Given the head start of most other engineering disciplines, it is not surprising to see that software engineering is less established as a profession, despite its tremendous impact on the world and its growth.

McConnell and Tripp [151] compare software engineering with other disciplines with regard to professional development. It becomes obvious that a certain infrastructure for professional development in software engineering has developed in the last decade, but that this infrastructure is still very incomplete compared with more established professions. Despite the significance, there is still a lack of proper accreditation for educational institutions, and hardly any certification or licensing for practitioners. Important professional principles such as accountability are not properly established or legally supported.

In other engineering disciplines the title of "engineer" is considered as a legal status rather than just a descriptive term, so representatives of these disciplines attribute much more significance to it. Therefore some of these representatives insist that its meaning should be not diluted by an immature discipline such as software development. This is exemplified by the fact that at universities software engineering is usually taught in a Department of Computer Science, which is not part of a Faculty of Engineering. Examples of quarrels about the status of software engineering are described in [53].

The question whether the time is ripe to establish a licensing or reliable certification system for software engineers, as it exists for other engineering disciplines, was and remains a contentious issue. A licensing system, which is controlled by the government, would mean that practitioners could only do certain work after having passed an examination. The intention is to minimize the threat to the public, which relies more and more on software systems. Nowadays, software is essential for the regulation of traffic, emergency services, government administration and many other areas, and software failure

can result in substantial damage. The purpose of a licensing system would be to enforce that the professionals working in such areas have proved to be capable of bearing that responsibility.

Jeanette Morgan [158] stresses the need for a well-defined set of generally accepted software engineering practices, and its application for certification purposes. Certification is different to licensing in that it is not government controlled. Instead, it is typically controlled by professional organizations. With a meaningful certification system in place, software development personnel could be chosen and trusted on the basis of such certificates, and many of the common software failures could potentially be avoided. Morgan argues that a consensus on how the software industry operates, manifested in standards for certification, is necessary for it to be operational and trustworthy as a whole.

The 1994 CHAOS report [212] points out that software engineering differs in professionalism to more established engineering disciplines in that it does not have established processes for dealing with failure. It compares the collapsing of a bridge – a failure of civil engineering – with failure in a software project and notes that the former triggers an investigation and analysis of what went wrong, whereas in the latter failure is often covered up, ignored or rationalized. Jeanette Morgan [158] expresses a similar thought using the example of accounting: in the USA, the Financial Accounting Standards Board continually reviews accounting cases and issues pronouncements and rulings in order to prevent problems from reoccurring.

Although, unlike in other engineering disciplines, certification of software engineers is still in its infancy, there are examples of certification systems for software engineers. In 1998, the state of Texas recognized software engineering officially as an engineering discipline. People offering software engineering services to the public now have to acquire a professional engineer's license by passing an examination [53]. Since 2002 the IEEE is offering examinations for software developers to become Certified Software Development Professionals (CSDP). The latter examinations are largely based on the guide to the Software Engineering Body of Knowledge (SWEBOK) [2] that was created in a collaborative effort by the IEEE.

In a panel on SWEBOK on the International Conference on Software Engineering [11], participants expressed the need for clear requirements for the education of software developers, and the importance of an engineering attitude of formalism, professionalism, ethics, and accountability. In fact, there exists a Software Engineering Code of Ethics and Professional Practice (SEEPP) [99] which was developed in a collaboration of IEEE and ACM. However, there is the impression that there is a certain lack of interest in professionalism in the community. An engineering attitude with all the aforementioned properties would indeed not be everybody's cup of tea, considering that the importance of formalism in software development is a contentious issue, and accountability would mean

that software developers would have to take on a lot more responsibility.

In the SWEBOK objectives it is stated that "in other engineering disciplines, the accreditation of university curricula and the licensing and certification of practicing professionals are taken very seriously. These activities are seen as critical to the constant upgrading of professionals and, hence, the improvement of the level of professional practice. Recognizing a core body of knowledge is pivotal to the development and accreditation of university curricula and the licensing and certification of professionals." However, this is one of the main contentious points of the SWEBOK project.

Many people still think that with the current state of affairs, it would be impossible to create a certification system that could truly certify the capability of a software developer to produce high-quality software. This was also expressed in an ACM committee report [167] that led to the withdrawal of the ACM from the SWEBOK project. The report states the opinion of the committee that "our study and analysis has led us to the conclusion that the current software engineering body of knowledge efforts, including SWEBOK, are at best unlikely to achieve a goal of critical importance to ACM: the ability to provide appropriate assurances of software quality for software systems of public interest".

Barrie Thompson [213] discusses the standardization efforts that have been made in the fields of professional ethics and a body of knowledge for software engineering. The author points out certain disparities that hinder such efforts, such as the incompatibilities between professional and academic standards, a bias toward the USA resulting in the lack of an international view, and "turf wars" between the concerned parties. Considering the importance of looking at software engineering from a global perspective, these disparities seem particularly inadequate.

1.2.2 Consistency

According to the IEEE standard 610.12, software engineering is defined as "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software". However, the question is whether the current state of the art can really stand up to this claim. Is software engineering, as it is understood now, truly a systematic and disciplined endeavor? Is it understood as a system that is based on and can be explained with accepted principles? Is it what we would expect from an engineering discipline?

Jeanette Morgan [158] states that a relatively high variance of methodology can be observed in software engineering practice, which she describes as a "labyrinth" of varied practices and procedures with little consensus and many different opinions. In contrast to this, the way in which projects are executed in other engineering disciplines such as civil engineering, is much more standardized. According to Morgan, there are no generally

accepted software engineering practices that are consistently recognized, measured or applied. This is consistent with the view of Brooks, who argues that there cannot be a "silver bullet" for software engineering [30].

Unlike many other engineering disciplines which already exist for a long time and have developed a well-founded base of knowledge over time, software engineering is a relatively new discipline and such a base of knowledge has not fully developed yet. The guide to the Software Engineering Body of Knowledge (SWEBOK) [2] can be understood as such an effort. It tries to describe the overall structure of software engineering methodology in a systematic manner. However, it was and still is a contentious issue.

The ACM committee report [167] that caused the withdrawal of the ACM from the SWEBOK project doubts "whether there exists any process to articulate a core body of knowledge in software engineering that will directly contribute to the solution of the software quality problem." It states that such a body of knowledge could only be created by applying a generally accepted organizing principle, but that "no compelling organizing principle exists at present and we see no clear course of action that would be likely to lead to one in the next few years."

Robert, Abran and Bourque [188] discuss the organization of the software construction knowledge area of the IEEE SWEBOK, and propose a new structuring that is inspired by a general classification system for engineering knowledge. The authors note an absence of quantitative data and, to a degree, also of theoretical tools, remarking that this is quite surprising for an engineering discipline. They interpret this as a sign that software construction has not yet reached the status of a mature engineering knowledge area. This casts doubt on the aforementioned IEEE definition, which claims that software engineering is a quantifiable approach.

Bourque, Dupuis, Abran, Moore, Tripp and Wolff [27] describe a large study that was conducted to explore the principles of software engineering. It included two Delphi studies among software engineering experts, and a web survey among members of the IEEE Technical Council on Software Engineering. During this study suggestions for principles were collected and rated. Just the fact that such an elaborate study is necessary in order to find out what the principles of software engineering are indicates the lack of such established principles. In its very nature the study is based on opinions, suggesting that at this stage principles are a matter of opinion rather than practical evidence.

There is a certain degree of consensus about the suggested principles, but not strongly so. For most suggestion there were arguments for and against. Many principles could not be universally accepted, as there were nearly always caveats. The suggestion that most of the participants agreed was an important principle, "invest in the understanding of the problem", is so general that it is not specific to software engineering at all.

Before discussing software engineering principles, one might even have to step back

and ask oneself whether the metaphor of "engineering" is actually the one that describes software development best. Antony Bryant [31] elaborates this question by showing that metaphors may help a discipline to gain a certain understanding, but may also become an obstacle to further insights. He argues that the engineering metaphor, which dates back to the first NATO conference [160], helped to solve certain problems of the time, but may not be the most appropriate metaphor and should be reconsidered. Alternative metaphors are "writing" of software, which preceded the "building" metaphor of software engineering, "growing" software, and the metaphor of architecture.

Growing of software refers to incremental development and is explicitly mentioned, for example, in [30]. It is based on the fact that factors of software development are often very hard to predict, so that it is impossible to plan a whole project in detail upfront. There cannot be a "grand design" that fulfills all the needs and can be implemented without change. It is the very nature of most software projects that many parts of the plan are constructed on the way and that changes are necessary.

This is reflected in the notion of iterative and incremental development [133], which pays tribute to the fact that most software needs to evolve. This can only happen efficiently through a series of prototypes that can be considered as "smaller", reduced versions of the finished software. The prototypes are "grown" over time until the final product is reached. This notion does not conform with a traditional perception of engineering, which puts much emphasis on detail, precision, and planning before a prototype is constructed. A reason for this can be seen in the material nature and the associated cost to create a prototype in a traditional engineering discipline, whereas in the field of software, the final product is in fact nothing but an executable plan.

The architecture metaphor was notably influenced by Christopher Alexander [5]. This work has inspired the use of patterns in software development, e.g. as they were proposed for object-oriented design in [96]. Another important feature of this metaphor is its relation to humans. Architecture is intrinsically bound to the recognition and consideration of human needs. As is acknowledged through the field of human-computer interaction, the need to consider the human factor applies to most software as well. However, it is a common perception that the metaphor of engineering carries a connotation of disregard for social competence in favor of a very technology-oriented attitude.

All this questions the adequacy of the engineering metaphor for the domain of software development, and such questioning can do the discipline nothing but good. It is a precondition for deeper insights and its further development, and thus has to be faced. However, it also shows that the engineering metaphor is often more an expression of a desire in the software development community than a reality: the desire to come to terms with the matter of software in a way traditional engineering disciplines did with their matter many years ago. We have to admit to ourselves that our domain has not reached

this level of maturity yet.

1.2.3 Rapid Growth and Technological Change

Compared to other disciplines the world of software is a world of rapid growth and technological change. This is caused by a high potential, a high demand and fast technological progress. Software has a high potential because computers can be applied in almost every domain, with many different applications. There is hardly any other technology that influences to that degree the way people live in industrialized countries. The benefits are manifold, but computers also cause many problems, so that the necessity for change is imminent.

Edsger Dijkstra [64] describes the development of computing science, and how things that seem natural to us today have come a long way before they saw the light of day. He remarks that knowledge about computing has spread very much, but that this is not always visible because the number of people involved in the domain of computing has just grown faster. Computers have become a mass phenomenon in the industrialized countries, and even end-users participate more and more in the activity of software development [206].

Unfortunately, much more than any other scientific discipline, software engineering is subject to technological "fashions" that come and vanish. This is partly due to the fact that information technology is of high economic importance, and companies use their market power strategically in order to push and promote their own technologies and standards. New technologies are not necessarily innovative. They can also serve as economic tools in order to bind customers and control a particular market. But surely, there is also a fast cycle of innovation and reinnovation in the world of information technology.

1.2.4 Complexity

It is a common belief that software systems exhibit a particularly high degree of complexity, i.e. that software is more complex than the objects dealt with in other engineering disciplines. Frederick Brooks [30] describes complexity as one of the essential difficulties of software, and remarks that "software entities are more complex for their size than perhaps any other human construct because no two parts are alike; the elements interact with each other in some nonlinear fashion, and the complexity of the whole increases much more than linearly." This is corroborated by data from real projects, e.g. that presented in [121], which shows clearly that projects are much more likely to have problems the larger they are.

First of all, large software systems comprise millions of lines of code, which amount to an immensely complex state space with an extremely high number of degrees of freedom and an astronomical number of possible states. Computer programs are usually much

longer than the formulas used in classical engineering disciplines. Because of the discrete nature of computer programs, careful attention to detail is required in order to get a program completely right. On an algorithmic level, there are no shades between correct and incorrect solutions; a program is either correct or not.

In a discrete space such as the one spanned by computer programs, solutions do not exhibit any proximity: two computer programs that are correct with regard to a problem specification may look completely different, and minute changes of correct programs usually render the solution invalid. This makes computer programs, which are extremely malleable by nature, at the same time extremely fragile. There is a particularly strong necessity to carefully manage and control every change.

In contrast to this, common problems in other engineering disciplines are often continuous in nature. There is usually a continuum of solutions, some better than others. And the proximity of a good solution usually contains other good, or even better, solutions. This makes it possible to apply some very generic numerical optimization methods, such as the gradient method, simulated annealing or genetic algorithms, which would not be feasible for complex software.

John Reel [186] stresses the issue of complexity by claiming that "the basic problem of computing is the mastery of complexity." Many methods of software development control complexity through structure and abstraction: by imposing invariants that help a developer to navigate and comprehend a program, or by raising the level of abstraction on which the program is specified, thus simplifying its representation. Large programs get confusing very easily, particularly when they are not well-structured.

Modularity is one of the most important principles for controlling complexity, not just for software [177] but also for the objects of other engineering disciplines. A modular system is decomposed into modules, which represent clusters of an overall dependency relation. In other words, we want to decompose a system in a way so that the parts within a module have many interdependencies, and the dependencies between the modules are few and well-understood. Modules facilitate system development, as they can be treated separately from other modules and replaced by modules with the same interface.

In traditional engineering disciplines, which deal with material objects, it is much easier to decompose a system into modules than it is for the virtual objects dealt with in software engineering. This is because material objects often provide a natural modularity in physical space. Physical parts that are close together naturally influence each other, and parts that are far away from each other do not. The dependencies in software systems are much more abstract, and thus lack such an inherent organizational criterion. The space in which they live is not a physical one, but has an arbitrary number of sometimes completely unrelated and sometimes strongly related dimensions. There is no inherent proximity rule in that space that delimits the dependencies between a system's constituents.

Often, the complexity of software is compounded by the complexity of the process necessary to develop it. Conducting a software project is not easy and bears many hidden risks. Lederer and Prasad [135] point out that there are many factors that influence the success of a software project, and that it can therefore be difficult to estimate project cost reliably. In order to understand and control these factors, one has to recognize the importance of related disciplines, such as project management.

Successful software development requires just as much good management skills as it requires technical expertise, but unfortunately these two are rarely taught together. Management skills are mainly taught in the Faculty of Commerce, and unfortunately, the relationships between faculties are often not as much of a symbiotic, synergetic nature as one might wish. Students that want to acquire both good technical and management skills have to do the splits between these faculties.

The importance of project management has been documented in various studies. Whittaker [220] shows that most software projects fail because of failure at the management level, naming poor project planning and risk management, a weak business case and lack of top management involvement and support as the most important causes. Also Charette [40] names problems in the management as a very important reason for project failure.

1.2.5 Artificiality

Unlike other engineering disciplines, software engineering is not governed by laws of nature. As the word implies, software is soft, i.e. purely virtual and thus very malleable. It is not subject to many hard constraints like the material objects of other engineering disciplines, which has advantages as well as disadvantages. Physical laws and constraints often serve as a navigation aid, since they are naturally given invariants. In contrast to this, invariants for software are often much less obvious, formalized and understood. Implementing invariants consistently in order to improve the structure of a software system often requires a substantial amount of work.

Software is intrinsically situated on an abstract level that allows much room for arbitrary decisions in a system's design. Many such decisions are a matter of taste rather than discretion, which makes it hard to evaluate solutions and find the "best" one. Differences between solutions are often subtle, and the effects of those differences are not always obvious. Often it is exactly the arbitrary decisions in a design that pose a major impediment to software reuse and integration, making the development process much more difficult.

It becomes clear that the limits of software engineering are determined by the laws of human cognition rather than laws of nature, as it is the case in traditional engineering disciplines. Badly structured programs very easily result in cognitive overload. When developing software we have to perform a balancing act between its malleability and its

manageability. A complex system remains only manageable if we have the discipline to structure it in a way that is compatible with our cognitive capabilities.

Frederick Brooks [30] lists conformity and invisibility as two of the essential difficulties of software. With conformity Brooks refers to the fact that much of the complexity of software is arbitrary, "forced without rhyme or reason by the many human institutions and systems to which [its] interfaces must conform." By using the term "invisibility" he points out the difficulty with which software can be visualized. As he remarks, "software is not inherently embedded in space; thus do not permit the mind to use some of its most powerful conceptual tools. This lack not only impedes the process of design within one mind, it severely hinders communication among minds."

1.2.6 Changing Requirements

Changing requirements is particularly frequent in the domain of software development, and presents a major source of risk to every software project. Reel [186] describes this metaphorically by stating that software developers are trying to "hit a moving target". The 1994 CHAOS report [212] remarks that today's business environment is too volatile to allow enterprise software to be specified in detail upfront. It assumes that in traditional engineering disciplines such as civil engineering, requirements are more stable and can therefore be defined in more detail at the beginning of a project, making the project much more likely to succeed. Brooks [30] names changeability as one of the essential difficulties of software, noting that "the software entity is constantly subject to pressures for change".

One has to recognize that the task of specifying software is particularly difficult because the complexity of the application domain is added to that of software technology. Software engineers may be experts in the workings of software systems, but have essentially to learn the expert knowledge of the application domain, too, in order to truly understand and implement an application correctly. In contrast to this, other engineering disciplines usually do not have such a high variance of applications, and their practitioners are therefore more likely to develop a routine.

1.3 CASE-Tools

CASE stands for Computer Aided Software Engineering and describes tools that support the development and maintenance of software. The term emerged in the 1980's and was in particular associated with the methodology of structured analysis [227]. This older generation of CASE tools reached its peak in the early 1990's, and was mostly replaced by tools supporting object-oriented methodologies. Nowadays, most CASE tools

that support software development on a more abstract level are tailored to the Unified Modeling Language (UML) [171, 173], which has become the most prominent specification language for object-oriented modeling.

In the past many studies have shown that CASE tools have the potential to improve the efficiency and quality outcome of large software projects, e.g. [13], and today software projects that do not rely on tools such as IDEs are practically unheard of. Therefore it is not surprising that a lot of work has been done in this field during the last three decades. Today's software projects make extensive use of CASE technology, to a degree that many projects would not be feasible anymore without it. There exist CASE tools for every aspect of the software development life cycle, and most projects are based on a well-defined chain of CASE tools. CASE technology certainly offers many benefits, but unfortunately there is also evidence that the adoption and use of CASE tools can be problematic.

The next sections describe the different types of CASE tools, i.e. lower CASE, upper CASE and integrated CASE, and how these tools are used together forming a tool chain. Furthermore, the problems of CASE tools are discussed. These problems are important driving forces for the work of this thesis and thus need to be considered carefully.

1.3.1 Lower CASE

Lower CASE denotes the tools that are concerned with the artifacts of later phases of the development life cycle, in particular implementation and testing. Such tools usually work on representations with a lower level of abstraction, such as source code or even machine language. They comprise many of the traditional software tools, e.g. compilers and debuggers, as well as more recent ones such as tools for generative programming [54] or code refactoring [92]. Other tools of that category are ones that support software testing, or ones that are specialized to the management of lower level artifacts, e.g. for configuration management.

Although complex software systems are usually highly abstract in nature, the main representation for software remains source code, which is traditionally a textual notation on a low level of abstraction. Big enterprise applications, for example, are primarily specified on a level of abstraction that is similar to that of business processes, e.g. see [191]. This level of abstraction is mainly concerned with the structure of the domain, i.e. the business that is modeled, and not with the technical properties of the system. In contrast to this, source code as it is commonly used today depends very much on the technology of the system, and is not able to express such a system in terms of the domain. This is why lower CASE is widely used and thus very important.

Historically, a lot of abstraction was already gained by the introduction of *high-level languages*. These languages abstract from the technicalities of the machine language of

specific hardware systems and therefore constitute a major improvement for software development. But with regard to the specification of whole systems, these languages can still be considered as a very low-level representation. Thus, the field of lower CASE is a very technical one and requires primarily technological expertise.

1.3.2 Upper CASE

Upper CASE tools support the earlier phases of the software development life cycle, in particular the requirements analysis and design. Tools in this category offer support on a more abstract, less technological level, e.g. for project management, requirements elicitation, the specification of a design, or the management of higher-level artifacts. Such tools can be very generic, such as all-purpose diagramming or project planning tools, or very domain-specific, e.g. modeling tools for domain-specific models. Such tools may also be tied to a particular software development process, e.g. by supporting certain tasks only in a very particular way.

In recent years the idea of using more abstract notations for the development of software has attracted more and more attention. It is a hope that through the use of software models that are tailored to specific problem domains, a significant gain of efficiency in the development of software can be achieved. The idea is to define the models so that they offer the right structure and suitable abstractions to reflect domain knowledge. This is also referred to as *Domain-Specific Modeling* (DSM). Some models are executable, i.e. can be used in order to automatically create a running system. Others can be refined into an executable model by adding additional information. Examples for such models are [191, 76].

Such models are sometimes defined in the form of *Domain-Specific Languages* (DSLs), which are basically programming languages – although often not Turing-complete. As their name suggests, DSLs contain constructs specifically tailored to their domain, thus offering a more appropriate way to specify a system, or parts of a system, than a General-Purpose programming Language (GPL). A DSL is not necessarily less technology dependent than a general-purpose language since it may be defined for a particular technological domain, e.g. embedded systems. But if the domain is not primarily a technological one, then DSLs can be relatively technology independent. A recent DSL for business processes is, for example, the Business Process Execution Language (BPEL) [210].

In order to make full use of the advantages of software models, upper CASE tools are needed. Just having models is not enough because the handling of models can be cumbersome if not supported by tools. Such tools are used, for example, for creating, modifying and managing models and model data, for translating data between different models, generating additional data, or recovering model data from other representations. Computers are particularly suitable for supporting different perspectives on the same data,

e.g. with different levels of abstraction or different visual representations, thus allowing a developer to choose a perspective that is best suited for a particular task.

The importance of such tool support is well-known and has been discussed in the literature. One of the suggested principles in [27] – a study which tries to explore the principles of software engineering – is "control complexity with multiple perspectives and multiple levels of abstraction". The Rational Unified Process (RUP) – a commercially successful development process framework from IBM – emphasizes the usage of higher-level tools, frameworks, and languages in order to reduce complexity [132].

1.3.3 Integrated CASE

Upper CASE tools may be integrated with lower CASE tools, which is then referred to as integrated CASE. This makes sense, for example, in the context of *forward engineering*, which means that high-level artifacts, e.g. domain-specific models, are transformed into low-level artifacts such as source code. The high-level artifacts are typically produced by upper CASE tools, and the low-level artifacts are handled by lower CASE tools. Often low-level artifacts that were automatically generated from high-level ones still need some manual fine tuning, so that it is still necessary to use lower CASE tools. An integrated CASE tool would be able to handle both the high- and low-level artifacts, and could thus streamline the forward engineering process.

Similarly, *reverse engineering* is the process of recovering high-level information from the implementation of a system. Again, this process is concerned with both the high-level artifacts of upper CASE and the low-level artifacts of lower CASE, and can be streamlined with an integrated CASE tool. That way, a developer could, for example, investigate the parts of an implementation manually with lower CASE functions before selectively recovering high-level information.

Sometimes existing systems are changed or overhauled by first reverse engineering them in order to get a high-level specification, and then using that specification in order to forward engineer the reworked system. This is known as *reengineering*. It is another example where integrated CASE technology can be very helpful because both the reverse engineering step and the forward engineering step are likely to require a developer to actively work with both low- and high-level artifacts. In fact, the more complex the interaction between low- and high-level artifacts, the more can integrated CASE tools benefit the development process.

Round-trip engineering means that a developer works with two different types of artifacts which are transformed into one another, but these types of artifacts are not necessarily on different levels of abstraction. They could be just two different representations. Sometimes it is convenient for a developer to choose one representation instead of the other when performing a particular task. Round-trip engineering would mean that each

change done to one representation would automatically result in a corresponding change to the other, so that both always reflect the same data. Examples for this are GUI builders, which usually offer graphical and source code representations of a GUI, or class modeling tools, which are commonly tightly coupled with a source code representation.

1.3.4 The Tool Chain

As already anticipated in the previous sections, a whole range of CASE functionality is necessary to accommodate the needs of a big software project. In general, no single tool is able to provide all the desired functionality, so it is necessary to use a whole set of tools and use them in combination. Many of the common CASE tools are stand-alone applications which focus on supporting a very particular aspect of the software development process. Usually, a project incorporates many different tools with different purposes, forming a *tool chain* that stretches along the development life cycle. Such a tool chain can be complex, and it can only function properly as a whole if the tools can be sufficiently integrated.

However, it can be very cumbersome to integrate all the different tools involved in the development of large systems. Many tools use proprietary data formats for input and output, which makes it very hard to connect the tools and establish the data flow between them. It might be necessary to convert data before it can be used by a tool. Related tools should be able to share their data, so that previous work can be reused. Furthermore, the whole interaction between the tools has to be orchestrated.

Let us consider the example tool chain in Figs. 1.1 and 1.2. This tool chain illustrates the interaction of different tools for web application development. The double rectangles represent tools that are primarily designed to interact with a user, usually through a GUI. The 3D boxes represent tools that can run automatically, i.e. need no user interaction once they are executing. The document shapes represent the artifacts that serve as input and output of the tools. The arrows between artifacts and tools represent data flow. The tools in the upper part of Fig. 1.1 fall more into the upper-CASE category; the tools middle and lower part fall more into the lower-CASE category.

In the analysis and early design phase, developers can be supported by tools for user interface and data modeling. In the context of web applications a user interface model can, for example, be a navigational model that specifies possible sequences of web pages. Such a model could be visualized and edited graphically, e.g. as formcharts in the form-oriented analysis model [76]. The resulting artifact would be a user interface specification.

Data modeling is one of the most important activities in a software project because it lays the foundation for the business logic that is provided by an application. In the data model, essential properties of the application domain are reflected, therefore a data model can aid in the understanding of the semantics of that domain. Because of their importance, data models have inspired a lot of research, and there exist many different

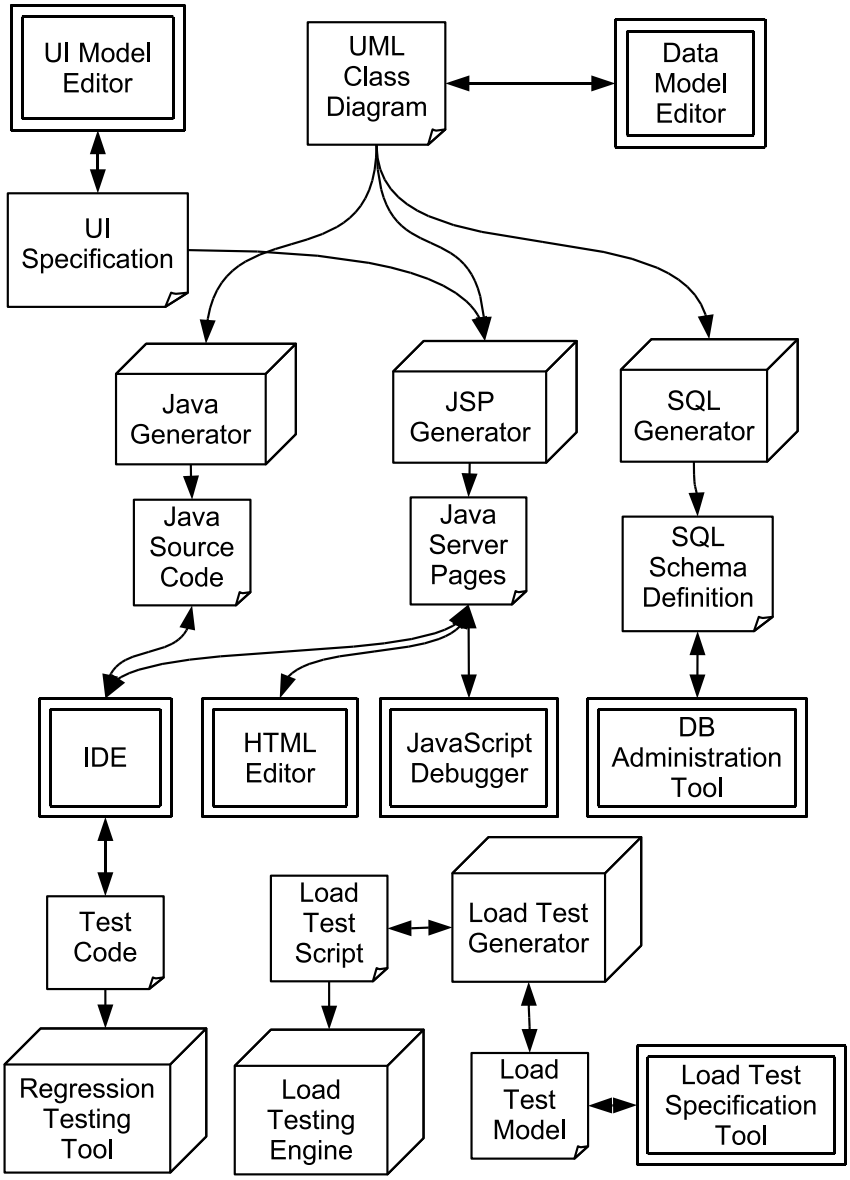

Figure 1.1: Example tool chain, part 1.

data specification languages. However, the basic concepts of those languages are usually very similar. The most popular modeling language these days is UML [171, 173], and most contemporary data modeling tools use UML class diagrams. There also exists a standard for the serialization of UML diagrams into XML [170].

If a project relies on several technologies, such as the one in the example tool chain, parts of the data model will most likely have to be represented in different languages. It would be cumbersome and error-prone to translate manually between different data modeling languages, therefore generators are ideally used in order to automate this task. In the example, there is a generator that generates Java class definitions and possibly database access code from the data model, and another one that generates a relational database scheme in SQL [116].

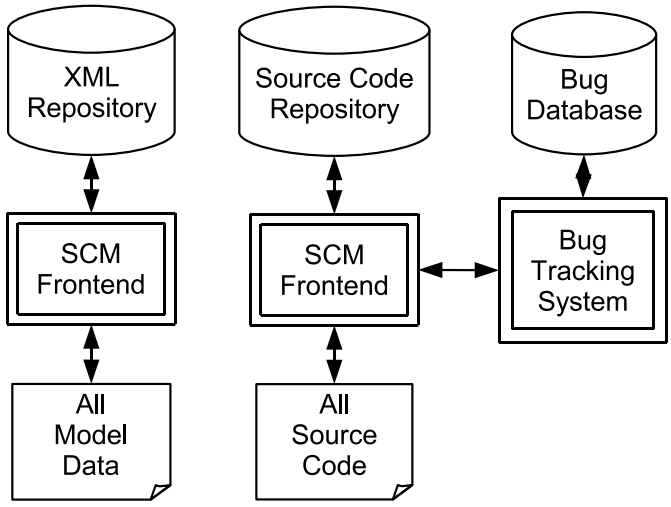

Figure 1.2: Example tool chain, part 2: tools for SCM.

The data model and the user interface model can potentially be used in order to generate a first approximation of the user interface implementation. Although this first prototype will most likely be extended and adjusted during the implementation phase, it provides a structured skeleton of important user interface artifacts, and thus can save a lot of work. An example of such a generator technology is Angie [77]. The data model is important for the user interface because many of its data types are reflected there [72]. Many web pages typically show typed data to the user or let the user enter typed data into the system.

In the example, the prototype generated from the user interface and the data models is implemented with Java Server Pages (JSPs) [178]. JSPs are a mixture of different technologies, and therefore the tool support required for them can include quite a number of different applications. Basically, a JSP is a HTML page with embedded Java code, so developers might want to edit them with both a HTML editor and a Java IDE. The HTML code determines the visual appearance of a page, therefore the HTML parts might be edited by a web designer, using the corresponding graphical tools. The Java code encapsulates the business logic and describes the way HTML is generated from a JSP.

In addition to that, web pages are themselves a hybrid technology because other different technologies can be embedded into them. For example, many web pages incorporate JavaScript [90] code, which is a scripting language that can be executed by modern web browsers. JavaScript is used, for example, in order to validate user input or enhance the user interface on the client side. Web developers also need tools to support such non-trivial embedded technologies, e.g. a JavaScript debugger to help them find errors in their JavaScript code.

Automatically generated code usually cannot be used without further modification. The generator output has to be fine-tuned to the individual needs of the project in the

implementation phase. Naturally, for modifying Java code, an IDE with support for Java is used, e.g. the Eclipse IDE [82]. Likewise, SQL code is usually handled with a DB administration tool, which helps to load, examine, modify, and test it with a database. Powerful database administration tools also offer support for debugging stored procedures and database performance tuning, which are very important activities of large software projects.

Software testing is so important that most process models consider it very explicitly, e.g. in the form of a designated testing phase. It is an accepted best practice to create a test suite for the software under development, which encompasses test cases for all important functions. In order to automate testing, there are many tools for regression testing, i.e. testing done after modifications have been applied to a program. Automatic regression testing is often integrated into the build process in order to make sure that modifications do not break the build, i.e. introduce new defects.

Particularly for enterprise web applications, it is important to make sure that the performance of the deployed system is high enough to deal with the expected load. This is called load testing. In order to obtain meaningful load test results, user input to the system has to be simulated in a realistic way, i.e. by applying a suitable load test model. Realistic laod test models have been described in [66]. Development tasks like load testing can potentially necessitate their own little tool chain: the example shows a tool for editing a load test model, a tool for generating a load test script from a load test model, and a load test engine that performs the actual tests.

Figure 1.2 shows a part of the tool chain which is needed for the whole development life cycle: tools for software configuration management (SCM). SCM is a support discipline concerned with the control of the evolution of complex software systems [86]. This includes, among other things, the control of all the artifacts and the control of changes done to those artifacts. Artifacts and changes have to be well-organized in order to deal with the complexity of large systems.

The example illustrates a bit of the complexity that can be introduced by SCM itself: earlier phases of a project might primarily deal with artifacts that can be represented as XML, such as UML models or office documents. Therefore, those artifacts might be managed by a tool that specializes in storage, retrieval and modification of XML, such as an XML database management system. Developers may not use such an XML repository directly, but through an SCM frontend that provides an appropriate user interface and may be tailored to the development process of the project.

In addition to this, there might be a separate repository for non-XML artifacts such as common source code. Again, access to the repository is very likely not done directly but through an appropriate SCM tool. Source code data is likely to be linked with other data such as data about defects. However, many bug tracking systems are separate from the

version control systems that are used to manage source code repositories. Bug tracking systems often maintain their own database, and need to be integrated with the source code repository in order to use both efficiently, e.g. through an additional SCM frontend.

1.3.5 Problems

As one might expect after considering the example tool chain in the last section, CASE technology does have its problems and challenges. Some of these challenges result from the heterogeneity of the tool chains frequently found in software projects, and the heterogeneity of their artifacts, as pointed out, for example, in [141, 155]. Several studies show that adoption of advanced CASE tools in industry has been slow, and the desired positive effects have not always been achieved [4].

One of the problems is the interoperability of tools. Since CASE tools ultimately have to function together in the tool chain, it is not sufficient to just select the best CASE tool of each functional category and use them in a project. Before two tools can be used together, project managers have to ask themselves if they are actually interoperable in a tool chain. Sometimes interoperability can be achieved by adding "glue" to the tool chain, e.g. gateways between different subsystems. However, this causes additional cost and possibly maintenance issues for the tool chain itself.

It is quite common that tools of the same category, e.g. UML modeling tools, provide principally but not entirely the same functionality. This can make the choice of a tool more difficult. On the one hand, there might not be a single tool with all the desired functions, so that one might consider using several tools in combination. On the other hand, having several similar tools results in redundancy, additional complexity and cost. Unfortunately, there often is a discrepancy between the functionality developers want and the one implemented in CASE tools [146]. Because of this, and the differing requirements of different software developing organizations, the ability to customize and extend CASE tools is particularly important.

Another challenge for CASE is accessibility of information. On the one hand, this involves the organization of all artifacts. Unfortunately, many tools have their own way of organizing their artifacts, so that this might not be trivial. Somehow, the different data management mechanisms of different tools have to be integrated in a way that makes it easy to find an artifact when it is needed. Some tools might, for example, use their own database for storing their data, while other tools might just use particular folders in the file system.

On the other hand, a developer might not just look for an artifact but for a particular piece of information contained in one or more artifacts. However, if different tools are used, the artifacts will most likely be heterogeneous in nature. Many tools use their own data formats, some of which are proprietary. Even if a format is open, it might still not

be easy to use. If a developer needs particular information, but this information is not readily accessible at the level of a tool's user interface, then he or she might need to spend a lot of time digging it out of a system's internal structures.

In addition to that, the information developers and project managers want is not always directly contained in the artifacts: sometimes aggregated information is needed that is calculated from potentially many differently structured artifacts. Typical aggregate information would be, for example, software metrics such as the number of lines of code in the project, the degree of dependency between modules, or other measures such as the number of man-hours spent on a particular task. It could also simply be a set of extracts from all documentation artifacts concerned with a particular part of the system. What ever it is, it might not be easy to aggregate information from potentially very heterogeneous artifacts.

However, such information can be very important for project management. It can be used for supporting decisions with quantitative data rather than just qualitative perceptions, and can thus help to steer a project in the right direction. Tools targeting such activities are known as decision support systems. With modern SCM tools such as version control and bug tracking systems useful data does accumulate, but without appropriate tool support it cannot be fully used. The importance of aggregated project data, such as measurements, is recognized in many process models. The higher maturity levels of the CMM [200], for example, put a lot of emphasis on quantitative analysis.

One of the main obstacles in CASE adoption is the complexity of CASE systems. This has been shown by several studies, e.g. [113, 126], and is known to be a major cost factor when CASE technology is introduced in an industrial environment [111]. Part of this complexity is created by the size and heterogeneity of the tool chain, which has to function as a whole. In addition to the complexity of the tool chain, there is the complexity of the CASE tools themselves.

Most CASE tools are complex software projects in their own right. In order to illustrate this, I have compiled information about the lines of source code plus markup language code in open-source CASE tools, which is shown in Fig. 1.3. Many tools in the list are immensely popular and widely used. The information was collected in February 2007 from [174]. Only open-source CASE tools were considered because such data is naturally available for them, in contrast to most commercial, closed-source tools. However, the sheer size of popular commercial tools such as the ones developed by Microsoft, Borland and IBM (usually ¿100 MB) suggest that the argument holds for them as well.

Of course, the lines of code metric does not correctly reflect the complexity of software. However, a big number of source code lines – excluding pathological cases – is indisputably a clear indicator. The figure points out that common open-source CASE tools already reach a respectable degree of internal complexity. Tools that offer a relatively limited

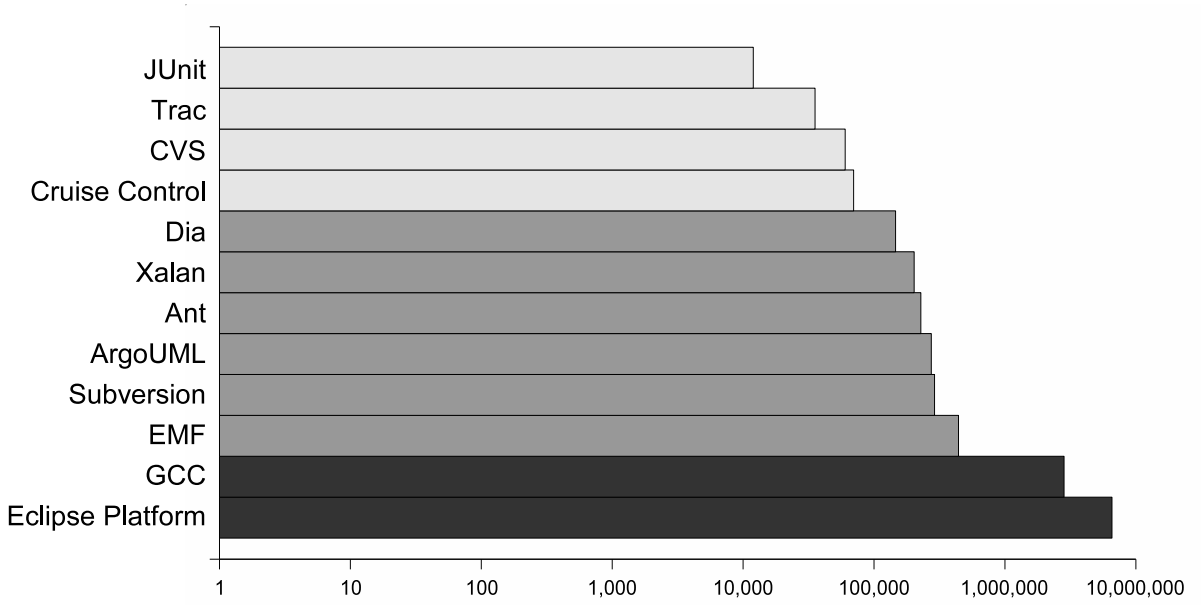

Figure 1.3: Lines of code (LOC) including markup code of open-source CASE tools (February 2007).

functionality already have LOC in the order of 10^4. Most of the common tools reach a LOC in the order of 10^5. There are some projects, e.g. the GNU Compiler Collection and the bare Eclipse IDE [82], which are huge, reaching LOC in the order of 10^6.

It is therefore not surprising that CASE tools often have a steep learning curve, due to their technological complexity. Learning how to use a new tool efficiently can take considerable time. Studies have shown that the costs of training can double the costs initially spent on CASE technology. The steep learning curve can even lead to an initial loss of efficiency after CASE adoption [126]. CASE tools were frequently abandoned before a gain in efficiency could manifest. The steep learning curve and the need to integrate the new technology with existing resources present significant hurdles companies trying to adopt advanced CASE technology have to overcome [111].

Other research points out that aspects of usability are very important factors for the actual CASE usage [138]. This is particularly evident with regard to CASE tool complexity and their steep learning curve. The user interfaces of such tools have to be designed to hide the complexity of a tool, yet make all its features accessible. A user interface must be easy to use, yet efficient to use. While novice users need a user interface that is intuitive in order to deal with the steep learning curve, professional users need fast access to the functionality in order to perform well.

Intuitiveness in the short run and efficiency in the long run are often conflicting demands: expressive, fast methods of interaction often tend to be too sophisticated for casual users. An example for this is the contrast between GUI-based tools and command-line tools: GUIs are often easier to use for beginners, while many professional users can type commands much faster than they could point and click in a GUI. Another example

is the use of pull-down menus versus the use of shortcuts. Good CASE tools have to find a balance between the demands of different groups of users.

It is not unusual that advanced CASE tools prescribe a certain software development process, which has to be adopted in order to benefit from its use [4]. This means that the decision to adopt a particular CASE technology can have a strong impact on the whole software development process. If a software developing organization is not ready to change the way it produces software, then attempts to adopt such CASE technology are doomed to fail.

This problem can be mitigated if CASE technology offers a high degree of customizability and support for integration, so that tools can be adapted to fit the desired development process, and not just the other way around. The ability to integrate with other tools is a particularly important requirement [123]. Unfortunately, such support adds to the complexity of a system, so that such a requirement is antagonistic to ease of use. Integration and customization of CASE technology to the needs of a company can be another significant cost factor because it frequently requires the help of external consultants [111].

CASE systems are enterprise applications and therefore face the same challenges as other enterprise applications, one of them being integration. Over the years, the problem of integration of different CASE tools has been addressed by many standards. But most of them – even big, government supported initiatives like the Portable Common Tool Environment (PCTE) [7] – have failed to gain widespread acceptance. This shows that integration of CASE technology is a complex problem that cannot be solved just by standardization. Despite the huge amount of work that has already been done, it is an area of ongoing research, and in need of new ideas and solutions.

1.4 A Platform for Model-Based Software Engineering

When talking about *model-based software engineering*, I mean that software engineering is done on a level of abstraction that is adequate for specifying the functionality of a system without taking into account implementation details. I concentrate on information relating to the requirements of a system, and if necessary, also to its design. For example, we might have to specify what data should be stored in a system, but not how the system utilizes hardware resources in order to do so.

The following sections introduce the AP1 system, which is the subject of this thesis. AP1 is a platform for model-based software engineering, and is therefore a CASE technology. Its main purpose is to integrate the different CASE tools of a tool chain. First, I describe the principles that were used to guide AP1's design. Then, the overall

architecture of the system is delineated. Finally, an overview of the thesis is given.

1.4.1 Design Principles

AP1 is motivated by some important requirements of CASE. I tried to address many of the problems described in Sect. 1.3.5. This led me to pay particular attention to the following set of design principles:

Simplicity The concepts used in AP1 should always be as simple as possible. Good simple concepts are powerful building blocks for more complex ones. If a more complex feature can essentially be reduced to simpler features, e.g. in the form of a usage pattern, this should be done. This makes good economic sense with regard to development and maintenance. Good concepts are independent of particular technologies, which are usually much too specific. Simplicity is often more important than performance because it is more substantial. Performance grows steadily with Moore's law, or can be achieved by means of optimization, but the use of overly sophisticated concepts can damage a system forever. Hence, simplicity must not be tempted to unnecessary compromise with technological circumstances, for the latter ones change steadily.

Maturity The concepts used in the system should be mature because conceptual faults are much harder to correct later on. Often, workable concepts can be expressed formally, or are in fact mathematical concepts themselves. Concepts that are interdisciplinary (e.g. mathematical or derived from common sense) are generally more mature. Novelty and popularity are only indirect indicators of quality and are to be considered with due care. Maturity of a technology commonly implies that it is readily available for a wide range of technological platforms, e.g. architectures and operating systems.

Reuse AP1 should reuse suitable concepts and technologies as much as possible. It is of no value to solve the same problems over and over again. New features should preferably make use of existing ones, and/or form a basis for other new features. This is related to the simplicity principle, as simple features are generally more likely to be reusable in more complex schemes than features that are already complex by themselves. Modular reuse of technological components makes it possible to profit from their improvement without much additional cost. Standardized components are much more future-proof than bespoke ones because a large user base holds a stake in it, and thus drives its stability and improvement.

Integration As discussed before, CASE tool integration is an important requirement, and lack of it causes problems. Integration of tools potentially results in added value

such as synergies and new functionality. Lack of integration potentially causes loss of data and functionality, and friction at the user interface. Integration is a multi-faceted feature and should happen on different levels: data integration, control integration of functionality, presentation integration of user interfaces, and process integration through seamless support for different activities along the software development life cycle.

Unobtrusiveness The methodological support provided by AP1 should not put constraints on the development process an organization uses. Concepts and features should be descriptive rather than prescriptive in nature, meaning that a team is free to use them but should not be forced to do so. CASE support should not excessively favor a particular development or programming paradigm, but offer flexibility for freedom of choice.

Customizability Every software development organization is different and therefore may want to use the features of a system like AP1 in a different way. To support these differences AP1 must offer a high degree of customizability. One way of doing so is the pattern approach, which means that specialized features are expressed by means of generic building blocks. Another way is to open up the system to the user, e.g. through reflection.

Extensibility AP1 is intended as an enabling technology. It is a platform for CASE tools rather than a complete CASE environment. As a result, it is of utmost importance that the system can be easily extended with CASE tools, i.e. that CASE tools can make use of the platform functionality easily. This is achieved, for example, by means of an open architecture that allows plug-ins and the use of well-accepted standards.

Usability Functionality becomes useless if it cannot be accessed by a user, e.g. because access is too complicated and requires a lot of learning. Usability is a major requirement in the face of the high complexity of CASE environments and software engineering in general. As a result, a system like AP1 should not be thought of on a purely technological level, but include considerations about human-computer interaction (HCI). Usability is particularly important in the face of structural dilemmas. Such dilemmas are, for example, antagonistic effects between the expressiveness and richness of a feature set on the one hand, and simplicity and manageability on the other hand.

These principles are neither orthogonal to each other nor completely conflict-free. For example, there is usually a trade-off between simplicity on the one hand and integration, customizability and extensibility on the other hand. Furthermore, the reuse of mature

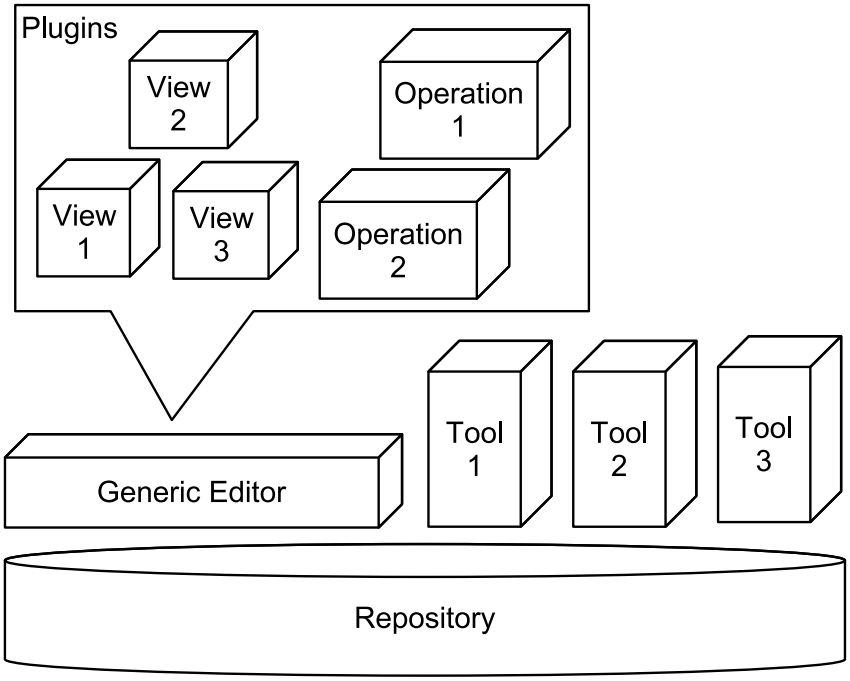

Figure 1.4: Architectural overview of the AP1 system.

concepts and technologies may impede innovation. It is therefore important to realize that many of the principles express ideals that may not be achievable in reality. However, they serve as a reasonable guide for all the design decisions made in this project and should hence be judged by their practical value.

1.4.2 Architectural Overview

In this thesis I propose an architecture for CASE-tool integration, which I call an abstract platform. It is called abstract because it abstracts from a particular tool chain configuration and particular tools, implementing functionality that most tools have in common. Rather than reimplementing it, tools can build on this functionality, which is why I use the term platform. On the one hand this makes it easier to create new CASE-tools, since the functionality of the platform can be reused, and on the other hand it inherently enables tool integration because tools share a common base. Figure 1.4 gives an overview of the architecture of the AP1 system.

The architecture of AP1 can be considered as consisting of different layers: a data management layer at the bottom, a presentation layer in the middle, and an application layer at the top. The components described below are related to the layers in the following way: the repository forms the data management layer, the generic editor provides functionality for the presentation layer, and the generative programming technology that I developed is an example for a concrete tool situated in the application layer.

The *repository* is the heart of the abstract platform. It is a common persistent stor-

age for all artifacts that are used in the software development process. The repository contains a metamodel, models and model instances, i.e. it stores all the data used during the software development process. Besides the artifacts which are directly relevant for software development, it also stores information about changes in the data, access control information and AP1's configuration.

All data in the repository is typed and extensible, so it is possible to define new models for artifacts. Besides very specialized models for particular applications, the repository also contains more general models that aim to capture all the important characteristics of a certain domain. Such models exist, for example, for source code and user interface layout.

The repository manages and protects the artifacts of a software project. CASE tools can be implemented on top of the repository, taking advantage of its data management capabilities. This is reasonable because all CASE tools, no matter how specialized their functionality, need to manage their input and their output. The repository offers a notification mechanism that informs CASE tools about data changes, so that tools can communicate with each other.

CASE tools can access the repository in a transactional manner. Non-functional features such as concurrency control, access control, and versioning are part of the repository, and thus need not be dealt with by the individual tool. CASE tools can access the repository either directly through a standard database interface, or through an object-oriented API. The repository supports *data and control integration.*

Integration of data has different aspects: a syntactic and a semantic one. *Syntactic integration* means that we end up using the same metamodel for models, so that the structure of the models is defined in the same terms. This means that we can access the models in the same manner. However, the structure of the models can be different.

Semantic integration means the we understand not only the structure of the models but also their meaning, and that this understanding is manifested in the system. Semantic integration can be expressed, for example, in mapping information relating the models to each other or to a common "supermodel", or operations that can sensibly make use of more than one of the models. Syntactic integration is usually much easier to achieve.

Since tools share the same data model, there should be a generic way to edit models and model data. On top of the repository, AP1 provides a *generic editor*, which is an application for visualization, editing, analysis and processing of artifacts. The generic editor can be looked at as a simple generic CASE tool. It offers essential functions that are usually duplicated in different tools.

For different tasks and data some data representations prove much more efficient than others. This is not only true for internal representations but also for the way data is represented in the user interface. Consequently, it is important to let users configure the

way they want to perceive and deal with data. Such representations of model data in the user interface are called *views*. Editable views do not only visualize data but also support its modification. The abstract platform supports multiple concurrent editable views on the data in the repository. Through the repository's notification mechanism, those views can be synchronized.

The generic editor provides generic views and basic operations for visualization and modification of models and model instances. The default view presents data in a tree-like fashion. Basic operations include data modification and reflection. The generic editor has an open architecture, which can be extended by plug-ins and configured through the repository. Plug-ins can implement new views and new operations on data. This makes it possible to efficiently implement CASE functionality as plug-ins with a high degree of reuse. The generic editor supports *presentation integration*.

As I have already mentioned, a typical tool chain in the software development process not only provides a user with ways to modify data, but also supports the automated processing of artifacts. Besides data management, modification and visualization, AP1 supports data transformation by providing its own model of generators. The model is stored in the repository and the operations necessary for generation are plug-ins for the generic editor.

The field that deals with automatic generation and enrichment of source code is known as generative programming [54], and its importance for real-world software development is growing. For data other than program source code, automatic processing is also common: numerical analysis of data sets, transformation of graphical models, typesetting of documents, encryption and data compression are typical examples. In this thesis I explain AP1's concepts for code generation. Non-programming related generators are also supported by AP1, but not part of the thesis.

Transformation and generation of artifacts is a very common function of CASE technology, and plays an important role in the integration of different models. Integration can be achieved by generating an instance of one model from an instance of another, with the domain knowledge about how the two models are related being programmed into the generator. Consequently, the generator model supports semantic data integration. Having means for syntactic and semantic data and control integration, AP1 provides means to model software development processes, e.g. along the lines of the procedures described in [26], paving the way for *process integration*.

AP1's architecture is similar to the integration framework architecture described in [224]. The repository of AP1 corresponds to the object management system in the integration framework, and the repository API to the integration agent. The generic editor corresponds to the common user interface component. The difference is that tools in the integration framework interact with the object management system through the integra-

tion agent, and with the user through the common user interface; i.e. all tools are framed by these two layers. AP1 allows tools to access the repository directly, and also allows them to have their own user interface. Its architecture is that of a layered platform where tools can be based on lower or higher layers, resulting in a lower or higher degree of integration.

It has already been shown, e.g. in [216, 228], that it is possible and highly reasonable to have meta tools for tool development. The abstract platform can be looked upon as such a meta tool, but one that is explicitly open to extensions and intended to integrate the various activities involved in the software engineering process. Its main paradigm is that of model-based software development, as it is also intended, for example, in Model-Driven Architecture (MDA) [156] and other approaches for executable specification. Its aim is to support the creation, analysis, transformation and recovery of models, thereby facilitating program creation, program analysis, forward and reverse engineering.

1.4.3 Structure of this Thesis

Because of the complexity of the software engineering process and the holistic nature of the abstract platform approach, my research is concerned with more than one field of computer science and software engineering. On the one hand this meant more work, but on the other hand it meant that the research covered a much wider spectrum and was therefore a much more complete learning experience. Such a wide spectrum seems important when considering that many of the concepts of different knowledge areas are related.

My approach is generally a theoretical as well as a constructive one: I identify and define concepts for the different aspects of model-based software engineering that exhibit certain advantageous characteristics. In order to provide a proof of concept, a prototype of AP1 was implemented. The thesis is part of the research programme described in [76]. The project answers questions that appear in the context of the integrated source code paradigm. I do not claim that all the concepts presented here are superior to existing ones or that AP1 can replace contemporary industrial systems such as Visual Studio or Eclipse. Instead, I am interested in the feasibility of these concepts under the given constraints. Most of the concepts as they are described here have not been subject to scientific examination before and constitute important feasible options that must be considered. Their identification and discussion leads to better understanding and opens the way for further development of the field.

Chapter 2 is concerned with the choice of a suitable data model for the abstract platform. It discusses the pros and cons of different popular and not so popular data models, and substantiates the choices that I made. Besides considering their characteristics and illustrating them with some examples, the chapter also gives a brief overview of the history

of the different data models.

Chapter 3 describes the problems and solutions that emerged in the implementation of the repository. It deals with the mapping between the parsimonious and the relational data model, the way more advanced features such as inheritance were implemented, and the development of a client library for enhanced repository access. The client library provides an object-oriented API for the repository, and offers additional functionality such as a read cache for better performance and a change notification service.

Chapter 4 describes a mechanism for fine-grained change control that was built into the AP1 system. It shows how the repository incorporates functionality for software configuration management, i.e. management of changes on artifacts and their different versions. It explains why this approach is particularly extensible, and can be used in different environmental configurations.

Chapter 5 discusses concepts for making user interfaces for content creation more robust. Consequently, it is concerned with issues related to HCI. Robust user interfaces ensure that the integrity of a system cannot be violated by users, i.e. they are able to prevent errors. This is particularly important for the AP1 system since it is intended to handle large amounts of valuable data. The generic editor applies most of the proposed concepts.

Chapter 6 describes how the principle of reflection can be transferred from the domain of programming languages to the domain of user interfaces in order to improve their usability. This chapter represents another study in HCI which was conducted in order to lay a foundation of AP1's user interface. AP1's generic editor relies heavily on the reflection principle.

Chapter 7 is about the generic editor. It shows how synchronized views are implemented, and how basic data manipulation works. It also shows how the generic editor can be configured and extended through the repository. Furthermore, the chapter points out AP1's capabilities for collaborative work, and how they derive naturally from its design.

Chapter 8 is about Genoupe – a novel technology for generative programming, which was developed in the context of this project. Genoupe integrates the concept of generic types with capabilities for reflection in a type-safe manner. It introduces the notion of generator type safety, which is particularly strong, and describes a formal type system that implements this notion for Genoupe. The Genoupe technology can be used on top of the AP1 system. It can be applied for different programming languages as well as more abstract model-based representations of source code.

Chapter 9 concludes the thesis, points out future directions and adds some final reflections and remarks.

2

Choosing the Right Data Model

A *data model* specifies a particular way of structuring data and operations through which the data can be used. This chapter describes differences between several data models and database technologies. For my project I had to chose one of them for the repository subsystem, which is depicted at the bottom of the architectural overview in Fig. 1.4 and outlined in Sect. 1.4.2. In order to make a well-founded decision, I examined their functional as well as non-functional characteristics. Many data models, even though widely used, are bound to a particular set of requirements and limitations. Therefore, popularity of a particular data model could not be the main criteria for such a choice.

Note that the choice of a data model can be independent from the choice of a data *representation*. The same data can exist in different representations, while exhibiting the same structural properties. For example, a concrete data model could be represented as text, as in a programming language, as graphics, as in diagrams, or in other less human readable forms, e.g. binary code. Dealing with model-based CASE data does not enforce a particular representation, but some kinds of representations are more suitable for certain purposes than others.

Section 2.1 discusses the relational data model. Section 2.2 looks at common properties of object-oriented data modeling. Section 2.3 describes the parsimonious data model, which is less known. Section 2.4 and 2.5 discuss the popular UML and XML data models, respectively. Section 2.6 concludes the chapter. Parts of this chapter were published in [143].

2.1 The Relational Data Model (RDM)

The RDM was first proposed in 1969 by Edgar Frank Codd [48]. It is a formal model that uses set theoretical concepts in order to describe the structure of data. It has since been very successful in the database world and has replaced many older data models used in database systems, such as the hierarchical and the network models. It soon became the predominant model used in industrial-strength database management systems, and remains so until today.

The basic notion is very simple: all data is stored in typed, mathematical relations. That is, data is stored in *tuples* $(a_1, \ldots, a_n)$ which are elements of a Cartesian product $A_1 \times \ldots \times A_n$. In addition to a type A, each of a tuple's components has a name a, and together they are called an *attribute*. The type of a relation is defined by its attributes. A *relation* itself is defined by its tuples, which contain values for the attributes, and also has a name.

A relation can define a *primary key*, which is a set of its attributes that is used to identify its tuples. For this to work, all tuples must have unique primary key values. Some relations use *natural primary keys*, which means that part of the data that is naturally stored in a tuple is used as primary key. Examples for a natural primary key are the passport number for a relation containing data about persons, or registration numbers for cars. Relations can also use *artificial primary keys*, i.e. artificially generated values that identify each tuple, such as running integer numbers or GUIDs. A third possibility is that of mixed keys which include artificially generated as well as natural information.

In order to define connections between different tuples, *foreign keys* are used. A foreign key is a set of attributes of a relation that is used as a reference to a primary key. For each attribute of the primary key, the foreign key contains a corresponding one. This way, tuples can refer to other tuples of a different or the same relation by citing their primary key values. All the possible kinds of associations between data elements can be modeled in that manner.

A *relational schema* is a set of relation types. It forms the basis of a relational database. In general, all foreign keys used in a schema refer to primary keys of relation types that are also part of that schema. That is, a schema usually forms a self-contained data model for a particular problem or domain.

2.1.1 Example

As an illustration and for further discussion I would like to consider the following relational schema S, which models records about persons and bank accounts:

$$S = \{Person(\underline{id}, name, address), BankAccount(\underline{number}, ownerid, balance)\}.$$

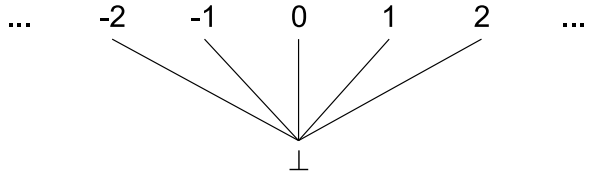

Figure 2.1: The flat domain of integer numbers.

The underlined attributes form the primary keys of the respective relation. While relation *Person* has an artificial primary key, i.e. one that does not carry any meaning outside the database, relation *BankAccount* has the account number as a natural primary key. Attribute *ownerid* of *BankAccount* forms a foreign key that refers to the *Person* tuple of the owner of an account. The types of the relations can be given as the Cartesian products that form the supersets of the relations:

$$\begin{aligned} Person &\subset Integer \times String \times String \\ BankAccount &\subset String \times Integer \times Double \end{aligned}$$

For this example I assume that the attributes have simple primitive data types, such as *Integer*, *Double* and *String*. Relational primitive types are different from the primitive types in most programming languages in that they are flat domains. A flat domain is a very simple complete partial order (CPO) with incomparable elements (i.e. incomparable according to the approximation CPO) and a bottom element $\perp$ that approximates all other elements [1], as illustrated in Fig. 2.1. This simply means that in addition to the primitive values, such types also have an element $\perp$ that represents the state in which the value of an attribute is unknown, e.g. undefined.

2.1.2 Associations and Multiplicities

One of the advantages of the relational data model is that associations, i.e. links between tuples that are defined via foreign key references, can be navigated in both directions. This is because the usual way of formulating queries, relational algebra (e.g. see [84]), establishes connections between foreign and primary keys by joining corresponding tuples together into a new relation. The join operation does not take into account which set of attributes is a primary key and which one is a foreign key. Instead, two relations are combined with a Cartesian product, and then the resulting tuples are filtered with a boolean predicate so that only those tuples remain where the values of primary and foreign key match.

The RDM has the disadvantage that certain multiplicities of associations between data

elements are hard-coded in a relational schema, and cannot be changed without changing the schema's structure. That is, the topology of a data model is not orthogonal to the concept of multiplicities. This creates a dependency between concerns that should ideally be separated, and results in maintenance problems during the evolution of a database.

For example, consider the association between persons and bank accounts. In the example schema, a bank account is owned by at most one person, i.e. exactly one person if we demand that *ownerid* must not be $\perp$. However, a person can own arbitrarily many bank accounts since many *BankAccount* tuples can have the same *ownerid* value. If we wanted to change the multiplicities so that each person can have at most one bank account but a bank account can have arbitrarily many owners, we would have to put a foreign key referencing a *BankAccount* tuple into relation *Person*. We would also remove the foreign key *ownerid* in *BankAccount* that references a *Person*. This is shown in the schema S', where *Person* now has the foreign key *bankaccount*:

$$S' = \{Person(\underline{id}, name, address, bankaccount), BankAccount(\underline{number}, balance)\}.$$

If we wanted to change the multiplicities of the association between persons and bank accounts so that a person can have arbitrarily many bank accounts and a bank account arbitrarily many owners – a many-to-many association, then the change would be even more drastic: there would be no foreign keys in the relations *Person* and *BankAccount*, but we would have to create a new relation that associates the tuples of the two relations. This is shown in schema S'':

$$\begin{aligned} S'' \;=\; & \{Person(\underline{id}, name, address), BankAccount(\underline{number}, balance), \\ & Ownership(\underline{personid}, \underline{accountnumber})\}. \end{aligned}$$

Relation *Ownership* contains a foreign key *personid* that references a tuple of *Person*, and a foreign key *accountnumber* that references a tuple of *BankAccount*. It is thus possible to join persons with their bank accounts by first joining *Person* with *Ownership*, and then joining the result with *BankAccount*. The primary key of relation *Ownership* comprises both attributes *personid* and *accountnumber*, so that each connection between a person and a bank account can be stored once.

2.1.3 Relational Database Management Systems (RDBMSs)

The terminology used in RDBMSs differs slightly from the one used in relational algebra, although the concepts are the same. Relations are called *tables*, tuples are called *rows*, and attributes are called *fields*. Modern RDBMS extend the basic relational data model with other concepts, such as constraints, triggers, views, stored procedures, and user-defined

functions. The standard language for creating, modifying and querying a database is SQL [116], which is more powerful than relational algebra. Furthermore, RDBMS offer functionality to support the efficient execution of database queries and the prevention of data loss, such as indexes and logs. More about theses concepts can be found in [84].

Modern RDBMSs are very mature and offer many advantages. They are very reliable, very efficient and offer advanced features for safety and security, such as transaction processing and role-based access control [190]. They support automatic checking of integrity constraints on the data, and event-based data management with triggers. Most good RDBMS can be programmed with stored procedures and extended with user-defined functions. They can be accessed over a network, and distributed using database replication techniques. With SQL, access to relational databases is relatively standardized. There are very good free open-source implementations, e.g. Firebird [211].

I implemented the repository of the AP1 system on a RDBMS because of the formal maturity of the relational data model, and the practical maturity of modern RDBMS. The relational data model reflects essential mathematical concepts, which allow it to define a database in a concise manner. Furthermore, RDBMS satisfy many of AP1's requirements, as we will see later on.

2.2 The Object-Oriented Data Model (OODM)

The object-oriented (OO) data model emerged in the context of object-oriented programming (OOP), which emerged in the 1960's. Probably the first language to support object-orientation is Simula [168], which was developed by Ole-Johan Dahl and Kristen Nygaard at the Norwegian Computing Center, Oslo. As the name suggests, the Simula language was intended for simulation of complex systems. It was successfully used, for example, for the simulation of telephone traffic systems, electronic circuits, aircraft surveillance, and neural networks [175]. Unsurprisingly, one of its inventors, Kristen Nygaard, was very involved in the field of operations research.

However, object-orientation was not used much for mainstream software development until the 1980's. Its influence became stronger with languages such as C++ [204], which is an OO extension of the popular C language, and the emergence of GUIs. The usage of OO languages for GUI programming was perceived as a good match. In the 1990's OO established itself as the predominant software development paradigm. The Java language, which also makes use of the popular C-style syntax, contributed a lot to its popularity with its virtual machine concept and use on the World Wide Web.

OOP came with a considerable hype as many of its advocates claim that it "revolutionized" software development. However, there is no clear evidence that object orientation makes software development significantly more efficient, e.g. see [184, 136]. In fact, several

studies cast a shadow on its alleged benefits [106, 37, 61].

Basic OODM concepts are classes, objects and inheritance. *Classes* are product types that contain typed data fields. *Objects* are values of class types. Objects are identified with object references, which are artificial values that usually simply describe the memory location of an object. *Inheritance* makes it possible to define a hierarchy on classes: a *subclass* can be defined as an extension to a *superclass*, which means that it inherits its fields and can be used in its place – a property also known as the Liskov substitution principle [140]. In that way, common parts of classes can be reused by extracting them into a common superclass.

OOP adds to these rather data-related concepts features for managing executable code, such as methods, method polymorphism and dynamic binding. It should be noted that these concepts are not new. OOP is characterized by the way they are mixed and used, e.g. in the form of OO design patterns [96]. Furthermore, OO languages often support features that are not typically OO, such as features from the functional programming paradigm. Consequently, OOP is not a "pure" concept by itself, i.e. not at all orthogonal to other programming paradigms. It is possible, and not unusual, that programs written in a non-OO language implement OO features by hard-coding them explicitly.

Methods are code routines that are associated with a particular class, and thus are meant to work primarily with the data defined by that class. The information hiding principle in OOP relies primarily on the fact that classes encapsulate data and code that accesses that data, and that data and code within a class can be shielded from external access. Similar to fields, also methods can be inherited from a superclass to its subclasses, so that functionality common to several classes can be extracted into a common superclass. It is important to note that this characteristic property of OO to bundle data and code is actually not present in all OO languages, e.g. CLOS [95] and Dylan [193] maintain separate hierarchies for methods and classes.

Method polymorphism means that there can be different versions of a method with a particular name. Through *overloading* it is possible to define several methods with the same name in the same class that are distinguished by their parameter types. *Overriding* makes it possible to redefine a method of a superclass in one or more of its subclasses. This is used through *dynamic binding*, also known as dynamic method dispatch: when a method is called on an object, and the method has been defined for different classes through overriding, then the definition that is used is chosen at runtime depending on the actual type of the object.

The emphasis on methods that are associated with classes, and the extensive use of advanced features for methods such as overloading and overriding, gives OOP a more behavioral focus. Objects are often not so much conceived in terms of what they are, i.e. the data they represent, but in terms of what they do. This leads to a stronger focus on

object interaction patterns, i.e. protocols defining how objects should be used. Classic database approaches are different in that they naturally focus on the data of a system, and leave most behavioral aspects to the application.

Common metaphorical misconceptions of OOP portray objects as actors, although they are merely data containers associated with executable code, which are passive entities. This can lead to confusion because the active entities are not really the objects, but the processing threads executing the methods. This becomes clear when dealing with multi-threaded systems in which some objects are shared by multiple threads. In such situations, the perception of classes as abstract data types becomes much more important than behavioral views.

The OOP ideal of a "natural" fragmentation of a system into classes does not always work out. For example, responsibilities of classes, i.e. the question which methods they contain, is in many cases quite arbitrary. A method has one intrinsically given parameter, which is the object on which it is called – usually available through the keyword `this`. But as soon as a method requires no information at all or more then one complex parameter, it is not clear which class it should be associated with. This is why many systems contain "static classes" that are primarily used as containers for methods.

The idea of OOP as "real-world modeling" with a direct correspondence to real objects and their behavior is rather a misconception. OOP does not produce software systems that are any more real than ones produced with different methodologies. This probably has its roots in OOP's emergence from the field of simulation, where such analogies are appropriate. In fact, the very nature of simulation implies that the aim is to emulate the behavior of real-world objects in a realistic way.

2.2.1 Object-Oriented Database Management Systems (OODBMSs)

OODBMSs emerged during the 1980's, and at the beginning of the 1990's there were many commercial products available on the market. The popularity of OODBMSs is related to the popularity of OOP in general, which grew also during that time. Another motivation was the fact that use of the established RDBMS technology from an OO program required additional work because of the differences between the OODM and the RDM. If the transient data of an OO program is to be stored or retrieved from a relational database, then an *object-relational mapping* has to be performed, which requires additional effort and potentially additional supporting technology. This difficulty is known as *object-relational impedance mismatch*.

Object-oriented databases are essentially based on the same data model as OO programming languages. However, while the objects in an OO program are usually only

stored in the working memory and thus transient, those objects can be made persistent with the help of an OODBMS. With good OODBMSs this requires only little additional code, and persistent objects can be handled in much the same way as ordinary objects. It is possible to access and manipulate objects using their respective methods instead of some generic database operations. This is also called *transparent persistence* or *persistent programming*.

OODBMSs avoid the object-relational impedance mismatch by essentially using only one data model: the one from the respective OO programming language. Only an OO model of the application data needs to be developed, instead of having to develop an additional relational data model. Furthermore, synchronization of persistent objects between the secondary storage and working memory may be automated. All this can make OO development more efficient than it usually would be when using a RDBMS.

There is no need to develop code for the caching between an RDBMS and the OO working memory because this is handled by the OODBMS: when an object is accessed through a query or the navigation along object references, and this object is not yet in the working memory, then it is automatically loaded, e.g. see [120]. In contrast to this, an OO program using a RDBMS usually has to load data from the database and create corresponding objects by itself. The cache that is created like this may not be tightly integrated with the RDBMS, so there may be issues with *cache synchronization*: the data in the database and the data in the cache may get out of sync, e.g. because other applications access and change the database concurrently.

In addition to the standard OOP features, many OODBMSs provide additional database-related functionality, such as transactions, concurrency control, support for data recovery and a declarative query language. However, different OODBMSs provided different interfaces to such functionality, e.g. different query languages. In order to foster interoperability between the different systems, several OODBMS vendors formed the Object Data Management Group (ODMG) in 1991. Together they developed several versions of a standard for ODB connectivity. The last one, called ODMG version 3.0, appeared in 2001 [38]. It defines, for example, the object query language (OQL), which is similar to the SQL language for RDBMSs.

2.2.2 Example

The following example shows how the data model from Sect. 2.1.1 can be implemented using the OODM, and code for retrieving and changing data. The example uses the Java language and the standard ODMG language bindings. First of all, classes have to be defined for bank accounts and persons:

```
1  public class BankAccount {
2    public String number;
3    public double balance;
4  }
5
6  public class Person {
7    public String name;
8    public String address;
9    public BankAccount bankAccount;
10 }
```

Now we can connect to a database, and retrieve objects with OQL queries:

```
1  Implementation impl = new com.vendor.odmg.Implementation();
2  Database db = impl.newDatabase();
3  Transaction ta = impl.newTransaction();
4
5  db.open("myDB", Database.OPEN_READ_WRITE);
6  ta.begin();
7
8  OQLQuery query = new OQLQuery(
9    "select p from Person p where p.name = \"Gerald Weber\"");
10 Collection result = (Collection) query.execute();
11 Iterator i = result.iterator();
12 while(i.hasNext()) {
13   Person p = (Person) i.next();
14   p.bankAccount.balance += 1000000;
15 }
16
17 ta.commit();
18 db.close();
```

The first line loads the vendor-specific ODMG database driver. Lines 2 and 3 create an object for a database and an object for a transaction. Line 5 opens database `myDB` for read and write access. Line 6 begins a new transaction. Lines 8 and 9 define a query that retrieves all Person objects with a particular name. Lines 10 to 15 execute the query and retrieve the resulting objects one by one through an iterator in a while-loop. Line 14 demonstrates how the retrieved objects can be handled just like ordinary ones. Line 17 commits the transaction, thus writing any changes made back into the database if the transaction succeeds. Line 18 closes the connection to the database.

2.2.3 Problems

OODBMSs support certain very fast access routes to data, i.e. those that are given by the references between objects. In the above example, one can easily get the bank account of a person because each `Person` object contains a reference `bankAccount` to its corresponding `BankAccount` object. Navigating along such object references or pointers is very fast because references usually directly describe the location of the referenced object. The standard OOP implementation of references or pointers is that of a variable which contains the memory address of a data object.

RDBMSs may need more time in certain cases to connect corresponding rows because the join operation on tables usually works differently: instead of using navigational techniques such as direct pointers, joins express such connections in a declarative manner using a logical equivalence expression on field values, i.e. foreign and primary keys. The key field values which are used for joining are usually not related to the location of a row, therefore a join has to work out indirectly which rows are connected by searching for rows with particular values. In certain cases this takes longer because it requires additional steps such as the usage of indexes or scanning of whole tables.

However, the pointer-based navigational access routes defined by OO data are only very specific ones. In the example, it is very easy to find the bank account owned by a particular person, but it is impossible to find the owner of a given bank account in a fast, navigational manner. There is no reference from `BankAccount` objects to the corresponding `Person` objects. Unfortunately, navigational access routes with references or pointers are unidirectional. For many cases, there might simply be no pointers between objects that one might want to relate to each other.

For general-purpose queries which do not always follow the same navigational pattern, pointer-based access paths fail. Therefore declarative querying techniques as they are used by RDBMSs are superior in this case, making it possible to relate data in new, unforeseen ways, e.g. see [154]. One also has to note that nowadays index lookup techniques are very mature, and joins can be performed very efficiently. In addition to that, computing hardware is getting steadily faster, with Moore's law still remaining a valid prediction, and even old performance bottlenecks such as hard disks are being replaced by much faster solid-state memory.

In the light of such drastic performance improvements, the necessity of pointers as a performance optimization is questionable. Pointers have intrinsic disadvantages in that they usually serve two purposes: identification and localization. As a result, it is usually not easily possible to change the storage location of an object without changing its identity, which normally must not change. This constraint impedes flexible memory organization and may lead to memory fragmentation.

Furthermore, pointers are likely to be only local identifiers because different database

clients organize their memory independently: some other database client may have a different pointer for the same data, or even the same pointer for different data. Even if all clients were using the same pointers for a particular database, an application might need to connect to several databases, thus potentially running into pointer conflicts again. The linear address spaces of different databases might naturally overlap.

Most importantly, pointers are not data identifiers at all. They are on a less abstract level. All they do is identify objects, which may contain any data elements. The same data element may be represented in different objects, making it clear that objects do not necessarily provide a very abstract description of the data itself. In contrast to this, the keys used in RDBMSs identify the data, and are thus more abstract.

RDBMSs separate the concern of identifying a data element from the concern of localizing it. This means that memory organization is very flexible. It is not important where data is stored, which means that this can be exploited for optimizations, e.g. by storing rows that are frequently accessed together on the same page in memory. Such optimization can happen a posteriori. Different database clients can manage their memory independently, and keys can be designed to be truly global data identification tags.

The idea of a persistent object store as it is implemented by OODBMSs can have advantages when new database operations need to be programmed on a very low level. Compared to SQL, OOP provides such a low level. However, this easily becomes a disadvantage when navigating over large sets of data because purely navigational data access with OOP renders certain cache optimization techniques infeasible.

Consider the example from the previous section. If a large number of Person objects is present in memory, and we want to get all their corresponding BankAccount objects, this would typically be done by dereferencing field `bankAccount` for each Person object in a loop. If one of those BankAccount objects is not in the working memory, it has to be loaded into it. If the working memory is full, then previously loaded BankAccount objects might need to be swapped back to secondary storage.

Consider now that several Person objects are sharing the same BankAccount objects. With increasing numbers of objects, it becomes more likely that BankAccount objects that are swapped to secondary memory are needed later on for other Person objects. As a result, certain pages have to be loaded from secondary memory more than once, which results in a significant performance loss. It turns out that the join techniques used in RDBMSs perform better in such cases because they are able to use a proper caching strategy. On the low level of loops and individual dereferencing operations, this is not possible.

Another disadvantage of OODBMSs is a certain lack of interoperability of features and tools. The ODMG standard has not been actively maintained since its last release in 2001. Despite the efforts of the ODMG, there are many vendor-specific differences, so that

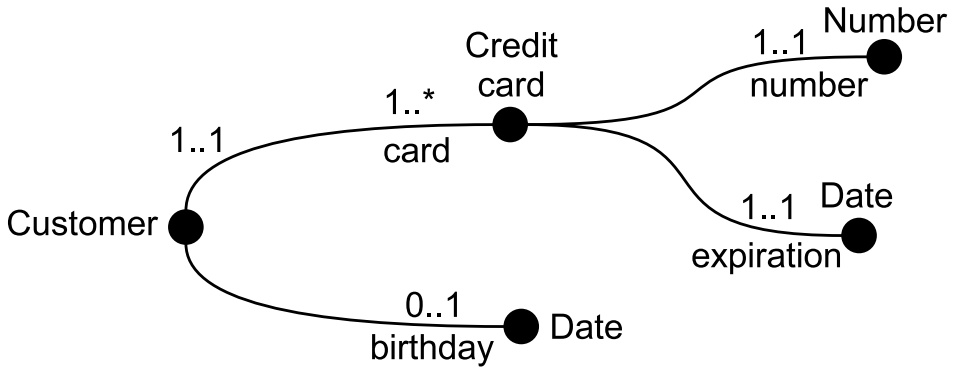

Figure 2.2: Example parsimonious data (PD) model.

tools for administration, reporting, backup, data mining, etc. cannot easily be used with different OODBMSs, e.g. see [154]. This heterogeneity is partly caused by the fact that OODBMSs do not have such a firm mathematical foundation as RDBMSs. RDBMSs are based on simple concepts from set theory and predicate logic, which foster clear, abstract and consistent data organization.

2.3 The Parsimonious Data Model (PDM)

The parsimonious data model (PDM), which is described in [76], is a very simple, formally well-defined data model, which relies – similar to the RDM – on sets and relations. The PDM is, in fact, a kind of RDM with a higher level of abstraction. It essentially consists of three concepts: *entity types*, *relation types* and *roles*.

The PDM distinguishes between entity types and relation types. This is similar to the distinction between primitive attribute types and relation types in the RDM. Entity types contain data elements of a system, and relation types describe the associations between the data elements. It is based on fundamental concepts of set theory, which distinguish sets on the one hand, and relations that are defined on these sets on the other hand.

It is easiest to understand the PDM with the help of an example. Figure 2.2 shows a simple model for data about customers with their credit cards and associated information. This figure uses the standard visual representation of PD models. Each customer has at least one credit card with a number and an expiration date. A credit card belongs to exactly one customer. Optionally, a date of birth is given for a customer.

Entity types are represented as filled circles. Labels at the circles beginning with uppercase letters denote entity type names. Entity types are sets of values, e.g. identifiers or primitive values. In the example, entity type Customer contains identifiers for all the customers, type “Credit card” contains identifiers for all the credit cards, and the types Number and Date contain all possible numbers and dates, respectively. The data elements

of an entity type are called its *instances*.

The connections between the circles represent relation types between the corresponding entity types. Relation types describe how data can be meaningfully represented by connecting the elements of entity types in a typed manner. The relation type between Customer and "Credit card" specifies that all data instances of the model will contain a relation pairing the identifiers of customers with the identifiers of their credit cards. The connections that can be made between instances according to a relation type, i.e. the elements of relations, are called *links*.

The connections of entity types to relation types are identified by roles. Labels can be put at the ends of the connecting lines in a diagram in order to give an entity type participating in a particular relation type a context-specific name. Type Date, for example, is used once in the context of an "expiration" date, and once as a "birthday". In order to distinguish role names from entity type names, they start with lower case letters. This is similar to the use of labels in the concept of records, where each component of a record is accessed through a label. The record concept is reused in the class concept of object-oriented programming.

Role names can be used to navigate in PDM data. They correspond to record labels, with the relation types being the record types. This is similar to the canonical OOP approach of data access through variable names of classes. If one has an instance of type Customer, then accessing the role with name "card" will yield all instances of type Card which are associated with that particular customer. Navigation with roles is equivalent to the use of joins in the RDM, with the relations of the PDM directly corresponding to the joined RDM relations.

Note that not all ends of the connecting lines need to have labels with role names. Role names are optional, and serve merely as a specification and navigation aid. If there is no role name for one of the connections of a relation type, this does not mean that there is no role. It merely means that the role is unnamed. It is still possible to navigate a relation of that type in the corresponding direction. Role names are usually omitted if there is no apparent need to navigate a relation in a certain direction: for example, it might not be very useful to navigate from a date to all the credit cards that expire on that date.

In contrast to the RDM, there is no explicit notion of attributes. All associations between data elements are modeled as relation types, and an attribute is simply a relation type which allows at most one data value to be linked. A PD model can be interpreted as a relational schema in which each association is modeled with its own relation. This is also known as the direct PD-relational mapping [76]. However, there exist more compact mappings between the PDM and the RDM.

Note that entity types and relation types can be represented multiple times in visual

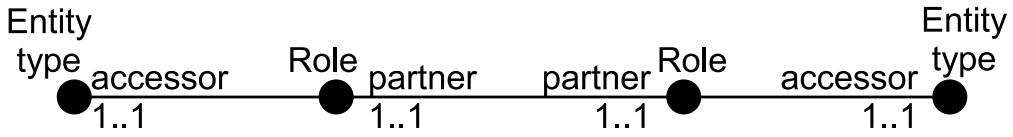

Figure 2.3: The PD metamodel of the AP1 system. Entity instances are associated to other entity instances indirectly by a pair of roles, forming binary relation types.

diagrams. In the example, the entity type Date is represented twice, once for the expiration date of a credit card and once for a customer's birthday. This way, crossing lines can be avoided and diagrams can be laid out more clearly. Furthermore, this emphasizes the set semantics of the PDM: it does not matter how many times a type is represented in the set of all types. As a result, models can be composed and decomposed more easily without loss.

The following is a formal definition of a PD model:

Definition 1 *A PD model is defined by a tuple* (E, P, R, e, p) *where* E *is the set of entity types,* P *the set of relation types and* R *the set of roles.* $e\colon R \to E$ *defines which entity type owns a role, and* $p\colon R \to P$ *defines which relation type a role belongs to.*

2.3.1 The PD Metamodel

When using models, we have to distinguish different levels of structural information. A model is basically a data type that allows us to specify data which adheres to certain constraints. This data can in turn serve as another model description for other data.

The PDM uses three levels of structural information: the *metamodel*, which describes how types can be specified, *models* which are types defined by the metamodel, and *model instances*, which are data following the constraints of a particular model. In some approaches all three levels are referred to as "models", and metamodels end up being called "metametamodels", which can easily lead to "meta-confusion".

The PD metamodel is illustrated in Fig. 2.3. It is called metamodel because all correct PD models are its model instances, i.e. all correct PD models are data that have the PD metamodel as their type. Consequently, the PD metamodel ensures that PD models are well-typed, and thus guarantees that they are syntactically compatible. The PD metamodel, in turn, is a valid PD model itself. Therefore, it is also an instance of itself.

Entity types are represented by instances of type "Entity type", and roles by instances of type Role. Each "Entity type" instance is connected to the Role instances representing the roles that the corresponding type can access. The "Entity type" instance is the accessor of those roles. The two roles at opposite ends of a relation type are called partners.

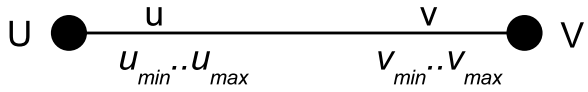

Figure 2.4: Multiplicities in the PDM.

The fact that the metamodel associates entity types directly with the roles they can access, and not the ones they own and can be accessed through, is different from the PDM diagrams. In the model diagrams accessible roles are drawn opposite and not next to the accessing entity type. However, having accessible roles directly associated with their accessor types in the metamodel makes it more convenient to retrieve them for navigation.

In contrast to other metamodels, e.g. the metaobject facility (MOF) [172] of UML, the PD metamodel is very simple. As Fig. 2.3 shows, two entity types and two relation types are sufficient. The figure shows four circles because entity types "Entity type" and Role were represented twice for better understanding. The PD metamodel is symmetric because the binary relation types used in PD models are undirected, and both their ends can be described in the same manner. The fact that the relation types in the metamodel are binary does not impose any limits on its expressiveness: an n-ary relation type can be transformed into an additional entity type and n binary relation types – a process also known as *reification*.

2.3.2 Multiplicities

As I have already indicated, the PDM also provides a notion of multiplicities. In the model of Fig. 2.2, for example, each customer has at least one credit card, and there is no maximum number of credit cards that a customer can have. This is indicated by the 1..* multiplicity at role "card". Each credit card, in turn, must be owned by exactly one customer, which is expressed by the 1..1 multiplicity at the opposite role.

Let us consider the relation type between entity types U and V that is illustrated in Fig. 2.4. $v_{min} \in \mathbb{N}$ indicates the minimum number of V instances that are linked to each U instance through role v. $v_{max} \in \mathbb{N} \cup \{*\}$ indicates the maximum number of V instances that are linked to each U instance through role v. If v_{max} is *, then there is no upper limit to the number of linked instances, otherwise $v_{min} \leq v_{max}$. If $v_{min} = v_{max}$, then this can be abbreviated by simply writing v_{max} instead of $v_{max}..v_{max}$. Analogously, u_{min} and u_{max} specify the lower and upper limits for the number of U instances that are connected to each V instance through role u.

Multiplicities are optional constraints, and thus need not be used unless a data model designer wishes to do so. If there is no multiplicity given for a role, then this role has a default multiplicity of 0..*. This default multiplicity reflects the most general case, which

is consistent with the notion of multiplicities being constraints. Each multiplicity that is different from 0..* is just a special case, i.e. a restricted form, of a 0..* multiplicity.

2.3.3 Primitive vs. Non-Primitive Entity Types

One has to distinguish two different kinds of entity types: primitive and non-primitive ones. This distinction is similar to the one drawn by many OO programming languages, such as Java or C#. Those languages distinguish primitive types from reference types. The values of primitive types are usually directly stored, while the values of reference types are just referenced by a pointer. As a result, primitive types are immutable, while reference types are mutable, i.e. a value can be modified with its reference staying the same.

The defining property of a primitive entity type is that it always contains the same instances. In the model in Fig. 2.2, for example, Date always contains all dates and Number all numbers. It is not possible for a program to add new instances to the type or remove existing ones. Primitive types do not necessarily have a very simple structure. They could describe, for example, Strings of arbitrary length, very high-precision numbers, or binary data.

The defining property of a non-primitive entity type is that it does not always contain the same instances. It may, at first, contain no instances at all, and instances may be added to them as time goes by. Such types are finite because instances have to be created explicitly. Their instances are unique keys that are used to identify data elements of a domain. Before such data elements can be used in a system, corresponding instances have to be created for them.

Analogously, in OOP objects have to be created before they can be referred to and used in a system. In an assignment statement, one can only assign a type-compatible object to a variable of a reference type that actually exists at that point in time. It is not possible to choose out of all the theoretically possible objects, but only out of those that were created before the assignment.

In the model in Fig. 2.2, for example, the entity types Customer and “Credit card” are non-primitive. The data about each customer and each credit card have to be entered into the system before they can be used, thus creating the sets of identifiers that make up the respective entity types. If the sets contained all theoretically possible instances for customers and credit cards, then the problem becomes deciding which instances actually refer to valid, existing customers or cards.

There would be a problem with constraints on entity types if a non-primitive entity type would contain all possible instances. By definition, a constraint on an entity type, such as a multiplicity, must be satisfied for all its instances. If a non-primitive entity type contained all possible instances, then all those instances would have to satisfy the

constraints. For example, they would have to have a minimum number of links to other instances, which would create an arbitrary set of data. It would be unclear how exactly such data would be defined, and how such data would be distinguished from "real" data that was entered by a user and not just added to satisfy constraints.

The notion that instances of non-primitive entity types have to be created, and do not exist by default, is natural and goes perfectly with the technical capabilities of a DBMS. Non-primitive entity types take on the role of record identifiers or primary keys in a relational database, with the associated entity types being like fields. Non-primitive entity types are used as composites, i.e. to organize data that belong together. A database can contain only a finite number of such records, and a DBMS can only check and enforce constraints on a finite number of them.

This point can be made clear with the example model in Fig. 2.2. This model prescribes that each customer has at least one credit card, and that each credit card has a number and an expiry date. Let us consider what would happen if entity type Customer contained all theoretically possible instances. All those possible instances would have to be associated with a credit card each, and all the credit cards would have to have credit card numbers and expiry dates. Of course, most of the Customer instances would not refer to valid customers, and the credit cards of those customers would naturally not be valid as well. What numbers and expiry dates would all these invalid credit cards have?

It clearly would not make sense, and only cause trouble. One could not scan through the sets of non-primitive instances, e.g. for query purposes, without having to decide which customers or credit cards are valid and which are not. Furthermore, there might be card number collisions between valid and invalid credit cards. Such problems do not exist if we accept that non-primitive instances only exist if they were explicitly created.

2.3.4 Inheritance

There is a need for subtyping when handling complex data, e.g. for data structures that should be used generically. Most OO languages define a common supertype Object for all other class types, so that a variable of type Object can hold objects of any class. Subtyping is very important for common language features that rely on polymorphism, such as overriding and dynamic binding. Some of them have been described in Sect. 2.2. Those features are heavily used in OOP, e.g. in order to support the "don't ask what kind" pattern, which avoids explicit case distinctions by using dynamic binding.

The subtype relation between types is usually established through inheritance. With inheritance it is possible to extend existing product types with additional components and functionality, without compromising the existing ones. Instances of subtypes can be substituted for instances of supertypes because properties that are valid for a supertype are also valid for its subtypes. This is also known as the Liskov substitution principle [140].

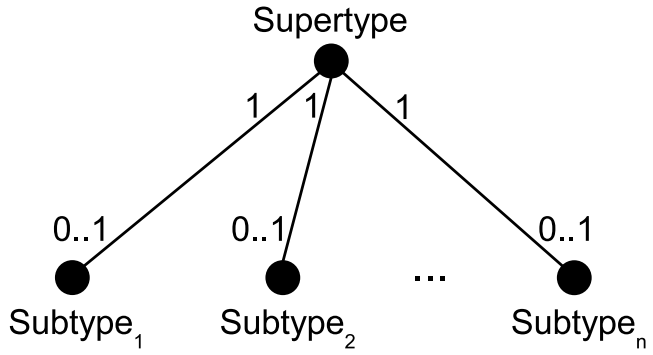

Figure 2.5: Attribute-oriented inheritance in the PDM.

Therefore, the subtype relation is also called "is-a" relation. Types that are linked directly or indirectly through inheritance are called *related types*.

Programming languages with inheritance are commonly divided into languages that support multiple inheritance (MI), and ones that support only single inheritance (SI). SI means that a subtype can inherit from only one supertype, whereas in MI a subtype can inherit from arbitrarily many supertypes. SI avoids some ambiguities and is much easier to implement due to the linear nature of common computer memory. Only few languages offer full MI, e.g. C++ and CLOS. Modern mainstream OOP languages such as Java and C# offer an extended form of SI: a class can only inherit from one superclass, but can implement several interfaces, which become part of its type.

The PDM proposes the notion of *attribute-oriented inheritance*, which is built on top of the ordinary PDM in the form of additional constraints. Attribute-oriented inheritance is non-intrusive in that it does not require any modification of the PDM. Furthermore, it is a pattern rather than a data model extension because no new concepts are necessary. This has the advantage that inheritance can be understood in terms of the PDM, and does not become an opaque feature like in most programming languages. In addition to this, inheritance remains flexible because it can be adjusted to individual needs with additional constraints.

Figure 2.5 illustrates the basic principle of attribute-oriented inheritance. The Entity types $\text{Subtype}_1, \ldots, \text{Subtype}_n$ inherit from entity type Supertype. The inheritance relation is modeled by ordinary PDM relation types. Navigation along the links of those relation types corresponds to *upcasting* and *downcasting* of the corresponding type. Upcasting means that the type of an instance is converted to one of its supertypes, downcasting means that the type is converted to one of its subtypes.

The multiplicities of the relation types make sure that inheritance is modeled correctly. For each instance of a subtype, there must be a corresponding instance of each of its supertypes, so that an upcast can be performed. For each subtype of an instance, there must be either one instance or none, so that downcasts can be made accordingly. Note

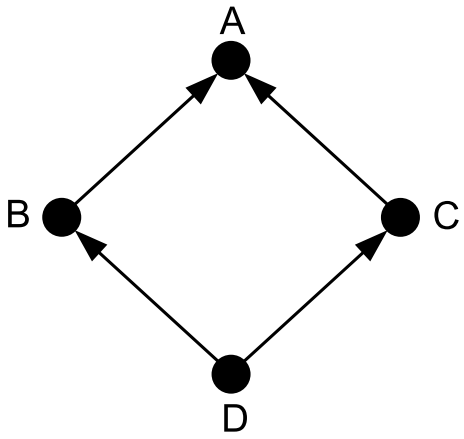

Figure 2.6: Example PD model using attribute-oriented inheritance.

that this notion of inheritance supports MI because each type can have arbitrarily many inheritance relation types to supertypes.

Figure 2.6 shows a PD model that makes use of attribute-oriented MI. Entity type D inherits from entity types B and C, which in turn both inherit from entity type A. In this diagram a shorthand notation for attribute-oriented inheritance relation types is used. Instead of specifying the multiplicities analogously to the model in Fig. 2.5, the inheritance relationships simply have an arrowhead pointing towards the supertype. This shorthand is similar to the notation for inheritance in OO data models, and also reminiscent of the $\rightarrow$ symbol for logical implication. The analogy to logical implication is appropriate because the existence of an instance of a subtype implies the existence of a linked instance of each of its supertypes.

The notation of Fig. 2.6 does not only abbreviate the constraints required for attribute-oriented inheritance, it also serves as a semantic marker. It distinguishes relation types that are actually meant to represent inheritance from ones that coincidentally have the same multiplicities. If there are several relation types with these multiplicities to the same entity type, this helps to decide whether a relation type represents inheritance or not. Such a semantic difference matters, for example, for upcasting and downcasting, in particular if a system is to support automatic upcasting, which is common. Such additional metainformation can be easily added to the PD metamodel.

MI allows a type to inherit the features of several other types, which may in turn inherit the features of other types. Features, in this case, can be understood as the data structure parts that are contributed by a type. The semantic problems of MI stem from the fact that two related types may inherit from the same type. The question is if features of a type that is inherited more then once are shared or represented more than once in data instances.

Such a problem arises, for example, in the model in Fig. 2.6. Is the feature of A,

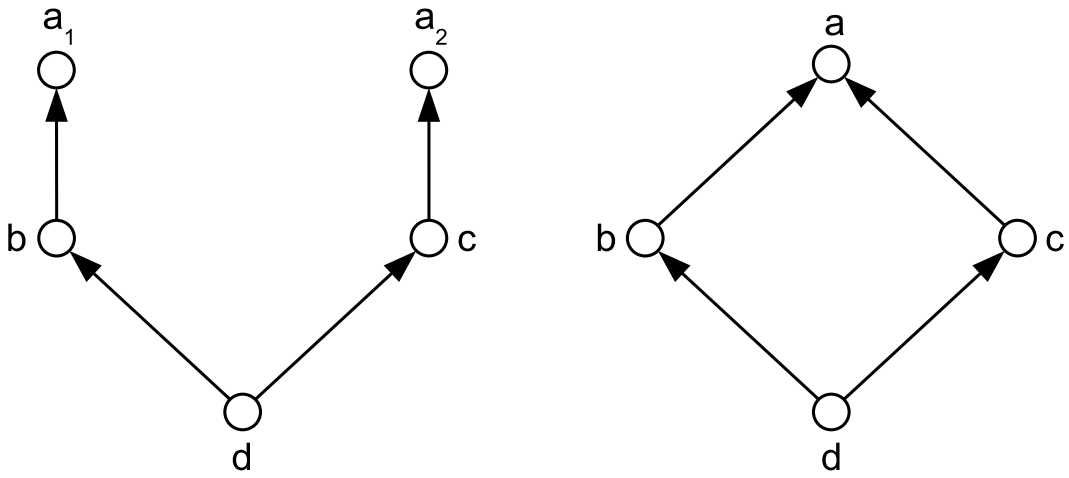

Figure 2.7: Two different model instances for the model in Fig 2.6: separate representation (left) and shared representation (right) of features that are inherited more than once.

which is part of both the feature of B and the feature of C, represented twice for every D instance? Let us consider this with a concrete example: let A be type Person, B be type Student, and C be type Employee. Both Students and Employees are Persons. Then, let D be type Tutor, which is both a Student and an Employee.

The question is whether the Person feature that is part of the Student feature of type Tutor is the same as the Person feature that is part of the Employee feature. Do we need to store two separate data sets of type Person for each instance of Tutor, or is it enough to have a single one that the data sets for Student and Employee share? In this case we would probably decide for Person to be a shared feature of Tutor, but there are other cases where separate representations of features may be preferable. Many MI languages allow a developer to choose between a shared or separate representation of features that occur multiple times within the same subtype.

The same choice has to be made when attribute-oriented inheritance is used for MI. Figure 2.7 shows two different possible instances of the model in Fig. 2.6. a, a_1 and a_2 are instances of entity type A. b, c and d are instances of entity type B, C and D, respectively. On the left side, the feature A, which is inherited twice by D, is represented twice. On the right side, feature A is only represented once, and thus shared by B and C. It is hence up to a developer which one of the two possibilities is chosen.

2.3.5 Example

This section elaborates the example of Fig. 2.2 a bit further. In particular, it provides a specification of the model using Def. 1, shows how a data instance can be defined formally, and how it can be represented graphically. Furthermore, an example is given of how the model can be mapped to an appropriate relational schema.

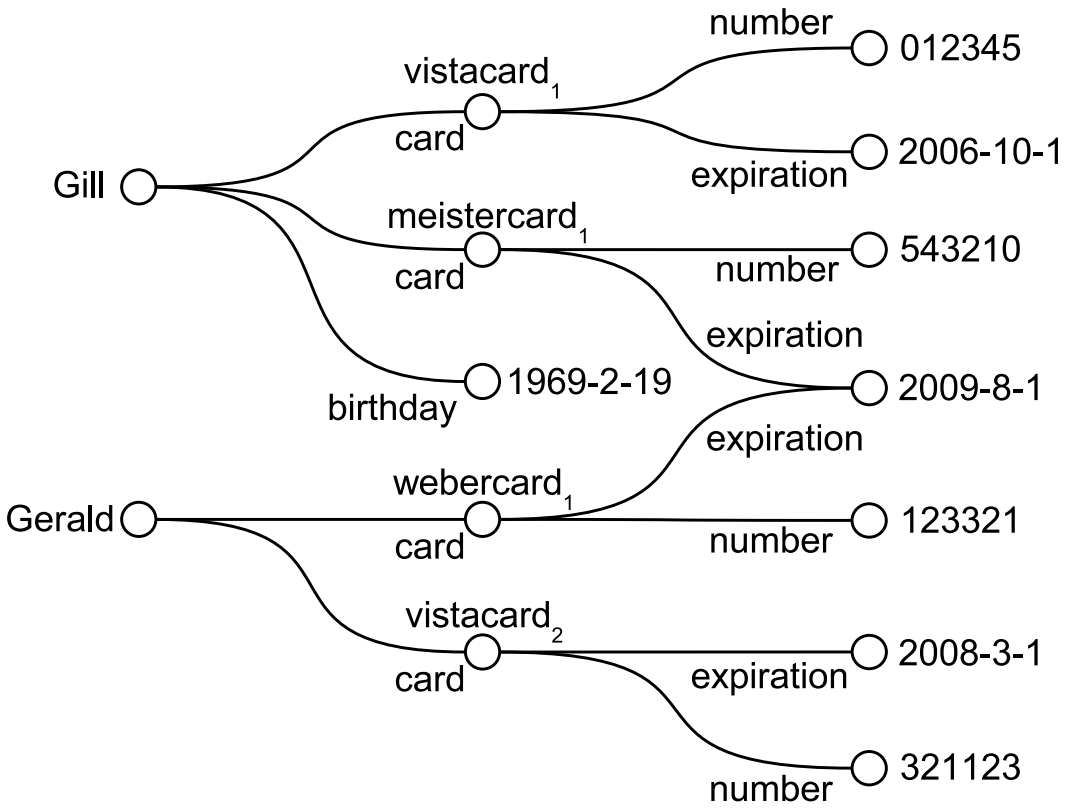

Figure 2.8: Example data for the PD model in Fig. 2.2.

According to Def. 1, the model of Fig. 2.2 can also be expressed as the following tuple:

$$
\begin{aligned}
(\ \{\ & \text{Customer}, \text{Credit card}, \text{Number}, \text{Date}\}, \{p_1, p_2, p_3, p_4\},\\
\{\ & \text{card}, \text{number}, \text{expiration}, \text{birthday}, r_1, r_2, r_3, r_4\},\\
\{\ & \text{card} \mapsto \text{Card}, \text{number} \mapsto \text{Number}, \text{expiration} \mapsto \text{Date},\\
& \text{birthday} \mapsto \text{Date}, r_1 \mapsto \text{Customer}, r_2 \mapsto \text{Card},\\
& r_3 \mapsto \text{Card}, r_4 \mapsto \text{Customer}\},\\
\{\ & \text{card} \mapsto p_1, r_1 \mapsto p_1, \text{number} \mapsto p_2, r_2 \mapsto p_2,\\
& \text{expiration} \mapsto p_3, r_3 \mapsto p_3, \text{birthday} \mapsto p_4, r_4 \mapsto p_4\}).
\end{aligned}
$$

The first component lists all the entity types. The second one lists all the relation types. Because the relation types do not have names in Fig. 2.2, the specification uses artificial names: p_1 is the relation type between Customer and Card, p_2 associates a Card with a Number, and p_3 a Card with an expiration Date. p_4 associates Customers and their Dates of birth. The next component lists all the roles, with the first four roles being the named roles from the diagram, and the following four roles, $r_1, \ldots, r_4$, their respective partners. The next component maps the roles to their owner entity types: role birthday to entity type Date, role r_1 to entity type Customer etc. The last component maps the roles to their relation types, i.e. card to p_1, r_1 to p_1 etc.

Figure 2.8 shows a diagram specifying example data, i.e. an example instance of the example PD model. The empty circles represent instances, and the connecting lines between them links. The labels at the circles represent instance identifiers, and the labels at the lines denote the roles with which instances are connected. The diagram does not specify the entity types of the instances, since it is relatively small and the correspondence between instances and entity types is apparent. However, it would be possible to add this information to the diagram, e.g. by including it into the labels on the circles.

Note that the multiplicity constraints are satisfied. One of the Customer instances has the optional birthday Date given, the other not. Both have at least one credit card, and all Cards have a Number and an expiration Date. Two of the credit cards have the same expiration Date, which is possible because each Date can be the expiration date of arbitrarily many credit cards.

The data can also be expressed formally with set-theoretical concepts:

$$
\begin{aligned}
\text{Customer} &= \{\text{Gill}, \text{Gerald}\},\\
\text{Credit card} &= \{\text{vistacard}_1, \text{vistacard}_2, \text{webercard}_1, \text{meistercard}_1\},\\
\mathit{links}(p_1) &= \{(\text{Gill}, \text{vistacard}_1), (\text{Gill}, \text{meistercard}_1), (\text{Gerald}, \text{webercard}_1),\\
&\quad (\text{Gerald}, \text{vistacard}_2)\},\\
\mathit{links}(p_2) &= \{(\text{vistacard}_1, 012345), (\text{vistacard}_2, 321123), (\text{webercard}_1, 123321),\\
&\quad (\text{meistercard}_1, 543210)\},\\
\mathit{links}(p_3) &= \{(\text{vistacard}_1, 2006-10-1), (\text{vistacard}_2, 2008-3-1),\\
&\quad (\text{webercard}_1, 2009-8-1), (\text{meistercard}_1, 2009-8-1)\},\\
\mathit{links}(p_4) &= \{(\text{Gill}, 1969-2-19)\}.
\end{aligned}
$$

Customer and "Credit card" are both non-primitive entity types, therefore their instances have to be listed explicitly. Date and Number are primitive entity types and their values are given naturally as the set of all dates and the set of all numbers. Therefore, they do not need to be further specified. Function *links* yields the links of a particular relation type that are actually present in the model instance. $\mathit{links}(p_1)$ contains the pairs for the links between the Customer and the "Credit card" instances, $\mathit{links}(p_2)$ the pairs for the links between the "Credit card" and the Number instances etc.

Figure 2.9 illustrates a possible mapping from the example PD model to a relational schema. The rectangles in the diagram represent relations, with the first line in bold print being the relation name and the remaining lines defining the attributes with their name on the left and type on the right side. For each of the two non-primitive entity types, a relation of the same name was created. I chose integer values as primary keys, corresponding to the instances of the non-primitive entity types. The primitive entity types were inlined into the relation of the respective non-primitive entity type that associates its instances with at most one of theirs. The names of the attributes are the names of

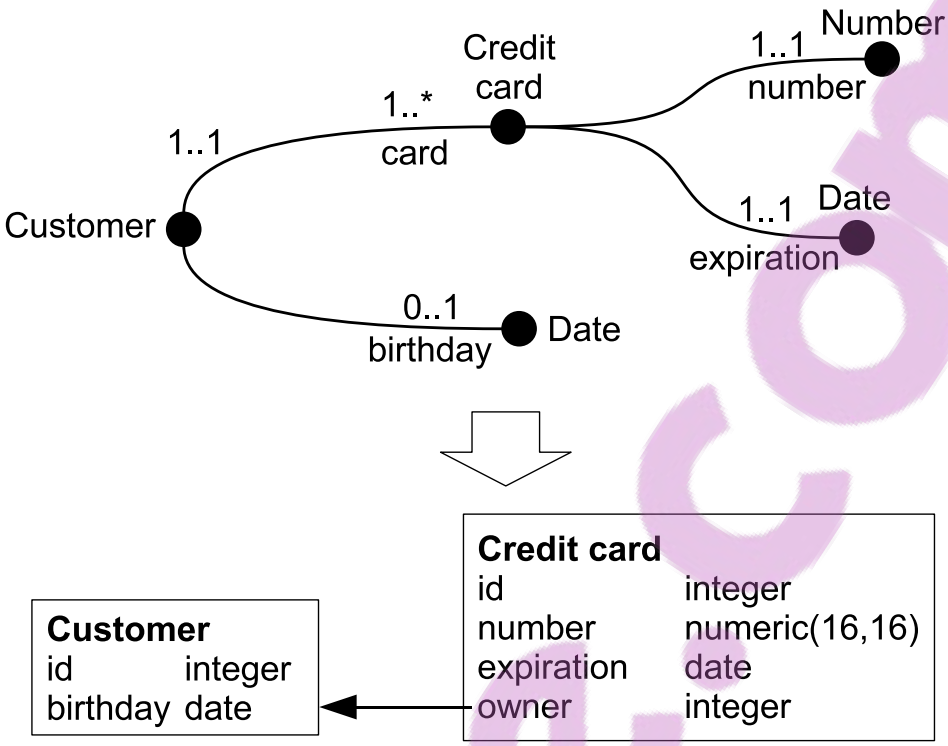

Figure 2.9: Example of mapping between the PDM and RDM.

the corresponding roles. The one-to-many relationship between Customers and "Credit cards" was implemented with a foreign key in attribute "owner" of relation "Credit card", so that each "Credit card" tuple can reference one Customer tuple.

2.3.6 Advantages

The PDM shares the advantages of the RDM, while offering a higher level of abstraction. It is simple and formally well-defined. It abstracts from low-level implementation details such as tables, foreign keys and joins, and a mapping between the PDM and the RDM can be performed automatically. Furthermore, the PDM offers additional concepts that are not directly supported by the RDM, such as multiplicities and a notion of inheritance.

While relational database schemas define multiplicities between data elements implicitly, the PDM allows the specification of them as additional constraints. The default semantics of a PD model without multiplicity constraints provides a maximum of flexibility by allowing arbitrary well-typed links in a relation. All multiplicity constraints are just special cases of this default case, which expresses the fact that single data values are in fact a special case of collections of values. Multiplicities can be added later on in order to refine a model in an iterative manner. The topology of a PD model is invariant with regard to changes of multiplicities, so that it can be defined independently of the multiplicity constraints.

A change of multiplicities in a data model that is implemented as a relational schema is likely to change the relations in the schema, as it has been described in Sect. 2.1.2. In many cases, foreign keys have to be changed or relations added. This impedes data model evolution because changes of the schema are substantial modifications that may not be easily made. During the evolution of a data model multiplicities are usually not completely understood from the beginning but have to be adjusted as the requirements are clarified. The PD model concedes this important degree of freedom to a developer by making multiplicities constraints that can be added later on, without changing the topology of a PD model.

One of the difficult and contentious issues of the RDM is normalization [17]. There exist different normal forms, which determine how much relation types have to be decomposed according to the functional dependencies that exist between sets of attributes. This can be complicated, and when done excessively may lead to many small tables that require a lot of joins.

In the PDM there are no tables which group data values together. Every association between data values is expressed in the form of relation types. Therefore, the PDM lifts the difficult issue of normalization from a rather technical onto a more structural level. Data modeling is done in a way that groups primitive data around non-primitive instances. Functional dependencies can help to determine what non-primitive types are needed in order to structure the data appropriately.

In contrast to object-orientation, where references are always directed and thus allow only unidirectional navigation, the PDM does not impose direction on relation types. This is a valid abstraction from a rather technical restriction that forces the user to think of implementation details too early in object-orientation. In many cases the question of whether associations should be directed is not relevant at all.

The PDM generalizes data structures in such a way that all the associations between data elements are modeled as relations, even the associations to primitive types. In OOP, for example, the associations between objects and their primitive field values are expressed rather implicitly. Primitive field values are often perceived as being natural parts of objects, not just associated with them, and only references are thought of as associations. The PDM makes all such associations explicit in a uniform manner, and can thus function with fewer conceptual notions.

The underlying relational structure of the PDM allows for a clean mapping onto a relational schema, so that RDBMS technology can be efficiently leveraged for storage and retrieval. This means that the PDM can make use of their features for concurrency control, transaction processing, recovery and security. As mentioned in Sect. 2.1.3, modern RDBMS are very mature, and can thus deal with high processing loads and huge amounts of data.

The PDM supports navigational techniques through the use of roles, as well as declarative techniques. Queries on a PD model can be expressed analogously to SQL. This makes PD model access very versatile, and allows the reuse of established standards. The PDM can be used, for example, to mitigate the object-relational impedance mismatch.

2.4 The Unified Modeling Language (UML)

UML is currently the most popular method for OO modeling. It was developed by the Rational Software Corporation in the 1990's, mostly by three of its employees, Grady Booch, and James Rumbaugh and Ivar Jacobson. Each of them had developed their own methods for OO software development, and UML can be seen as an effort to unify their different views. They themselves describe UML as a "graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system" [24]. UML was adopted by the Object Management Group (OMG) in 1997, an industry consortium including large companies such as HP, Sun and CitiGroup. The OMG now continuously maintains UML and develops it further. In 2005 UML also became an ISO standard [118].

The popularity of UML is closely related to the popularity of OOP. UML was quickly adopted by many software developing organizations, shortly after OOP languages such as Java gained widespread popularity. Thus it is not surprising that UML is very much tied to OOP, so that many of the arguments described in Sect. 2.2 apply to UML, too. Using technology that does not conform to common OOP patterns can result in a considerable impedance mismatch.

As Alex Bell [18] points out, the popularity of UML has frequently grown into insalubrious excesses: a hype about UML lead to what Alex Bell calls "UML fever", which is "a potentially deadly illness [...] plaguing many software-engineering efforts today". UML is often overestimated in a way that people believe it indistinctly legitimizes all modeling efforts or guarantees their success. It is forgotten that UML is a tool which has to be used with skill in order to get good results.

"UML fever" is reportedly causing significant increases in both the cost and duration of software development projects because of resources being wasted on UML modeling activities that provide little value for the software product [19]. People frequently place unrealistic expectations in UML, forgetting that it is just a modeling technique and not a software development process. Such unrealistic views are also caused by the fact that UML has become a buzzword that is exploited mercilessly for marketing. UML can be used for certain purposes during a software development project, but it can be misused for many more.

According to Alex Bell, UML modeling tools tempt inexperienced developers to spend a lot of time on very detailed models. Many UML tools with forward-, reverse- or

roundtrip-engineering capabilities only support models with a very low level of abstraction to better match the level of abstraction of source code. However, many experts think UML models are most useful as rough sketches. Scott W. Ambler, for example, suggests that the value of UML modeling lies in the modeling effort rather than the resulting model [18]. Modeling can be a very efficient activity for gaining a better understanding of a system, in particular as a group activity. Models should be "just good enough", and should not be used to capture the details of an actual implementation.

Some people forget that there have been useful modeling methodologies around for a long time, and that many complex software systems have been built successfully before the creation of UML. To many of the software engineers that started their careers in the "roaring nineties" [203], more traditional software engineering methodologies are simply unknown. There is no evidence that UML has significantly improved the situation of software development, but in the face of "UML fever" it has certainly caused some harm [18, 19].

With UML an attempt has been made to integrate many different OO development methods. As a general purpose modeling language, it is caught between many different requirements and expectations. Standards like UML are always political in the sense that many people with different interests are involved in their development, and many compromises have to be made. As a result, UML is very complex and feature rich to the extent that many people criticise it as being bloated, e.g. [93]. With 13 basic diagram types UML counts as a heavyweight modeling framework. The most frequently used UML diagrams are class diagrams, and also UML use case diagrams and sequence diagrams are relatively frequent, e.g. see [107]. Some of the UML concepts are used only very infrequently.

In order to get an impression of the UML's complexity, let us consider the metaobject facility (MOF) [172], which is the UML's metamodel. It comprises, in its core, about a dozen interconnected classes and a variety of other related concepts. The specification of version 2.0 of the MOF has 88 pages. This is significantly more complexity than many other metamodels, e.g. the PD metamodel, which was described in Sect. 2.3.1. The other parts of the UML 2.0 specification, put together, exceed a thousand pages. It is therefore not surprising that problems in learning and adopting UML are common.

Henderson-Sellers [107] sums up expert opinions from people who have been actively involved in the development of UML version 2.0. Most of the experts are rather disappointed by the outcome. UML is not formally defined, and the lack of a formal foundation makes its semantics imprecise, full of ambiguities and even some inconsistencies. As one of the experts puts it, many people do not make a secret of the fact that "it's a hack". While semantic imprecision can be appropriate for informal usage, it is unsuitable for automated tasks such as the generation of code. For example, it may cause incompatibility

between tools. Other points of criticism are inflexibility that impedes reusability, and a lack of focus on practically relevant concepts.

2.5 Extensible Markup Language (XML)

XML is a general-purpose markup language, which was developed in particular for representing data on the web. Like HTML, it is a subset of the Standard Generalized Markup Language (SGML), which in turn goes back to the Generalized Markup Language (GML) developed by IBM in the 1960's. XML first emerged in 1996, and became a W3C Recommendation in 1998. Currently there are two latest versions of XML, which were both published in 2006: XML 1.0 Fourth Edition and XML 1.1 Second edition [223]. They differ in the way certain characters are represented.

The original intention of XML was to solve a common problem of data representation on the web. On the web data was and mostly still is represented in the form of HTML documents, which not only contain the data but also information about its presentation such as layout, colors and font settings. Unfortunately, HTML evolved over time into a rather chaotic mass of features, many of which are browser specific. This is due to HTML's original feature set, which is insufficient for many applications, and the "browser wars" between corporates fighting for dominance on the Internet. The idea of XML is that it can be used purely for structured data representation, without defining its presentation. Data can be enriched with presentation information later on when necessary.

Although the complete XML specification and its associated standards have become large and complex, the basic concepts of semistructured data such as XML are simple [33]. This is certainly one of the reasons why XML also became popular as the basis of formats for data exchange between different applications. By now there are a myriad of such XML-based special-purpose languages, e.g. XHTML, RSS, SVG, XUL and XAML to name a few of the better known ones. Another factor for the success of XML is certainly that it is supported by large companies, such as Microsoft and Sun. XML became so popular that most big commercial RDBMSs now offer special features for XML storage, e.g. see [42].

Over time XML has become a focus of different interests, commercial as well as academic. Its success has certainly made it a hype and a frequent buzzword of the IT industry, which can be a dangerous thing. Analogously to the "UML fever" [19] mentioned in Sect. 2.4, "XML fever" can also be a deadly disease. Michael H. Kay puts it like this [124]: "When a simple technology like XML becomes widely adopted, lots of people jump on the bandwagon, and decide to use it as a vehicle for their extensive technological ambitions". In terms of its structure XML is not new, but basically a reinvention of the hierarchical data model, which was replaced by the RDM.

XML is commonly used for data representation in contemporary CASE technologies,

therefore it must be considered as one of the options for the storage of model-based CASE data. In the following, a comparison is made, in particular with relational database technology. Section 2.5.1 discusses some of the differences between XML and the relational data model (RDM). Section 2.5.2 describes shortcomings of XML.

2.5.1 XML and the Relational Data Model

The XML and the relational data models are logically equivalent. For each XML schema a logically equivalent relational schema can be created and vice versa. However, many XML-relational mappings are possible. A focus of research has been the storage and retrieval of XML data on the basis of relational database technology, e.g. [91, 194, 127, 6].

The simplest approaches store XML data by creating edge tables that contain all elements and attributes of XML documents and link them with foreign keys. When processing XML queries, which are based on path expressions, such approaches require many joins along the path denoted in such a query. This is not performance efficient, therefore more advanced approaches exist that combine larger parts of an XML schema into single tables. Such tables use several columns to store multiple edges in a single row, an approach known as *inlining*, so that less joins are necessary. But it is usually not entirely possible to avoid the "shredding" of XML data into many tables and rows, and the resulting necessity for joins. However, by using specialized indexes queries based on path expressions can be evaluated very efficiently on RDBMSs [219], and most big commercial RDBMSs provide special support for XML data. In the following sections I discuss more of the differences between XML and relational data.

XML versus Relational Data Representation in a DBMS

In general, using the physical structure of an XML document for storage and retrieval of that document has advantages and disadvantages. If the structure of a document is left as is, storage is fast because no restructuring is needed. The queries on XML data follow the tree structure of the document, so having the document stored in its original structure offers performance benefits for query processing as well. Data that are accessed together because they are closely related in the tree structure are likely to be stored on the same memory pages, resulting in a good locality and less pages to be read.

However, if the stored documents are modified, having to preserve the document's original structure in memory may not be possible without a loss of performance. If, for example, a big subtree is inserted into the tree of the document, the memory structure has to be reorganized in most cases to make space for the insertion. The mismatch between the linear structure of conventional computer memory and the tree structure of XML results potentially in memory collisions and fragmentation. In contrast to this, the

tables of a RDB are inherently linear and unordered so that memory management is more flexible and such problems can be avoided, possibly at the cost of a reduced locality. Only the technique of clustering, which is sometimes used in RDBMSs, results in hierarchical physical structures similar to those of XML, having advantages and disadvantages similar to the aforementioned ones. If an XML document makes use of element references, such references have to be resolved during query processing, similar to relational joins. In such cases having the XML tree structure in memory does not provide performance benefits. DBMSs that use XML as the basis of internal data representation may be suitable for information storage and retrieval with path expressions, but not for data that is updated frequently.

Data Integration

XML is frequently introduced as the de facto standard for exchanging and integrating data on the Web. It is important to understand what makes XML suitable for data integration and how it is actually related to the Web. First of all, one has to note that, like any multi-purpose data model, XML is inherently only capable of syntactic data integration. By storing data in XML, we can read and access its structure using a generic XML parser. However, this does not mean that we understand the meaning of the data and can relate it to other data in a meaningful way. Syntactic integration is a prerequisite for, but does not include, semantic integration. As described, for example, in [63], semantic integration can be achieved by translating between different concrete data models. XML comes with accompanying technologies such as the Extensible Stylesheet Language (XSL) that can be used to define transformations between different XML schemas or from an XML schema to a different representation. However, this is not a unique property of XML. There are many other methods for data transformation.

Self-Descriptiveness

XML is often described as being self-descriptive. It is important to be aware of the degree to which XML documents describe the data they contain and how this differs from other data models. XML is a textual format, and thus it can be read with any text editor. It is therefore often referred to as human-readable, but this alone only contributes to accessibility but not to its self-descriptiveness. The fact that data is human-readable does not imply that it is human-understandable. XML data can be typed, which means that an XML document can optionally specify a schema that describes its structure. Apart from the possibility of a well-specified structure, self-descriptiveness of XML documents comes down to the fact that XML data elements have textual labels which serve as their names. If these names are chosen well then they may betray the semantics of the respective data

elements to a human reader. In general, a machine operates only on a syntactic level. It may infer a schema for an XML document, if it is not already given.

Is XML more self-descriptive than other data models? I want to make a comparison with the relational data model as it is used in SQL. First of all, relational data always has a schema, and a relational schema is well-defined, similar to an XML schema. All tables and columns of such a schema have names, similar to the element and attribute names of an XML document. SQL is a textual language and can be used for data as well as for metadata. It incorporates a data manipulation language (DML) for data modification and retrieval and a data definition language (DDL) for the specification of data types and constraints. With SQL one can thus incorporate a relational schema as well as data in a single document. Relation and attribute names would be just as much part of such a document as element and attribute names in XML.

Usability

XML emerged in the late 90's, which was the time of the big bubble of web-based e-commerce. In spite of the damage the web-hype has done to the reputation of Internet technology, the World Wide Web has established itself firmly as the primary medium of the digital world. HTML has gained huge popularity, and its simple textual markup structure has fostered a lot of end-user development. This trend can be seen as an important historical factor for the development and proliferation of XML. HTML is a user interface description language and therefore tailored to the representation of data. XML is a general-purpose data model because it does not imply any particular semantics. Users are free to, or rather compelled to, specify the semantics on their own. This is what is meant by XML's separation of content and layout.

When comparing XML and textual relational notations of data it becomes clear that the nested, hierarchical way used in markup languages is often easier to handle for humans than the relational graph representations consisting of sets of edges. The reason is that the hierarchical representation groups together related data elements so that logical association is reflected in the physical locality of the representation. In contrast to this, the elements in a set of edges do not possess such a hierarchical structure but exist usually in a plainly linear representation. An example for this are SQL insert statements, which are logically equivalent to sets of tuples.

Many data such as web pages and other textual documents have a mostly hierarchical structure, so that the hierarchical markup representation is adequate. However, if data are not mostly tree-structured but have, for example, merely the structure of a directed acyclic graph, then the advantage of such a notation diminishes and tool support becomes more important. Hierarchical textual representation becomes less and less suitable with the degree to which the data deviates from a purely tree-like topology because the degree

to which associated data can be grouped locally decreases.

In XML we have to make use of node references using node identities in order to specify associations that do not fit into a tree, or specify data redundantly. Unfortunately there is a disparity between the identity of XML data by element identity and the identity by data value [131]. Element identity is representation dependent because data can be represented redundantly, i.e. the same data can be represented in several duplicate XML elements. Value identity is purely on a logical level, and the only choice in SQL.

One has to note that with proper tool support, e.g. a structural editor, the question about the underlying physical representation of data is irrelevant. The user only sees the data through the user interface of the tool, so what matters is the representation of the data in the user interface. A good tool offers several representations, so that the user can choose if, e.g., the data are represented as a tree or in a graph-like visualization. On this level the physical representation may at most affect non-functional requirements. XML is known to have shortcomings with regard to performance, space consumption and security [217].

A property that certainly encourages the use of XML is the possibility to define ad hoc structures very easily. The syntax of XML is relatively simple and data can be stored without previously specifying its schema. While this may be an advantage if the aim is to store and retrieve small amounts of data relatively easily, it can become a serious problem on a larger scale. As I know from my work in the e-commerce industry, many organizations use XML data without having a proper schema definition, and in the long run this leads to confusion and additional cost. Furthermore, myriads of different competing XML-based standards have emerged over the years in nearly all domains, with only few of them being widely accepted. In the domain of e-commerce, for example, there are many incompatible XML-based "standards" [63], and even more in-house standards used by individual companies. XML has not solved the interoperability problem caused by the presence of too many competing standards, but rather lifted it onto the level of different XML schemata and semantics.

2.5.2 Problems of XML

There are cases where XML distinguishes between different representations of data although they are logically equivalent. This significantly drags down the level of abstraction and complicates data integration and the language as a tool itself. For example, XML introduces the notion of attributes, although they are equivalent to leaf elements, i.e. elements without further subelements. It means that schema developers frequently have to make irrelevant decisions whether a data value should be stored as attribute or leaf element, but it is exactly those arbitrary decisions that can render schemas incompatible. Another distinction is that between subelements and references: XML distinguishes be-

tween nested elements and references to equivalent elements that are defined somewhere else. In general, distinctions in the representation that are not founded in the logical structure of data and are not made transparent to users are an unnecessary source of incompatibility. In the following sections I briefly discuss some of the shortcomings of XML.

Ordered Data Elements

In XML elements are always ordered, and this is in many cases an overspecification. Even if the order of elements is not relevant for an application, it has to be maintained by a DBMS, creating an overhead. Hence, the DBMS loses flexibility when managing the data: either it has to use the order in memory or maintain additional ordinal data. Furthermore, having order on data when it is not part of a system's specification violates a rigorous information hiding principle: a user may exploit document order to create behavior that does not conform to the specification. RDBMS do not have to care about the order of records and can therefore manage storage space more easily. Order is only sometimes needed for real data, and if so, often naturally given by ordinal attributes, e.g. a timestamp. If order of data is important, it can be modeled in the relational schema, see for example [209].

Mingling of Roles and Types

Most data models make a distinction between the type of a data instance and the role in which that data instance is used. This is important because instances of the same type can be used in completely different roles which must not be confused. For example, a data instance of type Person could be related to another Person instance as its child, parent, employer etc., i.e. with different roles having very different semantics.

Roles are supported explicitly by some data models. However, whether roles are supported or not, they are an intrinsic semantic concept of most data and therefore must be expressed somehow. Roles are a natural concept of data because data elements can relate to one another in many different ways. A role describes the meaning of a data element in the context of a particular relation/association.

Types and roles are orthogonal concepts: while a type defines the internal structure of a data element and its meaning, roles define its relations to other types. If we only had roles, we would lose the ability to identify and reuse many structural equivalences between data elements, i.e. those between different roles referring to the same type. If we only had types, we would have difficulties distinguishing the aforementioned differences in the way elements are related to each other.

In XML, the relation of elements and attributes to types and roles is unclear, and

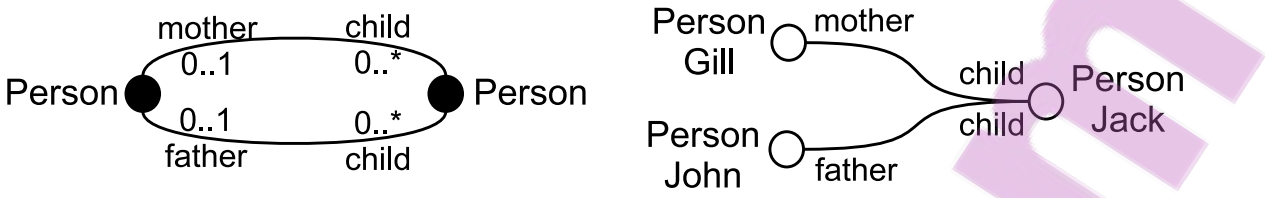

Figure 2.10: A simple PD model (left) and one of its data instances (right).

both elements and attributes are used for types as well as roles without consensus. This causes heterogeneity and incompatibility, and thus impedes data integration. In order to illustrate this, let us consider the PD model in Fig. 2.10, where types and roles are clearly distinguished.

The left side of Fig. 2.10 shows a concrete PDM data model describing two recursive relations between instances of a type Person: the mother-child and the father-child relations. The only entity type in this model, Person, is represented twice in order to disentangle the recursive relation types and thus make the diagram clearer. The following listing shows a simplistic XML representation of the model instance on the right side of Fig. 2.10. Note that it is assumed the XML examples are well-typed, i.e. that they conform to an appropriate XML schema. Instead of providing schema definitions, which are usually hard to read, I prefer to illustrate XML types by means of prototypical XML data examples.

```
<person> <name>Jack</name>
   <person> <name>Gill</name> </person>
   <person> <name>John</name> </person>
</person>
```

This example shows how the Person element is used to signify a type, i.e. for making clear that Jack, Gill and John are persons. Since the person elements are nested, they are somehow related, but we do not know how. In order to specify this we need to encode role information, for example such as in the following:

```
<person> <name>Jack</name>
   <mother> <person> <name>Gill</name> </person> </mother>
   <father> <person> <name>John</name> </person> </father>
</person>
```

As we see, the two nested Person elements each have a different semantic relation to its surrounding element. This could not have been expressed without encoding role information. However, it might not always be obvious which element (or attribute) denotes a role and which one a type. Elements describing roles and elements describing types are

formally indistinguishable, although roles and types are fundamentally different concepts that deserve a formal differentiation. This unclarity is a shortcoming of XML as a data model.

The Root Problem

XML not only requires data to have a mostly hierarchical structure, but also requires it to have a single natural hierarchy in order to be suitable as data representation. The standard ways to query elements in XML, XPath and XQuery, rely on path expressions that identify an element in the tree by the path from the root. Path expressions as a way of addressing elements can have a severe impact on the way XML data is stored, e.g. see [127]. Unfortunately, for many data models the choice of the root is rather arbitrary, i.e. many different roots would be possible. Let us, for example, look at a variant of the previous XML snippets:

```
<person> <name>Gill</name>
   <child> <person> <name>Jack</name>
      <father> <person> <name>John</name> </person> </father>
   </person> </child>
</person>
```

This variant looks very different from the ones above, but contains the same information. child is simply the role on the other end of the mother-child association that can exist between two Person instances. And we could produce many more such variants for this, the same data. The problem of having to choose a root element may recursively reappear for several subtrees of an XML document. This makes the representation of data in XML ambiguous and the identification of data elements using path expressions very much dependent on the particular representation that is chosen for a data model. This can be a hindrance to data integration, since we have to be unnecessarily strict and enforce the use of one particular, arbitrary representation. The relational data model does not have this anomaly.

2.6 Conclusion

The choice of the right data model can have a significant impact on a system, e.g. on its performance and usability, therefore I had to weigh the different alternatives carefully. Eventually I chose the PDM because of its very simple and formal structure and its unobtrusive pattern approach when it comes to advanced features. Another very important point is that the PDM can be mapped losslessly onto the RDM. No DBMS supports the PDM directly, therefore I chose to base the repository subsystem on RDBMS technology.

I did not choose the OODM because I do not want to prescribe an OO software development approach. As pointed out in Sect. 2.2, OODBMSs are not generally better than RDBMSs, as is sometimes suggested. What makes OODBMSs modern is the fact that most contemporary software projects make use of OOP. OODBMS technology certainly makes sense in the context of the OO paradigm, although it is not a necessity due to good object-relational mapping tools. However, the OO paradigm is not necessarily the best one for all situations.

Most better-known contemporary CASE modeling tools are in fact UML tools, therefore the data model proposed by UML had to be considered as well. As I pointed out in Sect. 2.4, UML is strongly tied to OOP, therefore it does not provide the flexibility required for programming language neutral CASE support. Furthermore, UML has become increasingly complex and lacks a formal basis.

Considering the different arguments of Sect. 2.5 it became clear that XML does not match the requirements of the CASE model repository either. XML is not a guarantee for successful data integration but just a tool, and like any other tool just as good as its user. There are many XML-based standards, but many such "standards" sprout up like mushrooms, which can be a source of instability. XML mingles issues of representation into the main high-level purpose of a data model, i.e. storage, manipulation and retrieval of data on a logical level. XML does not achieve the same level of abstraction as the RDM.

3

The Repository

This chapter describes the repository, which stores all processed artifacts. It is the central part of the abstract platform since whenever a data model is accessed, this is done through the repository. In contrast to a simple file system, the repository is a typed data store, and has a number of additional properties that are important for the industrial environment in which big software engineering projects take place.

First of all, I consider general requirements in Sect. 3.1. Section 3.2 provides an overview of the repository. I decided to implement the PD model in a relational database because of the similarity of the PD model to the relational data model and the maturity of today's RDBMS technology. More about this decision can be found in Sect. 3.3. There are two basic possibilities for implementing the PD model: with a static relational schema, which is described in Sect. 3.4, and with a dynamic relational schema, which is described in Sect. 3.5.

Section 3.6 describes the notion of operations used by the repository. Section 3.7 shows how reflection is performed: existing models can be introspected and read as PDM instances, and PDM instances of the PDM metamodel can be manifested in the repository through intercession. Section 3.8 discusses issues of data interchange, and Section 3.9 describes the repository client library, which offers advanced functionality such as a local read cache and a mechanism for event notification. Section 3.10 discusses some related work. Parts of this chapter have been published in [142].

3.1 Requirements

Before describing the design and implementation of AP1's repository, I would like to clarify the main requirements that I chose for this component. The following list comprises functional as well as non-functional requirements. In the following, I will discuss each requirement and highlight why it is important. Afterwards, I will explain why it was chosen for AP1's repository in particular.

Types Generally, I want the repository to contain typed data because this establishes structure, making it easier to find and access particular data and to prevent errors. Structural invariants such as types are very important for making a system easier to understand, develop and maintain because they constrain the space of possible states.

Expressiveness The data model used in the repository must be expressive enough to define arbitrary finite models and model data. CASE data such as source code can be quite complex, and contains many recursive data structures and some context-sensitive features. The repository must be able to handle such data.

Well-definedness The repository must not allow syntactic ambiguities, i.e. information that can not be understood clearly on a syntactic level. Natural language, for example, is usually full of syntactic ambiguities such as association ambiguities. Furthermore, there should not be two structurally different data sets representing the same syntactic content. This is a sign of redundancy of features in a data model, and introduces arbitrary variance into data sets. Note that I am not concerned with semantic ambiguities, which depend on the way the repository is used. They cannot be prevented by the repository itself, but users can do so by defining clear meanings for all the data models.

Ergonomy The repository should be easy to use for people with data modeling experience and not introduce concepts that go against the general knowledge established in the data modeling community. Thus, established concepts should be reused as much as possible. The repository should also try to mitigate any kind of impedance mismatch that might occur when using it with other technologies.

Remote Access The repository should be accessible over the network. The repository contains all the valuable accumulated work of a project, and thus must not be lost under any circumstances. Therefore, a repository is usually kept on a server in a controlled physical environment, e.g. a dedicated server room. However, work on the project is usually done from a number of different places, such as different offices or even from home. Only through the use of the Internet as a medium between

developers and repository can a system accommodate such a modern, distributed working environment.

Concurrency The repository should be able to handle multiple users concurrently. This is important because software development projects are rarely done by just a single person. Most software projects are performed by teams, which can greatly vary in size. The importance of collaborative work activities such as pair programming or collective planning is widely recognized by modern software development processes, and therefore needs appropriate CASE support. Problems such as lost updates can occur when concurrent work is not properly orchestrated.

Change Control The repository must be able to control all the changes of the data it contains. This means that changes are logged and that they can be undone, redone and merged with other changes. Ideally, it should be possible to combine change information with other information, e.g. information about defects. The field concerned with change control in software development is known as software configuration management (SCM).

Security It should be possible to control and restrict access to the repository. This is particularly important in a commercial setting, where data can be very important for the competitive edge of a company. It must not be possible for unauthorized people to access sensitive data. Another aspect of security is the prevention of accidental changes. A common policy is that of minimal access, which means that each person has only those access rights they actually need to do their job. This ensures, for example, that people do not accidently interfere with areas of work they do not know much about.

Performance Operations on the repository should have a reasonable performance. There should not be operations that cause extreme performance slowdown. The more frequent an operation is, the faster it has to be. Many software development tasks that are supported by CASE technology are interactive, e.g. graphical data modeling. Therefore, responsiveness is very important. Developers should not need to wait for the system when entering data because this would reduce productivity.

Availability & Reliability Conceived for use in a production environment, I want the repository to be available at all times. If developers cannot use the repository at any time, potential work is lost. Unavailability of critical CASE tools can lead to schedule and cost overruns. The repository, as the heart of the CASE infrastructure, has to be a stable system. Furthermore, it should be fail-safe, i.e. able to recover after a failure. Every system might fail at some point in time, and fail-safety at

least means that the damage that can be caused by a typical failure is known and limited. After a failure, the system is brought into a safe, consistent state.

Transactionality Operations on the data should satisfy the ACID principle [114]. While contributing to the concurrency as well as the reliability requirements, transactionality is of such a high importance in many production environments that I list it as a separate requirement. While transactions are common in the database world, they are not very common for CASE technology, although they would be equally useful.

Types, expressiveness, well-definedness and ergonomy were chosen because the repository is designed for *model-based* software engineering, and these requirements are important when dealing with model data in general. Remote access and concurrency were chosen because one of the aims of this project was to support collaborative work. Change control and transactionality are important with regard to software configuration management, which is necessary in larger software projects. Security, performance, availability & reliability are generally important for industrial environments, such as the software developing organizations targeted by AP1.

3.2 Overview

Figure 3.1 gives an architectural overview of AP1's repository. The main component is the repository database, which runs on a RDBMS. The repository database contains tables for all the data, i.e. all the different artifacts. It also contains tables for the metadata, i.e. the data describing all types.

The database also stores other objects in order to realize the functionality required for the repository. It contains user-defined functions and stored procedures that simplify and optimize access to all the data. Stored procedures are more efficient than dynamically generated queries because they can be precompiled by the RDBMS. With stored procedures it s possible to provide a convenient API for the database with a higher level of abstraction. Furthermore, the database contains functionality for change control, which makes use of triggers and sequence generators. This is described in more detail in Chapt. 4.

The database can be accessed over a network. Commands such as queries, insertions and updates are sent to the database using the SQL language. In order to provide a higher level of abstraction and mitigate the impedance mismatch between OOP and the RDM, I provide a client library that can be used to access the repository in an object-oriented manner.

The repository client library offers an object-oriented API to the repository. It performs a mapping from the relational data to PDM data, and vice versa. As a result, data

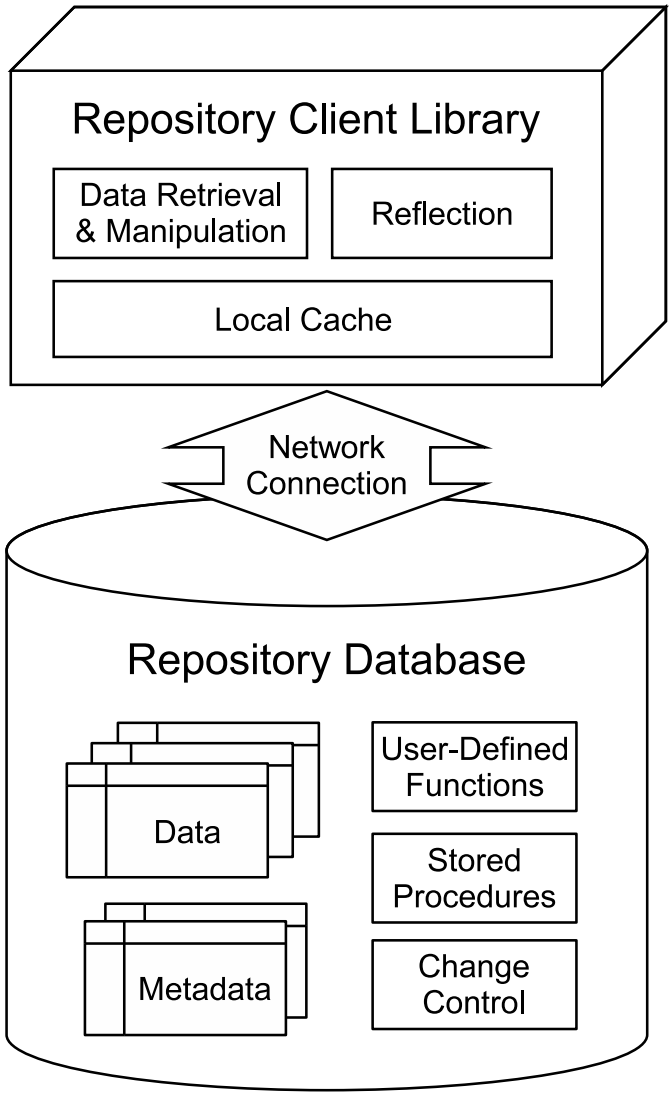

Figure 3.1: Overview of the repository.

retrieval and manipulation can be performed using the more abstract PDM. Furthermore, the library offers functionality for reflection. Existing data types can be introspected, i.e. data describing them in terms of the PD metamodel can be queried. New data types can be created and existing ones manipulated using intercession.

In order to improve performance, the repository client library implements a local read cache. When navigating between entity instances of a PD model, the repository database is only accessed if the requested data is not already in the cache. This makes read access on a local working set very efficient. The local cache is automatically synchronized with the repository database.

Figure 3.2 shows how the repository can be used. CASE tools can either connect directly to the repository database using the SQL standard, or use the more abstract repository client library. Many CASE tools and client libraries can use the repository database concurrently, since the RDBMS implements mechanisms for transactional con-

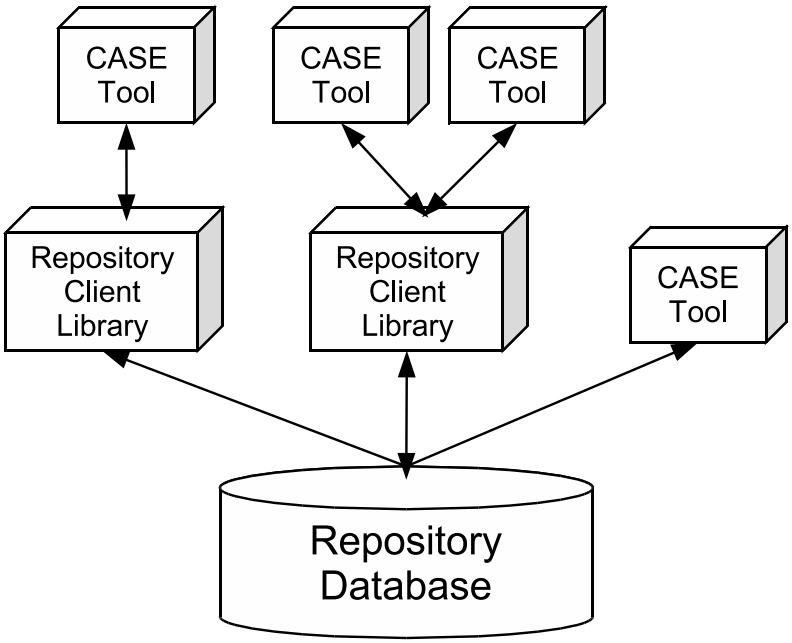

Figure 3.2: Usage of the repository.

currency control. Several tools can use the same instance of the repository client library, thus sharing the local cache and potentially making repository access more efficient. The repository client library implements its own mechanism for concurrency control, making sure that the tools do not interfere with each other.

3.3 Using a RDBMS

Several approaches have been proposed for the implementation of a shared repository for CASE tool integration. Some systems, e.g. the one described in [100], use relational DB technology. Other ones, e.g. PCTE [7], use object-oriented databases. And many recent approaches favor the use of XML database systems, e.g. as presented in [148]. All these approaches have advantages and disadvantages, and I have discussed many of them in Chapt. 2.

AP1 uses a relational database management system (RDBMS) for the repository. As described in [100], this has the following advantages: RDBMSs are a very mature technology that is widely used and highly reliable. They rely on the simple and theoretically sound relational data model (RDM). RDBMSs offer a well-understood, well-known and standardized interface through the SQL language [116], which allows powerful operations to be specified at a high level of abstraction.

A RDBMS is able to check and enforce various constraints on a DB, and its functionality can usually be extended in various ways, e.g. by triggers, stored procedures and user-defined functions. Furthermore, there exist a plethora of applications supporting the

creation, use and maintenance of a RDB, such as DB administration tools, backup and replication tools, application generators, and interfaces to various programming languages.

Most RDBMSs inherently support networking, so that a repository built on this technology can be accessed remotely. This is commonly done using industry standards such as Open Database Connectivity (ODBC), which is based on the international standard ISO/IEC 9075-3:2003 [117]. Furthermore, modern RDBMSs offer rich features for efficient, highly concurrent transaction processing and security, e.g. encryption and access control. All these advantages can be leveraged by AP1.

However, using a RDBMS also has disadvantages, and it is a challenge for AP1 to overcome these. Disadvantages are described in [100, 125] and include an impedance mismatch between the RDM and the more complex data structures required for CASE, possible performance penalties when these data structures are accessed, and a lack of built-in advanced features such as version management. I will address these shortcomings and describe how they can be solved.

Regarding performance, one has to keep in mind that RDBMSs were designed for and are widely used for commercial transaction processing applications with huge throughputs. Therefore, a lot of work has been invested into RDBMS performance optimizations. One of the approaches to a potential performance problem is, thus, the usage of a modern RDBMS that has been developed with performance-intensive applications in mind.

AP1 by default uses the Firebird RDBMS [211], which has a modern, multi-threaded architecture. It uses multiversion concurrency control [22]. This means that the RDBMS uses different versions of a database in order to prevent readers and writers blocking each other, while maintaining the integrity of the data. Furthermore, many features of the SQL standard are supported. It would be possible, though, to use a different, equally feature-rich and SQL-compliant RDBMS instead.

Wolfgang Keller [125] notes that one of the major problems for using RDBMSs for CASE are long transactions. Many CASE tools use a check-in / check-out workflow in which data is worked on by developers over a longer period of time. This causes problems with the concurrency control mechanisms of common RDBMS, which are usually designed to handle transactions that commit after a short period of time. The data accessed by a long transaction needs to be protected from other, concurrent transactions in order to ensure integrity. A transaction accessing the same data either has to wait for the long transaction to finish, or needs to be rolled back.

The AP1 repository circumvents the problem of long transactions by encapsulating single operations into their own transaction respectively, see Sect. 3.6. Larger units of changes can still be safely managed and undone due to the change control mechanism described in Chapt. 4. This mechanism also makes it possible to use an asynchronous check-in / check-out workflow without the need for long transactions.

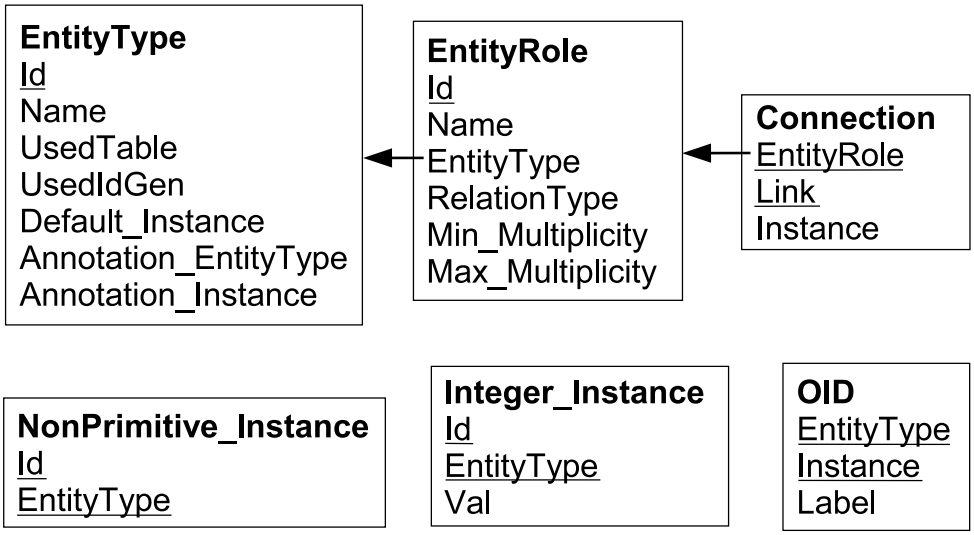

Figure 3.3: Static relational schema for the PDM.

3.4 Mapping the PDM onto a Static Relational Schema

In the following I will discuss how the PDM can be mapped onto a relational schema. The first option I discuss uses a static relational schema. In the static schema implementation, the relational schema is the same for every PD model. That is, the same tables are used to store data of different relation and entity types. This is similar to simple edge-based approaches for XML-relational mapping [91].

Figure 3.3 shows the relational schema that I used for this implementation. It contains a table for entity types, one for roles, one for connections of instances to links, and several tables to store the instances of different entity types. This implementation supports PD models with n-ary relation types. In the following I will discuss those tables one by one:

EntityType As the name indicates, this table is used to store information about all available entity types. Each entity type is represented by a row and identified by an artificial primary key `Id`. Column `Name` contains a string and is optionally used to represent the identity of an entity type in a more human-readable form.

Since this table lists all entity types, with data for different types potentially having a different structure, I use column `UsedTable` to store the name of the table in which data of the respective entity type is stored; this table must have a structure that is suitable for that entity type. The following columns, `UsedIdGen`, `DefaultInstance`, `Annotation_EntityType` and `Annotation_Instance` are not essential for the static schema approach; they are examples for additional data that one might want to associate with an entity type.

Column `UsedIdGen` contains the name of a surrogate value generator, which is used to create new identities for instances of the respective entity type. `Default_Instance` is the identity of one of the instances of that entity type and can be used as a default setting for insertion operations; `Annotation_EntityType` and `Annotation_Instance` form a pointer to an instance of a usually different entity type, which contains additional metadata for an entity type. This makes it possible to annotate entity types with arbitrary metadata, since entity types can be linked to arbitrary PDM data with these columns. The semantics of such additional metadata annotations can vary between different entity types and are up to the modeler.

EntityRole The `EntityRole` table defines relation types. Actually, each row in this table represents a role and not a relation type. This is necessary since relation types – although they are usually binary – can have arbitrary arity in this approach, and thus an arbitrary number of roles. Roles have an artificial key `Id` and an optional human-readable name. Column `EntityType` denotes the entity type this role is connected to, and column `RelationType` is an artificial identifier for the relation type this role belongs to. Roles that belong to the same relation type have the same `RelationType` value. For binary relation types, multiplicity constraints can optionally be given in columns `Min_Multiplicity` and `Max_Multiplicity`.

Connection Table `Connection` defines the links by which entity instances are connected. Analogous to table `EntityRole`, a row in `Connection` is actually not a complete link but just information about one end of a link that is connected to a particular entity instance. Column `Instance` refers to the entity instance that is connected, and `EntityRole` denotes the role which is used to connect it. The entity type of the connected instance can be inferred from the corresponding row in the `EntityRole` table.

Since there is a row in table `Connection` for every instance connected to a link, I need column `Link` in order to identify all the connections that a link comprises. Like `RelationType` in table `EntityRole`, column `Link` is an artificial identifier. One of the invariants that must hold for a valid dataset is that the rows in table `EntityRole` that correspond to the rows in table `Connection` having identical `Link` values must have identical `RelationType` values, i.e.

$$\forall r_1, r_2 \in \text{EntityRole}: \forall c_1, c_2 \in \text{Connection}:$$
$$(c_1.\text{EntityRole} = r_1.\text{Id} \wedge c_2.\text{EntityRole} = r_2.\text{Id}$$
$$\wedge c_1.\text{Link} = c_2.\text{Link}) \rightarrow r_1.\text{RelationType} = r_2.\text{RelationType}.$$

The primary key of table `Connection` comprises the columns `EntityRole` and `Link`.

This is because only these two values together identify the connection of an instance to a link unambiguously. The same instance can connect with the same role to different links. The same instance can connect to the same link with different entity roles. But only one entity instance can be connected to a link with a particular role.

NonPrimitive_Instance In order to store data efficiently, I use different tables to store the instances of different entity types. Non-primitive instances are represented by rows of table `NonPrimitive_Instance`, and consist only of artificial primary key values. The primary key comprises the `EntityType` as well as an `Id`, so that the instances of all the non-primitive types can be stored in that table without any risk of key collisions.

Integer_Instance The columns of table `Integer_Instance` are the same as those of table `NonPrimitive_Instance` plus column `Val` for the actual value of the primitive instances stored in this table. That is, primitive values also have an artificial primary key, so that all instances – primitive or not – can be referenced uniformly. Analogous to table `NonPrimitive_Instance`, this table contains all instances of primitive entity types that store their instances as integer values. Consequently, column `Val` is of an integer type.

This table is only an example of the way instances of primitive types are represented. The static relational schema contains analogous tables for other kinds of primitive entity types. There is a table X`_Instance` for every primitive entity type X with specific storage requirements. These tables are structured like `Integer_Instance`, but each time column `Val` has a different type. Which table is used for the storage of the instances of a particular entity type can be seen in column `UsedTable` of table `EntityType`, which contains the table name.

OID This table is not essential for the static relational schema, and illustrates a possible extension that allows the definition of opaque identities (OIDs). The idea is that every instance can be given a textual label, so that users can recognize instances more easily than with the artificial keys. Columns `EntityType` and `Instance` identify an instance, and `Label` contains a human-readable instance name.

An advantage of the static relational schema for PDM data is its compactness with regard to the number of relational tables, and its homogeneous structure. It consists of only a few tables, and these tables can contain an arbitrary number of entity types, relation types, instances and links. The data in the tables are very regular. They consist of large numbers of similar rows, which mostly take up very little space each.

However, the static relational schema has significant disadvantages. There is a strong impedance mismatch between the PDM and the static relational schema. Because data

is essentially represented in a unified, untyped manner, such as in table `Connection`, the type system of the RDBMS is poorly used. The types of the PDM do not correspond to types in the RDM, but only to data tuples. The features that the RDM offers to structure data are only used to describe a PD metamodel, but not to describe the different PDM types.

The RDBMS cannot be adequately used to check constraints of individual PD models, since all models are thrown into the same tables. As a result, constraints such as multiplicities have to be enforced by the database applications rather than the database. Some inherent integrity constraints such as the correspondence between instances, `Connection`s and `EntityRole`s are not trivial.

In order to traverse links, quite a number of join operations have to be performed: from instances over `EntityRole`s and `Connection`s to other `EntityRole`s and instances. This affects the performance of the system negatively. Because of this impedance mismatch, the power of SQL can only be leveraged with difficulty. Too many joins make it very hard to formulate, maintain and understand complex queries. The joins are often recursive, e.g. between the roles of a relation type or the connections of a link.

3.5 Mapping the PDM onto a Dynamic Relational Schema

In the dynamic schema implementation, the correspondence between PDM types and relations is much closer than in the static mapping. Types of PD models are mapped onto tables, as we would usually define them on a RDBMS. As a result, the number of relational tables in the dynamic schema increases monotonically with the size of the PD model that it reflects.

I propose a mapping similar to the compact mapping suggested in [76]. The mapping is illustrated in Figs. 3.4, 3.5 and 3.6. The basic idea is to map each relation type to a relational table, and to merge tables where it is possible. This is similar to schema-aware approaches for XML-relational mapping [194, 6], which use relational schemas that are tailored to the types of the data that should be stored. Such approaches produce compact schemas by inlining the representations of certain data elements, such as attributes, into other related ones.

In the following sections I will discuss the mapping of relation types with different multiplicities to relational tables. The focus is on binary relation types between two non-primitive entity types, or a non-primitive entity type and a primitive entity type, since these are by far the most common ones. I will not restrict myself to the plain RDM, but also discuss common constraints which today's RDBMSs offer. Such constraints can be

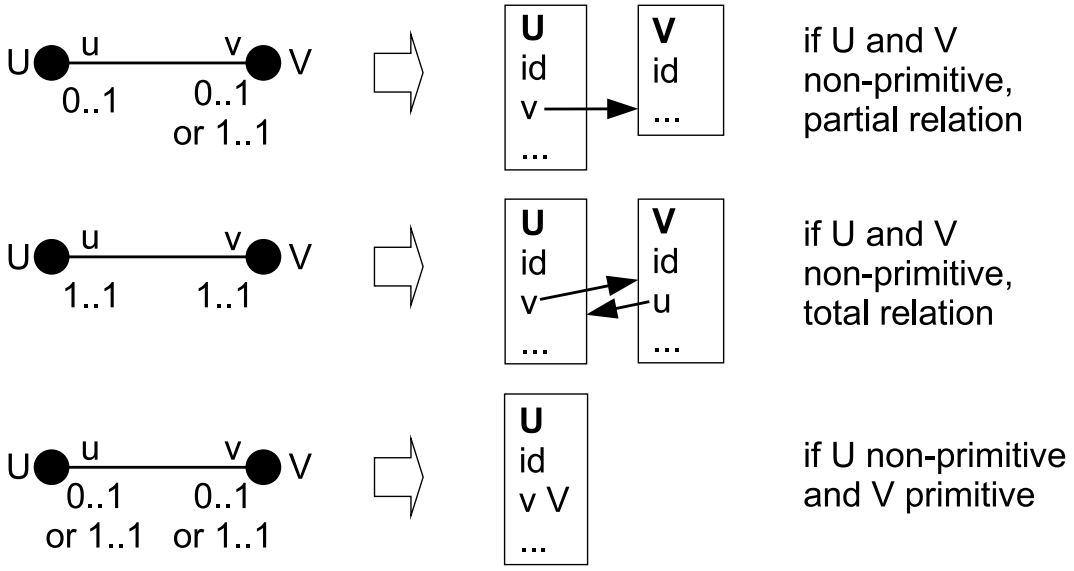

Figure 3.4: Mapping of one-to-one PDM relation types onto the RDM.

used to express multiplicities of relation types more accurately.

Generally, each non-primitive entity type is mapped to a table with an artificial primary key. Primitive entity types do not have their own tables. However, some relation types require additional tables. In Figs. 3.4, 3.5 and 3.6 these additional tables, of which at most one is necessary for each relation type, are called R.

Tables representing non-primitive entity types are called U or V. Since tables for non-primitive types potentially accumulate columns from different relation types connected to them, an ellipsis in the illustrations of those tables indicates that more columns may actually be added. Note that I will frequently consider $0..x$ as well as $1..x$ multiplicities together (with x being either 1 or *) because the difference between them is often only expressed in additional constraints, but not in the corresponding tables illustrated in the figures.

3.5.1 One-to-One Relations

Figure 3.4 illustrates the mappings from relation types with one-to-one multiplicities onto relational tables. I will go through the different cases of one-to-one relation types, which differ in the kind of the connected entity types.

U and V non-primitive, partial relation In this case, a U instance is associated with at most one V instance, and vice versa. However, not all instances of U and V may have such a link, therefore we speak of a partial relation. This is illustrated in the top part of Fig. 3.4.

Both U and V have their own tables with artificial primary keys. The foreign key in U referencing a row in V makes sure that at most one V is associated with each

U. In the case that there is a 1..1 multiplicity on role v, it is an advantage to have a foreign key in U referencing a V instead of the other way around: every U has a V, so the space for the foreign key in U will never be wasted. A foreign key in V would potentially have null values and thus waste memory. I suggest adding a foreign key constraint to table U so that the RDBMS checks referential integrity, making sure column v in U does indeed reference a row in V:

```
CONSTRAINT U_v_FK FOREIGN KEY (v) REFERENCES V(id)
```

In order to make sure that each V has at most one U associated with it, one should use a unique constraint on column v of table U. This constraint ensures that each row in V is referenced at most once, meaning that it has at most one row in U associated with it. It is expressed by the following SQL snippet:

```
CONSTRAINT U_v_Unique UNIQUE (v)
```

If each U has exactly one associated V and not just at most one, I require the value in column v of table U to be set. This is done by adding a `NOT NULL` constraint on column v.

U and V non-primitive, total relation In this case, there exists a strict one-to-one relation between U and V: each U instance is associated with exactly one V instance, and vice versa. All instances of U and V have such a link, therefore we speak of a total relation. This is slightly more difficult than the previous case. A possible mapping is shown in the middle part of Fig. 3.4.

I use the foreign key, `NOT NULL` and unique constraints for column v of table U, as well as equivalent constraints for column u of table V. The foreign key and `NOT NULL` constraints make sure that there is a total relation, i.e. that each U is associated with a V, and each V with a U. The unique constraints make sure that no U and no V is referred to more than once.

Furthermore, I would need to make sure that the foreign key references between U and V are symmetric. That is, I want to ensure that the rows of U and V form proper pairs. This property can be expressed as

$$\forall u \in \mathrm{U}\colon \exists v \in \mathrm{V}\colon u.\mathrm{v} = v.\mathrm{id} \wedge v.\mathrm{u} = u.\mathrm{id}.$$

Alternatively, one could implement the relation type like a partial relation, as described before. This would solve the symmetry issue, but potentially allow a partial

relation. I would need to make sure that all rows of V are referenced by a row in U. This property can be expressed as

$$\forall v \in \mathrm{V}\colon \exists u \in \mathrm{U}\colon u.\mathrm{v} = v.\mathrm{id}.$$

Such constraints are not supported by all RDBMSs. They would be implemented as SQL `CHECK` constraints, or alternatively by triggers before `UPDATE`, `INSERT` and `DELETE`. They can lead to a significant performance slowdown, therefore one might decide not to enforce these constraints through the RDBMS at all.

U non-primitive, V primitive This case is illustrated in the bottom part of Fig. 3.4. Since U is a non-primitive type, it has its own table with an artificial primary key. The relation to primitive type V is expressed by adding a field v of primitive type V to U. This is the way tables for non-primitive types accumulate their primitive fields.

If each U has not just at most one but exactly one associated V, then I add a `NOT NULL` constraint to column v. The fact that each V has at most one U is expressed with a unique constraint on column v of table U. The case that each V has not just at most but exactly one associated U is improbable because the range of primitive types is usually very wide, and this would mean that all possible values must be present in the table. This unusual case is expressed by the following property:

$$\forall val \in \mathrm{V}\colon \exists u \in \mathrm{U}\colon u.\mathrm{v} = val.$$

3.5.2 One-to-Many Relations

Figure 3.5 illustrates the mapping from one-to-many relations of the PDM to relational tables. The relational tables for $0..x$ multiplicities are the same as the ones for $1..x$ multiplicities (with $x \in \{1, *\}$), just the constraints necessary for these two cases differ. However, the constraints here are basically the same as those that are used in the previous section on the mapping of one-to-one relations.

U and V non-primitive This case is illustrated in the top part of Fig. 3.5. Because U and V are non-primitive entity types, each of them has a separate table with an artificial primary key. Each V is associated with at most one U, therefore V gets a foreign key column u that references one of the rows in U. This way, many rows in V can reference the same row in U, assuring that there can be arbitrarily many V rows associated with one U. A foreign key constraint can be used to ensure referential integrity.

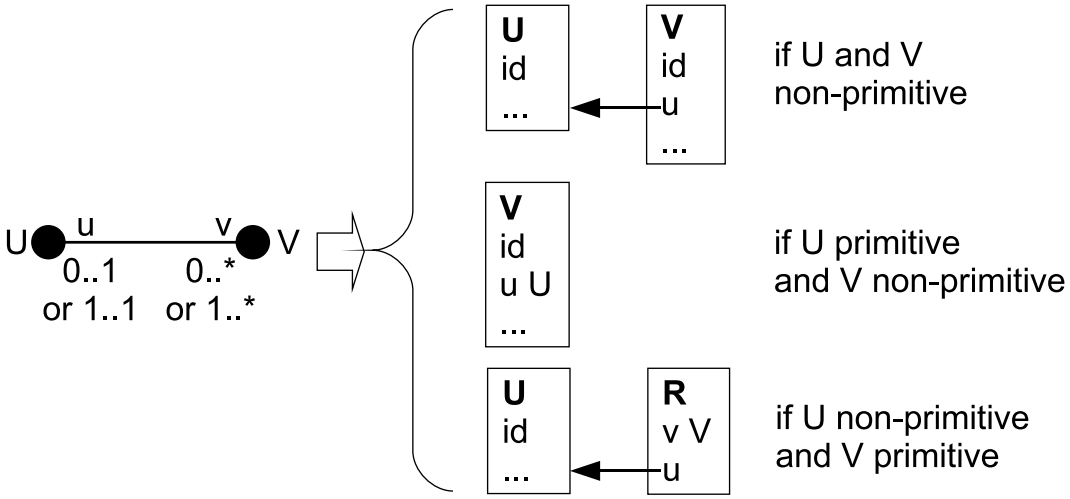

Figure 3.5: Mapping of one-to-many PDM relation types to the RDM.

If there should be exactly one U associated with every V, then a `NOT NULL` constraint on column u is added to table V. If there should be at least one V associated with every U, then this can be enforced with a `CHECK` constraint or triggers, by preserving the property

$$\forall u \in \mathrm{U}\colon \exists v \in \mathrm{V}\colon v.\mathrm{u} = u.\mathrm{id}.$$

U primitive, V non-primitive This case is illustrated in the middle part of Fig. 3.5. Because V is non-primitive, it has its own table with an artificial primary key. The U values are stored in a column u of that table. Every V instance corresponds to a row in table V, so it is associated with at most one U. There can be multiple Vs with the same value in column u, consequently a U can be associated with multiple Vs.

If there should be exactly one U per V, then a `NOT NULL` constraint is added to column u. The case that there should be at least one V per U is very uncommon because of the usually wide range of primitive types, and the fact that there had to be a row in V for every possible value of U. One would have to enforce the following property:

$$\forall u \in \mathrm{U}\colon \exists v \in \mathrm{V}\colon v.\mathrm{u} = u.$$

U non-primitive, V primitive This case is illustrated in the bottom part of Fig. 3.5. There is a table for non-primitive type U, and an additional table R for the mapping of this particular relation. R contains a column v for a primitive value of type V and a column u that is a foreign key to a row in table U. A foreign key constraint on column u can be used to ensure referential integrity.

A unique constraint on column v of table R makes sure that there is at most one

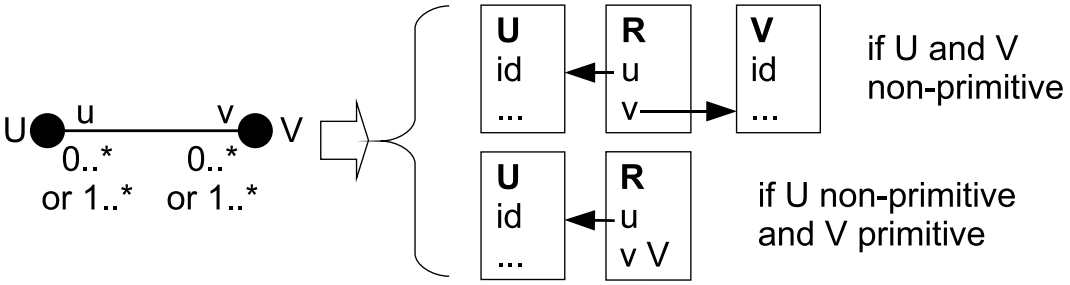

Figure 3.6: Mapping of many-to-many relations PD to the relational data model.

row for each V value in R, and consequently at most one U associated with every V. Because multiple rows in R can reference the same row in U, a U can be associated with multiple Vs. Furthermore, `NOT NULL` constraints can be added to the columns u and v of table R. Then, insertion and deletion of links in the PDM correspond to `INSERT` and `DELETE` operations on rows of R. Updates on R are unnecessary.

If there should be exactly one U associated to each V, which is a rare case for primitive entity types because they usually have too many values, one would make sure that all the possible values of V are present in table R, i.e. that

$$\forall v \in \mathrm{V}\colon \exists r \in \mathrm{R}\colon r.\mathrm{v} = v.$$

If there should be at least one V for every U, one would have to make sure that

$$\forall u \in \mathrm{U}\colon \exists r \in \mathrm{R}\colon r.\mathrm{u} = u.\mathrm{id}.$$

3.5.3 Many-to-Many Relations

Figure 3.6 shows the mapping between many-to-many relation types and relational tables. Again, I discuss the different cases that occur here, and treat the cases where there should be at least one associated element and the cases where there may also be none together.

U and V non-primitive This case is illustrated in the top part of Fig. 3.6. The two non-primitive entity types have their own tables U and V with artificial primary keys. A many-to-many relation between them is modeled by connecting the two tables U and V with an additional table R. R has two foreign key columns u and v, referencing rows in table U and V, respectively, and thus enables arbitrary associations between Us and Vs.

Referential integrity can be ensured with foreign key constraints, and `NOT NULL` constraints on u and v make sure that data about associations are not incomplete. A unique constraint on columns u and v together ensures that there are no duplicate

associations. If at least one V should be associated with each U, then one needs to make sure that

$$\forall u \in \mathrm{U}\colon \exists r \in \mathrm{R}\colon r.\mathrm{u} = u.\mathrm{id}.$$

Analogously, if at least one U should be associated with each V, one needs to make sure that

$$\forall v \in \mathrm{V}\colon \exists r \in \mathrm{R}\colon r.\mathrm{v} = v.\mathrm{id}.$$

U non-primitive, V primitive This case is illustrated in the bottom part of Fig. 3.6. It is very similar to the mapping of one-to-many relations between non-primitive and primitive entity types. There is a table with artificial primary key for entity type U, and an additional table R for the relation, which contains a foreign key column u referencing a row in U and a column v of primitive type V.

One should add a foreign key constraint on column u of table R to ensure referential integrity. `NOT NULL` constraints on columns u and v avoid incomplete associations between Us and Vs. A unique constraint on columns u and v together is used to avoid duplicate associations. There can be several rows in R that refer to the same U but contain different V values, thus a U can be associated with multiple Vs. There can also be rows in R with duplicate V values but references to different Us, so that one V can be associated with multiple Us.

If there should be at least one V per U, then the following property must hold:

$$\forall u \in \mathrm{U}\colon \exists r \in \mathrm{R}\colon r.\mathrm{u} = u.\mathrm{id}.$$

If we want at least one U per V, which is quite unusual, then we need a check constraint ensuring that table R contains all possible V values, i.e.

$$\forall v \in \mathrm{V}\colon \exists r \in \mathrm{R}\colon r.\mathrm{v} = v.$$

3.5.4 N-ary Relations and Relations between Primitive Types

N-ary relations are not very common, but sometimes they occur. However, it is always possible to transform an n-ary relation into a new entity type R and n binary relations, as illustrated in the left part of Fig. 3.7. Each binary relation links an R to exactly one U_i ($i \in \{1, \ldots, n\}$), so that the elements in R are representing the n-tuples that would occur in the corresponding n-ary relation. This transformation is known as reification.

The mapping of n-ary relations to relational tables can be seen in the right part of Fig. 3.7. Note that there is no straightforward notion of multiplicity constraints for n-ary relations, as with binary relations, therefore I will not consider multiplicities here. An

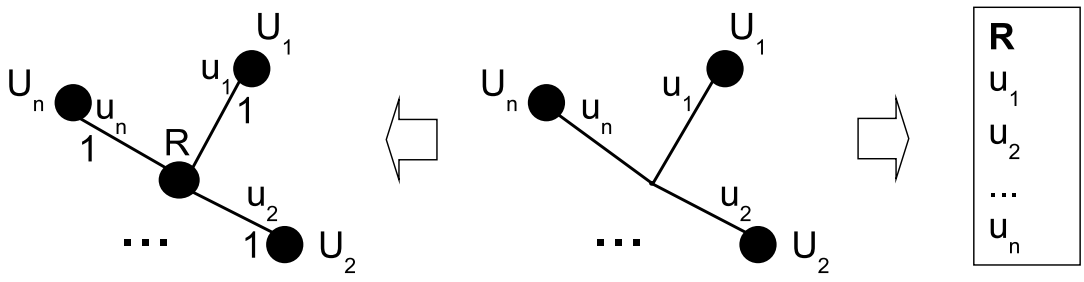

Figure 3.7: Reification of n-ary relations into a new non-primitive entity type and binary relations, with mapping to the RDM.

n-ary relation is mapped to a single table R. If U_i is a primitive type, then column u_i of R has type U_i. If U_i is a non-primitive type, then column u_i is a foreign key referencing a row in table U_i, the table representing the non-primitive type.

Foreign key constraints can be used to ensure referential integrity. To avoid duplicate associations, I add a single unique constraint on all the columns $u_1, \ldots, u_n$. `NOT NULL` constraints on the columns $u_1, \ldots, u_n$ can be used in order to prevent incomplete associations, i.e. links between less than n instances.

Like n-ary relation types, direct relations between primitive types occur rarely. And if they occur, they can be reified in just the same manner. In this case, $u_1, \ldots, u_n$ are all primitive types. Such relation types may be binary, i.e. n=2. This is just a special case of the general reification transformation described above.

3.5.5 Inheritance

Two different ways to implement inheritance were evaluated. One approach uses a common artificial key for all related instances. The other approach is similar to the implementation of ordinary binary relation types, and gives all related instances their own artificial keys. Related instances, or relatives for short, are those that are connected with each other by the inheritance relation. The concept of attribute-oriented inheritance, which is used in the PDM, has been described in Sect. 2.3.4.

For this discussion, I will use the example model illustrated in Fig. 3.8. In this model, an entity type D inherits from two entity types B and C, which in turn both inherit from an entity type A. This example of multiple inheritance, where the inheritance hierarchy forms a collapsed tree, is suitable for pointing out all the differences between the two implementation approaches.

Shared Keys

When implementing inheritance with shared keys, all related instances share the same artificial key. This is illustrated in Fig. 3.9, which shows a model instance of the PD

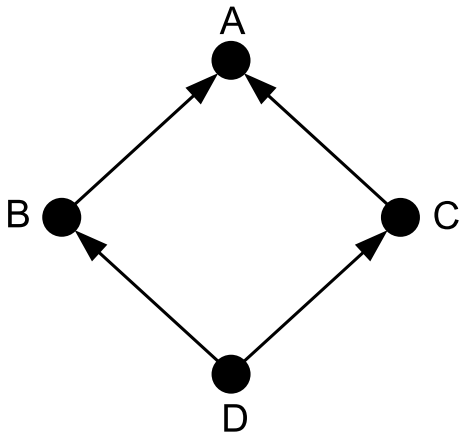

Figure 3.8: Example PD model using attribute-oriented inheritance.

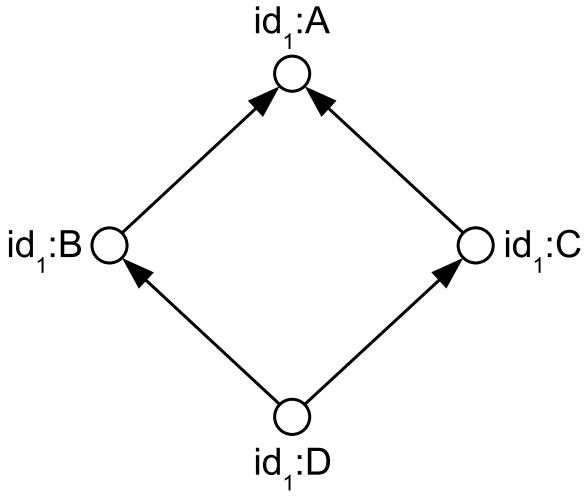

Figure 3.9: Example of inheritance with key sharing among relatives.

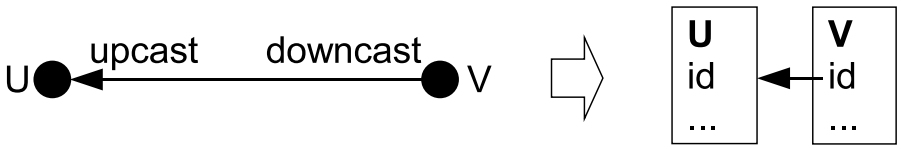

Figure 3.10: Mapping attribute-oriented inheritance onto the RDM with shared keys.

model in Fig. 3.8. The labels at the instances consist of the instance's key, a colon and its entity type.

All the instances in Fig. 3.9 are related, and therefore have the same key. The only way to distinguish between the different related instances is by their type. This is used in the relational implementation of attribute-oriented inheritance with shared keys, which is illustrated in Fig. 3.10.

The idea is to use the fact that related instances have the same keys to perform joins between them. Let us consider entity types U and V, with U being a supertype of V,

which are represented in tables of the same names. The column `id` of table V is not only used as primary key, but also as foreign key to a row in table U. Because of the inheritance relation, each V instance must have a related U instance. Therefore each V row must refer to a row in U. This is implemented by a foreign key constraint on column `id` of table V.

This has the advantage that no additional memory is needed for the implementation of inheritance. Inheritance relation types can be navigated by joining primary keys. Furthermore, no additional indexes are required because primary keys already have indexes by default. As a result, this implementation approach is very resource efficient.

However, shared keys affect the way features can be represented that are inherited more than once, such as the feature represented by entity type A. A is inherited from both B and C, and therefore the instances of B and C are connected with an A each. D inherits from both B and C, therefore each D inherits feature A twice.

As already mentioned in Sect. 2.3.4, features that are inherited more than once can either be represented separately or only once and be shared by all respective relatives. Some programming languages that support MI, e.g. C++, let the developer choose which representation should be used. There are situations where a separate representation is more appropriate, and others where a shared representation would be the preferred one.

One of the problems of the implementation of inheritance with shared keys is that only one of the feature representations is supported: features that are inherited more than once are always represented only once, and thus shared among all relatives. This is because each group of relatives can only contain at most one instance of every entity type. Since all related instances share the same key, two related instances of the same type could not be distinguished and are therefore infeasible.

Another disadvantage is that in order to identify an instance, two pieces of information are needed: the key of the instance and its type. This is because the same keys are used in different related types. If each instance had a unique key, a key value was sufficient for identification, but this is in conflict with the idea to implement inheritance with shared keys.

Finally, the implementation of inheritance with shared keys introduces a dependency between related instances: they have to have the same key. This can be a major disadvantage for the implementation of algorithms that create new related instances. Such an algorithm cannot create related instances independently from one another, but has to communicate the key of a new superinstance to all its new subinstances.

Separate Keys

When implementing attribute-oriented inheritance with separate keys, each instance has its own unique key, regardless of any inheritance relations. In contrast to the aforemen-

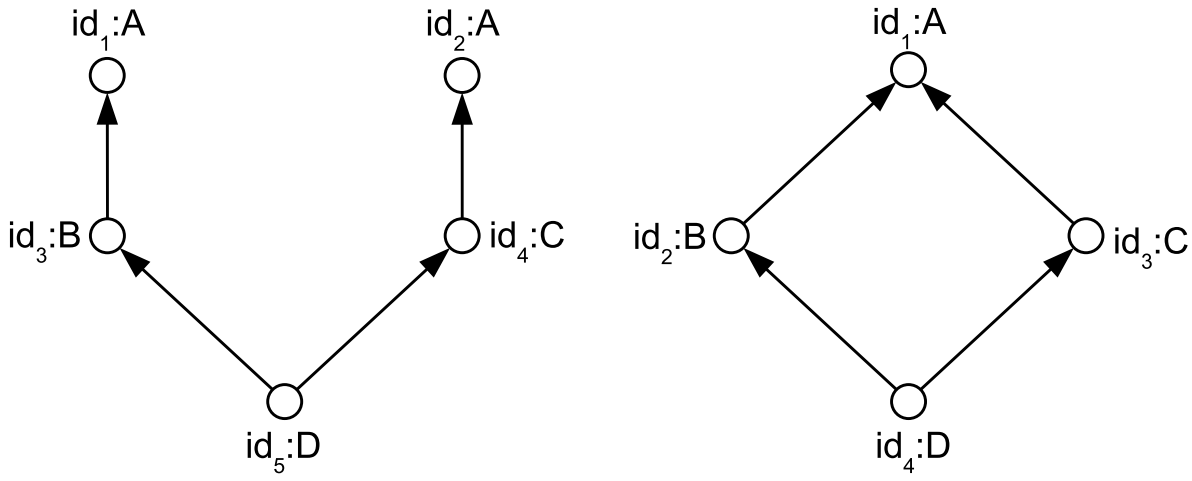

Figure 3.11: Examples of inheritance with separate key for each instance: shared (left) and separate (right) representation of common features.

tioned shared key approach, a key identifies an instance unambiguously. The information needed for the inheritance relation is not inherently part of the implementation of an entity type, but must be added explicitly. As a result, the inheritance relation types are implemented just like ordinary one-to-one relation types, according to the previously depicted mapping rules.

The implementation of inheritance with separate keys allows for both the shared as well as the separate representation of features that are inherited more than once. This is illustrated in Fig. 3.11. The left part of the figure shows a PD model instance of the model in Fig. 3.8 which represents the two inherited A features separately. The right part shows a model instance in which there is only one representation for feature A, i.e. the representation is shared. Note that all involved relations are one-to-one, despite the fan-out of two from instance id_1. The two outgoing links of instance id_1 belong to two different relation types.

The separate keys implementation approach fosters simplicity and uniformity, while offering a high degree of flexibility. The mapping of inheritance relation types is simple because it is the same as the mapping of ordinary one-to-one relation types. Only one uniform set of mapping rules is necessary. That is, even if an application does not know about the concept of inheritance, it can make use of inheritance relations. All the different notions of multiple-inheritance are supported.

3.5.6 Example

Figure 3.12 illustrates an example mapping between a PD model and a relational schema. Customers are associated with addresses and orders. Addresses consist of a street name and a house number. The example contains many-to-many, one-to-many and one-to-one

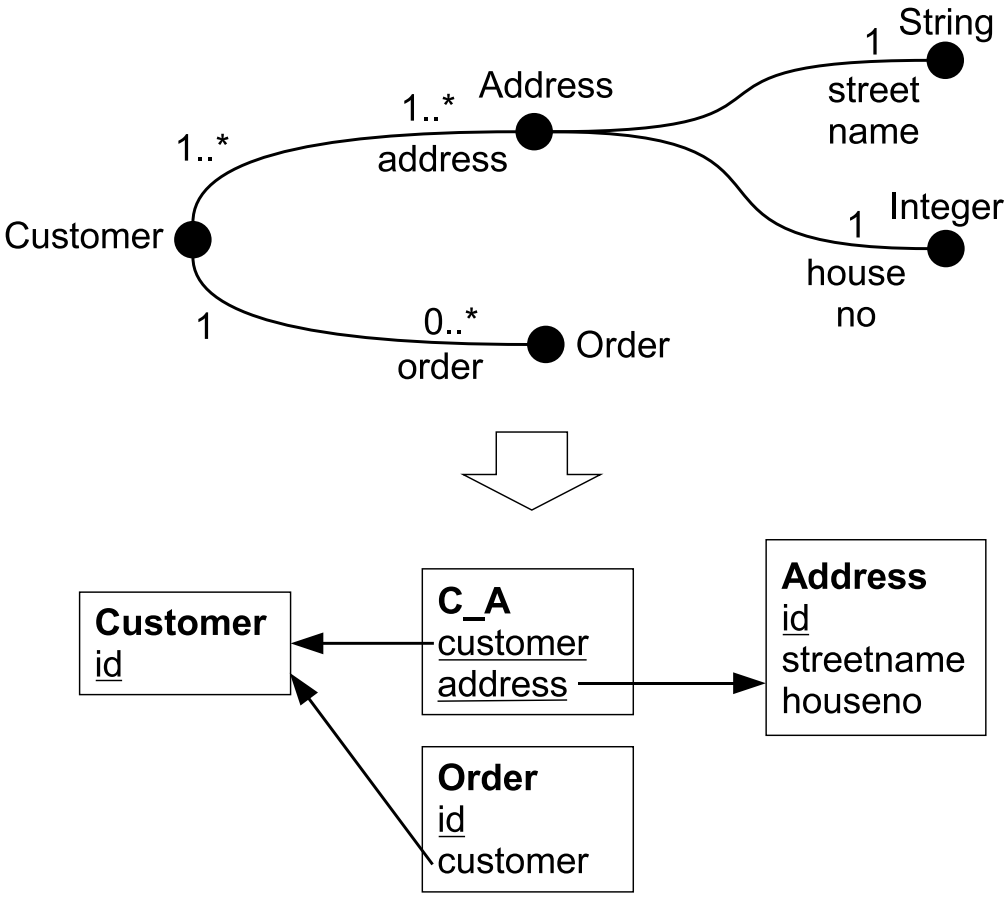

Figure 3.12: Example mapping between a PD model and a relational schema.

relation types. Primitive as well as non-primitive entity types are used.

3.5.7 Conclusion

I implemented both the static and the dynamic schema in the way outlined in the previous sections. Both implementations were tested and examined with regard to their time and space efficiency, and their ease of use. Testing consisted of storing large test data sets, and accessing those data sets using different navigational patterns.

As already discussed, the dynamic schema has several theoretical advantages over the static one, and this was confirmed during my tests. The static schema is really slower and more cumbersome to use. It stores each link in a separate database record instead of grouping them, and thus eventually takes up significantly more memory. One point of uncertainty in the dynamic schema that became clear during the tests was how efficiently new tables can be added to a database. In particular, space efficiency could have been an issue with large numbers of tables if each table had taken up a lot of space. In the Firebird RDBMS, which was used for this project, a new table increases the size of a database by about 10kb to 20kb for a moderate number of typical columns (10-20, no fixed-length

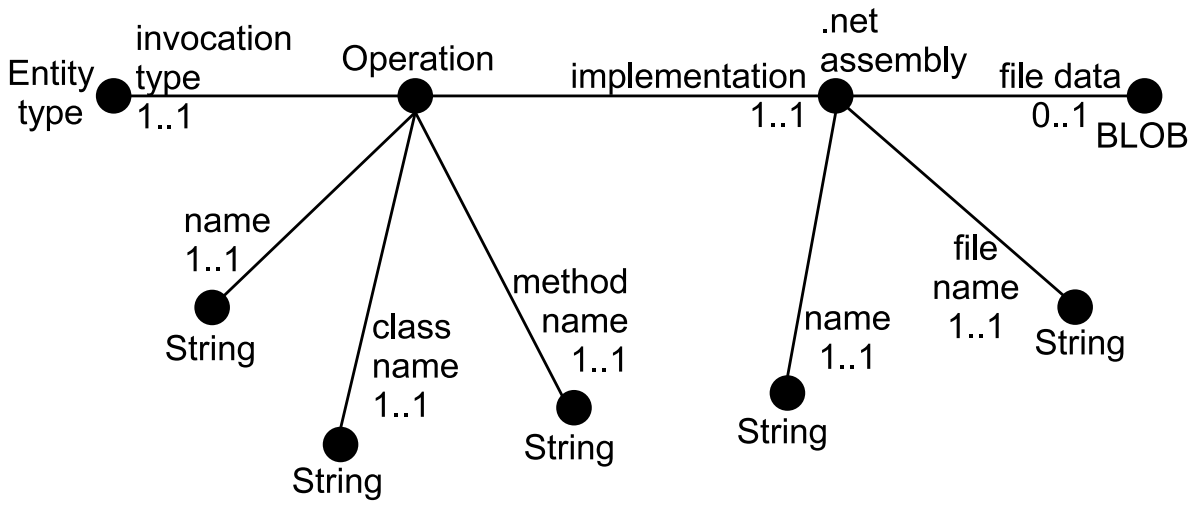

Figure 3.13: PD model of operations in the repository.

arrays). The initial size depends mostly on the complexity of the primary key, since the primary key automatically gets an index for faster row access. Adding rows does not cause much additional storage overhead. Consequently, adding tables dynamically turned out not to be a problem. Considering its advantages, the dynamic schema was chosen as the default for the repository.

Furthermore, I implemented and tested both implementation approaches for inheritance. The tests showed that the shared key approach was slightly more memory efficient. However, its aforementioned dependency problem turned out to be a crucial drawback, as it made algorithms that create new related instances a lot more difficult. This became apparent during the development of a generation algorithm similar to the one described in Chapt. 8, which was used during testing. Therefore, the repository implements inheritance using the separate keys approach.

3.6 Operations

The repository supports a special notion of operations on data. Operations are programs that take a single instance as parameter and are executed in a single database transaction. Consequently, each operation is defined on a single entity type, which is called the operation's *superparameter* type. All available operations are specified in the repository.

Figure 3.13 shows the PD model for operations. Each operation is represented by an instance of non-primitive entity type "Operation". An operation instance is associated with its superparameter type. Furthermore, each operation has a human-readable name.

In the current system, operations are implemented on the .net platform. Therefore, all systems are supported for which an implementation of the .net common language runtime (CLR) exists. Operations are static methods which are compiled into .net assemblies.

Those .net assemblies can themselves be stored in the repository.

Each operation instance refers to an implementing .net assembly, and the names of the class and the static method in that class which implements the operation. Assemblies are represented as instances as well. Each assembly has a human-readable name and a file name for the assembly file. Furthermore, it is possible to store the whole assembly file in the repository.

The advantage of storing the assembly as a binary large object (BLOB) in the database instead of just referring to a file is that all repository clients can immediately use them. All they have to do is load the BLOB and feed it into the appropriate .net assembly instantiation functions. Then, the class and method name can be used with the reflection features of .net in order to access the respective method.

While developing an assembly, it is easier to keep it in the file system for development purposes. This is because most .net development tools, e.g. the popular Visual Studio IDE, store their artifacts only in the file system. During development the assembly is likely to be changed and recompiled frequently, as well as accessed by other tools such as debuggers. In this case, operations are invoked by loading their assembly from the file system.

3.7 Reflection

When talking about reflection on the PDM, I am talking about *structural reflection* [62] that comprises *introspection* and *intercession*. Structural introspection means that it is possible to dynamically read metadata about data types and code. Structural intercession means that it is possible to create new data types and code dynamically. In this chapter I am concerned with structural reflection of data types only.

Reflection is generally a powerful feature, but with the power come new possible risks. It means that programs can modify their own internals at runtime. This may make a system vulnerable to accidental corruption with unpredictable consequences. It may also allow the violation of certain information hiding and access control principles. Therefore, reflection features require a particularly careful design.

3.7.1 Introspection

Introspection means that it is possible for a program to query information describing itself. The information read during introspection is called metadata, and the types of this metadata are called metatypes. Instances of meta entity types are called metainstances, and links of meta relation types are called metalinks. This section considers how PDM information can be made accessible to developers and applications.

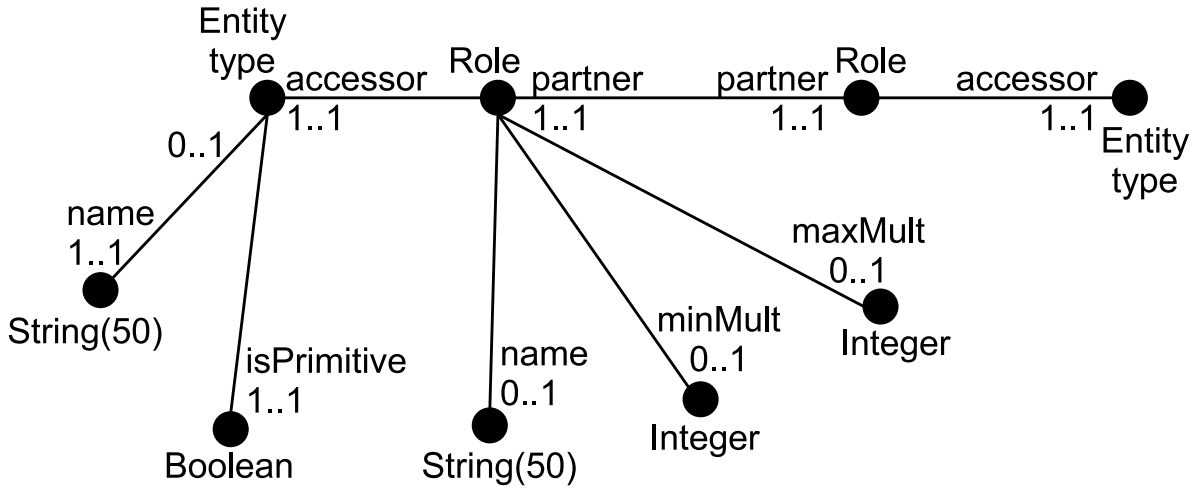

Figure 3.14: Detailed illustration of the PD metamodel.

When discussing introspection of data structures, one is interested in metadata about types, i.e. in this case metadata about PD models. The idea is to use a metamodel, which is a PD model that describes the constituents of all well-formed PD models. The metamodel can be used in order to express metadata about PD models in a well-typed manner.

The process of representing internal entities such as data types is – like the transformation of n-ary relation types – known as *reification*. The reification operation takes a part of a data model as a parameter and returns metadata that represent that particular part. Metadata can be handled much like ordinary data, and can thus be processed by ordinary programs.

The PD metamodel used in the AP1 system is illustrated in Fig. 3.14. There is an entity type "Entity type", which represents entity types, and an entity type "Role" to represent roles. An entity type can have an arbitrary number of roles, which is indicated by the fact that there is no multiplicity constraint at that end of the relation type. Each role is associated with exactly one entity type, which is its accessor.

Note that in a relation type, the accessor of a role is the entity type opposite to the owner of that role. While the owner of the role is the entity type that is accessed through the role, the accessor is the entity type from which the role can actually be used. It is more important for navigation to efficiently get the roles that can be accessed by an entity type. This is facilitated by associating roles with their accessors instead of their owners.

Because of the usage of binary relation types, the PD metamodel is symmetrical. Each role has exactly one "partner", i.e. the role at the other end of the respective relation type. The partner role has its own accessor. Roles and entity types have a number of primitive attributes, i.e. associations to single values of primitive entity types. For clarity, these attributes are only represented once, i.e. for only one of the two occurrences of the types

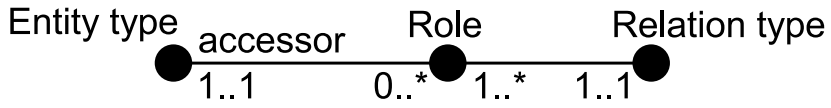

Figure 3.15: The PD metamodel for n-ary relation types.

"Entity type" and "Role".

Entity types have a textual name "name" and a boolean predicate "isPrimitive" indicating whether they are primitive types. For roles a textual name is optional. Furthermore, roles have optional multiplicities "minMult" and "maxMult". If a multiplicity value is $\perp$ – represented as `NULL` in the database – this means that there is no minimum or maximum limit, respectively.

For the sake of completeness I also want to point out how the PD metamodel would look for n-ary relation types. This is illustrated in Fig. 3.15. The relation type between entity types and roles stays the same, but there is an additional entity type "Relation type" for relation types. This is because arbitrarily many roles can be associated with an n-ary relation type. Type "Relation type" groups all roles belonging to the same relation type.

Because of type "Relation type", the n-ary metamodel has one more level of indirection. This makes navigation more complicated and also slower, since more joins are required to connect the roles of a relation type. Considering how rarely n-ary relation types are actually used, and how easily n-ary relation types can be reified, it seems natural to base the repository on the binary PD metamodel.

Introspection in AP1 is based on the fact that for all entity types and entity roles there is appropriate metadata stored in the repository. As I will describe in Sect. 3.9.3, this metadata is also used in order to access the data in the repository's RDBMS through the PDM. The repository ensures the consistency of metadata through access control. It is not permitted to delete or modify metadata so that it would not match the actual data types they describe any more.

As a consequence, there is no special functionality required for introspection. The metadata is accessible much like all other data in the repository. All a program needs to do in order to perform introspection is to find the metadata it needs to work with. This can be done in a navigational way using the repository client library, or in a declarative way using SQL.

3.7.2 Intercession

Structural intercession means that programs are able to modify themselves. In this section, I look at the way programs can modify the PDM types that are defined in the

repository. The common way intercession is supported is through the modification of metadata, i.e. by changing internal entities through corresponding changes on the metadata that describe them. AP1 also takes this metadata manipulation approach.

In program code, dynamic intercession can easily cause execution errors. Imagine, for example, code that accesses particular data fields. If these fields are deleted during runtime by means of intercession, the code that tries to access them will fail to work. There are many examples of how new potential execution errors or more subtle semantic errors can be introduced into a program if the usage of reflection is not safe and deliberate. An example for a programming language with full dynamic reflection is MetaJ [65].

A common approach for intercession based on metadata is to perform changes on internal entities as soon as their corresponding metadata is changed. This is called *implicit intercession* because changes are propagated immediately from the world of data into the world of internal program entities just by changing metadata using normal data operations. In contrast to that, a system might propagate changes on metadata only when instructed to explicitly, e.g. when an operation is invoked. This latter approach is called *explicit intercession*. With implicit intercession on PDM data types, each modification of metadata would be immediately reflected back into a corresponding modification of the underlying PD model. However, this approach can cause problems if it is not implemented carefully. When data types are dynamically modified, the integrity of their data may be violated. Careless use of immediate reflection, e.g. the deletion of an important attribute, can immediately lead to data loss and data corruption.

AP1 uses explicit intercession for the creation of new data types and the modification of existing ones. This means that modifications of metaobjects have no immediate effect on the underlying data model. Intercession is not something that occurs automatically but an operation, as described in Sect. 3.6, that has to be invoked explicitly. Intercession is performed according to the mapping rules in Sect. 3.5. The superparameter of the standard intercession operation is an instance of entity type “PD model”. This entity type is connected by a many-to-many relation type to entity type “Entity type”. That is, an arbitrary number of entity types can be associated with a PD model instance. This makes it possible to use intercession on well-defined sets of entity types and the relation types between them. Primitive entity types do not need to be associated with PD model instances. If a non-primitive entity type is associated with a PD model, then using the intercession operation on the model will automatically include all relation types between it and any primitive entity type.

Each entity and relation type metainstance is associated with a boolean predicate “is materialized” which indicates if the respective type has been manifested in the database already. The intercession operation will ignore all those entity and relation types for which this predicate is true. Once an entity or relation type has been materialized, its metadata

becomes permanent. Permanence of the metadata means that the metalinks cannot be removed any more. This ensures that once a PD model has been materialized, its metadata cannot be changed. It is important for the integrity of the system: if we allowed existing types to be changed, what would happen to the data that might already have accumulated for that type? It might get corrupted or lost. In addition to that, permanence ensures that code that was once written to use a certain type in a correct manner will continue to work. On a RDBMS, permanence of metadata can be implemented with triggers that intercept forbidden modifications on materialized metainstances. The permanence property does not impede extensibility. Adding new entity and relation types to an existing, materialized PD model is non-intrusive and does not put its integrity at risk. At worst, minimum multiplicities of a new relation type greater than zero may cause violations for existing instances, which do not yet have links for the new relation type. However, such violations are not fatal. They are reported and made visible, so that they can be corrected later on.

3.8 Data Interchange

Data interchange is an important issue for CASE tool integration. If there are several repositories, it has to be possible to import and export models and model instances from and to the file system. This is because the file system is still the most common form of data storage for end-users, and frequently used to transport data on portable storage media. Furthermore, it must be possible to exchange data with other programs.

AP1 uses SQL scripts for import and export. In order to support smooth integration of databases and prevent key collisions when integrating data into the repository, instances are identified by universally unique identifiers (UUIDs) [119], also known as globally unique identifiers (GUIDs). These artificial identifiers are sufficiently small (128 bit) and can be efficiently processed and generated on the fly.

The GUIDs for non-primitive instances are generated without any parameters. They are opaque identifiers that are independent of the natural information content of an instance. Because of their uniqueness, non-primitive instances can be generated independently on different systems. When two different data sets are integrated, there will be no collisions between GUIDs of instances that are not meant to be the same. If both data sets contain the same GUID, then one can be certain that they refer to the same instance.

The GUIDs for primitive instances can be generated using the primitive values themselves as parameters. The most general approach, which is also part of the standard for UUIDs, is to use a cryptographic hash. This ensures that each primitive instance has a unique GUID, and also that GUIDs which are independently generated for the same primitive value are the same.

For certain typical primitive types it is possible to make the value directly part of the corresponding GUID. A feasible approach would be to retain the Media Access Control (MAC) address suffix of the GUID, using a particular MAC to identify each primitive type, and to substitute the prefix with the respective primitive value. Examples for primitive types that could be transformed into GUIDs include integers of various sizes, booleans, single- and double-precision floating-point values, characters and small strings. The advantage would be that such GUIDs would not only serve as unique identifiers but also contain – in readable form – the values they refer to themselves.

GUIDs as artificial keys have other advantages. Although they are relatively large for artificial keys, they are smaller than many natural keys, which often consist of more than one attribute. For example, the full name of a person takes up frequently more than 16 characters, and it may only be unique together with other attributes such as the date of birth. Credit card numbers are usually stored as 16 characters, so take up the same amount of memory as GUIDs. The uniformity of GUIDs makes it possible to safely identify values of many different types. With GUIDs it is hence possible to use key columns flexibly in an untyped manner. Such a column can be used to refer to any type that uses a GUID as an artificial key. This is exploited, for example, in the change control mechanism of the repository, which is described in Chapt. 4.

The Firebird RDBMS makes use of a prefix compression technique. In indices, adjacent keys can be stored more efficiently if they share a common prefix. If, for example, a key has a string value ABCDE, and the next key has a value ABCDF, then prefix compression is applied: instead of storing the whole string ABCDF, the RDBMS just stores that the second key shares a prefix of length 4 with the first one, plus a one character postfix F.

Prefix compression is of interest for the storage of GUIDs because the standard method for the generation of GUIDs includes the MAC address of a computer's network interface controller as a suffix. A MAC address is 6 bytes long and identifies, with rare exceptions, a network interface controller uniquely.

In the AP1 system, GUIDs are usually generated by the repository server itself. Therefore it is very likely that many of the GUIDs in a repository share the last 6 bytes. In practice, many GUIDs share even a longer prefix, up to 15 bytes if they were generated on the same machine within a certain interval of time. To leverage prefix compression for the efficient storage of GUIDs, the user-defined functions for GUIDs in the repository store GUIDs in reverse order. As a result, the common postfixes become common prefixes.

SQL data import scripts essentially consist of `INSERT` and `UPDATE` operations. If a non-primitive instance already exists in the database it is imported into, then the respective `INSERT` operation fails. In general, if a command in an SQL script fails during an import operation because it violates a uniqueness constraint, one can be sure that the conflicting element in the database is the same as the one in the script if GUIDs are used to identify

them. Consequently, the command can be safely ignored.

Idempotent insertion can be realized using the `MERGE` statement, which is a relatively new SQL command. Instead of using an `INSERT` statement directly, `MERGE` can be used in order to perform an insertion conditionally. Similarly, it can be used for conditional updates. Let us consider the following `INSERT` statement:

```
1  INSERT INTO PDModel (Id)
2  VALUES (hex_to_guid('2fe6f0944062db11afc0b95b08f50e2f'));
```

In line 2 the user-defined function `hex_to_guid` is used, which converts a GUID in standard hexadecimal representation into the reverse binary representation that is used in the database. If this PD model instance is already stored in the database the command will fail, and the transaction in which the command is executed will abort. This is certainly not what is desired here; I want to ignore commands that cause key collisions without impeding the execution of the others. This can be achieved with the `MERGE` operation:

```
1  MERGE INTO PDModel m
2  USING
3   (SELECT count(id) FROM PDModel
4    WHERE id=hex_to_guid('2fe6f0944062db11afc0b95b08f50e2f')) n
5  ON (n=1)
6  WHEN NOT MATCHED THEN
7    INSERT INTO PDModel (Id)
8    VALUES (hex_to_guid('2fe6f0944062db11afc0b95b08f50e2f'));
```

In line 3 and 4, the number of occurrences of the identifier that is to be inserted in the target table is determined in a subquery. Since identifiers are unique, there is at most one such occurrence. Line 5 checks if there is an occurrence, which would mean that the insertion would cause a conflict. Lines 6-8 perform the insertion in case the identifier does not yet occur.

Note that the SQL scripts used for import and export have a highly regular structure, even more so than XML, so that very high compression of these scripts is possible. Furthermore, the structure of such scripts is not strongly recursive such as XML, which makes them much faster to parse and process.

Even if tools access the repository directly, the mechanisms for notification and change management will still work. As I have already mentioned, the repository also contains the program code of data operations. As a consequence, operations can also be exchanged between tools, which enables better functional integration. Data operations directly manipulate the database, which makes it possible to read their results using just SQL.

3.9 The Repository Client Library

A major problem arising when using a RDBMS for the repository is the impedance mismatch between the RDM and the way data is usually accessed in programming languages, e.g. see [226]. Impedance mismatches are common when working with different data models in parallel, and can cause different impediments [222]. Most contemporary programming languages, such as C# which was mainly used for the AP1 project, follow the OOP paradigm. As a consequence, most CASE tools have to deal with an object-relational impedance mismatch when using the repository database directly.

The data structures used for CASE can be very complex. In particular, they can be highly recursive, e.g. for source code. Storing such data structures in a relational database usually results in fragmentation: the data has to be spread over several normalized tables and usually comprises several rows, which have to be reconnected by joins in order to yield the original data. This problem has been described in the context of XML-relational mappings, e.g. [6]. As discussed in Sect. 2.1, the schema of a database depends heavily on the multiplicities in the represented data structure and can vary significantly with multiplicity constraints; e.g. new tables can be necessary when representing many-to-many relationships. All these factors make direct access of an application to the repository database more difficult.

AP1 tackles these difficulties by providing an object-oriented library on top of the database, the repository client library, which allows access to the data in the repository in a much more convenient manner using the PDM. This library is written in the C# language and can therefore be used on all platforms that offer a common language runtime (CLR) for the .net platform. Applications accessing the repository can either access the RDB directly using SQL, use the library, or both.

There exist proven techniques for making persistent data accessible through OOP [201], e.g. by means of database access layers that perform object-relational mapping automatically [125]. However, this problem is different so that these techniques cannot be directly applied: the repository client library performs a mapping between the RDM of the database and the PDM, offering access to the PDM through an object-oriented API. It does not map the RDM directly to the OODM in the sense that every type of the RDM data gets its corresponding class. But with appropriate syntactic extensions, PDM access can be made to look like that.

Most object-relational mapping techniques implement a mapping only for a fixed number of statically known types, not for all possible types in general. This is because the simplest approach for this mapping is a generative one where a database access layer is statically generated for a set of given types, and because in most real systems the types are known statically. However, because of AP1's reflection capabilities and the extensi-

bility requirement the repository client library must support access to new, dynamically created types as well. To enable this, the API of the library defines classes that reflect all the concepts of the PDM, so that every constituent of PD models and model instances can be accessed by an object-oriented program.

Common relational-mapping techniques cannot be applied directly, but some of their typical patters are naturally useful for managing different aspects of the underlying relational database. SQL code descriptions have to be stored appropriately, attribute mapping methods and type conversions have to be applied. Tables, identities, changes, transactions and connections have to be managed. Such patterns are used by the repository client library, and described, for example, in [226, 125].

The repository client library abstracts from low-level implementation details like tables, foreign keys and joins. It takes care of the mapping between the PDM and the RDM by applying the mapping rules of Sect. 3.5. Furthermore, it offers advanced services such as a local read cache and an event notification mechanism. All this is described in the following sections.

3.9.1 Examples

Let us first have a look at the way the repository can be accessed using the repository client library. The following C# source code example demonstrates some of the basic classes and read operations:

```
1  FbTransaction ta = Server.BeginRead();
2  Instance i = Instance.LoadObject("Type PD model", ta);
3  i = i.DowncastTo(EntityType.Load("Entity type", ta), ta);
4  Role nameRole = Role.Load(i.Type, "name", ta);
5  Console.WriteLine((String)i.GetInstances(nameRole, ta)[0].Value);
6  Console.WriteLine(i.GetString(nameRole, ta));
7  Console.WriteLine(i.GetBool("isPrimitive", ta));
8  EntityType t = (EntityType)i;
9  Console.WriteLine(t.Name);
10 Console.WriteLine(nameRole.GetInt32(Role.Role_MinMult, ta));
11 Server.Commit(ta);
```

Line 1 begins a new read-only transaction. In line 2, an instance of entity type Object is loaded. This entity type has a similar function to class Object in most OOP languages: it serves as the root of the inheritance hierarchy. Class `Instance` represents instances of all entity types. It defines a static method `LoadObject` which can be used in order to load an Object instance which has a particular textual label associated with it. The

instance that is loaded is the Object superinstance of the metainstance representing the entity type "PD model".

Line 3 performs a downcast from the Object instance to its subinstance of metatype "Entity type". The `DowncastTo` method gets an argument of class `EntityType`, which represents entity types, that is metainstances of entity type "Entity type". Class `EntityType` offers a static `Load` method with which entity types can be loaded by their name. Note that all read operations require a reference to the transaction on which they are executed.

Line 4 loads a metainstance of entity type Role. Roles are represented by class `Role`, which has a static method `Load` for loading a role with a particular name that is accessible from a particular entity type. The accessor type used is the type of instance `i`.

Line 5 reads the instance that is connected to instance `i` through role "name", which was loaded in the previous line. The instance, which represents a primitive value, is printed on the console. Method `GetInstances` of class `Instance` is used to navigate from an instance to all those instances that are connected through a certain role. The result is a collection of `Instance` objects.

Since an entity type has exactly one name, the collection contains only one instance in this case – a primitive instance of type String. We get this first and only instance from the collection and read its value from field `Value`. The static type of field `Value` is `Object` because the exact type of the values depends on the entity type of the respective `Instance` object. Line 5 performs an explicit downcast to the actual dynamic type `String`.

Line 6 has the same effect as line 5: the name of the entity type represented in instance `i` is printed on the console. This time, instead of using `GetInstances`, a convenience method that performs the appropriate list access and type cast for us is used. The `GetString` method of class `Instance` reads a single instance of entity type String by navigating the given role.

Line 7 reads another primitive instance that is connected with the "Entity type" metainstance that represents entity type "PD model". This time the Boolean value connected with role "isPrimitive" is read and printed on the console. In contrast to the previous line, a different convenience method is used that accepts a role name instead of a `Role` object, which can make source code a bit shorter.

In line 8 the `Instance` object is downcasted to class `EntityType`. This is valid because the instance is in fact an "Entity type" metainstance, and class `EntityType` is a subclass of class `Instance`. This shows how introspection works, with internal entities such as types being accessible data entities at the same time. The downcast `Instance` object is assigned to a variable `t`.

Line 9 again prints the name of the entity type represented by instance `i`. This time the convenience fields of the downcasted variable `t` are used. Class `EntityType` has fields and getter methods for all the important instances associated with an entity type

metainstance, e.g. for the one corresponding to role "isPrimitive" and the metainstances for all the accessible roles of the entity type.

Line 10 prints out the minimum multiplicity associated with role "name" of entity type "Entity type", which is one. Just as `EntityType` is a subclass of `Instance`, `Role` is, too. Consequently, we can use the navigation methods of class `Instance` on `nameRole` in order to get the data that is associated with the corresponding Role metainstance. `Role_MinMult` is a static field of class `Role` which contains the `Role` object for the minMult role of entity type Role. Class `Role` has such static fields for all important roles, such as the ones in the PD metamodel. Line 11 commits the transaction.

The next C# sample listing demonstrates simple write operations:

```
1   FbTransaction ta = Server.BeginWrite();
2   Instance i = Instance.LoadObject("Jack", ta);
3   foreach (Role r in i.Type.GetAccessibleRoles(ta))
4   {
5       foreach (Link l in i.GetLinks(r, ta))
6       {
7           Console.WriteLine(l.Instance1 + ", " + l.Instance2);
8           l.Remove(ta);
9       }
10  }
11  i.Set("name", EntityType.String.LoadInstance("Gill", ta), ta);
12  Server.Commit(ta);
```

Line 1 begins a read/write transaction. Line 2 reads an instance of entity type Object with the label "Jack". The following outer loop iterates over all the roles that are accessible from instance `i`. The roles are, of course, not directly associated with the instance but rather with its entity type.

Lines 5-9 contain an inner loop. Method `GetLinks` reads all the links that are connecting a given instance through a certain role. Links are represented by objects of class `Link`. The inner loop iterates over each link that uses role `r` to connect `i` to another instance. Line 7 prints out the two instances that make up a respective link. Line 8 removes the link from the data. Consequently, the two nested loops remove all the links that are connected to instance `i`.

Line 11 creates a new link between instance `i` and an instance of entity type String. The role through which `i` is connected with the string is role "name", i.e. the string becomes the new label of the instance. We have to get the `Instance` object for the string value with method `LoadInstance`, which is defined on class `EntityType`. The method has to be invoked on the `EntityType` object representing the entity type of the

desired instance. Objects for all common entity types can be found in static fields of class `EntityType`. Line 12 commits the transaction.

3.9.2 Syntactic Improvements

In this section I consider some desirable syntactic extensions that would improve the readability of PDM accesses in source code. Such extensions are commonly known as "syntactic sugar" since they do not add any new functionality, but give existing functionality a nicer interface. The creation of syntactic extensions is unfortunately not supported in many programming languages, also not in C#. Therefore it was not possible to implement them in the repository client library. Nevertheless, it is worth pointing out what a suitable syntax for PDM access could look like.

There are programming languages such as Common Lisp [202] and generative programming tools such as Jasper [165, 166] which offer the capabilities necessary for implementing such syntactic language extensions. These technologies offer syntactic macro mechanisms, which can be used to create new syntactic constructs and translate them into known ones at compile time. This additional translation phase is known as macro expansion.

One possible extension is the introduction of a block construct for transactions. As we have seen in the previous section, every PDM operation requires a reference to the transaction in which it should be executed. Usually a sequence of operations is executed in the same transaction, so it could improve clarity to factor out the transaction reference into a surrounding construct instead of repeating it for every operation over and over again. This could, for example, look like this:

```
1  transaction readonly {
2      Instance i = Instance.LoadObject("Type PD model");
3      i = i.DowncastTo(EntityType.Load("Entity type"));
4      Console.WriteLine(i.GetBool("isPrimitive"));
5  }
```

The operations are taken from the first source code example of the previous section. It is now clear that all operations refer to the same transaction without having to state this explicitly every time. Furthermore, the protocol of first beginning and then committing a transaction is now part of the syntax, so that it can be enforced by the syntax checker. In this way, it is not possible, for example, to forget to commit the transaction because the syntax checker would detect the missing closing curly bracket.

Furthermore, it would be possible through the use of polymorphism to integrate the use of PDM types more closely with the syntax offered for the types of the respective programming language. Such polymorphism is supported in particular by many functional programming languages, e.g. Haskell [214] or ML [221], which use very advanced

type systems with intricate type inference capabilities, such as the Hindley-Milner type system [57].

The cast operator, for example, could be overloaded in a way that makes it possible to use it also on PDM instances. Furthermore, it would be conceivable to introduce literals for PDM elements such as entity types, roles and instances, or automatic casts to and from the native literal values of a programming language. Such automatic casts already exist for some OOP languages in the form of autoboxing, which casts primitive values to corresponding objects and back where appropriate.

With appropriate extensions, line 3 of the previous example could be changed to look like this:

```
i = (EntityType) i;
```

Navigational access as in line 4 could look more similar to the way it is commonly done in most programming languages, using the `.` operator:

```
Console.WriteLine(i.isPrimitive);
```

A link insertion that replaces an existing link such as the one in line 11 of the last example in Sect. 3.9.1 could look like this:

```
i.name = "Gill";
```

Or alternatively, if the multiplicities allow for multiple links, the `+=` operator could be used. Let us assume that `i` contains an instance of an entity type Customer which has a one-to-many association with type Order. The `new` operator could be overloaded to be applicable on entity types:

```
i.orders += new Order();
```

Now assume that entity type Order is associated with type Date through a role "date". It would be possible to extend navigation in a way such that it works analogously to the join operation on multiple data elements. For example, if we wanted to get the order dates of all the orders made by a customer that is represented by instance `i`, it could be done in this way:

```
Collection<Date> dates = i.orders.date;
```

For this to work correctly, a programming language is required that is able to perform dynamic dispatch according to the return type of an operation. That is, the type system would have to perform typecasts accordingly, similar to the convenience methods `GetString` and `GetBool` etc. The `.` operator would be polymorphic with an appropriate definition for every return type that should be handled automatically, such as type Date in the example. Alternatively, the type system itself would have to be extended to include type rules for the PDM.

3.9.3 Metadata-Based PDM Access

In order to understand how the repository client library maps the PDM operations onto the relational schema of the repository database, we need to have a closer look at the metadata that is stored about entity types and roles. The schema includes all the necessary information about the storage of instances and links in database tables. The following SQL snippet shows the definition of table `EntityType`, which contains the "Entity type" metainstances:

```
1  CREATE TABLE EntityType
2  (
3      Id CHAR(16),
4      Name VARCHAR(50),
5      IsPrimitive Boolean,
6      IsMaterialized Boolean,
7      TableName CHAR(31)
8  );
```

The first fields are known from the definition of the PD metamodel in Fig. 3.14. Field `IsMaterialized` indicates whether a metainstance is actually materialized, i.e. if corresponding database tables have been created in the repository. If that is the case and the entity type is non-primitive, then the name of the table containing its instances can be found in `TableName`. This makes it possible to integrate existing database tables into the repository and access them with the repository client library while keeping their original names.

The next SQL listing shows the definition of table `EntityRole`, which contains the "Role" metainstances. Note that it was not possible to call this table `Role` because of a clash with an SQL keyword.

```
1  CREATE TABLE EntityRole
2  (
3      Id CHAR(16),
4      Name VARCHAR(50),
5      Accessor CHAR(16),
6      Partner CHAR(16),
7      MinMult INTEGER,
8      MaxMult INTEGER,
9      IsToSupertype Boolean,
10     IsMaterialized Boolean,
11     OwnerTableName CHAR(31),
```

```
12      OwnerColumnName CHAR(31),
13      FKTableName CHAR(31),
14      FKColumnName CHAR(31),
15  );
```

The first fields are known from the PD metamodel. Analogously to table `EntityType`, field `IsMaterialized` indicates whether the respective relation type of the role is implemented in the database. If it is, then the following fields contain all the information needed to navigate the relation type's links.

Field `OwnerTableName` stores the name of the table that contains the instances that can be accessed through the respective role. Field `OwnerColumnName` stores the name of the column in which the accessible instances are stored. For a non-primitive instance, the `OwnerTableName` will be the same as the `TableName` in the corresponding "Entity type" metainstance, and the `OwnerColumnName` will be the artificial key column `id`.

As described in Sect. 3.5, many relation types are modeled with the help of foreign keys. Columns `FKTableName` and `FKColumnName` make it possible to specify any foreign keys that might be involved in the relational implementation of relation types. `FKTableName` specifies the name of a table that contains a foreign key to instances of the entity type owned by the role. `FKColumnName` contains the name of the foreign key column in that table.

Let us consider some examples that use table `EntityRole`. The following SQL listing inserts the pair of roles for the relation type that associates an entity type with a name:

```
1   INSERT INTO EntityRole
2     (Id, Name, Accessor, Partner, MinMult, MaxMult, IsToSupertype,
3     FKTableName, FKColumnName, OwnerTableName, OwnerColumnName,
4     IsMaterialized)
5   VALUES (hex_to_guid('5b8a986c4062db11afc0b95b08f50e2f'), 'name',
6           hex_to_guid('518a986c4062db11afc0b95b08f50e2f'), null, 1, 1,
7           'F', null, null, 'EntityType', 'Name', 'T');
8   INSERT INTO EntityRole
9     (Id, Name, Accessor, Partner, MinMult, MaxMult, IsToSupertype,
10    FKTableName, FKColumnName, OwnerTableName, OwnerColumnName,
11    IsMaterialized)
12  VALUES (hex_to_guid('5c8a986c4062db11afc0b95b08f50e2f'), null,
13          hex_to_guid('4a8a986c4062db11afc0b95b08f50e2f'), null, 0, 1,
14          'F', null, null, 'EntityType', 'Id', 'T');
15  UPDATE EntityRole
16  SET Partner=hex_to_guid('5c8a986c4062db11afc0b95b08f50e2f')
```

```
17  WHERE Id=hex_to_guid('5b8a986c4062db11afc0b95b08f50e2f');
18  UPDATE EntityRole
19  SET Partner=hex_to_guid('5b8a986c4062db11afc0b95b08f50e2f')
20  WHERE Id=hex_to_guid('5c8a986c4062db11afc0b95b08f50e2f');
```

Line 1-7 inserts the metainstance for role "name" into the repository. The `id` and the foreign key to the accessor are GUIDs which are the same for all repositories. These data are part of the core definitions. The storage of the instances that can be accessed with this role is specified in line 7.

`FKTableName` and `FKColumnName` are `null` because "name" is implemented as a column and does not require a foreign key. Columns `OwnerTableName` and `OwnerColumnName` indicate that the instances of the owner of the role are stored in table `EntityType` in column `Name`.

Lines 8-14 insert the metainstance for the unnamed partner role of "name". Line 14 specifies relational storage of the instances of the owner type. Again, no foreign key is specified. The owner of this role is the non-primitive entity type "Entity type", therefore the table name in `OwnerTableName` is `EntityType` and the column name in `OwnerColumnName` is `Id`.

Lines 15 to 20 connect the two partner roles by updating their `Partner` columns so that they refer to the other role instance, respectively. This has to be done separately from the insertions because of the cyclic dependency between the two partner roles. I could not insert the foreign key to the second role metainstance in the creation of the first role metainstance because the second one does not exist yet.

The RDBMS that was used implements *immediate constraints*, which means that after each command the constraints must hold. In contrast to that, some RDBMSs implement *deferred constraints*, where constraints must hold only after each transaction. With deferred constraints I could have inserted the `Partner` foreign key values right away. But with immediate constraints I have to update at least the foreign key column of the first instance after the second one was inserted. For better clarity, I chose to set the values for column `Partner` for both the metainstances after their insertions.

The next example shows the metadata about the relation type that associates a role with an entity type:

```
1  INSERT INTO EntityRole
2    (Id, Name, Accessor, Partner, MinMult, MaxMult, IsToSupertype,
3    FKTableName, FKColumnName, OwnerTableName, OwnerColumnName,
4    IsMaterialized)
5  VALUES (hex_to_guid('638a986c4062db11afc0b95b08f50e2f'), 'accessor',
6    hex_to_guid('528a986c4062db11afc0b95b08f50e2f'), null, 1, 1,
```

```
7     'F', 'EntityRole', 'Accessor', 'EntityType', 'Id', 'T');
8   INSERT INTO EntityRole
9     (Id, Name, Accessor, Partner, MinMult, MaxMult, IsToSupertype,
10    FKTableName, FKColumnName, OwnerTableName, OwnerColumnName,
11    IsMaterialized)
12  VALUES (hex_to_guid('648a986c4062db11afc0b95b08f50e2f'),
13    'accessible roles', hex_to_guid('518a986c4062db11afc0b95b08f50e2f'),
14    null, null, null, 'F', null, null, 'EntityRole', 'Id', 'T');
```

Line 1-7 insert the role through which the accessor entity type can be accessed from a role instance. Hence, the owner of this role is entity type "Entity type", the value of column `OwnerTableName` is `EntityType` and the value of `OwnerColumnName` is `Id`. The accessor entity type of a role can be obtained by using the foreign key column `Accessor` of table `EntityRole`.

The partner role "accessible roles", which is specified in lines 8-14, does not specify a foreign key. If I want to query all the accessible roles of an entity type, I can join the `Id` of the type with the foreign key column `Accessor` of table `EntityRole`, which was specified in the previous role. I left out the updates for column `Partner`, which are analogous to the last example.

The next example shows the metadata for the implementation of the many-to-many relation type between entity type "PD model" and entity type "Entity type". A PD model can contain many entity types, and each entity type can be included in many PD models.

```
1   INSERT INTO EntityRole
2     (Id, Name, Accessor, Partner, MinMult, MaxMult, IsToSupertype,
3     FKTableName, FKColumnName, OwnerTableName, OwnerColumnName,
4     IsMaterialized)
5   VALUES (hex_to_guid('b8088a874062db11afc0b95b08f50e2f'), 'types',
6     hex_to_guid('558a986c4062db11afc0b95b08f50e2f'), null, null, null,
7     'F', 'PDModel_EntityType', 'EntityType', 'EntityType', 'Id', 'T');
8   INSERT INTO EntityRole
9     (Id, Name, Accessor, Partner, MinMult, MaxMult, IsToSupertype,
10    FKTableName, FKColumnName, OwnerTableName, OwnerColumnName,
11    IsMaterialized)
12  VALUES (hex_to_guid('b9088a874062db11afc0b95b08f50e2f'), null,
13    hex_to_guid('518a986c4062db11afc0b95b08f50e2f'), null, null, null,
14    'F', 'PDModel_EntityType', 'PDModel', 'PDModel', 'Id', 'T');
```

Lines 1-7 define the role "types" through which the entity types of a PD model can be accessed. Since the owner type "Entity type" is a non primitive type, `OwnerTableName` is `EntityType` and `OwnerColumnName` is `Id`. This relation type is implemented with a new table `PDModel_EntityType` that contains two foreign key columns. As specified in the columns `FKTableName` and `FKColumnName`, column `EntityType` of that new table refers to an entity type metainstance.

Lines 8-14 describe the unnamed role that is owned by entity type "PD model". It can be used to obtain all the PD models in which an entity type is used. Hence, `OwnerTableName` and `OwnerColumnName` are `PDModel` and `Id`, respectively. The extra table for the relation type `PDModel_EntityType` also has a foreign key column `PDModel` referring to PD models, therefore `FKTableName` and `FKColumnName` contain these names. The updates for column `Partner` were left out again.

The last example shows the metadata for the recursive relation type between partner roles:

```
1  INSERT INTO EntityRole
2    (Id, Name, Accessor, Partner, MinMult, MaxMult, IsToSupertype,
3    FKTableName, FKColumnName, OwnerTableName, OwnerColumnName,
4    IsMaterialized)
5  VALUES (hex_to_guid('6d8a986c4062db11afc0b95b08f50e2f'), 'partner',
6    hex_to_guid('528a986c4062db11afc0b95b08f50e2f'), null, 1, 1,
7    'F', 'EntityRole', 'Partner', 'EntityRole', 'Id', 'T');
8  UPDATE EntityRole
9  SET Partner=hex_to_guid('6d8a986c4062db11afc0b95b08f50e2f')
10 WHERE Id=hex_to_guid('6d8a986c4062db11afc0b95b08f50e2f');
```

The remarkable thing with this relation type is that it is modeled with only a single "Role" metainstance. This is due to the fact that this recursive relation type is symmetric, i.e. for any two roles r_1 and r_2

$$(r_1.\text{partner} = r_2) \rightarrow (r_2.\text{partner} = r_1).$$

For non-symmetric recursive relation types it makes sense to have two different role metainstances. For example, if we modeled a linked list, the recursive relation type associating the elements of a list would have two different roles: one for getting the previous list element, and one for getting the next one. However, if we used two different roles for symmetric relation types, both roles would always lead us to the same instance. This would be clearly redundant, therefore only one role is offered by defining just a single role metainstance that is partner of itself.

3.9.4 Caching and Event Notification

Another important part of AP1 is the caching and event notification architecture in the repository client library. Entity instances and links between them, as well as the metainstances for entity types and roles, are cached as objects in memory and updated whenever they are changed in the database. Consequently, each application using the repository via the client library enjoys the performance boost of an up-to-date read cache, and can communicate synchronously with the other applications.

The repository client library provides a dynamic, intertransactional cache with an avoidance-based cache consistency protocol. The protocol is based on the transactional protocol of the RDBMS, and uses synchronous declaration of write permission faults and update propagation in order to implement an overall read-one-write-all (ROWA) behavior. Thus it can ensure one-copy serializability. For an overview of database caching protocols and terminology see [94].

There are systems, e.g. the DBCache system [25], that use a similar approach for caching. But in contrast to most other systems, which use the RDM to represent the data of a relational database in the cache, AP1's local cache is based on the PDM. In the DBCache system, cache consistency can be compromised due to cache latency, which cannot happen in AP1. Due to AP1's reuse of the RDBMS's concurrency control mechanism for the cache, transactional consistency is guaranteed all the time.

Writes are done directly to the database (write-through). This is important to realize synchronous communication between tools through immediate event notification. If an application changes the database, the change is detected by a trigger and a change event is sent to all connected database clients. Clients can react to change events. Whenever an instance of the repository client library receives a change event, it updates its cache.

All changes that occur in a repository are logged in a table `ChangeLog`. Furthermore, the repository keeps a sequence counter `ChangeNum` which is used to assign an ordinal running integer number to each change. `ChangeNum` can also be used to read the number of the last change. How exactly the triggers and the change log are implemented, and how they can be used for undo/redo and version management, is described in Chapt. 4.

Transactional Access

Figure 3.16 shows a sequence diagram that illustrates the different steps of transactional repository access of an application using the repository client library. The direction of time in this diagram is downwards. The boxes at the top show the different parts that are involved. The rounded boxes represent operations that are performed on the different parts. Arrows signify that an operation is invoked by a part or enclosing operation.

The diagram starts with a client application requesting from a local instance of the

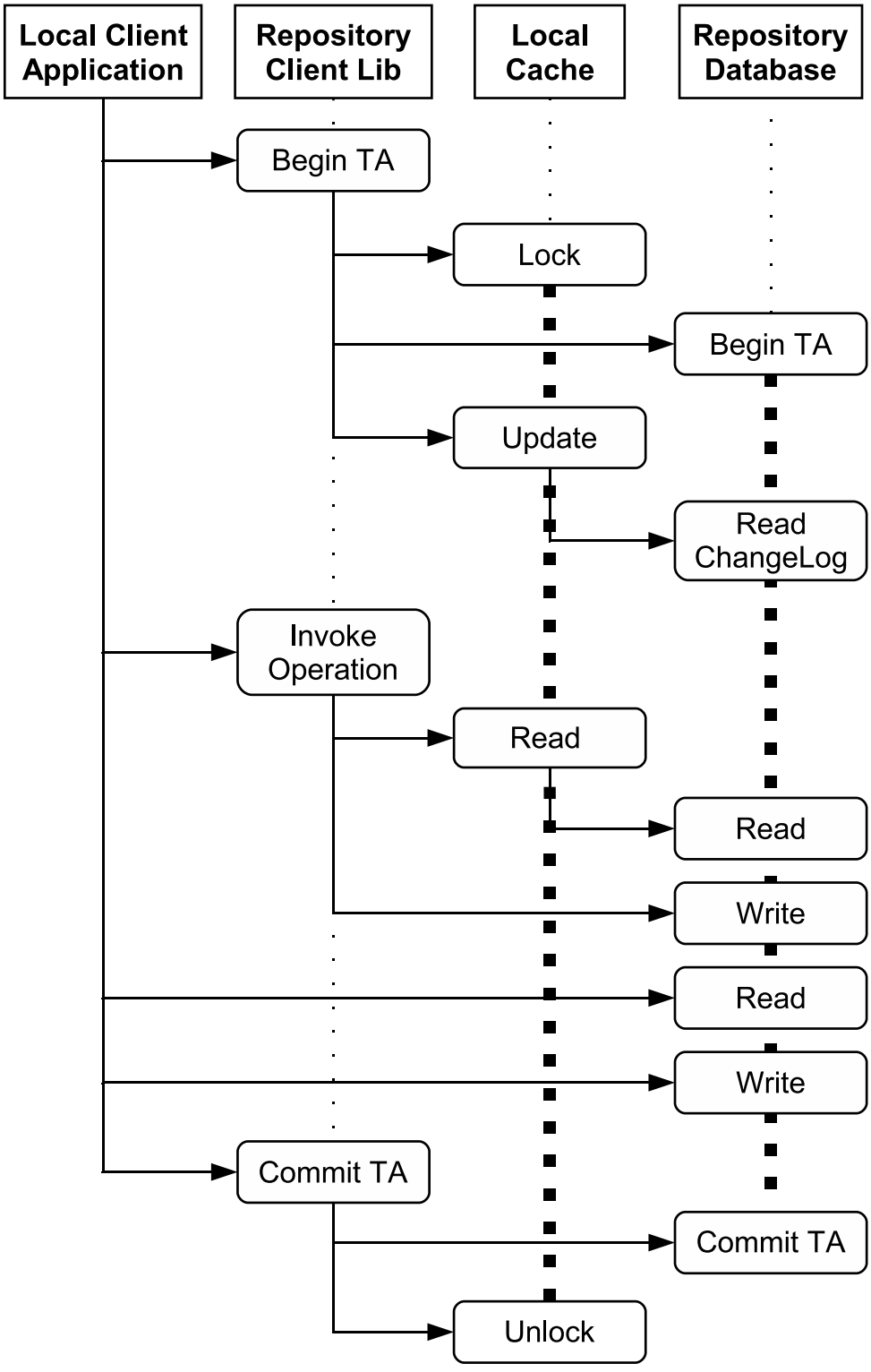

Figure 3.16: Sequence diagram of transactional access with the repository client library.

repository client library that a new transaction should begin. At this point I assume that the library's local cache is unlocked. The first thing the repository client library does is lock its local cache.

The lock is a concurrency synchronization construct that helps to make sure that a

resource – in this case the local cache – is used by at most one thread at a time. If a thread tries to acquire a lock that has already been obtained by a different thread, then the thread is put into a waiting queue and has to wait until it is its turn to get the lock and use the resource. Each thread must acquire the lock before using the resource that the lock is meant to protect, and release it afterward to let other threads use the resource. In the diagram, the thick dotted line below the local cache indicates when it is locked.

Acquiring the lock first makes sure that no other local client applications can use the local cache of this repository client library instance at the same time. This is important because different transactions must be isolated from one another. It is a reasonable restriction because there are usually not many separate tools running concurrently on the same machine. An alternative would be to implement a more sophisticated concurrency control mechanism in the library, which might not actually be used much. If need be, different tools running on the same machine can use more than one instance of the client library to relax the restrictions imposed by the mutual exclusive access to the local caches.

After acquiring the lock on the local cache, the client library begins a new database transaction on the repository database. This ensures that the state of the local cache is older or the same as the state of the database as it is visible in the database transaction. This is important because the local cache can be updated to match the state of the database transaction. The thick dotted line below the repository database signifies the duration of the database transaction.

If I changed the order of the first two steps then I might end up having a local cache with a state newer than that of the database as seen from the transaction. This simply means that the state of the transaction is too old. The database itself always contains the latest state as there is no write-caching.

The next step is to bring the local cache up-to-date with the state of the database as seen from the database transaction. First the library checks whether such an update is actually necessary by reading the number of the last change from the database. Each local cache records the number of the last change that it has been updated with. Consequently, if the number of the last change in the local cache is equal to the last change number of the database, no updating is necessary. The number of the last change in the local cache can never be greater than that of the database because every change happens first in the database, and only afterward in the local caches.

If the number of the last change transferred to the local cache is smaller than the number of the last change in the database, then the local cache reads all the instances from the `ChangeLog` table that have not yet been read. Note that these might not be all the new instances, but just the ones that are visible from the database transaction. There may be newer ones that were committed after the database transaction began.

The local cache updates itself according to the changes read from the database.

Changes are only of significance if the affected instances and links are actually part of the local cache. Storing all changes in the local cache is not practical since it would likely end up having a lot of information that is not relevant for the local client applications. Each client application has a working set, i.e. a set of instances and links that it uses. The aim of a good cache is to capture this working set. At this point, event notification takes place, which I will describe later on.

Once the local cache is up-to-date, control is given back to the client application that started the transaction. The next action illustrated in the sequence diagram is the invocation of an operation. The AP1 framework offers various operations, from basic ones such as navigation to more sophisticated ones such as intercession or generation, which can be used by client applications.

When an operation is invoked, this is always done on a transaction. So if something goes wrong, it will not have any effect, no matter how complex the operation was. The sequence diagram illustrates how an operation performs reads and writes. A read is always performed on the local cache. If the cache contains the data that is requested by the read, then the read can be performed very efficiently. This is typically the case for the current working set. If the data is not part of the cache – e.g. because the local working set is shifting – the cache will load it from the database. Such data becomes part of the cache.

If the cache is full, e.g. because a limit has been set to its size or there is not enough fast memory, then parts of the cache can be removed. This is usually done with the help of heuristics since the system does generally not know for sure which data will be read in the near future. Typical heuristics are to remove “least recently used” (LRU) or “least frequently used” (LFU) data elements, or to use a combination of the two.

As already mentioned, writes are always done directly on the database, using the database transaction. So the cache is not involved in write operations at all. During a transaction, it is not even modified when a write is performed. It stays locked and can thus also not be modified by other threads. As a consequence, client applications cannot expect to modify data and then read the modifications from the cache.

This is not a problem because applications are naturally aware of the modifications they do. Whenever a modification is relevant for an operation, a client application can keep its own records about it and act accordingly. Having the local cache reflect the modifications of the running transaction would be very complex and usually not justified by the added value it would provide. It would mean that the cache had to use a versioning mechanism [22] similar to the one used in the RDBMS. Modifications of a transaction would be cached under a new version, so that they can be undone in case the transaction is aborted.

The sequence diagram also illustrates that it is possible for a client application to access the repository database directly, using SQL. Both read and write access is possible. Of

course, such access is not based on the PDM view on the data, nor does it benefit from the local cache. But very powerful queries can be expressed with SQL in a declarative manner. Furthermore, the query optimizer of the RDBMS can be utilized in this way, which can result in a performance boost for complex operations.

After a client application has completed an activity, it has to commit the transaction for modifications to take effect. The commit command of the repository client library issues a commit for the database transaction, and then unlocks the local cache. If the cache was unlocked first, it would be possible for another client application thread to start a new transaction before the database transaction was committed. This new transaction would not see the modifications of the old transaction, therefore that order would potentially hamper the flow of communication in the system.

Event Notification

The repository client library allows applications to subscribe to events connected to the objects in the cache: e.g. an application can be notified when a link is deleted or added to an instance. This mechanism forms the basis of AP1's control integration, since applications are able to communicate with others just by writing to the repository.

Figure 3.17 shows a sequence diagram illustrating the way change event notification is implemented in the AP1 system. As in the previous sequence diagram, time is oriented downwards. Whenever a change happens in the repository database, this change is logged and connected clients are notified. This is done with the help of database triggers. The Firebird RDBMS supports event notification as a native feature, so that this approach is well supported by the database. Details about the change log and database triggers can be found in Chapt. 4.

A change can happen at any time in the repository database. The exact position of the "Change" shape below the repository database is not important. Without the loss of generality, I assume that at the time of the change the repository client library instance under consideration is executing a transaction. As I have discussed in the previous section, such a transaction causes the local cache of the library instance to be locked.

Each repository client library instance has a handler that is invoked on a separate thread whenever a change notification is received from the repository database. The first thing that handler tries to do is to lock the local cache. Of course, this cannot be done while a transaction is performed with the library instance, but the thread will be queued for access to the local cache and eventually acquire the lock.

The change notification handler proceeds in the same manner as the operation for beginning a new transaction described in the previous section. After the local cache has been locked, a new database transaction is begun. The handler determines if the cache is up-to-date by reading the number of the last change from the database and comparing

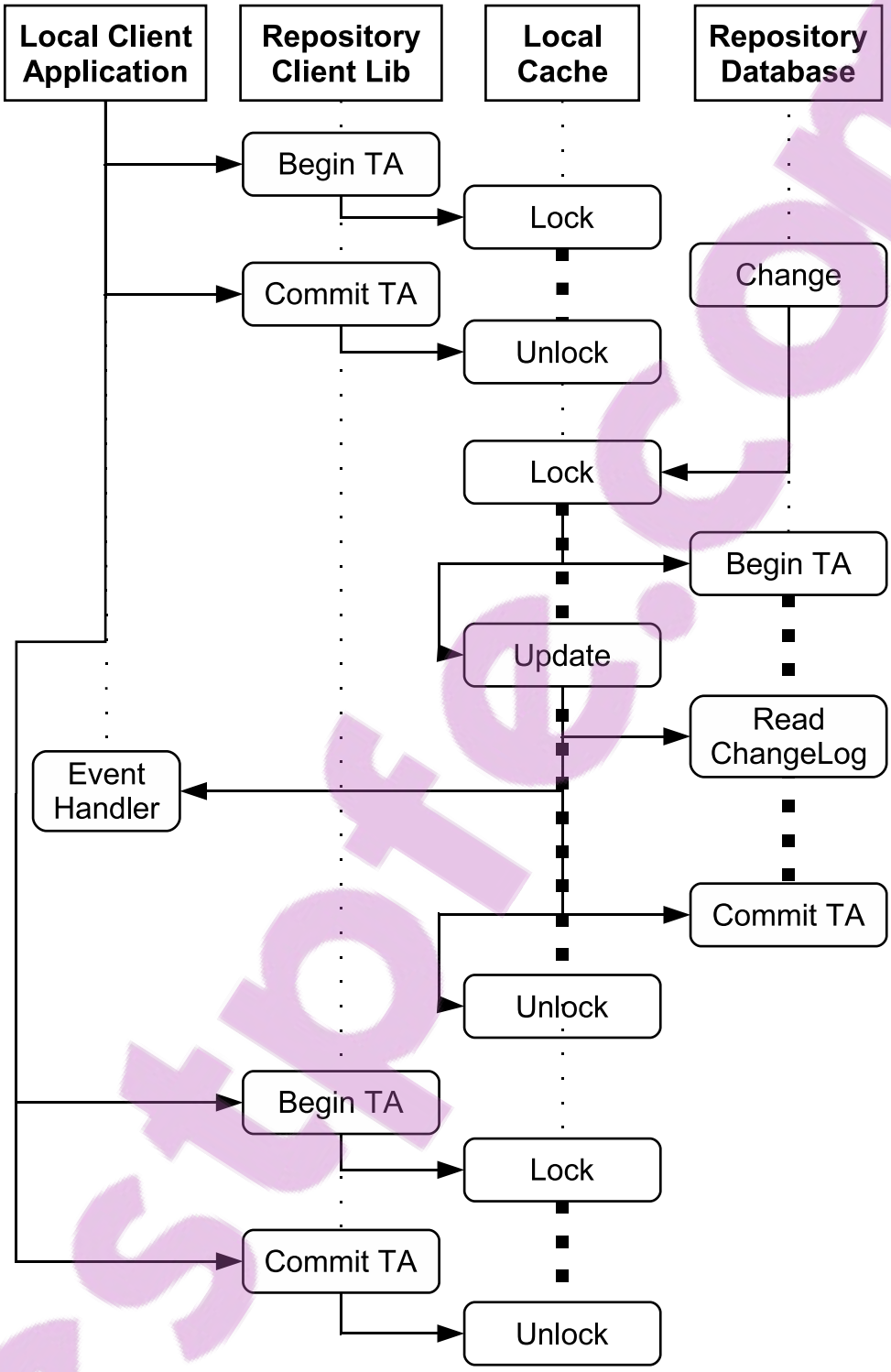

Figure 3.17: Sequence diagram of change event notification with the repository client library.

it with the number of the last change in the cache. The changes that have not been replicated in the cache yet are loaded and processed.

Again, only those changes are selected that affect the data in the cache, and the changes are applied. The objects representing instances and links in the cache can have

event handlers associated with them. The most important events are the addition of a new link to a particular instance with a particular role, and the deletion of a particular link.

If an object that was affected by a change specifies a relevant event handler, then this event handler is called after the cache has been brought up-to-date with all the new changes. The event handler can perform operations in the context of the database transaction, and can also benefit from the local cache. After all changes were processed, the database transaction is committed and the lock on the local repository is released. Now a client application thread can begin a new transaction.

3.10 Related Work

Most of today's software development projects use a file system in order to store their artifacts because most tools load and save their data in the form of files. Usually repository technologies with functionality for software configuration management are used on top of the file system. Popular examples of such repository technologies are the Concurrent Versions System (CVS) [39], Subversion [49], IBM Rational ClearCase [20] and MS Visual SourceSafe [192]. AP1's repository is generally different from such file-based technologies in that it supports structured data storage. Data is not managed on the level of files, which are relatively big chunks of data with unknown internal structure. Instead, the structure of all data is explicitly captured. AP1 also supports software configuration management. However, due to its structured nature, functions such as change control can operate on a much more fine-grained level than the file-based systems. Chapter 4 elaborates on this.

There are other structured repositories that reuse file-based repositories. An example of such a system is the Ragnarok software development environment [47]. It uses its own format in order to store structured information about software components in files. The files, in turn, are stored and versioned by means of an unstructured, file-based repository. For structured data, such systems cannot offer the performance of structured database systems since the underlying technology does not consider the structure of the data. The reuse of an unstructured repository does not help with the access and manipulation of structured data and metadata; it only takes care of versioning it. The Ragnarok system, for example, does not support any metadata manipulation; it only offers a single built-in model, which can only be used for the description of software components.

The Eclipse Modeling Framework (EMF) [32] is an extension of the Eclipse IDE that supports the definition and editing of structured data. It can generate Java code for basic data viewing and editing from a model specification. EMF targets the presentation layer, and is not primarily concerned with data storage. EMF uses its own metamodel, the EMF Ecore model, which was influenced by the OMG MOF. By default, model data are stored

in XML files using the OMG XML Metadata Interchange (XMI) [169] standard. EMF is not a repository technology but a framework for model-based editing, with the option to import and export data using the file system. There are additional Eclipse plug-ins that support the storage of EMF data in a relational database, but they do not offer advanced functionality such as change control.

iRM [179] is a repository similar to AP1's, which is based on the MOF. Like AP1, iRM provides data management capabilities and reflection functionality for introspection and intercession of types. In general, iRM provides similar functionality for data and metadata management as AP1. However, iRM's store for metadata is separated from the data store. Thus data and metadata are not as tightly integrated, i.e. they cannot easily combined as in AP1. Applications can access iRM through an API, or through a non-standard relational query language similar to SQL. However, iRM does not leverage existing RDBMS technology, but instead implements its own DBMS. RDBMS standards are not supported. The iRM system, like many other XML-based repositories, does not provide a mechanism for version control.

The repository [189] of IBM's AD/Cycle CASE platform [155] is, like AP1's repository, also based on a relational database and can thus store data in a structured manner. Tools can access the database directly, or use an API that provides an entity-relationship (ER) model [41] view on the data. The API provides various extensions to the ER model, and also allows the grouping of data into objects that are accessed through methods. This does not mean that the data model of the repository is directly integrated with that of an OOP language, but merely that the repository offers a similar notion of data abstraction. The existence of different versions of an object is only anticipated in the data model of the repository. In contrast to AP1 there are no built-in mechanisms for version control. Only a centralized configuration is supported. Objects are locked while used by a tool, which causes issues of long transactions and a potential loss of productivity due to an unnecessary serialization of the workflow. AD/Cycle uses a different data store for flat files, the library. As a result, flat file data and structured data are not tightly integrated. All in all, AD/Cycle is a heavy-weight system. It comprises many different components and introduces numerous specialized constructs. Many of its notions are not orthogonal. Furthermore, there are software and hardware dependencies on other IBM products, making AD/Cycle a highly proprietary system.

In the 1990s, after IBM introduced AD/Cycle, many other companies released their own repositories as well, e.g. the Microsoft Repository [21]. The Microsoft Repository is also built on a relational database, offers an object-based view onto the data, and has shortcomings that are similar to those of AD/Cycle. Many of those repository systems were similar, and all of them eventually failed in gaining widespread acceptance.

Reasons for failure of CASE technology were discussed in Chapt. 1. Among them are

complexity and heterogeneity due to the usage of proprietary technology. AP1's repository is a very lightweight system that offers many advanced features such as change control through standard, non-proprietary RDBMS features alone. Additional features, such as the PDM perspective on data and event notification, rely on a small number of relatively simple concepts.

3.11 Conclusion

In this chapter I described how a structured repository can be implemented on top of a relational database, using a mapping between the RDM and the more abstract PDM. The repository offers functionality for structural reflection, and uses concepts for facilitating data integration such as GUIDs. The impedance mismatch between the PDM and object-oriented programming languages is addressed with an object-oriented repository client library, which offers advanced functionality such as a local cache and event notification to applications. The resulting architecture is novel and has been proven feasible in a proof-of-concept implementation.

Naturally, there are also potential disadvantages. For example, one of the characteristics of the dynamic mapping described in Sect. 3.5 is that the number of database tables grows with the number of entity types. While this was relatively unproblematic for the particular RDBMS I was using, this can be a major obstacle for other RDBMS. In some RDBMSs, creating a new table is a costly operation and tables – even when they are empty – take up a lot of space. With such systems, the memory consumption of a large number of entity types, which are necessary in the context of CASE, might be unacceptable. Furthermore, database tuning is much easier for static schemas than for dynamically changing schemas. The latter case is relatively rare in practice and can therefore cause problems with some RDBMSs. As a result, it is not possible to implement the proposed architecture on every RDBMS, despite the standardized SQL interface they usually offer.

The fact that not all RDBMSs offer an event mechanism to the user is another factor limiting the applicability of the AP1 architecture. Such an event mechanism is crucial for cache synchronization and change notification in the repository client library. It is also necessary for other, similar applications such as database replication, which one might want to use together with the AP1 system. An event mechanism could be added by modifying the RDBMS if it is possible to do so, or by adding an additional layer on top of it. However, this can be quite complicated and can therefore rarely be considered as a practical solution.

Another potential problem lies in the centralized nature of the repository system. Since many clients can be connected to it simultaneously, the repository database may become

a performance bottleneck. The AP1 client library offers a read cache, but this cache has to be synchronized at the beginning of every transaction, and write operations are not cached at all. As a consequence, the number of developers that can use a repository at the same time is limited. Fortunately, this is an old problem that has attracted a lot of attention, therefore there are ways to mitigate it. First of all, a lot of effort has gone into making modern RDBMSs suitable for high-performance transaction processing because this is what such systems are commonly used for. There exist various methods of database tuning. Furthermore, I propose to use AP1's repository in a decentralized configuration if performance should become a problem. This possibility is described in Chapt. 4.

Nevertheless, the repository presented here also has many advantages such as the possibility to exploit the rich features of RDBMSs, which was explained in Sect. 3.3. As we will see in Chapt. 4, the mechanisms used for cache synchronization and event notification can at the same time be used for change control, resulting in a slim, elegant design. I will continue to improve the repository with new ideas, such as the literal inclusion of primitive values into GUIDs, which was anticipated in Sect. 3.8.

4

Change Control

One of the challenges of complex software projects is the management of all the artifacts involved, such as model data, documentation and source code. It is very important that the artifacts can safely be stored and retrieved, and that changes to artifacts are managed carefully. In particular when many developers work on the same project, it is important to track every change, detect conflicting changes and offer the possibility to undo and redo changes.

In this chapter I describe the solution for change control that is used in the repository of the AP1 system. It is based purely on relational database features and is able to manage changes on a very fine-grained level. Because of the open nature of the repository, it lets users define their own version models, and thus offers a particularly high level of customizability.

In Sect. 4.1 I introduce the problem domain of software configuration management. Section 4.2 describes how the PDM is utilized as the basis for change control. Section 4.3 describes how changes in the database are recorded in a log, and Sect. 4.4 shows how the change log can be exploited in order to undo or redo changes, and customize the version model. Sections 4.5 and 4.6 discuss synchronous and asynchronous centralized collaborative development, respectively. Section 4.7 discusses decentralized collaborative development. The chapter concludes with Sect. 4.9.

4.1 Introduction

Complex software projects have to deal with a large number of artifacts, such as source code, model data, documentation, configuration data, project plans, and others. The ability to manage all these artifacts is critical, and therefore projects with a certain complexity use software configuration management (SCM) tools that support the developers in this task. In this chapter I present a system for SCM which is different from most other such systems in that it is built on the standard features of a RDBMS.

Common configuration management tools keep all project artifacts in a shared *repository*. Developers can access the repository, and store, retrieve, delete and change artifacts. Usually the unit of granularity in the store is that of a file, and the repository as a whole is organized as a tree, as in most file systems. Access to the repository is possible over a network, and can be restricted with an access control mechanism.

Usually developers have their own *working copy* of the artifacts they are working on locally on their computer. Changes are done to the working copy and do not immediately affect the repository. A common practice is to finish a single functional change, which may consist of many individual changes to several artifacts, on the working copy before committing it to the repository. If this is done correctly, then other developers will only see changes when they are mature enough not to disturb the overall integrity of the repository, e.g. if they do not break the build.

When changes are committed to the repository, new *versions* are created. Versions represent the state of one or more artifacts at a given point in time. The repository keeps track of and archives all versions that are created during the development life cycle. That is, developers can retrieve old versions, and undo or redo the last version. Most SCM tools also support *merging* of different changes done to the same base version into a single new version. This is why such tools are also called *version control systems*.

There are different *version models*, i.e. models that define what is versioned, how versions are identified, and what operations exist on versions. Two of the most popular version control systems, for example, the Concurrent Versions System (CVS) [39] and Subversion (SVN) [49], differ in the way they version artifacts: while CVS versions individual files, and does not version the folder structure that contains those files, SVN versions the repository as a whole, including its folders. When a file is changed in CVS, then a new version of that file is created. When a file is changed in SVN, then a new version of the whole repository is created. A general overview of other, older version models can be found in [50].

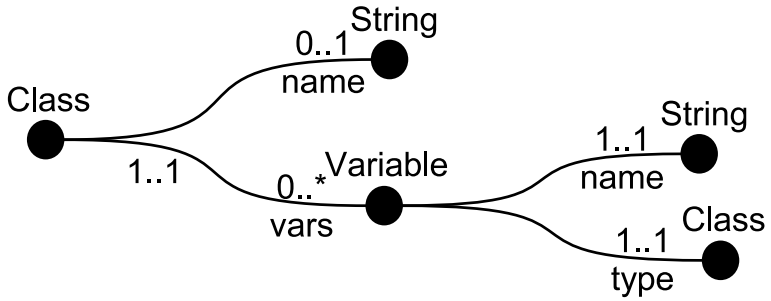

Figure 4.1: Example of a partial PD model for source code.

4.2 A Fine-Grained Perspective on Relational Data

In terms of Conradi and Westfechtel [50], the PDM forms the product space of my change control system, i.e. the space in which the data that is changed lives. The PDM makes change control relatively easy because data are represented using only few concepts, so that there are only two change operations that need to be considered. The PDM decomposes tables, which can potentially have a complex internal structure, into entity types and relation types, which have a very simple structure when considered individually. Changes are operations on links: a link can either be added to a relation, or removed from it. Changing a field of a row to a different value generally corresponds to the deletion of an existing link, and the addition of a new one.

Figure 4.1 shows a simple partial PD model for source code: classes that have names and member variables, with each variable declaring a name and a class type. This PD model will serve as a running example. The following listing shows a relational schema for it, defined in SQL, which was created according to the mapping rules described in Sect. 3.5:

```
1   CREATE TABLE Class (
2     id CHAR(16) NOT NULL,
3     name VARCHAR(64),
4     CONSTRAINT Class_PK PRIMARY KEY (id)
5   );
6
7   CREATE TABLE Var(
8     id CHAR(16) NOT NULL,
9     name VARCHAR(64) NOT NULL,
10    class CHAR(16),
11    CONSTRAINT Var_PK PRIMARY KEY (ID),
12    CONSTRAINT Var_FK FOREIGN KEY (class) REFERENCES Class(id)
13  );
```

```
14
15  CREATE TABLE String (
16    id CHAR(16) NOT NULL,
17    val VARCHAR(64),
18    CONSTRAINT String_PK PRIMARY KEY (id)
19  );
```

Each table represents one of the entity types. I have given the table representing entity type Variable the name `Var` in order to avoid a clash with a built-in keyword. All instances are identified by 128 bit GUIDs [119], so that there cannot be any key conflicts. These GUIDs are represented with type `CHAR(16)`, and differ only in the first bytes if generated on the same machine. Therefore, GUIDs from the same machine in consecutive index entries can be efficiently compressed by some RDBMSs.

The relation type between Class and Variable is implemented as a foreign key from table `Var` to table `Class`. Relation types to primitive types, e.g. the relation types between Class and String and Variable and String, are implemented by inlining the primitive type as field into the respective table. I chose a field type of `VARCHAR(64)` for the values of the String instances.

Note that also the primitive type String has a corresponding table, although the String values are inlined directly into the other tables. This is done in order to also have GUIDs for all primitive types: table `String` has field `id` for a GUID and a field `val` for the associated primitive value. The GUID of a String element is calculated with a hash function from the corresponding String value, as described in [119], so that the same String value will always have the same GUID. This way, all instances can be uniquely identified with a GUID.

The PDM can be used to represent all artifacts that have to be stored in a versioning system. It is especially suitable for structured artifacts, including source code but many other formats as well. For office documents, for example, XML representations have now been standardized and can be stored in the PDM.

4.3 The Change Log

The heart of the proposed change control mechanism is a change log, which is essentially just another database table. All changes, i.e. all insertions of new links and removals of old ones, are to be recorded in this log. Each change corresponds to one row. The following SQL code defines the change log table:

```
1  CREATE TABLE ChangeLog (
2    id CHAR(16) NOT NULL,
```

```
3     repository CHAR(16) NOT NULL,
4     number INTEGER,
5     dateTime TIMESTAMP,
6     instance1 CHAR(16),
7     entityRole2 CHAR(16),
8     instance2 CHAR(16),
9     kind CHAR(1) NOT NULL,
10    CONSTRAINT ChangeLog_PK PRIMARY KEY (id)
11  );
```

From the PDM perspective, changes are instances. Therefore they are identified with a GUID, which is stored in field `id`. This GUID identifies not only a change but also the version, i.e. the state of the whole repository after that change. Potentially, there can be many repositories in a software development organization, e.g. for different projects. Therefore, each repository is identified with a GUID, and a change log entry contains the GUID of the repository in which it was created in field `repository`.

Field `number` represents the running number of a change within the repository in which it was created. This field establishes a total order on all the changes of a repository. It is set automatically with a sequence generator each time a new `ChangeLog` row is inserted. These change numbers correspond to versions of the whole repository and are only unique in the context of a particular repository. This is similar to the concept of revision numbers in SVN, but on a fine-grained level. Field `dateTime` contains a time stamp signifying when a change occurred. This field is not essential for change control, but an example of additional information that can easily be added in order to extend or customize its functionality.

Fields `instance1`, `entityRole2` and `instance2` identify the link that has been added or removed. `instance1` and `instance2` contain the GUIDs of the instances that are connected by the link. `entityRole2` is a GUID that identifies the role through which the first instance is connected to the second one. It refers to the corresponding Role instance of the metamodel, and because each role belongs to exactly one relation type, the relation type of the link is identified as well. Field `kind` signifies whether the link was added or removed, by containing either the character `A` or the character `R`.

The fact that changes are stored alongside ordinary data in the database means that they enjoy the same benefits. Access to change data is controlled by the RDBMS, which ensures its integrity and reliable storage. The change data reflect the history of the creative input that goes into a project, and are therefore part of it. Storing them in the database means that they can also be queried and analyzed efficiently like ordinary data, which is significant for the management side of SCM. A project manager can, for example, see how much effort is spent on individual parts of a system, or correlate data about changes and

defects. Such instrumentation is particularly essential for the development of complex software, and has been documented in the development process literature, such as the CMM [200].

Log entries are permanent, which means that once an entry is inserted it cannot be deleted or changed. Only insertion is possible. This can be enforced with the standard access control mechanism. It is important in order to prevent corruption of the log. It is possible to insert change log entries of other repositories into a repository, but all log entries need to have appropriate repository GUIDs in order to tell them apart. Naturally, change log entries should not be "forged", i.e. nobody must insert change log entries that do not correspond to actual changes in the respective repository. This can also be prevented by using appropriate access control.

In the next sections, I will describe how change log entries can be created automatically with triggers. There are many systems that create log entries using triggers, but without an appropriate model it is not possible to log changes so that they can be efficiently used for change control. If a system should log changes using the relational data model as a basis, it would not be clear how a change could be stored in a structured manner. A change could occur on an arbitrary column with an arbitrary type, so a row in a change table might need columns for all possible data types in order to capture every possible update. In order to capture insertions and deletions of rows, such a system would have to devise a scheme for identifying arbitrary rows, which is not straightforward either. This is why most systems with some kind of logging create logs only for particular tables, or create unstructured log entries, by just storing every SQL command as a string.

4.3.1 Logging Changes of Foreign Keys

Non-primitive entity types are implemented with their own table. In the example model, the non-primitive entity types Class and Variable have corresponding tables `Class` and `Var`. Each of these tables has a trigger that is executed after every SQL `UPDATE` operation on the respective table. These triggers have a regular structure that depends solely on the structure of the PD model, and are generated automatically in the AP1 system. Triggers can be created with standard SQL using the `CREATE TRIGGER` command.

Each foreign key column that is part of the table of a non-primitive entity type implements a one-to-one or one-to-many relation type to some other non-primitive entity type. This is, for example, the case for field `class` of table `Var`. In order to log the changes of such a relation properly, several cases have to be considered. The following listing shows the slightly abbreviated part of the update trigger of table `Var` that handles changes of field `class`:

```
1   IF (OLD.class IS NULL AND NEW.class IS NOT NULL)
2   THEN
3     INSERT INTO ChangeLog(Id, Repository, DateTime,
4       Instance1, EntityRole2, Instance2, Kind)
5     VALUES (create_guid(), REPOSITORYID, 'now',
6       OLD.id, ROLE1ID, NEW.class, 'A');
7   ELSE IF (OLD.class IS NOT NULL AND NEW.class IS NULL)
8   THEN
9     INSERT INTO ChangeLog (...)
10    VALUES (...,OLD.id,ROLE1ID,OLD.class,'R');
11  ELSE IF (OLD.class IS NOT NULL AND NEW.class IS NOT NULL
12    AND OLD.class<>NEW.class)
13  THEN BEGIN
14    INSERT INTO ChangeLog (...)
15    VALUES (..., OLD.id, ROLE1ID, OLD.class, 'R');
16    INSERT INTO ChangeLog (...)
17    VALUES (..., OLD.id, ROLE1ID, NEW.class, 'A');
18  END
```

Lines 1 to 6 handle the case where the Variable instance corresponding to the row that was changed did not have a link to a Class instance, but now a link is inserted. The prefix `OLD` represents the row before the update, and `NEW` afterward. The old `class` value was null, and the new one is not. Consequently, a change entry has to be created that describes the link between the row representing the Variable and the row representing the Class. `ROLE1ID` is the GUID of the Role instance representing the connection with which a Variable is associated with a Class. `create_guid` is a user-defined function that generates a new GUID. `REPOSITORYID` is the GUID for the repository in which the change took place, and `'now'` denotes the current time.

Lines 7 to 10 handle the case where the Variable instance corresponding to the changed row had a link to a class instance, but this link was removed. The non-`NULL` `class` value was overwritten with `NULL`. The link in the change log entry is given by the GUID of the Variable, the GUID of the Class, and the GUID of the role that associated them.

Lines 11 to 18 handle the case where an existing link between the Variable and a Class instance was replaced by a new one. This happened because the old non-`NULL` value of `class` was overwritten with a new non-`NULL` value. The fact that the link was overwritten is only evident when considering the multiplicities of this relation type in the PD model or its relational implementation. Therefore both a removal and an insertion of a link are recorded in order to have a consistent log for all possible relation types.

Note that it is not possible to test just for `OLD.class<>NEW.class`: if `OLD.class` or

NEW.class are NULL then the whole expression evaluates to false. This is because of the semantics of NULL, which is in this case that of "unknown". If any value is compared with a value that is unknown, then it cannot be told for sure whether the comparison is true. Hence, the body of a conditional, which is only executed if the condition is true, would not be executed. That is why the different combinations of NULL and NOT NULL need to be considered.

In addition to the trigger after UPDATE, a trigger after INSERT is needed. This is necessary because an INSERT statement can also set values for each of the columns. This trigger is much simpler because only the case that the new value for a column is NOT NULL and the old one does not exist needs to be considered. A row which has just been inserted represents an instance that could not have had any links before. The INSERT trigger code for field class of table Var, for example, would look like this:

```
1  IF (NEW.Class IS NOT NULL) THEN
2    INSERT INTO ChangeLog (...)
3    VALUES (..., NEW.Id, ROLE1ID, NEW.Class, 'A');
```

The example model does not include a many-to-many relation. Such relations are implemented with dedicated tables that contain two foreign keys, one to each of the associated types, respectively. For such relations, an UPDATE trigger would remove a link and add a link if one or two of the foreign keys in a row have been changed. An INSERT trigger would always record the insertion of a new link, and a DELETE trigger would always record a link removal.

Note that, except in the tables that represent many-to-many relations, no rows need to be deleted from the repository. All other rows represent instances, and although the PDM makes it possible to remove links that may exist between instances, there is no necessity for removing the instances themselves. For primitive instances this is natural because they have a fixed number of elements anyway. A non-primitive instance without links corresponds to a row that solely contains a GUID as its primary key, all other columns being NULL. Such an instance can still be linked with others.

4.3.2 Logging Changes of Non-Key Fields

As mentioned in Sect. 4.2, if a non-primitive entity type refers to an instance of a primitive type, such instances are inlined into the non-primitive entity type's table. This preserves the natural structure of relational tables, and makes access to those primitive instances very efficient. As a result of such inlining, relational schemas may contain columns which represent links to primitive instances, without the GUID of such an instance actually being present in the table.

In order to associate a GUID with every primitive instance, tables like the `String` table in the example are defined. Such tables are permanent like the change log, i.e. only new rows can be added, but rows cannot be updated or deleted. This is because they express the mapping of primitive values to their GUIDs, which is invariant. Consequently, triggers recording changes on links to primitive instances need to use these tables in order to get the primitive instances' GUIDs. This is supported with stored procedures that provide the GUID for a primitive value of a certain type, and made efficient with indexes on the primitive values of such tables. For entity type String, for example, the stored procedure `GetStringIdByValue` is defined, shown in the following listing:

```
1   CREATE PROCEDURE GetStringIdByValue
2   (val VARCHAR(64))
3   RETURNS (id CHAR(16))
4   AS BEGIN
5     SELECT id FROM String
6       WHERE val=:val INTO :id;
7     IF(:id IS NULL) THEN BEGIN
8       id = create_string_guid(val);
9       INSERT INTO String (id, val)
10      VALUES (:id, :val);
11    END
12    SUSPEND;
13  END
```

In lines 5 and 6 the procedure queries the `String` table for the GUID of the given value `val`. This search is performed with the help of an index defined on column `val`. If found, the GUID is stored into variable `id`. In SQL commands, the variables are distinguished from column names by a preceding colon. A primitive value might not be present in the table, and if this is the case, the procedure will create a new row with a new GUID in lines 8 to 10. When the procedure terminates, it returns a valid GUID for the primitive instance.

The following listing shows part of the code in the trigger after `UPDATE` on table `Var` that handles changes of field `name`:

```
1   IF (OLD.Name IS NOT NULL AND NEW.Name IS NOT NULL
2     AND OLD.Name<>NEW.Name)
3   THEN BEGIN
4     SELECT Id
5       FROM GetStringIdByValue(OLD.Name)
6       INTO :OldId;
```

```
 7     INSERT INTO ChangeLog (...)
 8     VALUES (..., OLD.Id, ROLE2ID, :OldId, 'R');
 9     SELECT Id
10       FROM GetStringIdByValue(NEW.Name)
11       INTO :NewId;
12     INSERT INTO ChangeLog (...)
13     VALUES (..., OLD.Id, ROLE2ID, :NewId, 'A');
14   END
```

In trigger code that handles changes of non-key fields, the same cases have to be considered as for foreign key fields because primitive fields can be `NULL` as well. The listing above shows only one of those cases. Procedure `GetStringIdByValue` is used in order to get GUIDs for old as well as for new primitive values. Other triggers are constructed analogously.

4.3.3 Performance

Several tests were run in order to compare the performance of different SQL operations with and without change control. All tests were performed on the example schema, with the Firebird 2 [211] RDBMS running on a Pentium 4 with 3.4Ghz and 2GB RAM. For every single test run a new set of data was created, so that there were no cache interactions between test runs.

All tests were set up to have 100 rows in table `Class`. The `INSERT` tests added rows to table `Var`, averaging over all insertions to get a precise time for a single insertion. The `UPDATE` tests were set up with a varying number of rows in table `Var`, all of which were linked to a random Class respectively. For each test run, all `Var` rows were updated, and the time for a single update was calculated by averaging. For each test, the size of table `Var` was increased in steps of 100 from 100 to 5000. For each size, five test runs were performed.

The measurements for a single operation remained relatively constant over different sizes of table `Var`, therefore the test results can be meaningfully represented by the average times of the different tests. These averages are shown in Fig. 4.2. The results are as expected in that the additional time required for an operation with change control is related to the number of additional write operations necessary for creating a change log entry. There was hardly any overhead for the read operations on table `String` because the RDBMS could mostly perform them on the cache.

In the best case, inserting a new complete row into `Var` takes about twice as much time with change control as it takes without. In this case, the value for the column `name` is already present in table `String`, so that no new row needs to be created there. Only

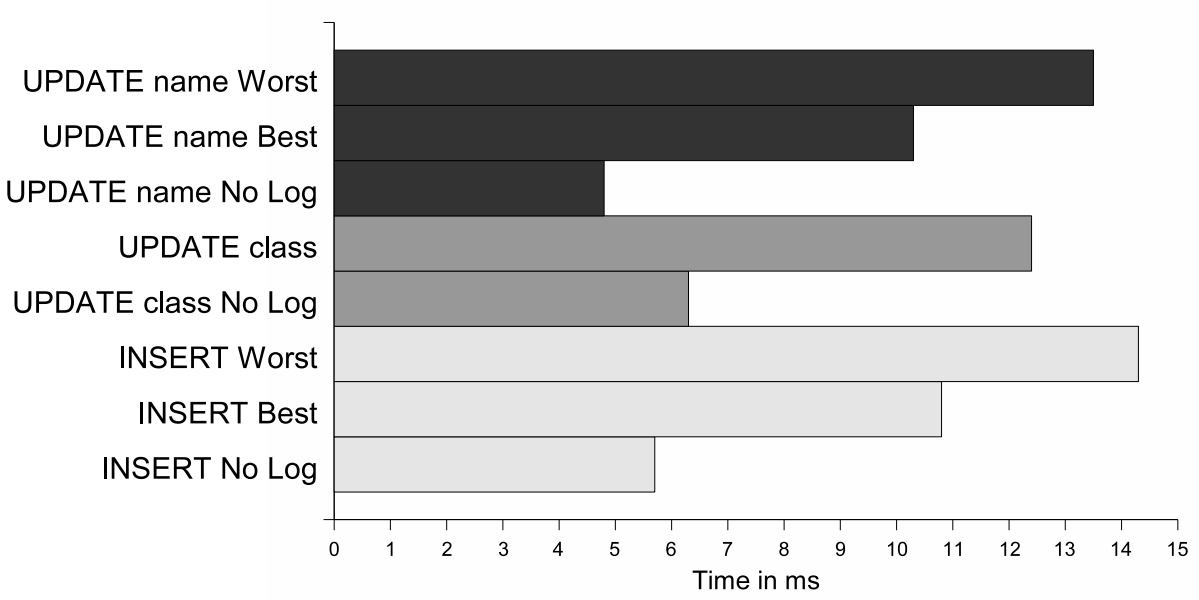

Figure 4.2: Performance of SQL operations with and without change control.

one additional write operation is necessary: the insertion of the change log entry. The worst case is the one where the value for the column `name` is not present in table `String` and has to be inserted.

Updating field `class` of table `Var` takes about twice as much time with change control as it takes without. This is due to the additional insertion of the change log entry. In the best case of an `UPDATE` on the primitive field `name`, i.e. when an entry for the value already exists in table `String`, the time with change control is roughly twice as much as without because of the additional insertion of the change log entry. In the worst case, i.e. if a corresponding entry in `String` does not exist, some additional time is used for the insertion of the `String` row.

4.4 Using the Change Log

It is possible to use the change log for non-linear undo and redo. For this, an arbitrary subset of all the changes that happened in the repository can be chosen. Because changes are just instances themselves, they can easily be linked with other data. The AP1 system allows users to edit all the entity and relation types in a repository with a structured editor, and thus users can define relation types between the Change entity type that is given by the change log and their own models. These possibilities will be described in the following sections.

4.4.1 Undoing Changes

A subset of the changes done in a repository can be undone by undoing each of the changes in descending order of field `number`, i.e. reverse to the order in which they occurred. This overcomes common restrictions. The undo/redo mechanisms offered in most tools allow it only to undo the last changes, and a change can only be undone if all its succeeding changes were undone. My approach allows it to undo changes no matter when they were done. It is not even necessary that the changes that are to be undone are consecutive. This is also known as non-linear undo.

This is possible because of the compositional nature of the PDM. Relations are just sets, and inserting or removing a link does not change the other links in the set. These operations do not disturb the structure of a relation. Invalid links are prevented by the fact that relations are typed.

If I were using a version model where there can potentially be dependencies between changes, then this would not be possible and versioning would be much more complicated. Consider, for example, the commonly used model that is based on files. If I create a new file in such a model and change that file, then the change inevitably depends on the creation of the file. I could not, for example, undo the creation of the file and then undo the change because after the first undo the file would be gone.

Undoing PDM operations is straightforward because removal of a link is naturally the inverse operation of link insertion. Thus, it is possible to undo an insertion by removing the respective link, and vice versa. Furthermore, the two operations are naturally idempotent, i.e. doing them more than once does not change the result. The data cannot be corrupted by undoing an operation more than once. Again, this is due to the set semantics of relations. Once a link has been inserted into the relation, inserting it again does not change it. And once a link has been removed, removing it again does change the relation either.

Whenever a sequence of change entries is undone, this causes a corresponding sequence of new change entries to be created for the resulting changes. That is, if a single undo operation has an effect, then a new change entry is created that makes it possible to undo this operation as well. Change entries can be regarded as merely providing the information necessary to undo respective changes, with the undo operations causing new changes of their own. In this way, the continuity of the change log as the history of a system is preserved.

4.4.2 Redoing Changes

Analogously to the undo operation, a subset of the changes done in a repository can be redone by redoing each of the changes in ascending order of field `number`, i.e. the

order in which they occurred. The redo operation offers the same flexibility as the undo operation, being non-linear in nature. A single change entry is redone by simply repeating the respective link insertion or removal. The indempotence of link insertion and removal ensures data integrity. Analogous to undo, redoing changes that affect the repository results in corresponding new change log entries.

4.4.3 Managing Groups of Changes

One of the advantages of this change control mechanism is its customizability. The changes recorded in the change log are maximally fine-grained, i.e. they cannot be subdivided into finer changes. But those changes can be grouped into arbitrary coarse-grained units. All change log entries are instances of the entity type Change, and this entity type can be arbitrarily used in other PD models. A user can define a new entity type that groups changes and thus specifies subsets of them. Such new entity types can then be used in order to view, manage, undo and redo changes on a more coarse-grained level.

One of the groupings built into the AP1 system is that of transactions. For every transaction, an instance of type Transaction is created. The triggers logging the changes done by that transaction associate the Change instances they create with its Transaction instance. All complex operations in the AP1 system, e.g. code generation, are performed in their own transaction. Thus, the sets of changes required to redo or undo them are naturally available through Transaction instances. The database schema is extended in the following way:

```
1   CREATE TABLE TransactionLog (
2     id CHAR(16) NOT NULL,
3     number Integer,
4     CONSTRAINT TransactionLog_PK PRIMARY KEY (id)
5   );
6
7   CREATE TABLE ChangeLog (
8     ...
9     transactionId CHAR(16),
10    CONSTRAINT ChangeLog_FK FOREIGN KEY (transactionId)
11      REFERENCES TransactionLog (id)
12  );
```

Field `number` of table `TransactionLog` records the order of the transactions within the repository. The foreign key column `transactionId` of table `ChangeLog` refers to a `TransactionLog` entry. In this way, it is possible to extend the system with additional

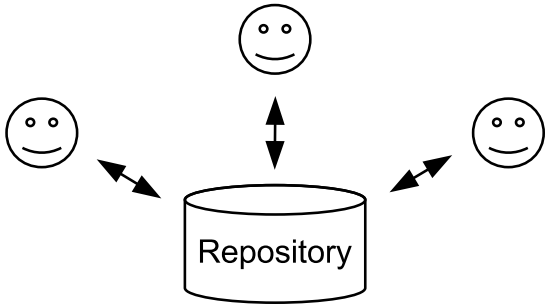

Figure 4.3: Synchronous centralized collaborative development.

information, e.g. the ID of the user who did a change, without disturbing the basic change control mechanism.

Various composition relations are conceivable for user-defined grouping of changes. For example, changes could be grouped according to the functional changes they cause in a software product. Furthermore, developers could categorize groups of changes, e.g. into bug fixes, new features, refactorings etc., or add log messages documenting them. In the AP1 system, the creation of such new user-defined version models is supported with the same tools that are also used for the creation and modification of other, non-versioning-related models.

4.5 Synchronous Centralized Collaborative Development

The standard mode of operation in the AP1 system supports synchronous centralized collaborative development. This is illustrated in Fig. 4.3. All developers work on the same central repository. Their views on the repository are synchronized. Whenever a developer changes the state of the repository, the change is propagated to all others.

Developers connect with their development tools to the remote RDBMS that manages the central repository. Development tools are database clients. All changes done by the developers are directly performed on the repository, so that conflicts between concurrent changes are handled by the transaction mechanism of the RDBMS. The transaction mechanism prevents concurrency problems: lost updates, for example, cannot occur because the RDBMS will not allow two transactions to read and change the same data at the same time. Dirty reads, i.e. reading of inconsistent intermediate states of the data by other transactions, cannot occur because transactions are isolated from one another.

The change log is updated immediately after a change. A new change causes the RDBMS to notify all connected clients, so that those clients can update their internal data immediately. The notifications are sent out by triggers. A change made by one

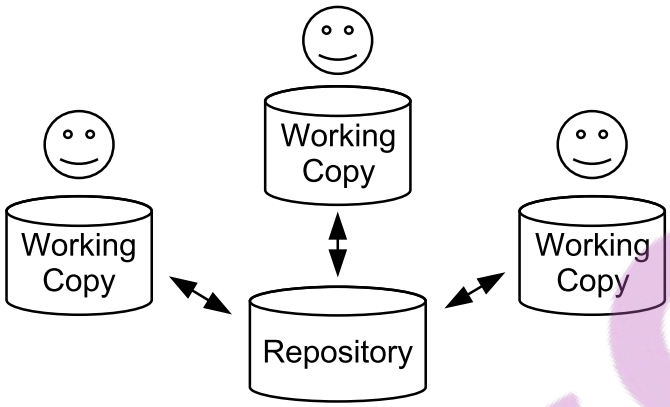

Figure 4.4: Asynchronous centralized collaborative development.

developer becomes immediately visible for all developers. The AP1 system implements a read cache for development tools, so that the central RDBMS is not overly burdened with read requests. As a consequence, synchronous collaboration techniques such as pair programming can be applied in a distributed environment.

4.6 Asynchronous Centralized Collaborative Development

Asynchronous centralized collaborative development, which is illustrated in Fig. 4.4, is the most common form of distributed software development. For most open-source projects, for example, this method is essential. There is still a central repository, but instead of working on it directly, developers work on their own local working copies. A best practice is that developers complete a change on their working copy and commit it only to the repository after it has been tested. In this way, many people can contribute to a project without breaking its code base. From time to time, developers update their working copy with the latest changes to be up to date.

Two challenges of asynchronous collaborative development are the prevention of lost updates and the integration of changes. There exist two common approaches: locking and merging. The former approach lets developers put a lock on parts of the repository they are currently working on, thus preventing others from interfering with their development. This approach is rarely used because it significantly reduces productivity by hindering other developers from working on parts that are already locked. Locking is commonly only possible on the granularity of whole files, which is very coarse. Therefore, this section will focus on the more modern merging approach.

For the approach presented in this paper, GUIDs play a very important role. Firstly, they avoid key conflicts between non-primitive instances. Developers can create new non-

primitive instances in a distributed fashion without two of them ever having the same GUID. Secondly, the way GUIDs are generated makes sure that the same primitive values will always have the same GUID. This is because those GUIDs are created as hash values on the primitive values.

Developers each work on their own local database. They can update their working copy by querying the central repository's change log for changes that occurred after they did their last update. Change log entries are small and can be sent over a network efficiently. When developers have completed a modification, they try to commit it by sending their new change log entries to the central repository, together with the number of the change that was last updated from the repository. The repository server checks if other developers committed any changes after the committing developer last updated their local working copy. If so, the developer is forced to update their repository before doing the commit.

When an update is done, a developer has to check if the new changes from the repository conflict with new local changes. *Syntactical conflicts* are changes that could not coexist, e.g. two different names for a Variable instance, and they can be detected automatically by the client. *Semantic conflicts* that do not cause syntactic clashes are changes that violate the correctness of an artifact on a semantic level, and therefore cannot be detected automatically in general. However, a client can, for example, warn a developer if there are two recent changes that are likely to be conflicting. A conflict is more likely, for example, when changes are close, or in the same cluster of instances and links. In every case, conflicts have to be resolved manually by a developer as there is no general way of deciding for one change or another automatically.

The advantage over the standard approach, which merges files textually regardless of their internal structure, is that changes are merged syntactically, i.e. according to the structure defined by the PDM. Syntactic merging [50] is only very rarely supported in other systems.

4.7 Decentralized Collaborative Development

It is also possible to use the presented mechanism for decentralized collaborative development, which is illustrated in Fig. 4.5. In this setting, there is no central repository, but instead every developer has their own. Repositories communicate with other repositories in order to commit or update changes to and from each other.

Both a synchronous and an asynchronous configuration are possible. A simple approach for synchronous collaboration is the use of database replication technology between the repositories. Such technology handles the task of synchronizing the different repository databases over a network. Asynchronous collaboration is possible analogously

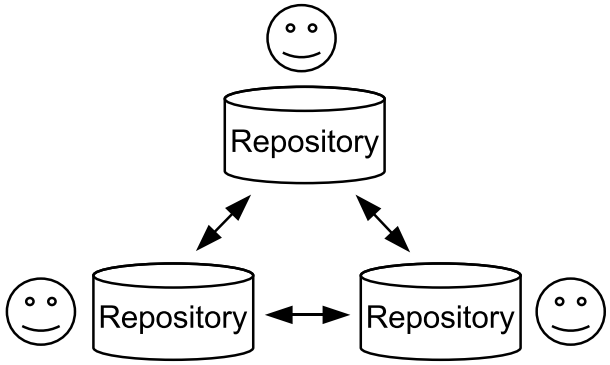

Figure 4.5: Decentralized collaborative development.

to the way it is done in a centralized setting. Each repository can act as a working copy or the central repository.

4.8 Related Work

In a configuration management system, one has to distinguish the version model from the product model [50], i.e. the way versions and their interrelations are represented and the way the actual product data are. The most common product models are file-oriented paradigms, e.g. RCS [215], CVS [39] and SVN [49]. They differ mostly in their version models, in that more modern systems version whole snapshots of the repository, including the folder structure, while older ones version each file individually. The data structure of the version model of such systems is essentially opaque to the user. In my approach, both the version and the product model are much more fine-grained, and their openness makes it possible to reuse and extend them.

The COOP/Orm system [147] enables a more fine-grained product model by defining a hierarchical structure for artifacts that can be structured that way. It also uses a concept of minor revisions, i.e. versions that can be created temporarily and locally by a developer in order to create a finer version model. However, individual changes are not accessible by a developer.

The viability and the advantages of a RDBMS approach to SCM for real projects were shown by the Sorceress system [195]. The same advantages apply to my approach. However, the product model of the Sorceress system is still file-based, and within a file unstructured, text-oriented, and thus much more coarse-grained.

The CSCWriting system [137] uses so-called AID tags, which correspond to change log entries, on textual, unstructured data. It is shown that such entries can be efficiently used for merging. Furthermore, the role-based access control (RBAC) model [190] is integrated into the version control model. Interestingly, the SQL standard already incorporates RBAC, so that my RDBMS-based, structured approach can also make use of it.

Other SCM systems are based on the object-oriented data model. The Adele system [85], for example, is a database system that can store objects in different versions and relations between them. However, Adele is highly proprietary and does not use common standards.

Also the Molhado system [162] uses an object-oriented artifact model, but changes are only recorded if a developer explicitly asks for it, and not all parts of the data are versioned by default. Molhado's data model is much more complicated than the PDM; the whole system is bound to Java code and therefore on a less abstract level.

Change propagation and response graphs (CPRGs) [102] have been proposed for managing changes and constraints of OO data. Changes are represented in a way similar to my approach, but are more complex due to the differences between the OODM and the PDM. CPRGs have a stronger focus on change event routing and handling, with changes being propagated along designated relationships. By default, change descriptions are ephemeral. My approach records all changes permanently in a global change log, and focuses on version management rather than change propagation. Nevertheless, my change data can also be used for consistency management and event notification in general, as discussed in Chapt. 3.

As [86] points out, contemporary SCM tools have the tendency to incorporate more and more features, and thus become too big and unwieldy over time. My approach mitigates such a trend by reusing many features of the underlying RDBMS. Modern database systems already solve many of the problems that typically occur when managing data in an industrial environment.

4.9 Conclusion

I presented a novel mechanism for logging and controlling changes of data on a fine-grained, structured level. The mechanism is, as a part of AP1's repository, based on a relational database. The proposed approach is sufficiently time efficient, and offers many advantages over common versioning systems. Undo and redo can be performed on a fine-grained level, and the version model can be adjusted to individual needs. Different configurations are possible, such as synchronous or asynchronous collaboration, and centralized or decentralized repositories. Due to its structured nature the system supports syntactic merging.

A potential limitation of this mechanism is its memory usage. Because of its fine-grained nature, the number of change log entries can become very large, especially if many developers use the system simultaneously. As can be seen in Sect. 4.3, a single change log entry can consume about 100 bytes. As a result, large changes may consume more memory than equivalent changes in unstructured delta-encoding-based approaches.

While memory is nowadays usually not an issue any more, it might not seem worth spending a lot of memory on change control in circumstances where a less powerful, more memory efficient approach might be sufficient. The memory consumption can be reduced with techniques such as prefix compression, which was described in Sect. 3.8, but may still be unacceptable in some situations, such as in environments where memory is scarce.

Another limit of my approach is its incapability of appropriately managing changes within primitive, inherently unstructured data such as text. Assume, for example, that a paragraph in a text document is represented as a String instance. If the paragraph is changed, the corresponding change log entry would only express that change in terms of one link being replaced by another. For a small change in a large paragraph, a new String instance would have to be stored for the new version of that paragraph, resulting in a high degree of redundancy. Unstructured change control approaches would typically store such changes using delta encoding. This allows them to capture the differences between versions instead of having to store each version in full. Consequently, such approaches would be better suited for systems that deal mainly with inherently unstructured data. Due to the strong proliferation of unstructured data and file-based version control systems, it would be of interest to explore hybrid approaches that combine structured and unstructured mechanisms. This is future work, and could help make the transition between the different approaches less difficult.

5

Robust Content Creation with Form-Oriented User Interfaces

In this chapter I describe how content can be created in a way that ensures its integrity at all times, and how the user interface for such a content editing program can be modeled using the methodology of form-oriented analysis. The chapter discusses aspects concerning the data that is being created, as well as aspects of the content editor itself. I show that technological features such as typing, opaque identities and user transactions can facilitate the process of content creation as experienced by the user significantly, and that these features can be effectively incorporated when using the form-oriented analysis model.

This study is significant for the AP1 system because software development is a special case of content creation. The content is the set of artifacts modified by the developers during a project, such as specification documents and source code. In Chapt. 1 it was pointed out that usability is an important concern for CASE technology, therefore studies about the HCI between developers and tools are essential. The issue of robustness is particularly important for CASE because the artifacts of a software development project are usually valuable and should not be corrupted. It is helpful if the integrity of these artifacts cannot be violated easily. Finally, AP1's generic editor is essentially based on a form-oriented user interaction model, and incorporates many of the features for robustness described in this chapter.

Section 5.1 introduces the issue of robustness for user interfaces. Section 5.2 gives a

short overview of the form-oriented user interface model. Section 5.3 describes different notions that can help to make content modeling more robust. Section 5.4 shows how the interaction in desktop applications can be specified with the form-oriented model, and how such interaction can be aligned with a concept of transactionality. Section 5.5 delineates a general approach to content creation that avoids some of the pitfalls of common approaches. Section 5.6 concludes this chapter. Parts of this chapter were published in [72].

5.1 Introduction

For many people computers are tools that, ideally, help their users to express and manage the products of their creativity. Their significance is not their functionality in itself, but the way it can be utilized by the user who wants to take advantage of the manifold possibilities of digital content. As more and more computer systems find their place in our everyday life until people can hardly avoid them any more, it becomes especially important to make their benefits easily accessible and shield the end-user [206] from the shortcomings of computer technology. What we need are *user interfaces*, which deliver functionality in a way suitable to the user, in contrast to *system interfaces*, which deliver functionality in a way suitable to the system.

In this chapter I consider a particular problem: the possibility that a user can modify the data of a system in such a way that the data's integrity is violated. If this is possible, and the nature of such violations is not reported adequately, users can easily find themselves at a dead end. Even if such inconsistent states could in principle be overcome quite easily, such problems can take up quite a lot of valuable time. One approach would be to expect the user to learn how a program should be used properly. But the fact is that most people are not *power users* but *casual users*, who use most programs rather occasionally. Spending time on learning how to use a program is an investment that not everyone is willing to make, and not every program is worth the time. Although software engineering professionals fall certainly into the category of power users with regard to software development tools, and are generally able to avoid inconsistencies, such usability problems can nevertheless lead to annoying mistakes. If developers have to deal with a wide range of tools, e.g. because they are working on different projects, resolving such inconsistencies is likely to be more time consuming. As an anecdotal example, the make build tool uses the tab character as a semantically important marker, although it is visually indistinguishable from a sequence of space characters. Inexperienced users may be tempted to use space characters instead of the tab character, running into errors that are unexplainable just by the look of their make source file. Similar problems exist with other tools that process textual source code.

This chapter considers systems for the creation of digital content, how to model them and how to give them a property which I call *robustness*. My approach is to design systems in a way that guarantees a form of data consistency at any time. Rather than dealing with error detection and error handling, these systems focus on *error avoidance*. The ability of a system to avoid any interaction that might cause inconsistency is what makes the system robust.

In order to be able to make meaningful statements about usability of content creation systems, we are in need of a *theory* of such systems, and fortunately we already have such a theory at hand: the form-oriented analysis methodology [76]. While it is not limited to the specification of content creation systems, one of the contributions of this chapter is to show why it suits this purpose very well and how it can be applied to the modeling of content editing systems with various forms of interaction. I will frequently refer to the creation of web content in order to exemplify different points discussed in this chapter.

5.2 The Form-Oriented User Interface Model

Form-oriented analysis [76] is a methodology for specifying submit/response-style information systems. This means that you can specify systems in which information is received and displayed to a user, in which the user can submit information to the system and thereby get a response that is made visible and opens up new ways of interaction. The form-oriented model can be used for many different application types, such as web applications [75, 80], but it is important to observe that this model can be of particular value in the specification of content editing systems.

A basic notion in this model is the concept of *actions* and *pages*. Pages are parts of the system that report information and offer possibilities of interaction to the user. These possibilities of interaction are called *forms* since they usually allow the user to provide more bits of information, the parameters. Once the user submits the information entered into a form, this information is sent to another part of the system which is an action. An action processes the information in some way and might also access different kinds of data stored in the system, and eventually sends a result back to a page.

This alternating structure of pages and actions can be formally described as a bipartite typed state machine, and a simple way of visualizing it is a *formchart*. Figure 5.1 shows a simple formchart that describes a simple editor for structured data. The bubbles represent pages that display information to the user, while the boxes represent actions that can be invoked from pages. In this figure the captions describe the content that is displayed or the functionality that was activated, respectively. The transitions between pages and actions describe the possibilities of user interaction.

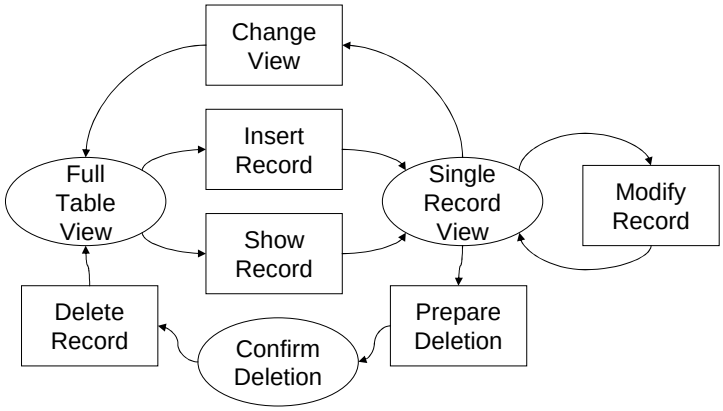

Figure 5.1: Formchart model for simple data editor.

5.3 Content Modeling

As we will see, the way content is modeled can have an immediate impact on the way a user interface can provide access to it. A suitable content model does not only make it easier to implement a good user interface, but can be of significant help to the user when interacting with an application.

I will also discuss a suitable implementation strategy for my content model. For this purpose, I will use the relational data model [48]. Using this industry standard for data management is not just a matter of implementation, but it also mitigates a problem which can be of direct concern to the end-user: data ownership. Of course, usually users have the legal rights on their own creations, but that does not change the fact that many commercial programs for content creation hinder end-users to use the data the way they like. Commercial programs often store data in their own proprietary data format, which can make it difficult to use that data in other programs. In the extreme case we are bound to a single program and, consequently, to a single user interface. In contrast to that, the way data is organized in a relational database is known and made transparent by a RDBMS. Users will never be bound to a single interface due to the support for SQL as a standard access language.

The next three Sects. 5.3.1, 5.3.2 and 5.3.3 discuss aspects of content modeling that can directly provide added value to a user interface. It makes clear that the way we handle content as end-user can directly benefit from the way we think about content when modeling its structure.

5.3.1 Typing

Data types play a significant role in many application domains and have to be represented in the user interface: dates and currencies are classical examples, email addresses are a further example. The use of types is the essential technique to impose structure on

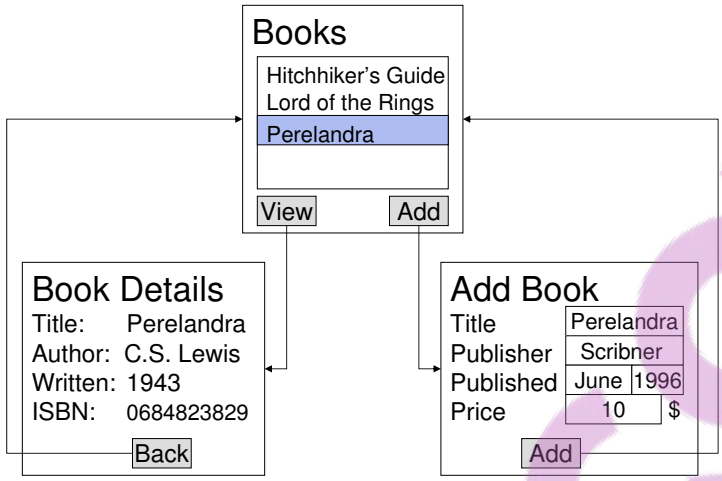

Figure 5.2: Screen diagram of simple book inventory.

content. We divide the universe of possible values into subsets, which may be disjoint or not, depending on the kind of type system we use, and each of these subsets is called a type. A value that is part of such a subset is said to be an instance of the corresponding type.

With the relational data model we are able to give every bit of data in a system an explicit type, which means that the type of the data will be immediately evident in the system. A type is not only used to express structural properties of its instances, e.g. the fact that they are numbers. At least as important as the structural properties are the semantic properties of data that can be expressed by giving them types. A number, for example, can be defined to express a postal code, the size of a salary or the size of a person's shoes.

SQL offers a number of atomic types, i.e. types that cannot be divided any further, such as types for representing numbers and strings. It also offers a way for creating composite types, so called relations, that are made up of basic ones. Each atomic part of a relation is called an attribute, and each attribute has a label that identifies it uniquely within the relation it is defined in.

The properties of content that we make explicit by distinguishing types are not just important for the internal logic of a system. In the area of end-user development the support for typing gets a different motivation. The types are not mere helpers for the learned software engineer, and the reporting of type errors is not the right presentation of type concepts any more: in end-user development the system itself must manage typing issues once users have declared their typing preferences. The purpose of typing here is therefore to free the user from the possible mistakes that are understood as typing errors. As a consequence, the user interface does not offer the wrongly typed choices in the first place. The type system is therefore facilitating development of content that is type-sound.

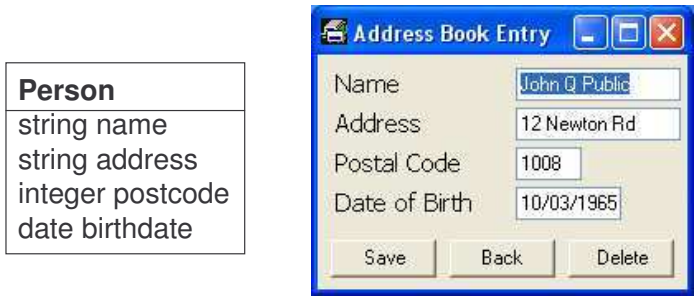

Figure 5.3: Record type and "Single Record View" user interface.

Let us consider user interface development with screen diagrams – a user interface diagram type which is, in contrast to formcharts, not backed by a typed data model. Figure 5.2 shows a screen diagram for a simple book inventory management application. Screen diagrams are probably the most frequently used user interface diagram type because they correspond directly to what the user will see on the screen, which makes them very intuitive. Arrows starting at control widgets and leading to other screens indicate how the user interface changes when certain interactions are performed. However, as Fig. 5.2 demonstrates, the lack of a data model makes user interface development with screen diagrams prone to type errors: it can easily happen that representations of the same content on different screens do not match properly, e.g. items might be missing or the format or level of detail of certain parts of the content might be different. With a typed data model the content is well-defined, and such mistakes can be effectively avoided.

To illustrate the relationship between the type system and the user interface, let us consider the page "Single Record View" of the user interface model in Fig. 5.1 with the assumption that the system we are modeling is an address management program. In the real system the page will probably look similar to that in Fig. 5.3. On the left side of this figure, we see a visualization of the type that represents an address book entry; on the right side we see the user interface for editing data of that type.

It is plausible that each editable field in the GUI corresponds to an attribute in the data type, and furthermore, in a good user interface the type of the attribute influences the way in which the user can interact with the different GUI components of the page (see also Sect. 5.4.1 for this kind of interaction within a page). In the example, the user should probably be able to enter arbitrary strings into the editable fields for name and address; however, the fields for postal code and date of birth should only allow the user to enter a number and a valid date smaller than the current date, respectively. This context sensitive restriction of user input is an important principle for the creation of robust user interfaces, i.e. user interfaces that do only permit interaction that leads to a feasible system state.

The World Wide Web has a strong tradition of text-based development because HTML

Figure 5.4: Article view of the Wikipedia.

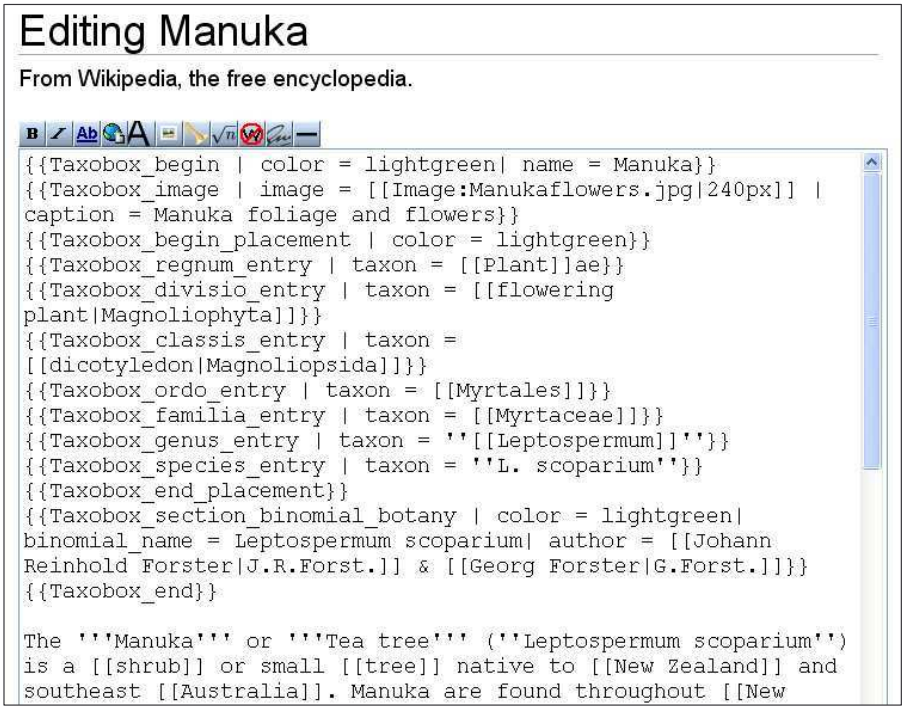

Figure 5.5: Article editor of the Wikipedia.

is often rendered differently on different browsers, and can often only be avoided with appropriate manual fine-tuning. Many authors just use a text editor to create their HTML pages. This is encouraged by the human-readable markup style of HTML. Although text-based editing works well for smaller web pages, it reaches its limits when applied to large and complex ones, and can easily lead to mistakes. For example, the user interfaces of common wiki systems [52] do not only offer functionality for viewing pages, but also functionality for editing them, as can be seen in Fig. 5.4. The creation and modification

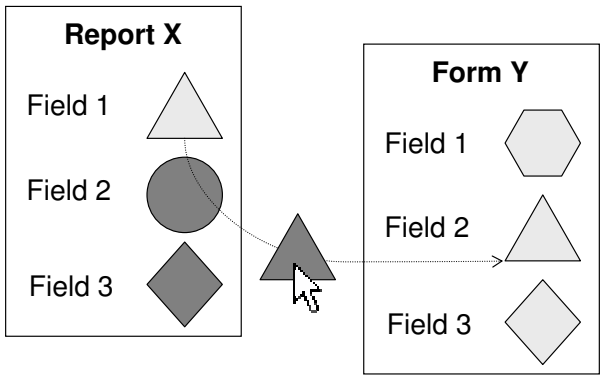

Figure 5.6: Handling opaque identities.

of pages is made available to the end-user. However, the user interface used for viewing is altogether different from that for editing, as can be seen when comparing Fig. 5.4 with the corresponding Fig. 5.5. Editing is done in a purely textual fashion. The problem with text based editing views of web pages or web applications is that they create an inherent point of instability. The user can break the system by entering incorrect input into the edit panel. This can be overcome by systems that use an edit panel similar to the content view [79]. Form-oriented techniques can help in the development of such robust user interfaces, and support the structured information that tends to emerge over time [78]. The aim of the user interface technology proposed here is to guarantee that user input keeps the system in a working state. The approach is different from the text based editing paradigm, which is essentially untyped, and lifts content editing onto a typed level instead.

5.3.2 Opaque Identities

Opaque identity is the technical term for an interface concept that can be realized in quite a number of ways. Opaque identities refer to the way we identify or name things. If we want to name things, be it people, data fields or articles, the best strategy depends on how a name will be used. For example, if we use the name in a discussion, or in a plain text, we necessarily have to have a plain name. But if we want to drag an article from one form of a user interface to another, we are not interested in a plain representation. We are rather interested in a unique identity, not prone to synonyms. Furthermore, we want it to be free of spelling problems and to have type consistency, so that an object can only be used as it should be.

The overall solution to this is the opaque identity. Instead of cutting and pasting, say, a textual object ID from one user interface to another, an abstract representation is offered. In the most general form, this is a small icon that can be cut from, pasted and dragged into the input field of a form where it fits, similar to the user interface depicted

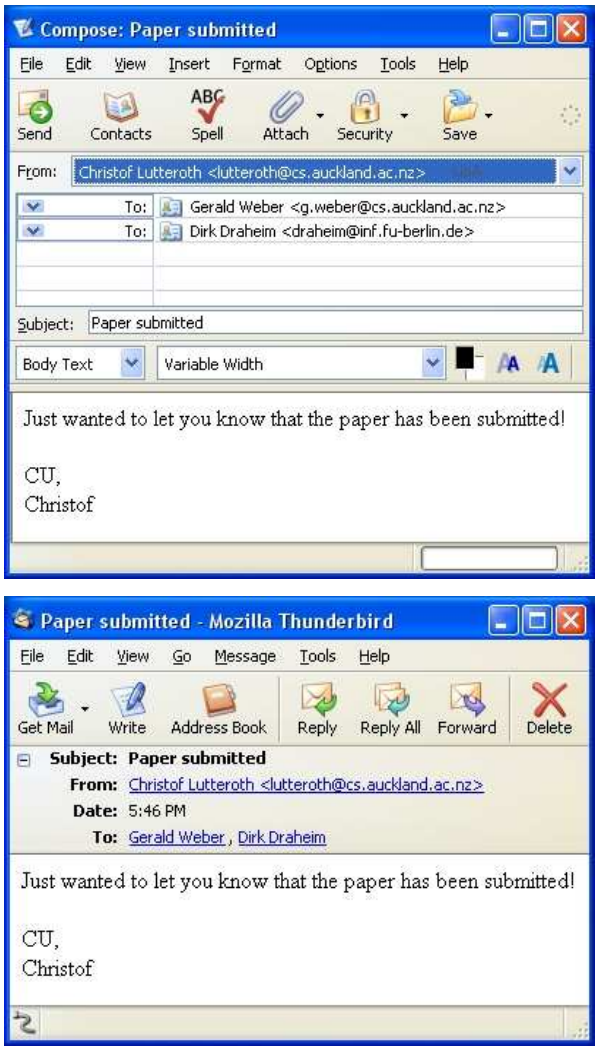

Figure 5.7: Edit window and view window of of the Mozilla Thunderbird email client showing the same email.

in Fig. 5.6. But beside this very iconic representation, the concept of opaque identities refers to a very effective user interface solution that lets the user reference objects without manipulating their names or IDs directly.

Figure 5.7 shows an example where a user interface handles email addresses, which could be represented with opaque identities, in a user unfriendly way: in the edit window the two recipient email addresses are accessible in a table-like widget, but it is not possible to mark and copy multiple cells of this table or all of it. The second window views the same email after it has been created, possibly on the sender or the receiver side, and this time the emails of the recipients are represented as running text. But again, it is not possible to mark and copy multiple email addresses here. The user interface uses different widgets to handle email addresses, and by their looks these widgets seem to offer certain standard functionality which they actually do not. By contrast, the concept of opaque identities strives to unify the way such content is represented and handled.

Relational database systems support the concept of opaque identities by offering the mechanism of primary keys. Primary keys are values in data entries that identify each entry uniquely. Furthermore, a relational database system takes care that there can never be two data entries in a relation with the same primary key. This makes a relational database particularly useful for the implementation of robust systems.

5.3.3 Referential Integrity

In most systems we have different kinds of data that are related to each other. In the address book example, we might not just keep track of the people in the address book, but also of the companies they are affiliated with. Consequently, we would not only have to store entries for people and companies, but also information about which person belongs to which company. Once we have such kind of referential information, there are a number of things that can go wrong.

Of course, a person can only be affiliated with a company that exists in the database. So when setting a person's affiliation, users should only be allowed to choose from the companies for which data entries are available. It is also convenient if users can immediately create a new company entry when they need it, at the place of the user interface where the affiliation of a person is set. Another important question is what happens when a company entry is deleted, although it is referred to by person entries. Whatever strategy is chosen to deal with that case, it is crucial that after the deletion there are no remains left in the data that might raise the impression that the entry still exists. There must not be references that do not refer to valid data, and this has to be reflected in the user interface. The property of data that every reference refers to a valid data entry is called *referential integrity*.

One distinctive robustness issue with HTML concerning referential integrity is the avoidance of broken hyperlinks. Not only text based editing of HTML is prone to broken links; the absence of a high level infrastructure for links makes this a persistent problem. On the other hand, broken hyperlinks are still a relatively easy to fix class of errors. The frequency of updates that could create broken links can be considered low, and at any time the link is broken, it can be fixed by looking up the correct place. There exist alternatives to HTML that solve the problem of broken links [56, 150]. If we want to move on from a simple hypertext to a web application, robustness becomes a much more complicated issue. Looking at today's implementation technologies such as PHP, web applications have by no means simple semantics. The task of developing robust web applications is much more complicated than developing correctly linked websites.

Fortunately today's relational databases are able to ensure referential integrity automatically. It is simply not possible to refer to a non-existent data entry, and if a referenced entry is deleted, the database system takes a predefined action. A user interface that bases

on such a database system is robust to referential violation.

5.4 Two-Stage Interaction

In form-oriented systems, two kinds of user interaction are distinguished. On the one hand, a page usually offers the user ways of interaction that alter or extend the content seen on the respective page, such as editable fields to enter data and other kinds of GUI controls. This is subsumed under the term *fine-grained interaction*. On the other hand, most pages also offer ways of submitting information in forms to the system which cause not necessarily the page content but the overall system state to change and possibly lead to other pages. For this the term *coarse-grained interaction* is used. Usually the end-user will first interact with a page on the fine-grained level, providing input to and changing the content of a form, before submitting the data in a form and inducing a data change on the coarse-grained level. Therefore the overall notion is also called two-stage interaction.

In the following two Sects. 5.4.1 and 5.4.2 I will discuss fine-grained and coarse-grained interaction in more detail. I will illustrate how they can be modeled using the form-oriented paradigm and how their technological implementation relates to an added value for the user interface. In the case of coarse-grained interaction, this will lead us to a new notion, that of user transactions.

5.4.1 Modeling Fine-Grained Interaction

The defining characteristic of fine-grained interaction is that the changes it causes to the system are only ephemeral. The persistent state of the system remains unchanged. Fine-grained interaction is used mainly for the following:

- **Input or modification of form data**
  This might be a textual modification of some sort of editable input field, such as the ones we saw in Fig. 5.3. But it might as well be a modification of some other nature, such as a change of a graphical object on the page.

- **Changes of the data view**
  These do not change the data itself, but rather the way it is perceived by the end-user. Typical examples for this are elision in tree-view-like widgets and sorting of table views, but it might as well be something like a shift from visual to auditive page representation.

- **Ephemeral side-effects**
  These are effects that neither change the representation of the page, nor the persistent system state. Common examples of that category are a feature to copy data

to a clipboard, or one to toggle between different modes of operation of an input device, such as the overwrite and insert modes of the keyboard.

When entering or modifying form data on a page, the data in the form might temporarily become inconsistent. This is usually the case when the information in the form is still incomplete or currently being modified. Consider, for example, the editable field for the postal code in Fig. 5.3. It would be possible and a good idea to make sure that only numbers are entered into the field during input and that the input does not exceed a certain length, but we would probably not want to set a lower bound for the length while input is in process. For a new address book entry the field would be initially empty, and naturally the length of the data would grow gradually. While the data in a form is inconsistent for one reason or another, it must not be possible to submit it to the system. Therefore, the "Save" button in the example cannot be activated until the data in the form is complete and all right. The transient inconsistency of the data during fine-grained interaction does not compromise the integrity of the system because the general system state does not change until a coarse-grained submission of consistent data is done.

A question that arises is how fine-grained interaction can be modeled with the form-oriented user interface model. The answer is simply that there is only one general way of modeling, which is the one described in Sect. 5.2 and illustrated in Fig. 5.1. Form-oriented models are flexible enough, though, to accommodate for both the needs of coarse-grained and fine-grained interaction.

Data can be submitted on a page, which means that it is sent to an action, which in turn sends back data for a new page. In the case of fine-grained interaction, the submitted data needs not be consistent because it is not integrated into the general persistent data model. This makes it possible to model input or modification of form data as a submit/response cycle.

Furthermore, a submission may send data that is not perceptibly represented on the page. This data is usually either a superset or subset of the data displayed on the page, or additional information, e.g. about how the page is rendered to the output devices such as the screen. In order to model changes of the data view, the page is given additional rendering information about the way the data should be represented, and possibly it is given more information than will actually be perceptible on the page. For instance, a table view might only display a subset of the columns in a table, depending on which ones it is configured to show, and a tree view will display only the root nodes of subtrees that have been collapsed by elision. Once the rendering parameters are changed, the data is sent to an action, where it is possibly transformed, e.g. sorted in a different way, and rendered on a page again, probably the same one it was rendered on before. Also in the case of ephemeral side-effects, additional hidden data is used to keep track of the ephemeral changes. This hidden data might, for example, be the content of a clipboard.

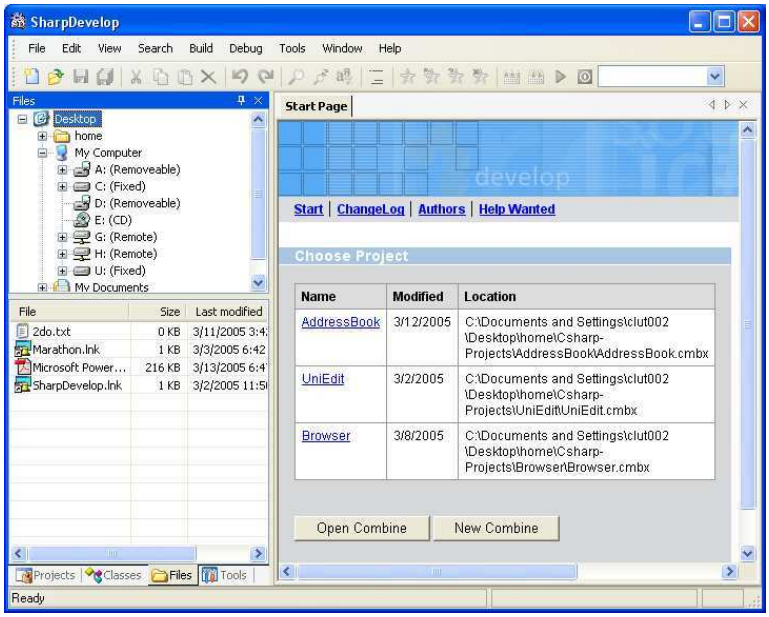

Figure 5.8: Start page of the SharpDevelop IDE.

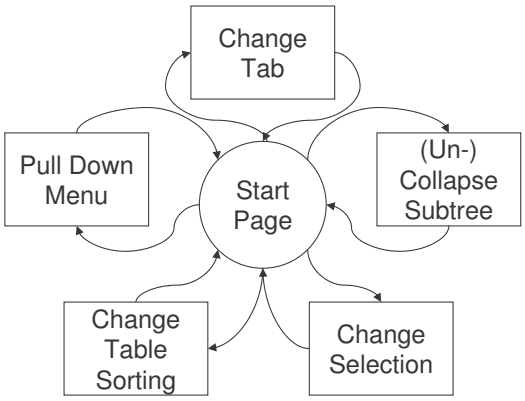

Figure 5.9: Fine-grained interaction formchart model.

In order to illustrate the use of the form-oriented model for fine-grained interaction, let us consider how this kind of interaction could be modeled for a considerably complex user interface. Figure 5.8 shows the start page of an integrated development environment (IDE), a program for the creation of software. It contains mainly a menu bar and a tool bar at the top, a tab control with different tabs on the left, and a panel with a list of existent development projects on the right. On the currently open tab there is a tree view, which shows the local computer's file system hierarchy, above a table view, which lists the files in one of the file system's folders. Such a user interface could be modeled in arbitrary detail, but the overall structure of the model would look like Fig. 5.9. The different actions change the data sent back to the page in a way that a pull-down menu is shown, a different tab is opened, a subtree is collapsed or unrolled, a different file selected, or the table sorting changed.

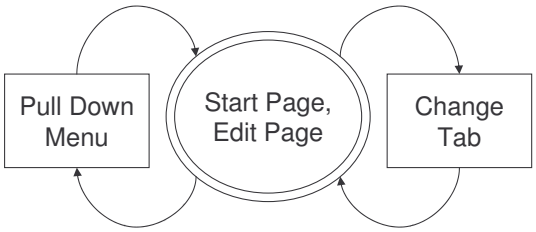

Figure 5.10: Modeling crosscutting concerns of the UI with state sets.

Many possibilities of fine-grained interaction are usually accessible on multiple pages of the system. For example, in the case of the IDE there is a page for editing source code that offers the possibilities to pull down a menu and choose between tabs, just like the start page. In many systems parts of the user interface are available at many different places, and therefore crosscut the system's overall structure. Attaching the corresponding actions to every single page would not only mess up the model but fall short of the fact that these parts are always the same, which would compromise maintainability. Therefore, form-oriented analysis offers a model entity called a *state set* that allows the modeler to attach model parts to a whole set of pages. Figure 5.10 shows how to model the common parts of the fine-grained interaction on the start and edit page.

Another question is how to separate the model for fine-grained interaction from the rest of the model, so that we are not unnecessarily confronted with too much details all the time. This problem can conveniently be solved by decomposition of form-oriented models. With this method different parts of the user interface can be modeled separately, with some parts possibly appearing in multiple submodels, and the different submodels can be arbitrarily merged.

5.4.2 Coarse-Grained Interaction with User Transactions

The characteristic property of coarse-grained interaction is that it can cause the persistent data of the system to change. When dividing interaction into the fine-grained and coarse-grained categories, one of the deciding factors is whether the respective action should cause an ephemeral or a persistent change. In most cases of interaction, such as the ones depicted in Fig. 5.1, the decision is a natural one.

Every action that relies on more than trivial user input and naturally involves a change in the data should be modeled as coarse-grained interaction because, as I will discuss now, this does not only prevent data loss but also offers other added value. When fine-grained interaction is interrupted by some system malfunction, e.g. a loss of power, all ephemeral data will be lost. However, it will be possible to restore the persistent state of the system, so that the overall loss is insignificant.

For coarse-grained interaction I use another mechanism to make the system fail-operational, i.e. capable of recovering and functioning after a failure. I give it the name *user transactions*. Every operation that is performed on the persistent data within a single submit/response-cycle is atomic. Naturally, in the implementation this will correspond to the concept of a database transaction, which means that the operations are guaranteed to adhere to the ACID principle [114]. Among other things, what will happen is that either all operations are successfully completed, or none, so that the persistent data will never be left in a half-unfinished corrupted state. When coarse-grained interaction is interrupted, the data of the form that triggered the corresponding action will be lost. However, the integrity of the persistent data will be unaffected. All of today's relational database systems come with a built-in capability for transaction processing, so that this feature can be used with no additional effort.

Another very desirable feature that can be implemented with transactions at some additional cost is version control. A common feature in modern content creation programs is an undo/redo function, which allows the end-user to revoke actions that have been performed before. However, this feature is usually just implemented as ephemeral side-effect, and during a system malfunction this information is lost. Version control makes the history of coarse-grained changes in the system persistent, thereby allowing to search for and reuse any old version of the content that is edited. It also enables the user to manage different versions and keep track of content evolution, which is an absolute necessity in larger projects.

Relational database systems are usually not only capable of transactions, but also capable of concurrency control. In distributed systems, where transactions can be invoked from any place that is connected to the network, this is necessary to avoid interference of transactions that are executed concurrently. This means that if coarse-grained interaction is modeled using the paradigm of user transactions, the system will automatically be able to handle multiple remote users at the same time. At no additional cost we get a distributed multi-user interface system.

5.5 Configuration vs. Construction

When talking about the process of content creation, I distinguish two different views of what is actually done. Traditionally the process of content creation starts out with nothing or a small amount of data. This data is increasingly extended over time, but there may also be phases in which the amount of data decreases. However, the content may also, deliberately or accidentally, be perturbed, rendered invalid or become altogether corrupted during creation. For example, parts of data may be fitted together in a way that does not make up a complete model yet. I call such content and the way it is created

a *construction*.

Using my terminology, constructions are not based on a robust user interface model. The fact that a state of inconsistency can be reached is opposed to my notion of robustness. Therefore I propose a different view on content creation: *configuration*. In the paradigm of configuration, every feasible content is a point in a multidimensional, possibly infinitely dimensional space – the *configuration space*. Each dimension of that space is a possibly infinite semantic domain, containing values and a special value representing an undefined state. In the process of creation, we start out with the point that is undefined in all dimensions, and transform that point into another point that contains either more or the same amount of information as the point before. The only exception to this is when the end-user deliberately deletes parts of the content. But even then, if we assumed that a deletion would create a new version of the content and that all prior versions are part of the configuration, the aforementioned property would still hold. Each single point, as well as the process itself, is called configuration. The sequence of points that the system passes through during configuration is called *configuration history* and can formally be understood as a chain in a complete partial order, starting at the bottom element. The fact that all points in the configuration space represent consistent content, together with the fact that we can never leave this space, makes sure that a user interface based on that paradigm must be inherently a robust one.

5.6 Conclusion

In this chapter I introduced different ways and methods to make user interfaces of content creation systems more robust. I discussed issues of data modeling that are of immediate significance to the user interface, how the form-oriented methodology can be applied, and how it can be used to model fine-grained and coarse-grained interaction. Eventually, I proposed a new paradigm for the creation of content – configuration – that can serve as a formal basis for the creation of robust user interface models.

All these concepts can be applied to content creation as it is done in software engineering. The artifacts of software projects are valuable assets, and therefore robustness is particularly important. As described in Chapt. 7, I have successfully applied concepts for robustness in my CASE tool prototype, the generic editor. My experience with the generic editor has shown that these concepts are not only practically applicable, but also very useful.

6

Reflection as a Principle for Better Usability

This chapter explores the principle of reflection and its relation to HCI. I define reflection in a wider sense that can be applied to the world of user interfaces, and argue that the new, generalized notion of reflection can benefit the usability of a system significantly. The chapter discusses concrete approaches for the design of reflective user interfaces, and shows that the reflection principle is in fact already used in many existing applications.

The concepts presented in this chapter are generic, and can be applied to CASE tools. They aim at improving the usability of a system, which is a very important issue for CASE technology. As a result, this chapter is of direct relevance for the presentation aspect of the AP1 system. The generic editor, which is described in Chapt. 7, makes heavy use of the reflection principle.

In the context of computer science, reflection is known as a principle from the domain of programming languages. In its most common form, it enables a program to get information about itself and its runtime environment. With this information a program can, for example, adapt itself to new data structures or changes in the runtime environment. However, the principle of reflection is not just a programming concept. Although it has been discussed systematically in the programming domain, this is just one of its applications. In this chapter I show that reflection is applicable to and relevant for the domain of user interface design and implementation as well.

Reflection in user interfaces is in fact ubiquitous. It is intrinsically related to important functions of a program, such as functionality for help or features for program customization, and therefore affects its usability. The reflection principle can be used as a lens to gain a better understanding of user interfaces, and inspire new ways for improving them.

The notion of reflection in user interfaces is described in Sect. 6.1. In this section I discuss concepts and terminology that were elaborated in the context of programming languages, and use them in order to create a taxonomy for features found in user interfaces. To the best of my knowledge such an analogy has not been described before. Section 6.2 describes different approaches that can be used in order to create reflective user interfaces. Many of them are well-known but have never been discussed from this point of view. Section 6.3 describes some examples of reflective features in the user interfaces of relatively common applications, and shows that – although this has never been stated explicitly – there are in fact many reflective features in existing applications. Section 6.4 discusses related research, and Section 6.5 concludes the chapter. Parts of this chapter were published in [145].

6.1 Reflection and HCI

Reflection is common in many programming languages and can provide a high degree of flexibility during the runtime of a program. For an overview of reflection in different programming paradigms, see [62]. Reflection in a programming language refers to the capabilities of a program to read and modify information about itself or its runtime environment. This information is called *metadata*. On a HCI level, I refer to reflection as the ability of a user interface to represent and support the modification of itself and its application. The metadata is the information about the user interface or the application that is represented or modified.

Commonly two different kinds of reflection are distinguished: *structural* and *behavioral* reflection. Furthermore, each kind of reflection can be split up into two operations: *introspection* and *intercession*. In the following sections I describe these different aspects, which are summarized in the table of Fig. 6.1, and what they mean on the level of HCI.

6.1.1 Structural Introspection

In the context of programming languages, structural introspection means that a program can read information about its own structure, i.e. about its data structures and its program code. For a user interface it usually means that it presents not only application data but also information about the structure that this data have or can have, i.e. metadata. This provides insight to a user as to what data can be processed by an application and how the

	Introspection	**Intercession**
Structural	Representation of data structures & implementation	Modification of data structures & implementation
Behavioral	Representation of information about the UI	Modification of the UI

Figure 6.1: The different aspects of reflection in a user interface.

different elements of that data relate to each other. A data management application, for example, could help users find the information they need by revealing the data schema of its database. Although less common, structural introspection can also mean that information about a program's implementation is revealed in the user interface. For example, a system might reveal to the user that its functionality depends on some other program, which has to be installed first. This can help a user understand technical problems in case the application is not working correctly.

6.1.2 Structural Intercession

Structural intercession in a programming language means that types and/or program code can be modified by a program. For a user interface this usually means that it supports modification of structural information about the application's data; i.e. metadata about data structures can be changed. This enables users to adapt the structure of the application data to their needs. For example, templates of any kind fall into this category: they describe common aspects of a group of data instances and thus determine their structure. If we can use templates in a text processing application, it is much easier to make sure that documents are consistently structured or that a particular visual design is preserved. Structural intercession can also relate to the implementation of an application. This means that the user interface supports to some degree modification or configuration of the internal functionality. For example, a user interface might offer functionality for extending the system with plug-ins, so that the user can extend or reduce the functionality offered by the application.

6.1.3 Behavioral Introspection

Behavioral introspection in programming languages means that it is possible to read information about the behavior of the runtime environment. For example, it is possible for a program to look into the code of the interpreter, i.e. the abstract machine, that executes it. I use this term if a user interface offers users the possibility to acquire

information about how the system behaves towards the user. In other words, it means that the user interface reveals information about itself. For example, it might show what happens when a particular button is pressed, or how a particular setting of a control affects the system behavior. A system with behavioral introspection has a user interface that is in a way self-explanatory because it grants the user a look under the hood of the system into the underlying mechanisms. Its semantics are made perceivable for the end-user.

6.1.4 Behavioral Intercession

In programming languages, behavioral intercession means that you can change the behavior of the runtime environment, i.e. the way the runtime environment behaves towards a program. I use this term if a user interface makes it possible to change the way a program behaves towards the user. In other words, the user interface of a program can be configured. This is very important, for example, for professional users who need to tailor the user interface to their professional environment in order to maximize productivity, or for better accessibility of an application. An application might allow users to change the controls available on the user interface, or use alternative input and output devices.

6.2 Approaches for Reflection in User Interfaces

This section discusses approaches for creating user interfaces with reflection capabilities. Some approaches are well-known and some are new. None of the approaches have been discussed in the context of reflection before.

6.2.1 Generic User Interfaces

Many applications are able to process different types of data. Most popular text processing applications, for example, allow the user to edit documents of different formats, such as ASCII text, ODF or HTML. This is possible because these applications use an underlying data model that is flexible enough to deal with the particularities of the different data types. Whereas the user interface generally stays the same, such an application informs the user about the characteristics of a data type in some way or other, thus providing some degree of structural introspection. A text processing application, for example, will not provide the same functionality for ASCII text documents as for HTML. Some applications provide more structural reflection by allowing the user to import and use data of a previously unknown data type. Some spreadsheet applications, for example, are able to import data from structured text files with different formatting or delimiting characters. An even higher degree of structural reflection is provided when an application allows

the user to define and use completely new data types. This is the case, for example, in some desktop databases. The key element in all these applications is that they provide a generic user interface for several, potentially very different, data types. Internally this requires a flexible data model, and may also involve structural reflection of data types by the program itself.

Another common way of implementing generic user interfaces is to apply the "don't ask what kind" principle. This means that code is able to process data of different types by making use of dynamic binding. In object-oriented programming languages this is done with method polymorphism, whereas in other languages function hooks are used. It is commonly used in component technologies such as Microsoft's Object Linking and Embedding (OLE), which allow data of one application to be embedded and edited in others without them knowing about each other. For example, it is possible to embed a spreadsheet into a text document and change the spreadsheet from within a text processor. This does not only allow applications to provide a higher degree of structural introspection into different data types, but also enables a higher degree of behavioral intercession by providing new means for the extension and configuration of the user interface.

6.2.2 Metadata Integration

In some cases the reflective parts of a user interface, i.e. those that deal with metadata, can be integrated with those that support data operations. While the data of an application serve as its productive input and output, its metadata may describe what data types exist and how they are structured. Often the metadata describes commonalities of data instances, which can be important for preserving the integrity of a system, or just useful for information reuse. For a text processor, for example, such metadata would typically include the page size setting, the default font and page formatting. The user interface could allow a user to set metadata globally, i.e. for all data instances, or with more sophisticated mechanisms. Many applications support a notion of templates, which can be used to set metadata for groups of data instances. In many applications the part of the user interface responsible for data is separate from the part allowing metadata access, as pointed out in the left part of Fig. 6.2.

In the world of programming languages metadata and data are usually treated in the same way. Metadata is just special kind of data, represented with the same data model and accessed with the same operations. Sometimes an analogous approach is possible in a user interface: data and metadata can be represented in the same user interface and possibly even modified with the same functions. I call this metadata integration because the user interface for metadata is integrated with the user interface for data, which can be realized elegantly if the internal representations of data and metadata are integrated as well. This is depicted in the right part of Fig. 6.2.

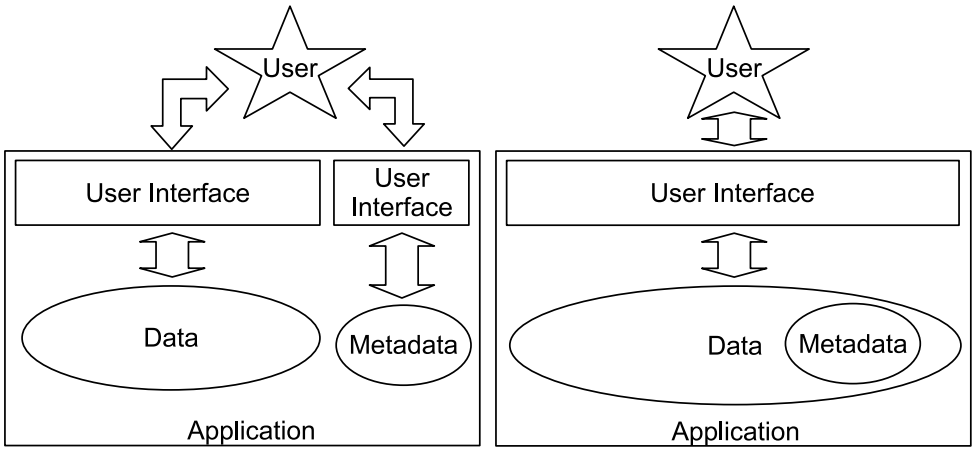

Figure 6.2: A system with a non-reflective (left) and a system with a reflective user interface (right).

A common problem with reflection in programming languages is known as *meta-confusion*, which means that the different data and metadata levels that might exist are easily confused. We have to be careful to avoid the same problem for metadata integration in the user interface: if it is not immediately clear if a user is modifying data or metadata, i.e. a simple data element or a metadata element that might change the way the application behaves, this will lead to mistakes. Metadata integration may enhance accessibility and result in a coherent internal design, but metadata and data should be distinguished clearly in order to prevent a loss of clarity.

6.2.3 Plug-in Architectures

A certain degree of reflection can be achieved by using an architecture that can be extended by plug-ins. Insertion and removal of plug-ins is often supported in the user interface so that it can be done by the end-user. Sometimes plug-ins just add internal functionality to a system, which results in structural intercession, but sometimes they also extend or modify the user interface, resulting in behavioral intercession. The ability of a program to let the user see what plug-ins are installed, and possibly how they are configured, enables a degree of introspection. An example of such a plug-in architecture is the Eclipse IDE.

6.2.4 Direct Data Access

A different approach for reflection is possible if the way data is handled within an application comes very close to the way end-users should be able to handle the data. Instead of creating a new layer of functionality for the end-user, we can provide a front-end for the existing data structures and operations. This can produce very elegant, flexible and minimalistic designs and expressive user interfaces. For example, such an application

can leverage the reflection capabilities of its runtime environment, thus reusing a great deal of functionality. A data management application that should be able to let the user use new, formerly unknown data types, written in a programming language that supports reflection, could use existing features for data introspection, dynamic loading, and introspective access. Reflective capabilities of the user interface could thus be directly delegated to reflection in the implementation. A direct data access architecture is, for example, the naked objects approach [187].

6.2.5 Document-Oriented User Interfaces

Document orientation [74] is a concept that naturally fits the idea of reflective user interfaces as illustrated in Fig. 6.2. It has much in common with the concept of direct data access, specifically with regard to metadata. In conventional applications, metadata is often presented in auxiliary dialogues, such as the print dialogue. One of the disadvantages of such dialogues is that the lifespan and scope of their data is usually not evident to the user: the data in the dialogue may be only transient, e.g. exist only for the duration of a particular operation, or might be permanent and reappear if the same dialogue is opened again. If the data is permanent, it might not be global for the application but bound to one of its documents. Print settings, for example, might just be valid for a particular document.

The document-oriented approach is able to provide the user with an intuitive understanding of lifespan and scope by using the document metaphor instead of ad hoc auxiliary dialogues. The idea is to make metadata part of a document instead of storing it at an indefinite location. This change does not necessarily affect the user interface visually, but clarifies its semantics: the lifespan and scope of the metadata corresponds to that of the document in which it is embedded. Through metadata integration and direct data access, the metadata in a document can be accessed in the same consistent manner as other document data. This eliminates the need for software developers to create separate user interfaces for such data.

Differences in the scope of metadata can be expressed by using a suitable decomposition mechanism. For large documents, decomposition is necessary anyway, and therefore such a mechanism is usually already in place. In text processing applications, for example, a common feature allows it to store a complex document as a collection of several documents, with one of the documents usually functioning as hub. This main document contains links to all the other documents. Such a mechanism allows for reuse of data: a document containing frequently needed data can be linked into all the places in other documents where this data is needed. Similarly, a document containing metadata can be linked into all the places where the metadata should be used, thus creating an appropriate scope for that metadata. This approach not only reuses existing decomposition

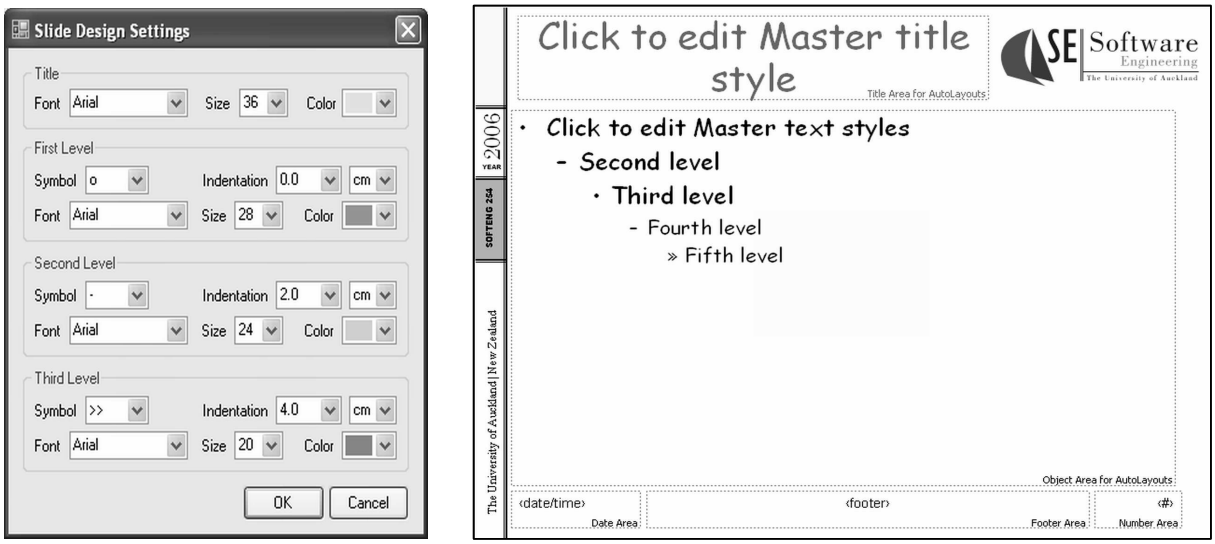

Figure 6.3: Auxiliary dialogue and master slide in a slideshow editor.

mechanisms, but also allows for more end-user control.

6.2.6 Templates and Master Instances

One choice for reflective user interfaces that follows the document-oriented approach and also supports the intuitive use of reflection is the usage of master instances, such as master slides in slideshow editors. Master instances are different from style sheets in that a master instance resembles an ordinary part of a document. In a slideshow editor, for example, they look much like other slides and are therefore instances. A master instance can be formatted like an ordinary part of the document, but any formatting operation on the master instance is automatically applied to all parts of documents that are based on it.

This approach to present metadata has become standard in slideshow editors. These editors offer a master slide view as shown on the right side of Fig. 6.3. The master slide has many properties of ordinary slides. However, formatting the master slide has immediate effects on all slides that are based on it. It would be conceivable to represent the same metadata in an auxiliary dialogue such as the one depicted on the left side of Fig. 6.3, but then the advantages of a reflective user interface would be lost. The master slide concept integrates data and metadata on a presentation level, allowing application of the same operations. The WYSIWIG capabilities of the normal slide view are reused, providing a much more direct feedback to the user than the numerical values in the auxiliary dialogue.

In common slideshow editors access to the master slide is rather hidden, in one of the pull-down menus. Furthermore, a master slide is associated with a single file. The analogy to normal slides could be furthered by storing and managing master slides like normal ones. This would make it much easier to handle master slides, and to utilize decomposition mechanisms as pointed out in Sect. 6.2.5. For example, the style of many

sets of slides could be maintained by linking each of them to a common master slide document.

The template concept is similar to that of master instances. It is also used to express the commonalities of a set of instances. The emphasis here lies usually on static parts that appear in all instances, with changes at well-defined places in between, as in a cloze. The template concept is applied, for example, in web page generators, such as Java Server Pages (JSP) and Active Server Pages (ASP). If a JSP generates webpages containing a table, then there will usually be a table in the JSP as well, although potentially not with all the rows that may appear on the generated web pages [67]. As a consequence, JSPs can to a degree be edited like ordinary web pages. Templates are also commonly used for generative programming [54].

6.3 Examples

This section discusses what role reflection plays in existing technologies. We will see that reflection is used in several systems, although the underlying principle has never been discussed explicitly. I will also describe cases where reflection is absent but could be applied to improve the usability of a system.

6.3.1 Unix-Style Operating Systems

Unix-style operating systems are known for a motto which says that "everything is a file". This is true for system entities like console connections, connected hardware devices, some configuration settings and, of course, all ordinary data. The structure of the system can be introspected and certain settings can be changed, thus this approach is an example of structural introspection and intercession. While most files contain ordinary data, others contain information about internal system entities, i.e. metadata. The metadata can be accessed and processed in the same manner as ordinary data files, e.g. using the same set of command line tools. With the file system being the user interface, data and metadata are handled similarly and thus a degree of metadata integration is achieved.

The idea of the file system as the operating system's main user interface is taken a step further in the Unix successor, the Plan 9 operating system from Bell labs [182]: even more system entities are represented as files, including windows, processes, and almost anything else available in the operating system. This is an example of how a homogeneous, integrated user interface for data and metadata can yield a slim and elegant design, that can facilitate the system's usage: instead of having to use a specialized interface when accessing a system entity such as a process, a user can do the access through the file system using the normal file system commands. However, like in programming languages,

the principle of reflection is still used on a rather technical level. The file system approach provides by itself rather an interface for developers than for end-users. But such a reflective interface on a lower level can form the basis for a reflective interface on a higher level: a direct data access user interface layer on top of the file system could use the same reflective metaphor and thus provide the same advantages, e.g. allow users to do many tasks homogeneously through the file system using an appropriate GUI.

6.3.2 MS Windows

In the following I examine the user interface for some typical system configuration tasks in the Windows operating system. A practical example for achieving behavioral introspection is a function for help. This gives users the possibility to look up parts of the user interface that they do not understand, and hence gain knowledge about system behavior. In order to enhance usability, a help function can be context-sensitive and offer even very small pieces of information, e.g. the tool tip labels that appear when the mouse button is hovered over certain parts of some UIs.

There are also other ways of representing the underlying mechanisms and concepts of a system, which are less canonical and may require some creativity and innovation. In graphical desktop environments like MacOS and MS Windows, for example, windows can be minimized into icons in a task bar, and this is visualized by an animation that shows the window shrinking into its icon on the screen. While this may be superfluous eye-candy for computer savvy users, it suggests to unexperienced users that the minimized window is not gone and where it can be found. A similar effect is applied for window maximization.

The use of a registry based on a directory service is an example of structural reflection. The metadata of many programs is presented in a single interface. This instance of a reflective user interface is a replacement for setting environment variables in a traditional batch file that is executed on startup. The registry makes it possible to look up particular settings in a central place, and navigate metadata in a structured manner. In contrast to this, batch files become much harder to handle with a growing number of settings due to their unstructured ad hoc nature.

Auxiliary settings dialogues, such as the ones in Fig. 6.4, are similar to the settings in the registry. They usually exhibit a similar structure, but usually operate on the level of a single application instead of the whole system. A reflective user interface would encourage a uniform representation for such settings, such as the tree view on the right side of Fig. 6.4. With such a uniform representation, settings can be treated rather like documents instead of having to create ad hoc dialogues. Tabbed panel approaches such as the one on the left side of Fig. 6.4 require a rather hand-tailored user interface. They can lead to strange imbalances if the number and size of settings is very different from rubric to rubric, leading to some nearly empty and some overfull tabs.

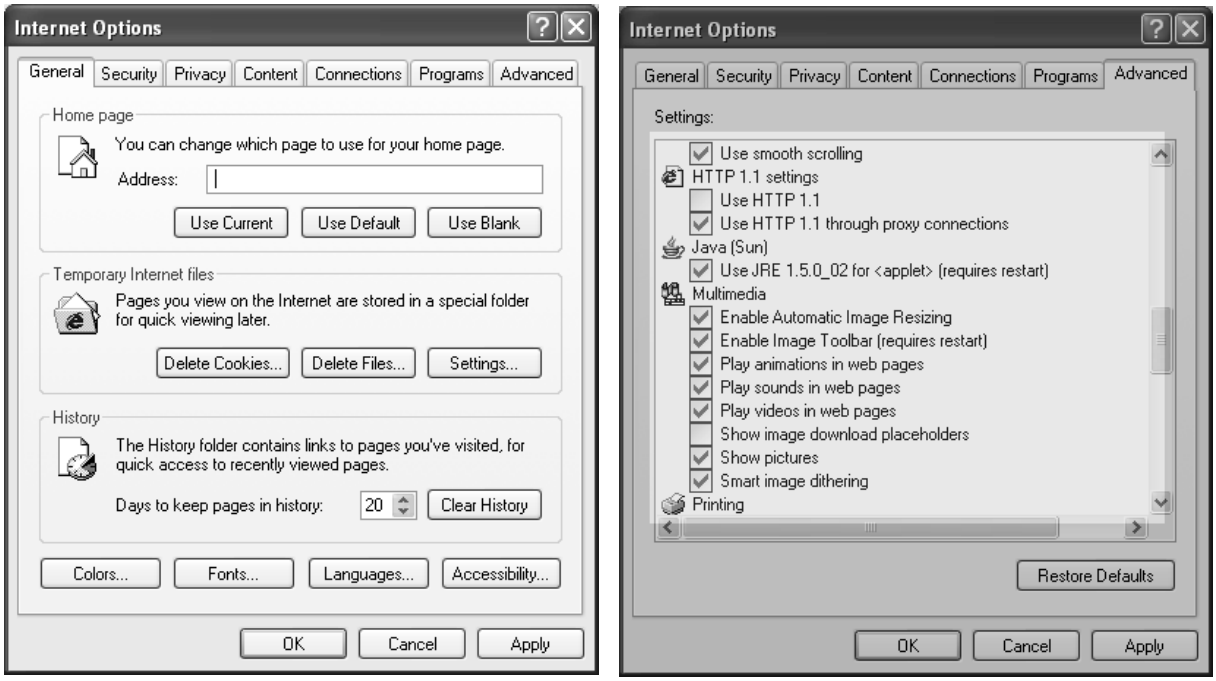

Figure 6.4: Two different ways to present advanced settings.

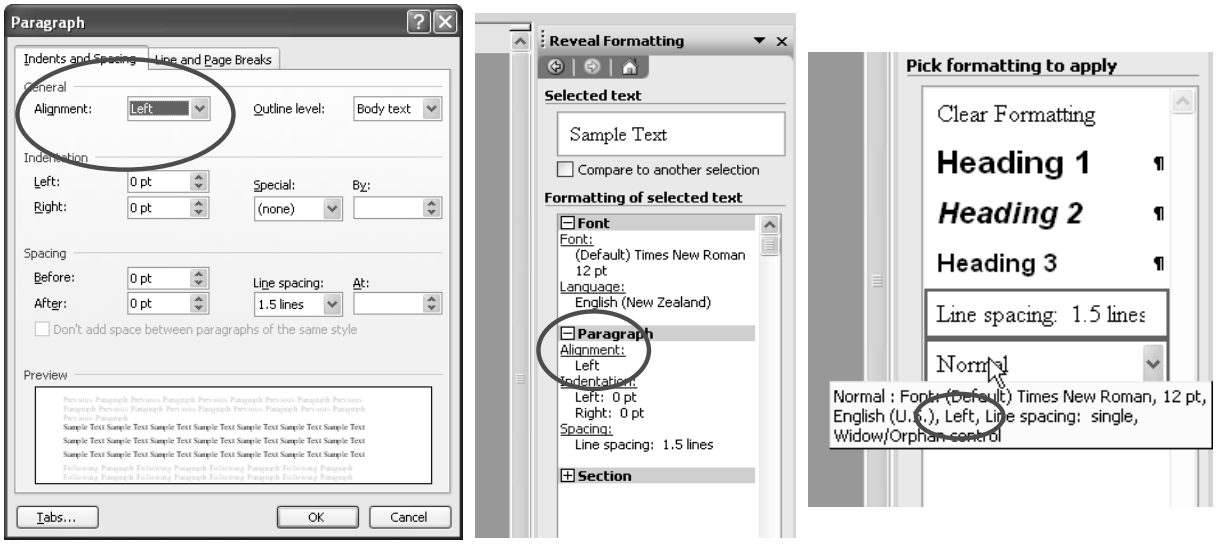

Figure 6.5: The same metadata is presented in several different ways

6.3.3 Office Applications

In many applications, there are several different ways in which the same metadata is presented to the user. For example, in a current version of the popular text processing system MS Word, the font size used for a paragraph can be set in at least three different ways, as illustrated in Fig. 6.5. This heterogeneity is not only redundant but can also be confusing for a casual user, who might not always be aware of how the different ways of changing something relate to one another. To add to this complexity, in the case of MS Word the user interface tends to change between different versions. Reflective user

interface approaches encourage a clean mapping to underlying operations, resulting in more concise user interfaces.

In contrast to slideshow editors, the concept of a master instance is less established for text processors such as MS Word. It is possible to define invariants for a document, e.g. settings for certain paragraph types, but the user interface for such settings is separated from the documents. The user interface parts in Fig. 6.5 are rather similar to the left part of Fig. 6.3 than to the right part, where a master instance is used. However, the master instance approach would be conceivable for word processors. In the master instance document parts such as sections with titles and paragraphs could correspond to document part types. Changes of those parts would result in immediate corresponding changes of the documents that use the master instance.

Many desktop applications allow a user to configure the tool bars, i.e. the panels with buttons for common functions, by adding/removing buttons and moving a tool bar to different locations. Another example is the possibility to define shortcut key combinations for functions that a user wants to be accessible from the keyboard. Other examples for behavioral intercession include the setting of big fonts or buttons for better accessibility on the screen, changing the order, size and location of different panels, and configuration of the dialogues that a system offers.

6.3.4 Meta-CASE Tools

Meta-CASE tools, e.g. the one described in [228], support configuration and generation of specialized visual editors for 2D graph-like diagram types. Because they usually aim at being as generic as possible, i.e. being able to generate many different such editing tools, they have to explicitly deal with metadata that describes the structure of a diagram, its appearance, and the behavior of the editor. A user has to define a model for a diagram type, and then specify how each of the model elements are visualized. Providing both information about models and functionality for modifying it means that the user interface of a meta-CASE tool naturally supports structural reflection. Being able to observe and change the editing functionality that is provided for a generated diagram editor means that behavioral reflection is supported as well.

To illustrate the idea of meta-CASE tools, Fig. 6.6 shows a simple model for state machine diagrams and its relation to a visual representation. The model specifies types for states, transitions and labels on transitions, which are shown as filled circles. Associations between the types are shown as connections between the circles: each transition refers to one source and one target state, and has one label. The graphical representation of a state is a circle, and that of a transition an arrow between the circles of its source and target states. The label of a transition is placed on its corresponding arrow. In the same manner many different diagram types can be defined. Tools that can be generated

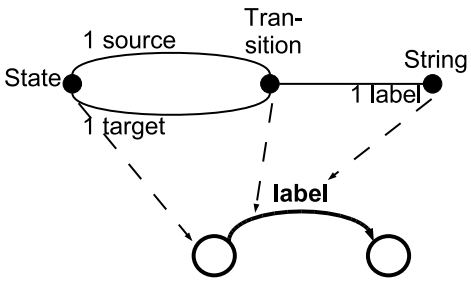

Figure 6.6: Model (top) and graphical representation (bottom) of a state machine diagram.

with such specifications are, for example, editors for data models, process diagrams and electronic circuits.

6.4 Related Work

Reflection, in one way or other, has also been described in other work as an important user interface concept, although not as explicitly as I do in this chapter. For example, ISO 9241-10 [115] names self-descriptiveness, suitability for learning and suitability for individualization as general principles for achieving the ergonomic requirements of user interfaces. Self-descriptiveness is, in fact, a direct result of behavioral introspection, and related to ease of use for the casual user and the difficulty of learning. About the suitability of applications for learning is said that "rules and underlying concepts which are useful for learning should be made available to the user" so that "the user is able to obtain information on the model on which the application is based". This kind of transparency in a system is achieved by structural introspection. Suitability for individualization is directly related to structural and behavioral intercession.

One of the criteria for performance-centered user interfaces [152] is to "provide performers easy access to and control of desktop support components, interface presentation, and functions". Access to interface presentation and functions refers to behavioral reflection, while desktop support components include different kinds of help system. Help functionality in a program, e.g. as described in [105, 149], is a typical example of behavioral introspection, since its purpose is to explain to the user how to interact with the system in order to accomplish certain tasks. Different approaches for help generation have been described that can provide behavioral introspection in the user interface by performing behavioral introspection on an internal model of the system [205, 34, 176, 159]. Dutke and Reimer [81] indicate that help systems that explain the functions of a program result in a better learning performance for users than help systems that merely provide a list of actions to perform. The former approach, i.e. offering insight about the system to the user, is the very heart of introspection.

Adaptability and extensibility are important for user interfaces as well. Approaches

for adaptability have been discussed for web-based systems, e.g. [89], and graphical user interfaces, e.g. [74], and are examples of behavioral intercession. Grundy and Hosking [101] describe an approach for adaptability and extensibility of user interfaces that is based on metadata about UI components and related to reflection in programming languages. Those problems have also been addressed in toolkits like the ones described in [16, 134] and user interface management systems, e.g. [183].

The instrumental interaction model [14, 15] reflects concepts and functions of an application as instruments, which are first-class objects in the user interface. It relies on design principles that are inspired by programming languages, with one of them being reification, i.e. the representation of metainformation as ordinary data. Consequently, this methodology targets the creation of reflective user interfaces. One of three desired properties of user interfaces that are mentioned is reinterpretability, meaning that "users can change input/output devices, add or remove interaction techniques, even program their own functions", which is pursued by making interactions first-class objects of an application. This does, in fact, describe the principle of behavioral intercession.

Some of the usability heuristics described in [163] are related to reflection as well: "visibility of system status" can be achieved by introspection. "Recognition rather than recall" is supported when metainformation is visible in the user interface, and this information can be helpful in order to "help users recognize, diagnose, and recover from errors". "Flexibility and efficiency of use" is supported with behavioral intercession, and "help and documentation" is provided through behavioral introspection.

6.5 Conclusion

In this chapter I described how the principle of reflection can benefit the domain of user interfaces. The different aspects of reflection form a taxonomy for many application features that strongly affect usability. I have discussed several approaches to reflection in user interfaces and have examined user interfaces for reflective features. Examples show that these concepts can be found in and are relevant for real applications. To the best of my knowledge the reflection principle has never been systematically applied to the domain of user interfaces before.

As described in Sect. 1.3.5, CASE tools face many usability challenges. In particular, learnability and customizability are problematic issues. Reflection in the user interface is a principled approach that can improve the transparency of a system for the user, and help the user to adapt a system to individual needs. The generic editor, which is described in Chapt. 7, has a highly reflective user interface. Experience with the generic editor has shown that reflection, when applied systematically, can significantly reduce the complexity of the user interface.

7

The Generic Editor

The generic editor is the default user interface to the repository. As its name suggests, it is a generic data manipulation tool that can be used to edit any PDM data in the repository. It provides configurable views that are suitable for the representation of most data, and access to all operations (see Sect. 3.6). As a result, it supports elementary editing, i.e. the insertion and removal of individual links, as well as more sophisticated tasks that were implemented in operations, such as the code generation described in Chapt. 8. All tasks on repository data can potentially be carried out just using the generic editor, although a more specialized tool might be able to perform a particular task more efficiently.

In order to support integration of CASE tools, the generic editor is designed with an open plug-in architecture that allows it to be extended with new views. A view is a modular GUI component that can be used in the context of the generic editor's user interface. Views can be used to represent data in a particular manner and offer new methods of interaction.

The basic idea behind the generic editor is similar to IDEs such as Visual Studio and Eclipse. However, the generic editor is a research vehicle for novel concepts, such as the concepts for robustness and reflection described in Chapts. 5 and 6. These new concepts are prototypically implemented from the ground up. As a result, undesirable interactions with old concepts are avoided, so that I am able to explore the new concepts in their pure form. It does not imply that the concepts of existing IDEs are bad, or that my concepts are better. It is a necessary precondition for my research work.

Section 7.1 specifies some of the general requirements of the generic editor. Section 7.2 gives an overview of its architecture. Section 7.3 describes the editor's workbench, and Sect. 7.4 its views. Section 7.5 outlines how the editor can be customized, and Sect. 7.6 how it can be used for collaborative work. Parts of this chapter have been published in [142].

7.1 Requirements

I have chosen the the following main requirements for the generic editor. They specify what the editor should do, and elucidate at the same time why the editor is a necessary component at all. They are directly motivated by the problems of CASE technology described in Sect. 1.3.5, and the general requirements for the user interfaces of IDEs such as Visual Studio and Eclipse.

Universality Any operation that can be done on the repository by a CASE tool could alternatively be done manually using the generic editor. The basic functions of the generic editor provide, in a sense, a complete minimal user interface because they support elementary editing, i.e. the insertion and removal of links. As pointed out in Sect. 4.2, all changes in the repository can be reduced to sequences of these two link operations. Naturally, developers will use more specialized tools for particular tasks, but there is always the option to fall back to a manual process using the core functionality of the generic editor. This is particularly important since there are only few specialized tools for AP1 at this stage. Furthermore, it allows me to use the generic editor to test all the functions of the repository.

Customizability It should be possible to customize the different components that make up the generic editor, such as views, to individual needs. It should also be possible to extend the generic editor with new ways of representing and interacting with data. This requirement appears particularly important regarding the large variety of software development domains and development practices. The different domains each benefit from a particular set of data representations and operations. In order to accommodate that variety, the generic editor needs to implement concepts for configurability and extensibility, such as the concepts for reflection in the user interface that were presented in Chapt. 6. For example, a plug-in architecture has been identified as a powerful and common means for achieving extensibility.

Integration While the repository was designed to support data integration, the generic editor should support integration of CASE technology on the level of the user interface. This is commonly referred to as presentation integration. Besides requiring the generic editor to be extensible, it means that different components can be formed

into a cohesive whole. Not only should the user interfaces of separate tool components be displayed in the same GUI, but they should ideally also adhere to the same HCI principles. If the parts that make up the user interface vary too much, it easily becomes cluttered and unmanageable. Chapters 5 and 6 have identified unifying HCI principles, and these principles are used in my prototype to address this requirement.

Usability The generic editor should be easy to understand and efficient to use. As discussed in Chapt. 6, reflection helps to achieve this by improving transparency. Robustness, as proposed in Chapt. 5, helps to avoid application errors. My strategy is to use these concepts in the design of the generic editor whenever possible. Eventually, the aim is to achieve a usability for model-based editing of data such as source code that is better than that of text-based approaches.

The requirements in this list are general, and can thus be very complex. They are not entirely orthogonal, nor is this list exhaustive. The general nature of the requirements is inevitable because of the general function that the generic editor fulfills within the AP1 system. The generic editor is not a ready-made CASE tool; rather it is a reusable base of features to satisfy common needs. In order to give my design efforts a more specific direction, my strategy is to address the requirements by implementing the concepts of Chapts. 6 and 5. I will refer back to the requirements during the chapter.

7.2 Overview

The generic editor can be seen as an open IDE or workbench, with an extensible plug-in architecture similar to that of popular IDEs such as Eclipse [58] or MS Visual Studio [198]. Like the IDEs, it provides a core of common features, and relies on plug-ins for more specialized functionality. This core is designed to encourage reuse and foster integration. The main function of the generic editor is to provide multiple synchronized and possibly updatable views onto the data in the repository. Therefore, the reader is advised to become familiar with the concepts explained in Chapt. 3 before reading on.

Figure 7.1 shows a rough architectural overview of the generic editor. The editor is based on the repository and can be extended by plug-ins to support specialized functionality. This example shows three plug-ins for graph handling: an adjacency matrix view plug-in, a plug-in that produces a 2D layout view of a graph, and an operation plug-in that generates a minimum spanning tree (MST) for a given graph. As I will explain in Sect. 7.4, views are actually implemented with the help of operations. As a result, operations form the core concept of the generic editor's plug-in architecture.

All of these plug-ins reuse functionality already implemented by the generic editor,

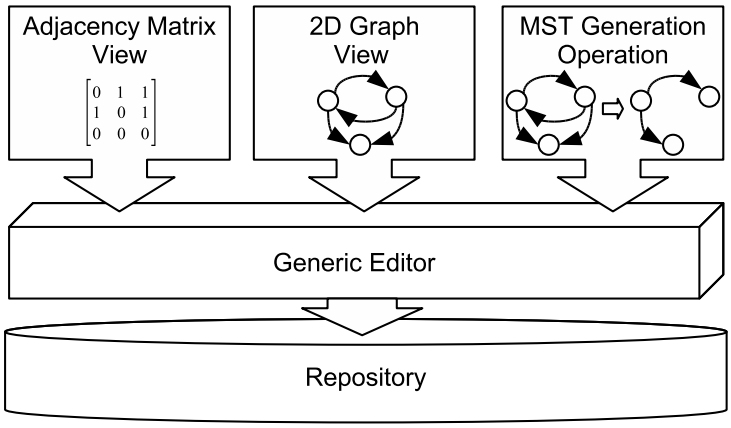

Figure 7.1: Overview of the generic editor with example plug-ins for graphs.

such as workspace management (see Sect. 7.3) and features for robustness (see Sect. 7.4.5). A plug-in's functionality is accessible in a standard way through operations, as explained in Sect. 7.4.3, which makes plug-ins potentially easier to use than stand-alone tools with their own separate user interfaces. Furthermore, all plug-ins use the repository, so they can share data about graphs efficiently. Because the plug-ins reuse the functionality of the repository and the generic editor, they can be developed and maintained more efficiently compared to stand-alone tools that had to implement this functionality all by themselves.

Most common IDEs use an accumulative approach in which extensions bring their own user interface for input and output, creating their own sometimes very unique style of integration. With the addition of many extensions, in particular extensions that were developed by different organizations, this may lead to heterogeneity in the UI. In the extreme case, it may even lead to a cluttered UI with decreased usability. In contrast to this, AP1 aims at a *conceptual integration* of different functionalities by moving input, output and even parts of the UI itself into the repository. In other words, the model-based data management capabilities of the repository are reused for as many data as possible. This will be further explained in Sect. 7.4.

The architecture of the AP1 system plays a significant role for the usability properties of the generic editor. This significance of software architecture for interactive systems has also been described by Nigay and Coutaz [164, 51]. Their model of system organization, which is derived from the Seeheim user interface model [180], distinguishes a functional core, an interface with the functional core (IFC), a dialogue controller and a presentation techniques component (PTC). These components form a layered architecture, with the layers being arranged in the aforementioned order. This bears a certain resemblance to the layered architecture of the AP1 system, as it is illustrated in Fig. 7.5.

In the AP1 system, the repository represents the functional core, and the common IFC is the repository client library, which was described in Sect. 3.9. Two functionalities of the IFC are semantic enhancement and semantic delegation, and both of them are present

in the client library: access to the relational repository database is enhanced by providing a PDM interface, and read access to the repository is delegated to a local cache.

The dialogue controller in the model of Nigay and Coutaz is a multi-agent system of significant complexity, which can be modeled formally with process algebra. It bridges the gap between IFC and PTC. In the AP1 system it is easier to bridge this gap because the generic editor, which forms the PTC, uses the direct data access approach described in Sect. 6.2.4. As a result, no complex dialogue modeling is necessary.

The generic editor has the same motivation as the user interface services of the AD/Cycle system [8]. Also some of the concepts used in AD/Cycle are similar to those of the generic editor. For example, AD/Cycle offers a notion of configurable workspaces, and a focus on direct manipulation of data objects. AD/Cycle uses an object-action paradigm, which allows users to chose actions in the context of a selected data object from a context menu. Similarly, the generic editor also offers a notion of workspaces (see Sect. 7.3), and a mechanism for operation invocation through direct manipulation (see Sect. 7.4.3).

However, in AD/Cycle system these concepts remain on the same level as is commonly offered by modern desktop environments. Configurable workspaces mean that data is presented in windows that can be freely arranged on the screen. Object handling is essentially done in the same way as in a file system browser: data objects are represented as graphical icons, and users can organize them hierarchically. Applications can be associated with data object types and invoked through mouse interaction. In the generic editor, the concepts are more general and applied on a more fine-grained level, resulting in a user interface that is different from those commonly present in desktop environments.

7.3 The Workbench

IDEs are often called workbenches. Similar to real-world workbenches, different CASE tools can be integrated and arranged in an IDE's user interface. A user can adjust a workbench by changing the locations and sizes of the different user interface parts, which are commonly referred to as *panels*. Consequently, the term workbench refers in particular to the user interface of an IDE.

In the generic editor, the panels of the workbench are views, which present data of the repository to the user and possibly allow the user to interact with them. Views are described in Sect. 7.4. The workbench itself is the component that manages all available screen space, and allows a user to distribute it among different views. The workbench can be regarded as a configurable frame for views on the repository.

In the following, I will discuss concepts that are commonly found in IDEs and other applications, such as tiling, stacking and workspaces. These concepts are important for allowing a user to configure the user interface by allocating screen space appropriately. I

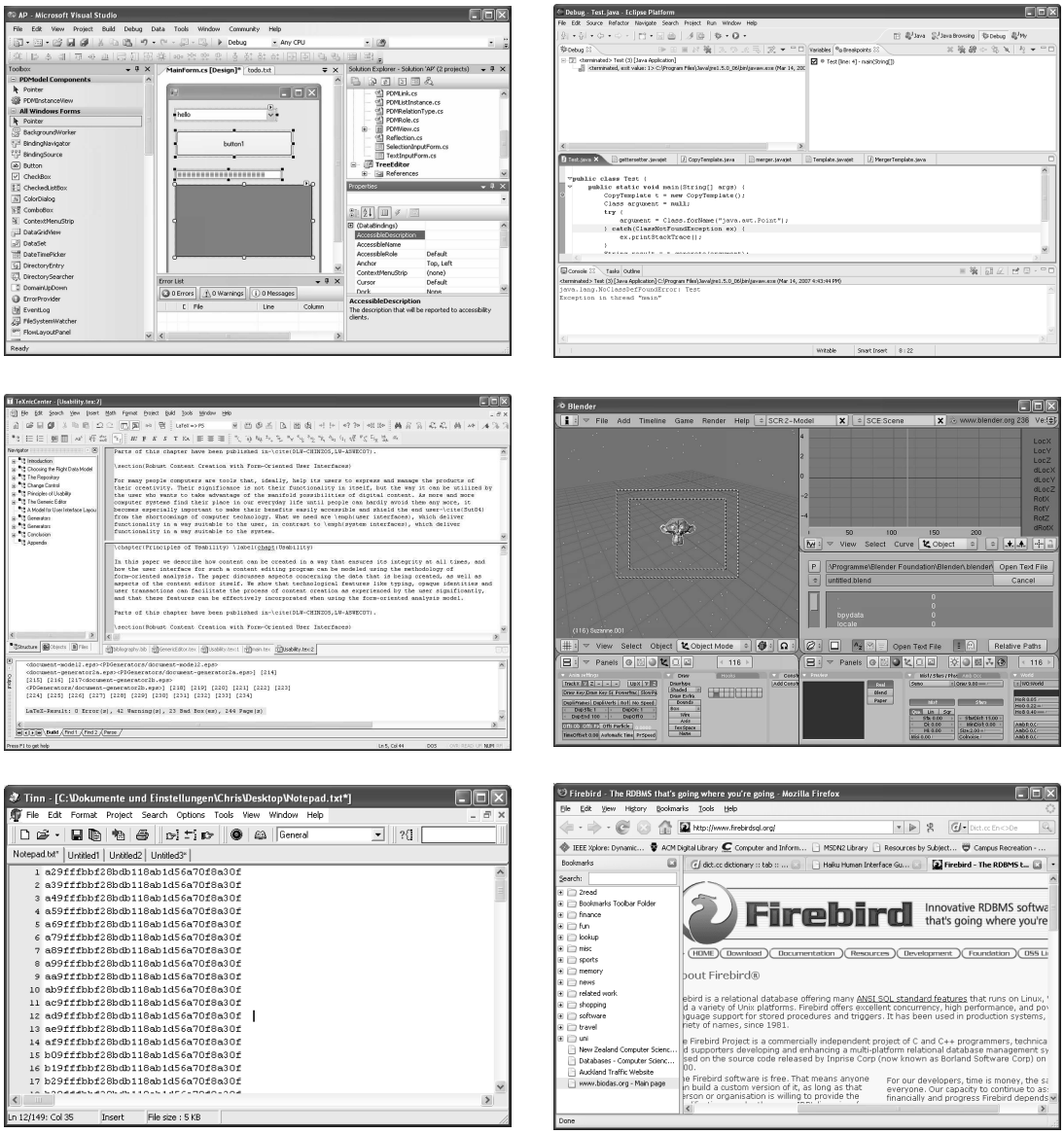

Figure 7.2: Screenshots of tiling and stacking user interfaces: Visual Studio (top left), Eclipse (top right), TeXnicCenter (middle left), Blender (middle right), Tinn (bottom left), Firefox (bottom right).

will describe how some of these concepts are used in the workbench of the generic editor.

7.3.1 Tiling and Stacking

In popular IDEs and some other multi-document applications, a certain pattern for arranging panels has emerged. This pattern, which is called *tiling*, splits a workbench up into non-overlapping areas called tiles. Tiling is often combined with another popular pattern for arranging panels: *stacking*. Stacking means that a tile can contain multiple panels, which are usually accessible through small tabs sticking out from them.

Figure 7.2 shows screenshots of several different applications that use tiling and/or

stacking. The two screenshots at the top are taken from two very popular IDEs: MS Visual Studio and Eclipse. As one can see, both systems use the tiling approach for partitioning the workbench into rectangular areas, and the stacking approach for putting several panels into the same area.

In the middle of the figure we see screenshots of applications that are not directly related to software development. The left one shows TeXnicCenter, which is a graphical environment for the LaTeX typesetting system. TeXnicCenter offers a limited capability for tiling by aligning several of its panels in a non-overlapping fashion. Furthermore, it uses stacking to group related panels, such as the panels showing the source code for the text documents being typeset. The right side shows Blender, which is a 3D modeling application. Blender strongly supports tiling; the entire user interface can be subdivided arbitrarily into rectangular areas. For each area, the type of panel shown in it can be chosen by the user.

Stacking has also become popular in other multi-document applications, such as text editors or web browsers. The screenshot in the bottom left of Fig. 7.2 shows Tinn, a "tabbed text editor", which stacks text documents. In the bottom right a screenshot of the Mozilla web browser is shown. One of Mozilla's popular features is "tabbed browsing", which means that panels showing web pages can be stacked in a browser window.

Tiling can be used in order to create a non-overlapping user interface layout without waste of screen real estate. The available screen space is partitioned, i.e. distributed completely. Stacking can be used to overlap groups of panels in a controlled, clean manner, thus using screen space more efficiently. Tiling and stacking can be regarded as best practices of GUI design.

However, there are also many applications that use overlapping user interfaces. Their user interfaces typically consist of several windows, which can overlap arbitrarily. A rationale for tiling is that overlapping user interfaces tend to hinder user interaction that relates to more than one of those windows. If different windows contain information that is important for a single task, then those windows are likely to be used in close succession or even simultaneously. However, the user has to explicitly switch between them in order to resolve occlusion and trigger input focus change. This costs time and thus slows down the task. If this slowdown is alleviated by arranging these windows in a non-overlapping manner, then the necessity for them being separate windows in the first place may be questioned.

The aforementioned slowdown of multi-window user interfaces is one of the reasons I chose a tiling-based user interface for the workbench of the generic editor. Another reason is that tiling is a well-known, proven concept, which will be familiar to most users. The generic editor allows a user to decompose a window into rectangular areas, which are called *workspaces*. This is illustrated in Figure 7.3.

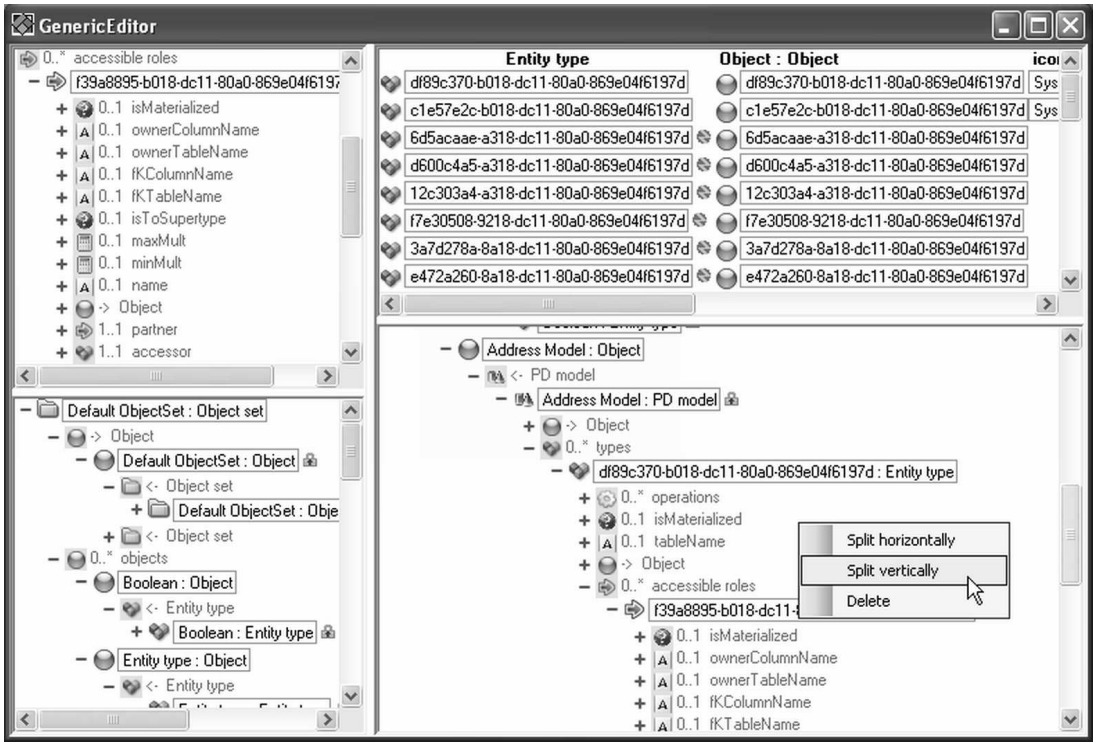

Figure 7.3: Tiling in the generic editor.

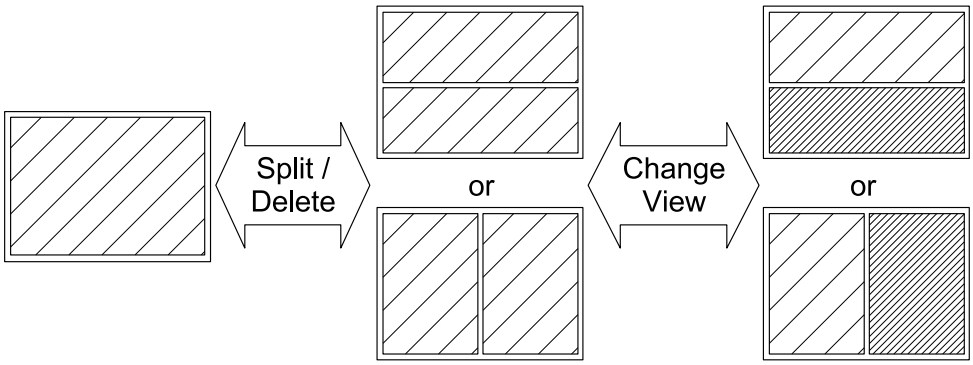

Figure 7.4: Operations on workspaces for workbench configuration.

The novelty of my workspace concept is its recursive use: workspaces can contain other workspaces. Internally, a workspace is represented as an abstract type with three subtypes: each workspace is either a single panel containing a view, or a horizontal juxtaposition of two workspaces, or a vertical juxtaposition of two workspaces. Consequently, all the workspaces in a window form a containment hierarchy that is a binary tree.

Configuration of the workbench is done using four simple operations, which are illustrated in Fig. 7.4. All of these operations can be invoked by right-clicking on one of the working areas and choosing the respective operation from the context menu. Logically, the operations have one of the visible workspaces as parameter. The four operations are the following ones:

Split Horizontally Invoking this operation on a workspace splits it in half horizontally, resulting in two workspaces with the same view in it. These two workspaces are called *siblings* because they were created from and are *children* of a common *parent* workspace. Horizontal splitting makes it possible to arrange workspaces on top of one another.

Split Vertically Invoking this operation on a workspace splits it in half vertically, resulting in two sibling children with the same view in it. This way, workspaces areas can be arranged side by side.

Delete Invoking the delete operation on a workspace deletes it and causes its sibling to take up its space. If the sibling has already been split up further, then the space is taken up by the sibling's children (or grandchildren etc.). Of course, deletion is only possible if there is more than one workspace in the workbench window.

Change View Children inherit the view displayed in them from their parents. In order to change the view type used in a workspace, a user has to invoke an operation with a corresponding side-effect on the user interface. That is, views are changed by invoking "apply view" operations on instances of suitable entity types. For each view type and each entity type that it can represent, there is such an "apply view" operation. This will be explained in more detail in Sect. 7.4.

Every split operation creates two children with a movable separator between them. The separator can either be dragged up and down or left and right, depending if it is a horizontal or a vertical split. This makes it possible to adjust the size of workspaces.

The data that describe a workspace configuration are stored in the repository, and can be edited like all other data. As a consequence, the generic editor is an example for the document-oriented user interface paradigm [74]. If the data describing a workspace currently displayed in a window is modified, the modifications are immediately reflected on the screen. For example, if a leaf workspace is replaced with a vertical juxtaposition of two new workspaces, then these two new workspaces will immediately show up. This is an example of how the generic editor implements reflection in the user interface, as described in Chapt. 6. Workspaces are updated after changes using the event notification mechanism of the repository, which was described in Sect. 3.9.4. How this mechanism is used for view synchronization will be described in more detail in Sect. 7.4.

The generic editor offers an operation for applying a workspace configuration to the current window. This operation is invoked through the context menu on an instance of entity type Workspace, as further explained in Sect. 7.4.3. That way, users can define their own workspace configurations and switch between them. They can choose an optimal user interface configuration according to the task they want to perform.

Also other IDEs provide functionality for managing user interface configurations. However, they do not let users access configuration data in the same way as other data, and they do not offer such a flexible configuration model. The concept of workspaces in the generic editor was specifically designed to satisfy the customizability requirement given in Sect. 7.1. The generic editor uses metadata integration (see Sect. 6.2.2) to make workspaces first-class data instances. The recursive organization of workspaces makes it possible to form groups of related workspaces – subtrees in the containment hierarchy – that can be reused in other workspaces. For example, a workspace for data model editing and a workspace for source code manipulation might both include general views such as an overview of all project artifacts and a list of modifications for undo/redo.

7.3.2 Universal Usage of Workspaces and Views

When comparing the user interfaces in Fig. 7.2 with that of the generic editor, another difference catches the eye. Most applications have very specialized parts in their user interface, e.g. menu bars, tool bars and status bars. These parts are often displayed all the time and serve very specific purposes of central importance. In contrast to that, the generic editor does not have any such specialized parts, but consists entirely of workspaces with panels, which are also called views. This is why the user interface in Fig. 7.3 looks so homogeneous.

Workspaces with views are used for all the constituents of the user interface, i.e. their usage is universal in this regard. Views can contain passive as well as active and interactive user interface parts, so that it is possible to recreate any non-overlapping user interface by configuring a workspace with suitable views. There can be views that act as tool bars, menu bars, status bars, etc. Views are essentially just GUI panel controls, and their content is completely up to the programmer. As we will see in the following, the term “view” can be explained by their strong ties to the data in the repository.

By contrast, in all the applications presented in Fig. 7.2 the usage of panels is constrained, i.e. non-universal, which makes the user interface more heterogeneous and less flexible. All of the applications have user interface elements that cannot be changed, e.g. menu bars. Most of them distinguish different types of panels, such as primary panels for editing or viewing of documents and secondary panels for auxiliary functions. Panels of a type can only be handled in a certain way, e.g. put into a certain region of the user interface. The presence of different panel types does not only limit the customizability of the user interface, it also complicates its implementation.

7.4 Views

Views are the parts which make up the user interface of the generic editor. As described in Sect. 7.3, they can contain any GUI controls and can offer any kind of interaction. They may also be purely passive views that just present data and do not allow any interaction at all. It was also pointed out that views are implemented with the help of operations: each view can represent instances of particular entity types, and the view defines an "apply view" operation for each of these types.

Operations have access to the workspace GUI control they were invoked from, so they can modify it arbitrarily. As we have seen in Sect. 3.6, operation instances refer to the .net assembly they are implemented in, and this assembly is also stored in the repository. Besides the method that implements an operation, the assembly can contain an arbitrary amount of other code. As a consequence, all the code necessary for a view can be stored in the same assembly as the methods of its "apply view" operations. A method for applying a view clears the current workspace and adds to it a new instance of the view's GUI control.

There are certain characteristics that most views should have in common, so that a coherent design and minimum level of quality is achieved. First of all, views generally present the data in the repository, and nothing else. In the AP1 system, the repository is the source of all data, and external data usually has to be imported into the repository before it is used. The generic editor is, in a sense, stateless: anything that is seen in its views is stored somewhere in the repository. All the components outside the repository do is to cache the repository's state. The main strategy that I use to address the integration requirement given in Sect. 7.1 is based on the statelessness: if data is integrated in the repository, this integration will also show up in the generic editor. Consequently, the repository is used for as much UI-related data as possible, and this data can be linked to any other data.

The statelessness of the generic editor is similar to other thin and ultra-thin client technologies such as many web-based systems. Statelessness is an important property, for example, in the HTTP protocol [88], which is also used for web services [104, 46]. Other examples of stateless client technologies include the remote desktop protocol (RDP), which is used in the Windows operating system, and the X protocol, which is the standard for Unix and its derivatives. There are also dedicated hardware terminals, e.g. the Sun Ray series or old mainframe terminals, which operate as stateless clients.

The generic editor shares some of the advantages of other stateless client technologies. First of all, when the client fails, no data gets lost. The protocol between client and server is standardized, therefore different clients can be used. Clients can be hot-swapped, i.e. exchanged during a session, because they do not contain any essential data that is not

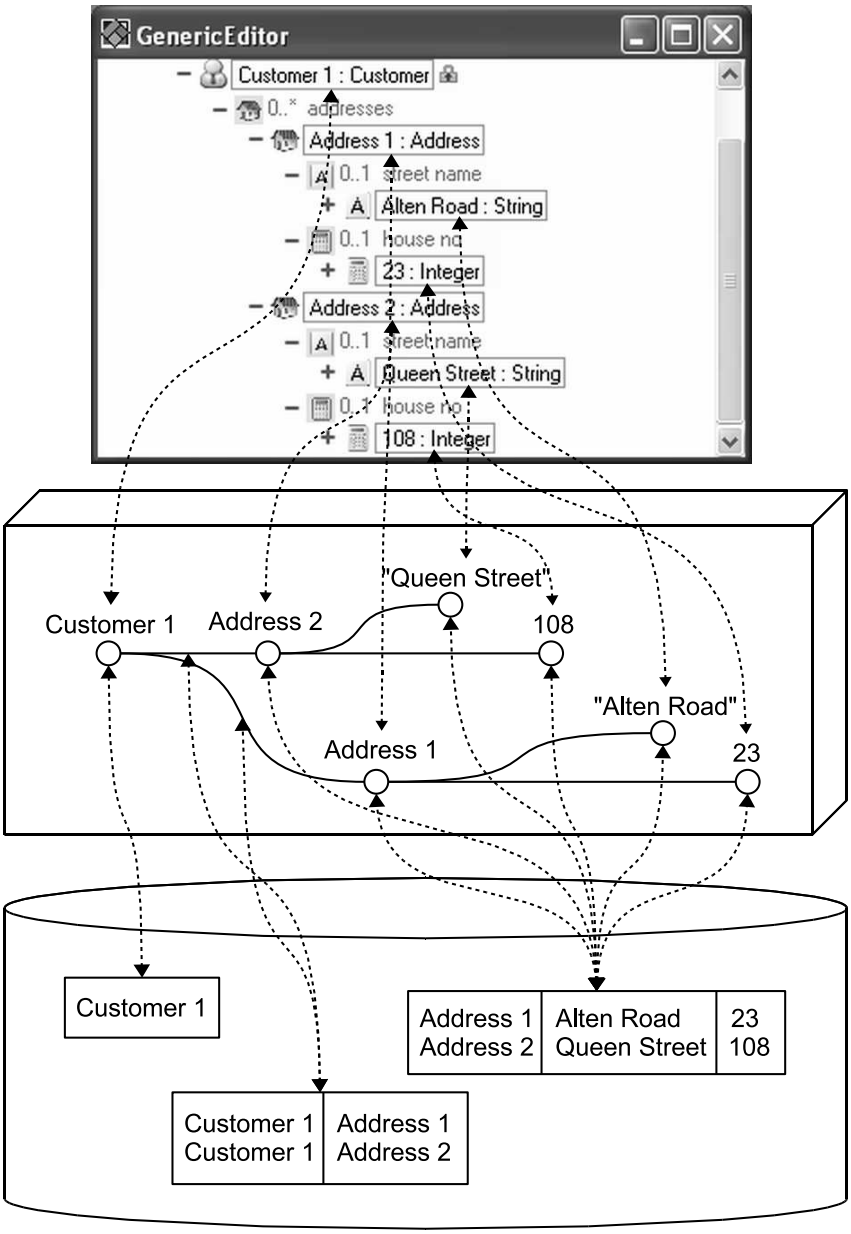

Figure 7.5: Data synchronization between the RDB (bottom), the repository client library (middle) and CASE tools such as the generic editor (top).

also available on the server. This enables session mobility, i.e. sessions can be relocated to different clients.

Figure 7.5 illustrates how the synchronization between the repository and the views built on top of the client library works, considering as example the standard tree view of the generic editor. At the bottom, the repository database is depicted, which contains tables with all the data. When a table is changed, a trigger is fired, which logs the change in the database and informs each connected client library instance. This is done using an event mechanism that sends event messages over database network connections. The change log and the triggers have been described in Chapt. 4.

In the middle part, we see one of the client library instances. As described in Sect. 3.9, the repository client library offers an object-oriented PDM interface with event notification and a local read cache. The objects in the local cache reflect the state of the data illustrated in the repository, using the perspective of the PDM. When a change event from the database is received by a client library instance, it decodes the event information and determines if the change affects the objects in the local cache. If it does, the library instance updates the cache accordingly. The objects that represent instances and links in the local cache can be used to modify the corresponding data in the database, as expressed by the PDM-relational mapping in Sect. 3.5. This interaction is indicated by the arrows between the database tables and the objects in the local cache.

The generic editor, depicted at the top of the figure, represents cached PDM instances in the views of its user interface. It offers a base class `InstanceControl`, which implements functionality that most GUI representations of instances have in common, such as operation invocation as specified in Sect. 7.4.3. In the screenshot at the top of the figure, we see how instances are represented in a tree view as objects of class `InstanceTreeNode`, which is a subclass of `InstanceControl`.

As described in Sect. 3.9.4, the client library forwards relevant change events to the corresponding objects in the local cache. These objects offer event hooks, which can be used to execute event handlers whenever a corresponding event arrives. Links define an event that is fired when the link is removed. Instances have an event that is fired when a new link is connected to the instance. Views use these events in order to synchronize their GUI controls with the objects in the local cache. They define event handlers that perform corresponding changes in the GUI whenever the PDM data in the cache change.

For example, if the link between the instances “Address 1” and “Alten Road” is removed in Fig. 7.5, then the remove event handler that is defined for all the links displayed in the tree view will remove the corresponding `InstanceTreeNode` object from the GUI. If a new link is added to instance “Address 1”, then a different event handler will be fired. If the link was connected through one of the roles displayed in the tree view, i.e. role “street name” or role “house no”, this event handler will create a corresponding `InstanceTreeNode` object and add it to the GUI.

Users can manipulate PDM data by interacting with views. In order to support data manipulation, a view has to implement event handlers for the events of its GUI controls. These events are defined by the GUI toolkit. Event handlers use the methods of the PDM data objects in the cache, which implement the PDM-relational mapping, in order to perform the desired operations on the underlying DB.

For example, clicking the “Remove link” context menu entry of the “Alten Road” `InstanceTreeNode` control invokes an event handler of the tree view. This event handler invokes a method `Remove` on the object representing the link between instance “Address

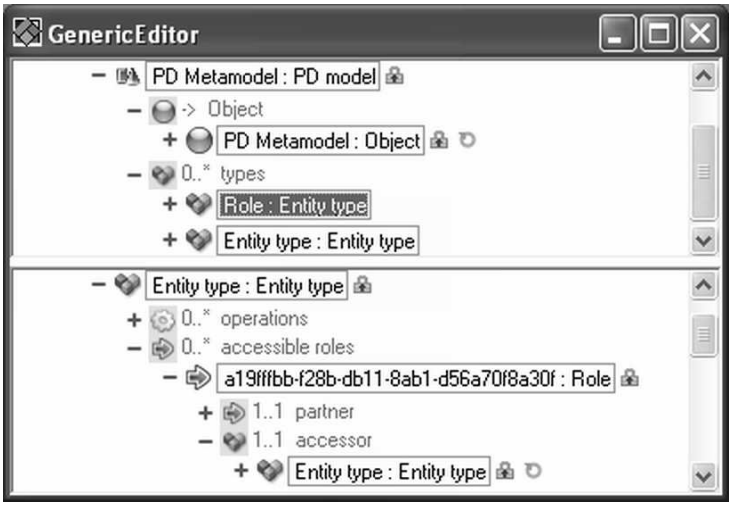

Figure 7.6: Two workspaces using the generic tree view.

1" and instance "Alten Road". Method `Remove` sets the street name column of the row for instance "Address 1" in the address table of the DB to `null`. This, in turn, causes the DB to send a change event back to the client library, which eventually leads to the removal of the "Alten Road" `InstanceTreeNode` control from the GUI. As we can see, views will only display the data that are stored in the repository, i.e. only after they are committed.

In the following sections, I will describe the generic tree view, which is the default view of the generic editor, and the generic table view. I will then proceed to discuss different features and desirable properties that these views exhibit. By following the principles of the generic tree and table views, new views with similar characteristics can be implemented.

7.4.1 The Generic Tree View

Figure 7.6 shows a screenshot of the generic editor with two workspaces. Both workspaces use the generic tree view, which can be used to view and edit any PDM data in the repository. The generic tree view can be used as a general purpose exploratory tool, with browsing capabilities similar to other schema-driven exploratory database tools, e.g. [36]. The black text elements with boxes around them represent instances, and the gray text elements represent roles, which can be used to connect instances to other instances. Since all links in the repository can be accessed with the tree view, it satisfies the universality requirement given in Sect. 7.1.

An instance is represented as an identifier, a colon, and the type of the instance. Instances below a role are connected to the parent instance above them through that role. Roles contain information about multiplicities, i.e. how many instances can be connected by the role minimally and maximally, or information about inheritance relations between

instances. Multiplicities are shown in the form $min..max$, and an inheritance relation is indicated by an arrow `->` going from subinstance to superinstance. Both instances and roles have icons that signify the type of an instance or the type that owns a role. Thus, information about the structure of data is made visible.

Instances and roles can be selected by clicking on them with the mouse. Selection is indicated by a kind of inversion of the light and dark colors in the respective label and its background. The label background is changed to blue and the text to white. In the screenshot, instance "Role" of type "Entity type" is selected. In the first line of the workspace at the top, we see an instance representing the PD metamodel: it has the name "PD Metamodel" and is of type "PD model". All types are subtypes of type Object, therefore an Object instance can be accessed from "PD Metamodel" through a role "Object". In the line below we see a role "types" with multiplicity 0..*, which can be used to access the entity type metainstances of the types contained in the PD model. If we navigate from instance "PD metamodel" via that role, we reach two entity type metainstances: "Entity type", which is the metainstance describing PDM entity types, and "Role", which is the metainstance describing PDM roles. Note that ordinary instances are represented in the same manner, e.g. an instance of a type Customer with a role "addresses" and two associated Address instances would result in a similar tree.

The tree view supports elision, i.e. by clicking on the plus sign to the left of an instance or a role the accessible roles of that instance or the instances connected through the role become visible. Conversely, if the roles of an instance or the instances of a role are visible, a minus sign is shown, and clicking on it hides the accessible roles or instances. This way, a tree view can be customized to show only those parts that are of interest for a particular task.

The workspace at the bottom of the screenshot shows metainstance "Entity type" in more detail. Symbols on the right of an instance indicate certain properties of that instance or the link it is connected with. For example, the padlock symbol indicates that a link is permanent and cannot be changed. This is the case for most of the links shown in this workspace because the entity type "Entity type" represented by the metainstance is materialized and used already, and thus cannot be subject to much change any more.

It is possible to navigate along cycles in the PDM data. For example, a cycle shown in the bottom part of the screenshot leads from instance "Entity type" via role "accessible roles" to an instance of type "Role". From there, instance "Entity type" is again reachable through role "accessor". The fact that a cycle has been reached is indicated by a symbol showing a little arrow coiled into a loop.

Figure 7.7: The generic table view.

7.4.2 The Generic Table View

Figure 7.7 shows the generic table view. This view can be used to handle PDM data in much the same way as relational data, using a table. The table lists all the instances of a particular non-primitive entity type, with each entity instance occupying a row. The instances shown here are of entity type "Address", so the table lists all the addresses that are currently present in the repository.

The columns correspond to the roles that are accessible from the instances in the table. For roles that have a maximum multiplicity of one, the cells in the respective column are either empty or contain a control representing a connected instance. For roles with a maximum multiplicity larger than one, the respective column contains cells with lists of instance controls.

7.4.3 Operation Invocation with Superparameters

In common IDEs it is often not immediately clear how to invoke a particular functionality because there are different established ways to do so. A particular functionality could be accessible from the menu bar, a tool bar, a control in one of the panels, a context menu, or a combination of them. The user interfaces of the common IDEs are usually very rich, with numerous buttons, menus and specialized panels, so finding a particular functionality may take some time. The functionality is spread out over all these different user interface elements. While this may not be a problem for experienced users, it is not a very elegant solution and leads inevitably to more complexity. The generic editor provides a proof-of-concept for a radically different approach: the parts of the user interface for invoking functionality are integrated with the ones for viewing and entering data. There are, by default, no menu or tool bars that consume valuable screen real estate. Instead, the generic views offer the ability to invoke operations directly on PDM instances. The aim of this approach is to enhance the usability of the generic editor, as given as a requirement in Sect. 7.1.

As described in Sect. 3.6, functionality is available in the form of operations, which are basically methods that can be invoked on the instances of a particular entity type.

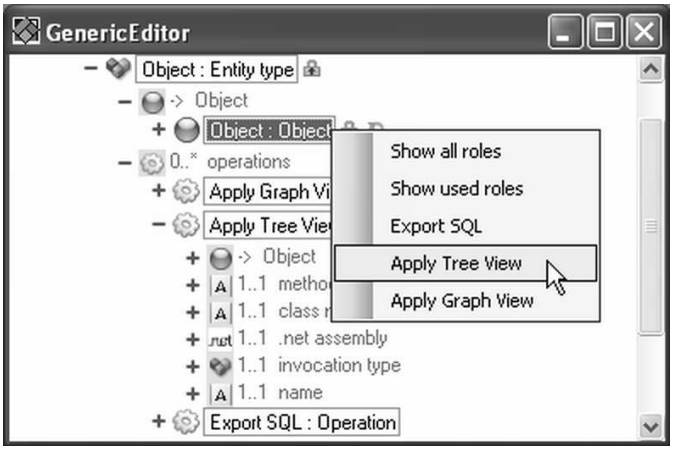

Figure 7.8: Invocation of operations in the generic editor.

Operations have a single superparameter, which comprises all information relevant for their invocation. Each operation is associated with a superparameter type and invoked on its instances. This is similar to the concept of a form, as it is present, for example, in web applications [76]. Forms describe superparameter types, but are at the same time a concept of the user interface. Users can enter values into the form, and submitting the values corresponds to an operation invocation.

In order to invoke an operation in the generic editor, one simply has to edit an instance of its superparameter type and set appropriate parameter values. Once such an instance is available, the operation can be invoked through the instance's context menu, as illustrated in the screenshot in Fig. 7.8. This provides a very structured and homogeneous approach for the invocation of functionality, which avoids the heterogeneity that can make common IDEs hard to use. It can be regarded as a special form of direct manipulation [112, 196].

The screenshot shows the context menu of instance "Object" of type "Object". It contains two operations for tree view customization, "Show all roles" and "Show used roles", an operation "Export SQL" for data export, and two operations for applying views, "Apply Tree View" and "Apply Graph View". The screenshot also shows part of the metainstance describing entity type Object: through role "operations" the Operation instances are accessible that define the operations in an Object's context menu.

The context menu contains all operations that are applicable to the type of the respective instance, but no more. Thus, no irrelevant operations are shown. If operations were represented by buttons, there would be two possibilities: either the buttons for operations that cannot be invoked in a particular situation are disabled, resulting in a waste of screen space, or the set of buttons would change dynamically, which would be a non-standard behavior that does not conform to common user expectations. However, the behavior that the entries of the context menu change depending on the object that is right-clicked does conform to common user expectations and no screen real estate is wasted.

Operation invocation as a function of the context menu is built into a generic instance control, class `InstanceControl`. All views that base their representation of instances on this control automatically inherit this feature. Thus, invocation of operations as a feature of the user interface can be efficiently reused, which encourages a coherent behavior of all views. This addresses the integration requirement given in Sect. 7.1.

Specialized ways for editing superparameter instances can be realized by different views. That is, if the default views do not represent the instances of particular superparameter types well enough, this can be remedied by a new view. For example, a view could include specialized controls such as sliders for floating point values, a calendar for the selection of dates, a file browser for the selection of files, or a palette tool for the selection of colors. Operations as abstractions of system functionality are cleanly separated from the concern of how they are accessed, so that arbitrarily many user interfaces can coexist for supporting access to an operation. By contrast, many common applications do not provide such a separation, but instead hard-wire the application logic with the GUI.

Operations usually modify the data in the repository, but they can also change the user interface of the generic editor. As indicated in Sect. 7.3.1, examples of operations with side effects on the user interface are the ones for changing the view of a workspace. But apart from changing a view entirely, operations can also be used for customizing views, e.g. for hiding unused roles in the tree view.

7.4.4 Reflection

The user interface of the generic editor uses reflection, as introduced in Chapt. 6. The screenshots in Figs. 7.6, 7.7 and 7.8 illustrate the reflective nature of the repository and the generic editor: data and metadata are integrated in a single internal data model. As mentioned in Sect. 7.1, reflection in the UI is one of my strategies to improve the customizability and usability of AP1.

All metadata is handled like data, i.e. represented and accessed with the same user interface. Most operations can be used on data as well as on metadata, e.g. editing operations or functionality for search. Both data and metadata use the same mechanisms for caching, event notification, and version control. Metadata types can be extended just like data, so it is possible to link any data with any metadata. For example, this can be used to attach documentation to data types.

Also new views inherit these reflective properties because they are rooted in the overall architecture. Integration of data and metadata happens already in the repository. The function of a view is really only that of a user interface that looks straight into the core of the system. The internals of the system are meant to be exposed to the user because it is safe to do so. A sufficient level of abstraction and safety is already achieved through the RDBMS. Section 7.5 describes more reflective properties of the generic editor.

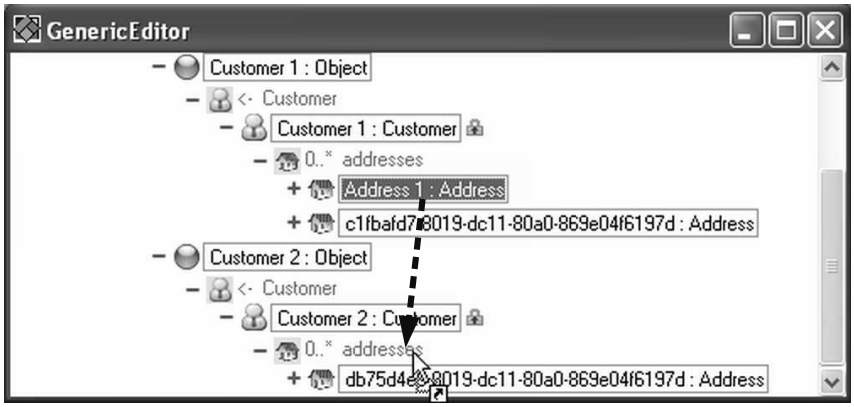

Figure 7.9: Typed drag&drop in the generic editor.

7.4.5 Robustness

As described in Chapt. 5, robustness is the property of a user interface to prevent data inconsistency in an application, despite a user not handling the application correctly. I described various ways for improving the robustness of a user interface, and the generic editor leverages most of them. The generic editor emphasizes robustness, i.e. avoiding errors, rather than just reporting them. This is another strategy aiming at satisfying the usability requirement.

All information in the user interface of the generic editor is typed and handled in a type-safe manner. For example, instances can be dragged and dropped from one place to another in order to create new links, but it is only possible to drop instances onto roles that suit their types. Figure 7.9 illustrates this for the generic tree view: an instance of type Address is dragged to a type-compatible role. Instances can have labels and icons associated with them, and can thus be handled as opaque identities, as described in Sect. 5.3.2.

The capability for typed drag & drop is built into the standard GUI controls for instances and roles. As a result, drag & drop also works between different views. This addresses the integration requirement given in Sect. 7.1. Class `InstanceControl` implements the standard control representing instances, and contains functionality that makes it possible to drag it with the mouse pointer. Class `RoleControl` implements a control representing a role that is accessible from a particular existing instance. It contains functionality that makes it possible to drop instances onto it. `RoleControl` implements a predicate method that activates the dropping capability only if a dragged instance is type-compatible with the role. A successful drop creates a link between the instance referred to by the role control and the instance that was dropped. All the views that use these controls or subclasses of them automatically support drag and drop.

An analogous approach is taken for making keyboard input type-safe. For example, an input field for an Integer instance only accepts numeric characters as input. Functionality

for typed input fields is provided as part of the generic editor, therefore keyboard input behaves consistently in all the views.

Many constraints, such as referential integrity, are enforced by the RDBMS. And since the user interface of the generic editor only reflects the state of the repository database, violations of those constraints in the user interface cannot occur. This relies on the statelessness property of the generic editor, which guarantees that there is no local state conflicting with that of the repository.

As described before, all operations invoked by the user are encapsulated in transactions, so that they cannot interfere with the actions of other users. As a consequence, their effects become only visible after they were successfully completed and a consistent new state has been reached. This implements the concept of user transactions, which was introduced in Sect. 5.4. Transactionality is also very advantageous for the development of new operations: if something goes wrong during an operation, e.g. because of a programming fault, developers can start afresh from the initial state without any data corruption. This is very convenient for the automation of software testing.

7.4.6 Non-Modality

The generic editor tries to avoid the usage of different modes of human-computer interaction. That is, the user interface ideally behaves in the same manner all the time. This is different from other typical desktop applications, which have a tradition of introducing new modes of interaction for certain tasks. Modal dialogues are the most common form of modality; they force the user to perform a particular action before continuing their work with the rest of the user interface. Modal dialogues usually interrupt the normal workflow of the user. When such a dialogue appears, much of the normal functionality of an application becomes inaccessible. Sometimes information becomes invisible because also the functions for reading data cease to work. A modal dialogue can even hinder a user to perform simple desktop operations such as focus change between windows. This is why such systems are said to have *blocking* user interfaces.

Modal dialogues should only be used when user intervention is critically important because they take away the user's attention. The user's attention is to be considered a most valuable resource and must not be wasted. However, there are many programs that use modal notifications for non-critical issues, which can be very annoying for a user. Examples are the Thunderbird email client and the Windows Explorer. The main user interface of an application is usually completely blocked until the modal dialogue is completed. Modal interaction forces the user to perform a task in a prescribed manner instead of their own. It takes away degrees of freedom from the user interface. This can be useful if users are not experienced enough to do a task by themselves – a pattern known as "wizard" – or in cases where immediate user intervention is absolutely necessary, such as

in time-critical applications. However, in many cases modal interaction causes frustrating interruptions and is likely to be detrimental to a user's productivity.

The generic editor tries to avoid modality as much as possible. There is only one repository state, and data about this state is just represented by each view. There are no "intermediate states" of the user interface that have to be controlled by modal dialogues. The state is held consistent by the RDBMS. Consequently, critical situations that might require immediate user intervention, such as corrupted data or data disparities are very unlikely to occur.

The only situations where a certain degree of modality is used is for data selection and input: operations are invoked through the context menu, which introduces a new mode. Furthermore, in the generic views instances are chosen through input controls that appear only when they are needed: for example, creating a link to a new String instance opens a panel that allows the user to type in a string. These are examples of fine-grained interaction as described in Sect. 5.4. However, what distinguishes this form of modality from the one used in modal dialogues is that it is *non-blocking*: all other controls in the user interface remain accessible. The context menu or an input control closes automatically without harm if a user starts to interact with a different control. Furthermore, this form of modality is directly induced by the user, and not forced onto the user as it is frequently done by modal dialogues.

7.5 Customizability

The generic editor can be customized in various ways, e.g. by adding workspaces to the workbench or plug-ins for views and operations. Customization is done by configuring well-specified free parameters. All configuration information is part of the repository, and customization can therefore be done by editing the corresponding model instances. Common IDEs provide complex, nested configuration dialogues, which do not scale well and may become cluttered. For example, if a new plug-in is installed in the Eclipse IDE, new panels are usually added to its global configuration dialogue. With an increasing number of plug-ins, this dialogue gets quite bulky. By integrating configuration data with all other data, the AP1 system offers a cleaner and more structured approach, which utilizes reflection in the user interface as described in Chapt. 6.

As mentioned in Sect. 3.6, all operations available to the user are instances in the repository, and we can add, delete and modify operations by changing corresponding instances and links between them. In Fig. 7.8, for example, we see how operation "Apply Tree View" is associated with type Object: instance "Apply Tree View" is connected to metainstance "Object" through role "operations". Data about the user interface are stored in the repository as well, e.g. workspace configurations as described in Sect. 7.3.

Views can store configuration information specific to them, such as data about their layout. For example, when a user hides a subtree in a tree view, this change is recorded in an instance of entity type "Tree view layout". The layout model of the tree view is out of the scope of this chapter, but I want to point out the effects of using the repository for such data. By using the repository's event notification mechanism accordingly, a change of such data can trigger an immediate corresponding effect in the respective view. By using operations for selecting the layout instance that should be used for a view, users can reuse and share view layout information.

The storage of configuration information in the repository and its accessibility over a network have another positive effect on extensibility and maintenance: extensions and updates can be deployed centrally and activated dynamically for each user. A feature for dynamic extensibility of IDEs and its benefits have been described in [225], and AP1's repository can be leveraged in a similar way. Generally, program code is part of the repository, i.e. the program code of the operations, the views, and other parts of the system. As a consequence, the repository can act as an application server. Because the code for operation and view plug-ins is bound dynamically, new plug-ins become accessible immediately for every user.

In contrast to [225], my approach is not only used for plug-ins. The program that starts the generic editor performs a bootstrapping process: it checks if there is a new version of the generic editor available in the repository, and if so, caches it locally and runs it. If a component of the generic editor is updated while the editor is running on a client machine, the update is immediately detected through AP1's notification mechanism. An update means that a new link is added to the place where the respective component is stored, which fires a "link added" event. The event handler waits until the respective client has finished its pending transactions, loads the new version of the updated component, and restarts the generic editor. Because all state information is stored in the repository, a client will restart in exactly the same state as it was terminated, with the user possibly not even noticing the update. After the restart, the only difference is that the local memory cache of the generic editor is empty, and its version is up-to-date. This feature, which is also known as hot-deployment, reduces the influence of maintenance on productivity because system downtime is minimized. Hot-deployment was convenient when testing and debugging the generic editor: changes could be done and parts could be exchanged during runtime, so that testing could proceed smoothly without much interruption.

7.6 Collaborative Work

As indicated in Chapts. 3 and 4, the AP1 system supports distributed synchronous collaboration, among other modes of collaborative work. Multiple users can access the same

data with the generic editor, and whenever a change to that data is made, it shows up in all editors immediately, as explained in Sect. 7.4. As a consequence, users can develop content of any kind together with other users, thus making distributed teamwork possible.

Cheng et al. [43] describe possibilities and benefits of the integration of features for collaborative work into an IDE. Their approach, which includes the integration of chat, IM and screen sharing, has been implemented for the Eclipse IDE [44]. Interestingly, in AP1 many of these features come naturally as a side-effect: chat and IM are nothing but a list of text messages in the repository that are edited collaboratively. All the changes made on that list can be followed live by the participating users.

While the very nature of screen sharing is that several users see exactly the same screen, users of AP1 have the possibility to choose the view that suits their purpose best. Every user can watch the data being collaboratively edited with their own settings. It is also possible for users to share GUI settings, e.g. the layout information that describes the appearance of a view. This makes it possible to create an experience similar to screen sharing, but with more flexibility and possibly better performance. Unlike many screen sharing systems, screen data is not transferred in the form of bitmaps but in the form of layout model parameters, which are much smaller.

As in [44], data can easily be associated with other data. This can be leveraged in order to improve the quality of the data and the efficiency of the development workflow. For example, chats about bugs can be linked with corresponding parts of the source code, documentation and bug tracker tickets. Related information can be clustered and thus made more readily available. These capabilities can benefit many collaborative tasks such as bug tracking and code review.

Synchronous collaboration, e.g. pair programming, can effectively improve code quality, and there is evidence that this works even in distributed environments [12]. AP1 can support this in a similar way to Sangam [110], which is an Eclipse plug-in that replicates certain input events of the IDE on several clients. But Sangam can only replicate some events, and conflicts with parts of Eclipse's functionality that have not been designed with synchronous collaboration in mind. AP1 supports distributed synchronous collaboration inherently and for all events affecting the repository.

7.7 Conclusion

I described the generic editor, which is the prototype of a model-based IDE with a high degree of customizability. The user interface of the editor is drastically different from those commonly found in IDEs, and makes heavy use of concepts for robustness and reflection. It demonstrates how a consequently model-based design with a minimalistic set of features can result in an expressive and versatile application.

The generic editor is being used and evaluated. During my tests it has shown several practical advantages. The repository approach successfully improved the stability of the system and prevented data corruption. Failures on the client-side did not affect data consistency. The generic editor could be restarted in the same state, reducing the impact of a failure on productivity significantly. Furthermore, the generic editor's capabilities for distributed collaborative work were very useful for creating content and testing the system as a team. Several people could construct models and data together while communicating via VoIP, similar to team programming. When team members found a defect in the system, they could call other team members located somewhere else, and examine the erroneous behavior together in real-time.

Naturally, this prototype is not a complete IDE yet. In addition to this, it is not clear if the unconventional user interface of the generic editor will be popular with software developers. From a theoretical standpoint, it does have advantages. However, only experience can tell how these advantages will manifest in practice. Nevertheless, the concepts explored in the generic editor are important and form a basis for a lot of future work.

Since the generic editor is based on the repository, the potential limitations of the repository as described in Sect. 3.11 also apply to the editor. For example, the centralized nature of the repository can lead to delays in the editor. All modifications in the generic editor are directly written into the repository, and the read cache of the client library does not speed up these write operations. However, as pointed out before, performance can be improved in many ways, and the immediate storage of modifications has significant advantages.

The generic editor represents a principled approach for the modification of model-based data. It is intended to be the model-based equivalent of a text editor. Its minimalistic but complete user interface makes it a tool that is more suitable for power users. For unexperienced users, user interfaces that are more specialized for particular tasks may be more appropriate. This is comparable with using a text editor for changing a textual configuration file of an application vs. using a graphical configuration dialogue.

The generic editor paves the way for a lot of empirical research work. For example, experiments could compare its user interface with those of other IDEs in terms of learnability, user satisfaction and speed. Other possible future work is concerned with distributed, synchronous collaboration. For example, does model-based collaborative editing work as well as textual pair programming? How much can different views of the same data for different collaborators benefit such distributed collaboration tasks? I will continue to work on the generic editor, and try to find answers to these and similar questions.

8

Code Generators

This chapter describes a mechanism for generative programming that generalizes the concept of generic types by combining it with a controlled form of reflection. This mechanism makes many code generation tasks possible for which generic types alone would be insufficient. The power of code generation features is carefully balanced with their safety, so that static type checks can be performed on generator code. This leads to a generalized notion of type safety for generators.

As indicated in Chapt. 1, code generation is a very important part of CASE technology. Therefore, I found it necessary to explore general concepts of code generation that could be integrated into AP1, as an example on the application layer. The result is the Genoupe language. Although implemented for and tested with a specific programming language, Genoupe is by no means restricted to any language in particular. It specifies general concepts that could be integrated into all common programming languages. Therefore, Genoupe achieves the level of abstraction appropriate for a system such as AP1.

Section 8.1 introduces the domain of generative programming. Section 8.2 describes the Genoupe language, which integrates code generators into the C# language, and provides source code examples of possible applications. Section 8.3 introduces the notion of generator type safety, which is a particularly strong notion of type safety for generators, and gives malformed examples of Genoupe code that cannot be given a correct type. Section 8.4 presents the novelties of Genoupe's type system. Section 8.5 describes how Genoupe is integrated into the AP1 system. Section 8.6 discusses related work, explaining

how Genoupe is different from similar approaches, and Sect. 8.7 concludes this chapter. The Genoupe approach for generative programming, including parts of this chapter, has been published in [73, 71, 70].

8.1 Introduction

Generators are a cornerstone of today's software engineering, especially in the area of enterprise application development [76]. There exists a large variety of tools for the generation of database interfaces, GUIs and compilers, and even many modeling tools can be subsumed under the notion of generators. Besides these very specialized examples of code generation technology, many systems have been developed that offer a more generic approach toward code generation. Some of these systems allow the user to extend a programming language with new constructs which trigger the generation of customized code.

In many cases it is not easy for a user to develop own code generators, even when using systems that support this explicitly. The user has to have knowledge about how a generator receives its parameters, how code is represented and processed, how code is emitted, and how a generator is deployed. The answers to these questions vary greatly from technology to technology.

Code generation is a sensitive area because it depends on parameters, and the usual data structure involved, a syntax tree, is not trivial. A generator may work well most of the time but can potentially fail with some rare actual parameters, and an error may not be obvious but express itself in some slightly malformed parts of generated code. Using generators always bears the risk of introducing hard to find bugs, while a good generator has the potential to provide an economic and solid solution to a common problem. Complexity in the development of code generators leads to generators that are more error-prone.

In this chapter I show how the concept of code generators can be made accessible to the user directly in object-oriented languages and how a type system can be extended to take generators into account. The aim is to make generators part of a program and not of the compiler while retaining the safety properties of a typed language. No internal knowledge of the compiler should be required, and the generation process should be transparent for the user.

Placing generators into the language itself instead of into a compiler affects the language syntax as well as its semantics and safety; the challenge lies in integrating the new constructs syntactically without interfering with existing semantics. Typed languages usually offer a high degree of safety through the use of type systems, and type checkers are able to detect many potential execution errors statically. However, with the new con-

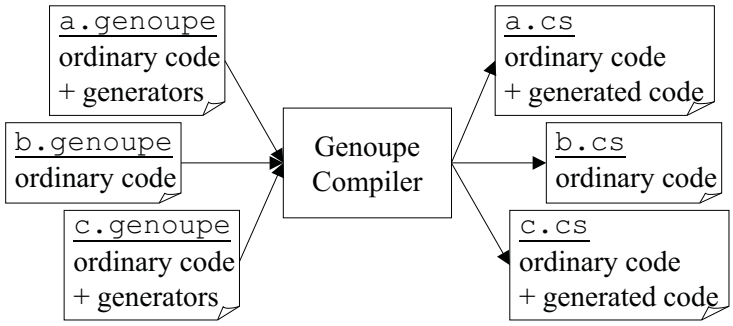

Figure 8.1: The Genoupe compilation process.

cept of generators new types of potential execution errors are introduced, namely those that happen when code generation produces ill-typed code. Consequently, code generation poses new challenges to type systems.

8.2 The Genoupe Language

My concept for the integration of generators into object-oriented programming is called Genoupe. It was developed from the language Factory [68], which integrated reflective generators into Java, and implements a similar but strongly revised concept for C#. Genoupe introduces a syntax that is reminiscent of that of generic types, although it is not limited to classes or interfaces.

Similar to generic types the template paradigm is used, but in contrast to simple genericity, templates can contain generator code written in a special compile-time level language. This sublanguage is kept in an imperative style and along the lines of the C# language itself, so that a C# programmer will intuitively understand its meaning. Also the type system is analogous to the runtime one, but simpler for ordinary types, since usually not as many features are needed here as for runtime code. With respect to generated types the type system gets somewhat more sophisticated, and I need a whole set of essentially new type rules. However, this is well worth it because, as we will see in Sects. 8.3 and 8.4, the new type system makes it possible to detect parts of a generator that can potentially generate malformed code, in contrast to just detecting code that is malformed itself.

In the Genoupe language a generator can be embedded into the source code like an ordinary type definition. Source code files written in the Genoupe language have the name suffix `.genoupe` and are compiled to ordinary C# source files with the same name but `.cs` suffix (see Fig. 8.1). Each time a generator is applied with new arguments, new types with unique names are created. If a generator is applied more than once with the same arguments in a compilation run, the corresponding code is generated only once.

In a generator actual type parameters can be accessed through so called *generator*

variables. These are variables that, in contrast to runtime variables, hold objects at generation-time and make them accessible in the generator code. Analogous to the parameters in an ordinary method, each declared generator parameter creates a generator variable, which can be used in *generator expressions*. A generator expression describes a value that is used at generation-time, just as an ordinary expression describes a value that is used at runtime. It is very similar to an ordinary C# expression in the sense that most generator expressions are valid C# expressions.

One specialty of generator expressions is that, with the same values assigned to the generator variables, two structurally equal generator expressions describe the same value. There are no non-deterministic effects such as random values, which are not needed in code generators. As we will see in Sect. 8.4, this will help to rule out some potential generation errors statically.

Usually generator expressions are used to introspect type parameters and extract or construct the information that is needed for intercession, i.e. information that represents code that should be made part of the generator output. In order to make the value of a generator expression part of the generated code, the generator expression is enclosed in `@` characters and placed into the code template at a position where the entity that is represented by the expression's value is allowed to occur. At generation time all generator expressions are evaluated and substituted by the code represented by their values.

If I want, for example, to generate a particular type in a declaration of a generated class, I would create a generator expression that evaluates to a `Type` object representing the desired type. This generator expression would be placed, enclosed in `@` characters, at the position in the source code where I would normally place a type name. During generation-time, Genoupe substitutes the generator expression by the name of the type represented by the type object in the generated code. Genoupe makes use of the standard C# metaobject protocol, so that it is obvious in most cases which type represents which language entity. For example, types are represented by `Type` objects, methods by `MethodInfo` objects and identifiers by `String` objects.

In addition to generator expressions, Genoupe provides imperative control constructs for code generation. Control constructs can be used to generate syntax elements conditionally or iteratively and create new generator variables. `@if` is used for conditional generation: it is followed by a generator expression serving as conditional, which has to evaluate to `Boolean`. If the conditional evaluates to true during generation, then the entities in its body are generated, otherwise not.

`@for` allows to iterate over the elements of any `ICollection` object: it requires the name of an iterator variable and a generator expression that evaluate to an object the class of which implements the `ICollection` interface. For each element in the collection, one iteration is performed and the entities specified in the body of the loop are generated

once. The respective collection element can be accessed from within the body through the iterator variable.

`@const` allows to create a new generator constant, with a value given by an arbitrary generator expression. This construct is merely a convenience: instead of repeating the same generator expression at different places in a generator, the value of the generator expression can be defined as a constant once and accessed by a name. Again, this is possible because with the same values assigned to the generator variables, two structurally equal generator expressions describe the same value.

In the following sections I will look at different examples of Genoupe generators, ranging from simple parametric polymorphism to more sophisticated class extensions.

8.2.1 Parametric Polymorphism

One of the simplest applications for Genoupe is parametric polymorphism, which is also known as generic types. The following generic stack generator has a single parameter `T` of type `Type` and generates a stack class for elements of type `T`:

```
1  public class Stack(Type T)
2  {
3      private Stack s = new Stack();
4
5      public void push(@T@ x) {
6          s.push(x);
7      }
8
9      public @T@ pop() {
10         return (@T@) s.pop();
11     }
12 }
```

The generator parameter declaration in line 1 looks a bit similar to a method declaration with a single method parameter. Like a method, a generator can declare an arbitrary number of parameters with arbitrary types. In lines 5, 9 and 10 I insert generator expressions containing only the generator parameter in order to generate correct type declarations and type casts.

The generator makes sure that the type safety of the stack can be checked statically: only objects of type `T` can be pushed onto the stack, and the pop method returns only objects of type `T`. If I programmed a stack without parametric polymorphism, I either had to hard-code particular types into the stack, making it less reusable, or declare a very general type such as `Object`, which would make the stack statically unsafe.

8.2.2 Generating Class Extensions

Genoupe can be used for the generation of useful extensions. In contrast to ordinary inheritance mechanisms, which also extend classes, a generator can adapt the extension it generates to the class that is extended. This makes it possible to address static cross-cutting concerns [128].

The following example is a class generator that takes a class and generates a corresponding subclass with an adapted `ToString` method. This method is used, for example, when an object is printed on the console. Such a `ToString` method, which simply returns the names of its class' fields together with string representations of their values can be very useful for debugging purposes.

```
1   class WithToString(Type T) : @T@
2   {
3       override public String ToString() {
4           String s = "";
5           @foreach(F in T.GetFields()) {
6               s += @new Literal(F.Name)@ + ":\t" + this.@F.Name@ + "\n";
7           }
8           return s;
9       }
10  }
```

In line 1 I declare a single type parameter `T` for this generator. In line 5 this parameter is introspected, and I iterate over all its fields. For each field `F`, code is generated that adds the name and the value of `F` to a string `s`. Note that the expression `@new Literal(F.Name)@` in line 6 generates a string literal for the field name, while in expression `this.@F.Name@` the identifier of the same field is generated, so that it is accessed in the generated code.

The next code snippet shows a generator that takes a class `T` and an array of field names `FNames` for that class. It generates a subclass of `T` that extends it by a new method `Randomize` that assigns random values to the fields of `T`. This can be useful, for example, for the generation of test data.

```
1   public class Randomizeable(Type T, String[] FNames) : @T@
2   {
3       public void Randomize() {
4           Random r = new Random();
5           @if(FNames!=null) {
6               @foreach(FName in FNames) {
```

```
7                   @const F = T.GetField(FName);
8                   @if(F.FieldType==Double)
9                       this.@F.Name@ = r.NextDouble();
10                  else @if(F.FieldType==Boolean)
11                      this.@F.Name@ = (r.NextDouble()>=0.5);
12                  // ...handle other data types...
13              }
14          } else {
15              @foreach(Field in T.GetFields()) {
16                  // ...generate assignments for all fields...
17              }
18          }
19      }
20  }
```

In line 5 we see the `@if` control construct of the generator language for conditional generation. It checks if an array of field names has been given at all, and only then the `FNames` array is used. In line 6 we see the `@foreach` construct, which is used for iterative generation. Its only difference to the `foreach` construct of C# is that the static type of the iterator variable needs not to be declared. In line 7 I define a new generator variable with a constant value, holding the `FieldInfo` object of a respective field. In the following lines, depending on the type of the respective field, I generate a statement that assigns to the field a compatible random value. The field's identifier is generated with a corresponding generator expression of type `String`. In the else-clause of the outermost `@if`, I can handle the case that an array of field names was not given. In this case, which works analogous to the given code, random values would be assigned to all fields of `T`.

8.2.3 Generating Proxies and Wrappers

A common pattern for modifying the behavior of existing classes or bridging incompatibility is the use of proxies [96] and wrappers. With Genoupe both of these can be generated automatically, which makes it possible to address dynamic crosscutting concerns [128].

The following class generator takes a type parameter `T` and creates a subtype of `T` that overrides and wraps `T`'s methods. A class generated by this generator behaves like `T` but logs all method calls and exits, which can be useful for debugging purposes.

```
1   public class Logger(Type T) : @T@
2   {
3       public String Log = new String();
```

```
4
5       @foreach(M in T.GetMethods()) {
6           @const Pars = M.GetParameters();
7
8           public override @M.ReturnType@ @M.Name@
9           (@foreach(P in Pars) { @P.ParameterType@ @P.Name@ })
10          {
11              Log += @new Literal(M.Name)@ + " called.\n";
12              base.@M.Name@(@foreach(P in Pars) { @P.Name@ });
13              Log += @new Literal(M.Name)@ + " exiting.\n";
14          }
15      }
16  }
```

In lines 8 and 9 I use generator expressions to generate the signature of each of `T`'s public methods. A list of method parameter declarations is generated by iterating over all the parameters and generating each parameter declaration individually. The same approach is used in line 12 in order to generate the list of arguments for a method call. The `Literal` objects constructed in lines 11 and 13 represent generated string literals, opposed to generated identifiers.

The next example shows a class generator that generates a security proxy for a given class. Like in the previous example, the generated class is a subclass of the given one, so that it can be used in its place. Genoupe can generally be useful for supporting the implementation of design patterns [96], such as the proxy, the observer or the template method patterns.

```
1   class SecurityProxy(Type T) : @T@
2   {
3       @foreach(M in T.GetMethods()) {
4           @if(M.IsPublic) {
5               override public
6               @M.ReturnType@ @M.Name@(@M.GetParameters()@) {
7                   if(/* access permitted */))
8                       base.@M.Name@(@M.GetParameters()@);
9                   else
10                      throw new SecurityException();
11              }
12          }
13      }
```

```
14  }
```

In line 1 I declare the type parameter `T` and specify that the generated class is a subclass of `T`. In line 3 I iterate over all methods `M` of `T`. I want to protect only those methods that can actually be accessed from outside, therefore the following generation of a wrapper method is conditional and only performed for public methods. The following part generates a method definition that overrides the respective public method represented by `M`. In line 6, a return type, an identifier and parameters are generated with appropriate generator expressions. While a type can be generated with a `Type` object and an identifier with a `String` object, a list of parameter declarations can be generated with a collection of `ParameterInfo` objects, as it is provided by method `GetParameters`. As we can see in line 8 where I invoke the original method, the same collection can also be used to generate a corresponding list of actual parameters. In line 7 I generate code that checks if the method access is permitted. This part highly depends on the actual application and is therefore left out in this example. If permission is given, the requested method is called, otherwise an exception is thrown.

8.2.4 Generating Interfaces

Another application for Genoupe and generative programming in general is the generation of system interfaces, be it a GUI, a database interface, a web interface or an API. The following Genoupe code sketches out a class generator that generates a GUI form for editing a selection of fields of a given object. Many tasks of interface generation can be done in a similar way. This example can be seen as a merely programming-language-based instance of model-based user interface development [55]. The model is described in terms of the programming language's type system.

```
1   class EditForm(Type T, FieldInfo[] V) : Form
2   {
3       @foreach(F in V)
4           // declare GUI controls needed to edit field F
5
6       @T@ X;
7
8       @constructor@(@T@ x) {
9           this.X = x;
10          @foreach(F in V) {
11              // initialize GUI controls
12              // set event handlers
```

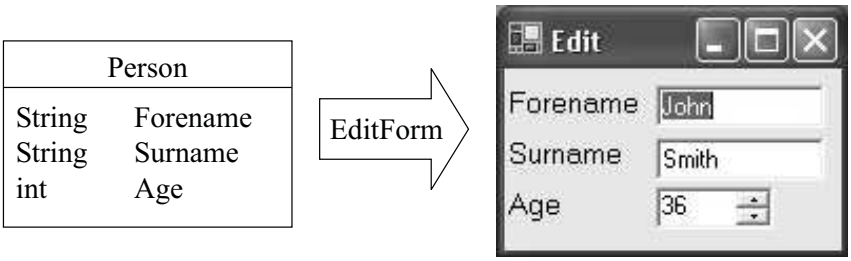

Figure 8.2: The EditForm generator.

```
13          }
14      }
15
16      @foreach(F in V)
17          // define event handlers
18  }
```

I define a class generator that declares a parameter `T` for the type of the objects that should be edited. Parameter `V` holds an array of `FieldInfo` objects representing the subset of the fields of `T` that should be editable in the generated GUI. First, in lines 3-4, the controls that are needed for the construction of a suitable GUI are iteratively generated. For most fields this could be a `Label` and a `TextBox`. In line 6, I declare a member variable `X` that holds the object to edit. In lines 8-14 a constructor is generated. Its identifier is generated with the keyword `constructor`. The constructor stores the object to edit and sets up the GUI controls and their event handlers appropriately. In line 16-17 event handlers are generated that propagate changes in the GUI to corresponding changes in the fields of `X`. In Fig. 8.2 we see how the input and output of a generation could look like.

8.3 Generator Type Safety

When dealing with metaprograms, i.e. programs that process other programs or themselves in some suitable representation, a whole set of new sources of execution errors comes into play. *Generation errors* in generators are those parts of the generator program that can potentially generate malformed code, which in turn may cause execution errors when executed. Of course, I also want the generators to be free of execution errors themselves. In addition to normal type systems, which can only detect potential forbidden errors in the code that is type checked, a new kind of type system is needed that can also detect parts in generators that can potentially generate ill-typed code. This requirement leads to a new notion of type safety, which I want to call *generator type safety*. It is the property of a generator not to be able to generate ill-typed code, i.e. code that may cause a

forbidden execution error. If a generator is not generator type safe, it contains one or more *generator type errors*, i.e. parts in the generator code that are responsible for the generation of ill-typed code. A type system that can detect generator type errors is called a *generator type system*.

Before I describe the generator type system of Genoupe in the next section, let us look at examples of malformed generators that can potentially generate ill-typed code. The following generator generates a class with a single field:

```
1   class C(Type T)
2   {
3       @T@ x = 1;
4   }
```

The fact that `x` is assigned a numerical value restricts its possible type. The type parameter `T` however is not subject to any such restriction. This is clearly a generator type error that leads to some arguments producing type-correct code and others not.

The next example demonstrates another issue of type compatibility.

```
1   class C(T istype Component)
2   {
3       @T@ x = new Button();
4   }
```

The Genoupe keyword `istype` makes it possible to set a bound for type parameters, i.e. parameters of type `Type`. Line 1 signifies that parameter `T` is a type parameter and that all possible arguments represent types that are either class `Component` itself or one of its subclasses. In the generator body, a member variable `x` with type `T` is defined, and a `Button` object is assigned to it. `Button` is a subclass of `Component`, but what if `T` is a subclass of `Component` but not compatible to `Button`, i.e. not either `Button` itself or one of its superclasses? The generated code is type correct iff `T` is `Button` or one of its superclasses.

The following example is a class generator that has a string parameter `ID`. As the name suggests, the string is used to generate the identifier of a local variable in a method.

```
1   class C(String ID)
2   {
3       void m() {
4           int @ID@ = 1;
5           x++;
6       }
7   }
```

In line 5 a variable `x` is incremented. Since there are no other variable definitions in the generator, `x` must be defined in the preceding line where the identifier of a variable is generated by a generator expression. If the generator is given the argument `"x"`, the generated code works just fine, otherwise it is ill-typed. This is also known as the problem of *inadvertent capture* [130].

The next generator contains a conditional generation.

```
1   class C(String X)
2   {
3       @if(X.Equals("hello")) {
4           String y = "world";
5       }
6
7       void m() {
8           Console.WriteLine(y);
9       }
10  }
```

The definition of the member variable `y` is only generated when `"hello"` is the string argument in `X`. Again, there are cases where this generates an error and others where it does not.

My last example illustrates a generator type error that can occur in iterative generation.

```
1   class C(Type S, Type T)
2   {
3       @foreach(F in S.GetFields()) {
4           @F.FieldType@ @F.FieldName@;
5       }
6
7       void m() {
8           @foreach(F in T.GetFields()) {
9               Console.WriteLine(this.@F.FieldName@);
10          }
11      }
12  }
```

The first generative iteration replicates the field definitions of type parameter `S`. The second one in method `m` generates statements that access and print the values of fields as defined in type parameter `T`. Clearly this can only work if `S` contains fields with identical

name for all the field definitions in `T`, which is of course the case when `S` and `T` are bound to the same type.

All these generator type errors also occur in real generators. For example, such errors are introduced when applying inconsistent changes: one part of a generator is changed without adjusting other parts accordingly that are affected by that change. The errors in real generators are usually harder to find than in our small examples: real generators are bigger, and the involved parts may be distributed over the code. And as mentioned before, a generator type error may not always cause a problem, and hence remain undetected.

Note that the Genoupe language has another property which makes its generators safer than those in many other languages: if all the methods used in generator code terminate and generators are not used recursively, which is usually unnecessary, a generator is guaranteed to terminate. This is because my looping construct, the @foreach, iterates over collections without modifying them, and the collections contain of course only a finite number of elements. In C++ templates, for example, we must use recursion when we want to repeat something arbitrarily often. C++ templates can potentially recurse endlessly, and only a limited recursion-depth prevents this [54]. In other technologies that use a Turing-complete language for metaobject manipulation, such as CLOS [95], OpenC++ [45] or Jasper [165, 166], generators potentially do not terminate as well.

8.4 The Genoupe Type System

In order to detect generator type errors, I developed a generator type system that is compatible with and extends the type system of the host language C#. Its notation is similar to the one used in [35]. It consists of rules with judgments about the correctness of certain program parts in their pre- and postconditions, and only the programs that can be derived by those rules are considered type correct. However, in some respects my type system deviates from the way in which type systems of object-oriented languages usually work. I use an environment Γ, which keeps track not only of the signatures of declared runtime variables but also of the signatures of generator variables. The signature of a runtime variable can contain generator expressions because its identifier and type may be generated by them. For handling conditional and iterative generation of declarations correctly, definitions that are generated conditionally or iteratively have special signatures, and Γ is also used to store additional facts about the code portion that is being type-checked.

Rather than delivering a complete description of the type system, this chapter focuses on explaining the main concepts by looking at some exemplary type rules. These rules can be found in Table 8.1, and I will go through them one after another. Rule [*Env Var*] describes how the signature of a generated variable can be included into Γ. The two

judgments in the precondition state that a correct generator expression of type `String` is needed for the variable's identifier, and a correct generator expression of type `Type` for the variable's type. The :: symbol associates a generator expression with its type. In the postcondition the new environment is a conjunction of the old Γ and the new signature. The : symbol associates the identifier of a variable with its type. Rule [*Env then*] makes it possible to register in Γ that a generator expression *Gexpr* evaluates to true. The generator expression must be of type `Boolean` and the opposite, i.e. that *Gexpr* evaluates to false, must not be registered in Γ already. As the name of the rule suggests, this rule is used for type-checking in the then-clause of an `@if` construct, where the generator expression describing the condition of the `@if` is known to be true. Analogous to this, rule [*Env loop*] makes it possible to register in Γ that an iterator variable of a `@foreach` contains an element of a particular collection, which is the collection over which is iterated.

Rule $[Def\ Var]$ describes how a variable definition can be generated with suitable generator expressions and what its signature looks like. The $\therefore$ symbol associates a signature to a definition. A signature is a set of facts that describe a definition. Rule $[Def\ @if]$ describes the conditional generation of definitions. In the second and third line of the precondition, the facts *Gexpr* and $\neg$*Gexpr* are included in the environment when it is demanded that the declarations D_1 and D_2 have the signatures Sig_1 and Sig_2, respectively. Consequently, the judgment in the postcondition states the correctness of an `@if` with D_1 in the then- and D_2 in the else-clause. The signature of the $@if$, which becomes part of the environment during type-checking, has two parts: one describing the signature of the generated definition in the case that the condition is true and one describing the signature of the generated definition when it is not. Rule $[Def\ @foreach]$ describes the iterative generation of definitions. In the second judgment of the precondition I demand that D is a correct definition with signature Sig. The environment states that ID is an iterator variable that contains an element of the collection described by *Gexpr*. The signature of the resulting `@foreach` is again a special one: it signifies that for any generator variable X, with X being an element of some collection described by *Gexpr*, there is a signature that looks like the signature of D, only that each occurrence of ID in that signature is replaced by X.

The rules $[Expr\ Var\ 1]$, $[Expr\ Var\ 2]$ and $[Expr\ Var\ 3]$ will hopefully clarify why these unusual elements are needed in Γ. They all specify how a generated variable can be used in a generated expression. Rule $[Expr\ Var\ 1]$ states that if there is a generated variable declared in Γ, an expression can be generated that uses it by generating its identifier with a corresponding generator expression. Rule $[Expr\ Var\ 2]$ describes under which circumstances a variable can be used that has been generated in the then-clause of a conditional generation: it can be used if Γ states that *Gexpr*, the condition under which the variable was generated, is true. Analogous to this rule, there is also one for using a

$$[Env\ Var] \quad \frac{\Gamma \vdash Gexpr_1 :: \texttt{String} \quad \Gamma \vdash Gexpr_2 :: \texttt{Type} \quad Gexpr_1 \notin \mathrm{Dom}(\Gamma)}{\Gamma \cup \{Gexpr_1 : Gexpr_2\} \vdash \diamond}$$

$$[Env\ then] \quad \frac{\Gamma \vdash Gexpr :: \texttt{Boolean} \quad (\neg Gexpr) \notin \Gamma}{\Gamma \cup \{Gexpr\} \vdash \diamond}$$

$$[Env\ loop] \quad \frac{\Gamma \vdash Gexpr :: \texttt{ICollection}}{\Gamma \cup \{ID \in Gexpr\} \vdash \diamond}$$

$$[Def\ Var] \quad \frac{\Gamma \vdash Gexpr_1 :: \texttt{Type} \quad \Gamma \vdash Gexpr_2 :: \texttt{String}}{\Gamma \vdash @Gexpr_1@\ @Gexpr_2@; \therefore \{Gexpr_2 : Gexpr_1\}}$$

$$[Def\ @if] \quad \frac{\begin{array}{c}\Gamma \vdash Gexpr :: \texttt{Boolean} \\ \Gamma \cup Sig_1 \cup \{Gexpr\} \vdash D_1 \therefore Sig_1 \\ \Gamma \cup Sig_2 \cup \{\neg Gexpr\} \vdash D_2 \therefore Sig_2\end{array}}{\begin{array}{c}\Gamma \vdash @\mathrm{if}(Gexpr)\ \{\ D_1\ \}\ \mathrm{else}\ \{\ D_2\ \} \\ \therefore \{Gexpr \rightarrow Sig_1,\ \neg Gexpr \rightarrow Sig_2\}\end{array}}$$

$$[Def\ @foreach] \quad \frac{\begin{array}{c}\Gamma \vdash Gexpr :: \texttt{ICollection} \\ \Gamma \cup Sig \cup \{ID \in Gexpr\} \vdash D \therefore Sig\end{array}}{\begin{array}{c}\Gamma \vdash @\mathrm{foreach}(ID\ in\ Gexpr)\ \{\ D\ \} \\ \therefore \{\forall X \in Gexpr.Sig[X/ID]\}\end{array}}$$

$$[Expr\ Var\ 1] \quad \frac{(Gexpr_1 : Gexpr_2) \in \Gamma}{\Gamma \vdash @Gexpr_1@ : Gexpr_2}$$

$$[Expr\ Var\ 2] \quad \frac{\{Gexpr,\ Gexpr \rightarrow Gexpr_1 : Gexpr_2\} \subseteq \Gamma}{\Gamma \vdash @Gexpr_1@ : Gexpr_2}$$

$$[Expr\ Var\ 3] \quad \frac{\{ID \in Gexpr,\ \forall X \in Gexpr.(Gexpr'_1 : Gexpr'_2)\} \subseteq \Gamma \quad (Gexpr'_1 : Gexpr'_2)[ID/X] = (Gexpr_1 : Gexpr_2)}{\Gamma \vdash @Gexpr_1@ : Gexpr_2}$$

Table 8.1: Exemplary type rules of the Genoupe generator type system.

variable that has been generated in the else-clause of an `@if`. Finally, rule [$Expr\ Var\ 3$] handles the usage of variables that have been generated in a `@foreach`. Such a variable can be used if Γ states that the usage of the variable is in the body of a `@foreach` loop that loops over a collection described by the same generator expression as the collection of the loop in which the variable was defined. This means that the collections of the two loops are equivalent. In the loop in which code is generated that uses the variable, the iterator variable may have a different identifier. Therefore I replace the X in the variable's signature by the ID of this loop's iterator variable.

8.4.1 Limitations

Like most type systems, the Genoupe type system is restrictive: it forbids not only programs that are obviously incorrect but also many others which do not contain generator type errors. In the rules for the `@if`, for example, it is required that a conditionally generated variable must be used in the body of a conditional with equivalent condition. Logically it would be enough, though, to require that the condition of the defining conditional *implies* the condition of the conditional in which the variable is used. Analogously, if variables are generated in a `@foreach`, it would be sufficient to demand that they are used in a loop that iterates over a *subset* of the collection in the defining iteration. Because the underlying problems are undecidable, I did not try to solve them, although it would be possible to address these issues using approaches from logical programming such as constraint solving and model checking. Note that this is a popular way for type systems to deal with issues that restrict the way a language is used but do not really limit its applicability: C# and Java, for example, do not really check whether a method with a non-void return type returns a value; they merely check if a superset of possible execution paths returns a value.

The possibility to generate arbitrary identifiers with generator expressions brings about lexical problems: a generated identifier might be malformed, e.g. it might clash with a keyword, or might not be unique. Both these problems could only be solved if I restricted the way identifiers can be generated. But if I did that, Genoupe would lose flexibility and potentially the ability to produce clear human-readable names, and the language would become more complicated. The more freedom I allow for the generation of identifiers, the more complex a collision detection scheme would have to be in order to avoid this problem. I decided not to implement any such restriction or detection scheme and take the risk of lexical collisions, which is inherent when working with a textual source code representation. The responsibility for handling the generation of identifiers carefully lies with the programmer of a generator.

8.5 Integrating Genoupe into the AP1 System

Integrating the Genoupe concepts into the AP1 system is not difficult. In fact, it simplifies the implementation of Genoupe due to AP1's structured repository and its notion of operations. The implementation of Genoupe as a textual stand-alone precompiler and its integration into the AP1 system are illustrated in Fig. 8.3. In this figure, data artifacts are represented as document shapes, with a folded bottom right corner, and processing components as boxes.

In the precompiler implementation, the initial artifact is textual Genoupe source code, as described in the previous sections. Before processing the generator code, the source

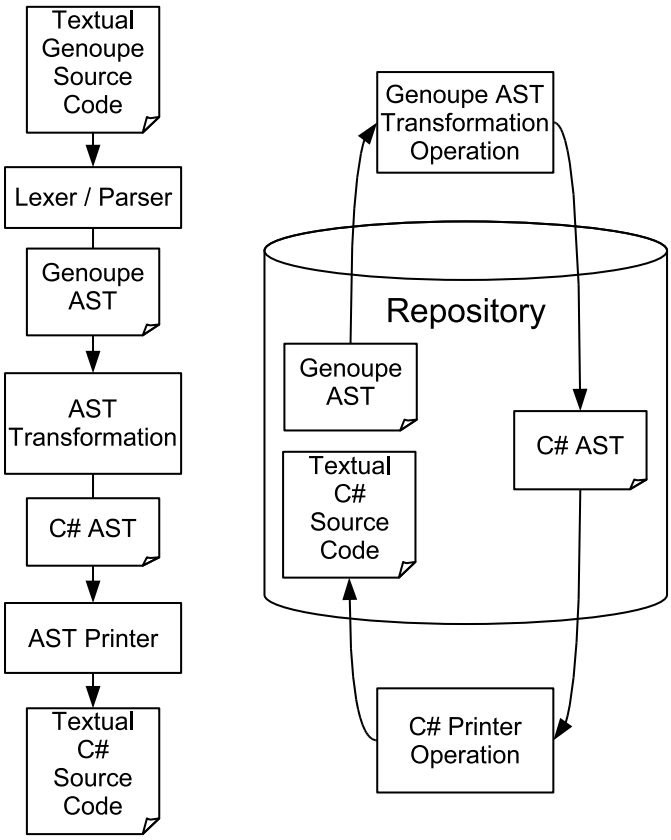

Figure 8.3: Implementation of Genoupe as a textual stand-alone tool (left) and integration with the AP1 system (right).

code has to be scanned by a lexer and parsed into a Genoupe abstract syntax tree (AST). This is a routine compiler construction task [3], but there are pitfalls such as potential syntactical ambiguities that have to be handled. The grammars of modern languages such as C# can be quite voluminous. Consequently, the construction of a good lexer and parser can consume a significant amount of time.

The actual generation work is done by the AST transformation component. This is essentially a tree parser which takes the Genoupe AST as input, eliminates the Genoupe-specific tree nodes, and adds appropriate C# tree nodes to the AST instead. It is the heart of Genoupe, and specified on the relatively high level of typed, abstract syntax. Consequently, the transformation steps can be formulated quite concisely. Finally, the C# AST has to be serialized by another tree parser, i.e. printed back to a textual C# source code representation.

The integration of Genoupe into the AP1 system, as shown on the right side of Fig. 8.3, eliminates the need for a Genoupe lexer and parser. As a platform for model-based software development it is designed to deal with structured data such as ASTs directly. The structure of a Genoupe AST can be reformulated directly as a PD model, which can be managed in the repository. Programming with Genoupe is done in a structured way,

by editing instances of that PD model, with tools such as the generic editor.

Generation is implemented with an operation that performs the transformation between a Genoupe and a C# AST. This operation is essentially the same as the transformation component of the stand-alone tool, with the difference that it is based on a PDM representation of the involved data. Just like Genoupe ASTs, C# ASTs can be specified as model instances of an appropriate PD model as well.

Another operation, which is essentially the C# AST printer of the stand-alone tool, can be used in order to transform the C# AST into a string containing corresponding textual C# source code. As a result, also C# code can be managed using the repository, developed with the help of structured tools such as the generic editor, and exported for usage with textual tools such as compilers. In the same manner, support for other programming languages can be added.

8.6 Related Work

Genoupe is an extension of *genericity* or parametric polymorphism as found, for example, in ADA or Java [28, 29]. With parametric polymorphism it is possible to program components that are uniformly reusable for many types. However, these generic type parameterization mechanisms are at the same time type abstraction mechanisms: the construction of the type cannot be exploited in the parameterized software component – at most it can be exploited up to a bound, known as bounded parametric polymorphism. Therefore it is useful for container libraries, e.g. C++ Standard Template Libraries, but it is not as powerful as Genoupe.

The original *C++ template mechanism* does not allow for the enforcement of properties for actual type parameters as, for example, supported by the notion of bounded parametric polymorphism [35, 181]. Ad-hoc solutions to provide some level of concept checking for C++ templates, such as specialized macros [197] and static interfaces [153], have been generalized by the introspection library approach in [229]. This approach targets user-customized checks for both compile-time adaptation and diagnostics.

The *new C++ templates* standard allows in principle Turing-complete meta-programming with static reflection in C++ [9], sufficient, e.g., for an interface generator for a relational database [10]. However, it is still less powerful than Genoupe. For example, it is not possible to generate function names dependent on a parameter. It does not support any static notion of generator type safety; type-checks are done with the ordinary C++ type system. Furthermore, a template metaprogram may not terminate. The Turing-completeness makes it impossible to analyze the generating templates exhaustively.

Aspect-oriented programming aims at the handling of crosscutting concerns in programs. AspectJ [128] is a Java extension for aspect oriented programming, which offers

two approaches: dynamic and static crosscutting. Crosscutting does not help with type-dependent generative problems, e.g. the implementation of a transparent data-access layer. Static crosscutting allows to extend the signature of classes and interfaces, but not in an adaptive manner: we can add a new method to a class from within an aspect – so-called member introduction – but still the method has to be specified literally and cannot be made dependent on some parameter.

The concept of *runtime reflection* dates back to Lisp [199] and has been subject of major interest in the functional programming community. The combination of parametric polymorphism with reflective features in Generic Haskell [109, 108] benefits from the theoretical well-understood type system of the host language. In the context of the object-oriented functional programming language CLOS [129, 95], a mature metaobject protocol has been elaborated. In [218] CLOS is used to prove the value of metaprogramming by embedding representations of common object-oriented design patterns [96] into programs.

Multistage programming [207, 208] is an approach that focuses on runtime program generation and execution. The programmer is supported by constructs for partial evaluation and program specialization, and several properties of runtime generation can already be ensured statically. An implementation of the multistage programming approach is provided on top of the object-based functional programming language O'Caml [139]. The language Metaphor [161] results from extending a subset of an object-oriented language such as C# or Java with the multistage constructs of the functional programming language MetaML [207, 208], i.e. a construct for building representations of expressions, a construct for splicing code and a construct for running staged evaluated code. With its multi-staged language design Metaphor achieves type-safe generation of code that makes use of the reflection system of the base language.

Jasper [165, 166] is a reflective syntax processor for Java. It provides mechanisms for *static reflection*. It does not follow the template approach; instead it allows for metaprogramming through the extension/modification of the syntax processor itself [69] – an architecture that is known as *open compiler*. It supports universal metaprogramming and is as such more powerful, but less understood.

8.7 Conclusion

Genoupe implements a concept for generative programming that integrates reflection by means of a metalanguage into a template mechanism reminiscent of genericity. It can be used to solve common problems of generative programming and offers advantages compared to other languages with respect to the degree of integration of the runtime and the metalanguage and safety:

- Genoupe places the concept of generators into the language instead of relying on

an external tool driven approach, thus minimizing the interface to the user and avoiding potential errors.

- It integrates well with an object-oriented host language and can be seen as a generalization of genericity. It uses similar syntax for runtime and generator code, which makes it easy to use and understand.
- A wide range of common applications of generative programming can be addressed.
- Genoupe offers an particular high degree of static safety for reflection by means of a type system that is able to detect generator type errors.

More information about Genoupe and implementations of the Genoupe system can be found on the project web site, `http://www.genoupe.formcharts.org/`.

9 Conclusion

AP1 provides powerful features and novel concepts for the implementation of model-based CASE tools, emphasizing ease of integration and customizability. Its architecture is, to the best of my knowledge, unique. In Sect. 9.1 I list the major achievements that were presented. Section 9.2 points out future directions of the AP1 project. Finally, Sect. 9.3 adds some concluding reflective remarks.

9.1 Achievements

The AP1 project was, and still is, the source of several publications. It has dealt with and found solutions for problems of various domains. In order to start with the right assumptions, I had to ask questions about software engineering and CASE tools in general. To find the right design for the repository, I explored various methods of data modeling and data management. Furthermore, I considered approaches for data mapping, event notification, caching, change control and software configuration management.

Usability is a significant factor for CASE, and a cause of many difficulties. To this end I came up with new concepts and ideas for making user interfaces more consistent, expressive and intuitive. The generic editor implements my ideas, and offers advanced functionality such as different methods for collaborative work. Finally, I studied the field of generative programming, which is of major importance for CASE technology, and developed a better notion of code generators.

The following list highlights the major achievements of this project:

- **A comparison of different data models that takes into account their various strengths and weaknesses.** I studied some of the most popular data models, and evaluated them with regard to the requirements of a model-based repository. I found that information about the disadvantages of popular data models is often imprecise and not discussed systematically. My analysis of XML identifies potential shortcomings that have not been described before. Such work is important in order to retain a scientific view on commercially popular technologies, which tend to become blurred due to heavy marketing.

- **A relational implementation of the PDM with support for introspection and intercession and an object-oriented interface.** I compared different mappings between the PDM and the RDM. This led to a new, efficient implementation strategy that uses GUIDs and metadata in order to keep track of PDM entity types, roles and instances. Models are formally described in terms of a PD metamodel, which is much simpler than other common metamodels. This makes it possible to abstract from some of the technical details of the RDM and make data modeling easier. I developed an object-oriented interface that addresses the object-relational impedance mismatch and paves the way for a deeper integration of the OODM, the PDM, and the RDM.

- **An architecture for a repository that is based on a relational database, with a mechanism for synchronous event notification.** I presented a novel architecture for a model-based repository that leverages the capabilities of a standard RDBMS. In contrast to other systems which implement their own data store, reusing a RDBMS results in a very lightweight and standards-conformant solution. Furthermore, I developed a novel, PDM-based technique for caching and synchronous event notification that allows tools to communicate through the repository, using a standard object-oriented event mechanism.

- **A flexible approach for fine-grained change control in a relational database.** I developed a novel technique for logging and managing all the changes in a relational database. This technique makes it possible to control changes on a much finer level than the systems that are currently used for version control. Unlike other systems, my approach allows a very high degree of customization, such as the choice between centralized and decentralized, synchronous and asynchronous distributed collaboration.

- **Several approaches for making user interfaces for the creation of content more robust.** I identified and defined robustness as an important property of a

user interface, and discussed how user interfaces of content creation applications can be made more robust. Robustness in a user interface has never been systematically discussed before, although it is a significant issue for many types of applications. Therefore this work is a contribution towards making not only software development tools but also other systems more user-friendly.

- **A new interpretation of the reflection principle for the domain of user interfaces, with significant implications for usability.** I showed that the reflection principle known from the domain of programming languages can be meaningfully applied to user interfaces. This is novel work that identifies and discusses several approaches for improving a user interface's customizability, transparency and consistency. I demonstrated that many existing user interface concepts can be classified using the reflection taxonomy, and that reflection represents a suitable lens for developing new approaches for usability.

- **A highly customizable architecture for a generic, structured, multiple-view editing tool with a clean, robust and reflective user interface.** I successfully implemented approaches for robustness and reflection in a generic model-based editing application, thus proving their feasibility. My editor is based on the repository, and demonstrates how a software development application can benefit from such a structured, transactional, active data store. The editor provides multiple, synchronized views on the data of the repository and supports new ways of distributed collaboration. Collaborators can use different views while synchronously editing the same data. The high degree of customizability is achieved by storing all configuration data in the repository and reacting to changes immediately. This novel architecture makes it possible to synchronously update and share views, operations, user interface layout data and even the program code of the editor itself.

- **A new concept for code generators with a type system for generator type safety, which is a particularly strong notion tailored to the needs of generative programming.** I developed Genoupe, an extension of the C# programming language, that offers a novel code generation concept. Genoupe combines generic types with a safely restricted form of structural reflection, using the standard C# metaobject protocol. It is powerful enough to perform common code generation tasks, but offers a higher degree of safety than similar generator technologies. I introduced the new notion of generator type safety, which accounts for the fact that code generators are metaprograms, and as such may not only contain type errors themselves but may create ill-typed code. I identified and discussed several examples of generator type errors, and formulated a generator type system that is capable of detecting generator type errors statically.

9.2 Future Directions

There are several ideas that have not made their way into this thesis. This is not surprising considering the huge scope of CASE technology. I intend to continue working on the AP1 system, using it as a research vehicle as well as a tool that can benefit professional software development. The following sections point out some possible future work.

9.2.1 A Configurable Meta-CASE View

Many common diagram types essentially rely on a structure similar to ordinary 2D graphs. The shape of the nodes and vertices varies, e.g. nodes can be boxes or circles with labels and also vertices can have labels and different arrowheads. But such diagrams, in particular data models, are usually networks of connected shapes. This can be generalized, in the sense that many such models can be described as typed graphs plus type-specific information about shapes.

Meta-CASE tools such as [228, 100] exploit this by letting users define their own CASE tools. The term CASE tools is used here in a very restricted sense, meaning visual model editing tools. Instead of supporting only a single model type and hard-coding the look of model parts into such a visual editor, meta-CASE tools let the user customize these things. A user can define a data model and associate information about the visual representation with each individual data type. General visual concepts such as connection points and connectors are mapped to general data modeling concepts such as roles and associations. As a result, a good meta-CASE tool can be used to specify several different models and edit their model instances.

I have already developed a general model for 2D graph-like visual structures, and have also explored some algorithms for automated 2D graph layout. My intention is to add meta-CASE functionality into the AP1 system in the form of a customizable and general-purpose 2D graph view. This would make it possible for users to create new views for different types of diagrams directly from within the generic editor, as configurations of this general-purpose view.

9.2.2 An Abstract Model for Generators

In this thesis I have presented the Genoupe model for code generators. While this model can be applied as a powerful tool for generative programming, it is not general enough to support the generation of arbitrary artifacts. Furthermore, there is no straightforward way to visualize and edit Genoupe generators in a graphical manner. The paradigm for Genoupe is that of programming rather than visual construction.

I created a model for generators that is intrinsically based on the PDM. As a result,

data for arbitrary PD models can be generated, not just source code. It is designed to retain the expressiveness of the Genoupe model, so that problems from the domain of generative programming can be addressed in a similar way. I implemented this generalized notion of generators as operation and tested it with various PD models. Further research on this approach will be done in the future.

The generalized generator model has a visual representation that allows it to illustrate generators in a PDM-like fashion. As a result, generators can be drawn and edited graphically. With regard to the fact that many of the common generated data types are non-linear in nature, it seems plausible that such an inherently non-linear graphical representation may foster a higher degree of usability and ease of understanding.

The model uses techniques that are similar to those of graph-grammars [83]. Graph grammars have already been applied to the domain of model-driven software engineering, e.g. see [59, 103, 60], and have been found to be a valid approach.

9.2.3 An Abstract Model for Source Code

I am considering a model for source code that is abstract in the sense that it can be used to specify programs on a high, relatively technology-independent level. It should enable developers to specify system logic without having to consider technological details. Generators should be used in order to produce an implementation for a particular target language automatically. This overall notion would be similar to that of model-driven architecture (MDA) [156].

Besides raising the level of abstraction, the advantage of such a language would be the prevention of technological lock-in. For example, if a company decided to build a valuable system based on Java technology, the future of that system would be tied to the future of the Java platform. If, for some reason, that platform became obsolete at some point in the future, the system would become a legacy, causing expensive maintenance problems. This is what happened with the Cobol language several decades ago, and there is no conclusive reason that it could not happen again. However, if the system was programmed in a language that was tailored for generation into different target languages, the value of such a system could potentially be preserved, even if one target language became obsolete.

9.2.4 An Abstract Model for Graphical User Interfaces

I developed and implemented a model for the description of graphical user interface layout [144]. This model utilizes simple mathematical concepts such as ordinal and linear constraints in order to specify the way GUI controls are arranged. The model enables dynamic adaption of the layout, e.g. when the size of a window is changed.

Later on, the model was further developed to allow more flexibility for GUI developers, and formalized as a problem of linear programming. The new version is called the Auckland Layout Model (ALM), and was implemented and tested as a layout engine for the .net platform. The ALM is already used in the user interface of the generic editor. However, due to space limitations it was not possible to include it in this thesis.

I plan to develop a view for the generic editor that allows users to create graphical users interfaces in a WYSIWIG fashion using the ALM. This is similar to existing GUI editors, which form a common part of all the major IDEs. However, like the abstract source code model the ALM aims at technology independence: it should be possible to generate GUI implementations for different target technologies from a single ALM specification.

9.2.5 Models for Text Documents and Graphics

At a relatively early stage I started to think about the value of structured data manipulation – such as offered by the generic editor – for areas only marginally related to CASE. Many typesetting systems such as the LaTeX system structure their data, e.g. in the form of sections, figures and bibliographic citations. Furthermore, the entities of such systems are essentially treated like data objects, with the possibility to reference, reuse and annotate them. However, systems like LaTeX are based on linear textual input, thus creating an impedance mismatch to the inherently non-linear data structures. By using the AP1 system to manage and edit such data, a whole range of lexical, syntactical and semantic errors could be ruled out.

I believe that tools for the creation of graphics, especially technical drawings, could equally benefit from the features of AP1. Instead of using an ad-hoc WYSIWIG approach, as most popular desktop drawing applications such as OpenOffice Draw or MS Visio do, also graphical data could be created in a model-based fashion. A view could still offer interactive controls for WYSIWIG representation and direct manipulation of such data, but a user could also deal with that data on a more structured, conceptual level. Although all graphics applications need to have an internal structured data representation of some sort, this representation is rarely accessible for the end-user.

9.3 Reflections

Someone once told me about a similarity between a PhD thesis and age: “When you are very young, you try to look older than you are. Later on, its the other way around. When you have just begun writing your thesis, you are very eager to make it look as much as possible, whereas later on, you try to keep it short in order to respect the page limit.” This is certainly also true for my thesis.

It is said that there exist stereotypical highs and lows in the life of a young scientist, such as the submission of the first paper, or its rejection. One of them is certainly the shocking discovery that someone else has already published a paper about exactly the same thing. With growing maturity we learn that related work is never about exactly the same thing, and is in fact a good thing. Dirk and Gerald helped me to understand such principles of scientific practice, and provided me with the necessary inspiration and encouragement.

All in all, the time in Berlin, working with my colleague Dirk ("Brennt noch Licht im Clubhaus?"), and the Auckland years working with Gerald ("TIA!") have been a great time! The PhD project has been an adventurous academic journey through many different areas of research, which was not only interesting but also fostered an all-round understanding of the discipline. Now that it is drawing to a close, it feels, in fact, more like a beginning than an end.

Bibliography

[1] S. Abramsky and A. Jung. Domain Theory. In *Handbook of Logic in Computer Science*. Oxford University Press, 1994.

[2] A. Abran, P. Bourque, R. Dupuis, and J.W. Moore. *Guide to the Software Engineering Body of Knowledge-SWEBOK*. IEEE Press, 2004.

[3] A.V. Aho, R. Sethi, and J.D. Ullman. *Compiler: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[4] M. B. Albizuri-Romero. A retrospective view of CASE tools adoption. *SIGSOFT Softw. Eng. Notes*, 25(2):46–50, 2000.

[5] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[6] Sihem Amer-Yahia, Fang Du, and Juliana Freire. A comprehensive solution to the XML-to-relational mapping problem. In *WIDM '04: Proceedings of the 6th Annual ACM International Workshop on Web Information and Data Management*, pages 31–38. ACM Press, 2004.

[7] M.J. Anderson and B.D. Bird. An evaluation of PCTE as a portable tool platform. In *Proceedings of the Software Engineering Environments Conference*, pages 96–100, 1993.

[8] JM Artim, JM Hary, and FJ Spickhoff. User interface services in AD/Cycle. *IBM Systems Journal*, 29(2):236–249, 1990.

[9] Giuseppe Attardi and Antonio Cisternino. Reflection Support by Means of Template Metaprogramming. In *GCSE '01: Proceedings of the 3rd International Conference on Generative and Component-Based Software Engineering*, LNCS 2186. Springer, 2001.

[10] Giuseppe Attardi and Antonio Cisternino. Template Metaprogramming an Object Interface to Relational Tables. In *REFLECTION '01: Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, LNCS 2192. Springer, 2001.

[11] D.J. Bagert, H. Saiedian, R. Dupuis, M. Shaw, P.A. Freeman, and J.B. Thompson. Software engineering body of knowledge (SWEBOK)(panel session). *Proceedings of the 23rd International Conference on Software Engineering*, pages 693–696, 2001.

[12] Prashant Baheti, Edward Gehringer, and David Stotts. Exploring the efficacy of distributed pair programming. In *XP/Agile Universe 2002: Proceedings of the Second XP Universe and First Agile Universe Conference*, pages 208–220. Springer, 2002.

[13] Rajiv D. Banker and Robert J. Kauffman. Reuse and productivity in integrated computer-aided software engineering: an empirical study. *MIS Quarterly*, 15(3):375–401, 1991.

[14] M. Beaudouin-Lafon and W.E. Mackay. Reification, polymorphism and reuse: three principles for designing visual interfaces. *Proceedings of the Working Conference on Advanced Visual Interfaces*, pages 102–109, 2000.

[15] Michel Beaudouin-Lafon. Interactions as First-Class Objects. In *Proceedings of the ACM CHI 2005 Workshop on the Future of User Interface Design Tools*. ACM Press, 2005.

[16] Benjamin B. Bederson, Jon Meyer, and Lance Good. Jazz: an extensible zoomable user interface graphics toolkit in Java. In *UIST '00: Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology*, pages 171–180. ACM Press, 2000.

[17] Catriel Beeri, Philip A. Bernstein, and Nathan Goodman. A sophisticate's introduction to database normalization theory. In S. Bing Yao, editor, *VLDB '78: 4th International Conference on Very Large Data Bases*, pages 113–124. IEEE Press, 1978.

[18] Alex E. Bell. Death by UML fever. *Queue*, 2(1):72–80, 2004.

[19] Alex E. Bell. UML fever: diagnosis and recovery. *Queue*, 3(2):48–56, 2005.

[20] David E. Bellagio and Tom J. Milligan. *Software Configuration Management Strategies and IBM Rational ClearCase: A Practical Introduction*. IBM Press, May 2005.

[21] P.A. Bernstein, T. Bergstraesser, J. Carlson, S. Pal, P. Sanders, and D. Shutt. Microsoft Repository Version 2 and the Open Information Model. *Information Systems*, 24(2):71–98, 1999.

[22] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control - theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, 1983.

[23] B. Boehm. Project termination doesn't equal project failure. *Computer*, 33(9):94–96, 2000.

[24] G. Booch, J. Rumbaugh, and I. Jacobson. *The unified modeling language user guide*. Addison-Wesley Professional, 2005.

[25] C. Bornhövd, M. Altinel, C. Mohan, H. Pirahesh, and B. Reinwald. Adaptive database caching with DBCache. *Data Engineering Bulletin*, 27(2):11–18, 2004.

[26] R. Bosua and S. Brinkkemper. Realisation of an integrated software engineering environment through heterogeneous CASE-tool integration. In *Proceedings of the Conference on Software Engineering Environments*, pages 152–159. IEEE Press, 1995.

[27] P. Bourque, R. Dupuis, A. Abran, J.W. Moore, L. Tripp, and S. Wolff. Fundamental principles of software engineering–a journey. *The Journal of Systems & Software*, 62(1):59–70, 2002.

[28] Gilad Bracha, Norman Cohen, Christian Kemper, Steve Marx, Martin Odersky, Sven-Eric Panitz, David Stoutamire, Kristen Thorup, and Philip Wadler. Adding Generics to the Java Programming Language: Participant Draft Specification. Technical report, SUN Microsystems, April 2001.

[29] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: adding genericity to the Java programming language. In *OOPSLA '98: Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, SIGPLAN Notices, pages 183–200. ACM Press, October 1998.

[30] F.P. Brooks. No Silver Bullet Essence and Accidents of Software Engineering. *Computer*, 20(4):10–19, 1987.

[31] Antony Bryant. It's engineering Jim . . . but not as we know it: software engineering – solution to the software crisis, or part of the problem? In *ICSE '00: Proceedings of the 22nd International Conference on Software Engineering*, pages 78–87. ACM Press, 2000.

[32] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework*. Addison-Wesley Professional, 2003.

[33] Peter Buneman. Semistructured data. In *PODS '97: Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 117–121. ACM Press, 1997.

[34] David E. Caldwell and Michael White. CogentHelp: a tool for authoring dynamically generated help for Java GUIs. In *SIGDOC '97: Proceedings of the 15th Annual International Conference on Computer Documentation*, pages 17–22. ACM Press, 1997.

[35] Luca Cardelli. Type Systems. In *Handbook of Computer Science and Engineering*, chapter 103. CRC Press, 1997.

[36] M. Carey, L. Haas, V. Maganty, and J. Williams. PESTO: An Integrated Query/Browser for Object Databases. 1996.

[37] M. Cartwright and M. Shepperd. An empirical investigation of an object-oriented software system. *IEEE Transactions on Software Engineering*, 26(8):786–796, 2000.

[38] R.G.G. Cattell, Douglas K. Barry, Mark Berler, Jeff Eastman, David Jordan, Craig Russell, Olaf Schadow, Torsten Stanienda, and Fernando Velez, editors. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann Publishers, January 2001.

[39] Per Cederqvist. Version Management with CVS, 2002.

[40] R.N. Charette. Why Software Fails. *Spectrum*, 42(9):42–49, 2005.

[41] Peter Pin-Shan Chen. The entity-relationship model – toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[42] J. Cheng and J. Xu. XML and DB2. In *ICDE'00: Proceedings of the 16th International Conference on Data Engineering*. IEEE Press, 2000.

[43] Li-Te Cheng, Cleidson R.B. de Souza, Susanne Hupfer, John Patterson, and Steven Ross. Building collaboration into IDEs. *Queue*, 1(9):40–50, 2004.

[44] Li-Te Cheng, Susanne Hupfer, Steven Ross, and John Patterson. Jazzing up Eclipse with collaborative tools. In *eclipse '03: Proceedings of the 2003 OOPSLA Workshop on Eclipse Technology Exchange*, pages 45–49. ACM Press, 2003.

[45] Shigeru Chiba. A Metaobject Protocol for C++. In *OOPSLA '95: Proceedings of the 10th Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 285–299. ACM Press, 1995.

[46] R. Chinnici, M. Gudgin, J.J. Moreau, J. Schlimmer, and S. Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language, 2004.

[47] H.B. Christensen. The Ragnarok software development environment. *Nordic Journal of Computing*, 6(1):4–21, 1999.

[48] Edgar F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[49] B. Collins-Sussmann, B.W. Fitzpatrick, and C.M. Pilato. *Version Control with Subversion*. O'Reilly, 2004.

[50] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys (CSUR)*, 30(2):232–282, 1998.

[51] J. Coutaz. Architectural design for user interfaces. In A. van Lamsweerde, editor, *Proceedings of the 3rd European Software Engineering Conference*, pages 7–22. Springer, 1991.

[52] W. Cunningham and B. Leuf. *The Wiki Way: Collaboration and Sharing on the Internet*. Addison-Wesley, 2001.

[53] J.W. Cupp. Reviewing the professionalization of software engineering: can small colleges remain viable? *The Journal of Computing in Small Colleges*, 17(1):132–146, 2001.

[54] K. Czarnecki and U. Eisenecker. *Generative Programming - Methods, Tools, and Applications*. Addison-Wesley, 2000.

[55] P.P. da Silva. User Interface Declarative Models and Development Environments: A Survey. In *DSV-IS '00: Proceedings of the 7th International Workshop on Design, Specification and Verification of Interactive Systems*, LNCS 1946, pages 207–226. Springer, June 2000.

[56] W. Dalitz and G. Heyer. *HyperWave : The New Generation Internet Information System*. Morgan Kaufmann, 1997.

[57] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 207–212. ACM Press, 1982.

[58] Jim D'Anjou. *The Java Developer's Guide to Eclipse*. Addison-Wesley Professional, October 2004.

[59] Juan de Lara and Esther Guerra. *Formal Support for Model Driven Development with Graph Transformation Techniques*. Proceedings published as Vol. 157 of CEUR, 2005.

[60] Juan de Lara, Hans Vangheluwe, and Manuel Alfonseca. Meta-modelling and graph grammars for multi-paradigm modelling in AToM^3. *Software and System Modeling*, 3(3):194–209, 2004.

[61] I.S. Deligiannis, M. Shepperd, S. Webster, and M. Roumeliotis. A Review of Experimental Investigations into Object-Oriented Technology. *Empirical Software Engineering*, 7(3):193–231, 2002.

[62] F.N. Demers and J. Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In *Proceedings of the IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, 1995.

[63] L.M. Díaz, E. Wüstner, and P. Buxmann. Inter-organizational document exchange: facing the conversion problem with xml. In *SAC '02: Proceedings of the 2002 ACM Symposium on Applied Computing*, pages 1043–1047. ACM Press, 2002.

[64] E.W. Dijkstra. Computing Science: achievements and challenges. *ACM SIGAPP Applied Computing Review*, 7(2):2–9, 1999.

[65] R. Douence and M. Südholt. A Generic Reification Technique for Object-Oriented Reflective Languages. *Higher-Order and Symbolic Computation*, 14(1):7–34, 2001.

[66] D. Draheim, J. Grundy, J. Hosking, C. Lutteroth, and G. Weber. Realistic load testing of web applications. In *CSMR'06: Proceedings of the 10th European Conference on Software Maintenance and Reengineering*. IEEE Press, 2006.

[67] Dirk Draheim, Elfriede Fehr, and Gerald Weber. JSPick - a server pages design recovery. In *7th European Conference on Software Maintenance and Reengineering*, LNCS. IEEE Press, March 2003.

[68] Dirk Draheim, Christof Lutteroth, and Gerald Weber. Factory: Statically Type-Safe Integration of Genericity and Reflection. In *Proceedings of the 4th International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*. ACIS, 2003.

[69] Dirk Draheim, Christof Lutteroth, and Gerald Weber. An analytical comparison of generative programming technologies. Technical Report B-04-02, Institute of Computer Science, Freie Universität Berlin, January 2004.

[70] Dirk Draheim, Christof Lutteroth, and Gerald Weber. Generative Programming for C#. *ACM SIGPLAN Notices*, 40(8), August 2005.

[71] Dirk Draheim, Christof Lutteroth, and Gerald Weber. Integrating Code Generators into the C# Language. In *Proceedings of ICITA 2005: The 3rd International Conference on Information Technology and Applications*. IEEE Press, 2005. to appear.

[72] Dirk Draheim, Christof Lutteroth, and Gerald Weber. Robust content creation with form-oriented user interfaces. In *Proceedings of CHINZ 2005 – 6th International Conference of the ACM's Special Interest Group on Computer-Human Interaction*. ACM Press, 2005.

[73] Dirk Draheim, Christof Lutteroth, and Gerald Weber. A type system for reflective program generators. In *Proceedings of GPCE 2005 – Generative Programming and Component Engineering*. Springer, 2005.

[74] Dirk Draheim, Christof Lutteroth, and Gerald Weber. Graphical user interfaces as documents. In *Proceedings of CHINZ 2006 – 7th International Conference of the ACM's Special Interest Group on Computer-Human Interaction*. ACM Press, 2006.

[75] Dirk Draheim and Gerald Weber. Modeling Submit/Response Style Systems with Form Charts and Dialogue Constraints. In *Workshop on Human Computer Interface for Semantic Web and Web Applications (HCI-SWWA)*, LNCS 2889, pages 267–278. Springer, 2003.

[76] Dirk Draheim and Gerald Weber. *Form-Oriented Analysis - A New Methodology to Model Form-Based Applications*. Springer, October 2004.

[77] Dirk Draheim and Gerald Weber. Specification and Generation of Model 2 Web Interfaces. In *APCHI 2004 - 6th Asia-Pacific Conference on Computer-Human Interaction*, LNCS 3101. Springer, June 2004.

[78] Dirk Draheim and Gerald Weber. A Qualitative Analysis of Emerging Collaborative Web Structures. Technical Report UoA-SE-2005-5, Software Engineering Programme, The University of Auckland, 2005.

[79] Dirk Draheim and Gerald Weber. End-User Development of Web Applications. Technical Report UoA-SE-2005-4, Software Engineering Programme, The University of Auckland, 2005.

[80] Dirk Draheim and Gerald Weber. Modelling Form-Based Interfaces with Bipartite State Machines. *Journal Interacting with Computers*, 17(2):207–228, 2005.

[81] S. Dutke and T. Reimer. Evaluation of two types of online help for application software. *Journal of Computer Assisted Learning*, 16(4):307–315, 2000.

[82] Eclipse Foundation. Eclipse IDE. http://www.eclipse.org/.

[83] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools*. World Scientific Publishing, River Edge, NJ, USA, 1999.

[84] R. Elmasri and S.B. Navathe. *Fundamentals of database systems*. Addison-Wesley, 2000.

[85] Jacky Estublier. The Adele configuration manager. pages 99–133, 1995.

[86] Jacky Estublier. Software configuration management: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 279–289. ACM Press, 2000.

[87] N. Fenton, SL Pfleeger, and RL Glass. Science and substance: a challenge to software engineers. *Software*, 11(4):86–95, 1994.

[88] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. RFC 2616: Hypertext Transfer Protocol–HTTP/1.1, June 1999.

[89] J. Fink, A. Kobsa, and A. Nill. User-oriented Adaptivity and Adaptability in the AVANTI Project. In *Proceedings of the Conference "Designing for the Web: Empirical Studies"*, 1996.

[90] D. Flanagan. *JavaScript: the definitive guide*. O'Reilly, 2002.

[91] D. Florescu and D. Kossmann. Storing and Querying XML Data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[92] M. Fowler and K. Beck. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[93] R.B. France, S. Ghosh, T. Dinh-Trong, and A. Solberg. Model-driven development using UML 2. 0: promises and pitfalls. *Computer*, 39(2):59–66, 2006.

[94] M.J. Franklin, M.J. Carey, and M. Livny. Transactional client-server cache consistency: alternatives and performance. *ACM Transactions on Database Systems (TODS)*, 22(3):315–363, 1997.

[95] R. G. Gabriel, D. G. Bobrow, and J. L. White. *Object Oriented Programming - The CLOS perspective*. MIT Press, 1993.

[96] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[97] R. L. Glass. Failure is looking more like success these days. *Software*, 19(1):104–103, 2002.

[98] R.L. Glass. The Software-Research Crisis. *Software*, 11(6):42–47, 1994.

[99] Don Gotterbarn, Keith Miller, and Simon Rogerson. Software engineering code of ethics. *Commun. ACM*, 40(11):110–118, 1997.

[100] J.P. Gray and B. Ryan. Integrating approaches to the construction of software engineering environments. In *Proceedings of the Eighth Conference on Software Engineering Environments*, pages 53–65. IEEE Press, 1997.

[101] J. Grundy and J. Hosking. Developing adaptable user interfaces for component-based systems. *Interacting with Computers*, 14(3):175–194, 2002.

[102] J.C. Grundy, J.G. Hosking, and W.B. Mugridge. Supporting flexible consistency management via discrete change description propagation. *Software Practice and Experience*, 26(9):1053–1083, 1996.

[103] Lars Grunske, L. Geiger, A. Zndorf, N. VanEetvelde, P. VanGorp, and D. Varró. *Using graph transformation for practical model driven software engineering*, pages 91–119. Springer Verlag, 2005.

[104] M. Gudgin, M. Hadley, N. Mendelsohn, J.J. Moreau, and H.F. Nielsen. SOAP Version 1.2 Part 1: Messaging Framework, June 2003.

[105] K.L. Halsted and J.H.J. Roberts. Eclipse help system: an open source user assistance offering. *Proceedings of the 20th Annual International Conference on Computer Documentation*, pages 49–59, 2002.

[106] L. Hatton. Does OO sync with how we think? *Software*, 15(3):46–54, 1998.

[107] B. Henderson-Sellers. UML–the Good, the Bad or the Ugly? Perspectives from a panel of experts. *Software and Systems Modeling*, 4(1):4–13, 2005.

[108] Ralf Hinze and Johan Jeuring. Generic Haskell: Applications. In Roland Backhouse and Jeremy Gibbons, editors, *Generic Programming: Advanced Lectures*, LNCS 2793, pages 57–97. Springer, 2003.

[109] Ralf Hinze and Johan Jeuring. Generic Haskell: Practice and Theory. In Roland Backhouse and Jeremy Gibbons, editors, *Generic Programming: Advanced Lectures*, LNCS 2793, pages 1–56. Springer, 2003.

[110] Chih-Wei Ho, Somik Raha, Edward Gehringer, and Laurie Williams. Sangam: a distributed pair programming plug-in for Eclipse. In *eclipse '04: Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology Exchange*, pages 73–77. ACM Press, 2004.

[111] Clifford C. Huff. Elements of a realistic CASE tool adoption budget. *Commun. ACM*, 35(4):45–54, 1992.

[112] E.L. Hutchins, J.D. Hollan, and D.A. Norman. Direct Manipulation Interfaces. *Human-Computer Interaction*, 1(4):311–338, 1985.

[113] Juhani Iivari. Why are CASE tools not used? *Commun. ACM*, 39(10):94–103, 1996.

[114] International Organization for Standardization. International Standard 10026-1:1992. Information technology – Open Systems Interconnection – Distributed Transaction Processing, 1992.

[115] International Organization for Standardization. ISO 9241-10: Ergonomic Requirements for Office Work with Visual Display Terminals (VDT) – Part 10: Dialogue Principles, 1996.

[116] International Organization for Standardization. International Standard 1975:1999. Information Technology – Database Language SQL, 1999.

[117] International Organization for Standardization. ISO/IEC 9075-3: Information technology – Database languages – SQL – Part 3: Call-Level Interface (SQL/CLI), 2003.

[118] International Organization for Standardization. ISO/IEC 19501:2005: Information technology – Open Distributed Processing – Unified Modeling Language (UML) Version 1.4.2, 2005.

[119] Internet Engineering Task Force (IETF) Network Working Group. RFC4122: A Universally Unique IDentifier (UUID) URN Namespace, July 2005.

[120] S. Jianling, D. Jinxiang, and H. Zhijun. Transparent Access to Persistent Objects in Object-Oriented Databases. In *TOOLS '97: Proceedings of the 24th Conference on Technology of Object-Oriented Languages and Systems*. IEEE Press, 1997.

[121] C. Jones. Patterns of large software systems: failure and success. *Computer*, 28(3):86–87, 1995.

[122] M. Jørgensen and K. Moløkken-Østvold. How large are software cost overruns? A review of the 1994 Chaos Report. *Information and Software Technology*, 48(4), 2006.

[123] Elisabeth Kapsammer, Thomas Reiter, and Wieland Schwinger. Model-based tool integration - state of the art and future perspectives. In *Proceedings of the 3rd International Conference on Cybernetics and Information Technologies, Systems and Applications (CITSA 2006)*, 2006.

[124] M.H. Kay. XML five years on: a review of the achievements so far and the challenges ahead. pages 29–31. ACM Press, 2003.

[125] W. Keller. Object/Relational Access Layers: A Roadmap, Missing Links and More Patterns. In *EuroPLoP'98: Proceedings of the 3rd European Conference on Pattern Languages of Programming and Computing*, July 1998.

[126] Chris F. Kemerer. How the learning curve affects CASE tool adoption. *IEEE Software*, 9(3):23–28, 1992.

[127] L. Khan and Y. Rao. A performance evaluation of storing XML data in relational database management systems. In *WIDM '01: Proceedings of the 3rd Annual ACM International Workshop on Web Information and Data Management*, pages 31–38. ACM Press, 2001.

[128] G. Kiczales. An overview of AspectJ. In *ECOOP '01: Proceedings of the European Conference on Object-Oriented Programming*, LNCS 2072, pages 18–22. Budapest, Hungary, June 2001.

[129] G. Kiczales and J. des Rivières. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[130] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *LFP '86: Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 151–161. ACM Press, 1986.

[131] R. Krishnamurthy, R. Kaushik, and J.F. Naughton. Unraveling the duplicate-elimination problem in XML-to-SQL query translation. *Proceedings of the 7th International Workshop on the Web and Databases: colocated with ACM SIGMOD/PODS 2004*, pages 49–54, 2004.

[132] Per Kroll and Walker Royce. Key principles for business-driven development, October 2005.

[133] C. Larman and VR Basili. Iterative and incremental developments. A brief history. *Computer*, 36(6):47–56, 2003.

[134] E. Lecolinet. A molecular architecture for creating advanced GUIs. *Proceedings of the 16th Annual ACM Symposium on User Interface Software and Technology*, pages 135–144, 2003.

[135] A.L. Lederer and J. Prasad. Causes of inaccurate software development cost estimates. *The Journal of Systems & Software*, 31(2):125–134, 1995.

[136] Henry F. Ledgard. Technical opinion: The emperor with no clothes. *Commun. ACM*, 44(10):126–128, 2001.

[137] Byong G. Lee, N. Hari Narayanan, and Kai H. Chang. An integrated approach to distributed version management and role-based access control in computer supported collaborative writing. *J. Syst. Softw.*, 59(2):119–134, 2001.

[138] Diane Lending and Norman L. Chervany. The use of CASE tools. In *SIGCPR '98: Proceedings of the 1998 ACM SIGCPR Conference on Computer Personnel Research*, pages 49–58. ACM Press, 1998.

[139] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml System Release 3.08 - Documentation and User's Manual. Technical report, Institut National de Recherche en Informatique et en Automatique, July 2004.

[140] B. Liskov and J. Wing. Family values: A behavioral notion of subtyping. Technical Report MIT/LCS/TR-562b, 1993.

[141] B. Lundell, B. Lings, A. Persson, and A. Mattsson. UML model interchange in heterogeneous tool environments: An analysis of adoptions of XMI 2. In *MoDELS 2006: Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems*. Springer, 2006.

[142] Christof Lutteroth. AP1: A platform for model-based software engineering. In *TEAA '06: Proceedings of the 2nd International Conference on Trends in Enterprise Application Architecture*. Springer, 2006.

[143] Christof Lutteroth. A Comparison of XML and Relational Database Technology. In *Proceedings of the 5th New Zealand Computer Science Research Student Conference*, 2007.

[144] Christof Lutteroth and Gerald Weber. User interface layout with ordinal and linear constraints. In *AUIC '06: Proceedings of the 7th Australasian User Interface Conference*, pages 53–60, Darlinghurst, Australia, Australia, 2006. Australian Computer Society.

[145] Christof Lutteroth and Gerald Weber. Reflection as a principle for better usability. In *ASWEC 2007: Proceedings of the 18th Australian Software Engineering Conference*. IEEE Press, 2007.

[146] A. Maccari and C. Riva. On CASE tool usage at Nokia. In *ASE 2002: Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, pages 59–68. IEEE Press, 2002.

[147] Boris Magnusson and Ulf Asklund. Fine Grained Version Control of Configurations in COOP/Orm. In *ICSE '96: Proceedings of the SCM-6 Workshop on System Configuration Management*, pages 31–48, London, UK, 1996. Springer-Verlag.

[148] Katsuhisa Maruyama and Shinichiro Yamamoto. A CASE tool platform using an XML representation of java source code. In *SCAM '04: Proceedings of the 4th IEEE International Workshop on Source Code Analysis and Manipulation*, pages 158–167, 2004.

[149] J. Masthoff and A. Gupta. Design and evaluation of just-in-time help in a multimodal user interface. *Proceedings of the 7th International Conference on Intelligent User Interfaces*, pages 204–205, 2002.

[150] H. Maurer. *Hypermedia Systems and Applications: World Wide Web and Beyond*. Springer, 1997.

[151] S. McConnell and L. Tripp. Professional software engineering: Fact or fiction? *IEEE Software*, 16(6):13–18, 1999.

[152] Karen L. McGraw. Defining and designing the performance-centered interface: moving beyond the user-centered interface. *Interactions*, 4(2):19–26, 1997.

[153] Brian McNamara and Yannis Smaragdakis. Static interfaces in C++. In *Proceedings of the First Workshop on C++ Template Programming, NetObjectDays 2000*, October 2000.

[154] J. Melton and A. Eisenberg. *Understanding SQL and Java Together.: A Guide to SQLJ, JDBC, and Related Technologies.* Elsevier, 2000.

[155] VJ Mercurio, BF Meyers, AM Nisbet, and G. Radin. AD/Cycle strategy and architecture. *IBM Systems Journal*, 29(2):170–188, 1990.

[156] Joaquin Miller and Jishnu Mukerji. MDA Guide Version 1.0.1. Object Management Group, 2003.

[157] K. Molokken and M. Jorgensen. A review of software surveys on software effort estimation. In *ISESE'03: Proceedings of the International Symposium on Empirical Software Engineering*, pages 223–230. IEEE Press, 2003.

[158] J.N. Morgan. Why the software industry needs a good ghostbuster. *Communications of the ACM*, 48(8):129–133, 2005.

[159] B.A. Myers, D.A. Weitzman, A.J. Ko, and D.H. Chau. Answering why and why not questions in user interfaces. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 397–406, 2006.

[160] P. Naur and B. Randell, editors. *Software Engineering: Report on a Conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968*. Brussels, Scientific Affairs Division, NATO, January 1969.

[161] Gregory Neverov and Paul Roe. Metaphor: A Multi-stage, Object-Oriented Programming Language. In *GPCE '04: Proceedings of the 3rd International Conference on Generative Programming and Component Engineering*, LNCS 3286, pages 168–185. Springer, October 2004.

[162] Tien N. Nguyen, Ethan V. Munson, John T. Boyland, and Cheng Thao. An infrastructure for development of object-oriented, multi-level configuration management services. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 215–224, 2005.

[163] J. Nielsen. Ten Usability Heuristics, 2004.

[164] L. Nigay and J. Coutaz. Building user interfaces: organizing software agents. In *Proceedings of ESPRIT '91*, volume 91, pages 707–719. ACM Press, 1991.

[165] D. Nizhegorodov. Jasper: Type-Safe MOP-Based Language Extensions and Reflective Template Processing in Java. In *Proceedings of the ECOOP'2000 Workshop on Reflection and Metalevel Architectures: State of the Art, and Future Trends*. ACM Press, 2000.

[166] D. Nizhegorodov. Code-Generation Aspects of Jasper, a Reflective Meta-Programming and Source Transformations Processor. In *Proceedings of the ECOOP '02 Workshop on Generative Programming*. ACM Press, 2002.

[167] D. Notkin, M. Gorlick, and M. Shaw. An Assessment of Software Engineering Body of Knowledge Efforts, May 2000. Report to ACM Council.

[168] Kristen Nygaard and Ole-Johan Dahl. The development of the SIMULA languages. *SIGPLAN Not.*, 13(8):245–272, 1978.

[169] Object Management Group. MOF 2.0/XMI Mapping Specification Version 2.1, September 2005.

[170] Object Management Group. Unified Modeling Language: Diagram Interchange, June 2005.

[171] Object Management Group. Unified Modeling Language: Superstructure, August 2005.

[172] Object Management Group. Meta Object Facility (MOF) Core Specification Version 2.0, January 2006.

[173] Object Management Group. Unified Modeling Language: Infrastructure, March 2006.

[174] Ohloh Corporation. Ohloh – Open Source Intelligence Service. http://www.ohloh.net/.

[175] Jacob Palme. A short presentation of the SIMULA programming language. *SIGSIM Simul. Dig.*, 5(4):19–19, 1974.

[176] S. Pangoli and F. Paterno. Automatic generation of task-oriented help. In *UIST '95: Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*, pages 181–187. ACM Press, 1995.

[177] David L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[178] E. Pelegrí-Llopart and M. Roth. JavaServer Pages^{TM} Specification, November 2003.

[179] I. Petrov and S. Jablonski. An OMG MOF based repository system with querying capability – the iRM project. *Proceedings of iiWAS'04*, 2004.

[180] G. E. Pfaff, editor. *User Interface Management Systems*. Springer, Secaucus, NJ, USA, 1985.

[181] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[182] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, 1995.

[183] Jon H. Pittman and Christopher J. Kitrick. VUIMS: a visual user interface management system. In *UIST '90: Proceedings of the 3rd Annual ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 36–46. ACM Press, 1990.

[184] T.E. Potok, M. Vouk, and A. Rindos. Productivity analysis of object-oriented software developed in a commercial environment. *Software Practice and Experience*, 29(10):833–847, 1999.

[185] B. Randell. The 1968/69 NATO Software Engineering Reports. *History of Software Engineering, Dagstuhl- Seminar*, 9635, 1996.

[186] JS Reel. Critical success factors in software projects. *Software*, 16(3):18–23, 1999.

[187] Richard Pawson and Robert Matthews. *Naked Objects*. Wiley, 2002.

[188] F. Robert, A. Abran, and P. Bourque. A Technical Review of the Software Construction Knowledge Area in the SWEBOK Guide. In *STEP02: Proceedings of the 10th International Workshop on Software Technology and Engineering Practice*. IEEE Press, 2002.

[189] J.M. Sagawa. Repository manager technology. *IBM Systems Journal*, 29(2):209–227, 1990.

[190] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[191] A.W. Scheer. *Aris-Business Process Modeling*. Springer, 2000.

[192] Alexandru Serban. *Visual SourceSafe 2005 – Software Configuration Management in Practice*. PACKT Publishing, February 2007.

[193] A. Shalit. *The Dylan reference manual: the definitive guide to the new object-oriented dynamic language*. Addison-Wesley, 1996.

[194] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 7–10, 1999.

[195] Chris Sheedy. Sorceress: a database approach to software configuration management. In *Proceedings of the 3rd International Workshop on Software Configuration Management*, pages 121–126. ACM Press, 1991.

[196] Ben Shneiderman. Direct manipulation for comprehensible, predictable and controllable user interfaces. In *IUI '97: Proceedings of the 2nd International Conference on Intelligent User Interfaces*, pages 33–39. ACM Press, 1997.

[197] Jeremy Siek and Andrew Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *Proceedings of the First Workshop on C++ Template Programming, NetObjectDays 2000*, October 2000.

[198] C. Skibo. *Working with Visual Studio 2005*. Microsoft Press, March 2006.

[199] Brian Cantwell Smith. Reflection and semantics in LISP. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 23–35. ACM Press, 1984.

[200] Software Engineering Institute. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Addison-Wesley Professional, 1995.

[201] V. Srinivasan and D.T. Chang. Object persistence in object-oriented applications. *IBM Systems Journal*, 36(1):66–87, 1997.

[202] G.L. Steele. *Common LISP: The Language*. Digital Press, 1990.

[203] J.E. Stiglitz. *The Roaring Nineties: A New History of the World's Most Prosperous Decade*. WW Norton & Company, 2003.

[204] B. Stroustrup. *The C++ programming language*. Addison-Wesley, 1986.

[205] Piyawadee Noi Sukaviriya, Jeyakumar Muthukumarasamy, Anton Spaans, and Hans J. J. de Graaff. Automatic generation of textual, audio, and animated help in UIDE: the user interface design. In *AVI '94: Proceedings of the Workshop on Advanced Visual Interfaces*, pages 44–52. ACM Press, 1994.

[206] A.G. Sutcliffe and N. Mehandjiev. End-User Development. *Communications of the ACM*, 47(9):31–32, 2004.

[207] W. Taha and T. Sheard. Multi-Stage Programming with Explicit Annotations. In *PEPM '97: Partial Evaluation and Semantics-Based Program Manipulation*, SIPLAN Notices, pages 203–217. ACM Press, June 1997.

[208] Walid Taha and Tim Sheard. MetaML and Multi-stage Programming with Explicit Annotations. *Theoretical Computer Science*, 248(1-2):211–242, 2000.

[209] Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 204–215. ACM Press, 2002.

[210] OASIS Web Services Business Process Execution Language (WSBPEL) TC. Web Services Business Process Execution Language (WS-BPEL) Version 2.0, April 2007.

[211] The Firebird Foundation. Firebird RDBMS. http://www.firebirdsql.org/.

[212] The Standish Group. The CHAOS Report. http://www.standishgroup.com/.

[213] J.B. Thompson. A Long and Winding Road (Progress on the Road to a Software Engineering Profession). In *Proceedings of compsac'01: the 25th Annual International Computer Software and Applications Conference*. IEEE Press, 2001.

[214] S. Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1998.

[215] W.F. Tichy. RCS - A System for Version Control. *Software – Practice and Experience*, 15(7):637–654, 1985.

[216] Juha-Pekka Tolvanen. MetaEdit+: domain-specific modeling for full code generation demonstrated [GPCE]. In *OOPSLA '04: Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 39–40. ACM Press, 2004.

[217] S.J. Vaughan-Nichols. XML Raises Concerns as It Gains Prominence. *IEEE Computer*, 36(5):14–16, 2003.

[218] Daniel von Dincklage. Making Patterns Explicit with Metaprogramming. In Frank Pfenning and Yannis Smaragdakis, editors, *GPCE '03: Proceedings of the 2nd International Conference Generative Programming and Component Engineering*, LNCS 2830, pages 287–306. Springer, November 2003.

[219] Felix Weigel, Klaus U. Schulz, and Holger Meuss. Exploiting native XML indexing techniques for XML retrieval in relational database systems. In *WIDM '05: Proceedings of the 7th Annual ACM International Workshop on Web Information and Data Management*, pages 23–30. ACM Press, 2005.

[220] B. Whittaker. What went wrong? Unsuccessful information technology projects. *Information Management & Computer Security*, 7(1):23–29, 1999.

[221] Å. Wikström. *Functional programming using standard ML*. Prentice Hall, 1987.

[222] S.N. Woodfield. The Impedance Mismatch Between Conceptual Models and Implementation Environments. In *Proceedings of the ER '97 Workshop on Behavioral Models and Design Transformations: Issues and Opportunities in Conceptual Modeling*, 1997.

[223] World Wide Web Consortium (W3C). Extensible Markup Language (XML) 1.1 (Second Edition), August 2006.

[224] Nicholas Wybolt. Perspectives on CASE tool integration. *SIGSOFT Softw. Eng. Notes*, 16(3):56–60, 1991.

[225] N. Yap, H.C. Chiong, J. Grundy, and R. Berrigan. Supporting dynamic software tool integration via web service-based components. In *ASWEC 2005: Proceedings of the Australian Software Engineering Conference*, pages 160–169. IEEE Press, 2005.

[226] Joseph W. Yoder, Ralph Johnson, and Quince Wilson. Connecting business objects to relational databases. Technical Report WUCS-98-25, Washington University, September 1998.

[227] E. Yourdon. *Modern structured analysis*. Yourdon Press Upper Saddle River, NJ, USA, 1989.

[228] N. Zhu, J. Grundy, and J. Hosking. Pounamu: a meta-tool for multi-view visual language environment construction. In *Proceedings of VL/HCC'04 IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE Press, 2004.

[229] István Zólyomi and Zoltán Porkoláb. Towards a General Template Introspection Library. In *GPCE '04: Proceedings of the 3rd International Conference on Generative Programming and Component Engineering*, LNCS 3286, pages 266–282. Springer, 2004.