Views and Concerns and Interrelationships

Lessons Learned from Developing the Multi–View Software Engineering Environment PIROL

vorgelegt von
Diplom-Informatiker
Stephan Herrmann

Von der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

genehmigte Dissertation

Promotionsausschuß:

Vorsitzender:	Prof. Dr.-Ing. Günter Hommel
Berichter:	Prof. Dr.-Ing. Stefan Jähnichen
Berichterin:	Prof. Dr.-Ing. Mira Mezini

Tag der wissenschaftlichen Aussprache: 23.9.2002

Berlin 2002
D 83

Contents

Zusammenfassung

Software-Entwicklungsumgebungen sind komplexe Systeme mit besonderen Anforderungen an Modularität und Anpaßbarkeit. Diese Arbeit beschreibt die Entwicklung der Umgebung PIROL. Die Beschreibung ist dabei in eine Abfolge der folgenden 12 Themen gegliedert. (1) *Metamodellierung* ist das Grundkonzept, nach dem PIROL seine Daten gemäß einem objektorientierten Datenmodell zerlegt, so daß beliebige Werkzeuge auch auf die Daten anderer Werkzeuge auf sinnvolle Art und Weise zuzugreifen können. (2) Das Metamodell wird zur *persistenten Speicherung* der Daten auf Konzepte des Repositories H-PCTE abgebildet. (3) Die *Granularität* eines Metamodells ist für die Effektivität und Effizienz des Gesamtsystems entscheidend. PIROL unterstützt hybride Modellierung als Kompromiß beider Extreme. (4) Durch *Methoden* des Metamodells wird Verhaltensmodellierung für verschiedenste Aufgaben unterstützt. (5) *Ausnahmebehandlung* wird systematisch unterstützt. (6) Verschiedene Mechanismen zur Wahrung der *Datenintegrität* sind enthalten. (7) Das System wurde nach einer *Client–Server* Architektur entwickelt, deren zentrale Komponente eine "Workbench" ist, die die Repository-Sprache Lua/P ausführt. (8) *Steuerungsintegration* erlaubt durch verteilte Steuerflüsse das enge Zusammenspiel lose gekoppelter Komponenten. (9) Die koordinierte Zusammenarbeit *mehrerer Benutzer* wird unterstützt. (10) Die logische Unabhängigkeit von Werkzeugen wird durch das neue Konzept der *Dynamic View Connectors* erreicht. (11) *Allgemeine Dienste* sind in der Umgebung einheitlich verfügbar. (12) Das System unterstützt die *Weiterentwicklung.*

All diese Themengebiete sind sehr eng miteinander verzahnt und die Darstellung ist zu großem Teil der gegenseitigen Beeinflussung gewidmet. Es wird gezeigt, wie eine Großzahl der Entwurfsentscheidungen genau aus diesen Beeinflussungen motiviert sind. Die Beschreibung folgt damit dem Konzept der "Concern Interaction Matrix", das hier zur Bewältigung von Komplexität vorgeschlagen wird. Dabei werden Charakteristika einzelner Anliegen und einzelner Zusammenhänge herausgearbeitet. Die Beschreibung PIROLs wird durch die Liste der integrierten Werkzeuge, Ansätze von Laufzeit-Messungen und einige Betrachtungen zur Beurteilung abgerundet.

Abschließend werden verschiedene Konzepte rund um den Begriff "Sichten" erörtert. Sichten sind ein zentrale Anliegen von PIROL. Außerdem generalisiert die Diskussion über die mehrdimensionale Darstellung des Hauptteiles. Es werden Begrifflichkeit, Konzepte und Techniken für *Sichten* in der Softwaretechnik vorgestellt und diskutiert. Dabei wird die Brücke geschlagen von Sichten in objektorientierten Datenbanken, über aspekt-orientierte Softwareentwicklung bis hin zum allgemeinen "Concern Modeling", zu dem die o.g. Methode einen Beitrag leisten soll. Sichten werden dabei als ein zentrales Konzept der Softwaretechnik neben Abstraktion und Zerlegung beurteilt. Dynamic View Connectors sind ein wesentlicher Beitrag von PIROL, durch den Datenbanksichten und aspektorientierte Programmierung zusammengeführt werden. Zwar ist der Sichten-Begriff längst nicht so scharf definiert, wie die Begriffe Abstraktion und Zerlegung, aber gerade die Überlappungen und Diskrepanzen, die durch Sichten abgebildet werden können, machen dies Konzept zu einem starken Strukturierungsprinzip, das zwar einigen Aufwand zur Behandlung von Inkonsistenzen erfordert, aber andererseits hilft, komplexe Systeme handhabbar und wartbar zu gestalten.

Abstract

Software engineering environments are complex systems with special requirements regarding modularity and adaptability. This thesis describes the development of the environment PIROL. The description is structured as a sequence of the following 12 concerns: (1) *Meta modeling* is the basic concept by which PIROL decomposes its data in accordance to an object-oriented data model. This allows arbitrary tools to access data of other tools in a meaningful way. (2) For *persistent storage* the model is mapped to the concepts of the repository H-PCTE. (3) The *granularity* of a meta model determines effectiveness and efficiency of the system. PIROL supports hybrid modeling as a compromise between extremes. (4) By *methods* of the meta model behavior modeling is supported for a wide range of tasks. (5) *Exception handling* is supported systematically. (6) Several mechanisms for preserving *data integrity* are integrated. (7) The system follows a *client–server architecture*. As its central component, the "workbench" executes the repository language Lua/P. (8) *Control integration* allows for close cooperation of loosely coupled components by means of distributed control flows. (9) The coordinated cooperation of *multiple users* is supported. (10) Logical independence of tools is achieved by the novel concept of *Dynamic View Connectors*. (11) *Common services* are available throughout the environment in a uniform way. (12) The system is prepared for *evolution*.

All these concerns are tightly interlocked. A considerable share of the presentation is dedicated to such mutual interactions. Is is shown, how a large number of design decisions is motivated exactly by these interactions. The description follows the concept of a "Concern Interaction Matrix" which is proposed for managing complexity. Characteristics of concerns and their interactions are elaborated. The description of PIROL is completed by a list of integrated tools, initial performance measurements and evaluation.

Finally, several concepts relating to the notion of "views" are discussed. Views are a central concern of PIROL. Furthermore, the discussion generalizes over the multi-dimensional presentation in the body of this thesis. Notions, concepts and techniques for *views* in software engineering are presented and discussed. This discussion connects views in object-oriented databases, aspect-oriented software development and general "concern modeling", to which the method of "Concern Interaction Matrices" contributes. Views are regarded as a central concept of software engineering at the same level as abstraction and decomposition. Dynamic View Connectors are a significant contribution of PIROL that combines database views and aspect-oriented programming. The notion of "views" is defined with far less precision than abstraction and decomposition, but indeed by the overlap and mismatches, which can be captured by views, this concept is a strong principle for structuring software and information. Effort is needed for handling inconsistencies as they may arise, but after all, views are a suitable means for managing the complexity of systems and for designing these systems for evolution.

Part I

Introduction

Separation and Composition of Concerns

I.1 Efficiently Managing Complexity

This thesis is about separating and composing different concerns that influence a complex software system. So where should we start? Begin with an in–depth description of the software crisis? Certainly that was the time, when the need for separation of concerns (SoC) has first been articulated[1]. But software engineering is no longer an unknown discipline whose goals and tools are yet to be defined. Many techniques and methods have been developed during the past 30 years, that address the goals of modularization and SoC. This might give rise to the hope that SoC has matured by now. Different factors, however, tell us, that there is still quite a way to go until SoC really meets its expectations.

First, software is still becoming more and more *complex*. The problems to be solved by software are more and more challenging, the stock of legacy software continuously grows, and the interaction between different pieces of software becomes more and more an issue. That is to say, the problems to be solved by software technology are still becoming harder.

Also the *process* of software development calls for better modularization: the parts of a software system are to be developed by more or less separate teams, maybe at different companies, in different countries, at different points in time. This forces to strive for more or less self–contained modules, that can be developed independently from other modules. Distributed development may also lead to heterogeneity. At the same time composing a concrete system from modules should enable to build a greater variety of products with less development effort, in order to reduce the time to market. That is to say, the circumstances of software development call for a much more efficient process.

This conflict eventually led to a second focus of research: not only the *separation* of concerns is an issue, but also their *composition* requires closer investigation if re-use of existing parts is to contribute to efficient development of complex systems.

Finally, recent work has brought to light, why there has still been such a large gap between the separated concern in an analysis model and the tangled

[1]This notion is commonly associated with Parnas [Par72], but none of his articles of that time actually uses the words separation of concerns (although this topic is really addressed). It is more likely, that Dijkstra coined the notion, which has, however, not been verified by the author.

designs and implementations that resulted from it. For analysis and informal descriptions it is no problem to dedicate different sections to different views and concerns. However, most formalisms which are precise and constructive enough for expressing a software design and its implementation allow decomposition only along one dimension. Usually, only one hierarchy can be expressed in a really independent fashion, dominating all other concerns that need to be broken down to smaller pieces, too. This “tyranny of the dominating dimension” [TOHS99] triggered the search for multi–dimensional separation of concerns. The attempt to find a common label for different related approaches led to coining the notion “Aspect-Oriented Software Development” (AOSD)[2]. Chap. 16 will report on recent approaches that can be attributed to this field.

I.2 CO^5: Five Levels of Discussion

This thesis combines constructive, empirical and theoretical considerations. Five different levels of discussion have been identified by the following “slogan”[3]

> CO^5: Confederatively Connecting Concerns as Components and Collaboration

Some parts of this thesis will punctiliously present technical details of the software engineering environment PIROL and its language Lua/P. At this level pitfalls and success stories are to be reported that outline the process of developing a family of component based software systems which by their very nature include a great variety of perspectives, concerns and dimensions. This is where the concrete *concerns* determining PIROL will be discussed. Self–application is inherent to software engineering environments. So the reader should not be startled by observations like: “tool support for separation of concerns is a concern dimension in PIROL”.

The next obvious level tries to generalize the findings and observations towards design and implementation of *components*. This notion is one of the most desired in its field, yet definitions significantly diverge. This discrepancy is not solved in a general way, but only with respect to the task at hand. Components are defined as a means to an end, specifically as a technique for encapsulating the concerns influencing a system under development.

Less obvious is the distinction between regular components and *collaborations*. This focus is intended to introduce a shift towards a more systemic understanding of software. Interaction between its elements is put into focus when considering software. The concept of roles is to abstract over concrete elements or individuals when describing recurring collaborative interactions. It is taken into account that the structure of interaction does not strictly follow any hierarchical breakdown. Of course comprehensibility is greatest when collaborations and structural hierarchy conform to each other, but exceptions to

[2]From the discussion at hand, a previous candidate, “Advanced Separation of Concerns”, appears to capture better what is at the core, but meanwhile “Aspect-Oriented” is too strong a notion, to be subsumed under any other label.

[3]This was the working title of this thesis for quite a while.

the idealistic picture have to be anticipated as well. It is important to regard not only structural patterns when defining adaptable components, but also to regard behavioral variability when defining collaborations thus striving for lingual capability to express adaptable patterns of interaction between elements of a software system.

Even less obvious is the need to introduce *connectors* as first class entities. So far, the decomposition of a problem has been over–emphasized. Connectors complete the picture as they express how the system is put together from its parts. Only with explicit connectors, the system assembly can be modified due to (1) maintenance, (2) run–time administration or even (3) self–reconfiguration.

The most abstract level of discussion deals with the overall system metaphor that guides the process of developing systems. In order to support this metaphor also the languages used for modeling and developing software systems should be reconsidered. The proposed model is said to be a *confederation* of concerns. This is to imply an understanding, where a structural breakdown in multiple hierarchies builds the back–bone of a system that includes a great variety of interaction patterns. Some concrete interactions may strictly adhere to the hierarchical structure while others deliberately cross-cut any structure. Drawing a parallel from constitutional structures, legislation, e.g., is pretty much top-down, whereas the participants in a law–suit can be any legal entities both at the suing and the defendant sides. One participant may even be a member of another or both may be groups, which overlap.

So, this thesis is trying to bridge the range from technical details up-to considerations of metaphors. Great care has to be employed when applying metaphors from the living environment or human societies to computer–related concepts and artifacts. No claim should be made, to really portray complex systems from the real world by software entities and models. In contrast, the foundation of the techniques and concepts to be presented here stems from the experience of constructing a concrete software system or rather a family of such systems. Any consideration drawing parallels to the real world should be regarded as a shy attempt to re–use findings from other fields and disciplines of research. It is not computer science that has invented complexity. Complex systems have been around before mankind entered the stage. Mankind has — sometimes unconsciously — added new forms of structure and organization, some of which are viable others are not.

Modeling reality in order to capture its essentials in software greatly concerns the question of how humans build their model of reality. The patterns behind social structures are one way to understand one class of complex systems.

I.3 The Software Engineering Environment PIROL

The technical and empirical background of this thesis lies in the development of a concrete system family: PIROL is a framework for multi–view software engineering environments (SEE). The development of PIROL had started as a student initiative at Technical University Berlin. Its ability to raise the enthusiasm of many diploma candidates stems from the universality of the original vision: to construct SEEs that by a few simple and elegant concepts integrate the many facets of software development in a way that reconciles the *separation* of different *views* along different dimensions with a tight and smooth *integration* thereof. A major guide–line in the development of PIROL has been the ECMA reference model for SEE frameworks [ECM93]. By building upon this reference model, a strong focus of PIROL has always been to allow *integration* along the five dimensions data, control, presentation, process and framework. PIROL has never been a funded research topic on its own. Within the ESPRESS project, PIROL has contributed to the development of a proof-of-concept environment for a special combination of the Z and Statechart notations including editing support as well as a set of tools for verification and validation [BGHHm98].

Most recently a confrontation of PIROL's problems, as they are common to SEEs, with current work in the AOSD field has inspired the adoption and adaptation of special *programming language techniques* in order to provide even better pluggability of tools into the PIROL environment. This fits well into the picture, that a centerpiece of PIROL's design is an object oriented repository language, Lua/P. This language has proven to be a powerful concept for universal solutions to recurring problems. Lua/P and PIROL's object oriented *meta model* — written in Lua/P— define the universe in which objects, documents and tools 'live'.

Research concerning SEE frameworks has always drawn profit from and rendered new results to many different research areas. Among these fields are: software architecture, database technology, component technology to name the most prominent ones. Another recurring theme in SEE development is the concept of different views. This usually means that different tools operate on different views of the underlying database or repository.

This thesis collects a multitude of observations and experiences that occurred during the development of PIROL. It tries to give a comprehensive (though certainly not complete) picture of very different techniques for separating and composing the concerns of the particular PIROL system, comparing the chosen techniques with related approaches from the respective fields.

Although drawing from the experience of a specific system, many observations will be generalized to arbitrary SEEs, to component based systems in general, and to the research of AOSD as such. A special focus is dedicated to interactions between different concerns: these are the issues, that render separation of concerns difficult.

I.3.1 Concerns and Dimensions of PIROL

When exploring the design space for SEEs, a central issue is to choose an appropriate *software architecture*. By choosing for PIROL a three–tiered repository architecture with both explicit and implicit invocation, important decisions are made concerning:

- Persistent storage and access to shared data.

- Localization of common functionality in the middle tier.
- The possibility to compose a system from independent programs (tools), running in separate processes.
- Designated responsibilities for user interaction and view updating.

Other design decisions were due in the choice of the *database technology* to use. Choices range, e.g., from relational over object based and EERA models up-to object oriented database systems. Concerns to be considered comprise expressiveness of schema definitions, retrieval support, and language mappings.

Based on the chosen database a set of common *system services* has to be considered including access control, tool synchronization, data consistency, change propagation etc. The main issue here was to smoothly integrate these services with other concerns providing the uniform flexibility where needed yet providing these services — as far as possible — in a transparent and automatic fashion.

Flexibility is a central requirement of an SEE framework. In order to deliver a noticeable gain, the open–closed principle of classes in object–oriented languages (cf. [Mey97]) has to be extended to tools and the repository: each component should be closed saying it should be ready-to-use. At the same time it should be open in that it should be easy to perform certain modifications.

Three dimensions of flexibility had to be distinguished: flexibility across

- development time, i.e. the system should be easily maintainable.
- the life–time of a database, i.e. the data in the database should remain valid and usable while the system still evolves.
- different contexts, i.e. different views of the same or overlapping data should co-exist without negative interference.

It will be shown, how the last kind of flexibility, when pursued consequently, leads to a concept of views, that provided the greatest challenge during the development of PIROL. This is where *Dynamic View Connectors* (DVC) will be introduced as a specialized programming language feature, that draws from recent AOSD technology.

Generalizing over DVCs and similar concepts, further *language issues* will be discussed regarding flexibility, reuse and modularization beyond the level of classes. Techniques that are covered by this discussion range from late classification, class–based versus object–based inheritance, and type-safe refinement, connection and adaptation of graphs of mutually recursive classes.

Furthermore, different aspects of *component integration* will be made explicit. This discussion will include the dimensions of (1) the overall communicational style (explicit or implicit invocation and stream-based communication), (2) different protocol layers contributing to invocation of operations and functions, and (3) mappings of the data to be exchanged, including the mappings of basic types as needed for inter-language working as well as structurally mapping mismatching data models.

I.4 From Separated Concerns towards a Confederation of Components

Along the road from separated concerns towards a confederative system of components, collaborations, and connectors, some short–comings of main–stream programming models and languages have to be identified. Existing proposals for solving some of these problems are very different in nature. We find systems, standards, frameworks, methods and last but not least advanced languages that all contribute to a greater expressiveness for the many facets of relevant complex systems.

I.4.1 Dealing with Partiality

Starting from toy examples software engineers are inclined to believe, that a good system description — at whichever level of precision, at whichever stage in the software life cycle — will be *complete* with respect to all relevant aspects. Implementation is believed to be the transformation of one complete model into another. Taking Coplien's definition of a large system — anything that is "larger than one mind"[Cop99] — this belief is shattered. Complex systems scrape the limit up-to which a human mind is capable of identifying all relevant aspects that contribute to a given element of a software model.

Evidence is arising that we have to live with partiality when describing software systems.

When separating a concern into a module it is evident that the module is only a partial definition of a system. As long as decomposition happens only along one dimension, explicit interfaces can be specified or deduced, which more or less completely define how a given piece fits into the whole. However, as soon as several dimensions of concerns are regarded for modular break-down, true independence between these dimensions can only be achieved if their concerns "know" nothing about each other and only later — say during a deployment phase — concerns are *applied* to each other. The final system behavior will be composed of different concerns and the concern interaction is invisible in most concern definitions. Thus a module is no longer just limited in its scope within the system structure, but also to one or more aspects. Each seemingly atomic operation in a module may at run-time be interwoven with behavior that is attached from another module. This description is deliberately simplified. Research has only achieved initial steps towards this goal.

The oldest tradition of partial *views* stems from the field of database technology. Sect. 16.1 will summarize representative approaches to specifying views of different kinds of (post-relational) databases. Some of the criteria to be discussed are:

- Is a view read–only or can it be modified in a way that the original data is changed and re-computation of the view will afterwards yield the new, changed view?
- Are changes to the base automatically propagated to all its views?

- What kinds of derivations are allowed for defining a view? Answers will range from simple predicate–based filters up-to powerful re–arrangements of data.

- Is the complete database schema defined a-priori and are views derived from that global schema, or can views *add* to the global schema?

Another area employing the concept of views is the *Model–View–Control* (MVC) architecture (originally called a 'paradigm'). At given places, the close relation between architectural issues (MVC as a software architecture) and language issues (implementing MVC using implicit invocation) will be discussed.

Putting together some advanced database view concepts and the behavioral aspect, lay the grounds for recent programming models like *Subject Oriented Programming* or some techniques for *Collaboration Based Design.* Although most of these models don't emphasize the concept of views, notions like *subjectivity* allude to the same partiality as is expressed by views. In this fairly new area of research, terminology is not yet established in a consensual way. So from a high–level perspective the notions of aspects, concerns, dimensions etc. still need to be reconciled, but also on a more technical level the provided mechanisms call for a common theory laying down a complete set of choices in language design from which concrete programming models can be explained in a uniform terminology. In this respect, the rôle of PIROL is in exploring the usefulness of existing concepts by adaptation to the specific conditions of an SEE. Lua/P contains a few very *specialized* constructs. It will be shown, how experience from Lua/P contributed to another prototypical language called LAC and finally into a *general purpose* programming model called Object Teams, which aims at converging notions and techniques from different AOSD approaches. Other than in an outlook, Object Teams are, however, beyond the scope of this thesis.

After deciding on how to deal with partiality, the subject of *completeness* has to be re-addressed:

- Some view models still start with a complete system and only derive views from it.

- Tool support may be thought of, by which completeness of a model made up of partial views can be checked.

- The construction process may ensure some degree of completeness when defining the connectors that assemble partial components to a system. This should of course be supported by tools, too.

- Robust systems can be thought of, that discover situations of missing specification/implementation at run-time and recover by means of some fallback mechanism. This depends on the existence of meaningful default behavior that goes beyond writing "method not understood" to the console.

I.4.2 Striving for Larger Modules

During the past few years much has been written about software *components*. The problematic nature of this notion, caused a plethora of diverging definitions, has been mentioned before. However, the reason for wanting components is obvious: larger pieces of software should be more independent and easier integrated than before, in order to achieve higher levels of re-use. The problems with components are obvious, too: re-usability and independence call for closed, self contained components while composability calls for adaptability and openness of components. At the class level this dilemma is known – and to some extent solved – as the open–closed principle [Mey97]. It has not yet encountered a genuine, universal solution at any higher level of modularity.

Observations hint at the possibility that no single answer will solve the dilemma for all cases. At the same time, hints are given that techniques for defining component interfaces need additional concepts aside from those already covered by classes. Transition systems may be added for specifying state–based behavioral aspects. Another important notion turns out to be the concept of *collaborations*. The difficulty with collaborations may be the fact, that they don't always fit into a hierarchical break-down.

This thesis claims that, in order to reach a more powerful notion of components, it is necessary to give up some overly strict requirements to component interfaces – i.e. components may not be purely black boxes[4] – as well as softening the boarder between systems with a strict hierarchy and systems with no hierarchy at all. The goal should be to develop a technical concept that corresponds to the metaphor of confederated sub-systems. This thesis contributes to this goal mainly by its investigation on the notion of views.

I.5 Organization of this Thesis

This thesis is about identifying and separating the concerns of integrated environments and about the interactions that occur between these concerns. It covers these issues mainly in terms of technical solutions that help to build systems with the required properties of modularity.

When describing a complex system in natural language similar difficulties arise with respect to structuring the text. In this sense also the structure of this thesis is an experiment in multi–dimensionality. However, the structural requirements for a natural language text differ significantly from development documents and software code. Compilers and related tools are far better in dealing with circular dependencies and forward references. These tools need not *understand* those parts that have not yet been defined, they just need signatures in order to check static correctness. A human reader, however, is easily confused if all early chapters in a book heavily refer to notions to be defined in later chapters. Thus, the seemingly natural structure of dedicating one chapter to each concern in the system, is only useful in the ideal case, where each concern can be explained independent of any other concern in the system.

[4]Confer the discussion in [Kic94] ("Why are black boxes so hard to re-use").

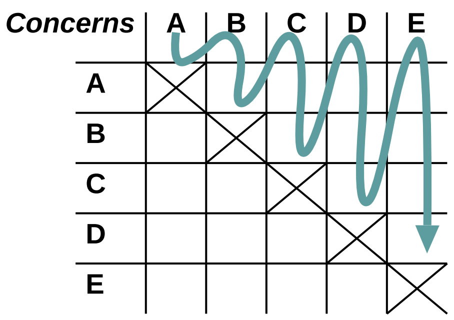


Figure 1: Path of describing concerns and their interactions

I.5.1 Description guided by a Concern Interaction Matrix

This thesis takes a different approach: following the path of evolutionary development, the description starts with a small core of concepts, that in itself defines a self–contained universe. Then as additional concerns are introduced to the discussion, each concern is first defined by itself and then related to (almost) each concern that has already been presented. The idea is to incrementally fill a concern matrix, which for each pair of concerns tells whether and how they relate to each other. We will call this matrix the *Concern Interaction Matrix*. Figure 1 visualizes this process. This really sounds like bad news, like resigning from the goal of keeping different concerns independent of each other. There is four answers to this.

1. When distinguishing between different layers of a system, those parts that are closer to the bottom of a system (or a hierarchy of abstractions), seem to have tighter interactions and even conflicts between concerns than those towards the top (or more abstract) layers. Those layers that contribute to a common infrastructure tend to be more complex than those components that define application logic and the user interface. The latter components may be large, but complexity is greatest where all different concerns meet.

   In other words: if complexity may migrate from client code to infrastructure components this may introduce hard problems for the development of the infrastructure, but the client code will eventually be easier to develop and maintain. So development of infrastructure is *intensive* while development of client code tends to be *extensive*.

2. This Concern Interaction Matrix should be considered in completeness, but it will occur that some cells in the matrix remain empty, signifying no interaction between the given concerns. It is still important to also visit these cells and to document why there is believed to be no interaction.

3. For many cases that bad news is true: complexity cannot be *eliminated*. All approaches to be presented in this thesis rather help to *manage* complexity. Standard solutions may be provided for standard concerns. Universal solutions may be a starting point for capturing arbitrary concerns, but for many cases the best help that can be provided may be a method

for managing the complexity that is inherent to the problem at hand and cannot be reduced by known techniques. The concept of a Concern Interaction Matrix may be a first step towards this method. At given points this concept will be contrasted with the notion of multi–dimensional separation of concerns and it will be examined to which extent a matrix may be used in fact to reduce a multi–dimensional space to two dimensions.

4. The limits of separation of concerns can still be pushed. This is a central issue of this thesis and also a fairly young research issue. After gathering experience with enhancing modularity of models and implementation, it is quite likely that documentation and texts like this thesis will eventually also benefit from these new understandings and techniques.

Most considerations of this thesis follow the given case study. The description of the software engineering environment PIROL will be the skeleton of this thesis. It will follow the evolutionary approach mentioned above. Starting with a small core of concepts the thesis will be organized as a spiral that as it moves out from the core incrementally widens the horizon and in each iteration revisits all concerns that had been introduced in previous iterations.

The following chapters of this thesis each capture one iteration in this process, so naturally the first chapters are quite short, and as the description moves out, chapters will grow. This is the order of concern dimensions:

1. An object oriented meta model for a seamless life cycle.
2. Persistence
3. Granularity of the data model
4. Behavioral modeling
5. Error handling
6. Data integrity
7. Client–server architecture
8. Control integration
9. Multi–user capability: sharing, protection and communication
10. A-posteriori integration of components
11. Common services across components
12. Evolution of PIROL

This path will be a somewhat idealistic view of the actual development of PIROL. Naturally, the actual order of introducing new features to the environment was not optimal. So at some places this thesis reorders things, but the structure of description essentially corresponds to the process of PIROL's incremental development. Also, the list of concerns was not defined in one go, but an appropriate set of concerns was only found while writing this thesis. Some concerns candidates where dropped from this list, because they turned out to be outside the technical core of PIROL. Other concerns only became explicit when analyzing intricate interactions between other concerns. In situations, where a well structured description of concerns seemed impossible, this could be solved by splitting one concerns into two, or introducing an additional concern.

The discrepancy between original development and documentation should not alarm us, it is in fact backed by Parnas' advice of faking a Rational Design Process [PC86].

The following issues are relevant, too, but no chapter has been dedicated to them:

- Repository objects versus documents,
- Process modeling,
- Static checking of Lua/P,
- Performance.

Some of these are relevant throughout, like objects–versus–documents and performance. Process modeling will be touched at places but appears to be fairly decoupled from the technical core. Static checking of Lua/P was planned, but within its prototypical setting found no appropriate solution.

I.5.2 Towards a comprehensive notion of views

The different levels of Part II will be blended into a generalized elaboration on the concept(s) of "views" within software engineering. Part III will relate views to abstractions and decomposition/composition; concepts, which are claimed to be of similar universality in software engineering. By analyzing different technologies for views — mainly from the areas of databases and programming languages — differences and similarities are identified.

I.5.3 How to read this thesis

This thesis combines different threads of argumentation into one linear story. Readers interested in only one part of these considerations may follow different paths through the document. Aside from four identifiable paths, the reader may always use the references at the margin, to move around in this document, which is facilitated by hyperlinks in the PDF version.

One thread of Part II is the description of the PIROL system. This thread is found mainly in the introductory part of each chapter. Within this thread a few sections that are marked "Background" — just like this paragraph — present the base technology used for implementing PIROL.

Background ▽

Also related work is interwoven with the presentation and discussion of PIROL. This is done, because the way in which other research relates to PIROL is very different for different works. There is little coherence between different cited publications other than by tying each publication to how things are solved

Related Work ▽

in PIROL. Related work will mostly occur towards the beginning or end of a chapter.

The third thread deals with analyzing how the different concerns interact with each other. This thread is made up of one section within each chapter titled *"X interacting with Y"*. Numberings of subsections therein correspond to the chapters to which they relate.

New Feature ▽

Within this thread sections are marked as either "Discussion", "Applications" or "New Feature". Only the latter will introduce new mechanisms. Sections marked "Applications" apply previously introduced mechanisms to the domain model (with the domain being "Software Development" this is PIROL's meta model). Sections marked "Discussion" introduce no new content and can safely be skipped when focusing on PIROL rather on the topic of concern interaction.

The forth thread summarizes the observed styles of concern interaction (*Section "Summary" within each chapter*). These summaries will be of most importance for consecutive parts, which will elaborate on general issues of managing concerns. Sect. 14.4 will collect the findings of these chapter summaries and draw further conclusions.

Part III elaborates on different uses of the word "views" in software engineering. This is to some extent a self-contained discussion but also tries to blend the following two levels: (1) PIROL as an application aims at *supporting* different views during software development. (2) The realization and documentation of PIROL *applies* several techniques for views and related concepts. Throughout Part III, discussion of findings from the literature will prevail, thus markings of different purposes of different paragraphs are not used there.

Part II

The Development of PIROL

Chapter 1

A Common Object-Oriented Meta Model for Seamlessness

The original motivation of PIROL is linked to the promise of object-oriented development to provide for *seamlessness* between the phases of the software life-cycle. The notions of objects and classes are said to provide a common conceptual basis enabling all phases to speak the same language [CY91, Mey97]. This is said to be a major benefit of object–orientation as compared to more traditional structured development. In fact, earlier approaches requested to create several documents that had no clear connection to each other. E.g., a functional decomposition by some kind of flow diagram had no mapping at all to the modular break–down as shown in a module diagram.

The focus on seamlessness has led to some over–simplifications that restricted the expressiveness of object-oriented models. More expressive language models in the field of aspect-oriented software development, show that classes are not the ultimate modules and inheritance and aggregation are not sufficient for certain kinds of module composition. On the other hand such languages introduce module interdependencies that are far from being obvious.

AOSD and AOP [16.2,16.3.2→]

Yet, the object-oriented approach helps to bring together at least the late phases of software development. In the early days of PIROL, also a *reference glossary* was favored at TU Berlin as an early means to capture the problem domain [Rei92]. Combining the concept of reference glossaries with object-oriented ideas results in classifying glossary entries, e.g., into classes, methods etc. *Data dictionaries* [CAB$^+$94, HLN$^+$90] of other approaches pursue the same goal. PIROL's primary contribution is, to bring together tools for different development phases, relating a glossary entry for a class to its views in analysis, design and implementation. Information is to be shared between phases, and links should make navigation between different documents really easy.

The technique chosen for integrating different phases and their tools is *meta modeling*, which is meanwhile well accepted for different purposes. Intentions of meta models range from language definition at different levels of formality and completeness to database design and tool integration. The "UML semantics" [OMG99], despite of its title, uses meta modeling only to define the structure of valid UML models. [Kla00] and related approaches use meta models to precisely define the semantics of a formalism. The notion "meta model" refers to any model of a formalism that in turn is used for modeling some application domain. In our closer context the meta model developed for PIROL is a model of object-oriented models. It tries to unify the concepts of different languages and notations. [Pet00] analyzes a set of object-oriented programming languages (*OOPL*), leading to a refined meta model that comprises the major constructs of many OOPLs. The complexity of the full meta model, which is not discussed here, reflects the differences between OOPLs that are far from being trivial, although some "marginal" OOPLs are already excluded from the analysis (none of the languages is an extension of a functional language (like CLOS [Kee89], Objective ML/OCaml [RV98]), none is prototype based (like Self [US87]) etc.).

The actual model of object-oriented models constitutes the `PRODUCT` package of PIROL's meta model. It was recognized early [Gro94] that also the development *process* may generate data that are to be stored in the repository. This yields a package `PROCESSES` defining entities such as `PROCESS_STEP` and `TASK`, `PERSON` and `GROUP`, `STATE` and `TRANSITION`, etc.

The question of integrating the `PRODUCT` and `PROCESSES` packages (and other packages to be introduced later) is taken care of by the root package `GENERAL`. Here a class `ANY_RO` is introduced as super–class for most other classes, which allows to link administrative and process–related objects to each product–related object.

1.1 Objects versus Files and Documents

Meta modeling interacts with persistence [→Sect. 2.2.1]

In order to clarify some terminology, an anticipation of the next chapter (Persistence and Object-Oriented Programming) will show the direction that guides the development of the meta model. The primary goal of PIROL's meta model is to enable software development tools to store artifacts in a common repository such that different tools can share information through the repository. An object in the repository will be called *RO* (repository object). Accordingly, the meta model is the RO class model (*ROCM* for short).

The repository is a replacement for a structure of directories in which more traditional tools store their artifacts as *files*. Meta modeling is used to make *public* the structure of artifacts. In the file based approach, each tool is responsible for the structure of its own files, without any external promise, that this structure will remain valid between versions of the tool. In the repository based approach this responsibility is centralized, which has the advantage that a tool may rely on the structure of data that has been stored by another tool. The drawback of this approach is its reduced flexibility with respect to evolution of tools and the meta model. We will address this conflict later.

Dynamic View Connectors overcome this conflict [10→]

It's not only the structure that is to be shared through the repository, but the meta model is supposed to define the *semantics* of software artifacts. This raises two issues that will be discussed in subsequent chapters. First, it is the question of granularity that determines how much "meaning" can be put into a meta model. Certainly, directories and files without public structure carry very little meaning. But, should each letter in a document, each line in a

Granularity [3→]

diagram be an RO, or where should we stop in decomposing a document into objects? Second, mere data modeling might give too much room for different *interpretations* by different tools. One step towards a more meaningful meta model is attaching behavior to ROs, other possibilities will be discussed in terms of consistency constraints.

Behavior [4→]

Data integrity [6→]

Decomposition of documents into objects is perhaps the most fundamental design decision of PIROL. In order to overcome the lack of files in our approach a virtual file system will be presented. Finally, the handling and retrieval of information in the repository is very different from handling files, because of the much larger number of ROs as compared to files (and because a repository contains an arbitrary *graph* of ROs as compared to the strict *hierarchy* of directories in a file system). In order to retain manageability of data in a repository, documents are reintroduced as *handles* by which sets of ROs can be extracted, displayed and manipulated by a tool. In the PIROL repository such documents are called *conceptual objects* (CO for short).

Virtual file system [13.4.1→]

1.1.1 Conceptual Objects

Conceptual objects are modeled by a specific RO class `CO`. Each CO represents a document and for this purpose maintains a set of `contained_objects` that contribute to that document. Of course, also a CO is a persistent object, but it represents a document which *as such* is not a persistent object, because it is decomposed into a number of ROs. COs are meant to hide this decomposition allowing the user to still work with an intuition of documents.

Considerations have been put forward, that the set of `contained_objects` should not be defined by an explicit set but by a *query*. Our current experience, however, states that such strategies of selecting all objects contributing to a document are more easily implemented within each tool, than by a generic query mechanism. So, currently the tools are responsible for maintaining the `contained_objects`–set while the user edits a document. The query approach might, however, be a useful future extension.

During the design of PIROL's meta model the question arose, where to put that information that pertains to *presentation* only. This can be described roughly as *layout*–information. PIROL's design confines ROs to *semantically relevant* information. ROs should contain only information that might possibly be meaningful beyond the scope of any single document. This excludes data like coordinates in a diagram, font information and the like. Still this information needs to be persistent. This dilemma is solved by assigning all layout information to the CO representing the document. For this purpose COs are implemented as resource managers, that may store additional properties for each contained RO. This happens in a hashtable like fashion, where a compound key, made of an RO and a property name, is used to index the table, that contains property values. The data structures used for this mechanism will be presented in Sect. 2.2.1.

For instance, a CO for a UML class diagram encapsulates graphical information about the (`CLASS`) ROs included in the diagram such as their positions, fonts, expand/collapse flags, etc. COs have an open structure: a tool may store

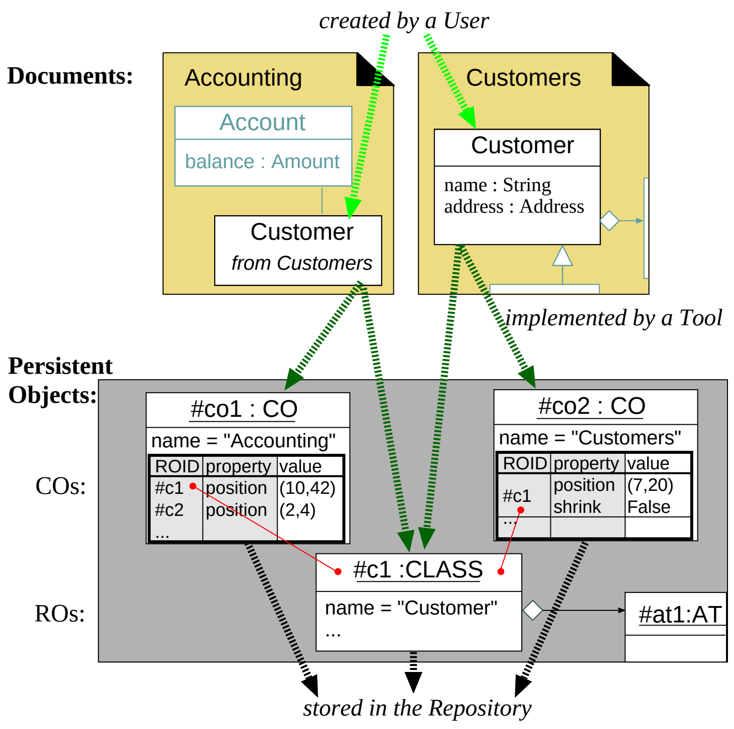


Figure 1.1: Documents are implemented as COs

arbitrary properties for any RO without prior declaration. To sum up, one could interpret a CO as a guide on how to "transform"a graph of ROs starting at a given root RO into a document.

For illustration consider the scenario presented in Fig. 1.1. In the upper part of the figure two documents are shown as they are displayed to the user of a graphical editor, representing two different packages, `Accounting` and `Customers`. The symbol for the class `Customer` appears in both diagrams (it is defined in one package and used in the other). The lower part of the figure shows a subset of the persistent objects that represent the two documents in the repository. Each `Class` object manipulated by the tool is represented by two different objects in the repository. Intrinsic information is stored within a `CLASS` object, while tool–specific properties are stored within conceptual objects, one for each document. E.g., the appearances of `Customer` in the `Accounting`, respectively `Customers` class diagrams, correspond to two RO–CO pairs in the repository, {`#co1, #c1`} and {`#co2, #c1`} respectively. The first line in `#co1` reads: `CLASS` object `#c1` is drawn at position (10,42) within the first document.

Dynamic View Connectors [10⟶]

This approach is developed from a database perspective and appears a little bit clumsy from the tool perspective. Dynamic View Connectors define a uniform abstraction layer on top of ROs and COs that smoothes this distinction.

COs are implemented as *cascaded* resource managers, i.e., each CO may refer to a `master` CO from which it "inherits" its properties. This technique has not been used to date.

1.2 Extending Lua for Object-Oriented Programming

The language to express PIROL's meta model is to be an OOPL. However, in order to be free to add concepts to this language as needed, two special purpose languages have been developed. The first prototype was called *DROSSEL* (descriptive reference object system structure extension language [Her94]). For the sake of fast prototyping it was based on a language that was easily available for extending: Tcl (tool command language [Ous94]). The lack of typing and a suitable syntax eventually led to a re–implementation using again an "extension language". This time Lua [IdFC96] was used and extended to Lua/P (Lua for PIROL). Using Lua proved far more successful than Tcl. In fact, Lua/P [Her00] turns out to be the central concept of PIROL by which most other techniques are integrated. Lua/P is not a general purpose OOPL, but a *repository language*, i.e., a domain specific language geared for programming a repository. The Lua/P interpreter, eventually called the PIROL workbench, is the centerpiece of PIROL.

Syntax of Lua/P. Although many issues of this thesis relate to the design of the language Lua/P, its *concrete syntax* should not be over emphasized. Most of the syntax used is in fact syntax of plain Lua. Lua's technique of associative arrays [IdFC96](see below) provides a flexible way for "declarative" programming, but this approach is bounded by the existing parser. Only small modifications are performed by a little *preprocessor* that is written in Lua and part of the workbench. This preprocessor is just a prototypical workaround used to allow a more intuitive syntax in a few places, where Lua cannot provide fully satisfactory solutions. Preprocessing is not totally robust, but this problem should vanish once a separate type checker for Lua/P exists, that could of course also perform the task of preprocessing. The current syntax of Lua/P can be found in Appendix A.1.

Discussion on type checking [14.2.3→]

So, Lua is the glue that *integrates* many concerns and techniques, while others are implemented anew in Lua and finally in Lua/P. In the following, the essential concepts will be presented that were used for the development of Lua/P on top of Lua.

1.2.1 Imperative core

Background
▽

The basic operational model of Lua is a very simple Pascal–like model. Programs consist of *blocks* that can be `functions` or control structures like `if` and `while`. Basic statements are assignments and function calls. A function call may be used as an expression (function) or as a statement (procedure), which should correspond to functions that do, resp. don't return a value. This correspondence is, however, an example of Lua's tolerance: functions with return value may also be used as a statement, ignoring the return value.

Variables don't need to be declared before use. Any undeclared variable is considered global. A variable may be declared `local`, which restricts its visibility and life time to the current block.

```
1  t = {}
2  t[1] = 2
3  t[2] = {}
4  t[3] = t[1]
5  t["name"] = "Joe"
6  t["age"] = 14
```

(a) Regular usage of a table

```
7  t = {}
8  t["Berlin"] = 1
9  t["London"] = 1
10 if t["Paris"] then
11     print("France")
12 end
13 if t["Berlin"] then
14     print("Germany")
15 end
```

(b) Table used as set

```
16 t = {}
17 t.name = "Joe"
18 t.age = 14
19 write(t.name.."is ")
20 print(t.age.."years old.")
21 if t.age == t["age"] then . . .
```

(c) Table used as record

```
22 t = {
23     "spring", "summer", "fall", "winter";
24     length = 4
25     elementtype = "string"
26     wordlengths = { 6, 6, 4, 6 }
27 }
```

(d) Manifest table

Explanations:	1,3,7,16:	{} creates an empty table.
	10, 13:	conditionals interpret every non-`nil` value as true.
	19, 20:	".." is the concatenation operator for strings.
	21:	always true (different syntax for the same expression).
	23:	list part with keys: 1,2,3,4, values: strings.
	26:	key: string, value: table (used as list or array)

Figure 1.2: Syntax of using Lua tables

1.2.2 Types

While variables have no (static) type, runtime values have distinguishable types with only a few automatic conversions like printing a number using its string representation. Lua basically supports four data types: `string`, `number`, `table` and `function`. Only strings and numbers have value semantics. Tables, functions and userdata (to be presented later) are passed using reference semantics.

The concept of *tables* unifies different structures like array, list, set, hashtable and record. This kind of data structure has been called "associative array" [IdFC96]. Different usage patterns turn such tables into a variety of complex data structures and a few small syntactic enhancements allow a very expressive style. At the implementation level a table is an ordinary hashtable, but its values *and* hash keys may be of any type (except `nil` keys). The standard ways of creating and accessing a table are shown in Fig. 1.2(a).

Using (integer) numbers as keys allows to use a table as a (dynamically growing) *array* (cf. lines 2–4 in Fig. 1.2(a)). By means of appropriate access functions such an array may also be used as a *list*. The standard functions `tinsert` and `tremove` in fact suffice for this task.

When using a table as a *set*, the set elements are used in position of the hash key and the associated values are either 1 (included in set) or `nil` (cf. Fig. 1.2(b)). Note, that there is no difference between a table slot, whose value is set to `nil` and a slot that has never been allocated.

If strings are used as hash keys, an alternative syntax emphasizes that this simulates *records* (cf. Fig. 1.2(c)). The value in a table slot may be of any type. This allows to nest tables at arbitrary depth. Different types may be combined within one table, resulting in complex structures and possibly in mixtures of the standard structures array, list, set, hashtable and record.

A table may be created and initialized in a declarative way (cf. Fig. 1.2(d)), such that a list part (implicit consecutive indices) and/or a record part can be given directly. Also nesting is allowed in manifest tables.

It is part of Lua's concept of *reflection* to allow iteration over all entries in a table yielding all $(key, value)$ pairs in the table. Similarly all global variables can be visited, which alludes to the global name space being a global, unnamed table.[1] Finally, the type of any value can be queried using the builtin function `type`.

1.2.3 Functions values

Functions are regular values that can be stored in variables or tables and can be passed as function arguments. The standard function definition

function name (a1, a2) *statements* **end**

is in fact just syntactic sugar for defining an anonymous function and assigning the function to a global variable:

name = **function** (a1, a2) *statements* **end**

When defining a function within a block, certain scoping issues have to be observed: local variables of the enclosing block may only be used when accessed as so-called upvalues. The `%` operator freezes the value of a local variable at the time of evaluating the function definition. This results in a *function closure* consisting of a function and an environment of frozen upvalues. Such a function closure can safely be passed outside the scope of its definition without danger of accessing obsoleted local variables. When calling a function closure, only the explicit arguments need to be passed. These mechanisms suffice to implement any style of higher order functions. E.g., a predefined function `foreach` allows a compact style of table iteration. Figure 1.3 shows how a function `map` can be implemented and used.

Another syntactical issue prepares Lua for object-oriented programming although plain Lua has no distinguished concept of objects or classes. Functions may be defined with the special syntax of using a colon as separator between a table and its field name (cf. Fig. 1.4(a)). Such a function can be called like methods in object-oriented languages. Figure 1.4(b) shows how the interpreter sees the code: at the definition side, the colon has the effect of adding an

[1] During the transition from Lua 3.2 to 4.0 this fact has been made explicit by a function `globals()` which returns as a table all global variables. After this transition, traversing global variables no longer needs a special function.

```
function map (list, func)
   local new_list = {}
   foreach (list,  function (index, value)
                        %new_list[index] = %func(value)
                     end
   )
   return new_list
end

alist = map({3,4,8}, function (num) return num * 2 end)
foreach (alist, print)
   ⇝ 1    6
     2    8
     3    16
```

Figure 1.3: Example of higher order functions in Lua

```
function table:function_name (a1, a2)
   statements
end


table:function_name (a1, a2)
```

(a) how to write and use a method

```
table.function_name =
   function (self, a1, a2)
      statements
   end

table.function_name (table, a1, a2)
```

(b) how it is interpreted

Figure 1.4: Methods in Lua

argument `self` to the front of the argument list. At the calling side the "object", i.e., the table at the left of the colon is inserted into the argument list.

These merely syntactical transformations allow to access the target of a "method call" as an implicit argument called `self`. The only price for this very light–weight convenience is the danger of confusing the dot and colon notations, which will not be discovered by the interpreter, because the resulting mismatch between formal and actual argument lists will be padded with nils.

1.2.4 Integration of client libraries

One of the primary goals of Lua is to act as an extension language, i.e., a language that can be added to any application, allowing for configuration and macro programming along certain functions that are exported from the application to Lua. This concept has first been made popular by Tcl/Tk [Ous94]. By now, there is quite a number of "embeddable interpreters" ranging from simplistic shell-like models up-to full blown functional and object-oriented languages. The integration platform for most of these approaches is a C API, by which the application can create and configure an interpreter for the extension language.

Embedding Lua into an application involves three mechanisms:

1. calling compiled application functions from Lua,
2. executing Lua statements from the application, and
3. passing values from C to Lua and vice versa.

(1) requires the compiled host program to register all functions that should be callable from Lua. This involves (a) writing a wrapper function (in C), that translates between different styles of argument passing and different data types, (b) making this function known to Lua as a value of type `function`, and (c) assigning it to a (global) Lua variable.

The reverse mechanism (2) is implemented by two simple C functions `lua_dostring` and `lua_dofile`.

Value passing (3) is supported (a) by C functions for accessing all Lua types and (b) by the capability of exposing arbitrary C references (technically `void*`) to Lua as values of type `userdata`. These values are opaque for the Lua side, except that they can be classified by an additional *tag*, that can be inquired by the Lua side. Usually the C side will allocate one tag for each reference type that is made visible to Lua. Lua programs can now dynamically query the tag as a type qualifier of `userdata` and use the appropriate functions to further process that value.

In this kind of architecture it is usually the compiled application (the "host" program) that plays the role of a "main". It at some point creates an interpreter instance and registers all functions to be exported. After that the flow of control depends on the kind of integration. Three common patterns exist:

- Control remains within the host program, which only for specific tasks asks the interpreter to evaluate a block of statements or the contents of a file.
- A shell like main loop reads lines at a console and executes line by line using the interpreter.
- An event based main loop has Lua functions registered as callbacks, such that external events may trigger the evaluation of Lua functions.

Such considerations about main program and subordinate modules are independent of the actual size of the involved components. So, in PIROL the approximate ranking of components from largest to smallest is shown in Fig. 1.5.

The size given for the meta model (5) is a minimum, which grows as additional tools and services are integrated.

From this little statistic, no simple architectural picture can be drawn, that clearly identifies "the application", "used libraries" and a small "scripting layer" on top of everything. Lua is used as an integration platform for very different modules. The largest module happens to be the one, that is best hidden from upper layers: the persistence module (1). Also, integration (2,6) is a reasonable part of the overall system. Which module is in control and which module is purely transparent depends in fact on the perspective by which one looks at the system.

Component		Code size		
		source	language	binary
(1)	A client library for persistence and related services (cf. Chap. 2)	N/A		1.6 MB
(2)	Lua level integration code and interpreter extension	10 KLOC	Lua	(237 kB byte-code)
(3)	A client library for inter process message passing (cf. Chap. 7)	10 KLOC	C	90 kB
(4)	Lua interpreter	7.9 KLOC	C	66 kB
(5)	The meta model (application logic)	5.6 KLOC	Lua/P	(143 kB byte-code)
(6)	C level integration code	3.5 KLOC	C	98 kB[2]
(7)	Lua standard libraries	2.6 KLOC	C	43 kB

Figure 1.5: Statistic of code size in PIROL

1.2.5 Meta programming using tagmethods

Lua is said to be an "*extensible* extension language" [IdFC96]. Of course, already registering C functions can be considered as an extension of the language. But there are other techniques that provide far more openness.

Firstly, all functions, including builtin primitives, are values, that can be passed around. Thus, it is easy to replace any builtin function by a customized version, while still using the original version, which for this purpose may be stored in some variable. This certainly allows to change the concrete semantics of certain operations, but never leaves the general style of programming.

Lua is, however, not a fixed language, but could rather be described as a *framework* for interpreted languages. The hot-spots of this framework are certain events during the execution of a Lua program, which are identified simply by the syntax of a statement. The most important events from this set are given by Fig. 1.6.

Now, a 'normal' language would strictly require the `o` in `gettable`/`settable` events to be a `table`, and the `f` in a `function` event to be of type `function` etc. Not so in Lua. To be precise, not the type decides about the interpreter's behavior, but the *tag* of a value. While types `number`, `string` and `function` each have a fixed tag (i.e., tag ≅ type), the tag of a `table` can be changed to a user-defined tag using `settag(obj,tag)`. Also, all userdata have tags distinct from all other tags.

[2]It appears as an anomaly, that the C level integration code is small in terms of lines of code but generates large binary code. This may be explained by the nature of this code, which is very repetitive and makes considerable use of macro programming. Wrapping C functions for Lua can to some extend even be automated by generators. This approach was not chosen, because many of these wrappers also perform slight adaptations, that could not be generated automatically. Still the complexity of this code is low, giving no rise to extensive comments. In terms of complexity the source code size is probably the more expressive metric, rather than the compiled binary size.

statement	event	description	signature
`o[i]`	gettable	retrieving a field from a table	(o, i)
`o[i] = v`	settable	assigning a value to a table field	(o, i, v)
`f(a1 ..)`	function	calling a function with arguments	(f, a1 ..)
`n1 < n2`	lt	comparing two numbers	(n1, n2)
`s1..s2`	concat	concatenating two strings	(s1, s2)
N/A	gc	collecting an object by the garbage collector	(o)
`i`	getglobal	retrieving a global variable	(i)
...			

Figure 1.6: Event types and signatures for tagmethods

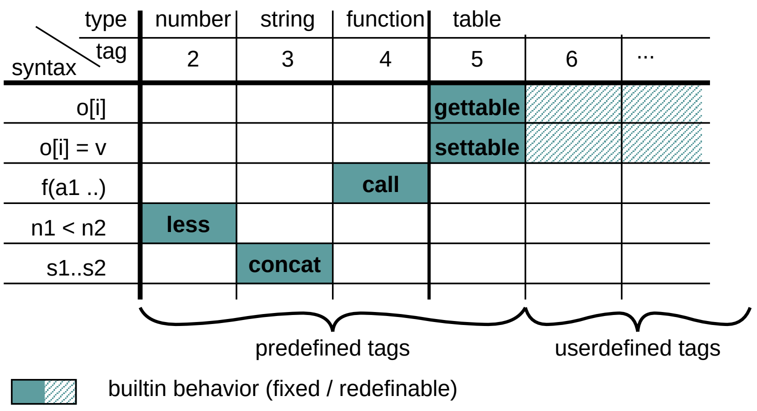


Figure 1.7: Matrix of tagmethods.

Given events and tags, the Lua interpreter at its core has a matrix (Fig. 1.7), with one column for each event and one row for each defined tag. Whenever a statement of, e.g., the gettable syntax `o[i]` is executed, function lookup is performed in this matrix using "`gettable`" and the tag of the primary value (here left side of bracket: `o`) as indices. If lookup yields a function, this function is executed with arguments as shown in the signature column of the above table. Such functions are called *tagmethods* because they specify for a tagged set of values how the interpreter should behave in certain events. The matrix of tagmethods is thus a mapping from syntactical patterns to language implementation. Of course standard tagmethods are built in, which define the "normal" behavior of the interpreter. These cannot be overridden. When creating a new tag, this is based on one of the types `table` or `userdata`. Tags derived from `table` are initialized with the builtin functions for tables, which can later-on be redefined.

Figure 1.8 gives examples of making numbers usable with the syntax of a table or function, and shows how to catch undefined global variables. The idioms `tag(1)` and `tag(nil)` are used to retrieve the predefined tags of types `number` and `nil`. Of course, these examples are not meant to be usable code, but brightly illustrate at which level the interpreter's behavior can be reprogrammed.

```
function times (o, i)
   return o * i
end

settagmethod(tag(1), "gettable", times)
seven = 7
print(seven[6])                    -- calls times(7,6)
   ⇝ 42
```

(a) `gettable`: number value used as table

```
function tripletimes (o, i, v)
   print(o * i * v)
end

settagmethod(tag(1), "settable", tripletimes)
seven = 7
seven[3]=2                   -- calls tripletimes(7,3,2)
   ⇝ 42
```

(b) `settable`: number value used as table

```
settagmethod(tag(1), "function", times)
print(seven(6))                   -- calls times(7, 6)
   ⇝ 42

settagmethod(tag(1), "function", tripletimes)
seven(3,2)                   -- calls tripletimes(7,3,2)
   ⇝ 42
```

(c) `function`: number value used as function

```
function timesseven (i)
   return i.." times seven"
end

settagmethod(tag(nil), "getglobal", timesseven)
print(six)                   -- calls timesseven("six")
   ⇝ six times seven
```

(d) `getglobal`:using an undefined global variable name

Figure 1.8: Examples of tagmethods

The `gc` tagmethod can only be defined for `userdata`. In fact, defining tagmethods for `userdata` is one of the most powerful features of Lua: this allows to export not only functions but also structured objects. Passing an opaque handle to Lua and defining `gettable` and `settable` tagmethods lets external structures transparently appear like Lua tables. This idiom allows very easy implementation of proxies, which has also been shown in [CRI97].

On the other hand, also standard Lua tables can be given new semantics by redefining their access functions. Firstly, `t=newtag("table")` allocates a new tag, copying the standard tagmethods from the standard "table" tag. Then, e.g., `gettable` and `settable` methods can be overwritten. Here the original versions are still available as builtin functions `rawgettable` and `rawsettable`.

The Lua documentation [RIC] illustrates a typical usage of the `gettable` tagmethod: implementing method dispatch as a dynamical lookup function for fields that don't exist in a given object (table) but may be found along a chain of `parent` references. A modified version of this technique is used for the first step of adapting Lua for PIROL: turning the unstructured language into an object-oriented one. The most important design decisions taken along this road are presented in Chap. 4.

At this point, only one fundamental distinction is discussed: unlike the examples from the documentation, where the linkage of tables via a `parent` reference knows only one kind of entities, Lua/P has a strict separation between objects and classes. While this is normal for most mainstream object-oriented languages, Lua seems to suggest a prototype based solution, where objects and classes cannot be distinguished. In this setting, inheritance is equal to *delegation* and method dispatch is implemented as a search through a chain of

```
function lookup (object, name)
   local value = rawgettable (object, name)
   if value then
      return value                          -- value is locally available
   end
   local parent = rawgettable (object, "parent")
   if parent then
      return parent[name]                   -- continue lookup at parent
   end
   return nil
end

object_tag = newtag("table")
settagmethod(object_tag, "gettable", lookup)

function new(fields)
   settag(fields, object_tag)
   return fields
end
```

(a) Implementing dispatch and creation

```
joe = new{              -- function call new({..}) abbreviated to new{..}
   firstname = "Joe",
   lastname = "Smith"
}

function joe:hello ()                       -- bind method to object
   print("Hi, I'm "..self.firstname.." "..self.lastname)
end

jim = new{
   parent = joe,
   firstname = "Jim",
   lastname = "Miller"
}

jean = new{
   parent = joe,
   firstname = "Jean"                       -- lastname inherited from joe
}

joe:hello()
   ⇝ Hi, I'm Joe Smith
jim:hello()
   ⇝ Hi, I'm Jim Miller
jean:hello()
   ⇝ Hi, I'm Jean Smith
```

(b) Using the new language

Figure 1.9: Prototype based object-oriented programming in Lua

parent objects. One of the best known examples of *prototype based languages* is Self [US87]. The flexibility of prototype based languages will be discussed at several occasions throughout this thesis. In Chap. 10 the central concept of Dynamic View Connectors will be presented, which in fact applies an object based delegation mechanism. However, at the core of Lua/P, objects and classes are strictly different things.

This section concludes with a short — condensed — example of how Lua can be turned into a prototype based object-oriented language (Fig. 1.9(a)) and how this can be used within Lua's flexible syntax (Fig. 1.9(b)). One abbreviation is used, that has not been explained yet: when passing a manifest table as only argument to a function, the adjacent braces and parentheses can be collapsed to just a pair of braces (i.e., in a call like `func({i1=v1,i2=v2})` the parentheses can be omitted; cf. lines (20,23) etc.).

1.3 Summary

This chapter has presented the conceptual backbone of PIROL: the decision to use an object-oriented meta model for all artifacts that are created within a software development project *and* the language Lua from which an OOPL is derived in order to implement the meta model. Lua has been presented as a framework for programming languages, that combines few fundamental concepts: one structured data type (`table`), an imperative core supplemented by higher order functions, a C level API for integration of client libraries and a radical mechanism for meta programming (`tagmethods`). In spite of the flexibility and diversity of these mechanisms, Lua remains a simple and light-weight language from which a variety of languages can be derived that don't suffer from unwanted complexity.

Subsequent chapter summaries will put a focus on concern interactions, classifying in which way the newly presented concepts interact with those presented before. This chapter only initializes this chain and has no interactions to report. But already hints have been given, how the decision for object-oriented meta modeling will have to be reconsidered in the light of the following chapters. When contrasting object based and file based techniques it should have become clear, that this is a very fundamental difference with impact on more or less every further consideration. At this point, no evaluation in this matter can be given, yet. But throughout this thesis we will look for traces, how this question has been dealt with in previous research and will compare this to findings and experiences from the PIROL project.

See, e.g., [2.4→]

Chapter 2

Persistence and Object-Oriented Programming

The first goal of PIROL's meta model is to allow persistent storage of information that should be used across different phases and their tools. At the time PIROL's concepts were formed, object-oriented database management systems (OODBMS) were still quite young. Because none of the system that were available to our analysis seemed to meet all requirements of PIROL, not a general–purpose OODBMS was chosen, but rather a system, that is specialized for building integrated software engineering environments (SEE) : PCTE (Portable Common Tool Environment) [ECM90]. This is not a true OODMBS but contains an object management system (OMS) that is structurally object based without supporting a notion of methods.

Related Work ▽

The Stoneman project [DoD80] was the first to use a database for tool integration. Since then, *data integration* has been recognized as a key issue in constructing tool environments. The three core dimensions of tool integration — data, control, presentation — have been identified by Wasserman [Was89], which is a fundamental reference in the field of tool integration. Wasserman mentions three possible techniques for data integration: files, relational databases and PCTE, which was only emerging as a standard at that time. It was Wasserman's central believe that tool integration would have to be based on such standards. Tools would have to be developed for standard interfaces. As he already observes the problem of agreeing on a single standard, he then proposes to decouple each tool from underlying technology by a clear cut interface where the object management system could be plugged into a tool. Such pluggability should be achieved by a layered architecture of each tool. Wasserman's tool environment "Software through Pictures" was a promising effort to open tool environments. It was, however, developed with too optimistic expectations concerning standards and pluggability, and maybe: too early. Given today's component technology, Wasserman's vision seems more realistic, than back in 1989. But perhaps, his misperception was not a matter of technical standards but of logical integration, which gives semantics to the integration of independently developed tools. This issue will be discussed in Chap. 10.

Still in 1993, in [ESW93] the authors discuss the requirements for a dedicated database system for process-centered software development environments.

Although PIROL may not be process-*centered*, we still view process support as an important goal. Thus the considerations of [ESW93] also hold for PIROL. The authors of [ESW93] identify a comprehensive list of requirements and issues (listed in a re-arranged fashion):

	Requirement	**References**
(1)	Syntax-direction	Sect. 13.3.2
(2)	Documents versus nodes in a graph-structured database	Chap. 1
(3)	Tree versus graph structure	Sect 2.2.1
(4)	Inter-document relationships	Chap. 1
(5)	Persistence and integrity	Chap. 2, Chap. 6
(6)	Efficiency	throughout and Sect. 14.1
(7)	Update incrementality	Chap. 3, Sect. 8.3.3
(8)	Data and behavior definition (in [ESW93]: *Data Definition Language/ Data Manipulation Language*)	Chap. 1, Chap. 4
(9)	Evolution (in [ESW93]: *change* and *schema updates*)	Chap. 12
(10)	Queries (in [ESW93]: *Reasoning support*)	— [1]
(11)	Consistency preservation	Chap. 6
(12)	Distribution	Chap. 7, Chap. 8
(13)	Multi-user support	Chap. 9
(14)	Access control	Chap. 9
(15)	Security	Sect. 7.3.2
(16)	Adjustable transaction mechanism	Sect. 9.1.7
(17)	Backtracking and versioning	— [2]
(18)	Views	Chap. 10, Part III

The conclusion of [ESW93] is — as indicated in the subtitle "The Goal has not yet been attained" — that none of the existing database management systems of that time meets all listed requirements. While they give quite a good ranking to PCTE, they criticize its performance problems with regard to a large number of small objects and the lack of encapsulation with operations.

Discussion of model granularity [3⟶]

Integration of methods in our meta model [4⟶]

2.1 Basic Data Model of PCTE

Background ▽

The data model of PCTE is in fact quite simple. This data model is not responsible for the complexity of PCTE. This complexity is rather due to the number of additional issues that are also dealt with, like referential integrity, access control, transactions, reflexive schema management, notifications etc. Also, these issues interact in more or less obvious ways.

[1]Queries have not (yet) been identified as a requirement for PIROL.

[2]Versioning is not fully solved in PIROL. See Sect. 9.1.7 for a discussion.

The basis of all this is a data model with basically three abstractions: *objects*, *links* and *attributes*. The corresponding types are defined in terms of the following properties:

attributetype	
type	one of `string`, `natural`, `integer`, `boolean` or `enumeration`

objecttype	
parent type	objecttype from which to inherit all properties.
attributes	attributetypes that are *applied* to this objecttype (see below)
links	linktypes that are *applied* to this objecttype

linktype	
cardinality	defined by either a key attribute or a range thereof
destination	objecttype of permissible link targets
category	strength: ranging from `implicit` to `composition` (see below)
attributes	attributetypes that are *applied* to this linktype

Attribute– and linktypes can be defined independent of any objecttype. Only *applying* an attribute– or linktype to an objecttype renders these definitions useful: applying an attribute– or linktype `ALT` to an objecttype `OT` defines that all objects of type `OT` have the property `ALT`. Furthermore, attributetypes can be applied equally to object– and linktypes. By separating definition and application of attribute– and linktypes the same definition can be reused for many applications, but this comes for the price of a flat name space: different objecttypes may not have attributes with identical names but different types.

The *inheritance* mechanism of PCTE is compliant with object-oriented programming languages in that it defines a hierarchy of type compatibilities. Links are typed to a target objecttype, but all linktypes are implicitly polymorphic. They also allow *subtypes* of the specified type.

The most intricate part of PCTE's data model is the concept of *links*. The *category* of a linktype defines a link's strength with regard to constraints of referential integrity. The strongest link category, `composition`, is used to define *compound objects* that can be manipulated in one step by certain operations. Additional *properties* define whether a link can be used for navigation, and whether it should be duplicated when copying an object. *Attributes* can be attached to links, and if one or more attributes are defined as a key, the link automatically has a one-to-many cardinality. In that case the link name is composed of the name of the linktype and the value of the key attribute(s). Each linktype also has a *reverse* type, such that links always exist as pairs. If no reverse linktype is specified an `implicit` linktype is generated.

Integrity issues [6→]

In the specification of PCTE each object or link may also have a *contents*, i.e. an uninterpreted block of binary data, intended for large chunks of data that are handled more like sequential files rather than like ASCII–strings. The PCTE implementation used in PIROL, H–PCTE [Kel92], is more flexible in that it allows to access any attribute declared as `string` in a contents fashion. It is left to internal optimization, whether string attributes are stored as intrinsic part of the object or as separate external files in the database directory.

An API exists for creating all type definitions interactively. H–PCTE also provides a tool, `ddlc`, that translates textual type definitions in a data definition language (DDL) to their internal representation.

Finally, type definitions are grouped to *schema definition sets*, which are the units of further scoping and visibility issues.

2.2 Persistence Interacts with other Concerns

2.2.1 Meta modeling

New Feature ▽

PCTE is focused on database technology. PIROL's meta model is a more or less standard object-oriented model. More precisely, PIROL's *repository language* Lua/P integrates the worlds of object-oriented programming and of repositories. This requires to map the concepts of PCTE to a style that is more fashionable for the OOPL Lua/P. This section gives details of how PCTE types and Lua/P types are mapped to each other. On page 41 the integration of PCTE mechanisms into the syntax of Lua/P will be shown.

Mapping PCTE types to Lua/P

The module concept of PCTE, schema definition sets (SDS), is directly mapped to *packages in* Lua/P. The type mapping is developed according to these requirements:

1. Map basic types (attribute types)
2. Map reference types (link types)
3. Map 1:n relationships
4. Optimize in order to reduce number of objects
5. Exploit link properties
6. Exploit reverse links
7. Allow for multiple inheritance

Basic and *reference types* (items 1 and 2) are straight forward. Only numeric types had to be unified. Lua/P only supports one type Integer dropping the cardinal type from PCTE and floating point numbers from Lua.

As *1:n relationships* (3) are a vital capability of PCTE they are directly supported by Lua/P in terms of a builtin type constructor **List**. Attributes declared as List can be considered as auxiliary Lua objects of a builtin class List with value semantics. I.e., these auxiliary objects need not be created explicitly nor is assignment to such a field valid. Only the methods of the builtin class List and Lua's builtin indexing mechanism `o[i]` allow to query and modify the list. Appendix A.2 gives the interface of List.

It is the responsibility of the internal classes implementing the List interface to synchronize a set of links in PCTE with a list in Lua/P. This is mainly an issue of mapping two sets of indices: In PCTE each link in a list has a unique integer key, which is immutable. Thus deleting an element may introduce "holes" in the list, while inserting may require to re–link all elements after the point of insertion. On the Lua/P side, however, list indices are always consecutive. In other words, in PCTE the list is optimized as to do reorganization only if really needed, while in Lua/P consecutive indices are ensured. This strategy is encapsulated by all List classes.

Caching. The design of PIROL generally provides for direct access to repository data without buffering in Lua. Lists are an exception to this rule. The index structure of a list is kept in a cache after reading. This is because building up this index structure is a time consuming operation and it is common to access the same list successively many times. The operations of class `List` can not be implemented efficiently without caching. Note, that only the index structured is cached not the data contained in the list. It should not surprise, that storing data in a database is well complemented by keeping a minimal set of data as a transient copy in a cache.

Specific optimizations. For the type mapping at hand it is important to minimize the number of objects in the repository in order to achieve a good performance. Many (auxiliary) objects in a typical object-oriented design can be replaced by light–weight tuples. Lists of tuples are a convenient way to save one extra object for each list element. This is made possible by optimized mappings of such lists to available constructs in PCTE. Three cases need to be distinguished regarding the type of tuple elements:

- **Only basic types**
  This case will be covered by packing techniques to be presented in Chap. 3.

Reducing the number of objects [3→]

- **Basic types and exactly one reference type**
  This is implemented in PCTE by links with link attributes. The reference component is mapped to the link, all other tuple components are link attributes.
- **More than one reference type**
  Only these tuples actually need auxiliary PCTE objects.

Lists of basic types (String, Integer, Boolean) have no direct representation in PCTE. Thus they also rely on the mentioned packing techniques.

Packing lists of basic types [3.2.2→]

In addition to carrying attributes, links have far more properties in PCTE (cf. requirement 5 in the above list) than references in traditional OOPLs have. It is not perfectly clear, which properties should really be visible in the repository language Lua/P but two categories should certainly be distinguished: `existence` links can be seen as regular references that also guarantee *referential integrity*. `Composition` links have an even stronger semantics: objects connected by composition links are considered one compound object. Several operations — like copying, moving — can be performed on such compounds as a whole. Lua/P supports this distinction by two separate clauses in the definition of a class's structure: **attributes** declares all basic type attributes and existence links, **components** declares composition links.

Each link in PCTE is automatically accompanied by a reverse link (item 6 in the above list of requirements), i.e., links always exist in pairs. This is very convenient for many queries to the repository (like: "what are the packages that refer to (import) this class"). This information is readily available even if it is not explicitly modeled in the meta model. On the other hand it is not safely possible to *manipulate* reverse links without side effects on the original link. Thus it suffices to provide reading access to reverse links. This requires no additional language construct but is encapsulated by methods of the meta model.

Methods of the meta model [4→]

As PCTE allows multiple inheritance (7) it was only straight forward to also support multiple inheritance in Lua/P. There has been a vehement debate about the dangers of multiple inheritance, and in the context of Java the word has be spread, that it should even be avoided completely. On the other side, conceptually sound languages like Eiffel [Mey92] show which mechanisms are needed to manage multiple inheritance. Quite contrary to both positions, our experience tells us, that — in the bounded field of programming a repository — even the most simple form of multiple inheritance delivers far more benefits than problems. The exact picture of multiple inheritance in Lua/P will be given in Chap. 4. An example of its usefulness in our context relates again to minimizing the number of objects and will be given just below.

Flexibility versus Performance

Many design patterns [GHJV95] that aim at flexibility propose to introduce auxiliary classes that don't originate from analysis but are needed only as a capsule for certain object properties that should be interchangeable in a flexible manner. PIROL's meta model requires exactly this flexibility. Fig. 2.1 shows the root class of the meta model, `ANY_RO` that models all those administrative data that should be associated to every object in the repository. By spreading these data over six classes, each of these parts can be specialized independently. Furthermore, factory methods allow to centrally enable such specializations for all objects that are created within a certain context.

If, e.g., a project decides to allow sophisticated version branching it will find the data that can be stored by class `VERSION` insufficient. A specialization of `VERSION` will be implemented and the name of that class is stored in `PROJECT.version_class`, such that throughout that project only instances of the specialized version class will be created.

This design achieves the desired flexibility but it is in fact untenable due to the exploding number of objects, which it causes. For every object created as a descendant of `ANY_RO` five auxiliary objects would have to be created. Only one of these objects, `STATE`, can easily be optimized by the Flyweight pattern, such that usually no more than a handful of `STATE` objects are needed per installation of PIROL.

For all other auxiliary objects an optimization is introduced using multiple inheritance (see Fig. 2.2). For the standard situation, where none of these classes is specialized, all data can be merged into one object, while still retaining the ability to apply dedicated and independent specializations. This is achieved

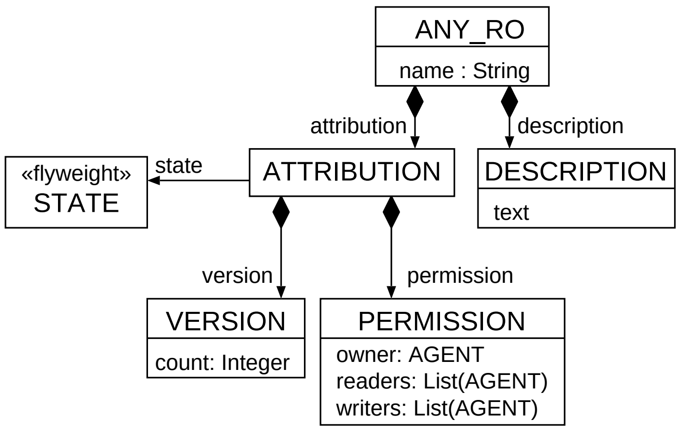


Figure 2.1: Structure of `ANY_RO` (before optimization)

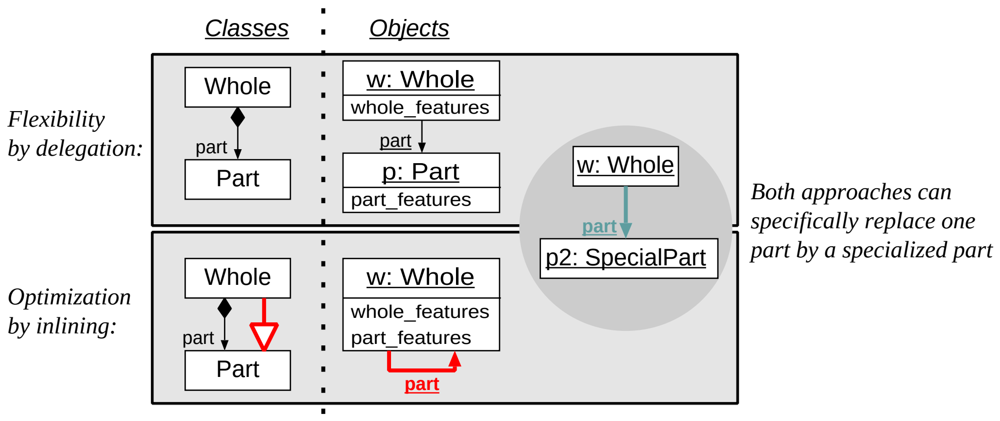


Figure 2.2: Optimization by object inlining

by having `ANY_RO` (indirectly) inherit classes `ATTRIBUTION`, `VERSION` etc. (see Fig. 2.3).[3] The references `attribution`, `version` etc. are still maintained and simply point to `self`.

In the standard case the number of objects is now reduced by a factor of five. If any facet of this compound object is to be replaced by a specialized variant it is still possible to create an auxiliary object and attach it to the host object. As a result flexibility and performance are reconciled and the overhead of auxiliary objects is only paid for when needed. We call this technique *object inlining* (Fig. 2.2).

Transient data

Finally, the presence of persistent data brings about an additional distinction that has effect throughout many other concerns: not all data are to be kept persistently, because this imposes performance penalties that are sometimes unacceptable. Chapter 11 will present common services, that are implemented

→ Sect. 11.3.2 brings the example that motivated transient data.

[3]Note, that `ANY_RO` also needs to inherit from an even more fundamental class `ANY`.

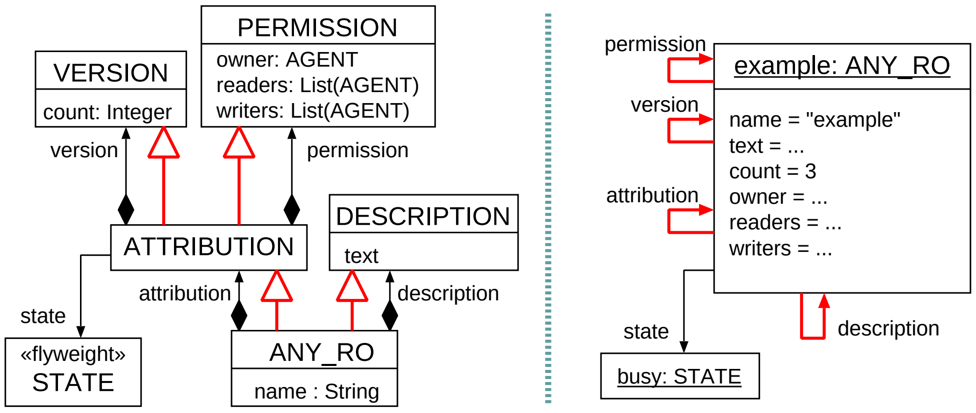


Figure 2.3: Optimized structure of `ANY_RO`

in Lua/P but require a performance that cannot be achieved using persistent objects.

Transientness can be declared in Lua/P at two levels: *attributes* can be declared as transient. This can be used to efficiently use Lua structures as cache for run time information. Transient attributes can be of any legal Lua type, no mapping to PCTE needs to exist. Thus transient attributes to some extent fall outside the general picture. Secondly, *classes* can be marked as transient. Objects of transient classes will never be stored in the repository, but are otherwise indistinguishable from repository objects. In order to hide the distinction between persistent and transient objects not only their classes must share the same general structure, but all attribute types that can be declared for persistent classes need to be rebuilt for transient classes.

Restrictions caused by this relaxed typing [7.4.2→]

This is trivial for basic type attributes. Reference attributes are implemented as Lua references to Lua handles of persistent objects. List, finally, had to be rebuilt completely due to the specialized mapping of persistent lists to sets of links in PCTE. This results in another List class with the same interface as for persistent lists, but a different implementation.

Object access

One aspect of persistent objects can usually be ignored when handling PCTE objects: loading objects from the repository is transparent for client code. The signatures of the PCTE API use handles (“object references”) to refer to objects without saying at which point in time an object will be loaded.[4] Client code may safely assume that any (part of an) object is loaded transparently whenever it is needed (on demand loading).

To give the exact picture of object access in H–PCTE, two more concepts will come into focus: objects are not loaded one by one, but as one database *segment* at a time. Thus H–PCTE is designed as a main memory database in

[4]In PCTE several operations do have a parameter that specifies whether objects should be loaded lazily or eagerly, but the consequences are too subtle to be of importance for this discussion.

order to reduce disk access.

Secondly, the architecture of H–PCTE includes a *server* such that a single process controls all access to a given repository. A segment may be loaded to one of two different locations: to the server process or to a client process. Segments loaded to the server require inter process communication for each access while segments loaded to a client pay for the enhanced performance by restricted visibility: no other client can access a segment that is loaded to a client.

Segments and object accessibility [9→]

Aside from issues of explicitly loading segments to memory, object access is encapsulated completely by only a small number of PCTE API function. However, a smooth integration of PCTE into Lua/P calls for even more transparency: object access should not require to explicitly use any access function, but statements like

o1.a1 = o2.a2

should automatically be mapped to the appropriate API function calls. The next section will give the implementation details of the given type mappings and of syntactical embedding of PCTE into Lua/P.

2.3 Implementation Issues

The previous section has defined the requirements for implementing Lua/P on top of PCTE's data model. Here, some implementation issues of this mapping will be discussed.

2.3.1 Implementing the mapping

The internal (run–time) representation of a class consists mainly of two Lua `tables` of access functions, one for reading, the other for writing. All access functions directly operate on the contents of the PCTE repository, thus avoiding any danger of inconsistency between PCTE and Lua/P. Both tables are indexed by attribute names. This relies on Lua's capability, to handle functions as regular values, that can be stored in variables. Secondly, use of *function closures* is made, such that *general* lookup functions are embedded into a closure containing some of the parameters as frozen values that are already known at the time of constructing the internal class structure.

Higher–orderedness in Lua [←1.2.3]

At run–time each object knows its class. Field access of objects is then redefined (using Lua's tagmethods). This technique allows to use PCTE *object references* as if they were the actual object. Any field access relative to an object reference is implemented by either a `gettable` or a `settable` tagmethod, that uses PCTE API functions to perform the repository access.

Extensibility by tagmethods [←1.2.5]

Fig. 2.4 sketches the techniques of this implementation: The client statement in line 30 triggers a `gettable` event on `o2`. Line 26 associates function `luap_object_lookup` with this event. Line 12 retrieves a lookup function from the corresponding class, for which the class initialization (sketched in lines 28/29) installed `pcte_object_get_string_attribute`. This function is invoked in line 14. Writing value `v` to `o1.a1` works in analogy. Note, that for a list attribute only a `read_func` is installed (lines 5–9). The lookup function for lists is a closure, that shows how information about the element type — which is available only at class definition time — is stored in this closure using an *upvalue* (`%elem_type`).

To sum up, an event of reading or writing an attribute is simply redirected to the appropriate access function, that is determined by the class of the current object and the attribute name. By this mechanism the most simple syntax of attribute access — both for lookup and in assignments — has the effect of directly operating on the persistent object in the repository.

The internal class structure also contains the table of methods (see Chap. 4) and knows about

- creation methods (see Chap. 4).
- list-classes of list attributes (selecting between simple lists and the different styles of tuple lists).
- class attributes (these are always transient and stored only here).

2.3.2 Meta model deployment

Having a repository concern and one of object-oriented programming, both manifesting at PIROL's meta model, raises the issue of transformation and installation of the meta model. What are the inputs and the tools for the deployment process? The mappings between Lua/P and PCTE, as introduced above, imply different representations of the same meta model. Only for a fixed set of core packages of the meta model, a DDL[5] specification is needed. This is part of the boot–strapping strategy. After that only Lua/P is used for definition of the meta model. The full boot procedure will be presented after behavior modeling has been introduced.

The boot process builds upon behavior modeling [4.2→]

2.4 Other languages for persistent meta models

Related Work ▽

The literature reports on many SEEs that decompose their data into persistent, structured objects. Judging from published articles, RPDE[3] seems quite closely related to PIROL. Unfortunately, no comprehensive description about the language used in RPDE[3] could be found. A conceptual comparison will be given in Part III.

From Files to Objects. It should be noted, that the transition from plain files to structured objects in a database was not performed in one single step. The concept of attributed software objects (ASO) [Lam94] combines both concepts by attaching additional attributes to files. Attributes are primarily used for version control, document states and build management, but user defined attributes may in arbitrary ways classify files and attach specific data. File based tools can completely ignore such extensions while specific replacements

[5]H-PCTE's Data Definition Language.

for commands like `ls`, `cat` and `make` exploit attributes for additional functionality.

Also PCTE can be used for a hybrid model, in which objects directly replace files, storing the traditional file contents without further decomposition/processing into object contents. The actual PCTE data model in this setting only captures administrative information just like in the ASO approach.

How much detail is covered by PIROL's data model will be subject of Chap. 3.

2.4.1 Arcadia

Within the Arcadia project [TBC+88], two extensions to Ada have been developed. The first, APPL/A [Sut90] is most interesting with respect to managing consistency of persistent structures and will be discussed later. The second language, PLEIADES, focuses on the following requirements: "high-level primitive type constructors, navigational and associative access over the same structure, persistence and consistency management"[TC93].

Languages for consistency [6.3,6.3.1→]

High-level *types* are an issue in PLEIADES, because it builds on a pre-object-oriented version of Ada. Re-usable structures for lists, graphs and relationships should not be a language issue in an object-oriented setting. PIROL features a general `RELATION` class with sub-classes like `GENERALIZATION` (between classes), `DEPENDENCY` (between arbitrary program units), which can be exploited by other RO classes and tools in very different ways. Such issues require no language extension, but can be handled by generic design of the meta model. More specifically, also Lua/P supports lists as a fundamental type constructor. While conceptually an RO class `LIST` would have provided the same functionality, built-in support for lists is crucial for *efficiency* and easily maps to one-to-many relationships in PCTE.

The tension between *navigational* and *associative* access relates closely to the transition from relational databases to object-oriented ones. Both have their preferred way of retrieval, and [TC93] concludes that both forms are needed for an SEE. In PIROL, the navigational access is emphasized, with no need to-date of supporting queries over large amounts of data. We are, however, confident, that once the need for queries would arise, one of the existing query languages for H-PCTE [Haa, Hen95] could readily be integrated into Lua/P.

Regarding persistence, [TC93] develops a reachability based approach, which is similar to PCTE's strategy of maintaining a connected graph of persistent objects. PLEIADES does not consider references other than containment for persistence. It is not clear, how the system copes with a link who's target is lost due to a missing containment link. In contrast to PCTE, PLEIADES supports implicit transientness by simply creating unconnected objects. Lua/P also supports transientness, but based on Transient declarations of attributes or classes. PLEIADES also introduces lazy retrieval of the transitive closure of reachable objects, i.e., on-demand loading. Similarly, a PCTE client need not worry about loading an object, either. In H-PCTE loading happens in chunks called segments. Such partitioning is not reported concerning PLEIADES.

Segments [←2.2.1]

APPL/A [6.3.1→]

The last feature of Pleiades, consistency management, will be deferred to our discussion of APPL/A.

2.4.2 GOODSTEP

The O_2 database [16.1.3→]

The GOODSTEP project [AAA+94] has had great impact on the development of OODBMS suitable for SEEs [EKS93]. Two systems are central to GOODSTEP with respect to the topics at hand: the OODBMS O_2 will be discussed more in detail in Part III, the tool specification language GTSL [Emm96] will be discussed here.

The name "GOODSTEP Tool Specification Language" requires to discuss what is a tool in an SEE. For this thesis the question has to be postponed until Chap. 7, because architectural considerations are decisive, here. Not so for GOODSTEP. Most issues in the mentioned articles about GOODSTEP and GTSL can be compared directly to our notion of a persistent meta model.

GTSL is a multi-paradigm language, it combines different concepts which appear most appropriate for different aspects of the system. First of all, it is an object-oriented language, that encapsulates persistent objects, much in the same sense as Lua/P (see Chap. 4 for methods in Lua/P). The central idea is a generalization over disparate abstract syntax trees for each document to one compound abstract syntax *graph*. Such a graph is spanned by a tree of aggregation edges and augmented with additional reference edges, which implement semantic relationships. Such relationships are usually created by analysis like name resolution, which links name applications to corresponding declarations. Also inter-document relationships play an important role, since these relationships are used for maintaining project-wide consistency. Similar to PCTE, GTSL supports implicit reverse links of all reference edges[6]. Multi-valued aggregation edges are supported by a LIST type constructor. All this very closely resembles Lua/P. A class in GTSL must be declared as either terminal (leaf) or non-terminal. The benefit of this distinction remains unclear.

Dynamic View Connectors [10→]

Views in O_2 [16.1.3→]

Despite of the use of an object-oriented meta model, GOODSTEP basically operates on documents, where the AST representation is just used internally. There is no concept comparable to COs or Dynamic View Connectors, providing for overlapping documents, which share common objects. The view concept of O_2 is said to be supported, too, but no details could be found in published papers. A special feature of GTSL relates to this dual view of documents: each non-terminal class contains an unparsing scheme, which defines how the abstract syntax is to be translated into concrete syntax. PIROL assumes that only some classes have a textual concrete syntax representation (other objects might only appear in a diagram or be displayed in a table or by a form based tool, etc.), and multiple representations are also anticipated. For this reason, unparsing is not built-in to PIROL but left to specific RO classes. A more general approach to external representations is discussed in Part III.

SOURCECODE [13.3.2→]

Garlan's display views [16.1.1→]

Instead of sharing objects between views, GTSL admits any redundancy incurred by documents with overlapping contents and adds consistency rules

[6]From [Emm96] it appears, that reverse links are not featured for aggregation links, which seems a bit odd.

concerning inter-document relations. Such rules are the second paradigm, which is supported by GTSL. In [Emm96] the author makes a strong statement about the style in which consistency should be specified, concluding that condition–action pairs are strictly preferable over any imperative concept. Consistency rules will be discussed in Chap. 6.

Language support for consistency [6.3→]

GTSL is said to also feature imperative concepts, the meaning of which is not perfectly clear, after object-oriented concepts (which build on imperative ones) are already included. The author assumes, that this label is motivated by the support for interactions. Interactions can be compared to common services of PIROL. Therefore, a discussion has to be postponed until later.

Services provided via context menu [11.2→]

Throughout the literature about object-oriented meta models, performance and complexity are important issues. In the context of GTSL, especially configuration management is said to be an issue, which "has not yet been sufficiently addressed" [Emm96]. Related articles report on the large number of objects [ESW93] and fine-grained concurrency control [EAMP97, EKS93] as being hindrance for using most available database systems. Hope is pinned on general OODBMSs like GemStone and O_2 of which the latter unfortunately disappeared from the market.

Views in O_2 [16.1.3→]

The main reasons against PCTE are said to be (a) the inability to manage large collections of small objects and (b) the absence of methods. Of these, (a) will be discussed in the following chapter (Chap. 3), and (b) is subject of Chap. 4.

2.5 Summary

The concerns meta modeling and persistence are *layered*, i.e., persistence is *encapsulated* in such a way, that no direct access to this layer should be needed. In this special case we even have a *transparent* encapsulation, such that regular usage of the upper layer (meta modeling) implies invocation of the lower layer (persistence) whenever relevant.

More concretely spoken, by reading and writing objects of PIROL's meta model, these objects are implicitly, i.e., without further action, stored and retrieved from the repository. Each Lua/P object (except for those, that are explicitly transient) is always persistent, without ever calling any `store` or `retrieve` functions.

The interaction of meta modeling and persistence is thus once and for all implemented by the *mapping* between PCTE and Lua/P. This mapping has been developed to combine convenience at the level of client programs with a good performance at the repository level. Note, however, that this encapsulation does not completely hide PCTE from all layers above, because PCTE covers far more concerns than just persistence. These other concerns will be encapsulated by different techniques, to be presented when those concerns come into focus.

Also note the difference between generic concerns and their specific applications: the *language* Lua/P is for certain parts determined by the data model of PCTE. Vice versa, the choice of a repository is restricted by the required properties of Lua/P. These interdependencies manifest in the concrete mapping between PCTE and Lua/P. This mapping, however, achieves an independence at the *application* level: packages written in Lua/P need not care about persistence issues.

Transparency has been achieved mainly for the sake of comprehensibility. This should not be confused with an assumed substitutability of the repository. Although PCTE is hidden from the Lua/P programmer, it has had significant impact on the design of Lua/P. Nobody should expect a re-implementation of Lua/P using a different repository or database to be a reasonable task. Different persistence technology may draw parallels from the system at hand, but would certainly yield a different flavor of a repository language. Evidence for this will grow, as further concerns are added to the discussion.

Finally, introducing persistence brings along the *choice* between persistent and transient data. Other concerns should, as far as possible, be unaffected by this choice, i.e., this difference should be transparent. A few other concerns, however, have to invest extra effort into this transparency.

```
function luap_define_string_attr (class, field)
    class.read_funcs[field] = pcte_object_get_string_attribute
    class.write_funcs[field] = pcte_object_set_string_attribute
end
function luap_define_obj_list_attr (class, field, elem_type)
    class.read_funcs[field] = function (obj, f)
        return Object_List:New(obj, f, %elem_type)
    end
end
function luap_object_lookup (obj, field)
    local class = luap_get_class(obj)
    local lookup = class.read_funcs[field]
    if lookup then
        return lookup(obj, field)
    else . . .                        -- accessing an undeclard field, raise an error
    end
end
function luap_object_write (obj, field, value)
    local class = luap_get_class(obj)
    local write = class.write_funcs[field]
    if write then
        write(obj, field, value)
    else . . .                        -- accessing an undeclard field, raise an error
    end
end
settagmethod(luap_object_tag, "gettable", luap_object_lookup)
settagmethod(luap_object_tag, "settable", luap_object_write)
-- Example definitions:
luap_define_string_attr(C1, "a1")
luap_define_string_attr(C2, "a2")
-- Assuming o1: C1, o2: C2, this client code:
o1.a1 = o2.a2
-- automatically calls:
v = pcte_object_get_string_attribute(o2, "a2")
pcte_object_set_string_attribute(o1, "a1", v)
```

Figure 2.4: Implementing object lookup

Chapter 3

Model Granularity

Fine grained data modeling is a powerful means for a tight integration of tools that are to share as much information as possible. It contrasts to data integration using the source text by eliminating the need for many tools to include their own parser. Also referring to an element within a text is much more difficult than referring to, say, an object in the repository. Text references always need to be interpreted and are fragile in one way or other: positions and names, that may be used for qualifying a reference, may change without notifying the referring instance.

Of course an object-oriented meta model could very well be used to decompose a document all the way down to single identifiers and symbols. This technique is, however, hardly usable for SEEs. A prominent approach to fine grained data modeling for SEEs has been standardized as extension of PCTE[PCT95]. Unfortunately no tool vendor ever really implemented this standard due to tremendous performance problems that should be expected. Database technology in fact imposes quite strict limits on the number of objects that can be accessed efficiently when, e.g., loading a document. Each persistent object has at least a constant overhead in space and access time. In a uniform approach, each object in the database requires its own access control and versioning. For very fine grained data this is neither acceptable nor needed.

Minimizing the number of objects [←2.2.1]

Quite a different lesson can be learned from the area of compiler construction and related tools. Such tools rely on a set of types that represent all constructs of a (programming) language in a tree or DAG structure, called *abstract syntax*. The definition of these types and many transformations are much more compact and perhaps more elegant when using a functional programming language rather than an object-oriented one. For this reason a previous instantiation of PIROL [BGHHm98], which was targeted at processing formal specifications based on their abstract syntax, used the programming language Pizza [OW97]. We made good experiences with Pizza's combination of object-oriented and functional techniques. In this setting the bottleneck was the serialization of Pizza objects. Serialization, which was used to write units of the abstract syntax as binary blocks into the repository, again imposed performance problems on the system.

In response to this experience, Lua/P was extended by some new features: The types needed for an abstract syntax or similar structures can be defined as *term grammars*. Terms as values of these types can be handled and stored efficiently by the workbench. Allowing term types for attribute declarations yields a smooth integration of medium grained objects and very fine grained terms. Finally a touch of functional programming in Lua/P allows concise implementations of algorithms over terms.

Algorithms over terms [4.1.3→]

3.1 Structured decomposition and composition

Data modeling is about decomposing concepts into atomic pieces of information. Equally important is the possibility to express structural relationships between pieces of data. These relationships are expressed at the type level by means of *type constructors*. We have already seen two type constructors in Lua/P: `Class` and `List`.

- Classes are the primary structure for any kind of data. They support the definition of three relationships: association, composition (the strongest form of association) and inheritance.
- Lists are the preferred container for elements of the same type. Elements may be simple values, objects or tuples.

Note, that already lists are introduced in Lua/P with the intention of reducing the number of persistent objects. Next follows the definition of another set of type constructors, that allow very fine grained data modeling even in the presence of a repository: term grammars.

3.1.1 Term Grammars

Terms are tree structures whose leaves are simple values or terminal symbols. Simple values are strings, integers, boolean or subtypes thereof. Lua/P provides four kinds of type rules (the LHS of each rule is a type):

subtype_of	The LHS type can be used wherever the RHS type is required. It has the same structure.
one_of	The LHS type has alternatives that are listed here. The alternatives still have to be defined.
one_of_const	Similar to the above but the alternatives are terminal symbols given by their representation.
production	The LHS type is a tuple of the types listed at the RHS. Production rules have no keyword.

Fig. 3.1 gives an example grammar defining a simple expression language. The names `e1, operator` and `e2` as defined in rule (2) are selectors for the components of an `expr` term. The second component in rule (3) is not named, so the type name `expr` is also used as selector. The `'?'` in rule (4) specifies that the last component (`else_exp`) is optional. The `expr` in rule (9) may occur zero to many times (denoted by `'*'`). Elements of such a list can only be accessed

```
  Grammar {EXPRESSION;
(1)  expr     = one_of{value, binexp, unexp, ifexp},
(2)  binexp   = {{e1: expr}, {operator: binop}, {e2: expr}},
(3)  unexp    = {{operator: unop}, expr},
(4)  ifexp    = {{condition: expr}, {then_exp: expr}, {else_exp: expr}, '?'},
(5)  binop    = subtype_of{STRING},
(6)  unop     = one_of_const{{uplus='+'}, {umin='-'}},
(7)  value    = one_of{INT, BOOL, varappl},
(8)  varappl  = subtype_of{STRING},
(9)  exprlist = {expr, '*'},
  }
```

Figure 3.1: Grammar `EXPRESSION`.

by their numerical index. Finally the whole grammar is given a name in order to make it a selectable name space.
Each type defined in a grammar can be used for attribute declarations as in

```
Class {SIMPLE_FUNCTION;
   inherit = ROUTINE,
   Attributes = {
      value : EXPRESSION.expr
   }
}
```

3.2 Model Granularity Interacts with other Concerns

3.2.1 Meta modeling

The techniques introduced so far allow for a kind of hybrid meta modeling, where the object-oriented concepts are used for higher level abstractions, that are to be shared by very different tools operating on very different views of the repository. At lower levels heavy weight objects are replaced by hierarchic, structured terms. These two techniques harmonize only if some integration issues are solved. Mainly three issues can be identified: syntax, semantics and references.

Discussion ▽

The *syntax* of accessing parts is the same for objects and terms: `compound.field` or `compound[index]`. The first form applies for attributes and named term components, while the second form applies to elements in a list or elements in a term with repetition. Such transparency is achieved by associating different tagmethods with ROs, lists and terms, which roughly implement the same behavior based on different low level APIs.

Lua tagmethods [←1.2.5]

In spite of this smooth syntactical integration, Lua/P programmers need to observe one difference: object-oriented structures are handled by reference semantics, whereas for some part term structures introduce *value semantics*. Details are given below in Sect. 3.2.2.

The last issue concerns *references* between the worlds of objects and terms. For now, we restrict this issue to stating that

- an object may *contain* a term,
- an object may point *into* a term using a *path expression*, and
- a term may refer to an object.

References between terms and objects [6.2.3→]

Path expressions are not yet implemented and references from terms to objects are not yet fully automated. However, these techniques need to be discussed in some more detail after raising the issue of data integrity.

3.2.2 Persistence

Discussion ▽

When motivating the additional mechanism for very fine grained data modeling, the focus on performance has already been mentioned. The straight–forward object-oriented approach to very fine grained data modeling conflicts with the performance requirement, when it comes to making these structures persistent.

By introducing terms as a separate technique, we can ensure that very fine grained data always exhibit a tree structure, which can be packed efficiently. Packed structures are then stored as one binary block into an attribute of PCTE type contents. So for the repository there is no overhead for very fine grained data as compared to storing the unparsed text. However, within the workbench, that block is unpacked again and thus exists as structured, type–safe data.

PCTE type "contents" [←2.1]

In the example of class `SIMPLE_FUNCTION` above, the repository would see `value` as one binary block, while the workbench sees a structured term of type `expr` according to the `EXPRESSION` grammar.

Implementation ▽

This concept is implemented by a C library, that is responsible for

- building type rules from a grammar definition,
- building terms while ensuring type correctness according to these rules,
- accessing term components,
- efficiently packing/unpacking terms to a binary stream (serialization).

△

Access to objects and terms is to some extent uniform, but the difference of reference versus value semantics remains, as stated above. This restriction is imposed by the mapping of terms to PCTE. Assignment of a term to an RO attribute considers the term value atomic. Please recall, that performance issues motivated terms in the beginning. Incremental modifications at the repository level would defeat this advantage.

At the Lua side, however, terms can indeed be modified incrementally. Programmers just have to be aware, that such incremental modifications only operate on a transient copy, which is made persistent only at the next assignment to an RO attribute. This is the only case, where the interface between Lua/P and the repository is semantically relevant for programmers.

Implementation ▽

Knowing about the capabilities of terms we can now present, how lists of basic types are mapped to PCTE. Lua classes `Pcte_Term_List` and sub-classes for String, Integer and Boolean, implement the List interface by a mapping to a Term of type `STRING_LIST`, `INT_LIST`, or `BOOL_LIST`, respectively. Persistence of such lists is achieved by storing the serialized form of the list term into a string attribute in PCTE. Thus, lists of basic types share all positive properties of terms while providing an interface that is compatible to other types of lists. This technique is far more efficient than directly using any capabilities of PCTE, which would involve some surrogate objects or links to carry all list elements as attribute values. In order to hide the value semantics of the underlying term implementation, modified lists of these types are collected and stored automatically at certain transaction points.

Class List [A.2→].

Requests as transactions [7.4.5→]

3.3 Summary

The distinction between a top level object-oriented meta model and a very fine grained refinement using term types yields a *hybrid* model, where both techniques co-exist. There is a clear relation between both sub-models: term types appear only as attribute types in the object-oriented model, while a term can never *contain* an object, only a reference. The persistence concern dictates an optimization of very fine grained data, that determines some design decisions. But by and large, this optimization is encapsulated by the *mapping* between PCTE and Lua/P.

Objects and terms appear *uniform* to some extent by virtue of their Lua encapsulation. This eventually includes the different mechanisms for references between terms and object structures. However, the difference of reference versus value semantics remains visible to Lua/P programmers.

In order for this integration of techniques to work smoothly also subsequent concerns have to respect the hybrid model.

Chapter 4

Behavior Modeling

Previous chapters have presented PIROL mainly as a specialized database management system. In order move on to an environment that implements certain services and actively integrates an open number of tools, PIROL needs concepts and mechanisms for behavior implementation. In order to provide behavior in close integration with repository objects, repository classes are augmented by method definitions. This lifts the structurally object-oriented repository PCTE to a full OODBMS, supporting the full model of object orientation.

Only two issues need to be specified:

1. Which "flavor" of object orientation is implemented by Lua/P?

2. Which architecture is used to implement method execution?

Ad (1): Lua/P has the regular semantics of method *redefinition* and *dynamic binding*. It allows *multiple inheritance*, but without refined mechanisms for conflict resolution. In case methods of the same name are inherited from more than one super–class, the first class (according to the order of the inherit clause) has priority, i.e., its methods overwrite methods of the same names from other super–classes. Naturally, methods defined in the sub–class overwrite *all* inherited methods of that name.

Overloading is not supported by Lua/P, i.e., methods are never distinguished by their signature. No class can ever have two separate methods of the same name.

Regular methods are also used for *object initialization*. No special constructor syntax (like in C++ and Java) is used, but each class may declare a list of methods, that are suitable for object initialization. If such a *creation method* is declared, object creation must use one of them. If no creation method is declared, an empty method `New` is implicitly defined as the only creation method. Creating an object is denoted by calling any creation method on the class:

```
object = CLASS:creation_method(arg1, ...)
```

This style of object creation mainly follows the style of Eiffel [Mey92].

Calling a super version of a method is done by a qualified (statically bound) invocation:

SUPER_CLASS.method(self, arg1, ...)

It is for technical reasons of Lua, that this call is denoted with a period instead of a colon delimiter. This allows unambiguous distinction between creation calls and qualified calls, but entails the necessity, to explicitly pass `self` as first parameter, which is usually hidden.

Method syntax using ":" [←1.2.3]

Qualified calls should only be used for super calls. They may be used to distinguish between the versions of one method from different super–classes when multiple inheritance is involved.

Ad (2): Lua/P is an interpreted language. The distinction to compiled languages is blurred a little by the fact, that also a byte–compiled variant of Lua code can be used, but conceptually this compilation is irrelevant. The Lua/P interpreter is a distinct component within PIROL, called the *workbench*. For the current discussion, it suffices to know that this workbench is a component that encapsulates the repository, to which it communicates through the PCTE API, i.e., the workbench incorporates the H-PCTE client library, transparently using its proprietary mechanism of inter process communication.

4.1 Behavior Modeling Interacting with other Concerns

4.1.1 Meta modeling

Applications ▽

The meta model started as a pure *data model* for shared storage of data in a common repository. In the era of object orientation it is kind of difficult to see such a structurally object-oriented model without thinking about methods as well. Of course, adding methods to RO classes is a very powerful technique, but the purpose of such methods varies greatly according to the concern that is being captured by ROs.

The ROCM (RO class model) is divided into subsystems or packages. This is how the core packages use methods:

GENERAL

The root classes `ANY` and `ANY_RO` have methods interfacing to the language itself (`conforms`, `get_class`), which could alternatively be implemented as special operators (cf. Java's `instanceof` operator). Also a method `eq` is needed to replace the test for object identity.[1] Here, two oddities from H-PCTE and Lua contribute to a little complication: the H-PCTE API makes no statement whether two different object references really refer to different or the same object. A function `Pcte_object_references_are_equal` must be called for reference comparison. Lua, on the other hand, considers test for identity (`==`) as even more fundamental than all those events that can be redefined using

[1]Note, that this applies indeed to object *identity*. Of course a field by field test for *equality* has to be implemented by methods, but for identity test OOPLs usually have a special operator (`=` or `==`).

tagmethods. I.e., no tagmethod can be installed for this event. For these reasons the programmer must be required to use the `eq` method for comparing object references instead of the `==` operator.

Methods `get_origin` and `get_origin_list` encapsulate knowledge about reverse links that otherwise don't fit smoothly into the object-oriented model. Consider, e.g., the nested structure of `SUBSYSTEM`s which follows the Composite Pattern [GHJV95]: although only links from parents to children are modeled (attribute `subsystems: List(SUBSYSTEM)`) each subsystem also *implicitly* knows about its containing super-system. Although this reference is unnamed it can be retrieved using low level PCTE operations that are encapsulated by this method. In our example this is invoked as `subsys:get_origin("SUBSYSTEM", "subsystems")`. This generates a query for an object of type `SUBSYSTEM` that has a link called `subsystems` pointing to the current object `subsys`. If it is unknown how many objects might have such a link, `get_origin_list` has to be employed resulting in a list of matching objects.

Reverse links [←2.1]

Class `ANY_RO` adds access control (modeled by the indirect association to a `PERMISSION` object). Methods are provided, that allow to change the permission for an object, which ensure that the explicit information in the `PERMISSION` object are always consistent with the effective permissions as defined by access control lists (ACL) at the level of PCTE. ACLs in PCTE allow a very fine grained control over different actions involving a given object or link. Many of these details are hidden by methods of `ANY_RO` and `PERMISSION`. For the user of PIROL it suffices to understand the attributes and methods of these classes.

This structure was depicted in Fig. 2.1 on page 39.

Some classes have simple retrieval methods for simplified access. E.g. class `CO` (conceptual object) has several tuple lists that are intended as mappings $RO \times key \longrightarrow value$. E.g., string values are stored in this tuple list:

Conceptual objects [←1.1.1]

string_resources: List{ro: ANY, key: String, strval: String}

The semantics of the desired mapping is implemented by these access methods:

CO:get_string(ro: ANY, key: String): String
CO:set_string(ro: ANY, key: String, strval: String)

Further classes in package `GENERAL` are `PIROL`, `PROJECT`, `WORKBENCH`, and `FOLDER`, which setup the top-level generic structure of a repository. Methods from these classes are used to

- retrieve a user's workbench object during startup of the workbench process,
- interface to mechanisms of distribution and control integration (see Chap. 7, Chap. 8),
- access common services (see Chap. 11).

PRODUCT

Methods in this package (and related packages `TYPES`, `RELATIONS`) only help to maintain the consistency of its objects and provide utility functions for information retrieval. A package `ROCM` (for RO class model) refines classes from `PRODUCT` for reflective definition of the ROCM. This package significantly contributes to PIROL's bootstrapping process.

Bootstrapping PIROL [4.2→]

PROCESSES

This package serves several goals of which only one mechanism is to be presented here, while other issues are deferred to Chap. 9.

Related Work ▽

While other approaches consider this a primary concern of its own, this thesis pre-assumes the necessity for some process support, but focuses on concerns that are closer to realization. Therefore, process support is only indirectly covered wherever the technical discussion is impacted by this high-level goal. See [ACF97] for an assessment of process centered SEEs.

Process integration [8.1.1→]

△

Package PROCESSES defines a state machine for the states of documents. It is assumed, that in a controlled development process each document has a state that gives evidence on where in the process this document currently sits. A typical sequence of such states would be *busy, proposed, published, accessed, frozen*[Lam94]. There is no general answer, to which degree software support for development processes should be *guiding* or *prescriptive* and how much of the involved workflows should be subject to automization [Gro94].

An automaton of document states is, in any case, a natural piece of automization that could be useful for about every project. Such an automaton consists of one STATE object for each state and one TRANSITION object for each transition from one state to another. Each document has a reference to exactly one STATE object and classes STATE, TRANSITION, GUARD, and ACTION implement the state machine that allows to advance the state of a document according to a defined transition.

As a refinement of these very generic mechanisms, classes PERMISSION_GUARD and PERMISSION_ACTION allow to relate the guards and actions in this state machine to roles and persons, providing for a role based workflow.

Classes PERSON and ROLE [9→]

New Feature ▽

Class ACTION also shows that it may sometimes be desirable to leave the level of static object-oriented programs: This class has an attribute `cmd_strings: List(String)`. Each string in this list is executed as a Lua/P-*script* whenever the corresponding transition fires. Here we profit from the fact, that Lua/P is capable of scripting, i.e., dynamic execution of statements that are given as strings. This is a notable abbreviation over regular object-oriented techniques, were each specific action would have to be programmed by a *subclass* of ACTION. Note, that subclassing in PIROL would involve the whole process of generating and installing PCTE types. This overhead is not justified if just a new implementation to a given method is needed. This can be done interactively without terminating any server process or launching any generator/compiler.

TOOLS

This package has to be deferred to Chap. 8.

4.1.2 Persistence

Discussion ▽

There is no need for any methods related to retrieving objects from persistent storage and storing modified objects back to this storage. We transparently encapsulated persistence such that every modification at the level even of single attributes is immediately made persistent. Methods of the meta model profit from this design and can be seen as orthogonal to the persistence concern. While this holds for objects, loading a segment from the repository needs to be controllable from the level of Lua/P and is encapsulated by methods of the meta model.

Segments in PCTE [←2.2.1], *as user context* [9.1.7→].

4.1.3 Granularity

New Feature ▽

So far, we have seen how to integrate behavior implementation into the object-oriented part of the meta model. Chap. 3, however, claimed that for the very fine grained parts of the meta model, not an object-oriented model prevails, but grammar based terms should be preferred. This hybrid model of object-oriented and term-like objects calls for a similar combination of techniques for implementing behavior. Three issues need to be considered:

1. Is behavior implemented by imperative methods or pure functions?
2. To which modules are methods or functions associated?
3. How are type-based conditionals implemented?

Ad. 1: Sect. 3.2.1 stated that the repository considers terms as *immutable objects* with value semantics. This seems to suggest term precessing by pure functions, but within Lua/P incremental modification of terms is actually no problem. Still, a functional style of programming with terms seems natural, as terms are born from experience with functional programming. The choice is, however, left to the Lua/P programmer.

Ad. 2: In Chap. 3, we said that term types tied to our experience with the Pizza language. Unlike Pizza, Lua/P does not support attaching methods to term types, because the latter are not full-fledged classes. Only list-like terms can be accessed using methods as defined by the interface `List` (see App. A.2). Other than these general purpose methods that are generically attached to all terms, terms are intended primarily for processing using *functions*, for which no corresponding concept of modules is enforced to date. This lack of an additional module concept is basically due to the lack of application of and experience with term processing. Currently these functions can either be global or local to other functions or methods. In the long run each complex operation over terms should be placed in a *module* of its own.

Ad. 3: In object-oriented programming, classes not only form the main unit of modularization, but also provide the basis for a fundamental mechanism of conditional control flow. *Dynamic binding* is the builtin mechanism of dispatching according to the dynamic type of an object. The corresponding mechanism in functional programming is type based *pattern matching*. In Lua/P this is done by a function `t_select` which matches a given term against a list of type patterns. Patterns are given by `t_case` branches. In the simple case, each pattern specifies a type and a function that should be executed, if the term is

```
    function expr2string (t)
      return t_select (t,
(1)          t_case ('@value',
               function (val)
                 return val
               end),
(2)          t_case ({'expr', '@binop', 'expr'},
               function (e1, op, e2)
                 return ("("..expr2string(e1)..op..expr2string(e2)..")")
               end),
(3)          t_case ('unexp', {'@unop', 'expr'},
               function (op, expr)
                 return ("("..op..expr2string(expr)..")")
               end),
(4)          t_case ('exprlist',
               function (list)
                 return "{"..
                      list: foldl ("",
                          function (e, col)
                            if col ~= "" then col=col..", " end
                            return col..expr2string(e)
                          end)..
                      "}"
               end),
             t_otherwise (
              function () return "?" end)
      )
    end
```

Figure 4.1: Using pattern matching for a simple pretty printer.

conform to that type. The function is called with the term as only argument. In addition to the top-level type a pattern may also give a list of types to which the sub-terms must conform. If such a pattern is matched, the sub-terms are passed as distinct arguments to the function. When the string *representation* of a term is desired this conversion can be automated by prepending the `@` operator to the type pattern. Finally a `t_otherwise` branch may provide a default function, that is used if no pattern is matched.

See Fig. 4.1 for a simple pretty-printing function for expressions according to the grammar of Fig. 3.1 on page 51. Note, that in object-oriented programming the standard technique for this problem would be to apply the visitor pattern, introducing far more overhead than the more functional approach.

The first branch in Fig. 4.1 matches subtypes of `value`, the next branch matches any term consisting of an expression, a binary operator and another expression. Branch (3) combines matching of top-level type (`unexp`) and structure (unary operator and expression). All operators and values are passed by their representation (use of `@`). Expressions are passed as terms. Branch (4) again is a simple match by type. It shows an application of the `foldl` function, which is borrowed from ML [Pau96]. We introduced `foldl` to Lua/P as one of the most general higher order functions, that iterates over a list, collecting the results through a second argument (`col`). In this example the effect resembles a smarter mapconcat function: the representation of all list elements are concatenated using `', '` as a separator except for the first element.

4.2 PIROL's Boot Process

Classes appear in four different representations in PIROL:

1. Lua/P source code,
2. Workbench–internal structures (plain Lua tables),
3. ROs as reflexive definition,
4. PCTE types.

Of these four, only two are visible: (1) is the input as created during development and customization of the environment. (3) provides a set of objects available for introspection of the current meta model. This level is currently not used for dynamic changes of the meta model except for one purpose: during deployment.

This is, how a concrete (customized) meta model is installed into PIROL: For the core package `GENERAL`, `PRODUCT`, `RELATIONS`, `ROCM`, `TYPES` and `PROCESSES` type definitions exists as schema definition sets (SDS) in PCTE's data definition language (DDL). For the same packages also Lua/P files are provided. At boot time the Lua/P packages are interpreted several times with different results.

PCTE type definitions [←2.1]

Building the internal structures. One pass of reading each Lua/P package is completely dedicated to building the internal representation of classes as defined in Chapter 2.3.1. The result of this pass is an encapsulation of predefined Lua/P classes, such that ROs can be created and modified without explicit calls to the PCTE API.

Internal structure of classes [←2.3.1]

Creating ROs as representation for introspection. A special Lua module, that is used only during the boot process, reads the Lua/P packages again, but changes the interpreter such that no internal structures are created, but instances of `ROCM_PACKAGE`, `ROCM_CLASS`, `ATTRIBUTE`, `TYPE` and `METHOD`. One of the ideas behind this transformation is to eventually do without any external files, but keep the meta model completely within the repository. Currently, method bodies are not stored in PCTE, but only for the lack of necessity. This encoding of the meta model is, however, used by generic tools, that may dynamically inquire an object's class and all its features. Convenience methods exist, like `CLASS:get_all_features_of_type (feat_type, feat_class)`, which are used by the PIROL Object Navigator (PON, see Sect. 13.1) for displaying, e.g., all string attributes of any given object. Finally, this intermediate representation is used for the next step: creating PCTE type definitions.

Creating PCTE type definitions. This step is only needed for packages outside the meta model core (core packages already have the PCTE types by compiling the given SDS files). Note, that additional packages can be added to an existing installation at any time, not only during first time booting. Class `ROCM_CLASS` simply has a method `compile_schema`, that creates a PCTE type definition with all attributes and links applied as defined by the Lua/P input and its mapping to PCTE. This is performed using a special part of the PCTE-API that allows to create and setup types programmatically. It is noteworthy, that type definitions can be created incrementally, most importantly: If one Lua/P package imports a type from another package that is (maybe due to circular imports) not yet installed, the importing package may already create an empty type in the other package. Only later, when the second package is installed, that empty type is stepwise filled by applying attributes and links.

Incremental type definitions and multiple views in PCTE [16.1.2→]

A future application of this technique might be to develop additional Lua/P packages *within* PIROL as instances of `ROCM_CLASS` etc. With a simple call to `compile_schema` each new class can immediately be used for creating persistent objects.

4.3 Summary

The main concern interaction of this chapter regarded the different levels of granularity and their styles of computation. These two styles co-exist without problems. Functional programming allows to add a new dimension of modularity, because many modules of functions may operate on the same set of types. Modular definition of hybrid traversals concerning objects *and* terms will be discussed later.

→Sect. 10.3.3 will discuss the current issue in the presence of DVCs.

By and large, the addition of methods and functions to the meta model is an *orthogonal* extension, that does not conflict with any previously introduced concern.

Model behavior can rather be seen as a *facilitator*, that helps to define interfaces (between sub-models *and* between concerns), encapsulate techniques and provide a uniform environment, in which services from very different concerns can be exploited in a seamless way. The boot process provided an example where the method `ROCM_CLASS.compile_schema` makes PCTE's capability of incremental schema definition available to clients without the need to worry about PCTE-specific details. The same hold for many methods in package `GENERAL`.

Chapter 5

Exception Handling

Now as we have methods attached to RO classes we have to take into account that each method invocation may fail. Exception handling is not a new issue for software engineering and language design. A very expressive model of *declaring*, throwing and handling exceptions is included in the Java language. Eiffel [Mey92] on the other hand provides a clean embedding of exception handling into its model of design by contract. The emphasis for PIROL lies not in copying one of these approaches or even combining the positive aspects of several approaches, but we were looking for a pragmatic, domain specific approach, that is just "good enough" for PIROL's needs. In PIROL this concern, which naturally cuts across all structures of a system, requires quite different considerations at the different levels of the system. For this reason the actual focus of this chapter lies in the single concern interactions, and similar considerations will follow in subsequent chapters.

5.1 Exception Handling Interacting with other Concerns

5.1.1 Meta Modeling

Applications ▽

At the current point of discourse, it is difficult to separate the meta modeling concern from the behavior concern with respect to their interactions with exception handling. In order to simplify this discussions, both aspects are discussed here.

Regular methods of the meta model do not make frequent use of exceptions[1]. Different conditions that may occur are rather handled in an explicit way. This style of programming emphasizes that exception handling should not be misused as a normal control structure for anticipated conditions.

Of course, the different packages of the meta model have to deal with very different conditions. These are the exceptions typically thrown within each Lua/P package:

[1]In Lua/P, the word "error" is used. In this text both words are used as synonyms, because focus lies on the mechanism rather on subtle differences in semantics.

Access control in a multi user environment [9→]

GENERAL: methods in classes `ANY` and `PERMISSION` dealing with modifying access rights of an RO may fail if the user is not owner of the RO (`NotOwner`).

PROCESSES: Several kinds of access exceptions may occur in the context of manipulating objects of type `PERSON` and `GROUP`, as these have the semantics of representing users and user groups. Several operations require administrator privilege. Most of these operations need to retain consistency with PCTE internal structures. It is the invocation of PCTE functions that may fail and thus throw an exception.

Compiling reflective objects to PCTE type definitions [←4.2]

ROCM: Also this package needs to observe and maintain consistency with PCTE and may throw an exception if it fails to do so.

Generally, exceptions are also used as a workaround for missing abstract declarations of methods: abstract methods are implemented as empty methods that throw an exception `AbstractMethod`. All in all, there is only a small number of methods in the meta model that actively throw an exception.

Thus, exceptions are normally thrown by other levels of the system, and methods of the meta model simply pass exceptions up to the top-level caller. No *catch* mechanism is included in Lua/P. This is in contrast to the capability of plain Lua, where a function call may be a “protected call”, which will catch any error within. This mechanism is only used by the implementation of the workbench in order to catch errors at top level.

5.1.2 Persistence

Discussion ▽

Data integrity [6→]

The occurrence of an exception during a method that may have modified persistent data might leave these data in an inconsistent state. Two additional concerns come to mind. The requirement of data integrity is to be introduced in the next chapter. Secondly, a solution for the problem at hand will certainly include *transactional* protection. Within this thesis, transactions are not considered a top-level concern. This might be arguable but the list of concerns not including transactions already spans a space in which all issues of transactions are covered. This is how transaction related issues are allocated:

Data integrity [6, 6.2.5→]

Transaction abort [7.4.5→]

Client requests and transactions [7.4.5→]

- Transactions are motivated by data integrity
- What cleanup is necessary when aborting a transaction?
- When are transactions started, committed and aborted?

5.1.3 Granularity

Discussion ▽

Having different levels of granularity seems not to influence exception handling. The other way round a special term type `ERROR` is *used* to model exceptions as light weight objects. An `ERROR` term simply combines an integer error code with a string message explaining the exception condition.

5.1.4 Behavior

Most aspects of this interaction have been discussed in the context of meta modeling. Here only one aspect is to be added, that relates to matters of pure language technology.

Applications
▽

Much effort has been put into making the workbench robust against *type errors* in Lua/P programs, as long as no type checker is available. This is mainly a matter of catching the attempt to access non-existent fields of an object. Other exceptions are thrown, when object creation violates the rules concerning creation methods. As soon as a type checker is available, these exceptions will be almost obsolete, but this code has still to be kept for the sake of reflective programming: several methods of the meta model, e.g., retrieve classes by their name, that might be stored in a string attribute. These applications cannot be checked by a type checker.

Creation methods [←Sect. 4 on page 55]

5.2 Summary

The most intricate interactions concerning exception handling are still to come in later chapters. In our first analysis and early drafts of this thesis, exception handling only occurred as a special form of interaction between concerns like data integrity and behavior. Only a closer look revealed that such an interaction could not be cleanly captured as an interaction between only two concerns, but other concerns got involved, too.

This entangling of several concerns in one interaction was taken as a signal, that exception handling should rather be treated as a concern of its own right, allowing for separate discussion of its relationship to all other concerns in the system. It should not surprise, that this newly identified concern adds a little less than a full new dimension to our discussion: from the perspective of exception handling the concerns of meta modeling and behavior seem to collapse to one concern. Without loss of expressiveness this chapter could have completely merged both concerns into one section. The concern interaction matrix need not be complete. Concerns that have empty cells in their row (column) of interaction may be considered weaker than others. They may also be harder to identify. But already a small number of interactions with other concerns should justify the separate treatment of a concern.

There is another oddity in exception handling. This concern is not alone derived from any system requirements. Of course, the requirement of robustness may call for exception handling. But exception handling should also be seen as a *mediator* between other concerns. By its very nature of crosscutting modules of the system, exception handling can influence the control flow in a way, that would otherwise be very hard to achieve. For certain conditions, exception handling is perfectly suited for "teleporting" the locus of control out of a module, just because another module somewhere deeper in the system signaled the violation of some condition.

Together with mechanisms of centrally installed exception handlers exception handling has some similarity to aspect-oriented programming and will be further discussed in that context.

Exception handlers [7.4.5→]
AOP [16.3.2→]

Chapter 6

Data Integrity

6.1 PIROL's Mechanisms for Data Integrity

6.1.1 Encapsulation

PIROL's meta model is in the first place a *data model* for the underlying repository. Methods are only added as an additional mechanism, they are not needed for accessing attributes of objects. We already saw assignments like

o1.a1 = o2.a2

which in contrast to "normal" OOPLs "break" the encapsulation of objects `o1` and `o2`.

Of course, encapsulation is a fundamental technique for ensuring *semantical integrity*. The problem with encapsulation is the single choice of decomposition criteria, disallowing overlapping modules. The effect is quite often, that for the sake of encapsulation one set of decomposition criteria is strictly enforced, while other criteria are not reflected in the system's module structure. Regarding those suppressed criteria, locality and comprehensibility are significantly compromised. This tension, which is the main motivation of *aspect-oriented software development*, will be discussed in Part III.

"Tyranny of the dominating decomposition" [15.1.2→]

PIROL partly relaxes the rules of encapsulation. In Lua/P all attributes are implicitly public. This renders possible some experiments regarding a modularization that cross-cuts the class structure and realizes a significant abbreviation of regular usage of ROs: Throughout large parts of the ROCM `get` and `set` methods would be needed in fact for *every* attribute declared. Such a wide spread usage of trivial access methods adds nothing to the comprehensibility of the system.

6.1.2 Towards Semantical Integrity

While language support for strict, one–dimensional encapsulation cuts both ways, *semantical integrity* should be taken very seriously. Many approaches pay respect to the finding that a structural model should be more detailed than – say – a UML class diagram, in order to carry the intended *meaning*.

Event	Type	Arguments	Description
Simple attributes:			
assign	req.	`value`	assignment of `value` to the resp. attribute of `self`.
get	req.	*none*	retrieve the resp. attribute from `self`.
List attributes:			
adding	notif.	`index, value`	`value` is being added to `self` at position `index`.
removing	notif.	`index, value`	`value` is being removed from `self` at position `index`.
append	req.	} all regular list functions	
remove	req.		
...	req.		

Abbreviations: req. = request — notif. = notification

Figure 6.1: Events for attribute guards

```
AttributeAccess CLASS.SimpleAttribute {
    Event = method (Arguments)...end,
    ...
}
ListAccess CLASS.ListAttribute {
    Event = method (Arguments)...end,
    ...
}
```

Figure 6.2: Syntax of attribute guards

The UML meta model employs well–formedness rules written in OCL[1]. Many database approaches allow to define consistency constraints that relate several values to each other (Examples, discussed in this thesis are [SHO95, Emm96]).

In this tradition, Lua/P introduces the concept of *attribute guards*. An attribute guard allows to intercept the event of accessing an attribute. Two kinds of events can be distinguished: requests and notifications. When intercepting a *request*, the guard is fully responsible of performing the requested action. When intercepting a *notification*, the requested action has already happened.

Also the kind of attribute must be considered: simple attributes define only two request events: `get` and `assign`. For list attributes each List–method corresponds to a request event, and two additional notification events are defined: `adding` and `removing`. Fig. 6.1 summarizes the events, Fig. 6.2 shows the general syntax of a guard definition. The implicit argument `self` refers to the base object for `AttributeAccess` and to the list in the case of a `ListAccess`. The base object can be retrieved from a list by `list.base`.

The effect of triggering an event on a guarded attribute largely depends on the style by which the guard is implemented. Several styles will be discussed in the following sections. By allowing any legal Lua/P statements within guards we are flexible to do very different kinds of things. An additional command allowed within a guard implementation is `raw_`*eventname*, i.e., the standard implementation of the event being intercepted. For lists these raw versions

[1]Object Constraint Language [OMG99]

are temporarily installed methods of List, while `raw_get` and `raw_assign` are temporarily available as global functions.[2] By this mechanism it is impossible to use these raw versions other than within a guard implementation. In other words: there is no way an attribute guard can be bypassed when accessing a guarded attribute.

Note, that this rule is even stricter than the regular class based encapsulation with `get` and `set` methods: (1) It comprises also protection of the contents of list attributes, which would be unprotected if a `get` method would simply return the container object. (2) Guards apply even when acting *within* one object: they are triggered on *every* access, irrespective of the context from which the access is triggered.

A future version of Lua/P might support a slight extension of the guard concept. With little effort guards could also be used as class invariants. An easy solution would associate a class invariant with all attributes on which it depends. Of course, this would require some static program analysis, for which infrastructure is missing in PIROL. Furthermore, methods should be allowed, to temporarily violate a class invariant (see also the discussion on transaction support in APPL/A, Sect. 6.3.1 on page 79).

Tolerating inconsistency [16.1.6→]

At the implementation level, a guard definition is a *meta program* that modifies a class by replacing a normal accessor by a guard. Very interesting effects can be obtained if such guards are attached *dynamically* to a class. Sect. 11.3.6 will report on an application of dynamically attached guards.

Observer mechanisms [11.3.6→]

Related Work ▽

Guards allow access oriented programming à la LOOPS' active values [SBK86]. There is also some similarity to properties in C# [C#01][3], which allow attribute-style access to class members provided that `get`/`set` methods are implemented. In Lua/P standard get and assign functions exist automatically, and guards need to be implemented only for additional actions. A built-in technique for lists and list guards exists in none of these languages. Other languages supporting comparable concepts for integrity are discussed below (Sect. 6.3). △

6.1.3 Avoiding Redundancy

Some constraints in the design of a database schema originate from some sort of *redundancy*. If information is replicated — possibly in different formats — extra effort is needed, in order to keep these interdependent data in sync. Fig. 6.6 below will show a guard that results from such redundancy. PIROL was, however, developed with the intention to *avoid redundancy* wherever reasonable. The fine grained meta model allows to store documents with much less redundancy than any file based approach. Another means for reducing redundancy is the use of functions that compute *derived data* on demand instead of storing these data. Many methods of the ROCM are introduced for exactly this purpose. Meyer proposes the principle of uniform access [Mey97] to hide the distinction

[2]This is another application of Lua function closures (cf. Sect. 1.2.3): these temporary functions encapsulate the values of the current object and the current attribute. The latter is not explicitly accessible within a guard but is implicit context information generated during guard definition.

[3]Lua/P guards where developed before C#.

```
Class ROUTINE {
   inherit FEATURE,
   attributes={
      arguments: List(ENTITY),
      type: TYPE
      signature: Derived(String),                              -- Declaration
      ...
   }
}
...
function ROUTINE:derive_signature ()                   -- Derivation Function
   local args = self.arguments:foldl ( "(",
      function (arg, pre)
         if not pre == "(" then pre = pre..", " end
         return pre..arg.signature              -- read another Derived Attribute
      end)
   args = args..")"
   if self.type then
      return self.name..args..": "..self.type.name
   else
      return self.name..args
   end
end
```

Figure 6.3: Derived attribute `ROUTINE.signature`

between attribute queries and query functions. In Eiffel, an expression `o.a` may either refer to reading an attribute `a` or calling a nullary function `a()` on `o`. Meyer argues, that for functions without side–effect, the client should not bother with the difference between such an explicit function and the implicit accessor by which the value of an attribute can be read. For this reason the syntactical difference between `o.a` and `o.a()` as it appears in many programming languages is leveled. In Lua/P only special features — declared as derived attributes — are interchangeable with attributes.

In Lua/P a *derived attribute* consists of two parts: a variable declaration and a function definition.

As an example of derived attributes, consider the signatures of routines as modeled in the RO-class `ROUTINE` (Fig. 6.3). Names, arguments and result types of routines are kept persistently either as direct attribute (name) or using separate repository objects of types `ENTITY` (arguments) and `TYPE` (result type). It should on the other hand still be possible to query the signature of a routine (encoded as a human readable string) with just one query. The structural definition of `ROUTINE` declares an attribute `signature: Derived(String)`. This attribute must be *implemented* by a derivation function, whose name is constructed according to the convention of prepending the attribute name with '`derive_`'. With these definitions in place, the expression `my_routine.signature` is evaluated as `my_routine:derive_signature()`. The body of the derivation function makes use of the compact style of the `foldl` higher–order function.

Higher order function `foldl` [←4.1.3]

The first benefit of these syntactical manipulations is that evolution is made easier, because now the implementation may switch between an attribute and a function without visible impact on usage clients. Switching from a function — i.e., a derived attribute — to a stored attribute is always safe. This typically happens, when a value that could be computed shall later-on be cached persistently for reasons of performance. The opposite — migrating from stored to derived — is only safe, if no client performs any writing access to the attribute, which would break in the case of a function. This asymmetry has been noticed and a similar construct exists, where both directions — reading and writing — are implemented.

redirect *construct* [10.2.3→]

The prevalent motivation for derived attributes as a middle ground between attributes and functions relates to component communication in PIROL.

Integration of derived attributes and change propagation [8.3.6→]

Avoiding redundancy should always be the primary goal. Only if this is not reasonable, attribute guards may be used to safeguard the integrity of redundant data.

6.1.4 Technical Integrity

Another issue of data integrity is *referential integrity*, which will be discussed below in Sect. 6.2.2.

While referential integrity ensures the existence of a target object for every link, type safety is to ensure that the type of the target object matches the expectations of the referring context.

Discussion of static typing [14.2.3→]

6.2 Data Integrity Interacting with other Concerns

6.2.1 Meta modeling

Applications ▽

Attribute guards were introduced as a means for enforcing the semantics of the meta model. With attribute guards we now have a way of encoding invariants over the meta model. Two styles can be distinguished: Restrictive and operational guards. Fig. 6.4 gives an example of a *restrictive* guard. The invariant is

self.main_routine:feature_of() = self.root_class

In other words: The main routine must be a feature of[4] the root class. If an assignment to `main_routine` tries to violate this invariant, an error is raised. The effect of raising an error within a guard will be discussed below (Sect. 6.2.5).

An *operational* guard is shown in Fig. 6.5. This guard ensures the constraint, that a class containing at least one abstract method must be abstract, too:

r.is_abstract ⇒ r.feature_of().is_abstract

[4]Class FEATURE (of which ROUTINE is a subclass) has no explicit reference to the containing class, but method `feature_of` can retrieve this information based on implicit reverse links (cf. 4.1.1) by a call self:get_origin(`"CLASSIFIER"`, `"features"`)

```
 Class {SYSTEM;
    ...
    attributes = {
        root_class: CLASS,
        main_routine: ROUTINE,
        ...
    }
    ...
}
...
AttributeAccess SYSTEM.main_routine {
    [assign] = method (routine)
        if self.root_class:eq(routine:feature_of()) then
            raw_assign(routine)
        else
            pirol_error(_ERROR.GuardException,
                "SYSTEM: main_routine must be of class root_class.")
        end
    end
}
```

Figure 6.4: A restrictive guard for `SYSTEM.main_routine`

```
AttributeAccess ROUTINE.is_abstract {
    [assign] = method (flag)
        if flag == self.is_abstract then return end       -- Nothing to be done.
        raw_assign(flag)                            -- Actually perform the assignment.
        if flag then                            -- Only consider transition false→true.
            local class = self:feature_of()
            class.is_abstract = flag                                -- Propagate the change.
        end
    end
}
```

Figure 6.5: An operational guard for `ROUTINE.is_abstract`

This guard does not raise an error but propagates the change to the containing class, if needed, thus enforcing the constraint operationally.

The next example, Fig. 6.6, shows an operational guard for an `adding` notification on `SUBSYSTEM.classifiers`. Here the constraint is

$$c \in \text{c.subsystem.classifiers}$$

saying a class must be contained in the list of classifiers of its subsystem. This constraint is necessary because the reference `c.subsystem` is partially redundant and must be kept aligned with the reverse `s.classifiers`. This is no full redundancy since a classifier may be referenced by many subsystems, but it refers to exactly one subsystem in which it is defined.

```
ListAccess SUBSYSTEM.classifiers {
  adding = method (index, class)
    if not class.subsystem then              -- Don't overwrite existing link.
      class.subsystem = self.base
    end
  end,
}
```

Figure 6.6: A list guard for `SUBSYSTEM.classifiers`

For this constraint class `CLASSIFIER` is primarily responsible, but adding a classifier to a subsystem should also synchronize the `subsystem` link, which is shown by the guard in Fig. 6.6. Associating the guard with the `adding` event generalizes from the concrete list method that is used. Thus the guard is triggered for each object that is being added to the list.

This section has shown examples of how attribute guards can give additional *meaning* to data in the repository. The examples were confined to relationships *within* the meta model. Sect. 9.1.6 will show, how attribute guards can integrate additional *low level features* of PCTE into the meta model. Sect. 8.3.6 will go the opposite direction showing an attribute guard that controls *distributed components*.

6.2.2 Persistence

New Feature ▽

The conceptual architecture of the PIROL workbench assigns to attribute guards a place very close to the bottom, right next to the persistence layer. I.e., any request to modify persistent data has to pass the corresponding guard, if one is defined. This ensures that persistent data *always* satisfy the guard conditions.

Another type of constraints is already ensured by the underlying repository. PCTE guarantees *referential integrity* at any time. This means that links[5] will never point to deleted objects. This can be compared to programming languages that have no explicit operator for deleting an object, but only references can be deleted. Objects are only deleted implicitly after the last incoming reference has been deleted (garbage collection). As a result of this mechanism — both in PCTE and in languages with automatic garbage collection — all objects and links always constitute a connected graph with an explicit root object from which all other objects are reachable via links.

This places a special burden on Lua/P: whenever an RO is *created*, PCTE requires a link pointing to this object, otherwise it cannot be persistent. It is, however, a very convenient and frequent practice to assign new objects to local variables, first. Only later within the same method (or its calling method) a persistent link attaches the new object to another RO. This is not possible with PCTE.

The first approach in Lua/P required that every object creation already spec-

[5]Only links of certain categories (cf. Sect. 2.1) This discussion does, e.g., not apply to implicit links.

ified an origin object and a link, by which the object could be attached to the graph of persistent objects. This required a special function `Linked_New` with non-standard syntax[6]. It could, however, not be tolerated that every object creation already *knew* this origin RO and the attaching link. Consider, e.g., a method `clone` that should create and return a copy of the target object. Only its application context can create the link needed for persistently creating the object. Not to speak of the confusion that would result, if different versions of `clone` would be needed, one for attaching the new object using a regular link, the other one for inserting it into a List. All this would break the encapsulation of PCTE in that too much knowledge about the repository would be visible within Lua/P. Programming in Lua/P would have to be aware of PCTE and its concern of referential integrity, while PCTE — other than programming languages — does not know about local variables.

The solution in PIROL involves an explicit modeling of *dangling objects*. Objects that are not created by `Linked_New` but using the syntax

```
lua_var = CLASS:creation_method(arg1, ...)
```

are attached to an object of type `WORKBENCH` using a list

```
dangling_objects: List{
   obj: ANY,
   persistent: Boolean
}
```

This list acts as an “orphanage” for newly created objects. Links from this list are used to keep objects alive that are not otherwise reachable. The next attachment of an orphan to a living object will cut the link from the orphanage — the object is “adopted”. Special housekeeping is needed in order to figure out whether an orphan is attached to a live object or just to another orphan. When the workbench process shuts down, it wipes the list of dangling objects deleting all orphans that have not been adopted. This deletion can be prevented by setting the `persistent` flag in the above tuple definition. Objects with this marking are not deleted, even if the `dangling_objects` link is the only link by which the object can be reached.

Tools may use that flag, if objects are kept in this list over a longer period of time. Thus an orphan can be protected against data loss, in case of unexpected termination of tool or workbench.

Discussion ▽ Note, that maintenance of the `dangling_objects` list involves some effort in caching administrative data and has to be considered during each creation of a link. This is complicated by the fact, that dangling objects my refer to each other transitively and changing the status of one of these objects (either deletion or attachment to a non-dangling object) can require to propagate that change to many other dangling objects. So, from the *performance* point of view, one might be inclined to avoid dangling objects completely.[7] Most statements of

[6]The syntactical problem originated from the need to combine object creation, invocation of a creation method for initialization *and* link creation into one function call. The resulting syntax is: Linked_New(originRO, "linkName", "className", "creationMethodName", arg1, ...).

[7]Surprisingly, measurements (Sect. 14.1.2) showed that for standard situations the penalty imposed by dangling objects can well be tolerated.

object creation in Lua/P could in fact be transformed into a call to `Linked_New`. This could be done by a simple *preprocessor* for Lua/P, which already exists for minor syntactical improvements. However, in a distributed setting this approach would place severe restrictions not only on methods of the meta model but also on the implementation of *tools*, which cannot be tolerated.

→See Sect. 7.3.2 for the full discussion.

△

Integrity of cached data

Caching of lists [←2.2.1]

Caching has been introduced as a natural complement to persistence. This will have to be considered in the discussion of aborting a transaction. The same holds for transient objects. Both kinds of data are outside the rule of PCTE.

Transaction abort [7.4.5→]

Transient data [←2.2.1]

6.2.3 Granularity

Discussion

▽

Term grammars extend the explicit modeling of structures below the level of classes. Thus, *structural integrity* can be ensured for any desired level of granularity.

Support for very fine grained *semantical integrity* is less explicit. Currently, guards can only be applied at the level of attributes of an object. Elements within a term structure do not have guards. Implementing guards for the semantic integrity of terms requires more effort but can be done using attribute level guards, since only those terms are persistent that have been asigned to an RO attribute.

Referential integrity needs to be reconsidered for term structures, because these structures introduce objects about which PCTE knows nothing. So, PCTE cannot be responsible for their referential integrity. The precise phrasing of the problem at hand is: how are references between terms and objects implemented and how can their integrity be ensured?

For object references from within a term, a naïve solution might use a unique object identifier encoded to a term of type `OID` which would be a subtype of `STRING`. This, however, conflicts with the requirement of referential integrity. Imagine the contents of some term referring by `OID` to an object which is later-on removed from the repository (by removing all links leading to the object). This would render the `OID`-reference invalid, i.e., pointing to a non–existent object. This can only be avoided if the repository is made aware of this reference.

New Feature

▽

The dilemma is solved by introducing an indirection called *indirect reference*: each object containing one or more terms has a list of objects, which are referenced by its contents. This list exists at the repository level as a set of links. Within a term only indices to this list are used. Translating indices to objects and back can be transparently automated by the functions that store and retrieve terms[8]. Thus the responsibility for referential integrity is split into two parts: the Lua/P interpreter is responsible for synchronizing references out of a term with actual PCTE references. PCTE is then responsible for the validity of this PCTE reference.

A similar situation arises when attaching *HTML contents* to ROs. For this

[8]Automatic translation is not yet implemented, i.e., the indirection is still explicit.

purpose each RO of type `ANY_RO` may have a `DESCRIPTION`[9], that carries a binary attribute `text`. When using HTML as the format for this text, references to other ROs may be included using a textual encoding to be shown later. Also these references from HTML to ROs are implemented using the indirection shown above, i.e., the encoding uses only numbers that are indices to the regular list

PIROL *URL protocol* [11.3.8→]

```
referenced_by_contents: List(ANY)
```

Another issue of referential integrity concerns the deletion of terms. Within the Lua/P interpreter (disregarding persistence), terms (and sub-terms) are handled using reference semantics. Terms are implemented by a C library, that is accessible from Lua only via API functions. From this follows, that Lua has only partial knowledge about references to terms. Additional reference counting is implemented in C, which aids the Lua interpreter in performing *garbage collection* of terms and sub-terms. This is not trivial because the Lua garbage collector may trigger collection of a term, that has sub-terms which remain reachable and must not be collected. This complication is, however, limited by the fact, that cycles in term structures are not allowed. A hook of the Lua garbage collector is used to decrement this count whenever a term ceases to be accessed from Lua. This results in a hybrid and multi-language garbage collector.

6.2.4 Behavior

Discussion ▽

Methods in two ways *help* to ensure data integrity: methods may be used as functions that compute data that would otherwise be stored in a *redundant* attribute. This reduction of redundancy avoids the problem of data integrity for some cases.

Secondly, access methods may implement conditional modifications of attributes, that don't allow invalid values and combinations. The drawback of such *access methods* is the difficulty of enforcing their usage. Every method that is added to a class may introduce a hole, by which the original access methods can be bypassed.

Conceptually, attribute guards are stricter than access methods, because they are attached directly to their attribute. Unfortunately, experience with attribute guards in PIROL has already pointed at situations, where this rule is too strict. Further investigation should show whether this issue should be solved by improved language technology or by methodology. The rule could be proposed that guards should be used where absolute strictness is adequate. As soon as the guard needs to be bypassed in any situation, its constraint should rather be implemented by a regular access method instead. This leaves the choice of providing alternative access strategies.

→See Sect. 9.1.6 for an example.

Another approach to more flexibility might be to declare different scopes for attribute guards. A strict guard is triggered by every attribute access and can never be bypassed. For less strict guards, methods of the same class might

[9] As shown in Sect. 2.2.1, this object is usually inlined with the `ANY_RO`.

be allowed to bypass the guard, enforcing strictness only for external access. Such scoping has not been implemented in Lua/P. Scoping should be part of a rigorous type system for Lua/P.

Discussion of static typing [14.2.3→]

Another reason for using attribute guards in places where one would expect regular methods will be presented when talking about distributed components.

Behavior implementation using guards [8.3.6→]

6.2.5 Exception handling

Discussion ▽

Exceptions cause a program to immediately leave the current call stack. This could result in severe data corruptions. Lua/P does not include a mechanism for catching exceptions because this aggravates the danger of data corruption. With exception catching, data may be in an inconsistent state *without anyone noticing* that an exception had occurred. A clean solution would have to follow the lines of Eiffel's *rescue* and *retry* mechanisms [Mey92], which ensure that the method causing the exception eventually either succeeds or throws the exception to its caller. Such a mechanism seemed too complicated for Lua/P and instead data integrity in the presence of exceptions will be ensured by transaction roll-back.

Transactions [7.4.6→]

Moreover, exceptions are used in order to *enforce* integrity constraints. Fig. 6.4 on page 72 presented a guard that raised an exception in the case of detecting a constraint violation. The exception is in fact a means of enforcing data integrity in such a case by rolling back the current transaction. Otherwise partial modifications might leave the object in an inconsistent state.

Every assignment in a Lua/P program can possibly trigger a `GuardException`. In Lua/P methods do not declare which exceptions can be thrown like this is enforced in Java. It is not clear whether such exception declarations can be adapted for this setting. Of course, `GuardExceptions` could always be regarded as `RuntimeException` according to Java terminology. `ClassCastExceptions` and `ArrayStoreExceptions` belong in this category, for which no declaration is needed. The common motivation is, that those exceptions are not triggered by a method call but by a more fundamental language construct. Their declaration would pollute method declarations to the brink of unreadability.

6.3 Language Support for Consistency

Related Work ▽

Several projects on SEEs have brought about language support for consistency. As we have pointed out several times, PIROL was designed with the goal of minimizing redundancy thus preventing inconsistencies in the first place. A different road is followed in [EAMP97], where redundancy is deliberately tolerated in the database schema. On the basis of detailed dependency constraints three strategies exist to choose from:

- *Constraint enforcement* simply disallows user input that would lead to inconsistencies,
- *Change propagation* calculates which document is affected by a modification and sends a message to the corresponding tool that is then responsible for re-establishing consistency, (see 8.2.2)

- *Violation toleration* means to let the user create temporary constraint violations. Tools should then highlight these violations in order to guide the user to eventually correcting the inconsistency.

In the sequel, two languages are discussed, that have dedicated support for consistency: APPL/A [SHO95] and GTSL [Emm96].

6.3.1 APPL/A

Within the Arcadia consortium [TBC+88] an extension to Ada has been developed, called APPL/A [Sut90, SHO95]. This language was developed before the introduction of object-oriented concepts to Ada. This may explain the dominant use of value semantics instead of references, which are at the heart of object-orientation[10]. Still, APPL/A features some concepts that are well worth comparison. The contributions of APPL/A are: relations, triggers, predicates and transaction support. All four concepts will be presented briefly; the focus regarding consistency lies on triggers and predicates.

Relations. APPL/A, being a pre-object-oriented approach, lifts relations as known from relational databases to the level of a programming language. Technically, relations are modules that encapsulate a list of tuples. As stated above, attributes of tuples have value semantics. Relations may have **out** attributes, which are implemented as a function with explicitly declared dependencies. APPL/A relations are said to be freely programmable. This includes the choice of lazy versus eager computation of computed attributes, which is completely up-to the programmer, which probably means, that no caching nor dependency management is built into the language. Thus, computed attributes have a similar intention as derived attributes in Lua/P, however with less support from the infrastructure.

Dependency management for derived attributes [8.3.6→]

Triggers. Triggers are defined in separate units, that are attached a-posteriori to relations, i.e., relations are ignorant of their attached triggers. Communication between a relation and its triggers is by implicit invocation in conformance with the Mediator model by Sullivan and Notkin [SN92]. More specifically, triggers can be attached to operations of relations either on acceptance or on completion. This can be compared to before and after methods of CLOS [Kee89] or before and after advice in AspectJ [KHH+01]. However, triggers do not have the opportunity to influence the behavior of the associated relation, neither by parameter passing nor by throwing an exception or otherwise aborting an operation.

Triggers can be compared to uses of attribute guards in Lua/P, where the guard propagates value changes to other persistent units. Just like list guards, triggers can be attached to either of the primitive operations. Lists in Lua/P

[10] Interestingly, [TC93] within the same project, argue that "the lack of identity in the semantics of relations and relationships that database systems define is inappropriate in many software engineering applications, where both types of objects may have to be shared by numerous other objects". APPL/A does not support references based on object identity.

have a richer interface than relations, which motivated the introduction of additional events adding and removing as the unions of all adding and removing operations. In contrast to triggers in APPL/A, Lua/P guards have full control over execution of the original operation, including modification of parameters and exception throwing. Triggers are top-level modules and some triggers can even be instantiated multiply. Guards in Lua/P are conceptually part of the class to which they apply. Technically, also guards can be attached from any point of the program (currently, they can, however, not be removed again). It is not perfectly clear, what scoping rules should apply to guards.

Finally, triggers have a synchronization problem that prohibits access to the underlying relation within a trigger. Guards in Lua/P have no such restriction.

Predicates. As a complement to triggers, predicates in APPL/A may express constraints over relations. A predicate may relate to more than one relation. Different modes exist, controlling visibility and enforcement of predicates. A "mandatory" predicate is globally visible, while "optional" predicates must be included explicitly to be applicable. Automatic enforcement can be switched on and off dynamically, default enforcement state can be declared in the predicate definition. If an operation of a relation results in a violation of an enforced predicate, the operation is automatically un-done and an exception is thrown.

Enforced predicates behave like guards that for certain conditions explicitly throw a guard exception. The most interesting property of APPL/A predicates is the support for dynamic activation and deactivation, which is not present in Lua/P. Technically, the distinction between triggers and predicates seems unnecessary, but from a method point of view, this makes the multi-paradigm approach more explicit. Triggers are used for coordination while predicates in a rule–like style express constraints. For this purpose, APPL/A features the three quantifications *every*, *some* and *no* like in

every s **in** Source_Repository **satisfies** bool_func(s) **end every**

Transaction support. APPL/A separates five properties of transactions. In this respect it is similar in intention to Ostermann's approach to untangling object composition [OM01]. The result is in both cases a flexible application of more or less orthogonal concepts instead of predefined bundles like "inheritance" (concerning composition) or "synchronized" (concerning transactions).

The *serial* and *atomic* specifiers define blocks with reader/writer synchronization (serial) and all-or-nothing behavior (atomic). In this terminology, PIROL only supports atomic transactions.

Integrity by means of transactions [7.4.6→]

Three specifiers, *suspend*, *enforce* and *allow* define blocks in which the set of enforced predicates is temporarily changed. This mainly encapsulates the (de-)activation of predicates as mentioned above. An *allow* block adds the capability to allow violations that exist at entry of the block to perpetuate but disallowing any new violations to be introduced.

Multi-paradigm support for consistency. Triggers, predicates and guards all introduce some capability of programming outside the pure imperative paradigm. In all cases the run-time system invokes user defined functions. Both approaches, APPL/A and Lua/P, are quite promising in this regard. Differences are imposed by different pre-assumptions. APPL/A builds on a traditional relational schema with value semantics. Lua/P extends object-oriented programming, thus relying on classes, objects and references. Conversely, APPL/A is more advanced in terms of synchronization, which builds on the strong concepts of Ada. The version of Lua used for PIROL does not support multi–threading. On the other hand, basic transactional support is built-in to PIROL without further coding effort in Lua/P programs.

Threads and transactions [9.1.7→]

Transactions and roll-back [7.4.6→]

6.3.2 GTSL

The GOODSTEP Tool Specification Language [Emm96] has already been discussed in Sect. 2.4.2. Here we only add to this by mentioning the concept of semantic rules in GTSL. Such a rule consists of a condition and an action. The condition is composed of the three fundamental predicates `CHANGED`, `DELETED` and `EXISTS`. From this follows, that rules correspond in fact to APPL/A triggers not predicates, because the given conditions are actually events[11]. Actions are defined as a list of statements. In this, GTSL rules appear as a minimalistic set of concepts. The main purpose of these rules is again propagating changes between documents in order to maintain consistency. Considering that GTSL has been developed later than APPL/A, more sophisticated concepts might be expected concerning consistency. Regarding transactions, the central article [Emm96] only mentions a few issues on object locking.

6.4 Summary

Data integrity is a very fundamental issue and imposes obligations to many other concerns (e.g., the scheme of indirect references, or hybrid garbage collection for terms). Although integrity cross-cuts many other concerns, the mechanism of attribute guards as a means to ensure this integrity, is such located in the overall architecture of PIROL that it applies below other mechanisms, i.e., it cannot be bypassed. At the same time it can exploit all concerns that are made available at the level of Lua/P, including and especially Lua/P methods.

This structure is, in a way, a circular dependency between concerns. This is achieved by framework technology: Lua/P is a framework, that provides as hot spots all events of accessing an attribute of an RO. Attribute guards may intercept these accesses and react in very specific ways. This is a restricted style of *meta programming*, providing the developer of the meta model (or extensions thereof) with a great power. It is obvious, that badly implemented attribute guards may compromise the system in unpredictable manners. Thus testing or other forms of verification or validation should be performed with special care

[11] It should be assumed, that `EXISTS` be only used for further qualifying an event. If this assumption is false, rules can indeed be used as predicates and triggers.

for all attribute guards, such that executing a guard either yields “the expected result” or raises an exception.

The implicit maintenance of referential integrity and the explicit mechanism of attribute guards *enrich* the technique of meta modeling towards carrying more semantics. For this mechanism exception handling acts again as a *mediator* that may influence the control flow and trigger roll–back of transactions (cf. Sect. 7.4.6). Attribute guards and methods compete in certain situations and methodological support should help in the choice of the most appropriate technique.

Allocating responsibility. One issue occurred, of which it is not clear, whether it should be solved by changes to the language or by guidelines on good design. Certain constraints by their very nature relate to two or more attributes, possibly of different objects. However, a guard has to be attached to one attribute in particular. If one chooses to attach guards to two related attributes and lets each guard propagate changes to the other attribute, an infinite recursion is programmed, which is difficult to detect. If both guards actually agree on the changes to be performed, it suffices to check before propagation, whether the update actually changes the current value. Thus, well-behaved guards will immediately come to a fixpoint and terminate. Such checking could indeed by performed by the run-time system, as to trigger guards only for value changes. It should be investigated, whether this yields unexpected results for situations, that might rely on guard invocation at *every* assignment.

Chapter 7

A Client–Server Architecture

All that has been said so far actually neglected the architecture according to which PIROL is built. This chapter will unveil how tools can connect as client components to a server that is called the PIROL workbench.

At the time when the development of PIROL started, the notion "component" was far less defined than it is today. This gives us a chance to take an unbiased look at the driving forces that led to PIROL's specific component model. With the transition to a distributed component architecture plenty of secondary requirements come into focus, many of which heavily interact with each other and with the concerns presented so far. This chapter will first present the new forces and mechanisms in a somewhat linearized fashion. The next two chapters are tightly coupled to the issues at hand: "Control Integration" and "Multi user capability". The distributed architecture will gain full relevance only in the light of these subsequent chapters. In the summary of chapter 8 some of the dependencies and interactions will be discussed with more details, some of which will only be presented in that chapter.

This chapter should not be mistaken as an attempt to *deduce* architecture and mechanisms for PIROL or even for component systems in general. The claim is: the presented combination of concepts and mechanisms forms a *sound* architecture that solves many issues of repository based environments while clearly defining which issues have not been solved completely and which forces should be taken into consideration in order to close the gap.

7.1 Decoupling and Integration

PIROL is meant to be composed from various tools possibly made by different developers. The set of tools, functions and services is to be kept flexible in order to service various projects with different needs. From a perspective of research in software engineering, flexible tool integration is even more important. New software technology mostly comes with new tools, but the usefulness of technology and supporting tools can only be evaluated in full, if the new tools can be used in combination with tools for other issues and dimensions of software development that have not been changed. A full-blown software engineering environment simply requires so much development effort, that no

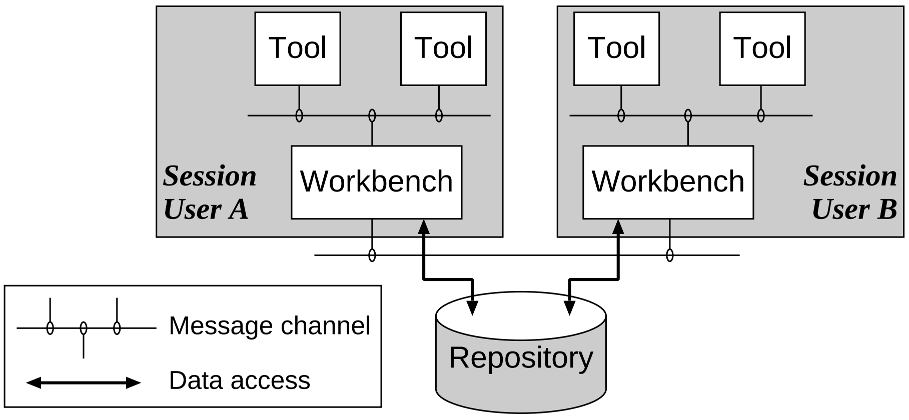


Figure 7.1: PIROL's three–tier architecture

research project, which may succeed to create a tool prototype for some new method or technique, has a chance to provide a complete environment that would support industrial strength evaluation in real world projects. Somewhat realistic comparison can thus only be achieved, if most tools can be reused over a long time and only the relevant tools for supporting new techniques are replaced.

So for a number of reasons decoupling of tools became an issue. One requirement on the road of decoupling is the independence of any single programming language. Tool developers shall be free in their choice of a programming language. Not only shall developers be free to use the language they are most experienced with. Also the existence of certain class libraries may be a reason to use different languages for different tools and tasks. This force can be summarized as the quest for the greatest possible independence of separately developed parts.

On the other hand all these independent parts should contribute to the overall system in such a fashion that users would perceive it as just one (compound) tool. In order to provide such a uniform appearance of separate modules concepts of integration come into focus. For the moment, however, let us focus on the client-server architecture and how tools connect to the server.

Dimensions of integration [8.1.1→]

7.2 A three–tier architecture for PIROL

The simple picture of PIROL's architecture given in Fig. 7.1 shows the three tiers *repository*, *workbench* and *tools*. It shows that each tool is only connected to a workbench. One workbench together with all tools running in its context is called a *user session*. The figure also shows two different kinds of connections: while the workbench exchanges data with the repository using the proprietary protocol of H–PCTE, for other forms of communication *message channels* are used, that will be presented in the next section.

These are the responsibilities of the different tiers:

Repository: Persistent storage of data.

Workbench: Execution of Lua/P methods, guards etc. Definition of a user's context. Coordination of tools.

Tools: Interactive manipulation of ROs and documents.

In order to further examine client-server communication in PIROL, the mechanisms for connecting tools to the workbench are to be presented next.

7.3 MSG: the communication channel

At the core of connecting tools in PIROL lies the MSG facility which is taken from the FIELD environment [Rei90]. The first prototype of PIROL had used the ToolTalk system [Sun93], which is a successor of MSG. While comparing ToolTalk and MSG the latter was chosen because of its portability across different Unix platforms. Also the availability of MSG's source code proved very useful because specialized mechanisms required modification of MSG.

While MSG is a quite sophisticated messaging facility, for the issues at hand it suffices to know that it realizes socket based communication between different *OS processes*[1].

This communication is mediated by a special message server but the benefits of this server will only be visible in the following chapter.

→ The issues of Chap. 8 will exploit this server.

7.3.1 Synchronization

The next dimension in message communication concerns the synchronization between OS processes. MSG provides two different functions for sending messages: the `send` operation can be seen as an event. The sending client does not wait for any acknowledgement or reply (asynchronous notification). The `call` operation waits for another process to `reply` to the message and receives the data of the reply (synchronous request).

Implicit invocation [16.3.1→]

Also at the listening side synchronization provides options. In its normal mode the MSG server keeps a message queue for each connected client, such that only one request is sent to the client and all further messages to the same client are held back until the request returns.

As part of the embedding of MSG into Lua, the workbench provides a function `eventually_call` which detaches a request from (a part of) its action. The protocol is illustrated in Fig. 7.2. This function hooks into the MSG client protocol, such that is sends a reply before performing further actions. Note, that generally replies are not explicit in the Lua encapsulation of MSG, but the return value of the function answering a request message will automatically be sent back as reply. `Eventually_call` bypasses this standard behavior, such that a synchronous request may be used to initiate an asynchronous action. The function `eventually_call` can be used several times before returning control to the caller, thus scheduling several actions for execution. The effect is, that after the reply tool and workbench continue operation in parallel.

[1]Since in the context of SEEs, the notion process is so heavily overloaded the prefix OS is used to discriminate *operation system processes*.

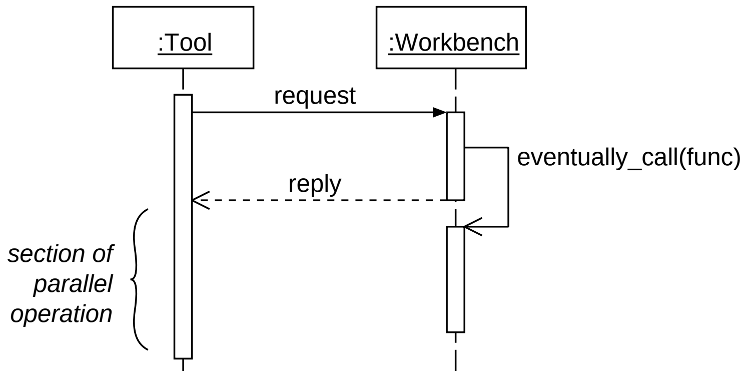


Figure 7.2: Delayed operation using `eventually_call`.

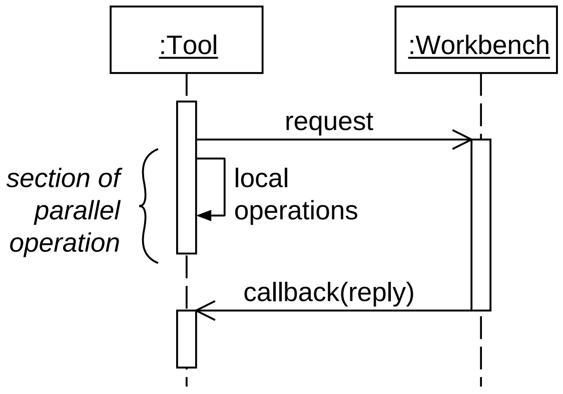


Figure 7.3: Detached request with callback.

Another mode of parallel operation can by achieved in MSG by detaching a reply from its request. See Fig. 7.3 for this protocol. By this mechanism a client can asynchronously send a request while registering a callback function to be invoked when the reply is received. This combines the semantics of a request with non-blocking behavior of a notice. The main difference to the previous strategy lies in when the reply is sent: before or after the section of parallel operation. In PIROL, only `eventually_call` is used. There was no reason to have the client (tool) determine which requests are detached and which are not.

7.3.2 Enhancements of MSG

When integrating MSG into PIROL some encapsulation and modifications have been carried out. The most important modification concerned the integration of MSG with those term types introduced in Chap. 3.

MSG and term grammars

In the context of MSG, term types are used for type–safe transmission of values, simple and structured. This makes use of the serialization capabilities of terms. Such packing of data is commonly called *marshalling*. Different client libraries map term types to native types of the involved programming languages, currently C/C++, Lua and Java. This serves for some degree of interoperability, with a limited number of basic types (INT, BOOL, STRING, CHAR) but unlimited capability to construct union and power types.

A special term grammar `MSG` is used to define all legal types of PIROL messages. Only a small number of message types with few variants in signatures are needed.

- `query`: a tool requests a value from the repository
- `roset`: a tool requests to set the value of an RO attribute.
- `execute`: a tool requests to execute a method of the meta model.
- `create`: a tool requests to create an RO.

Two additional message types will be explained in the next chapter:

Additional message types [8.2.1→]

- `changed`: the workbench broadcasts the change of a value in the repository.
- `tool_execute`: the workbench requests a tool to execute a method.

The complete grammar of messages is given in Fig. 7.4. It is a verbatim listing of the Lua/P grammar definition. Details for List–related messages will be given below (Sect. 7.4.3 on page 94). The benefit of this grammar is to provide type-safe messaging without hard-coding message types into MSG. In order to ensure that workbench and tools operate on the same grammar, the message server also maintains term grammars and provides these to each client as part of the connecting protocol. Type mismatches will be caught by the term library, when definition and usage of a type do not correspond.

Proxy classes

In order to further illustrate the meaning of the above message types, the Proxy design pattern [GHJV95] comes into play: Libraries of *proxy classes* encapsulate MSG based communication for each specific programming language that is used for tool implementation. The goal of the proxy pattern is to provide objects that encapsulate access to remote objects while offering the same interface as the "real", remote objects. For each *attribute* of the meta model the proxy class contains a get and a set method that translate the call into a proper `query` or `roset` message. Each *method* of the meta model is naturally mapped to an `execute` message. `create` messages will be dedicated a more detailed look later in this chapter.

Creation using proxies [p.92→]

See Fig. 7.5 for examples: It shows an RO, `face: ICON`, in the workbench and a corresponding proxy object in a tool. Accessing the attribute `ICON.name` happens through the two methods `getName` and `setName` which are implemented by sending a `query` and a `roset` request respectively. Method execution is presented by the example of `ICON.import_data`, implemented by a `execute` request. Creating an `ICON` RO finally is triggered by the static method `make_ICON` which maps to a `create` request using method `ICON:make`.

```
Grammar {MSG;
                                data types
  roid                 = subtype_of{STRING},
  feature              = subtype_of{STRING},
  roid_list            = {roid, "*"},
  entity               = {{name=STRING}, {val=entity_val}},
  entity_val           = one_of{STRING, INT, BOOL, roid, tuple,
                                STRING_LIST, INT_LIST, roid_list, tuple_list},
  tuple                = {entity, "*"},
  tuple_list           = {tuple, "*"},
  arguments            = {entity_val, "*"},
                     declaration of required reply type
  type_desc            = {{msg_type=INT}, {class_type=STRING}, "?"},
                              roset messages
  roset                = {roid, feature, roset_arg},
     roset_arg         = one_of{entity_val, roset_options},
        roset_options  = {roset_option, "+"},
           roset_option = one_of{append, replace, insert, delete},
              append   = {entity_val},
              replace  = {{index=INT}, entity_val},
              insert   = {{index=INT}, entity_val},
              delete   = {{index=INT}},
                              query messages
  query                = {roid, feature, type_desc, query_option, "?"},
     query_option      = one_of{length, item, search},
        length         = {},
        item           = {{index=INT}},
        search         = subtype_of{entity_val},
        searchall      = subtype_of{entity_val},
  query_bin            = {roid, feature, {host=STRING}, {port=INT}, arguments,"?"},
                              object creation
  create               = {{class=STRING}, creatorcall, "?"},
     creatorcall       = {{creator=feature}, arguments, "?"},
                             method invocation
  execute              = {roid, feature, type_desc, arguments, "?"},
                 (the folloging message will be introduced in Sect. 8.2.3)
  tool_execute         = {roid, feature, arguments, "?"},
                            change propagation
  changed              = {roid, feature, changed_arg},
     changed_arg       = one_of{entity_val, no_change, changed_options},
        no_change      = one_of_const{{nochange='<no change>'} },
        changed_options = {changed_option, "+"},
           changed_option = one_of{appended, replace, insert, delete},
              appended = {{index=INT}, entity_val},
                      user communication (see Chap. 9)
  inform               = {{receiver=roid}, {msg=STRING}, {obj=roid}},
}
```

Figure 7.4: Grammar MSG: message types in PIROL.

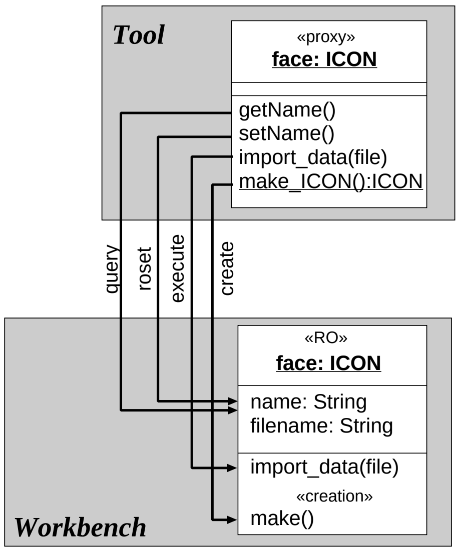


Figure 7.5: Messages in PIROL (1)

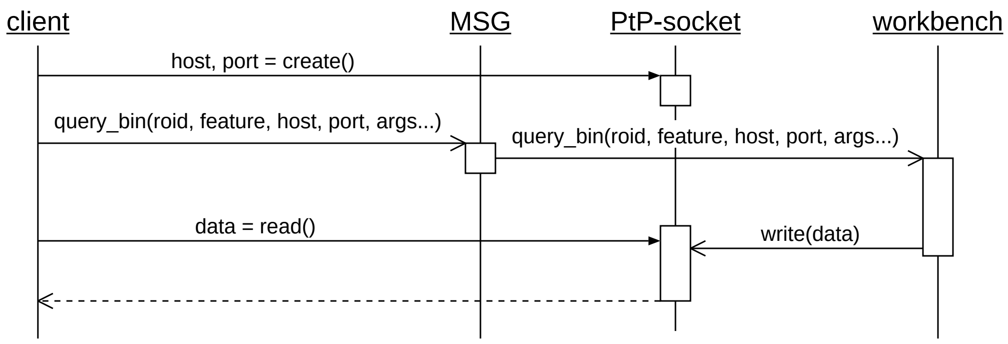

client
MSG
PtP-socket
workbench
host, port = create()
query_bin(roid, feature, host, port, args...)
query_bin(roid, feature, host, port, args...)
data = read()
write(data)

Figure 7.6: Reading data through a point-to-point socket

Point-to-point communication

All messages considered so fare are mediated by the MSG server. For certain situations point-to-point connections can be traded, through which large blocks of data can be transfered efficiently. The experience of the ESPRESS project[2] showed that large binary data should not be sent through the message server. In its adaptation for ESPRESS the PIROL repository had to store units of specification that were written in μSZ and encoded as parsed and serialized Pizza structures [BGHHm98]. These units typically had a binary size of several hundred kilobytes, which are not efficiently handled by MSG. Fig. 7.6 and Fig. 7.7 show the protocol for transferring binary data. Although Java-serialization of Pizza structures is no longer used in PIROL, serialized terms which replace this

[2]Between 1995 and 1998 a variant of PIROL was used within the ESPRESS project aimed at methods and tools for embedded system development using Z and Statecharts. Within this project special emphasis was on tools for analyzing a model in the combined specification formalism μSZ based on a common abstract syntax representation in Pizza[OW97].

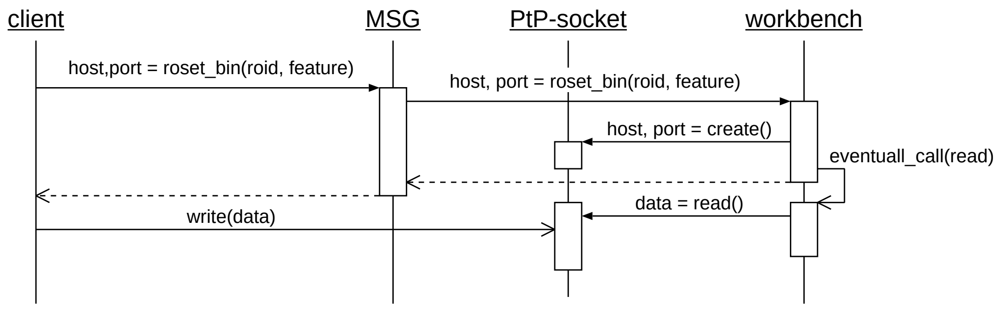


Figure 7.7: Writing data through a point-to-point socket

mechanism now may be treated in the same way. See below (Sect. 7.4.3) for details.

Security

Introducing inter process communication (IPC) into PIROL also raises the issue of security. Any user who has access to an MSG channel is potentially able to perform arbitrary operations in PIROL. Thus, connecting to MSG had to be secured using a cryptological protocol of authentication. For this purpose the open–source library md5–crypt is integrated into MSG and any client that wants to connect to the channel must prove knowledge of a shared secret. This is the exact protocol:

- The MSG server creates a lock-file with restrictive file permissions. This file stores the host and port information that is needed to connect to the server socket. Also, a generated "cookie" is stored in this file. As this file is only readable within the Unix account that started the MSG server, PIROL's security is founded on the security of Unix accounts. Note, that this always includes unlimited permission for Unix user "root".

- A client that wants to connect to the MSG channel, first reads this lock-file, from which it takes the information of how to connect to the server socket.

- In reply to the `connect` request, the MSG server sends a challenge to the client: it is a random word, that shall be used as "salt" for encrypting the "cookie" in the lock-file, thus proving knowledge of this shared secret. Note, that the "cookie" is never sent across the socket. It is important that this challenge is a random word, so that listening at the socket for the authentication of another client does not help an intruder, because he will always be challenged to encrypt the cookie with a different "salt".

- Only if the client is able to correctly encrypt the cookie, it is admitted to the channel. It is granted a credit of 100 points. Each message sent to the server decrements this credit. If the client runs low in credit, it is challenged again. If it fails to authenticate again the connection is closed by the server.

Note, that this protocol is not really dependent on any specific infrastructure aside from an internet connection between client and server. The lock-file may indeed be transferred using any secure protocol like `scp`. The assignment of limited credits should confine the damage incurred by an intruder that by some means "stole" an existing socket connection.

Minor fixes to MSG

Among the many minor fixes that were needed to use MSG for PIROL in the intended way, one issue relates to the sequence of processing. When de-registering a message pattern all resources associated with this pattern are freed. Unfortunately, also those arguments are part of the resources that are passed to the client's callback. This had the unpleasant effect, that a callback that de-registers its own pattern inadvertently also freed the arguments that it still needed for operation. This is solved by delaying the actual freeing of an active pattern until after its callback terminated. Of course, this kind of problem only occurs in a programming language with manual memory management. In PIROL this holds only for the libraries for MSG and term types (garbage collection of terms has been discussed in Sect. 6.2.3) plus different modules for integration into Lua and Java, which all are implemented in C. Other modules that are implemented in C are H–PCTE and Lua. These are, however, considered black boxes whose memory management is not a concern of PIROL.

7.4 Client-Server Architecture Interacting with other Concerns

7.4.1 Meta modeling

Discussion ▽

At first sight meta modeling and the client-server architecture seem to be orthogonal. Objects are defined by the meta model and the architecture provides mechanisms how these objects can be handled by different OS processes possibly on different nodes. Considering the impact of the file/object alternative, we find that granularity again becomes an issue. While file based environments are forced to exchange their data as complete files, a repository based environment is enabled to exchange much *smaller increments*. Through meta modeling it is possible to exchange and modify single attributes of ROs. If this capability is to be exploited by the environment the mechanisms for distribution must support the same granularity, too. PIROL provides three distinct levels of granularity: documents, objects/attributes, and term values. Here we analyze how the object/attribute level is made available across message based communication. This is the central level for meta modeling. The term level will be discussed below.

This conflict motivated the introduction of COs [←1.1.1]

Distribution affects handling of term values [7.4.3→]

Proxy classes

The message types introduced above support manipulation of data at the attribute level. These messages are encapsulated by language specific libraries of

Proxy classes [←p.87]

proxy classes. Through these proxy classes distribution can indeed be hidden to a large extent, such that tool implementors can make use of the whole meta model without great effort concerning the remote access to ROs.

New Feature ▽

The general presentation of proxy classes above omitted the details of creation in the presence of proxies. *Creation methods* of Lua/P have no direct equivalent in languages like Java, that do not support named constructors. For this reason, each Lua/P creation method is encapsulated by a static proxy method, that maps to a `create` message. In Java the name of this static method is by convention prepended by the class name, because Lua/P allows to redefine creation methods in subclasses, which cannot be reflected by Java's static methods. The return type of these methods would have to be redefined in order to return a properly typed proxy object for the newly created RO. Such redefinitions, although perfectly type safe, are not permitted in Java.

Object creation and referential integrity [←6.2.2]

Sect. 6.2.2 has argued that for object creation the preferred choice should be to always create an object along with a link that connects it to another object. The above technique of object creation via static methods of proxy classes does not work for such linked creation. Because the function `LinkedNew` could not satisfactorily be rebuilt in arbitrary programming languages, the creation of isolated objects had to be supported by Lua/P which is implemented using the "orphanage" `dangling_objects`.

Managing newly created objects [← 6.2.2 on page 74]

Creation of proxy objects for *existing* ROs runs under a different regime. In this case the RO must be known by its *ROID* — short for *repository object identifier* — which will be passed to the constructor. However, care must be taken, not to introduce inconsistency through unwanted duplication. To avoid this problem, the proxy library has to provide a caching mechanism, that stores all proxies that have already been created in the tool. Obtaining a proxy object must always go through this cache, that — according to the Flyweight-Factory pattern [GHJV95] — first looks for an existing proxy for the given ROID and only in case of a cache miss creates a new proxy and inserts it into the cache.

Reflection and interoperability

The first programming language that was connected to the workbench via MSG was Lua. In that case no explicit proxy classes were needed, because Lua's reflective mechanisms are powerful enough to easily create proxy classes on the fly, or rather, to use a proxy meta class plus tables of type information to map access to a proxy object to the appropriate messages.

Similarly, the workbench does not need wrapper classes that explicitly handle data conversion for access via MSG, but these conversions are implemented generically based on the reflexive information that is available an all levels: PCTE, Lua/P and term encoded messages.

A similar technique is used in the low level Java binding of MSG: values passed from Java to MSG are inspected regarding their Java types and accordingly encoded using terms. This low level binding uses untyped object arrays for method arguments. This level is encapsulated by proxy classes for type safe usage.

7.4.2 Persistence

Implementation ▽

Having tools access the repository on the level of single attributes helps to make persistence transparent for tools. No dirty flags have to be kept in order to write changes back to the repository, because writing is immediate for each request. Only one issue remains: within the workbench ROs are referenced by H–PCTE object handles. This is not possible for tools which have no direct access to the repository. Here a different technique has to be employed to refer to ROs. In fact tools should be able to use native references of their programming language, i.e., references to their proxy objects. Thus only the messaging layer is responsible for *marshalling* RO references and translating them into proxy references.

When marshalling an RO–reference, unique object identifiers are used: *ROIDs*. Following the discussion in Sect. 6.2.3, usage of object identifiers has to take care that such externalized references stay consistent with the repository contents.

Integrity of externalized references [7.4.6 on page 98→]

The central property of OIDs, their uniqueness, is already guaranteed by PCTE, which assigns an `exact_identifier` to each object[3]. Since this identifier is a plain string it can be re-used for the purpose at hand. Within PIROL this identifier implements the concept of ROIDs.

The PCTE specification defines that an exact identifier consists of a prefix denoting the PCTE installation and a postfix uniquely identifying an object within the installation. No further assumptions should be made. H–PCTE, however, uses a compound postfix consisting of a segment identifier and a number within the segment. Using this information it is easy to retrieve an object given only its ROID by simply parsing the ROID followed by a lookup within the segment's directory of objects.

Segments [←2.2.1]

This strategy for object lookup is, however, a possible slowdown for the environment, since lookup in the — possibly huge — segment directory is involved. For most cases this can be avoided by keeping a cache (implemented as a hashtable) that maps ROIDs to RO handles in the workbench. ROIDs are used mainly on the tool–to–workbench communication channel. Tools only operate on ROIDs that have previously been passed to them by the workbench, so each ROID used on the channel will already be present in the mentioned cache when needed.

In contrast, references to transient objects which do not have an `exact_identifier` assigned by PCTE are marshaled using an identifier that is only valid within the enclosing workbench session. Restrictions apply to the visibility of *transient attributes*: for these attributes Lua types are permitted that do not have a mapping to either PCTE or proxy objects. These attributes are not directly visible to tools. Other than that, transientness of data is of no importance for the techniques presented in this chapter.

Transient data [←2.2.1]

7.4.3 Granularity

Optimization ▽

The discussion about granularity motivated the introduction of term types into

[3]Note, that such uniqueness is not guaranteed for object handles at the PCTE API: different handles may refer to the same object and a handle (which is an opaque pointer) can be re-used for different objects over time (cf. Sect. 4.1.1 on page 56).

the type system of Lua/P. In PIROL, serialization has first been implemented for terms. For this reason transmitting terms via MSG is straight forward. It should only be noted that for terms no externalized identifiers are used, i.e., terms are passed across the message channel by value semantics, not reference semantics, which is well in line with how terms are treated by the repository.

Two optimizations are possible: the workbench retrieves term values from the repository by de-serializing the packed form that is stored. Using the protocol for binary data given in Fig. 7.6 and Fig. 7.7 the complete serialized term can directly be piped through a socket. This way the data flow bypasses MSG and the workbench and avoids the unnecessarily repeated de-serialization and serialization. Secondly, it should be possible to modify terms incrementally across the message channel. The latter is, however, not yet implemented.

No general solution can of course be given on how to map term types to different programming languages, i.e., how to encapsulate terms in proxy classes. The Java mapping uses Pizza [OW97], because Pizza's algebraic types are built on the same conceptual background as term types. For tool programmers these Pizza classes are convenient, even if the tool is actually implemented in plain Java, because Java classes can without problems access Pizza classes. The construction of converters between term values and Pizza object structures, however, is not yet automated and thus quite tedious.

→ Sect. 14.1.2 will discuss optimized queries for lists.

A third type constructor besides classes and term types exists, that should also be considered when mapping data structures for message based communication: `List`. In the context of communicating components, the question has to be answered at which level of granularity lists are send across the channel. The decision in this regard is partly left to the programmer.

New Feature ▽

Grammar of message types [←Fig. 7.4]

Note, how the set of message types has been constructed in alignment with Lua/P. Let's have a look on the details of adapting MSG messages for Lists. Firstly, the primary list *methods* are exported to the message channel by means of special message types. So the term types `length`, `item` and `search` are subtypes of the message field `query_option`. Similarly, `append`, `replace`, `insert` and `delete` are subtypes of `roset_option`. While `query` naturally only supports one option at a time, `roset` can in fact perform many modifications in one step, which is why `roset_options` is defined as a list of `roset_option`. Finally, a list can also be retrieved in a single step if no query option is given. In this case the whole list will be encoded as a term. Note, that objects contained in a list are only transfered by reference, i.e., an object list is encoded by a list of ROIDs.

7.4.4 Behavior

New Feature ▽

The three–tier architecture does not impose new requirements on Lua/P, which implements the services that can be accessed by clients. We have introduced proxy classes without saying who produces them. Class `ROCM_CLASS` has been presented in Sect. 4.2 as being responsible for compiling schema definitions from their RO–representation to PCTE schemas. A very similar mechanism is used to generate proxy classes from the same ROs of type `ROCM_CLASS`. The only difference is now, that proxy generation is language dependent and thus cannot be seen as intrinsic responsibility of these ROs. Instead separate Lua/P scripts are provided that traverse the RO structure producing a proxy class for each `ROCM_CLASS` encountered. The script for generating Java proxies is called `proxygen`. △

Through proxy classes tools can exploit the behavior implemented in RO classes.

7.4.5 Exception handling

New Feature ▽

As we have seen in Chap. 5, Lua/P has the capability of throwing classified errors or exceptions. It lacks, however, facilities to catch these errors. So, what can be done to prevent an exception from terminating the application, i.e., the workbench? With respect to the client-server architecture this can now be answered as follows. The decision whether a specific error can be repaired, or whether it should be reported to the user, or whether it can be ignored completely can usually be made only in the implementation of a tool, because only there knowledge about the *intention* of a failed operation is available. And only a tool can effectively report errors to the user. For these reasons all errors are propagated to the tool that initiated the failing operation. At the same time, each request from a client to the workbench is identified as a *transaction* that can either be executed completely or will be rolled back to the previous stable state. Among several housekeeping actions, a successful request commits all list attributes that are encoded as terms, i.e., lists for which incremental modifications are not instantly persistent.

Lists encoded as term [←3.2.2]

Exceptional control flow

Implementation ▽

Implementing error propagation involved intervention in the fundamentals of the different runtime systems. In order to obtain a good combination of convenience and performance a set of callbacks in different directions co-operate for catching and reporting common errors, especially access errors due to insufficient permissions. Fig. 7.8 shows the somewhat complex sequence for a request that is aborted due to a permission error.

The diagram is divided into a C part (left hand side) and a Lua part (right hand side). The actual pay-load of this sequence is encoded in the call chain 1.1⟶1.1.2⟶1.1.2.1 (colored in blue). Dispatched from the MSG interface, a call to the Lua part of the workbench is issued. Before calling the critical PCTE function, a temporary error function is installed in the module `error.lua` (1.1.1). When the PCTE interface detects an access error, it triggers the error function that had been deposited (1.1.2.2.1). The additional indirection of `my_func` and `pirol_error` is introduced for performance reasons: `my_func` assembles an error message (`txt`) and deposits this in `error.lua`. This assembly should only happen in the error case because it requires some repository access and is itself time consuming. Without this consideration, error code and message could have been set at stage 1.1.1 already. At the time that `trigger_error` terminates, the precise code and message are deposited in `error.lua`. Now a call to `lua_error` breaks the conventional control flow, terminates all active

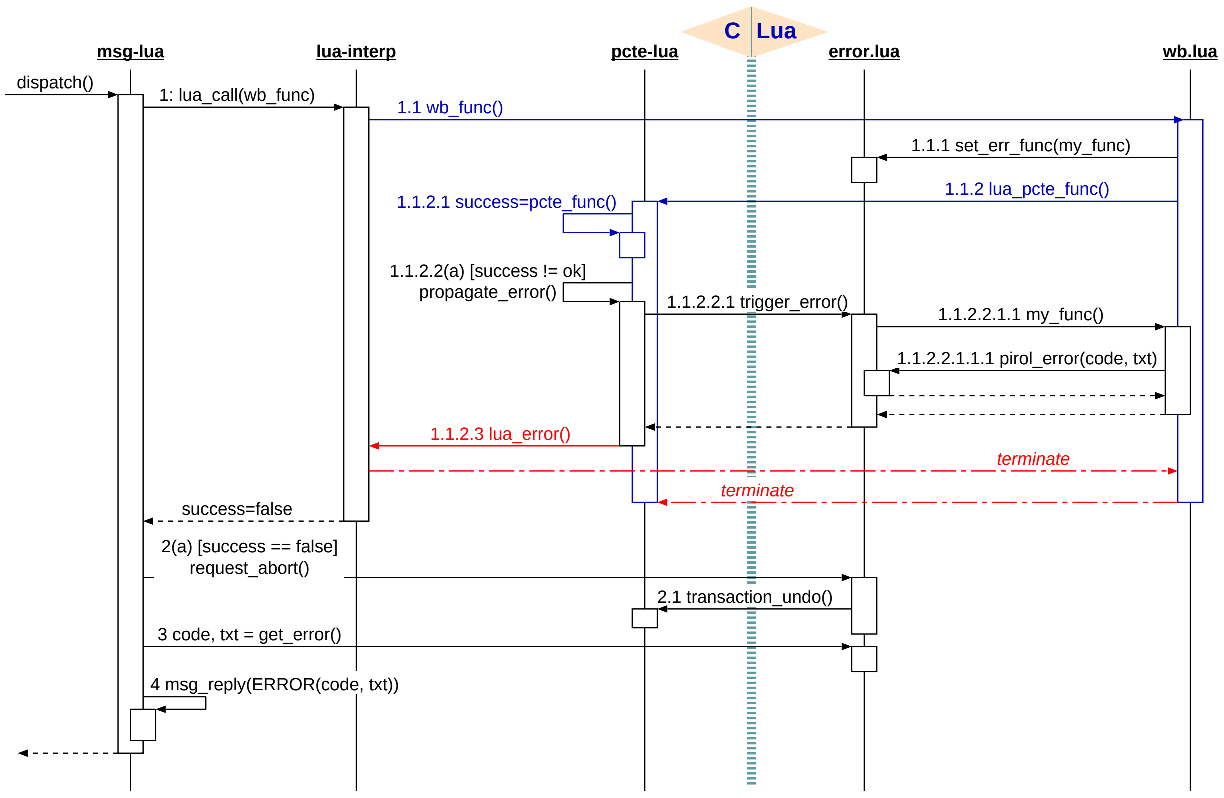

C
Lua
msg-lua
lua-interp
pcte-lua
error.lua
wb.lua
dispatch()
1: lua_call(wb_func)
1.1 wb_func()
1.1.1 set_err_func(my_func)
1.1.2 lua_pcte_func()
1.1.2.1 success=pcte_func()
1.1.2.2(a) [success != ok] propagate_error()
1.1.2.2.1 trigger_error()
1.1.2.2.1.1 my_func()
1.1.2.2.1.1.1 pirol_error(code, txt)
1.1.2.3 lua_error()
terminate
terminate
success=false
2(a) [success == false] request_abort()
2.1 transaction_undo()
3 code, txt = get_error()
4 msg_reply(ERROR(code, txt))

Figure 7.8: Control flow of aborting a request

procedures below `lua_call` (1) and returns a failure code. The MSG dispatching code first tells the error module to abort the current request. Then the stored error code and message are retrieved and an `ERROR` reply is assembled from this information and sent to the initiating MSG client.

Discussion ▽

Complexity stems from the fact that function `pcte_func` must throw an error from the third nesting level. This function cannot perform this on its own. It needs information from `wb.lua` which should not be computed in advance (for the given performance reasons). Also several levels of active procedures from both languages, C and Lua, have to be terminated prematurely. Finally, the `msg-lua` module must report the error to the client who issued the request.

The performance of this scheme has already been mentioned. The standard control flow of successful operations has only the very small penalty of call 1.1.1. Everything else not directly related to the pay-load function only happens in the error case. Modularity is also observed. Care has been taken to keep the MSG and PCTE interfaces completely decoupled. No direct dependencies exist. Lua functions `my_func` and also `request_abort` (see step 2(a)) are registered as callbacks. Thus this dependency is created only at run-time. Only `trigger_error` and `get_error` are currently hard-coded into the caller side.

The additional burden of exception handling is mostly done in a generic way. E.g., calls to `propagate_error` (1.1.2.1) are generated into the code by a macro that encapsulates all calls to the PCTE API. Only installing the context specific error reporting function (`my_func`) happens explicitly in client code (here: `wb.lua`). This effort cannot reasonably be reduced. Perhaps the style of error handling is a little bit unusual, but in fact the resulting coding convention comes very close to the `try-catch` construct of Java, which has no direct correspondence in Lua. The mentioned deficiency of `try-catch` is, however, avoided because exception catching only prevents the workbench from terminating while propagating the condition to the tool and providing transactional protection.

Aborting a transaction

In the above description a call to `request_abort` is mentioned without telling about the effect of the function. For the moment, two tasks can be identified.

- The current PCTE transaction is rolled back. This puts the repository into the previous stable state with all issues of data integrity observed.
- Some information that is cached within the workbench must be invalidated. This concerns all lists that have been modified within the transaction that is being rolled back.

List caching [←2.2.1]

Propagating exceptions to a tool

As an example for tool implementations, the Java library for PIROL tools uses a small layer of C code (embedded as native methods) for sending messages via MSG. These wrapper functions analyze the reply value, and if an `ERROR` term is detected, a Java `Exception` object is created and thrown. For this reason, every error code that is used by Lua/P must be mapped to a specific exception class in Java, all of which are subclasses of `PirolException`. This common superclass enables tools to catch all these errors using only one `catch` clause, while leaving the option to handle different exceptions separately. It is interesting, that the C code can thus produce exception conditions, which are passed up through the Java stack, such that the tool implementation doesn't even see that the exception was caused outside the tool. In other words, proxy objects are transparent to exceptions in that they don't behave differently than corresponding local objects would.

The same Java library also provides the facility of installing *exception handlers*. Each exception that is to be thrown at the MSG interface is first redirected to any installed handlers. Only if no handler successfully handled the exception it is thrown to the client code. This facility is typically used such that (a) every exception is reported in a tool's status line and (b) all `AccessExceptions` are further-on ignored, because this is in fact a quite "normal" situation in a distributed, multi user environment. The gain of this facility is the possibility to change reaction to all exceptions passed from MSG in one single location.[4]

Access control [9→]

[4]Interception of all PIROL exceptions was facilitated by the generic C layer, which dispatches all communication between Java and MSG.

The workbench finally doesn't need to catch any errors, because each operation performed by the workbench is carried out on behalf of a request from a tool. Note, that also Lua/P scripts are executed by a request that is issued by a really tiny tool: a script launcher. Now each request is a bounded piece of execution, which may succeed or fail. A failing request, however, does by no means force the workbench to terminate.

7.4.6 Integrity

Semantical integrity

Discussion ▽

The issue of data integrity is in part motivated by the presence of exceptions. By identifying *client requests* and *workbench transactions* data integrity is indeed ensured at the desired level. Each workbench session runs in a transaction, that is only committed when the workbench is idle for several seconds. The exact value is a parameter for optimization, because committing a transaction takes some time. Within each transaction every request coming in from a tool sets a *checkpoint*. Before a transaction is committed *roll-back* can go back to any previous checkpoint. More specifically, a failing request will roll back to the last checkpoint, before propagating the exception to the tool. The intention behind this is to never leave the repository in an inconsistent state. Failing request have no effect on the repository.

Aside from transactional issues, where distribution in fact helps to clarify the concepts for data integrity, distribution puts special emphasis on *transparency*. This must include all mechanisms for safe guarding data integrity. For attribute guards the mechanism of exception handling already mediates such that exceptions triggered by a guard violation cause a roll–back of the enclosing request and signal the error condition back to the initiating tool. Everything beyond this point is entirely within the tool's responsibility.

Attribute guards [←6.1.2]

Referential integrity

Externalizing references by means of ROIDs endangers referential integrity because PCTE does not know whether a tool still has a proxy for an RO that is to be deleted. Therefore, the workbench must keep PCTE links either to all objects that are referenced by tools, or all deleting operations must be wrapped, to check for objects that are possibly affected by deleting a link and then checking for externalized references to these objects. The first approach is a lot easier to implement, but affects each passing of a ROID to a tool, which might affect performance. Fortunately, this approach benefits from the ROID–RO cache mentioned in Sect. 7.4.2. The additional PCTE links — called *keep–alive links*— are simply duplications of cache entries. This means that (a) no additional computation is needed, (b) only after a cache miss a link has to be created and (c) garbage collection for cache and keep–alive links coincides.

For this multi–level *garbage collection* two strategies exist. Both strategies need to record a qualified variant of reference counts, where for each ROID–RO entry a list of tools is maintained that currently use the ROID. This relies on the capability to determine the sender of the current MSG request. Using this information and a mapping from MSG clients to tools the workbench records all tool–ROID dependencies whenever it returns a ROID to a tool.

Within the first strategy, only terminating a tool triggers garbage collection. This will delete all usage marks of the terminating tool and collect all entries with no further tools depending on them. As a refinement, the second strategy uses the tool's garbage collector as an additional trigger. Each proxy object that is being collected triggers a `collect` notification for the workbench, which then deletes the usage mark and possibly the ROID–RO entry. Note, that employing the tool's garbage collector for proxies, needs to ensure, that the proxy cache mentioned in 7.4.1 does not prevent collection of proxies. In some languages including Lua and Java this can be done using *weak references* for this cache. Currently, only the first strategy is implemented, i.e., garbage collection happens only on tool termination.[5]

Last but not least, terminating and starting the workbench process completely wipes the list of keep–alive links, since at both points in time no tool is running in the workbench's session. Using both events makes the system more robust against crashes of the workbench process.

Note, that all this care about externalized references is a direct result of decoupling tools from the repository in so far as no tool is a direct repository client, but tools communicate with the repository only via MSG and the workbench.

Summarizing referential integrity, three kinds of object references exist beyond the scope of PCTE: term–to–RO, HTML–to–RO, and ROID–to–RO (for externalized references). All of these are backed–up by keep–alive links, that prevent object destruction while an RO is still referenced. By the example of proxy objects it has been shown, how garbage collection cross-cuts many architectural layers: proxy objects, which could be cached using weak references only, are subject to the tool's garbage collector. A finalization method of proxies (or any comparable hook into the garbage collector) notifies the workbench about this collection. If no other tool uses the ROID the workbench may then delete the ROID–RO cache entry. If this actually frees the RO handle — i.e. the workbench keeps no further (private) reference to the RO — the Lua/P garbage collector may collect the handle and finally propagates this to H–PCTE. If the keep-alive link to the RO was in fact the last link to this object it is finally deleted in the repository.

Indirect references [←6.2.3]

Change propagation for consistent views

This section would be incomplete without at least mentioning another issues that arises because tools share data via the workbench: how are the views of different tools kept consistent? The full discussion about consistent views will be postponed until Chap. 8 introduced additional communication options.

Sect. 8.2.2 shows how this is solved using change propagation.

[5]This is further complicated by efforts to share one JVM for several tools. In that case, tool termination will go unnoticed, since there is no OS-process terminating before the last tool closes this shared JVM.

7.5 Summary

From a client perspective, the client-server architecture is — by and large — transparent. The implementation of this architecture cross-cuts many other concerns but this extension is quite straightforward giving hints at the near orthogonality between concerns. This chapter referred to the following techniques:

MSG	PIROL's communication facility.
Terms	Marshalling messages and their parameters.
Term-API	Encapsulation of Terms for the sake of interoperability between Lua, C/C++ and Java (so far).
Proxies	Providing transparent access to remote objects.

Transparency is achieved by means of

- the design of the concrete message protocol, closely aligned with Lua/P,
- fine grained, automatic repository updates,
- extending exceptions and garbage collection (safe guarding referential integrity) across the boundaries of OS processes.

This architecture also facilitates a due integrity issue: having explicit requests gives a reasonable boundary for transaction rollback in case of an exception.

The presentation in this chapter was artificially simplified, because the architecture was from its very beginning designed with support for a close control integration, which will now be presented in the following chapter.

Chapter 8

Control Integration

Through *separation of concerns* the PIROL system is split into separate *modules* or components, devised as clients (tools) and a server (workbench). From what has been said so far, tools may share data through the repository, but they would still appear as isolated tools, not an integrated environment, if we had no means to have tools *cooperate* and "speak" to each other. Only by further mechanisms for integration these separated modules are composed into a *system*. In the early 1990s, research in SEEs has performed different efforts of partitioning the design space for integration [WF91].

8.1 Integration

Environment integration is a multi-faceted issue. First, we will discuss different dimensions of integration, answering the question what (conceptually) shall be integrated. Secondly, we will briefly classify perspectives by asking, who is performing integration. Also the physical entities to be integrated need to be considered. We will divide the issue into pairwise relations between components.

8.1.1 Dimensions of integration

Related Work
▽

Early versions of the ECMA reference model for frameworks of software engineering environments (cf. [ECM93]) manifested an agreement on three classes of integration, *data*, *control*, and *presentation integration*.

We have already seen the answer to *data integration*. Meta modeling and persistence allow tools to operate on shared data rather than each tool keeping its data privately. The requirement of language independence adds to this issue, as now mechanisms for type level interoperability are needed that convert values between their representations in various programming languages [WWRT90]. In PIROL this interoperability is achieved using terms and their mapping to specific programming languages (marshalling and demarshalling).

Marshalling
[←7.3.2, 7.4.2]

Control integration is the major topic of this chapter: how can independently developed tools communicate? This presumes that a communication channel is made available and protocols are defined governing the communication on that channel. Given the channel and its protocol, the system can have a global control flow, that is not forced to stop at a tool's boundary.

Presentation integration lies at the border of technology and design. Wasserman [Was89] identifies four levels of presentation integration: window system (like X11), window manager (like mwm, the Motif window manager), windowing toolkit (like Motif) and guidelines for a specific *look&feels*. Only all four levels in concert provide for consistency across components. In this area, technology does not suffice, but only tools developed along a common style, using common metaphors and conventions allow the user to forget about their different origin and nature. Chap. 11 will present some ideas how also architectural decisions can contribute to a handling of tools that levels the boundaries between tools.

Common services [11→]

In addition to these three non-controversial issues, Wasserman mentioned two more dimensions: platform integration and process integration [Was89]. Of these, platform integration dealt with issues of operating systems and hardware platforms. This issue was dropped by later works. The other dimension — *process integration* — was, however, subject of many projects and publications. The 3^{rd} edition of the ECMA reference model [ECM93] adopted process integration and finally introduced *framework integration* as the fifth dimension. Note, that these areas do not directly relate to integration *mechanisms*, but process integration is seen "on top" other services, while framework integration mainly defines implementation constraints on integration mechanisms. For such reasons [WF91] differentiate mechanisms (data, control, presentation) from integration at the levels of end-user-services and process.

Process integration is closest to the *domain* at hand. Software development takes place mainly in the dimensions of the *product* being developed and the development *process*. In PIROL this observation is already reflected by corresponding packages of the meta model.

ROCM package PROCESSES [←4.1.1]

Framework integration regards the interface that the framework provides to tools and postulates that also administrative tasks of an SEE should be supported by the framework. In other words, the framework should provide services and facilities for its own configuration such that administrative tools can be constructed and used much alike regular tools. The effect is a reflective architecture of the framework. Examples for tasks in this category are installation and configuration of accounts, user groups, and tools. How framework integration is covered in PIROL will be discussed in Sect. 8.3.1 and Chap. 11.

8.1.2 Roles of developers

The focus on integration suggests to differentiate roles and activities in the process of constructing an SEE, as each role defines a different perspective on the issue of integration. The ECMA reference model identifies these roles:

1. Platform suppliers
2. Environment suppliers
3. Tool suppliers
4. Users

5. Environment adaptors
6. Process developers

Roles (1)–(3) can be seen as suppliers of components in a layered architecture. In the case of SEEs, also users (4) are developers, so in our context "user" will hardly ever refer to end–users, but to users of the SEE. Roles (5) and (6) are not very well defined in the RM, but hint at the growing importance of system assembly with its issues of adapting components and the framework, deploying components, and setting–up relationships between components, i.e., programming interaction between components. All these are roles and activities that are fully developed only in the field of component technology [Szy98] ("*develop-package-assemble-deploy software*") of which early works in SEEs are definitely influential precursors.

8.1.3 Pairwise integration

Thomas and Nejmeh emphasize that integration can only be fully understood, if we look at specific pairs of entities that are being integrated [TN92]. The following relationships can be identified within an architecture like PIROL's:

1. tool–to–workbench integration
2. workbench–to–workbench integration
3. workbench–to–repository integration
4. tool–to–repository integration
5. tool–to–tool integration
6. tool–to–user integration
7. user–to–user integration

Of these relationships, only (1)–(3) are implemented directly, while (4) and (5) are mediated by the workbench and (6) is mainly left to the individual tools except for a few common services. Finally, (7) will be a topic of the next chapter.

Context menus [11.2→]

Multi user capabilities [9→]

As we have said, tool or component integration is a multi-faceted issue. A good overview on the different conceptual models can be found in [Sta99], which mainly compares the works by Wasserman [Was89], Thomas and Nejmeh [TN92], and Brown et al [BFW91].

8.2 Elements of Control Integration

After data integration has been presented by previous chapters, this chapter presents the technology responsible for connecting tools in terms of a common control flow and shows how some details can be encapsulated by additional logical layers and concepts.

Related Work
▽

Control integration is the second core dimension of tool integration according to [ECM93] and [Was89]. At the time of Wasserman's writing, he could identify two alternatives for control integration: Hewlett Packard's Broadcast Messages Server (BMS)[Cag90] and messages services integrated with data management systems such as PCTE and Software Backplane [Pas89].

At the time the development of PIROL started, early versions of CORBA [OMG97] were available, but we decided not to use it, because CORBA seemed too heavy-weight. The requirements of PIROL were limited to different styles of passing messages. CORBA's focus on brokering requests to components that could reside on arbitrary nodes was not needed by PIROL. Instead, PIROL uses a modified version of the messaging facility MSG [Rei90], which granted great freedom in the development of PIROL. From todays perspective, and after having learned so much during the development of PIROL, a re-implementation of PIROL using todays improved CORBA-technology would certainly be rewarding.

Desktop integration [11.4→]

Also approaches for constructing desktop environments should be considered. While CDE's message passing is based on ToolTalk[Sun93] and HP's Encapsulator[Fro89], KDE uses a specialized lightweight facility called DCOP [Tib00].

△

Introduction of MSG [←7.3]

For the given reasons, PIROL implements control integration using MSG, which has already been briefly presented in the previous chapter. For the full picture of control integration a more in depth look at the architecture and mechanisms of MSG is required.

This is the architecture of MSG: a message server runs as a background process listening at certain sockets for

- clients to connect to the channel
- registered clients to send messages to the channel.

8.2.1 Multicast communication

Using MSG, Communication happens in a *multicast* style [Bro92, Rei90]: clients announce their interest in messages by registering message patterns. Each pattern is associated with a callback function of the client. The message server then distributes each incoming message to all clients that subscribed to a pattern which matches the message. Originally, this matching was performed using string based regular expressions. Depending on the policies for registering message patterns, this mechanism can implement anything from *point-to-point communication* (using tool identifiers in messages) up-to full *broadcast* (all tools listening to the same pattern(s)).

Message patterns in PIROL

In PIROL *message patterns* are defined using terms. By each message pattern a client announces interest in a set of messages. For this purpose, terms may contain *wildcards* which are (sub-)terms that have a type but no value. Two kinds of wildcards are supported: monomorphic and polymorphic wildcards.

When matching a term against a pattern, a *monomorphic wildcard* only matches a term of the same type, while *polymorphic wildcards* also match terms of subtypes of their defined type. Wildcards may be used both for atomic and structured terms. Each match for a wildcard will be captured and passed as an argument to the callback function. Note, that this style of dispatching based on pattern matching is equivalent to using the `t_select` feature.

Programming by pattern matching [←4.1.3]

Pattern matching may become critical for the system performance, although in PIROL message delivery has not yet been observed to be a bottleneck. Using wildcard terms as patterns allows quite efficient matching. While performing matching, most patterns can be excluded by just comparing the integer–encoded type of both terms. Only at the leaves of a term, strings may have to be compared, but string comparison will only test for equality. If performance degrades at a large number of patterns, by local optimization the MSG server could store all registered patterns in a tree structure where patterns of the same type are combined into one branch of the tree, such that complete branches can be excluded by just comparing the type at their root. Note, that pattern matching does not suffer from marshaling, since matching can efficiently be performed on the serialized representation of a term.

Message dispatch based on pattern matching is the only fundamental mechanism of this chapter. To blend this into the conceptual framework of all previous chapters, the following abstractions are introduced on top of the core multicast communication.

- Two new message types: `changed` and `tool_execute`,
- Representative objects and extended proxies,
- Strategies and protocols for communication.

These abstractions help to achieve various communication styles without explicitly falling back to the send/receive primitives.

8.2.2 Change propagation for consistent views

Introducing the `changed` message type serves the purpose of implementing change propagation between tools.

By data integration tools may operate on different views of shared data. We have already touched the requirement to keep the views of different tools consistent. This motivates change propagation[1] as one of the central mechanisms of PIROL. With this we mean notifications that propagate data changes performed on behalf of one tool to all other tools that depend on the same data. The resulting architecture is a modified model–view–control (MVC) style [KP88], where view and control are merged in tools while the workbench implements the model.

Consistency of data shared by different views [← 7.4.6 on page 99]

[1]Please note the different uses of this notion. In [EAMP97], e.g., the authors use reference edges between persistent entities in order to compute "change propagation" within the repository. In that setting, a tool may be required to react to a change notification by updating its *persistent* data, which is not covered by the mechanism being presented here.

History of Model View Control

Related Work ▽

Already the Arcadia architecture [TBC+88] very closely resembled the MVC style with the following correspondences. An *object management system* plays the role of a model, a *control component* executed what they call *process programss*[2]. Most interestingly, a user interface management component applies the concept of "Artists" as originally introduced by Myers [Mye83]. An Artist wraps a given model object such that operations, which are invoked by the control component, can be intercepted in order to notify a view object regarding necessary presentation updates.

The exact chain of research is difficult to re-construct, but Arcadia might have influenced the model view control style, as it has become popular through the Smalltalk-80 environment [KP88]. Also the concept of Artists using wrappers and method interception may have had impact on the later development of composition filters [BAWY95] and aspect-oriented programming.

Aspects-oriented programming [16.3.2→]

MVC in PIROL

While the original model–view–control architecture employs a bilateral observer pattern for notifications, in PIROL this mechanism is mediated by the message server. This way the workbench doesn't need to record data dependencies but simply broadcasts any data change to the message channel. Tools subscribe to notifications not directly at the subject component but at the mediating message server. For this, they register a set of message patterns and provide a callback function for each pattern. Callbacks are then invoked by all notifications that are relevant for the tool. Callbacks can be implemented generically for all incoming notifications, or individually for specific message patterns.

The effect of this mediated notification scheme is the same as in the classical observer pattern, only subscription is much more flexible with respect to detail levels.

Notifications are sent at the end of each request that is serviced by the workbench. So in terms of tool synchronization a request–notification cycle is atomic.

Integrating event sources

Integrating change observation for tools that have a graphical user interface (GUI) requires to harmonize different event sources, because GUI events are usually handled by a main loop, that can not easily be changed. For this purpose the Java library for PIROL tools uses a separate thread for listening to MSG events. Each incoming message is then wrapped in an `MSGEvent` object[3]. This event is then inserted into the system's main event queue. The usage of two threads is necessary because only so, both event sources can be observed simultaneously. Processing via a single event queue is needed because Swing

[2]Arcadia was designed as a process-centered environment, thus "the diverse software processes that users want to employ in developing and maintaining software" are defined using a process programming language [TBC+88].

[3]Oddly, this class must be a subclass of `AWTEvent`!

is in general not thread safe, classes using Swing should be called only from one thread. In addition, it is a known quirk in Java, that listening to a socket requires busy waiting, thus degrading the system performance. In contrast, integrating MSG into Tcl/Tk was a lot easier, because Tk already anticipated additional event sources that are associated with a file descriptor. The Tk main loop allows to install a file handler, that is fired, whenever data becomes available at the given file descriptor.

Trigger-deliver-react

Change propagation has successfully been split into three separate mechanisms:

Trigger The workbench detects each event that is relevant for change propagation,

Deliver MSG is responsible for delivering `changed` messages to all tools that have announced their interest by an appropriate message pattern,

React A tool registers callbacks that are free to implement any specific reaction as to update its display and internal state.

The advantage of such separation lies in separate extensibility on each of these three levels.

8.2.3 Remotely controlling tools

Representative objects

We already saw proxy objects for remotely accessing ROs thus hiding communication details from tool implementation. Fig. 8.1, which summarizes different communication patterns, gives a further example by the pair of `M: Menu` objects. Proxy objects are used by tools to access repository objects.

Proxies [←7.3.2]

For the opposite direction, RO `A: TOOL` (in the *Workbench* process) plays a role similar to proxies: in the example it provides an operation `select` which is implemented by sending an MSG request of type `tool_execute` to which the tool process responds. While such delegation is typical of proxies, RO in the workbench is already "the real object", and it only delegates selected methods, that need to be performed in the tool process, like, e.g., selecting an object for high-lighting. In order to discriminate the two styles of proxies, the latter kind will in the sequel be called *tool representative* or just *representative* object.

In [BFW91] the question whether tool process themselves are represented in the database is subsumed under "pervasiveness" of the database. This is, however, not further explicated.

Related Work

System-wide synchronization

The list of message types partly defines the communication protocol between tools and the workbench. However, the whole picture requires to also consider the behavior of both kinds of components. The mere fact that message can be

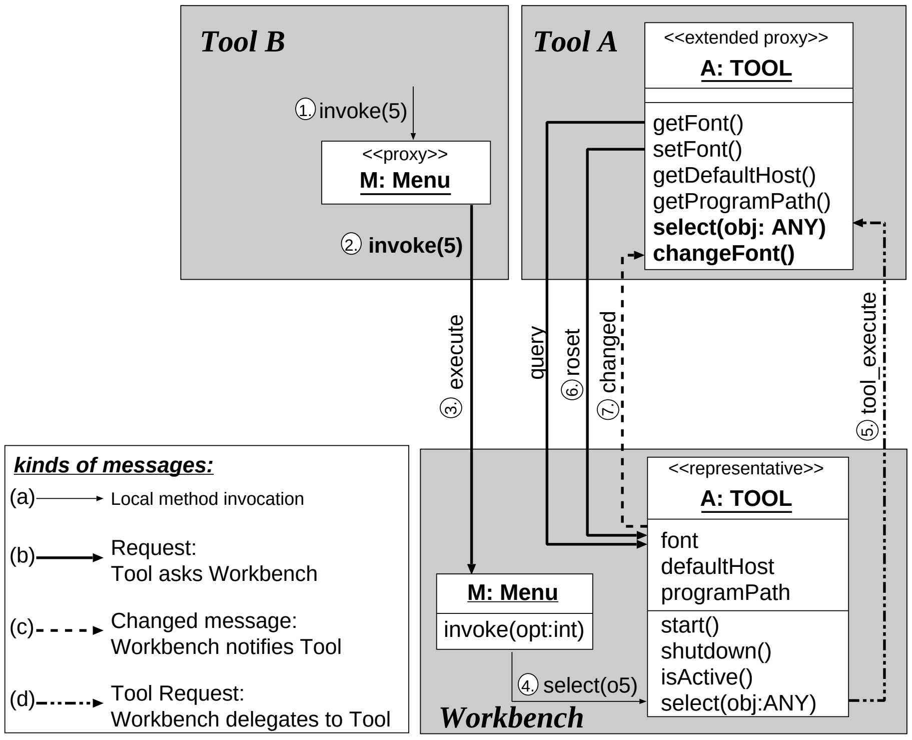


Figure 8.1: Messages in PIROL

exchanged in both directions possibly leads to various situations of dead-lock and infinite loops.

The following protocol problems have been identified:

1. Infinite Loop: A tool that issues a `roset` request usually has a pattern for observing changes of that value, that is just being modified. When the workbench broadcasts the change also the initiating tool may be notified which possibly causes the tool to redo the same change again issuing the same request over and over again. This situation will in the sequel be called *update recursion*.

2. Deadlock Workbench → Tool → Workbench: the workbench requests a service from a tool (`tool_execute` request), but that service may again require access to the workbench. The workbench currently is not re-entrant, i.e., while waiting for the tool's service to finish, the workbench is blocked.

3. Deadlock Tool → Workbench → Tool: A tool requests a method invocation from the workbench. If during this request the workbench requires a service from the same tool, the tool will still be blocked.

Experiments have been carried out to provide solutions to these problems, though it should be stated, that PIROL's protocol will remain unsafe unless a clean concept of transactions is consistently integrated.

Ad (1): a mechanism of "echo-blocking" has been introduced to MSG. By this mechanism, tools can chose, whether or not they want to receive the change notifications for any changes caused by the same tool ("echoing of updates"). For this reason the MSG server keeps explicit track of which tool initiated the current request[4], and change notifications by the workbench are marked as echo-messages, such that the MSG server can exclude the initiating tool from message delivery.[5]

A more elegant solution can be devised if a tool is designed specifically for PIROL's architecture. Such a tool may preferably decouple modifications of ROs via proxy methods and internal updates: Any action triggered from the GUI just sends a request to the workbench. Only when the workbench broadcasts the successful change, the normal callbacks for `changed` messages trigger the update within the tool. This mechanism is called *indirect modification* and implements a strict *control* → *model* → *view* call chain, where control and view happen to reside in the same component without communicating directly. This is safer than the mechanism of echo blocking because the latter will not notice if the effect of a request differs from the tool's expectation. Such differences may be caused by attribute guards or by overridden methods.

Ad (2): Every action of the workbench happens as an answer to a request of a tool. If the workbench passes the locus of control (using a `tool_execute` message) to a tool, the workbench would normally not be able to answer further requests, because it is still busy. This situation is simply solved by temporarily allowing further requests while waiting for a `tool_execute` message to return.

Actually only messages from the active tool are accepted, because these should be considered as belonging to the same control flow, while messages from other tools are still blocked. To be precise, re-entrance is not a problem. The problem is concurrency within the workbench, for which no proper mechanisms are provided.

Ad (3): This pattern has the same structure as (2). For lack of relevance it has however remained unimplemented. I.e., if this situation would occur it would in fact produce a deadlock, but no Lua/P methods that are used by tools actually send a `tool_execute` message to the same tool.

Lessons learned from these experiments. Echo blocking was difficult to implement, because several parts of the system have to collaborate for this mechanism. All of these parts in some way or other refer to tools, but all have different perceptions of tools:

- MSG assigns client numbers to tools (and to the workbench, which is just another MSG client).

[4]To be precise, there is a *set* of originators for the sake of nested requests. Currently the number of bits allocated for this set limits the number of clients to 32.

[5]An alternative solution, which is probably even more powerful and expressive, needs to introduce a notion of explicit control flows: One control flow includes all messages that are sent by clients while transitively responding the a request. Blocking would apply to all messages (except for the explicit reply) that originate from the same control flow.

- The workbench keeps as *representative* a `TOOL` RO. For illustration, see also RO `A: TOOL` in Fig. 8.1.

- Also, OS processes should be considered.

MSG had to be extended such that clients can retrieve their client number. The workbench on the other hand collects all information when a tool is started, such that the representative RO knows both the OS process ID and the MSG client ID. Depending on a tool's architecture the solution by indirect modification might be significantly more elegant.

In fact echo blocking remains an unsatisfactory work around, because it completely ignores the possibility, that a request may return unexpected results caused by an exception or by an attribute guard.

8.3 Control Integration Interacting with other Concerns

8.3.1 Meta modeling

ROCM package `TOOLS`

When introducing representatives (page 107), an RO class `TOOL` has already been mentioned. In fact, `TOOL` is just the root of an inheritance hierarchy modeling the following characteristics of tools.[6]

Definition vs. instance. Tool definitions of type `TOOL_KIND` contain configuration information for a program that is ready to launch. For the environment, a running tool is encapsulated by a tool instance RO (the representative) of class `TOOL_INSTANCE` and related.

Instance vs. container. Many tools correspond to separate programs, but in some cases several tools (separately invocable functions) are combined to one program running in one shared process. For running, a tool needs an instance of `TOOL_PROCESS` that can be re-used from a running tool or must be started anew. Starting `TOOL_PROCESS`es is the responsibility of `PROCESS_MANAGER`s. Each `TOOL_KIND` specifies, whether its instances are willing to share a process with other instances or tools.

Local vs. remote. Each `TOOL_PROCESS` can be assigned a `HOST` (or `EXTERN_HOST`, i.e., a host belonging to a different NFS), such that launching will take place on that dedicated host. This is to reflect special resource requirements of tools.

Other classes of ROCM package `TOOLS` are

`JAVA_TOOL_KIND`	Tools of this kind can be launched in a running `TOOL_PROCESS` by dynamic class loading and reflective invocation

[6]The initial structure of this package was designed during the ESPRESS project. After some evolution, [Mat02] has restructured this with respect to the kind-process-instance relationships.

	of a main method. Class and method name are stored in the `JAVA_TOOL_KIND` object.[7]
`DOCUMENT_TYPE`	A tool is usually specialized to operate on documents of a certain type.
`TOOL_POOL`	This subclass of `FOLDER` contains the tools that are available in a given session (attribute `tools` of `WORKBENCH`).
`MENU`	This implements the concept of workbench provided context menus.

Refining COs towards document types [10.3.7→]

Context menu [11.2→]

Split objects

Considering only tool-to-workbench requests, tools don't add to the semantics of ROs. Tools just provide user interfaces to ROs. Every manipulation takes place within the workbench. Using `tool_execute` requests this situation changes. Not only can the control flow be reversed, but objects can be constructed that only partially reside in the workbench.

New Feature ▽

Concerning each tool two cooperating entities exist: a repository object of type `TOOL` and the actual tool OS process. The `TOOL` RO has been introduced as the *representative* object. Consider the main object of the tool process as being responsible for the tool's functionality. Now this main object and the representative could be seen as facets of the same split object. Whenever the local object accesses its persistent configuration it uses the corresponding `TOOL`–RO via proxy methods. Whenever the workbench needs to invoke a function of the external tool object it delegates this call using `tool_execute`.

Example of extended proxy [←Fig. 8.1]

Such proxy objects are called extended proxies because they not only encapsulate the repository object but also add functionality. This addition may for example be implemented by subclassing a pure proxy class and adding the new methods in the subclass. While this could be called *remote inheritance*, it is not equivalent to ordinary inheritance, because extended proxies are not allowed to redefine methods that are already implemented in Lua/P, because such redefinitions are not known to the workbench, which would break the expected behavior of dynamic binding.

8.3.2 Persistence

At first sight, control integration and persistence appear completely decoupled. But both mechanisms in concert may in fact extend the functionality of tools. The representative object devised for encapsulating `tool_execute` messages may also be used to make internal state of a tool persistent. E.g., consider the window size by which a tool appears on the screen. Currently, the attribute `window_size` is only used for static configuration purposes, but it could easily be used for dynamic re-configuration as well: If a tool would propagate dynamic window resizing to its representative and also listens to changes of this attribute, window size would immediately become a dynamic, consistent and persistent property of a tool.

[7]Such launching happens in close cooperation with corresponding Java classes in package `pirol.tools` of the PIROL client library.

There is a simple reason, why this is currently not used in PIROL: ROs of type TOOL_PROCESS are discarded when the corresponding OS process terminates. TOOL_KINDs on the other hand can be shared by many tools, with possibly different window sizes on the screen. The above consideration suggest, to also keep individual TOOL_PROCESS objects across invocations and sessions.

Related Work ▽

Along these lines PIROL could easily imitate the concept of *perspectives* as it is provided by eclipse [Ecl]. An eclipse perspective is a pre-configured set of tools that are composed into one top-level window.

8.3.3 Granularity

Sect. 7.4.3 has shown the correspondence between granularity of the data model and the granularity as reflected by specific message types and their signatures. The same correspondence must hold for change propagation regarding structured values.

New Feature ▽

Options for list requests [←7.4.3]

The granularity of change notifications is very fine in order to reduce overhead in tools that would be caused if large pieces of data would be invalidated even if only small pieces actually changed. Generally, `changed` messages are sent in a similar fashion as `roset` requests. Normally, for lists only differences (`appended`, `replace`, `insert` or `delete`) are sent, no complete new values. For optimization many atomic list modifications may be packed into a single `changed` message. Such packing also folds different updates that refer to the same element thus minimizing the number of updates to be performed by tools.

Mapping lists of basic types [←3.2.2]

Modifications of terms are currently not supported by incremental notifications. Each assignment to a term type attribute causes the new term to be broadcast in full. Here lies some potential for optimization which is not yet pursued. List attributes that internally are encoded as terms combine both strategies: They record incremental changes in the style of `changed_options`. If any absolute setting of the list term happens (e.g., method `wipe` simply assigns an empty term), all incremental changes within the current request are canceled and only the final new value is broadcast in full.

Atomicity of a request–notification cycle is observed by folding several changes into one detailed `changed` message. The workbench thus guarantees that each request issues at most one `changed` message per attribute independent of the attribute's type.

Discussion ▽

Secondly, granularity is also an issue of designing tools. Using PIROL's style of control integration, a tool may at the same time be a complex piece of software with many user-visible features, and still be controlled in a fine grained way. This is not possible in environments, where the framework can only launch tools and wait for a result upon tool termination.

8.3.4 Behavior

Through proxy classes tools can exploit the behavior implemented in RO classes. The message protocol also allows to add behavior to the RO class model by integrating the functionality of tools. By means of tool representatives and the protocol of `execute` and `tool_execute` requests, tools may invoke each others functions.

After finding that PIROL's MSG protocol is not safe in terms of dead locks and update recursion, the obligation may be shifted to the level of defining the behavior of ROs. PIROL currently does not exhibit any protocol problems because the relevant situations simply don't occur by the way RO methods and tools are implemented. It is beyond the scope of this work to develop specification and verification techniques for this task, but work in software architecture suggests that this can indeed be done. In fact, one of the problems reported on page 108 was not discovered when working with the implementation but while specifying aspects of PIROL's protocols using the architecture description language Wright [Bil00].

Discussion ▽

8.3.5 Exception handling

Change propagation as presented in this chapter must pay attention to exception handling. Exceptions that are thrown in the workbench cause a roll-back to the previous transactional checkpoint. At that time the workbench may have already detected some change events, which it is about to broadcast to the MSG channel. Any request that is aborted by an exception must not broadcast any change event.

Transaction roll-back [←7.4.6]

Transaction abort [←7.4.5]

Control integration adds one more case to exception handling. This is, because a `tool_execute` request may also fail. For this to work, the workbench easily interprets any result of type `ERROR` and throws this error in order to terminate its active request. However, PIROL's Java client library does not yet support propagating exceptions to the workbench. Note, that this is only relevant for `tool_execute` messages, not for `changed` messages, because the latter are asynchronous messages, that do not send a reply to the workbench.

8.3.6 Integrity

Integrity largely benefits from the mechanisms presented in this paper. In fact, change propagation is a key mechanism for data consistency in PIROL. This will become even clearer, once derived attributes are seen under the light of change propagation, too.

Derived attributes

Special attention has to be paid to derived attributes. With concurrent tools it is not sufficient to query the value of a derived attribute once and continue working with this value. Another tool might cause a change of this attribute, which would cause a data inconsistency between tools. Although not stored persistently, tools should be able to treat derived attributes just like regular attributes, with the only exception that derived attributes are not directly writable.

Derived attributes [←6.1.3]

For these reasons the workbench has to calculate `changed` messages for derived attributes just like for regular ones. While executing the derivation method associated with a derived attribute, the workbench records all data dependencies, i.e., all objects that were read in order to calculate the derived value. If any of these objects is changed later-on the derivation method is evaluated again and if the calculated value differs from the previously calculated value, the new value is sent to all relevant tools by means of a regular `changed` message.

In effect, derived attributes combine the best of the two concepts 'function' and 'attribute': they are free of redundancy and also observable. With respect to change propagation, derived attributes differ from other attributes only by the way how `changed` events are *triggered*. Delivery and reaction may completely ignore the difference between regular and derived attributes.

Trigger-deliver-react [← 8.2.2 on page 107]

External semantics

Another integrity issue is the relation between a tool representative and the running tool. This can, however, be implemented using the same mechanisms as are used for other ROs. So this is more a matter of disciplined programming to inform the workbench about the internal state of a tool and also react to changes of such properties in the workbench. Additional attribute guards may be needed, so that a tool can veto a change concerning its representative, which it regards as an error. If, e.g., the `window_size` of a tool is to be set to a negative number, the tool should disallow such modification. The vetoing `GuardException` may be thrown directly in the attribute guard, or after communicating with the tool's OS process. Note that this requires an explicit `tool_execute` invocation because a simple `changed` event happens asynchronously and thus cannot be vetoed.

Such a guard for the `window_size` attribute demonstrates, how guards can be used to give meaning to certain attributes. This is not restricted to values within the repository or workbench, but also external components can be influenced by attribute guards. As another example consider the list of active tools that each `WORKBENCH` object records as running in its session. These links are setup by the creation method of `TOOL_INSTANCE`[8] and related classes. In order to keep the list and the actually running processes synchronized one could of course disallow any modification of the list of `active_tools` other than through closing the tool. The more flexible and powerful solution attaches a guard to the list, which intercepts the `remove` event in order to close the actual tool. Only if this succeeds, the object is actually removed from the list. Else, a `GuardException` is thrown. This illustrates, how, e.g., a generic browser can be used to control the component system: although the browser does not know about closing other tools, it might know about displaying object lists like `active_tools` and it might provide a remove operation on lists. This suffices to close any running tool just by removing its RO representative from the list of active tools.

[8]Class `TOOL_KIND` models all tools that are actually programs that can be started as separate OS processes. Class `TOOL_INSTANCE` represents the OS processes that are running this program (see Sect. 8.3.1 and Sect. 11.1).

8.3.7 Client-server architecture

The client server architecture implies quite restricted communication. The additional message types introduced in this chapter have only slightly extended this architecture. *Physical* communication only happens between a tool and the workbench. *Logically*, however, a tool may invoke methods of any other tool. Tool-to-tool messages are routed via the workbench, using the representative object as a reference. Also at a finer level of granularity objects within different tools and within the workbench may thus directly communicate with each other (again from a logical perspective). When comparing PIROL's architecture to modern component models, PIROL tends to provide finer grained control flows between fine grained entities. By the concept of representatives, some entities appear as components and as fine grained objects simultaneously. Both are different views of a common concept.

8.4 Summary

In an early version the contents of this chapter was integrated in the previous one (then called "A Distributed Component System"). The result was a bloated chapter about PIROL's component model with many concerns heavily tangled. That state was not satisfactory as it revealed no clear structure and was very difficult to read.

The separation into two chapters is artificial as no abstract requirements were easily extracted from the web of mechanisms. Since both chapters are very close to realizations and many of these mechanisms were part of PIROL from the very beginning, it was difficult to find a reasonable separation. It seemed justified to put "distribution" at the same level as, e.g., "persistence", but that headline seemed to entail all details of PIROL's component model.

The next attempt challenged "change propagation" as a major concern of PIROL. In fact some other concerns interact with "change propagation", which suggests those concerns to be on the same level. A reason against this attempt was the close relation between "change propagation" and "integrity". This looked like "change propagation" only became an issue, because something was interacting with "integrity".

Eventually, the most problematic tangling could be identified as relating to complex control flows in the overall system. "Change propagation" is certainly an important part in this concern, but those control flows are more fundamental. Only during the process of splitting the chapter into two, the coherent character of each new chapter appeared: Chap. 7 describes how to decouple components from a server and deals with different issues of interoperability. That defines the physical architecture. This chapter, on the other hand, presents the concepts needed for overcoming the overly rigid decoupling. On top of the physical architecture a common control flow connects all components to a tightly integrated system.

The separation into two chapters yielded two chapters of fairly similar size. Also few forward references were needed and some forward reference were in fact dissolved by rearranging subsections. All these observations hint at having found a good modularization of the issues at hand.

At the level of mechanisms this chapter introduced little more than just multicast communication based on registering message patterns. How this mechanism should be used is largely motivated by interaction with other concerns:

Integrity motivated change propagation,

Meta modeling suggested representative objects for uniform access to objects and tools,

Behavior utilizes the `tool_execute` message type for tying tool functions into the behavior part of the meta model.

Other concerns gave lower-level guidance on how to implement control integration. The presentation of this chapter should suggest that implementing control integration in an existing system that already obeys all concerns discussed before this chapter should be surprisingly easy *provided* that all architectural parts are accessible for modification, most notably the “middleware”, in our case MSG, and the libraries by which it is encapsulated. This is, however, speculation because change propagation based on some form of control integration was part of PIROL from its very first days.

8.4.1 Independence with regard to earlier chapters

The development of PIROL has both similarities and decisive differences when compared to modern component based software development (CBSD). In general, CBSD is faced with the selection of an appropriate component model along with its technical infrastructure like middleware and containers. This selection can either be delayed to a late point during design in case the application fits into an unspecific mainstream component architecture. For those applications that have very specific requirements towards their architecture specific technology has to be selected significantly earlier during development, maybe already during analysis.

Co-development of system and infrastructure

PIROL’s component model matured along with the actual system development. In the workbench-MSG-tools subsystem (i.e., everything above PCTE), infrastructure and application where developed simultaneously. Only so it was possible to keep the component infrastructure as presented in this and the previous chapter aligned with all concepts presented in earlier chapters. Any selection of existing infrastructure would have significantly restricted the design space of PIROL. The model presented in chapters 1 through 6 is very specific and normally would not allow a-posteriori transformation towards a standard component architecture. The development of PIROL did not follow the same sequence as its presentation in this thesis. Still I consider this presentation as re-assuring in the following sense: Most decisions presented in chapters 1–6 are of a more conceptual nature than the mechanisms presented here and in the previous chapter. The fact that the earlier chapters could be written in a self contained way with little forward references to these later chapters demonstrates that those concepts where not dictated by technology.

Referential structure of this thesis

In order to underpin the claimed absence of forward references, the LaTeX text processor has been used to identify all forward references across the border between chapters 6 and 7 (demonstrating the usefulness of explicit references)[9].

Out of a total number of 255 cross references within Part II, 14 references cross the mentioned chapter boundary. Most of these reference are of purely informative nature as they lead the road to further discussions which are not needed at the point of reference. Three references connect features presented in early chapters to motivations for their introduction that are given in later chapters. These features are: Derived attributes, attribute guards and dangling creation. While these references are considered valuable for the presentation, all features are also motivated in their context of introduction. The presentation and argumentation would still be sound without these references.

Finally, one caveat is expressed by one of these 14 forward references: transient attributes cannot be accessed by tools. This is a valuable hint that the kind of an attribute and its visibility should be separated (e.g., by introducing visibility modifiers in Lua/P).

Restricted visibility [←7.4.2] *of transient attributes* [←2.2.1]

Concepts versus technology

For a generic system like PIROL which provides infrastructure to systems built on top (concrete instantiations of PIROL), concepts and technology are highly interdependent. The degree of decoupling achieved by the presentation in this thesis shows that mechanisms indeed follow intended concepts and not vice versa.

8.4.2 Concerns and their relations

This thesis emphasizes the relations between different concerns. These relations are a key to motivation of design decisions. Relations say, why things are done this way, and what else must be changed if one concern is changed. In Sect. 15.2.4 we will see an approach of concern modeling that exploits relations for planning maintenance tasks. On a textual level, references are a comparably weak means for dealing with relations. For this reason, Fig. 8.2 gives an intuitive map of major concerns from this and the previous chapter. The following types of relations are shown:

[9]Numbers hold at the time of writing these lines, which is after chapters 9 and 10 have already been written.

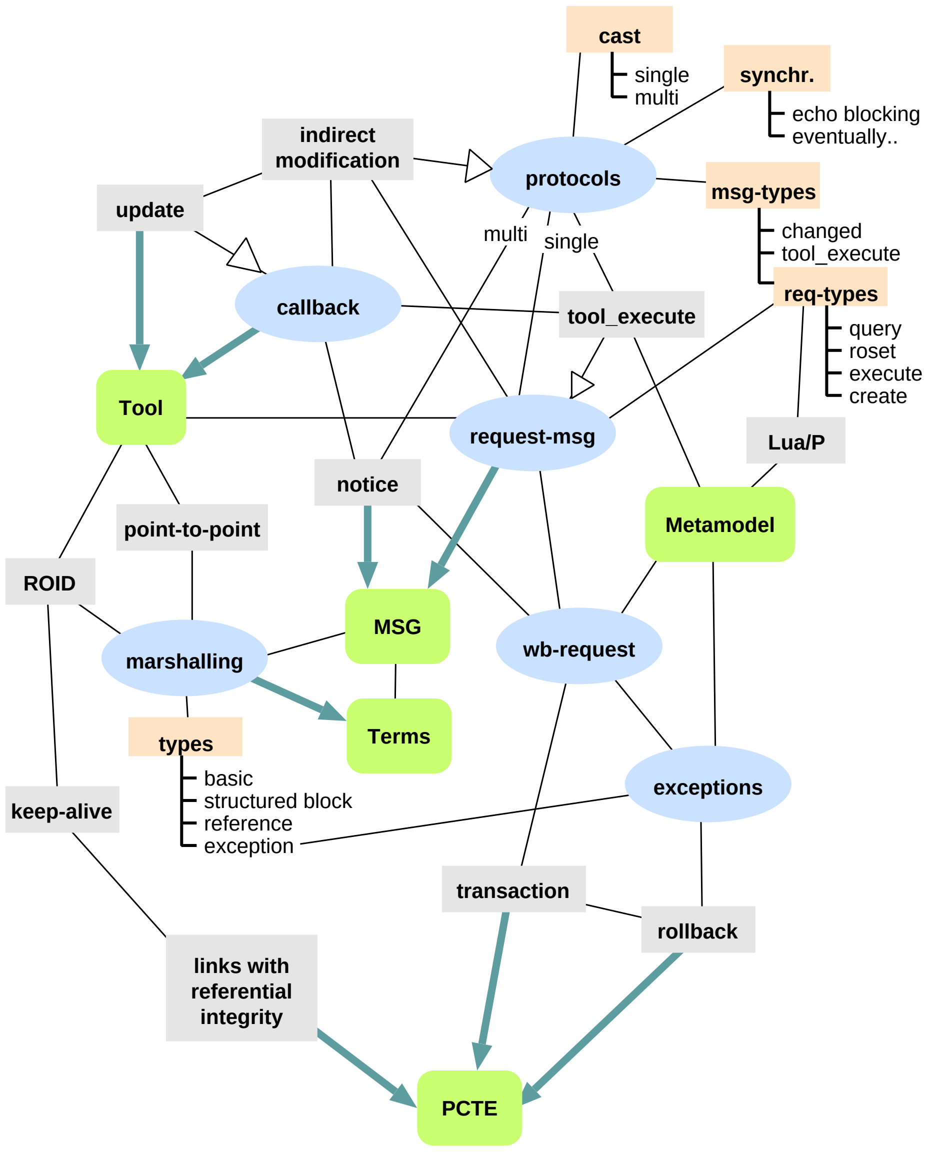


Figure 8.2: Concerns and their relations

simple line	——	unspecified relation
bold arrow	➡	points from a logical concern to its implementation
line with triangle head	—▷	specialization (like in UML)
“comb”	├ single └ multi	list of alternatives

The symbols for concern signify:

gray rectangle	▭ notice	concern with few relations
blue ellipse	⬭ callback	central concept that connects many concerns
green rounded rectangle	▢ MSG	physical concern (software)
bisque rectangle with “comb”	▭ types ├ sub └ reference	classification

What can be achieved by such maps? The author uses graphical representations mainly for arranging elements in a plain until a reasonable layout has been found. From his experience, finding a good layout often coincides with a better understanding of interrelationships, i.e., while arranging graphical elements a mental model is re-shaped and an intuitive understanding of grouping and graduated importance is formed. A layout may be considered reasonable if it has few intersections and short edges. Such criteria can be optimized by a spring-embedding algorithm. At the same time, background knowledge concerning the elements being represented in the diagram is also relevant, in order to come to a layout where conceptually related items are grouped together. From what has been said above, a combination of manual arrangement perhaps aided by algorithm support would be most helpful. Significant research has been done on visualizing software, where layout is a very relevant issue. The concern map given above differs from standard software visualization in that it connects elements and concepts from different levels of abstraction. Sect. 15.2.4 will discuss concern modeling in more detail.

Chapter 9

Multi User Capability

The three–tier architecture presented in Chap. 7 has not yet been fully motivated. Of course the workbench serves the purpose of executing Lua/P methods an coordinates tools. But this is not all. The existence of the middle tier is originally motivated by the requirement to serve as a multi user environment. In this concern the workbench is the point of reference for defining locality and cooperation between users. This thesis is not intended to give a complete overview of computer supported cooperative work (CSCW), though a analysis exists ([Kru00]) how some concepts of CSCW fit into PIROL. As each CSCW model makes some assumptions about the style of cooperation, PIROL tries to stay more general in order to find a platform that suites software development projects of all kinds. The fundamental concepts addressed by PIROL are

1. coordination of data access
2. user context
3. user communication

Item (1) can be subdivided into (a) issues of synchronization and change propagation and (b) access control. In conjunction with (2) it should be possible to work in a private workspace without continuous synchronization with other users such that only an explicit commit operation publishes data to other users. Other changes, however, should immediately be visible to all. The user context should further-on define the role under which the user currently operates, setup his permissions and provide local folders and settings. Item (3) calls for fundamental services on top of which all important styles of communication can be implemented that include, among others, automatic notifications, a mail service, blackboards, and online communication — both textual and through diagrams or drawings. The basic mechanism for user-to-user communication is a second MSG channel that connects all workbenches within a project. Finally, it should be possible to organize users in (nested) groups.

At this general level, i.e., before customizing PIROL for a concrete project or organization, little more can and need be said about the intrinsic requirements of multi user capability. The implementation of these requirements can best be elaborated by simply iterating through all previous concerns and explicating what has to be added in order to consistently fulfill these requirements.

9.1 Multi user capability interacting with other concerns

9.1.1 Meta modeling

Implementation ▽

The relationship between meta modeling and multi user aspects is twofold: The meta model comprises packages `GENERAL`, `PROCESSES` and `COMMUNICATION` that define the key abstractions for the issues at hand. Secondly, (almost) each RO should be subject to the mechanisms described here. These are the relevant classes:

Classes for access control.

`PERSON`	Representation of a user that registered in this installation of PIROL. Each `PERSON` object also corresponds to a user account at the operating system level.
`GROUP`	A group of persons. This may or may not correspond to a Unix group. If it does, it can be used to restrict access to a project to the members of this Unix group.
`ROLE`	This subtype of `GROUP` currently has no real difference to `GROUP`, it was introduced only for semantical reasons in order to express, that some groups like *administration* or *quality assurance* may have special rights.
`AGENT`	This common super-type of `PERSON` and `GROUP` is used according to the *composite* design pattern in order to allow nesting of groups. The above definition of groups needs to be rephrased to: a group is a list of *agents*.

Classes for explicit communication.

`MESSAGE`	This and its superclass `NOTE` define the elementary messages that can be sent to a list of agents. During delivery each message is wrapped for each recipient in an `ENVELOPE`. The envelope is owned and thus writable by the recipient, while the message is not copied and may not be writable for every recipient.
`ANNOTATION`	An annotation can be attached to any RO regardless of ownership and writability. This way, everybody in a project may comment on each object he sees, without requiring write permissions for the object. In a "normal" implementation, attaching an annotation to an object, would require write permission. In PIROL, this operation of attaching is considered an improper modification which is governed by different rules.

Improper modifications
[p.124→]

User context.

`WORKBENCH`	This is the central class for defining a user's context. Attributes `owner`, `current_group`, `current_project` setup a users permissions and responsibilities. `default_readers` and `default_writers` setup default permissions for newly created objects (comparable to, but much richer than the `umask` environment variable in Unix). Some `FOLDERS` are also contained, of which `inbox, outbox` and `tools` have special meaning. The tools folder contains (references to) all tools that can be launched by the current user.
`PROJECT`	The common context for all `members` of a project. It defines several defaults, the `product` under development and the `process` used.

Mail delivery [9.1.4→]

Context menu [11.2→]

Integration of multi user concepts.

`ANY_RO`	Superclass of all entities that shall be controlled by the mechanisms at hand. Each `ANY_RO` has an `ATTRIBUTION` which in turn contains a `PERMISSION` object. Those few classes that should not include these properties inherit from the more general class `ANY` instead of `ANY_RO`.
`PERMISSION`	Permissions are modeled in terms of a reference to an `owner` (`AGENT`), and a list of `readers` and `writers` respectively, i.e., agents that are allowed to read or write the object.

Class `ANY_RO` integrates access control for objects of all of its subclasses. On the other hand, messages and related items of communication may refer to any object in the repository. This way there is no need of a special concept for mail attachments or the like, because messages may simply refer to any document or even sub–item of a document, without the need of encoding and decoding. More specifically, messages may contain a body which is an HTML text, that — through the indirection shown in Sect. 6.2.3 — refers to arbitrary ROs.

References from HTML to ROs [←6.2.3]; PIROL URLs [11.3.8→]

9.1.2 Persistence

Implementation ▽

H–PCTE offers sophisticated techniques concerning access control. 23 distinct access rights exist such as `NAVIGATE`, `READ_ATTRIBUTES`, `DELETE_LINKS` or `OWNER`. Furthermore each object carries an access control list (ACL), that may grant or deny any set of rights for any user or user group. This is more detailed than what is needed in PIROL. First the large set of rights is grouped into owner-, read-, and write-rights. Next, PCTE functions are wrapped by Lua/P methods of the classes `PERMISSION` and `ANY_RO`. Finally, the functionality of PCTE is tied to these meta model classes by means of attribute guards. All attributes of `PERMISSION` are guarded in such a way that, e.g., adding a `PERSON` to the list of `writers` of an `ANY_RO` implicitly calls `ANY_RO:allow_read(agent)`. The latter function eventually leads to a call of `Pcte_object_set_acl_entry` from the PCTE API. This way for a client it suffices to manipulate the `PERMISSION` object in order to achieve any desired changes of access rights.

Care must be taken, when modifying an object's ACL not to lose the owner and read rights before these manipulations are complete. This requires a kind of lazy evaluation: when withdrawing rights for an agent this agent is only recorded in one of the lists `removable_writers` etc. A call to `commit_rights` is needed in order to make all these changes effective. The methods of `PERMISSION` schedule `commit_rights` for later execution by means of a workbench internal function `eventually_call`. Only after the current request has returned, all permission changes are made effective. Through `eventually_call` access control directly hooks into PIROL's request protocol. It is important that any errors that might occur when committing permissions can be detected in advance, because the delayed `commit` call cannot report errors to clients that have already received their reply.

Scheduling by `eventually_call` [←7.3.1]

Class `ANNOTATION` [←p.122]

When presenting class `ANNOTATION`, the concept of *improper modification* was introduced. This concept is implemented using a specific feature of PCTE. In PCTE, each link is accompanied by a reverse link. Creating a link does not require write permissions at the target object, but still adds the reverse link as an implicit[1] outgoing link of the target. So, when annotating an object `o` by an annotation `an`, only the link `an.target` is created explicitly, and the list `o.annotations` is in fact collected from all implicit reverse links of incoming `target` links at `o`.

If this solution using reverse links had not existed, we might have been forced to open permissions in PCTE very broadly and implement all PIROL access control in the workbench. This would have rendered the PCTE access control useless for PIROL and would probably cause considerable performance penalties, because access control outside the PCTE kernel is probably much more expensive. Instead, our solution does not discard functionality from PCTE but instead makes an additional feature of PCTE (reverse links) partially visible in Lua/P.

9.1.3 Granularity

Discussion ▽

Using a fine grained meta model enhances concurrent work, because conflicts in data access are less likely. Different users may manipulate different parts of the same document as long as they don't modify the same attribute. Ownership can be granted at the object level which is also much finer than in file based environments. Of course access administration contributes to the per-object overhead that led to the introduction of term types. So it is obvious that for very fine grained structures no detailed access control is feasible. From the perspective of access control, *terms* are atomic. This is the trade–off that had to be settled in order to balance detailed control and performance.

[1] `implicit` is the weakest link category in PCTE.

```
ListAccess PIROL.mailspool {
    remove = method (i)
        local envelope = self[i]
        if not (envelope.owner and
                 envelope.owner:eq(REPOS.workbench.owner)) then
            pirol_error(_ERROR.WriteAccessDenied,
                        "Cannot remove Envelope, not owner")
        end
        self:raw_remove(i)
    end
}
```

Figure 9.1: Specialized access control using an attribute guard

9.1.4 Behavior

As we have seen, Lua/P methods are one step in encapsulating the access control mechanism of PCTE in order to make it available to the environment. Methods are even more important for communication between users.

Mail delivery

New Feature ▽

The method `MESSAGE:send()` does all the work of sending a `MESSAGE` to all of its recipients. Two ways of delivery exist. If a recipient is logged in to PIROL, the method `PERSON:inform` is used to send a request in the workbench-to-workbench channel that will be received only by that person's workbench. The `MESSAGE` object — wrapped in an `ENVELOPE` — is passed as a parameter in that request and the receiving workbench reacts by inserting the envelope into the user's `incoming` folder, which is an attribute of class `WORKBENCH`. A request is used rather than a notice so that the calling workbench can find out, whether the request was delivered or not.

If the person is not logged in, the `MESSAGE` is appended to a global list `PIROL.mailspool`. For this list special access control is installed. Note, that in comparison, SMTP mail delivery requires a privileged process that appends mails to user mailboxes. In PIROL no additional process instance is needed but access to the `mailspool` list is hand-coded, such that everybody may append items to this list, but only the owner of a mail (the recipient) may read and remove it from the list. Such hand-coded access control benefits from two levels of control: in PCTE this list is accessible for all users. In Lua/P, however, an attribute guard restricts access to this list in the way described above (cf. Fig. 9.1). Mails in the `mailspool` list are retrieved when the recipient starts the workbench for the next time. The workbench startup procedure `WORKBENCH:load` calls `check_mail` which will move any new mail to the users `incoming` folder.

Discussion ▽

Sect. 9.1.2 argued that access control should be used from PCTE and the workbench should not re-implement this feature. This still holds for the general case. Attribute `mailspool` is a singular exception to this rule, which does no harm to the general picture. On the other hand the guard based solution demonstrates how specific elements of the meta model can effectively be adjusted using advanced features of Lua/P.

PIROL provides five fundamental methods by which mail delivery can be freely programmed using Lua/P.

- `PERSON:inform`
  Try to directly notify another user, pass a String message and an arbitrary RO as arguments.

- `ENVELOPE:send`
  Either `inform` the recipient or put this envelope into the global `mailspool` (this method also transfers access rights of the envelope to the recipient).

- `WORKBENCH:receive`
  Callback method for receiving mails while the workbench is running. Prints out a message to the console and inserts the message in the current user's `incoming` folder.

- `WORKBENCH:load`
  This method is called by the workbench on startup. Thus it can be used as a hook for adding more specific startup-behavior.

- `WORKBENCH:check_mail`
  A specific hook (called by `WORKBENCH:load`) that transfers any mail for the current user from `mailspool` to his or her `incoming` folder. This is how mail is received that was delivered while the recipients workbench was not running.

These methods are all that is needed to integrate a full-blown mail service into PIROL. Moreover, Lua/P can be used directly for implementing any special mail handling like filtering, sorting into different folders, auto-replying etc. The hook by which custom mail handling can be installed into PIROL is subclassing `WORKBENCH`, overriding the relevant methods, and upgrading one's workbench object to the new class. An even lighter-weight technique would be to install only one additional hook into class `WORKBENCH`, which would read a string attribute `mail_filter_script` and interpret this string as a Lua/P script on each incoming mail. This would allow customized mail handling without creating a new Lua/P class for each case.

The implementation of any customization, be it by subclassing or by a script attribute, has direct access to all properties of the incoming message as well to the full context of the current workbench. Thus, mail handling can be programmed in a much better structured way than using standard tools like `procmail`, which operate mostly string-based, performing all kinds of pattern matching.

9.1.5 Exception handling

Discussion ▽

Access control is one of the reasons why exception handling is so important in PIROL: every operation may possibly throw an `AccessException` because of insufficient permissions. Obviously it is not the responsibility of client code to check permissions in advance because this is — with few exceptions — not related to any specific operation but cross-cuts *every* operation. Tool implementations must respect this. They must be able to recover from access exceptions at every request.

Performance measurements regarding permission checks [14.1.1→]

The error types introduced by access control are `ReadAccessDenied`, `WriteAccessDenied` and `NotOwner`. The Java library for tools maps these errors to three exception classes with `AccessException` as a common supertype. The special role of access exceptions with respect to exception handlers has already been mentioned.

Exception handlers [←7.4.5]

9.1.6 Integrity

Discussion ▽

Fine grained access control helps to reduce conflicts but also places additional burden on data integrity. The problem is, that PCTE simply hides inaccessible objects and links such that these objects don't seem to exist. This might render a document semantically invalid if seen by a user who lacks the permission to see some of the objects contributing to the document. There is no easy answer to this because this user has no chance to find out, that he doesn't see certain objects.

As a consequence, strategies for consistent access rights are needed on top of the mechanisms presented here, such that no object within a document has less read rights than the document itself. This is not realized in PIROL.

Implementation ▽

We have already seen, how attribute guards guarantee that the information in `PERMISSION` objects is always consistent with the actual ACL in PCTE. Access control, which is performed at every object access, is a critical point for the performance of PIROL. One optimization regarding object creation is done using additional functions from the PCTE API. The lists `WORKBENCH.default_readers` and `WORKBENCH.default_writers` define the default ACL of the current PCTE process. PCTE uses this ACL for each object created in the same process. Again attribute guards are used to propagate the Lua/P values into the PCTE API. This produces, however, a situation, where the initialization of `PERMISSION` objects needs to setup this information *without* triggering the guard because in PCTE permissions are already correctly initialized. Thus Lua/P needs to access these attributes directly using the `raw_append` method of lists.

Discussion ▽

Visibility of this method should preferably be removed for any scope outside a list guard implementation. For this reason these few cases that depend on creating links without triggering a guard should probably be implemented in C as to encapsulate the entire `PERMISSION:init` method into an atomic operation. The advantage of such a solution would be to prohibit tampering with low level operations and bypassing attribute guards. The C–Lua interface seems to be a good encapsulation border for such purposes.

Static correctness [14.2.3→]

9.1.7 Client–Server Architecture

In a multi user environment additional measures have to be taken in order to coordinate concurrent and distributed data access. The workbench accesses the repository via the H–PCTE client library which connects to a central repository server using a proprietary protocol. Little is known about this protocol, but it adds a form of distribution to PIROL. Different workbench sessions can run on different nodes, but they need to share a common file system. This is a restriction that nowadays might not be tolerated, but that is nothing users of H–PCTE can decide about. The protocols between workbench and tools are quite elaborated. In contrast, access to shared repository data from different workbenches fully relies on PCTE mechanisms and has some deficiencies in PIROL. This has historical and technical reasons, some of which will show up throughout this section. First, I will, however, report on what is known about this shared data access with respect to PIROL's architecture. Investigating the forces involved in this concern along with their relations, should provide an understanding that is almost as valuable as a prototypical implementation.

Forces in designing fine grained access coordination

These are the forces, that have to be considered when designing coordinated access to fine grained data.

Segments. H–PCTE is intended to be a high–performance implementation of the PCTE standard. It draws its performance mainly from the fact, that it operates as a main–memory database. The unit of data that can be loaded into memory is called a *segment*. Segments can be loaded either to the H–PCTE server or to the client process, in our case the workbench. Loading a segment into the workbench is a significant speed improvement over accessing a segment that is loaded to the server.

From this it seems desirable to load as much data as possible to the workbench but unfortunately, loading a segment to one client makes it completely inaccessible for all other clients. So loading to the workbench is limited to data that are guaranteed not to be needed by other users, that is, a *local workspace*.

Working with a local workspace in a designated segment will involve moving objects from one segment to another. This interferes with the strategy of object lookup, that uses the segment information encoded into the `exact_identifier`, because the identifier remains unchanged and may thus refer to a segment in which the object will be unknown after moving. This is not a hard problem, but requires an improved lookup strategy, which is already implemented: PIROL encapsulates the `move` operation such that a weak link (of category `reference link`) still points from the old segment to the moved object. So, if an object is not found at the standard location[2]

_/.segments/*segment-id*.known_segment/*oid*.object_on_segment

it will be found on the same segment at *oid*.moved_object.

Versioning. Several considerations hint at a possible linkage between workspaces and versioning. Loading a segment makes all its objects inacces-

[2]The reference is shown in the syntax of PCTE path expressions, starting at the repository root object denoted by "_".

sible to all other users. So if a read-only copy should be kept for others this fits a model of version control, where development items are checked out to a private workspace while leaving the last saved version as a read-only copy in the repository.

Secondly, project–wide communication traffic regarding data changes should be reduced as compared to the flood of `changed` messages on the tool–workbench channel. The idea of an explicit and grouped commit operation has already been discussed. The above model of check–out to a local workspace provides for a check–in operation that can easily play this role of publicly announcing changes that have been made in the private workspace.

Commit [←p.121]

Related Work

▽

Unfortunately, versioning of fine grained data has not yet seen a solution that is practically useful at the user level. The GOODSTEP project [AAA+94] has invested several efforts to this issue. In [ESW93] the authors discuss different duplication strategies (lazy versus eager duplication of subtrees during creation of a new version) for versioned nodes and possibly also for reference edges. Such issues involve very intricate design decisions and the authors conclude that the seemingly exclusive strategies might need to be combined a hybrid strategy. Also [EAMP97] identified versioning as a problematic issue in using an OODBMS for SEEs. They report on having included a versioning mechanism in the O_2 OODBMS. Unfortunately, this database has meanwhile disappeared from the market. Most details on versioning in GOODSTEP can be found in [SS95]. This approach in fact solves how versioning of documents can be broken down to fine grained version relations of contained nodes. However, due to the dominant role of documents in GOODSTEP, this approach cannot be transfered to PIROL, where any object may belong to several documents simultaneously.

△

One problem, that has been identified in [Grü97] relates to the versioning of links[3]: usually databases only provide a mechanism for versioning objects. There is, of course, a desire to handle links only implicitly, i.e., consider them as part of their origin object. On the other hand it should be possible to express that the target of a link is in fact a specific subset of the history of the target object. A basic solution is depicted in Fig. 9.2. During the evolution of object `B`, its link to object `C` has been changed to point to object `A` instead. It is no problem to duplicate a link if a new version of the source object is created. The issue is: how are versions from the target history selected. For this purpose every link may carry an integer attribute `v`, which specifies how many versions from that actual target participate in this relation.[4] In the example, the link `B1.2`→`C1.2` specifies, that two versions, `C1.2` and `C1.1`, are legal target objects. The effect is, that this link attribute allows to regard links as full relations in the mathematical sense, where one source may relate to a defined set of (consecutive) versions at the target side. This clarifies that, e.g., `B1.3` and `A1.1` are not considered connected.

Two problems remain: (1) Variants within a version history would introduce

[3]In [ESW93]: *(reference) edges.*

[4]The link will always point to the most current legal target, i.e., the link attribute counts from the actual target along its chain of *predecessors*.

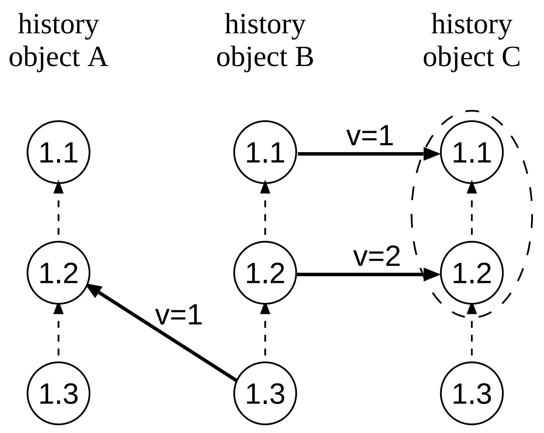


Figure 9.2: Versioning of links

ambiguities into this referencing scheme. (2) The source side of a link would require an explicit "incorporate" command, in order to accept a new version of a target object. Consider, how object `B` would acknowledge the creation of `A1.3`, which should have the effect of changing the link and incrementing the link attribute. It is not perfectly clear, *when* such incorporate command should be triggered and by what instance of the architecture. Implicitly using newest versions regardless of the context of creation might render a document inconsistent, because the context that creates `A1.3` will know nothing about the special rules of consistency in all referring contexts.

While both problems seem solvable, versioning support has not been integrated into PIROL. Note, that the above schema does not cause any additional indirections in the normal case: for most situations the link can be navigated without additional effort. The target history will have to be search for a most suitable version only if specific version selection rules require so.

Composition links. PCTE supports the distinction between reference (existence) links and composition links for the purpose of performing certain operations on compound objects with just one API call. This may be the desired behavior for moving an object from one segment to another. Versioning might require differently, because fine grained versioning should not unnecessarily copy complex structures if only specific elements are changed. Access control might again require a different rule. One could fancy a compound object representing, e.g., a class diagram. Withdrawing access to this diagram would automatically withdraw those rights also for all contained classes including their implementation structure. Such propagation of permission changes seems counter intuitive.

For these reasons, efforts on exploiting composition links in PIROL have been canceled.

Transactions. Currently the workbench runs in transactions. Requests from tools only set checkpoints within the transaction. Uncommitted transactions may, however, block the workbench processes of other users, if they try to access objects involved in the transaction. Currently, the workbench works around this problem by committing the transaction and starting a new one after a few seconds of idle waiting.

On the other hand, [DK95] have shown, how transaction roll–back can be used to implement a universal undo mechanism. This requires the transaction to remain open with only checkpoints in between. Also, [Pla99] has developed fine grained transactions, that help to reduce the danger of conflicts and blocking. It seems natural that a fine grained meta model with fine grained access control and versioning also requires a fine grained transaction model. This, however, might require PIROL to let tools operate in distinct PCTE–processes. While currently the workbench is the only PCTE–client in PIROL, much finer control would be possible, if PCTE knew about each tool. As we don't want to modify our architecture, the workbench would have to run several PCTE–processes as threads within one OS process, such that each tool had its own workbench thread, that answers all its request. This way, coordination of separate tool transactions becomes a matter of coordinating the several threads of the workbench. A multi–threaded workbench, finally, requires a multi–threaded Lua interpreter which was not available before Lua 4.1, of which only unstable versions where available at the time of this writing.

Thus during the development of PIROL experiments with a multi–threaded workbench were not possible. The previous discussion demonstrates the complexity and tangling of the concerns notification, segmentation, versioning, and transactions. Each of these fields has had plenty of research and development. Any subsequent development of a distributed, repository based environment should tackle the integration of the findings from these fields during the early design phase. This seems to be the major remaining challenge.

Concurrent editing

New Feature ▽

Any kind of online communication and cooperation can easier be implemented efficiently, if communication paths are shorter than the concepts discussed above. For this reason, PIROL features the notion of inviting another user to one's session. The idea is to perform concurrent editing within the same workbench session. All that needs to be done is:

- Give a copy of the MSG lock-file to the other user.
- Assemble a command for launching a tool, such that the other user can launch the tool from his or her shell rather than via the workbench.
- The invited user sets an environment variable to point to the foreign lock-file and starts the given command.

By this simple technique several users on several computers may operate simultaneously on the same document. It might be useful to identify invited users as guests. A guest could then be assigned restricted rights, e.g., by allowing to start only a limited set of tools, or opening only a fixed set of documents. Also marking the actual author of changes might be relevant.

MSG has already been enhanced for PIROL with the capability for identifying the client that issued a given request. Based on this distinction the workbench could use a restricted tool folder in order to disallow unwanted tools. The safest solution would of course include assigning a special password for guests so that already the MSG server can distinguish different users/groups. Note, that all restrictions and author markings would have to be implemented explicitly, because in this model all modifications would be carried out in the same workbench process. Only a multi–threaded workbench could actually run different threads on behalf of different PCTE users.

Class TOOL_POOL [←8.3.1]

The restrictions that should be added to this model of cooperations are expected to be of low impact on the system. It has been shown, that concurrent editing in this style is in fact practical. Note, that only three requirements have to be met: (1) there must be a secure channel for copying the lock-file, (2) the invited user must have (parts of) PIROL installed on his or her file system, and (3) he/she must be able to connect to a socket at the machine running the MSG server. The communication requires only low bandwidth. △

9.1.8 Control Integration

It has been shown how method `PERSON:inform` provides access to a second MSG channel connecting the different workbenches within one project. The registered callback at the other side is `WORKBENCH:receive`, which places the received object into the user's `inbox` folder. In analogy to tool representatives, objects of types `PERSON` and `WORKBENCH` serve as *user representative* and *environment representative* respectively. A user representative is an RO that allows communication with the represented user. Adaptation of class `WORKBENCH` through subclassing allows to adapt the environment and explicit calling methods of this class means to use builtin functions of the environment.

Tool representatives were introduced in Sect. 8.2.3.

Choice of notification mechanism. PIROL could make use of two different mechanisms for delivering change notifications. H–PCTE has a notification service and MSG could be used for notifications also between different workbenches. It might be easier to re-use the MSG notifications also between workbenches but the H–PCTE mechanism might be more specialized for this task. This choice is not expected to have great impact on other concerns.

9.2 Summary

Multi–user capabilities are a fairly difficult issue, because they can only be addressed, after quite some details about architecture and technology are already defined. On the other hand, support for access control has to be integrated in very low layers of the architecture. Difficulty is further imposed by the desired genericity, which should leave a choice of the most appropriate cooperation model to environment customization.

Attribute guards and exceptions are mechanisms that are sufficiently close to the core of the system to transparently support access control. An example application of guards concerns the hand-coded access control for `mailspool`. Other low-level mechanisms that are exploited by multi–user support are delayed calls (`eventually_call`) and reverse links (improper modification for attaching `ANNOTATION`s).

Multi–user support also applies translation between different representations of entities like users and tools. The former appear as Unix user, PCTE user and `PERSON` object (associated to a `WORKBENCH` object). The latter appear as MSG client, OS process, `TOOL` RO and should eventually correspond to one PCTE process that runs in its own workbench thread. In both cases, encapsulation of mappings happens using representative objects.

Chapter 10

Logical Component Gluing using DVCs

10.1 Striving for "logical" independence

The infrastructure presented so far enables "physical" "wiring" of tools and repository. MSG provides for inter process communication. Term types, marshaling and different language bindings allow to exchange data without loss of structure and type information. This kind of integration has from the beginning been motivated by the observation, that message passing provides for loose coupling as compared to repository based integration due to the need to agree upon a common data model ([Bro92]). As also pointed out in [Szy98], this basic, technical wiring is, however, not enough. Allowing each tool to be developed independently with its own data abstractions requires also some form of *logical pluggability* responsible for dealing with mismatches between the data models of tools and PIROL's meta model.

10.1.1 Anticipating meta model mismatches

The repository model includes only elements that contribute to the structure and semantics of the system under development. Classes in the `PRODUCT` subsystem of the repository model are, e.g., `SUBSYSTEM, CLASS, FEATURE` etc. (cf. Fig. 10.1). Different tools have, in general, their own specific abstractions. Throughout this chapter, a concrete graphical editor for (a variant of) UML class diagrams, called *ZooEd* [Nor97], will be used as an example. Its meta model — as presented in Fig. 10.2 — defines the abstractions on which the editor operates and which it needs to store in the repository.

Note, that the two models in Fig. 10.1 and Fig. 10.2 are completely independent of each other. For instance, when comparing the class abstractions in both meta models, namely `CLASS` in Fig. 10.1, and `Class`[1] in Fig. 10.2, it appears that the latter is mainly a graphical abstraction, defined in terms of its position and appearance (appearance is determined by the attributes

[1]Throughout this chapter class names in all capitals refer to classes of the repository model, while names with only an initial capital letter refer to classes of the tool model.

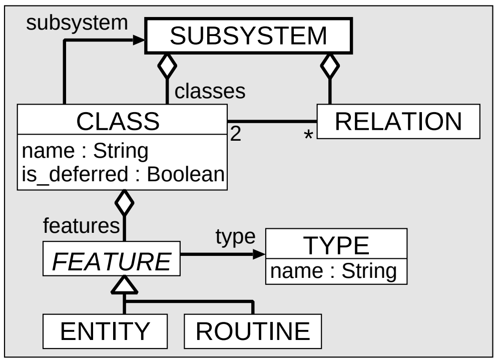


Figure 10.1: Extract from the repository's meta model

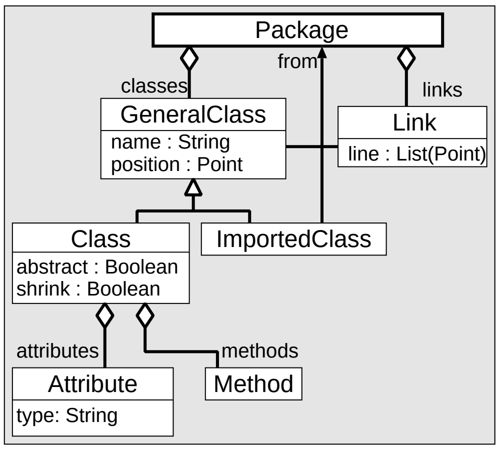


Figure 10.2: Meta model of a tool for UML class diagrams.

`shrink`, responsible for collapsed display of a symbol, and `abstract`, toggling italic/non-italic fonts). Furthermore, a `CLASS` object has a single list of `features` (cf. Fig. 10.1), whereas classes as seen by a UML–diagram editor keep separate lists for `Attributes` and `Methods`, which are drawn in different sections of a class symbol in a UML–diagram. Another difference concerns the representation of the type of an attribute. In Fig. 10.1, the type of a `FEATURE` (and its sub-classes) is represented by the class `TYPE`. On the contrary, a simple string is perfect for representing the type of an `Attribute` within a UML class symbol (cf. Fig. 10.2). Thus, some abstractions in a tool's meta model have direct correspondences to abstractions in the repository model, e.g., `name`, others can be somehow derived from abstractions in the repository model, e.g., `attributes` and `methods`, and yet others are completely new, e.g., the `position` in `GeneralClass`.

10.1.2 Storing shared versus tool–specific data

Given the differences in the meta models and our goal to facilitate the evolution of the integrated environment with new tools, tool–specific abstractions, e.g., a class diagram or any document, are not defined as part of the repository model.

For this distinction we have introduced conceptual objects (CO) as a handle for a document that itself is decomposed into many ROs. CO [←1.1.1]

This separation of *intrinsic* from tool–specific concerns into two distinct repository objects, i.e., beyond compile–time, is crucial. It allows e.g., the same `CLASS` object to appear at different positions and with different representations in different documents. The idea is that (a) there might be several tool–specific (overlapping) definitions corresponding to the same base abstraction, and (b) the decision which one to use might depend on run–time state and/or context.

If integration of different concerns was based on compile–time weaving of all partial definitions into one whole, as e.g., with IBM's model of Hyperspaces [OKK+96] or Garlan's model of basic views [Gar87], it would be difficult to express the fact that the same class object should appear differently depending on whether the package being shown in an UML class diagram owns or imports the class at hand. Furthermore, it would be impossible to have the same class simultaneously displayed in as many different positions, as there are UML documents on the display that contain the class.

Logical tool integration

Having intrinsic and tool–specific features of a tool object encapsulated within different objects in the repository also poses the question of how and where to (logically) integrate these two distinct feature sets to construct a single object as seen by the tool. Stated differently, the question is how to map the features of a tool object to respective features in the corresponding RO–CO pair in the repository. The approach taken in the first design of PIROL was to in-line this mapping functionality into each tool's implementation. If this mapping was not taken into account during a tool's development, its source code had to be modified at integration time. The drawbacks of this integration strategy are discussed in [HM00].

10.2 Dynamic View Connectors

A major contribution of PIROL is the introduction of emerging programming language concepts (most specifically: Pluggable Composite Adapters [MSL01]) into a complex integrated environment. The goal is to facilitate logical integration of independent tools into the environment. Note, that logical integration is not simply a mechanism which can be bound to existing components, but a new set of abstractions is introduced by which environment programmers can express the logical relationship between a tool and the environment. From the requirement to program this relationship using appropriate notions, a significant extension of the language Lua/P is motivated.

The new construct, called *Dynamic View Connectors* (DVC), is implemented as a special Lua/P class whose instances build a new abstraction layer between tools and the workbench. We call this additional abstraction layer *virtual repository*. Each tool has its own virtual repository — a simulation of a repository that matches the model of the tool. The virtual repository is implemented by DVCs on top of the given design using COs. They will, however, hide COs and provide for uniform access to RO attributes and properties in COs. In the following, we first present a rough sketch of what kind of objects live in a virtual repository (Sect. 10.2.1) before going into the details of how to define it (Sect. 10.2.3).

10.2.1 The structure of virtual repositories

It is obviously preferable to have a repository whose model matches the model of a tool to be integrated: the tool simply works with proxies to repository objects responsible for bridging the language barrier between the tool and the workbench and no adaptation is ever needed. However, in an environment where the repository must meet the needs of many tools, adjusting the repository to all tools would yield a bloated repository model with many redundant definitions introducing immense consistency problems, let alone its non-cohesive structure and name clashes.

Our solution to this trade–off is to simulate a best matching repository for each tool — this is what we call a *virtual repository.* Instead of having the structural mapping for a certain type in the tool's model being spread around the tool implementation, *virtual* repository *classes* (VC) are defined that encode exactly this mapping at a single place by providing an abstract view of underlying "real" repository abstractions (VC may also be used as an acronym for "view class"). A VC may also add new features as they are needed by the tool. There is a 1:1 relation between instances of the tool and virtual (or view) objects (VOs), the latter serving to the Lua/P interpreter as "dispatchers" for attribute accesses on the former.

Implementation of documents using COs [←1.1.1]

For illustrating the idea, Fig. 10.3 redraws Fig. 1.1 in the presence of VOs. Until now it was the tool's responsibility to map each object from its data model to an RO and a CO in the repository (this mapping was not represented in Fig. 1.1, as it is not localized in a single place within the tool). The additional layer in Fig. 10.3 performs exactly this mapping. Actually, only objects `#c1`, `#co1` and `#co2` are persistent. Objects `#ic2` and `#c3` exist only during a user session and merely encode a mapping function. However, for the tool they appear to be repository objects. That is why we call them objects of a virtual repository. A VO is identified by the ID–pair `(roid x coid)` referring to the RO–CO object pair (as already seen in the old design) that is now virtually merged into a single abstraction. Note, that this pair is not symmetrical, as each virtual repository is realized by one CO but several ROs.

One can think of VOs as implementing *roles* [RS91, WdJ95] of real ROs: e.g., `ImportedClass` is a role, that any `CLASS` object may acquire in the *context* of a UML class diagram. Acquiring a role includes the option to gain new properties (e.g., `position`). Note that the same base object may play different roles in different contexts. For instance, `#c1:CLASS` in Fig. 10.3 plays an `ImportedClass` role within the context of the `Accounting` diagram and a `Class` role within the context of the `Customers` diagram. The acquired properties differ from one role to the other. Other properties are shared from the base object (e.g. the `name`).

A base object plays a given role always with respect to a certain *context* — a virtual repository as implemented by a CO— which *comprises a set of col-*

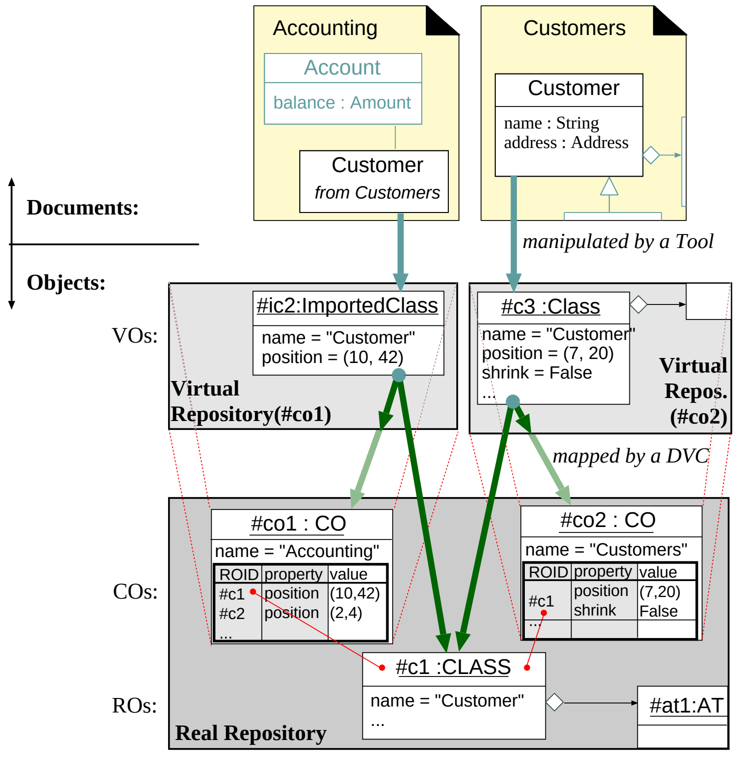


Figure 10.3: A virtual repository simulates tool objects

laborating objects. The idea is that the process of creating a virtual repository from the real repository model occurs not at the level of individual classes, but rather at the level of collaborating classes: When following the `features` association of a `CLASS` RO as seen from a `Class` VO, all reachable `ROUTINE` ROs are implicitly adapted to `Method` VOs. Adaptation of reachable ROs to VOs of the corresponding roles is automated by the workbench. This mechanism is called *lifting* and will be defined below. In fact, as we will see, both `Class` and `Method` will be defined together as types participating in the same higher–level abstraction, that of a `UML_CLASS_DIAGRAM`. Techniques for grouping a graph of objects as adaptations of some base graph of objects are being developed as *Adaptive Plug&Play Components*[ML98] and *Pluggable Composite Adapters*[MSL01].

10.2.2 Implementing tools to virtual class graphs

With the virtual repository abstraction being part of PIROL, each tool is written as a self–contained component to its own meta model — its functionality is encoded in a set of collaborations between the elements in its meta model. The latter is defined by a set of nested interfaces which is called the *expected interface* of the tool. This defines the tool's "view" of the repository, or, alternatively, an "ideal" repository model tailored to the specific needs of the tool. Fig. 10.4 shows part of a textual representation of the model in Fig. 10.2 — this is the expected interface to which the UML–diagram editor is implemented without

```
component interface GRAPH {
        interface Node { position : Point }
        interface Edge { line : List(Point) }
    }
}
component interface UML_CLASS_DIAGRAM inherit GRAPH {
        interface Package {
            name : String
            classes : List(Class)
            links : List(Link)
            method findClass (name: String) : Class
        }
        interface GeneralClass inherit Node {
            name : String
        }
        interface Class inherit GeneralClass {
            abstract : Boolean
            shrink : Boolean
            attributes : List(Attribute)
            methods : List(Method)
            creation place
            method place (name: String, pos: Point)
        }
        ...other interfaces
    }
}
```

Figure 10.4: Expected interface for UML class diagrams.

any knowledge of the repository model.[2]

Integrating a tool into the environment requires to "implement" its expected interface, by defining how elements of the tool model are composed from corresponding ROs and by introducing additional features that might be missing from the repository model. This is realized via a connector class. A connector instance is a first–class object that defines a virtual repository as an aggregate of collaborating VOs (following [ML98] we could say: participant graph) in terms of an aggregate of ROs. By being a first–class object it can be applied dynamically to perform tool integration, hence, the name *dynamic view connector*.

10.2.3 Mapping constructs

Like any Lua/P class, connector classes contain feature definitions and may inherit from other connector classes. Furthermore, a connector class nests a set of

[2]The interface `GRAPH` introduces some general abstractions, that are used by classes in `UML_CLASS_DIAGRAM`.

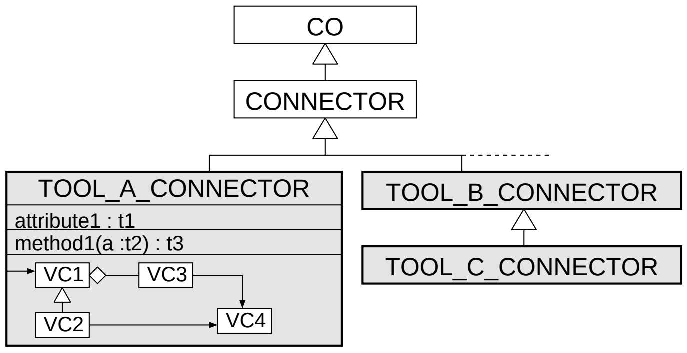


Figure 10.5: Hierarchy of connector classes

view classes, one for each interface in the tool's expected interface. The classes `CO` and `CONNECTOR` are predefined in the repository model with the connector inheritance hierarchy as shown in Fig. 10.5. Connectors supersede the conceptual objects mentioned in Sect. 1.1.1, so in the sequel the abbreviation CO is used for *connector object*.

The constructs that make up a connector definition will be illustrated by means of the example connector `UML_CLASS_DIAGRAM_CONNECTOR` in Figs. 10.6–10.8 which implements the `UML_CLASS_DIAGRAM` tool model of Fig. 10.4 on top of the repository model of Fig. 10.1.

Connector level definitions

Each connector class may **inherit** from another connector and it has **attributes** and **methods**. Lines 7–14 in Fig. 10.6 define that a UML diagram connector is a specialization of `GRAPH_CONNECTOR` (i.e., each UML diagram is a graph built from nodes and edges), and that it refers to the top-level `system` containing the current diagram. It also defines a method `connect` which is used for **creation** of a new connector instance, connecting it to a `SUBSYSTEM` RO.

Creation methods in Lua/P [←4]

Starting at line 15, the example DVC in Fig. 10.6 defines seven view classes (their names are put within boxes in the figure), one of which is designated as the **root** of the model. The RO and VO aggregates that are mapped by a connector both have a designated root object which is known within a CO as `root_ro` and `root_vo`, resp. The root view class `Package` is mapped to the repository class `SUBSYSTEM`.

View class definitions

Mapping view classes to repository classes is specified by means of the **roclass** clause in the definitions of the view classes. At run–time this entails a translation that is called lifting. The lifting operation takes an RO and wraps it into a VO. For each instance `vo: VC`, **vo.co** refers to the enclosing connector, while **vo.ro** refers to the repository object `ro: RC` of which `vo` is a view within `vo.co`.

Lifting [10.2.5→]

A **predicate** may be associated with a view class. For instance, both classes `Class` and `ImportedClass` define views on the same RO class `CLASS` as defined in the common superclass `GeneralClass`. Whether instances of `CLASS` should

```
 1  Connector GRAPH_CONNECTOR {
 2    root = Node
 3    class Node { adds = {position : Point} },
 4    class Edge { adds = {line : List(Point)}}, },
 5  }
 6  Connector UML_CLASS_DIAGRAM_CONNECTOR {
 7    inherit = GRAPH_CONNECTOR,
 8    attributes = { system : SYSTEM, },
 9    methods = { connect (root_ro : SUBSYSTEM)
10                  CONNECTOR:connect(root_ro)
11                  system = root_ro:get_system()
12                end
13    },
14    creation = connect,
15    root = Package
16    class Package { roclass = SUBSYSTEM, /* ... */ },
17    class GeneralClass { roclass = CLASS, inherit = Node, /* ... */ },
18    class ImportedClass { inherit = GeneralClass,
19      predicate ()
20        return not co.root_ro:eq(ro.subsystem)
21      end,
22      /* ... */
23    },
24    class Class { inherit = GeneralClass,
25      predicate ()
26        return co.root_ro:eq(ro.subsystem)
27      end,
28      /* ... */
29    },
30    class Attribute { roclass = ENTITY, /* ... */ },
31    class Method { roclass = ROUTINE, /* ... */ },
32    class Link { roclass = RELATION, inherit = Edge, /* ... */ }
33  }
```

Figure 10.6: Connector for UML class diagrams — Structure.

be seen as `Class`, respectively `ImportedClass`, depends on the result of evaluating the respective class predicates (lines 19, 25) at run–time. So, when following the association `classes` from a `Package` (cf. Fig. 10.2) within the context of a connector, each `CLASS` RO contained in the underlying association (cf. Fig. 10.1) is examined with respect to the predicate of each candidate view class `Class` and `ImportedClass`. The view class whose predicate function returns true is chosen for lifting.

Of course, a view class may **inherit** from another view class that is either defined within the same connector (e.g. `GeneralClass`) or inherited from a parent connector (here: `Node` and `Edge`).

```
1   class Class { inherit = GeneralClass,
2       predicate () return co.root_ro:eq(ro.subsystem) end,
3       uses = { abstract = is_deferred },
4       adds = { collapsed : Boolean },
5       filter = {
6           attributes : List(Attribute) = {
7               base = {features : List(FEATURE)},
8               predicate (f : FEATURE) return f:conforms("ENTITY") end
9           },
10          methods : List(Method) = {
11              base = {features : List(FEATURE)},
12              predicate (f : FEATURE) return f:conforms("ROUTINE") end
13          }
14      },
15      accept ()
16              collapsed = False
17              ro.subsystem = co.root_ro
18      end
19  },
20  class ImportedClass { inherit = GeneralClass,
21      predicate () return not co.root_ro:eq(ro.subsystem) end,
22      uses = { from : Package = subsystem }
23  }
```

Figure 10.7: Mapping details for `Class`.

Feature Mappings

Details of mapping a RO class to a view class are shown in Fig. 10.7 by the example of `Class` and `ImportedClass`.

`Class` and `ImportedClass` define two different views on the repository type `CLASS`[3], by making different sets of underlying `CLASS` attributes visible within the context of a UML tool. For some attributes the connector declares names and/or types that differ from those declarations in `CLASS`. Other attributes are added to the view class without counterpart in the repository class. The constructs provided for mapping features are uses, adds, filter, redirect and hide

uses. This construct is used for making features of a repository class visible within a view class. For instance, the attribute `abstract` in the definition of `Class` is identified with the attribute `is_deferred` in the repository by means of *renaming*. That is, the repository-level attribute `is_deferred` is made visible within the UML view under a different name, maintaining however the same type. The definition of the view class `ImportedClass` contains an example of a repository-level attribute that is made visible within the view under a different name and type. The definition of the attribute `from` in line 22 of Fig. 10.7 is

[3]The mapping roclass=CLASS is inherited from `GeneralClass`.

to say: The repository attribute `subsystem:: SUBSYSTEM` (defined in `CLASS`) is known within the view defined by this connector as the attribute `from: Package` (declared in view class `ImportedClass`). So at run–time, each `SUBSYSTEM` RO reached within the context of the connector via the `from` association, is lifted to a VO of type `Package`. In a similar vein, class `Package` (not shown in the figure) contains this declaration

```
uses = {
    findClass (name: String): GeneralClass,
}
```

This declaration refers to a *method* that is adapted just like the attributes shown above: within this connector the method result is assumed to be of type `GeneralClass`, that is, each resulting object is lifted to the view class `GeneralClass`. This declaration involves no renaming, so `findClass` is used under the same name in both contexts.

All view classes implicitly inherit from a predefined class `ANY_VO`. This class contains the following uses clause, making some standard features visible to all VOs:

```
uses={
    name, get_class, conforms, tostring;
    print_ro=print
}
```

adds. In contrast, attributes `collapsed` and `position` in `Class` do not have counterparts in the core repository model. These attributes are *added* to the model (Note, that `position` is already defined in the abstract connector `GRAPH_CONNECTOR` and is made available in `Class` by inheriting from the view class `Node`). Entries in the adds clause are regular attribute declarations.

filter. This construct applies to Lists in the repository model. The outgoing links of a `Class` VO — `attributes` and `methods` — are different *filtered views* of the same association `features` from the repository model. Keyword **base** creates this tie. A filter **predicate** decides which objects of type `FEATURE` contained in the repository association `features` will be included in which of the resulting view object lists, `attributes`, respectively `methods`. At run–time, those repository objects of type `FEATURE` that pass the filter will automatically be lifted to view objects of type `Attribute`, respectively `Method`. In this special case the filter simply inspects the run–time type of the target RO in order to include `ENTITY` objects in the `attributes` list and `ROUTINE` objects in the `methods` list (see Fig. 10.7, lines 8 and 12, resp.).

redirect. Fig. 10.8 shows a situation where predefined mappings are not sufficient. In the repository model (Fig. 10.1) the `type` of a `FEATURE` is represented by a repository object of class `TYPE`. However, the tool (Fig. 10.2) encodes the same information (the type of a feature) as a simple `String` in class `Attribute` — the view class corresponding to the repository class `FEATURE`. DVCs provide a

```
...
class Attribute {
    roclass = ENTITY,
    redirect ={
        type : String = {
            get ()
                return ro.type.name
            end,
            assign (value : String)
                    local type
                    if value == ""then
                        ro.type = nil; return
                    end
                    type = co.system:find_type(value)
                    if not type then
                        error("Unknown type "..value)
                    end
                    ro.type = type
            end
        }
    }
}
...
```

Figure 10.8: Manual redirection of attribute `type`

construct to hand-code such mappings by implementing a **get** function and an **assign** function. When going from the repository model to the tool model, the name of the `TYPE` RO will do perfectly, as shown in the implementation of get in Fig. 10.8. The other way around, translating a string representation of a type to a `TYPE` RO, when the user enters the type of an attribute via the UML diagram editor, is more tricky. It involves the `system` attribute of the enclosing connector: the method `find_type` of `SYSTEM` is invoked to find a `TYPE` RO whose name matches the string passed as a parameter to assign. Another example for the usefulness of the redirect construct is when an association between two classes in a connector should be mapped to a compound path between classes in the repository model.

Technically, the redirect construct is an improved version of derived attributes. The get function directly corresponds to the derivation function of a derived attribute. The assign function, however, is new, and allows for symmetrical derived attributes that can be read and written just like ordinary attributes. As we have seen, assign functions can be difficult to implement. Omitting the assign function of a redirected attribute is not a static error. The attempt of writing to a redirected attribute without assign function simply results in a `WriteAccess` exception to be thrown.

Derived attributes were introduced in Sect. 6.1.3 and discussed in Sect. 8.3.6.

hide. The opposite of uses has been introduced quite late. It was one of those changes that required less then 10 additional lines of Lua code. The intention is to cancel the effect of a uses clause of a parent class. This can be important, if an attribute is to change its status. By means of hide it is, e.g., possible to change a used attribute to a redirect implementation.

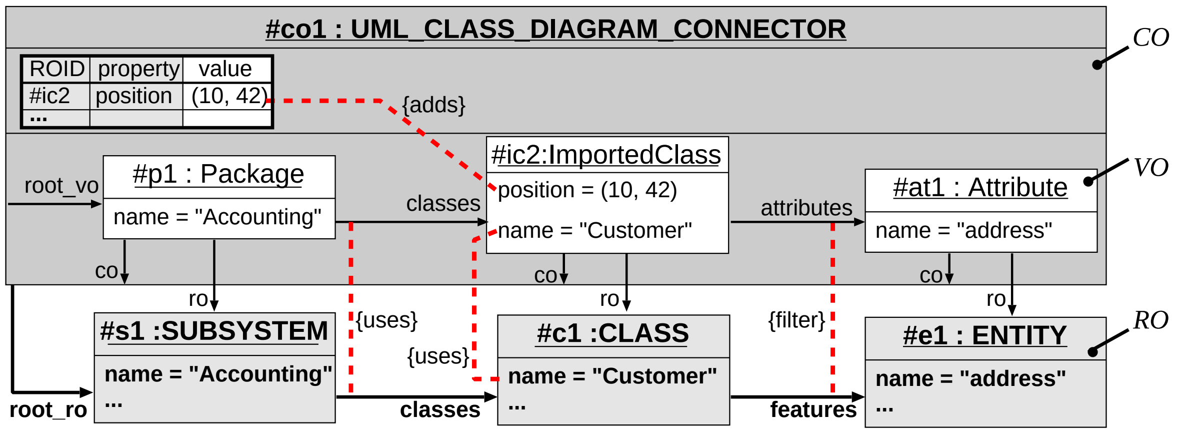


Figure 10.9: Run–time relations between ROs and VOs in a Connector.

10.2.4 Repository, view, and connector objects

Fig. 10.9 illustrates the run–time structure of connector objects and the relationships between the involved VOs and ROs, by the example of the UML_CLASS_DIAGRAM_CONNECTOR instance for the Accounting document in Fig. 10.3 (#co1). The dependencies labeled {uses} and {filter} illustrate, how values are shared between the involved VOs and their corresponding ROs. The {adds} dependency shows that the added attribute position is stored in the surrounding CO.

10.2.5 Conversions

Let us now consider how the relationships between ROs, VOs and COs, are established and maintained. Fig. 10.10 illustrates how root ROs enter a connector when the latter is created. As defined in Fig. 10.6 line 9, connect expects the root RO (in this case of type SUBSYSTEM) to be passed as a parameter. In Fig. 10.10, we assume that s1 is the RO for the Accounting subsystem (#s1 in Fig. 10.9). The execution of the first line in Fig. 10.10, will

(a) create an instance of UML_CLASS_DIAGRAM_CONNECTOR (#co1 in Fig. 10.9),

(b) set the root_ro of #co1 to #s1,

(c) initialize the attributes of #co1 (e.g., the system is initialized with the result of calling get_system on #s1),

(d) wrap #s1 into a Package VO (#p1 in Fig. 10.9) and set this VO to be the root_vo of the connector (#co1).

Of these steps, (b) is triggered by calling the inherited version of `connect` (line 10 in Fig. 10.6) and only (c) is explicitly specified by the programmer in the implementation of `UML_CLASS_DIAGRAM_CONNECTOR:connect`. The rest is taken care of by the Lua/P interpreter.

```
...
co1 = UML_CLASS_DIAGRAM_CONNECTOR:connect(s1)
ed = Workbench:get_tool_for_document_type(co1.class)
ed:show(co1.root_vo)
...
```

Figure 10.10: Connector instantiation

Lifting

In Fig. 10.10, `#s1` enters a connector explicitly: this is the case with root ROs. Other ROs might enter a connector scope implicitly when they are reached via a reference (from another already lifted RO) for which a view mapping is defined in a uses or filter declaration. Basically, it is the type defined in this mapping (the *declared type of reference*), that is used to automatically *lift* (wrap) an RO to a VO. However, the actual algorithm for lifting has to consider some more conditions.

Given an RO of dynamic type RC_{dyn} and a reference of static type VC_{stat} within the context of a connector instance of dynamic type C_{dyn}, the lookup of the virtual class VC_{dyn} to use for lifting — this algorithm is called *smart lifting*— proceeds like this:

1. Starting from (and including) VC_{stat} collect all sub-classes that have a roclass to which RC_{dyn} is conform, however, omit all classes that are defined in a connector that is a sub-class of C_{dyn}.[4]

2. Sort this list such that all classes defined in C_{dyn} precede classes defined in parents of C_{dyn}. Furthermore, within this sorting, sub-classes precede super-classes.[5]

3. From this list, take the first class, which either has no predicate, or whose predicate function evaluates to true for the given RO.

Lifting is finally performed by setting up a record (Lua table), consisting of the references

`ro`	The RO to be lifted
`co`	The CO in which the VO shall reside
`class`	The view class determined by the above strategy.

[4]**Rationale**: Only classes defined in C_{dyn} or one of its predecessors should be considered within the context of this CO. This could be regarded as dynamic binding according to the dynamic connector type. All other classes collected in this phase are possible views of RC_{dyn}.

[5]**Rationale**: This is the two-level dynamic binding.

This VO is internally cached in a transient attribute of the CO, in order to provide the same VO if the same RO is lifted again in the context of the CO *and* if this VO conforms to the required static type. Note, that lifting the same RO to a different VC_{stat} may very well be sound, because one RO may play several roles even within the same CO.

Initialization

When a VO comes into being, i.e., its base RO is lifted within a given connector *for the first time*, its added attributes, if any, are still uninitialized. The **accept** function within the definition of a view class initializes the added attributes of a VO (see lines 15–18 in Fig. 10.7 for an example). Invoking accept is done automatically by the Lua/P interpreter. Accept functions are written by the programmer just like creation methods/constructors, with the only difference, that accept operates on a VO whose used and filtered attributes are already available from the underlying RO and the references `ro` and `co` have been set. Only the VO's added attributes need to be initialized.

Lowering

The interpreter automatically converts a VO to an RO, — the **lowering** operation — every time a VO is passed outside the context of a connector, e.g., as an argument to a workbench request involving a different tool. A VO–to–RO conversion is equivalent to evaluating the expression `vo.ro`.

10.2.6 Tool integration with DVCs

Finally, let us briefly summarize the tool integration process in the presence of DVCs. As far as the tool definition is concerned, the requirement holds that the tool must be developed against an expected interface where some middleware can be plugged in that encapsulates any underlying data storage. Let us assume, that the interface of all persistent objects (the tool model) is defined by means of CORBA IDL or Java interfaces. These interfaces are part of the deliverable tool which should preferably be pre-compiled and be pre-linked *without* an implementation of these interfaces. The first step in integrating such a tool consists in providing a library of proxy classes that encapsulate all persistent objects and in addition to delegating access requests from the tool via middleware to the data storage, also allow to register observers for change notifications from the middleware. This step can be automated by tools such as an IDL compiler. In this vein, we provide a generator for PIROL proxies (cf. Fig. 10.11). Deploying a tool is then a matter of furnishing also the set of generated proxy classes for the chosen middleware.

The requirements so far implement what we call "physical pluggability". A tool that is developed according to these rules can be used with different implementations of middleware (MSG or CORBA or ...) and thus with different repositories or other data–stores, *provided* the structure of proxy classes conforms to the concrete repository model, which is very unlikely. In order to reconcile logical incompatibilities between the structure of the proxy classes

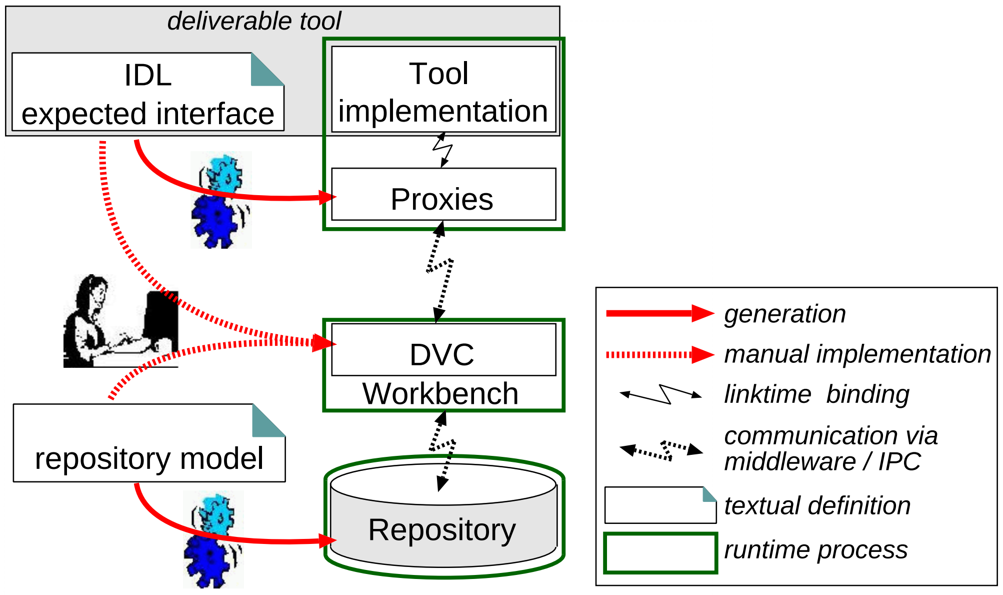


Figure 10.11: Elements and tasks of integrating a tool

and the concrete repository model, the integration of tools now includes an additional step. A dynamic view connector has to be developed (cf. Fig. 10.11), which is the only place in the system that knows about both the tool model and the repository model. The connector creates a specialized view of the repository, or, a "virtual repository", making the tool 'believe' it would operate on a repository, whose schema is identical to its expected interface. The tool will, however, never see objects of the real repository model, but only virtual objects, which simulate the expected structure and behavior on top of the repository model. Implementation of a dynamic view connector requires hand crafting, but this is confined to one module, and from todays experience we expect this to be a task of low effort.

10.3 DVCs interacting with other concerns

10.3.1 Meta Modeling

Discussion ▽

Dynamic View Connectors enhance meta modeling by introducing a second model level. Note, that DVCs are in fact limited to *one* additional level, because each VO must actually refer to an RO as its basis, i.e., neither VOs nor DVCs can be stacked. But already these two levels permit a powerful decoupling between the models of the repository and its tools. Some integration tasks could not easily be implemented if all tools were forced to be based on one common meta model. The new capabilities relate to a-posteriori integration of tools and environment evolution. For both scenarios, Dynamic View Connectors are able to bridge the structural mismatch between rather different models at repository and tool level.

Tool integration [← 10.2.6 on the preceding page]
Evolution [12→]

Also, Dynamic View Connectors add a new structuring concept: connectors encapsulate the objects handled by one tool instance into a *module*, that again is instantiable. Each connector instance (CO) defines a *context* which is populated by VOs.

On the other hand, VOs were designed as to resemble the structures of ROs in Lua/P as closely as possible. View classes may define the same types of attributes, the same style of methods, creation methods and inheritance. This was guided by the goal to make DVCs transparent for tools, i.e., to let VOs appear just like ROs.

10.3.2 Persistence

Implementation ▽

COs implement VOs[←10.2.1] *using hashtables* [←1.1.1]

Transparency of VOs requires persistent storage of VO attributes. This is implemented using the said hashtable in class `CO`. To be precise, this is not just one hashtable but a set of differently typed tuple–lists. The choice to use tuple–lists is motivated simply by the endeavor to manage without introducing a new type constructor Hashtable. While several optimizations of this encoding suggest themselves, measurements would be required in order to judge, whether this extra effort would pay off. Each list for a given resource type `X` has the shape

```
x_resources: List{
    ro: ANY,
    key: String,
    x_val: X
}
```

Such a list exists for each of string, integer, boolean, term and object. This allows to directly map view attributes of the given types, covering all simple and term types. If a view class wants to add a list attribute, this list has to be encoded differently. The implementation depends on the element type of the list.

basic types: Since lists of `String` and `Integer` are encoded as terms of type `STRING_LIST` and `INT_LIST` respectively, added lists of these types are stored in the `term_resources` list.

object types: These are implemented using the `object_resources` list and an auxiliary RO of type `LIST_OBJECT`. This class has only one feature: a list

elements: List(ANY)

This indirection is hidden by the access functions that are built for such and attribute.

tuple types: These are not implemented for VOs. In fact, defining a Lua/P resources–list in such a generic way that it could contain tuple lists of arbitrary tuple types would introduce considerable overhead in PCTE. The mappings used for tuple–lists of RO classes are not available for virtual classes, because no PCTE schemas are generated from the definitions of virtual classes.

> Tuple lists were introduced mainly for reasons of optimization. Since such optimization is not available for virtual classes, motivation is low or inexistent to add tuple lists in a virtual class.

Another difference between repository classes and view classes concerns the visibility of names. Each repository class name is also a PCTE type and since the current implementation does not use SDS prefixes in type names, class names must be globally unique — ROCM packages do not define name spaces. Not so for view classes. View objects are never stored in the repository as such, they only live within the workbench. So, for view classes it was easy, to let each connector class define a name space for all contained view classes. As a consequence, view class methods, e.g., are defined with qualified class names like in:

Schema definition sets [←2.2.1]

function MY_CONN.MY_VCLASS:my_method() . . .

10.3.3 Granularity

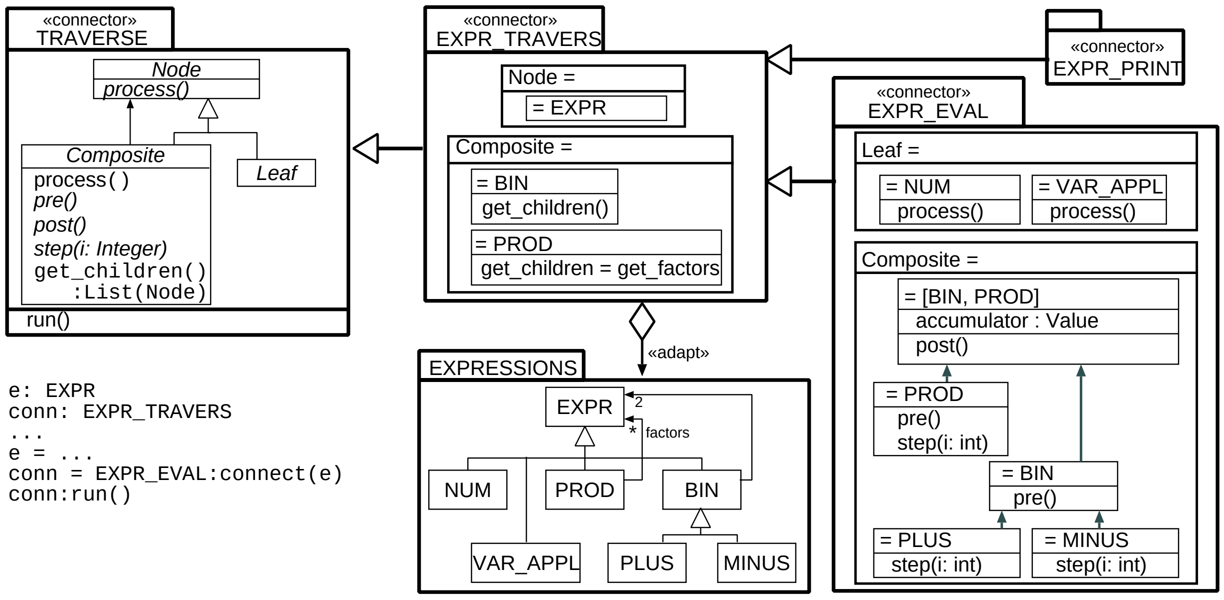
Discussion ▽

DVCs and the issue of granularity are by and large orthogonal. Except for tuple lists, virtual classes may have the same structure as repository classes. However, the internals of terms can currently not be re-mapped by a DVC. This might very well be a useful feature. Consider the example of abstract syntax trees (AST) encoded as terms. E.g., in the ESPRESS project where Pizza was used instead of Lua/P term types, a concept of *annotation layers* was developed, which in a generic way allowed to attach annotations to nodes of the AST [BGHHm98]. Using Pizza, a great deal of polymorphism was used to attach arbitrary types of annotations to arbitrary nodes and explicit retrieval functions were needed to access these annotations while hiding other annotations that were not needed in a specific context. If automatic mappings in the style of DVCs were used to also enrich term types, different tools could in fact work with different refinements of the same grammar simultaneously. While many concepts of DVCs could be extended to also cover term types, the implementation of such a capability would probably cost an effort in the same order of magnitude as implementing DVCs as they are, because differences at the detail level would probably prohibit reuse of any code of the DVC implementation. But this is little more than speculation.

When linking up with the experience from traversing term structures, an interesting combination can be thought of: In a similar vein as higher order functions allow to express traversal separately from structure, also DVCs can be used to implement reusable traversals in places where regular object-oriented design would suggest the cumbersome visitor pattern [GHJV95].

Applications ▽

Experiments have been carried out in separating a structure definition and behavior for a simple expression language just like Fig. 3.1. In this experiment, repository classes were used for the structure definition without further methods (package `EXPRESSIONs` in Fig. 10.12). An abstract DVC was developed as a generic traversal implementation (package `TRAVERSE` in Fig. 10.12). This DVC was bound to the `EXPRESSION` package by extending it to `EXPR_TRAVERSE`.



```
e: EXPR
conn: EXPR_TRAVERS
...
e = ...
conn = EXPR_EVAL:connect(e)
conn:run()
```

Figure 10.12: Modular implementation of functions over composite structures

Finally, concrete DVCs (EXPR_PRINT and EXPR_EVAL) implemented behavior for pretty printing and evaluating expressions. These concrete DVCs inherited from the generic traversal DVC and thus had to fill in only small specific pieces of code.

As an aside, this experiment demonstrated, that a DVC can also be useful without integrating an external tool into the environment, rather, the behavior that a DVC implements may also be seen as a tool that resides within the workbench.

A more comprehensive experiment is needed to explore the combination of DVC based traversals for medium grained objects while descending into the very fine grained items using higher order functions. Technically, this is definitely possible, the quality of the code that results from this approach should also be good in terms of decoupling, understandability and maintainability. In praxis this is yet to be proven.

10.3.4 Behavior

Discussion
▽

By the capability for automatic lifting, DVCs introduce a new behavior aspect: VOs are usually created implicitly. The special method accept has already been introduced, that takes over a responsibility that is similar to creation methods for ROs. The main differences are: accept is called implicitly (precluding any parameters to be passed to this method), on the other hand, accept is called on a partially initialized object, since the new VO already has its references vo.ro and vo.co set, and thus attributes from the uses clause are already available.

New Feature
▽

Creation methods
[←4]

Another technique is needed for creating a VO for which no RO exists yet. Although view classes have creation methods just like repository classes, this creation needs more context information. First, a connector class is needed to resolve the view class name. Second, a connector instance is needed to setup the `vo.co` reference. The RO can be created according to the view class' roclass declaration. So, in Lua/P the syntax for creating a VO including creation of the underlying RO is:

my_vo = my_co.MY_VCLASS:my_creation_method()

The connector class is inferred as the dynamic type of the instance `my_co`. It is the responsibility of the VO creation method, to invoke a proper creation method on its RO sub–object. This can be compared to invoking `super` within a Java constructor, giving a hint at the equivalence of the role–of relation between VOs and ROs and inheritance.

Delegation

New Feature ▽

Another even more decisive issue comes into focus regarding method calls that are delegated from a VO to an RO method. Consider self calls of an RO that are invoked within a delegated call from a VO. It is important to note, that such calls maintain the original self, i.e., the RO method is actually called with a VO target. The reason behind this lies in the possibility to override RO methods. Consider a template method `RC:tmpl` that calls a hook method `RC:hook`. In a view class `VC` that imports both methods (keyword uses), it should be possible to change the behavior of `tmpl` by redefining `hook`. Redefinition is done by implementing `VC:hook`. Given a `vo: VC`, a call `vo:tmpl()` uses the correct `VC:hook()`, if and only if during the execution of `RC:tmpl` the implicit parameter `self` is still bound to the VO. Only so, dynamic dispatch is able to invoke `VC:hook` (which [HOT97] call the "right outcome"). Care must be taken that — aside from this possibility of invoking overriding VO methods — `self` within `RC:tmpl` behaves exactly as `self.ro` would. All `RC` features are visible using their original name (before a possible renaming in `VC`). Overriding works only for methods that appear in the uses clause. Otherwise, methods of the same name are considered unrelated. This is to protect `VC` from methods that are included in `RC` after implementing `VC`, e.g.

In todays experience, complex behavior concerning VOs mostly involves calls between different VO methods. In this case, *smart lifting* already provides the desired dynamic dispatch. Overriding hook methods from RO classes is, however, an important technique, if behavior from RO classes is to be re-used in view classes.

10.3.5 Exception handling

Discussion ▽

Exceptions can be thrown in methods of repository classes as well as in virtual classes. So, exception handling and DVCs are mostly orthogonal. DVCs just add a few exception types, that can be thrown during lifting: `NoMatchingViewClass` and `LiftingAmbigous`. A fully elaborated type system would give warnings in advance regarding connector definitions that introduce any potential lifting problems.

Static type checking [14.2.3→]

10.3.6 Integrity

Change propagation

An important reason for providing language support for virtual repositories rather than hand-coding adapters in the tool relates to one mechanism for data integrity: *change propagation.*

Implementation ▽

In order for VOs to be observable concerning any attribute change, VOs must be tightly integrated with this mechanism. In PIROL's architecture, where connectors are interpreted in the workbench the workbench has all necessary information for tracking changes from a VO to the underlying RO and — more intricate — to track changes from an RO to all VOs that are views of this RO. These fine points had to be observed:

- Attributes may be renamed in view classes, so translations in both directions are needed.
- Lists may be filtered leading to different sets of indices.

The latter issue is solved by recording both indices with each element of a VO list: the local index and the index in the underlying repository list. In fact, one direction of the translation — VO–to–RO — is already needed for implementing the regular operations for VO list manipulation.

Computing changes in the VO list when the RO list has changed is not so straight forward. Firstly, given a RO list index and derived from this the PCTE link index, it must be determined, if and where (at which index position) this effects the VO list. Also for inserting–operations, the filter predicate has to be evaluated for the new element, in order to determine, whether it should be visible in this view. Taking into account, that any set of indices — including the actual PCTE indices — may need to be reorganized when inserting elements in the middle of the list, and also considering, that each RO list must know all VO lists, that are views of it, there is a lot of administrative information to be maintained internally.[6] While this list implementation is tangled code, belonging to very different concerns, the mechanism of change propagation is cleanly separated into three sub–issues

Trigger–deliver–react [←8.2.2]

1. The workbench (especially, the list implementation that has just been discussed) is responsible for computing dependencies and generates `changed` messages for every direct and indirect modification.
2. The MSG server is responsible for delivering these messages to all interested parties.
3. Each tool is responsible for properly reacting to an incoming `changed` message.

The very dense implementation in the workbench seems to be typical of infrastructure implementations at the very core of an environment. This high–effort implementation pays off, however, because it relieves all tools of the obligation to implement any of these translations.

[6]The implementation of object lists, in its three representations PCTE, RO list, VO list, consumes already 8 percent of the total Lua code implementing the workbench.

Predicates

Discussion ▽

Another issue in data integrity concerns both occurrences of predicates. The question is, whether it should be tolerated, that a predicate value might change after evaluation. Currently, it is assumed, that all attributes used in predicates are read–only attributes (e.g., in Java these should be marked `final`). At a closer look this could be differentiated:

- Class predicates are essential for an object to be usable in its role. A change in the value of class predicates would imply dynamic re–typing, which is *very* difficult to allow in a somewhat safe manner.
- If filter predicates in VO lists would evaluate differently over time, this would just mean, that an object might migrate from one list to another. Using change propagation this is actually a safe operation.

The latter, currently being unimplemented, would include the following measures:

- Each filter predicate should be setup as a derived attribute, such that data dependencies are recorded at run–time.
- As a new mechanism, a workbench internal observer mechanism would couple a filtered list to changes of its filter predicate concerning each element of the underlying RO list.
- Each value change in a filter predicate might then trigger an insert or remove event for the corresponding VO list.
- The former event would have to trigger update of the workbench internal representation of the VO list and a `changed` message on the MSG channel.

To–date no examples have been encountered that would justify this effort, that would certainly also have its impact on performance.

Derived and redirected attributes

Implementation ▽

The implementation for VO attributes that are hand coded using the redirect construct is actually a simple enhancement of *derived attributes*. The get function corresponds to a derivation function: both functions retrieve a value while recording data dependencies for later change propagation. The assign function, however, is new in the redirect implementation: it takes away the read–only restriction of derived attributes.[7] Note, that assign functions very smoothly fit into the workbench implementation, where *every* attribute is implemented by a pair of access functions. With assign, it's the ROCM programmer who has the burden of defining a correct assignment strategy, such that the following

[7] During the preparation of this thesis, the read–only restrictions of derived attributes has been overcome by also admitting `assign_xxx` methods in analogy to `derive_xxx` methods. The change was performed by inserting 13 lines of reflective code into the workbench — again demonstrating the power of Lua.

commutation always holds:

vo.at = value
⇝ value == vo.at

Guarded attributes

Discussion ▽

Despite all techniques and mechanisms for keeping data consistent, working with different views of shared data comes with the risk of compromising the *semantical integrity* of the shared data, since changes may be performed within a view, that does not see the full shared data model nor its invariants. Lua/P's mechanism of guarded attributes is the place where programmers of the meta model may encode semantical constraints. It has been shown that guards have a place in PIROL's conceptual architecture that sits right between PCTE and the evaluation of methods in the workbench, which guarantees, that regular usage of ROs (without `raw_xxx` functions) cannot bypass a guard.

Guarded attributes [←6.1.2]

Since view definitions operate as an abstraction on top of ROs, integrity constraints as implemented by means of guards apply to repository access independent of the view from which it originates. That is, PIROL's repository model comes with a meta–level exclusively responsible for encoding and maintaining constraints that ensure the semantic integrity of repository data, no matter what views will be defined over that data and how they access it. As an aside, note that changes on the repository caused by the execution of a policy associated with a guarded attribute are propagated to interested tools (views) in the same way as changes directly originating from some tool access.

Conversely, defining attribute guards for VOs is for conceptual reasons not supported. ROs do not know about all of their views[8]. So, the workbench has no chance to guarantee, that all relevant guards are triggered, if view classes could have guards. Guards that are triggered only sometimes are not useful, they cannot protect any invariant. This is not really a significant loss, since guards were introduced in the beginning as a means to coordinate access to shared data, such that no tool may compromise semantic integrity of data that are relevant outside its own context. This simply does not apply for VOs, since the specific facet of a VO that is not already included in an RO, is not shared by different tools. If as an exception to this rule tools share a virtual repository their VOs will already operate on the same abstractions.

10.3.7 Client-server architecture

RO–VO substitutability

New Feature ▽

DVCs have been designed with care to make the difference between ROs and VOs transparent as far as possible. This substitutability needs to be refined when considering data flows between tools and the workbench, because lifting

[8]Change propagation seems to require differently, but at a closer look, only those views are known, that are currently accessed by a tool of the enclosing user session. Change propagation is active only for the VOs involved in these views.

and lowering may be needed at this component boundary. Can the workbench recognize for each incoming request whether it relates to a virtual repository containing VOs (and of which CO)?

This question can be rephrased to: when is lifting and lowering to be performed? In PIROL this is solved rather dynamically, because Lua/P is an interpreted language, which is not transformed by a compiler.

Related Work ▽

In successor models of DVC [Her02b] lifting and lowering are inserted into the code at compile time or at load time at the latest. This can be done due to the availability of complete typing information. This strategy imposes some restrictions:

- There cannot be a common super-type of role objects and base objects. Each type must unambiguously be either a role-type or a base-type.
- No role object may be passed from one connector to another, because that would require lowering and lifting based on run-time values, not static analysis.

This solution was not used for Lua/P because type information is incomplete (local variables are not typed) and is not easily accessible at run-time. At run-time a Lua/P method is just a Lua function, its signature is stored only in persistent meta objects.

We also wanted to gain some experience with an unspecific type `OBJECT`, which is the super-type of all ROs and VOs.

Lazy lowering

New Feature ▽

One special situation exists, where an experimental concept called *lazy lowering* is quite helpful: Class `WORKBENCH` provides a general–purpose clipboard

selected_objects: List(OBJECT)

with `OBJECT` denoting the super-type of ROs (class `ANY`), VOs (class `ANY_VO`) and transient objects (class `TRANSIENT`). Any tool may store arbitrary objects in this list for later re-use — this is in particular useful for a copy&paste user interaction style. In case a tool stores a VO in this list, any other tool, not using the same connector, should only see the underlying RO, but for the case this VO is later-on passed back to the same tool, the context information is kept as long as possible. Firstly, it is not a problem to lower the object when passing it to a tool, that does not use the same connector. Secondly, keeping the VO instead of its RO helps to disambiguate in case an RO has more than one VO within the same connector, where passing only RO and CO into the lifting operation would not suffice. The remaining question is: How can the workbench tell when and how to lift an object that has been requested by a tool. Two sub-issues are relevant:

1. How to identify the connector instance to lift to?
2. How to determine the required view class for lifting?

Tools as contexts

Generally spoken, each tool will either operate on one virtual repository as defined by one CO or will not use connectors at all. Thus, the following translation can be used: each tool has associated zero or one CO. Class `TOOL_INSTANCE` maintains an attribute `target_co` where each tool instance stores the CO that defines the virtual repository for its target document.

This instance relation is declared by an association `document_types` of class `TOOL_KIND`. A `DOCUMENT_TYPE` is described — among other properties — by the name of the connector class to be used.

Returning to the instance level, the workbench uses the chain

$$\text{OS process} \longrightarrow \text{MSG client ID} \longrightarrow \text{TOOL\_INSTANCE RO} \longrightarrow \text{target\_co}$$

in order to lift all values to be sent to the tool within the context of the `target_co`. This is to say, the workbench identifies tool instances with CO contexts which helps to keep all lifting and lowering action local to the workbench. A tool may use VOs without an explicit distinction while relying on the workbench to perform lifting and lowering whenever needed.

Generally, lifting requires three input parameters: the base object (RO) the connector (CO) and the *declared type of lifting*. In a single program setting an assignment like

MyVOType myVO = wb.getSelectedObject();

would generate a lifting to type `MyVOType`. Now, the right hand side may be a workbench request while the assignment takes place within a tool. Since we want to keep the lifting operation local to the workbench[9], the requested VO type needs to be passed in the request. This is solved by introducing a field `type_desc` to all request messages. For each request that is issued, e.g., by a Java tool the expected return type is explicitly known. This information is encoded in the `type_desc` field, so that the workbench knows to which view class the object should be lifted.

Grammar MSG [←7.3.2]

Remote VO creation

Currently, `create` messages are implemented slightly differently from other requests: As shown in Sect. 10.3.4 on page 152, creation of a VO requires also a connector instance as its context. `create` requests are assembled in the Java tool library, where the current COID of the tool is also stored. Conceptually, this does, however, not differ from the above strategy, where the workbench manages the tool-to-CO association.

Views and replication

Related Work ▽

View objects do not yet involve replication mechanisms for minimizing remote calls. The approach described in [MMD+99] has conceptual similarities to DVCs while focusing on such optimizations for truly distributed systems. Strategies for transmitting partial views of objects in bulk-messages are also subject of [Bar02]. It is planned to exploit declarative specifications of views, such as

[9]There is no advantage in accessing the RO within a tool, when only the VO is of interest.

DVCs, for transparent optimization, which takes away the burden of explicitly choosing a strategy for parameter transmission.

Discussion ▽

Note, how smooth the idea of contexts can be mapped from OS processes upto DVCs. This is, however, in conflict with a desirable optimization. Sect. 14.1.1 will report on measurements of execution time of PIROL and its tools. These measurements urgently suggest, not to use too many OS processes running Java. Ideally, all Java tools should in fact run in only one JVM. Due to further problems with threads (the GUI library swing is not thread-safe), conceptually separate tools cannot be distinguished at the technical level, if they all run in one JVM. Currently, Java tools cannot make proper use of contexts without sacrificing in terms of performance, or more specifically: increased time for starting a new tool. △

10.3.8 Control Integration

Separate extension

Discussion ▽

Sect. 8.3.1 has shown, how a tool may remotely extend an RO. This is relevant for the aspect of distribution or separation of OS processes. View objects are a far more powerful concept for extending an RO without intrusive modification. VOs and extended proxies are somewhat similar regarding overriding: both elements may override existing methods and in both cases the overriding method is only effective when invoked on the extended or view object. Such overriding is, however, broken for extended proxies when used in a template–and–hook style. Extended proxies can only *forward* to their RO while VOs are implemented with true delegation with late binding of self. Also, the powerful constructs for mapping, which are present for VOs, are not available for extended proxies.

Lifting in distributed control flows

In the previous sub-section we have discussed, which part of the system is responsible for lifting objects that are passed to a tool. `changed` messages could be a problem for the strategies discussed, because in that case the workbench does not know, which tool will receive a given `changed` message. Here, the selective multi cast protocol jumps in without further ado. `changed` messages are issued separately for ROs and VOs and each tool only registers patterns for those object identifiers that it is observing, be they ROIDs or VOIDs. Only those cases, where the value of an observed attribute may either be an RO or a VO reference, things would be difficult again.

The general rule for lifting and lowering is: it must conceptually be performed each time a connector boundary is crossed (leaving lazy lowering aside). These kinds of crossing connector boundaries exist:

1. Invoking a method listed in uses.
2. Accessing an attribute listed in either of uses, filter or redirect.
3. A request from a tool that operates on a virtual repository.

4. Delegating a method to a tool using `tool_execute`, where the tool operates on a virtual repository.

5. Change propagation concerning an attribute that participates in a DVC defined mapping.

Of these, (1) and (4) describe events in the control flow, (2) and (5) relate to data flow, and (3) may by either control or data flow. Note, that (1)-(4) happen explicitly, while (5) is implicit.

10.3.9 Multi User Capability

Discussion ▽

Interestingly, both DVC and multi user capability introduce a concept of *context* which appear orthogonal towards each other. Aside from general considerations about partly invisible documents, no interaction between these two concerns have yet been observed.

10.4 Summary

Dynamic View Connectors are the most advanced feature of Lua/P. They combine a language extension with properties of database views. Both issues will be discussed more in depth in Part III.

Meta modeling is *enhanced transparently* by the additional layer. Connector instances are a *visible extension* which encapsulates the structure of COs. Associating connector instances with tool instances helps to automate context selection.

At the client level, Dynamic View Connectors appear *orthogonal* to most other concerns. This is achieved by distinct *mappings*, which simulate VO properties in *accordance* with RO properties. For some mappings, like consistency between lists and filtered lists, this transparency required considerable effort. Also, mapping added attributes to persistent CO properties has its price at implementation level.

Behavior implementation obtains a *new dimension.* VO–RO delegation introduces object–based inheritance to an otherwise class–based language. Implicit VO creation through lifting adds an initialization issue supported by accept functions.

Lazy lowering significantly increased scattering of VO management throughout the workbench. It even discloses the RO–VO distinction to client code, where a more static approach establishes better transparency. Thus, the experiment concerning lazy lowering has successfully demonstrated, that this feature, though realizable, is not worth its effort.

The general reflective design of PIROL is reflected by new root classes `CONNECTOR` and `ANY_VO`, which implement several universal properties, which subclasses may still adapt by overriding. Allocating such properties in root classes helps to keep the core Lua/P implementation lean. A default uses clause in `ANY_VO` has motivated the introduction of the hide keyword in order to adapt also inherited uses declarations.

Chapter 11

Common Services across Components

Aside from implementing a technical infrastructure, an SEE framework should provide functionality that can be re-used across tools. Considering the five areas of integration according to the ECMA reference model, data and control integration contribute to the technical platform, while presentation, process and framework integration can be seen as dealing with additional services.

Discussion of integration [←8.1.1]

Examples from the latter three areas will be presented here, by which the framework is enriched.

11.1 Tool administration

Package TOOLS [←8.3.1]

When presenting the essential classes of the ROCM package `TOOLS`, class `TOOL` has been introduced as the root of a hierarchy of classes that capture configuration information for tools. The main features of this hierarchie are summarized in Fig. 11.1-11.3.

Three levels of tool attributes can be distinguished: static configuration, launch-time configuration and dynamic properties. At the first level, classes `TOOL_KIND` and `PROCESS_MANAGER` allow to configure the environment by installing tools and configuring their general capabilities. The second level deals with launching tools, which implies to create ROs of type `TOOL_PROCESS` and `TOOL_INSTANCE`, which reflect a physical view (corresponding to OS processes) and a logical view. While these objects are created, they are connected to their defining objects (managers and kinds) and to the enclosing workbench session. Some properties at the third level (especially those of `TOOL_PROCESS`) are purely administrative and help to synchronize the external process with its representative, some properties are tool specific, but some common run-time properties help to define common interactions between tools.

As an example, a simple observer mechanism has been implemented regarding the `selected_objects` list. Each tool that implements the `notify_set-selected` method may register as an observer at any other tool. This has been used to have the focus of one tool automatically follow the selection within another tool. Thus, two tools can be "docked" together. This is useful if two

Class `TOOL_KIND`	
`for_class: String`	Only objects conforming to the class given here, can be passed as target to the tool.
`need_vo: Boolean`	If set to true, only a VO can be used as target.
`document_types : List(DOCUMENT_TYPE)`	All documents of the given document types can be displayed by the tool (target will be the `root_ro`/`root_vo` of the CO).
`instance_class: String`	Name of a subclass of `TOOL_INSTANCE` used for creating instances of this tool.[1]
`process_manager : PROCESS_MANAGER`	Link to the manager that can create processes for tools of this kind
`share_process: Boolean`	Can tools of this kind share a common tool process?
`multi_window: Boolean`	Can one tool of this kind simultaneously display different documents in different windows?
`window_size : RESOURCES.xy`[2]	Default size for new windows of this tool

Class `JAVA_TOOL_KIND` (subclass of `TOOL_KIND`)	
`factoryClass: String`	Name of the Java class that implements this tool kind
`factoryMethod: String`	Name of the method (of the class given by factoryClass), that in a running JVM creates a new tool instance

Class `PROCESS_MANAGER`	
`path: String`	Path of the executable for launching a tool process
`args: List(String)`	Arguments to pass when launching a new process
`tool_container: Boolean`	Are processes containers that can host several instances?
`host: HOST`	Preferred host on which to launch the process

Class `JAVA_PROCESS_MANAGER` (subclass of `PROCESS_MANAGER`)	
`classpath: String`	Classpath where the JVM searches class files
`java_args: List(String)`	Arguments to be passed to the Java virtual machine

Figure 11.1: Attributes for static tool configuration

tools show the same objects but with different details. When both tools are docked, it is easy to edit different properties of the same object in different views. Docking also helps to understand which displayed elements are based on the same object.

Discussion ▽ At a closer look the simple observer mechanism reveals a general pattern how tools can interact although being developed independently. Classes for tool representatives define some very basic properties. Other properties can be added by subclassing, but is important to have a minimal set of properties that all tools

[1]This mimics a type parameter of class `TOOL_KIND`.

[2]This term type defines two-dimensional entities by their x and y coordinates.

Class PROCESS_MANAGER	
processes: TOOL_PROCESS	List of all processes that have been launched by this manager (and are still active)

Class TOOL_PROCESS	
manager: PROCESS_MANAGER	The manager that launched this process
sharable: Boolean	(Setup from TOOL_KIND.share_process) Can this process be shared by several tool instances?
pid: Integer	OS process ID
msg_id: Integer	Identification in the MSG channel
instances: TOOL_INSTANCE	All instances being executed within this process
Class TOOL_INSTANCE	
tool_kind: TOOL_KIND	This tool is an instance of that kind
tool_process: TOOL_PROCESS	This tool runs in that process
session: WORKBENCH	The workbench session, this tool runs in[3]

Figure 11.2: Features for setup during launching

Class TOOL_INSTANCE	
message: String	The current tool status as it should be displayed in the tool's statusline (if available)
target: ANY	The currently displayed object
target_co: CO	The currently displayed document
selected_objects: List(ANY)	Internal selection of this tool[4]
connected_tools : List(TOOL_INSTANCE)	Tools that observe the current selection

Figure 11.3: Dynamic tool properties

share. It is now up-to each tool, to implement, e.g., a `notify_set_selected` method, with can be invoked remotely by a `tool_execute` message. The tool being observed is only responsible for keeping the list of selected objects in the repository up-to-date. The workbench executes the observer mechanism of method `set_selected`. Of course this could also be an attribute guard of `selected_objects`, which would be more flexible as tools wouldn't need to invoke specific methods. Representative objects finally are responsible for delivering the notification to their actual running tool. No harm is done, if the tool doesn't listen to this notification. Everything is optional. Note, how this separation in three responsibilities resembles the trigger-deliver-react chain of change propagation. The difference to the standard observer pattern lies in the distribution and the split identity of tools as a running process and a representative.

Change propagation by trigger–deliver–react [←8.2.2]

[3]This is the reverse of WORKBENCH.active_tools: TOOL_INSTANCE.

[4]This selection should not be confused with the workbench selection that is used as a clipboard for passing objects between tools. This should be solved by a different naming.

11.2 Workbench provided context menu

The previous section was concerned with administrative issues that more or less operate in the background. Those services contributed to PIROL's framework integration. It is now time to turn to presentation integration and a central concept concerning how users interact with multiple tools in an integrated environment. The following dilemma had to be solved:

- Tools are developed independently.
- Tools present ROs in very different ways.
- Users should be supported in building a mental model that is close to the graph of ROs in the repository disregarding the different representations in different tools.
- For this purpose it should be possible to perform certain operations on an object in *any* tool.

This cannot be stressed enough: users should be able to (partially) recognize ROs within their tools and documents in order to understand how different documents relate and what the combined model is like. Working with multiple views is only efficient if the nature of views is perceivable as something secondary that is presented instead of the "real" underlying model. It is the hidden commonality of different views, that must be exposed to users.

An essential step on this road is a minimal kind of uniform handling that is ascribed to an RO, independent of the context in which it is (partially) presented. The idea is, to have a *context menu*, that can be invoked for any RO and from any tool. This context menu should provide the same options throughout all tools.

The above dilemma is solved by implementing those context menus in a strict model-view-control style: tools are only required to *display* a context menu for any RO, which they are displaying. The contents of this context menu will be provided in a uniform way by the *workbench*. Finally, execution of any option that is selected in the context menu is delegated to the workbench that dispatches the request to the responsible unit. The common understanding of workbench and tools concerning context menus is defined by a Lua/P class MENU. Of this class, tools only need to know these properties:

```
Class{MENU;
  inherit=TRANSIENT,
  attributes={
    title : String,
    labels : List(String),
    submenus : List(MENU),
  },
}
-- Invoke a menu option:
function MENU:invoke(index: Integer)
```

From such (nested) descriptions, each tool should be able to construct a menu displaying the labels and sub-menus as given by the composite data structure (sub-menus are of course displayed in their parent menu by writing their title). The tool should expect to receive arbitrary nested structures of `MENU` objects. It need not know anything about the meaning of any of these labels and sub-menus. All it need do, is call `invoke` on the (sub-)menu, from which an option was chosen by the user and pass the number of the chosen label. This `invoke` call will of course go to the workbench, which knows well how to react to this choice.

Internally, the workbench has associated with each `MENU` instance a list of function closures that correspond to the menu's labels. So, all the workbench needs to do, is, invoke the correct function closure.

As all this is extremely generic, the question arises, as to how concrete semantics will ever be filled in into this concept. The answer is twofold. Firstly, the tool requests a menu definition for a specific RO because that's what the context menu is opened upon. Secondly, the workbench has a factory method `create_context_menu(obj: ANY)`, which constructs the context menu according to a project- or even user-specific strategy. Each specific operation may be constructed by a subclass of `MENU` that contains the semantics being sought after.

Currently, operations of the following categories are implemented in the default context menu:

- Adding objects to the workbench selection. Also clearing the selection and printing its contents to the console are supported.
- Launching a tool with the current object as its target.
- Sending an object to a tool that is already running. This should usually have the semantics of retargeting the tool or by some other means making the object visible within the tool.
- Open a document that is associated with the object.
- If the object is a document or a VO, a sub-menu is opened for the underlying RO which recursively supports all operations being listed.
- The observer mechanism presented above can be initiated by connecting the current tool as an observer of any running tool.
- Versioning operations where intended here, but since versioning implementation is not functional nothing can be invoked from the context menu.
- A simple workflow mechanism can be triggered by a sub-menu (see below).
- For exploring the repository a few more operations for printout are implemented (print description, print ACL etc.)

This is the current state of implementation and should give an idea of what kind of operations can be uniformly supported for (almost) all objects. Some of these operations will be presented with some more details below. Note, that integrating any new operation, which is by some means invocable from Lua/P, is a matter of adding few lines of Lua/P code to some class in the `MENU` inheritance hierarchy.

11.2.1 Determination of available tools

Class `TOOL_POOL` [←8.3.1]

All services related to launching tools are based on the list of available tools as implemented by the `TOOL_POOL` attribute of each `WORKBENCH` instance. This is the place where tools are registered for use by a given user. All tools contained in this list implement a method `can_handle`. The workbench calls this method with a given target RO as parameter in order to determine the tool's suitability for the context menu of that object. In the standard case, `can_handle` simply inspects the type of the target object and checks conformance to the type required in the tool's `for_class` attribute.

The service of displaying the target RO in a currently running tool relies on the `WORKBENCH`'s list of `active_tools`. It is up to sub-classes of `TOOL_INSTANCE` to decide how to display an RO that is sent to the tool. Alternatives are: completely retargeting the tool, or highlighting the RO in the current view, if the RO is found to be already visible. Also unfolding a visible object might be an option, if it reveals the target RO.

11.2.2 Document handling and creation

When a user wants to see a given object by opening a new tool window, it might not really be the pure RO that he or she is interested in, but very likely a document should be opened, containing the given object, possibly as its root object. This is reflected by two sub-menus titled "Views of *name_of_object*" and "Views containing *name_of_object*". The first sub-menu is generated by retrieving the list of COs associated with the RO via the `cos` list declared in class `ANY_RO`. The second sub-menu retrieves all COs that by their list of `contained_objects` refer to the given RO. This retrieval uses the function `get_origin_list` for navigating a set of PCTE reverse links.

Method `get_origin_list` [←4.1.1]

In both cases for each applicable document (represented by a CO) a menu entry is generated. Each entry has the capability of launching a tool that is associated with the type of the given document. If several such tools are found a sub-menu of these tools is inserted as the menu entry of the document type.

Class `DOCUMENT_TYPE` [←8.3.1]

Another possibility refers to creating a new view of a given RO. Which document types are applicable is determined by examining all `DOCUMENT_TYPES` in the `TOOL_POOL` with respect to their list of `ro_types`. If the given RO conforms to any of the `ro_types` of a `DOCUMENT_TYPE`, this document type is also considered for the sub-menu "Views of *name_of_object*". Note, that this option will only create documents of which the target RO will be the root.

11.2.3 User-to-user communication

This service will need further user interaction. Such interaction has to be implemented by some tool. As an easy solution we could always rely on menu options for launching the responsible tool. However, communicating with other users should be considered a very fundamental service. Currently, no special support is given for this, but it is easy to expose a designated communication tool as to appear at top-level in the context menu. This tool could, e.g., be the message editor MESSED [Kru00]).

11.2.4 Workflow support: document states

When presenting the behavior implemented in ROCM package PROCESSES — responsible for process integration — we saw a finite state machine for document states within a defined workflow. As an example of how the software processes can be integrated at the user interface, the state machine can be controlled from the context menu, too. The "transit" sub-menu presents all state transitions that can be fired on the target RO. This consults the STATE associated with the RO and may, depending on concrete transition GUARDs inspect any property of the target RO.

Class STATE and related [←4.1.1]

In addition to preventing inapplicable transitions to be fired, this sub-menu also acts as kind of an agenda, as it only lists those transitions, which are enabled for the current object, guiding the user to what can be done next. Fig. 13.5 on page 190 will give some illustration to this service.

11.2.5 Lua/P scripting

Many options in the context menu relate to launching tools or sending messages to running tools. The previous paragraphs showed how methods from the meta model (here: package PROCESSES) can be integrated into user interaction. A third option exists for implementing common functionality: Lua/P scripts.

Scripts have been introduced as a mechanisms to extend the behavior of class ACTION. An attribute of this class stores a Lua/P script as string for execution via the document state machine.

Scripts also play the role of simple non-interactive tools: A set of scripts exists for invocation from the console. By sending a file name (and parameters) to the workbench with a special command the workbench can be told to execute any Lua/P script.

While console-scripts are important for simple administrative tasks for which no interactive tool exists, embedding scripts into RO methods is more powerful in terms of integrating common services in a way that is accessible from each tool.

11.3 Common services interacting with other concerns

11.3.1 Meta Modeling

Common services rely on common properties that are attached to all ROs via class ANY_RO. The interface of this class defines several methods that can be applied to all objects no matter which tool is currently being used.

Certain structures (`WORKBENCH`, `TOOL_POOL` etc.) define the configuration for common services.

Finally, class `MENU` and subclasses define the structure how common services are organized, such that each tool can present the applicable services and invoke an action that has been chosen by the user, without the tool needing to know any details about this action.

11.3.2 Persistence

The decision of implementing the context menu by RO-classes and compiling the context menu on demand interferes with the relative low performance of creating a large number of persistent objects. In fact, object creation is among the slowest operations of H–PCTE[5]. This example motivated the introduction of *transient* classes and attributes. Class `MENU` inherits from `TRANSIENT` and thus its objects are never stored to the repository. With persistent objects, the context menu could not efficiently be implemented by objects in the workbench. Transient classes provide the uniform access of ROs and objects needed temporarily for common services. They realize a practical compromise of uniformity and performance.

Transient data [←2.2.1]

Framework integration also relies on persistence in the sense that tool configurations are persistent in a natural way using ROs of classes from the `TOOLS` package.

11.3.3 Granularity

No interaction could be identified how common services and granularity influence each other.

11.3.4 Behavior

The common services presented are partially implemented as methods of the meta model. Some of these methods finally result in launching a tool or sending a message to a running tool, but the integration always happens by methods of the meta model.

Methods of class `MENU` and subclasses implement the selection of those actions that are applicable for a given target object. This design gives maximum flexibility to customizing the menu not only according to an object's class, but also to its state, and to the current environment as defined by class `WORKBENCH` and dependent objects.

Discussion ▽

All services integrated via the context menu only take one parameter: the selected RO. If more parameters where needed, the architecture would have to be extended, because currently the menu is the only means of communication between the user and services that are implemented in the workbench. Plans exist, to also define standard interfaces for parameter dialogs. If the workbench

[5]Sect. 14.1.2 shows, that the difference between creation and other PCTE operations (roughly factor 10) may be shadowed by a much greater difference between accessing objects on segments that are either loaded to the workbench or to the H–PCTE server (roughly factor 100).

remains an invisible background process, also parameter dialogs need to be realized by tools, but the workbench could request specific dialogs by providing a dialog description, which in analogy to menu definitions would be given as a structure of (transient) workbench objects.

This demonstrates, how the model-view-control paradigm can be pursued consequently also for very generic and powerful user interaction.

11.3.5 Exception handling

The issue of common services is, in general, orthogonal to exception handling. Exceptions may occur during execution of a service in they same way, as they occur during any piece of execution in PIROL. Common services on the other hand introduce now specific exceptions.

There is only a pattern, how exceptions can be integrated into the user interface: each tool, by convention, reports errors that occur during a workbench request in its statusline. This statusline also corresponds to an attribute `message` of the tool's representative. In fact, tools use indirect modification for setting their status via the workbench. This allows other tools to inspect the current status of a tool including exceptions.

Indirect modification [←8.2.3]

11.3.6 Integrity

With all other mechanisms in place, there is no way how common services could specifically conflict with integrity issues. The tool docking mechanism applies a hand-coded observer mechanism. Experiments with dynamically attaching a guard to a class showed that it might be possible to unify different trigger mechanisms using dynamic guards.

11.3.7 Client server architecture

The context menu is implemented in a way that translates the model–view–control paradigm into a distributed architecture. Other issues will be discussed together with control integration.

11.3.8 Control integration

In contrast to plugin architectures of current IDEs like eclipse [Ecl], PIROL implements common services in the workbench, i.e., strictly separated from any tool. Not all services are implemented completely in the workbench but some are delegated to a tool. The essence remains: tools remain separate programs while still offering common services in a uniform fashion. Thus in PIROL integration of common services always uses PIROL's distribution mechanism MSG. This reduces the danger of different plugins interfering with each other, because all interaction has to go through MSG and the defined ROCM.

New Feature
▽

A gateway exists, by which PIROL can be integrated with todays most popular distribution technique, the world wide web. A client library exists (in Java), by which repository contents can be accessed as a graph of HTML pages linked by hyper references in the shape of URLs. For this purpose a special protocol `pirol` is defined, which defines access to the `DESCRIPTION` of ROs. The syntax of `pirol` URLs is as follows:

pirol://*roid*[/*index*]

Such a URL uniquely identifies an RO in the current repository[6]. If the optional index is given, this points into the list of `referenced_objects`. This is the indirection, by which object references from within a description text are made safe with respect to referential integrity. For clients of this Java library `pirol` URLs are transparent except that a connection to a PIROL workbench must be established. If this connection exists, references to ROs in the repository are resolved with no visible difference when compared to `http` or `ftp` references. All yield a stream from which the requested content can be read. References within one page can mix all defined URL protocols without further preparation.

Indirect references [←6.2.3]

11.3.9 Multi User Capability

Services for user-to-user communication have been discussed above (Sect. 11.2.3 on page 166).

11.3.10 Dynamic View Connectors

Services related to handling documents were discussed above (Sect. 11.2.2 on page 166). This, of course, includes documents that are implemented as DVC.

11.4 User Interface Management Services

Related Work ▽

Dimensions of integration [←8.1.1]

Literature on tool integration agrees on the importance of presentation integration. Several projects have developed explicit UIMS components that could be attributed to common services. Examples are Chiron in Arcadia [TBC+88] and a "User Interaction Manager" in UniForM [KBPO+95]. Already Wasserman [Was89] recognized that in addition to technical components presentation integration needs guidelines.

Presentation integration for SEEs is very closely related to desktop integration. The open source project KDE [S+00] is a good example, which features a combination of

- Libraries (based on QT) for standard GUI elements.
- Some common services implemented as daemon processes.
- Guidelines and conventions.
- A lightweight messaging facility.

Messaging is implemented by the Desktop Communication Protocol (DCOP [Tib00]), a subsystem which remarkable similarities to MSG. This demonstrates that tight presentation integration requires some support for control integration. Furthermore, a simple solution featuring send (asynchronous) and call

[6]Selecting a remote repository is not yet supported.

(synchronous) messages plus minimal techniques for multicast dispatching suffices as a infrastructure for different components running on the same desktop.

11.5 Summary

Common services are a high-level concern that builds upon several other concerns like meta modeling and behavior and the mechanism of attribute guards. The decisive issues regard the uniformity of these services and their omnipresent availability. Uniformity is achieved by defining important interfaces by ROCM classes in the `TOOL` and `MENU` hierarchies. Class `WORKBENCH` serves as the central agency for many configuration issues and for the run-time context. The architecture of context menus allocates structure and behavior to the workbench. By only implementing presentation and user interaction in tools the context menu still tightly integrates all services of the menu into all tools. Localizing implementation of common services in the workbench helps for a good modularization and makes many services independent from other concerns.

An architecture with narrower interfaces or weaker control integration might have posed greater problems for the flexible provision of common services but PIROL's architecture imposed no such problems. One restriction should be mentioned: a file based environment may allow to integrate a third-party tool for version management. In such a setting, version management is a pluggable, common service. This is conceptually impossible for a repository based environment, unless a versioning tools was available for the chosen repository. But then a smooth and easy integration of such a tool into PIROL seems quite unlikely.

Chapter 12

Evolution of PIROL

The development of PIROL happened in a very evolutionary way. In its current implementation it started as a stepwise extension of Lua. In this course first MSG was integrated, than classes, than PCTE. From this development experience exists with evolving the system without breaking its functionality.

It has always been a central goal of PIROL to provide a common basis for very different incarnations of this generic SEE. The process of customizing PIROL for a concrete project is a form of evolution, too.

A center piece of PIROL is its meta model (the ROCM). Extension and evolution of the ROCM deserve special investigation.

The ultimate goal of an evolvable SEE allows for two dimensions of development:

- Variants are derived from a generic core by customization.
- The core and all variants evolve over time and all changes should harmonize, no maintenance should have to be applied repeatedly and every instance should benefit from all relevant advances.

We have never reached a state, where several installations with very different customizations live in parallel while the core still evolves. We had several students working on different tools (and parts of the ROCM) and simultaneously new releases of the generic core were created, but all these installations contained very little customization. For this reason we have little hard evidence concerning the said two dimensional evolution. Still some concepts can be discussed that should give an idea of PIROL's properties regarding evolution.

Evolution of PIROL's core. The core of PIROL remains a complex piece of software. A perfect design and optimal modularity could never remove this complexity. The source code that is specific to PIROL is in the order of 20,000 lines of code. Half of this is C code, mostly used for integrating existing software into PIROL. The other half is Lua code that implements the workbench. Finally, some 4,000 lines of Lua/P code build the central parts of the ROCM which closely interfaces to the underlying layers. This is a small and dense system and it should be evident, that many locations within this code don't only implement one function but are relevant with respect to many different concerns, as they have been elaborated in this thesis. Evolution of this core also had to pay credit to the heterogenous technique of code written in C, Lua and Lua/P. Evolution without design decay was only possible with major efforts in *refactoring*. Systematic refactoring according to Fowler [Fow99b] was not possible mostly because the code under consideration is not object-oriented code, except for the top layer written in Lua/P. Still the desire to maintain good modularity of this code prevented it from becoming unmanageable. The experience gained during this core development with respect to source code structuring, while being valuable, is difficult to reproduce. This part of the work was probably more of good craftsmanship than pure research. Some interesting after thoughts will be collected in the discussion about Lua's suitability for the tasks at hand.

Discussion on Lua [14.2→]

12.1 Meta model evolution

Above the level of developing the PIROL core, evolution of its meta model — the RO class model (ROCM) — is a very interesting issue, as this meta model defines a semantical framework (on top of the technical framework of the core) by which tools and services are integrated. There seems to be a conflict because integration generally introduces coupling that hinders evolution. A central concept for flexibility in any object-oriented meta model is inheritance. While inheritance can be seen as a purely additive technique — and thus imposes no danger on existing parts — a central problem remains: if different tools operate on different classes within an inheritance hierarchy, which class should be used for object creation and who decides?

12.1.1 Upgrading

New Feature ▽

PIROL adds flexibility concerning the concrete type of objects by a mechanism called *upgrading*. The idea originated in considerations about typical workflows in software development: During the analysis phase the central notions of the application domain should be captured in what we call a *reference glossary*[1]. This glossary starts without any type distinction but only contains notions stored as `ANY_RO`, which for this purpose carry a `DESCRIPTION`. During a later phase some of these notions will be identified with classes of the system under development, others will become methods etc. The idea is now, not to create new objects of type `CLASS`, `METHOD` etc., but to re-use the existing objects (maintaining their description) and to *convert* these objects to a more specific type. This conversion is called *upgrading* and follows a special protocol.

Creation methods [←4]

Similar to creation methods, a class may declare a set of methods as `upgrade` methods. Just like creation methods, an upgrade method combines two tasks: it may implement (additional) object initialization and it performs internal object setup. In contrast to creation methods, an upgrade method has to explicitly call a special function `upgrade()` which performs the actual conversion. The intended usage will be explained by the example shown in Fig. 12.1.

[1]This is where PIROL draws its letter 'R' from.

```
 1 Class {CLASS;
 2    inherit = CLASSIFIER,
 3    creation = init,
 4    upgrade = from_any_ro,
 5    . . .
 6 }
 7 function CLASS:from_any_ro (subsystem)
 8    if self.class ~= ANY_RO then
 9       pirol_error(_ERROR.UpgradeError,
10                   "Must be of exact type ANY_RO in upgrade")
11    end
12    upgrade()
13    self.is_abstract = false
14    subsystem.classifiers:append(self)
            -- the guard of this list also sets the self.subsystem reference
15    return self
16 end
17 local ar = ANY_RO:make("Foo")
18 ar:print()
      ⇝ Foo:ANY_RO<020409_23h-3:175>
19 CLASS:from_any_ro(ar, my_subsystem)
20 ar:print()
      ⇝ Foo:CLASS<020409_23h-3:175>
```

Figure 12.1: An upgrade method and its usage

Lines 7–16 define a method that by line 4 is declared to be an upgrade method. This method is to be invoked (cf. line 19) on its defining class `CLASS` (this is similar to creation methods) and it receives an implicit argument[2]: the object to be upgraded (this is different from creation methods, where no object exists yet). Inside the upgrade method that additional argument appears under the implicit name "self". The method may now examine the type and state of this self object. If this fails to meet some precondition, an `UpgradeError` may be raised and the operation is aborted. Otherwise, a call to the special function `upgrade()` (line 12), which is available only in upgrade methods, performs the actual conversion. After conversion, the newly acquired attributes should be initialized (cf. lines 13,14). As a result, the *same* object that first was of type `ANY_RO` (cf. reply of line 18) is now of type `CLASS` (line 20 and reply).

[2]Precisely spoken: the argument is explicit at the caller but implicit at the callee.

12.2 Evolution interacting with other concerns

12.2.1 Meta Modeling

Upgrading has been introduced as a means to evolve the meta model possibly after some ROs have already been created. With upgrading there is no longer a single decision about an object's type, but object classification may occur in a chain of stepwise refinement.

Two different reason for using upgrading can be thought of:

- Within a complex meta model inheritance is used to relate concepts from different phases or activities. Upgrading propagates an object from one phase to the next. This is how upgrading was originally motivated.
- Inheritance can also be used for some sort of class versioning: If only compatible changes of existing classes are introduced — say during evolution of a tool — a new class version can always be introduced as a sub-class of the existing class. Thus both versions may safely co-exist within a system and for each object upgrading can be performed at any point in time in order to apply the new class version to the given object.

As to the second scenario, PIROL does not support explicit class versions other than by naming convention. Versions can, e.g., be marked by postfixing class names with version numbers, but such conventions are not enforced in any way.

12.2.2 Persistence

We were fortunate with respect to upgrading, because PCTE supports a `convert` function, which exactly matches the requirements of our `upgrade` special function. The rule behind both functions is, that conversion can change the type of an object only within an inheritance hierarchy and only from a super-type to one of its sub-types. This ensures that all contexts that still access the object by its old type are still valid due to sub-type polymorphism.

Upgrading could be the central technique in a broader scenario of schema evolution and class migration[3]. Green and Rashid have identified two central issues in schema evolution for object-oriented databases [GR02]:

1. "Existing objects need to be adapted in some way to conform to the new schema, so that they have the expected fields and methods. This can either be performed:
   (a) by the use of *transparent view wrappers*, which act as they were instances of the corresponding class from the new schema;
   (b) or by physically converting the object into an instance of the new class, which entails dynamic reclassification.

[3]I.e., the technique of objects migrating from one class to another. Sometimes this is called object migration, which in turn can be confused with the migration between different nodes in a distributed environment. That's why we stick to "class migration".

2. It may be necessary for old applications to continue to access the database as if it still conformed to an older schema — that is, backward compatibility may be required.”

They argue that requirements regarding schema evolution significantly differ across applications, which motivates a flexible infrastructure that allows to easily adopt different policies. Item 1.(a) will be discussed below in 12.2.10. 1.(b) is partially solved by upgrading. The restriction is that upgrading does not allow arbitrary type modifications, but only specialization. By this restriction 2. is trivial.

12.2.3 Granularity

A mechanism like upgrading is not suitable for terms, since terms are handled with value semantics thus allowing assignment of a specialized term, which has no side effect across references, since no references to terms exist.

12.2.4 Behavior

Firstly, upgrade methods provide a hook by which arbitrary behavior can be associated with the event of upgrading an object. Furthermore, simple scripts could be written, to upgrade a defined set of objects to a given new class in one bulk operation.

A different path of evolution regards the implementation of Lua/P methods. As long as method signatures are not changed the implementation of a Lua/P package can be reloaded into even a running workbench process. This is valuable during development, because turn-around times with stopping/restarting the workbench are intolerable when only small changes are to be tested. Generally, this may also become important for installing new tools without shutting down a running session.

12.2.5 Exception handling

Upgrading may raise `UpgradeError`s. This is in analogy to `CreationError`s. Conversely, one could imagine designated exceptions to be used as a *trigger* for upgrading. Some databases detect mismatches between expected and actual class versions as exceptions and allow to catch such exceptions in order to perform the necessary conversion transparently on demand. This is not realized in PIROL.

12.2.6 Integrity

During upgrading two levels of integrity can be identified. Syntactical integrity is guaranteed by the restriction to perform reclassification only from a superclass to one of its (indirect) sub-classes. Semantical integrity can only be enforced by domain specific code. In a similar vein as creation methods and attribute guards, upgrade methods allow to implement arbitrary policies for allowing only sound conversion.

12.2.7 Client–server architecture

Decoupling tools from the workbench by means of the middleware MSG enforces clean interfaces. Behind these interfaces evolution is without impact to the other side. Changes in the ROCM can be reflected by newly generating the proxy library. In this scenario, adapting a tool to a new version of the meta model is not further supported but must be carried out manually. Below (12.2.10) we will discuss improved scenarios using DVCs.

12.2.8 Control integration

The consistency between `tool_execute` methods and actual tool implementation that should react to these messages is not supported by special mechanisms.

Upgrading also has impact on the protocol for change propagation. No new message type was introduced for the event of upgrading an object, but this change is broadcast using a regular `changed` message referring to the implicit `class` attribute that every object has. However, usually it is very difficult for a tool to react to an upgrading event in a meaningful way, because languages like Java do not support class migration. It is safe anyway, to continue working with the old proxy, to which the converted object still conforms.

12.2.9 Multi User Capability

During evolution it might be desirable to allow different users to work with different versions of a tool. This might require to also allow different versions of the meta model. PIROL has limited support to configure installation paths on a per-user basis. The structure of the ROCM is of course persistent in the repository. Lua/P methods may, however, be used in different versions by different users.

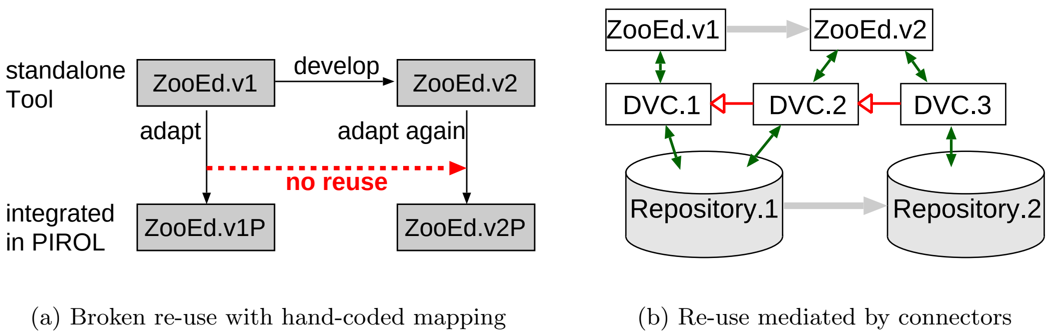
12.2.10 Dynamic View Connectors

So far, we have seen two concepts for flexibility towards the development of the meta model: DVCs and upgrading. Both help for certain situations of evolution. Here we can finally discuss, how these concepts relate to each other and to existing approaches.

Improved Maintainability

We have discussed how to use DVCs to concisely lay down the relationship between a tool and the repository in terms of bridging possible mismatches in their meta models. DVCs confine concerns of structural mapping to explicit lingual units, building a well-defined layer in the overall conceptual architecture of the system. The separation of basic tool functionality from tool integration concerns, introduced in this way, significantly improves modularity with the expected consequences: the system is easier to understand, maintain, and evolve.

For illustration, let us return to our case study: the integration of the UML-diagram editor *ZooEd* [Nor97] into PIROL. This is visualized in Fig. 12.2(a) as



(a) Broken re-use with hand-coded mapping (b) Re-use mediated by connectors

Figure 12.2: Evolution of ZooEd

ZooEd.v1 (the original) and *ZooEd.v1P* (integrated into PIROL). One can envisage an upgrade of the UML diagram editor *ZooEd.v2* that supports presenting the symbols for inner classes à la Java [AG97] as graphically contained within the symbol of the enclosing class. Evidently, the integration of *ZooEd.v2* into a repository could reuse much of the structural mappings of *ZooEd.v1*, since the data model of *ZooEd.v2* would be a “refinement” of *ZooEd.v1*'s model. However, if the adaptation code is hand-coded into the implementation of *ZooEd.v1P*, the structural mapping does not exist in one place and for this reason cannot be reused. On the contrary, when using DVCs, structural mapping is decoupled from the tool's implementation. Hence, the structural mapping concern can be reused with different versions of a tool. Integrating *ZooEd.v2* with DVCs would be performed by defining a new connector class that inherits from the connector class for the old *ZooEd.v1* (cf. Fig. 12.2(b)).

The other way around, changes in the repository model (realized by subclassing and upgrading) can be absorbed by modifications in the DVCs such that tools can continue to operate without being modified. For existing tools two alternatives exist: (a) they may carry on working with the same objects, that by means of polymorphism still match the tool's interface; (b) the connector may be changed to reflect the new classes. As an example of stepwise evolution consider the model of our advanced UML–tool that supports inner classes as graphical containment. Here `Class` has an additional attribute, called `inner_classes`. As long as a corresponding feature does not exist in the repository model, the connector that integrates the tool with the repository would implement `inner_classes` as an added attribute. Later, we might introduce `JAVA_CLASS inherit CLASS` with an attribute `inner_classes: List(JAVA_CLASS)` into the repository model (upgrading existing objects to the new class) and change `inner_classes` to become a used attribute. For the diagram editor nothing will change. But, now any other tool, e.g., a source code editor can share the information about inner classes after a proper modification of its connector.

Thus in the course of maintaining the overall system, DVCs can be used to control which changes should remain local to a tool or the repository and which information should be sharable between tools, giving the maintainer full flexibility with little effort.

Combination, Evolution and Adaptation of Schemas

Related Work ▽

Different approaches have been developed to deal with the fact that the first design of a data model for a system will usually not service all its possible applications later on. The goals have been *flexible deployment*, *evolvable schemas* and *runtime configurability*. When to apply one or the other technique greatly depends on the kind of system and the importance of aspects like persistence, pluggability etc.

Database views [16.1→]

The development of object-oriented databases (and especially repositories for SEEs) has brought about several techniques that allow manipulation of schemas. View mechanisms, if dealing with persistent views, allow to assemble or *combine* the actual schema from different partial views. When compiling the single view definitions into the system, they are merged, resulting in one central schema of which all existing views are simply subsets (cf. e.g., [Gar87]). This technique decouples the components of a system to a certain degree. It is crucial that a technique from this category should support resolution of conflicts between different views (name clashes and mismatching types). It is also necessary to control, not only by naming, which properties are to be shared between views, in case equal names are introduced unintentionally or corresponding properties are introduced by different names. Note the similarity of view techniques for databases with the general-purpose programming technique of Subject Oriented Programming [HO93]. Such view techniques provide, however, no (dynamic) instantiability of views, i.e., they cannot use view objects to model dynamic roles (in the sense of, e.g., [RS91]) of their base object. This also entails the danger of allocating huge database objects, of which only a small slice is ever used, as each object may need to carry all attributes that are possibly used by a view of the object, even if that view is never needed for that object.

Aspect-oriented software development [16.2→]

Roles [8.1.2→]

Using DVCs, PIROL applies a view mechanism that supports dynamical *adaptation* rather than static combination of schemas. DVCs not only decouple the system's components (including all needed conflict resolutions) but also allow to dynamically apply a view definition to a set of base objects. Even multiple views of the same type may thus be created at runtime. Consider the example of different class diagrams sharing the same class (in one diagram it is defined, in another it is used as an imported class): both diagrams display a symbol for the same class, however, at different positions. With DVCs it is no problem to multiply add the `position` attribute (with different values at runtime) to the same base object (resulting in distinct view objects). Many other approaches fail in this situation.

Regarding the evolution of a system after initial deployment, two topics are discussed: schema evolution and class migration. From the wide field of *schema evolution*, we only consider one mechanism, that is interesting in terms of dynamic flexibility: the ability to attach *transformer* functions to database types as, e.g., in [Obj98]. Transformers are client functions (i.e., they are provided by the developers responsible for schema evolution) that are invoked when an object is accessed that belongs to a type for which a new version is installed into the database. At this point the DBMS is responsible for technically upgrading the object to the new type while passing the control to the transformer function (if one is provided) in order to re-initialize the object as to put it into a consistent state with respect to its new type. This technique is very similar to DVCs' accept function. However, schema evolution generally uses in–place modification of the one schema, while view application leaves the central schema untouched. This is important when considering reusability of partial schemas across installations.

Class migration (cf. [Su91, WdJS95]), on the other hand, has been discussed as a *controlled form of schema evolution*: new types may be introduced as subtypes of existing types. Then each existing object may at any time be upgraded to the new type (move to the new class), because this conversion is now invoked explicitly as opposed to the automatic transformation in the course of regular schema evolution. Another application of class migration is *late classification* as introduced in this chapter. Both cases employ the same technique, only the intention differs. In PIROL class migration is supported by the *upgrading* mechanism. Note the analogy between the accept method in the context of dynamic lifting and upgrade methods within class migration: both allow to transform and re–initialize an object as it enters a new context and is enriched with further properties.

Discussion ▽

We made the experience that extending classes as intrusive modification and class adaptation as a dynamically, multiply applicable mechanism — although being quite similar techniques — very well supplement each other. A careful analysis is needed to select the best means for each single case. Generally subclassing is more restricted because adaptations through subclassing are strictly linear: Given a class A with subclasses B and C, an A object may only be upgraded to *either* B *or* C and once this upgrade has taken place, it cannot be reversed and the object may never acquire a type that is not a subtype of its current type. Hence, class migration is restricted to linear sequences of type refinement. It is appropriate if and only if the newly typed object completely replaces the old one. If the type change is only valid in a specific context and if similar adaptations starting from the same base object are to be expected, an intrusive change of the base object is problematic. In those cases *dynamic views* should be preferred over more specialized base objects.

It is also a matter of *sharing* information, whether the base model of a database should be modified or adapting views should supplement it. Also the domain of an extension should be considered: are all objects of a type to be extended (use evolution/migration), or is only a small subset involved (use adaptation). Matters of *cardinality* may also require instantiable views: it may be difficult to have one base object play different roles within the same context and it is impossible to have the same base object play the same role within different contexts with different context dependent properties, unless views are additive on the instance level rather then being manipulations of base types.

The combination of controlled class migration (upgrading) and dynamic views (DVC) — both supplied with a mechanism for custom re-initialization (upgrade method resp. accept) — spans a design space, that provides reusability, pluggability and separation of concerns for a wide range of situations in development and evolution of component based systems.

12.2.11 Common services

Evolution of common services is by and large decoupled from tools by the concept of workbench provided context menus. Conversely, integrating the upgrade mechanism into tools in a uniform way, has been considered. This is relevant for the scenario of passing an object from one tool to another, e.g., via the workbench selection. It is, however, unlikely that upgrading can be triggered without further user interaction. In the case of *ZooEd* a sub-menu has been added to the editor-specific menu: "Insert as...". Here the user can choose a class to which the RO in the workbench selection should be upgraded before insertion into the drawing area. Options are `CLASS` and `ZIMOO_CLASS`[4]. This implements the scenario from the beginning of this chapter (Sect. 12.1.1 on page 174), where an unspecific object is refined to a `CLASS` thus linking existing documentation to the class diagram under development.

12.3 Summary

Throughout this thesis, evolvability has been an implicit goal. More precisely, *modularity* has been paid great attention to. This chapter showed how adding just one more technique, upgrading, PIROL turns out to be a system with good evolvability. At a macroscopic perspective, i.e., considering only tools and the meta model as evolvable units, evolvability is mostly achieved by DVCs. This technique adds to the physical decoupling using MSG a dimension of logical pluggability, thus perfecting the decoupling between tools and the workbench.

[4]This special class goes back to the original intention of *ZooEd*: support for hybrid models in terms of discrete and continuous behavior.

Chapter 13

Tools and Supported Activities

The larger part of PIROL provides generic infrastructure for SEEs. This chapter shows the current instantiation of PIROL with respect to the tools that are integrated.

ESPRESS Tools. This current state does not include a tool suite that had formerly been integrated within the ESPRESS project [BGHHm98]. This suite was concerned with editing and analyzing specifications in the μSZ combined notation [Web96] consisting of a statechart [Har87] view and a Z [Spi92] view. Emphasis regarding tool integration was placed on managing a set of analyzers and transformers. A project specific ROCM package was added that contained the dependency management for all the different intermediated formats that were needed to feed the specification through different tools. Control integration was used to hide the invocation of tool chains. Users only requested certain views of the specification and the workbench took care of transparently invoking those tools that were needed to compute a view that was not yet stored in the repository. Also invalidation of views caused by changes to the original document were automatically detected and derived views were re-computed when needed. In this architecture it was crucial to provide the option to launch certain heavy-weight tools — such as the theorem prover Isabelle [Pau94] — only once per session and to feed single computational tasks to the running tool instance. As an interactive tool Statemate [HLN+90] was (partially) integrated as well as XEmacs with a sophisticated mode for μSZ. Also a generic dialog was developed that provided a uniform GUI for all analysis and transformation tools.

Related Work ▽

Few projects approach the task of integrating tools for formal methods and existing commercial CASE tools in all aspects covered by the ESPRESS instantiation of PIROL. The approach of the UniForM Workbench [KBPO+95] is quite similar to our's. Whereas our integration languages Java and Lua/P are object-oriented, UniForM employs a functional language (Haskell) for this task. Also the architecture differs: UniForM shows a greater number of general components, called managers. We implement the functionality of these managers by just one component, the workbench, thus reducing the number of interfaces. Interestingly, the UniForM Workbench, just like PIROL, uses H-PCTE as its repository.

Other approaches like the Concurrency Factory [CGL+94] or AutoFocus [HSSS96] present more homogeneous environments, with most tools written from scratch and less emphasis on an active repository as a component for integration of persistent data and control.

Details about the ESPRESS tool environment can be found in [BGHHm98].

Interactive tools. Beyond the special requirements of ESPRESS the integration of tools into PIROL has a special focus on interactive tools for browsing and editing different views of a software model.

13.1 PON — PIROL Object Navigator

This tool was originally developed by Bertram Stahl in his diploma thesis [Sta98] and has been developed since by the author of this thesis. Basically, PON is a generic browser for repository objects.

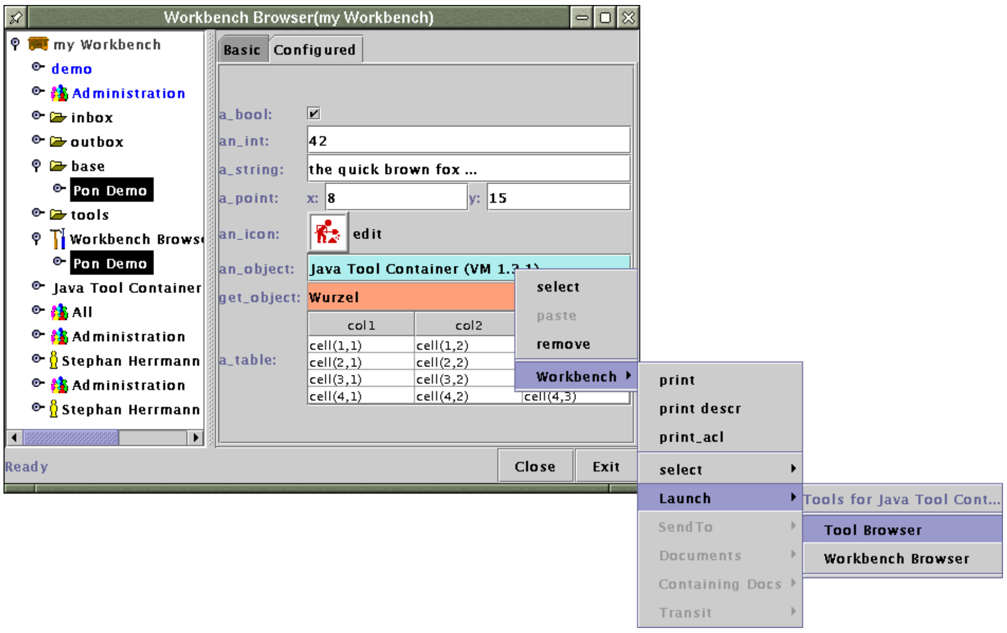
13.1.1 Basic capabilities of PON

The basic idea of PON is very simple: its left window pane shows the hierarchical structure of a set of ROs in a tree view. The right pane shows details about the object that has been selected in the left pane. In contrast to file browsers, from which PON borrows the basic idea, PON's tree view is only a hierarchical *view* of a set of interrelated objects. Within the repository the graph of objects has no structural constraints. As to the question which inter-object-links will be shown in the tree view, configuring PON comes into focus (see the next section).

Hierarchical view of RO graphs

The RO graph may contain cycles. For this reason the tree view has to be filled on demand, i.e., each node is first shown collapsed and only upon user request all its child nodes are retrieved and displayed. As a consequence of cycles, an RO may appear several times within the tree view, i.e., cycles are flattened/expanded to recursive trees. Of course, multiple occurrence of one RO can be confusing for the user. There is a simple way to find out about an object's identity. Objects can be selected in the tree view, which results in highlighting this object and also displaying its details in the detail view. Selecting a multiply visible RO highlights all its occurrences in the tree view. This gives a clue to the user, that none of these occurrences can be edited without effecting also the other locations referring to the same object. As an example see the highlighted object "Pon Demo" in Fig. 13.1.

Tool specific operations in the tree view allow to modify the link structure by common place *cut, copy, insert, replace, remove* actions. Pasting an object onto a node in the tree view requires to specify which link type should be created.



a_bool: ✓
an_int: 42
a_string: the quick brown fox ...
a_point: x: 8 y: 15
an_icon: edit
an_object: Java Tool Container (VM 1.
get_object: Wurzel
a_table:

col1	col2	
cell(1,1)	cell(1,2)	
cell(2,1)	cell(2,2)	
cell(3,1)	cell(3,2)	
cell(4,1)	cell(4,2)	cell(4,3)

Figure 13.1: A snapshot of PON

This is reflected by a sub-menu "*insert as child*" that lists the available link types. Note, that copy-paste operations work across tool instances because the workbench selection is used as a clipboard.

In the hierarchical view objects are listed together that are reachable from their parent node by different association. When moving the mouse over each object, a "tooltip" displays the name of that association.

Discussion ▽

Another style of display has been considered that makes this structure more obvious. A third (middle) view could be constructed that for the currently selected object displays different lists of associated objects on different tabs of the tabbed pane. This would raise clarity of such lists, but handling a three-view window is not trivial because then two views could be used for navigation and it is not clear which object should be displayed in the detail view.

Working on RO details

The detail view of PON provides different property sheets for the currently selected RO. Configuring PON will be discussed in the next section. The general structure of a property sheet is a list of label-value pairs. The label displays the name of an RO attribute. The value component presents its content.

Figure 13.1 shows different field types that are supported by the detail view. Figure 13.2 shows the structure of the corresponding RO class. All fields with white background and the check box are editable. The red field "get_object" presents the result of a Lua/P function and the light blue field "an_object" shows the name of an associated object. Also this field is editable, but users should be informed by the colored background that changes made in this field do not change the link of the current object but the name of the referenced object.

```
Class {PONDEMO;
   inherit = ANY_RO,
   attributes = {
      a_bool : Boolean,
      an_int : Integer,
      a_string: String,
      a_point: RESOURCES.xy,
      an_icon: ICON,
      an_object: ANY_RO,
      a_table : List {
         col1: String, col2: String, col3: String
      }
   }
}
function PONDEMO:get_object() : ANY_RO
```

Figure 13.2: Structure of the demo class `PONDEMO`

All object fields ("an_icon", "an_object" and "get_object") provide the context menu by which the given object can be further examined or manipulated. Field "a_point" demonstrates a structured value of type `RESOURCES.xy`. Both elements are editable. Field "a_table" finally presents a list of string tuples as an editable table. Columns are generated for each tuple component, lines for each element in the list.

Using the detail view, PON supports editing attributes of most types in the repository. Still there is no "save" button or menu option. By convention edited attributes are written to the repository once "enter" is pressed in the corresponding widget.

Remotely controlling PON

A running instance of PON understands these messages:

select Highlight a given object in the hierarchy view and display its details.

notify_set_selected
This has the same effect as `select` and hooks into the inter-tool observer protocol

Tool docking [←11.1]

retarget Use the given object as the new root for the hierarchy view and also select the object.

new_win Open a new window with a given object as root node.

fold Collapse the subtree starting at the given object.

unfold Display the direct child nodes of a given object.

Connecting PON to another tool using the inter-tool observer protocol allows to use PON's detail view as a universal property editor for objects being viewed in other tools. Tool docking avoids additional efforts of using several small tools instead of a big one and helps for an integrated impression of the environment.

13.1.2 Configurability of PON

During the design of PON emphasis has been placed on genericity and configurability. For common tasks no other browser than PON is needed, because PON is not specialized for particular object types or structures. Basically it can display all contents of the repository. The ROCM is a very rich model in order to capture information for different views. Therefore, it would overwhelm the user if all information was displayed for each object. To help focus on relevant information, PON does not apply a reflective technique[1] for displaying *all* information, but configuration objects are used to tell PON which properties to show for given RO classes. For this purpose each `PON` RO — sub-class of `JAVA_TOOL_KIND` — stores a mapping of RO class names to configuration objects `PON_NODE_CONFIG`. The latter allow to configure for each RO class (1) which associations should be used to build the tree view and (2) which attributes should be displayed in the detail views using which widgets.

Class *JAVA_TOOL_KIND* [←11.1]

Fig. 13.1 displays the "Configured" tab of the detail view which uses the information stored in `PON_NODE_CONFIG` objects. There is also a "Basic" tab, that is constructed by reflection. For this purpose, PON queries all attributes of basic types, that are defined for the class of the current RO. A detail view is constructed automatically containing simple text fields for these attributes. According to the concept of providing only selected information in a custom structured order, the "Configured" tab is preferred over the "Basic" tab. The latter can be seen as a fallback for classes for which PON has not been configured.

When working with custom configured instances of PON, it is important that any number of such configurations can be used simultaneously. For each configuration one instance of `PON` (sub-class of `JAVA_TOOL_KIND`) is created and linked to the tools folder. The workbench handles each configuration as a separate tool. As an example, the menu in Fig. 13.1 displays the tools "Tool Browser" and "Workbench Browser", which are two configurations of PON.

13.1.3 Framework design of PON

Configurability recommends a framework-style design, because the set of available widgets for the detail view should be decoupled from the general architecture. The central class in this design is `DetailField` which defines the protocol how PON communicates with any widget that displays one RO attribute. This abstract class already implements quite a bit of functionality like loading, reloading, reacting to `changed` messages and even a context menu. Many of these functions are implemented as template methods that rely on

[1]There is one exception to this rule: the "Basic" tab discussed below.

hook methods to be implemented in sub-classes. Given this general protocol, configuration happens by storing the name of a widget class — sub-class of `DetailField` — for each attribute to be shown. Constructing the detail view in a PON instance is a matter of instantiating widgets by reflective creation using the Java package `java.lang.reflect`.

Related Work ▽

Configurability by reflective creation is also a central technique of Java Beans [Sunb]. In fact, Java Beans have been considered for the design of PON. At the bottom line it was easy enough to rebuild this little infrastructure of Java Beans, of which no further concepts were needed. △

Update recursion [←8.2.3]

PON uses two different strategies for avoiding update recursion. When the enter key is pressed in a detail field, the new value is remembered locally and a `roset` message is sent to the workbench. When a change notification is received, its value is compared against the stored value and in case of equality the notification is recognized as being the acknowledgement of the fields own request. For all operations in the tree view (cut, copy, replace, insert, remove) indirect modification is applied as to ensure consistency with the repository.

Indirect modification [←8.2.3]

13.2 Graphical editors

13.2.1 ZooEd — ZimOO Editor for class diagrams

Dynamic View Connectors [←10]

This tool was the major case study for DVCs . It has been developed independently of PIROL by André Nordwig [Nor97]. ZooEd is intended for creating hybrid models describing discrete and continuous behavioral aspects. ZooEd had first been integrated into PIROL manually, i.e., solely by inserting PIROL specific code into ZooEd's sources. During this development two problems have been observed: the problem of structural mismatches that was discussed in Sect. 10 and the problem of update recursion. The former laid the ground for developing the DVC model, the latter was solved by several flags that made the control flow explicit thus enabling different behavior depending on the original trigger of an action: was the operation initiated by user interaction in ZooEd or by a notification from the PIROL workbench? Integrating ZooEd into PIROL was facilitated by ZooEd's clean design and especially by the consequent application of the command pattern for each relevant user interaction. As "relevant" we consider a self-contained operation as opposed to the many intermediate steps of moving a symbol in the tool's canvas, e.g. While the editor needs to draw the symbol in real time, the repository should only be involved when moving is finished, i.e., upon release of the mouse.

Update recursion [←8.2.3]

Aside from its special support for hybrid models, ZooEd is a classical editor for UML class diagrams.

The distinction between `Class` and `ImportedClass` that served as an example throughout Chap. 10 was in fact introduced only by the integration of ZooEd into PIROL. While it represents a typical kind of mismatch between two meta models, in our specific case neither the repository nor ZooEd draw such a distinction. Sect. 10.2.3 focused on how the DVC uses class predicates to decide which view object should be created for a given RO, it should now be revealed, how this distinction is displayed by ZooEd. Both view classes under

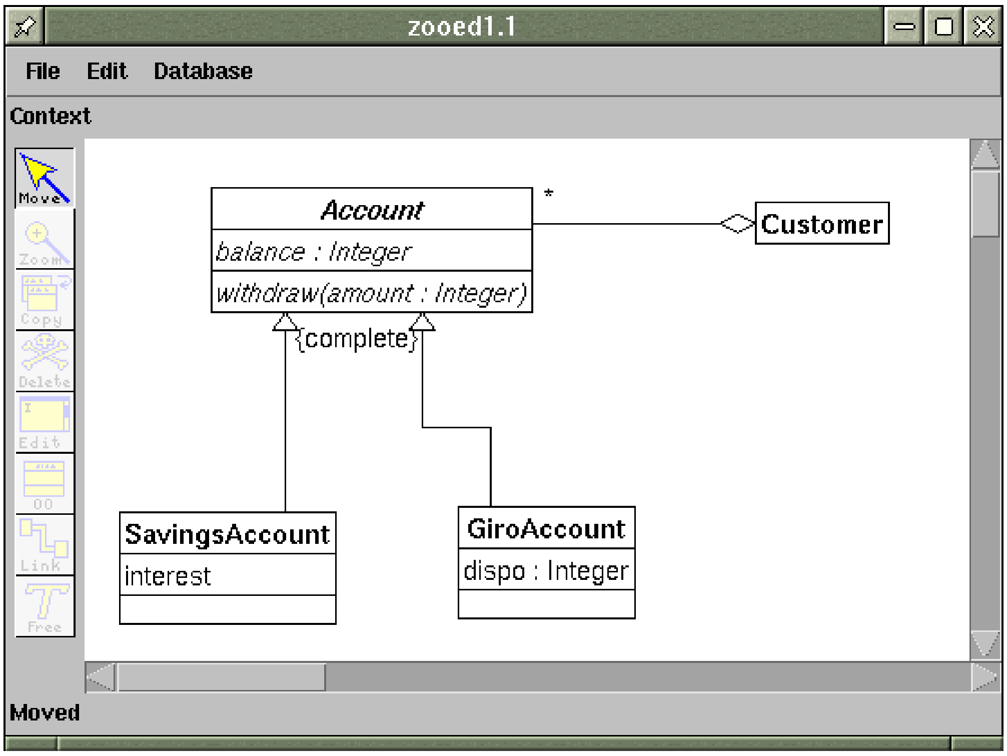


Figure 13.3: A simple class diagram being edited using ZooEd

consideration have `accept` functions associated, which initialize the `collapsed` flag, such that regular `Class` VOs are presented with their details, whereas for an `ImportedClass` the symbol is presented in a collapsed state. Since ZooEd already knows how to interpret the `collapsed` flag, this is all that was needed to display both kinds of classes differently.

User interaction of the integrated ZooEd falls into three categories: (1) ZooEd specific operations are provided unchanged except for a set of operations specific to the Smile simulation environment [KFS95] that have been removed from the integrated ZooEd. Also (2) PIROL specific operations are accessible via the PIROL context menu. This menu is added to ZooEd's context menu if the "shift" key is being pressed while the context menu is opened. It is important that the context menu is still available in its raw ZooEd version, because it is so frequently used that the delay imposed by the workbench context menu can not be tolerated throughout. Finally, (3) a special menu "Insert as..." is provided. This menu has already been presented in Sect. 12.2.11. The combination of inserting an object from the workbench selection and upgrading it to a class usable for ZooEd is specific to the integrated version of ZooEd.

Redefining the "semantics" of ZooEd diagrams

Another experiment concerning DVCs has been carried out, by which ZooEd was turned into an editor for finite state machines. The task was setup like this: given the existing model of state machines as defined by RO classes `STATE`,

Document state machine [←4.1.1]

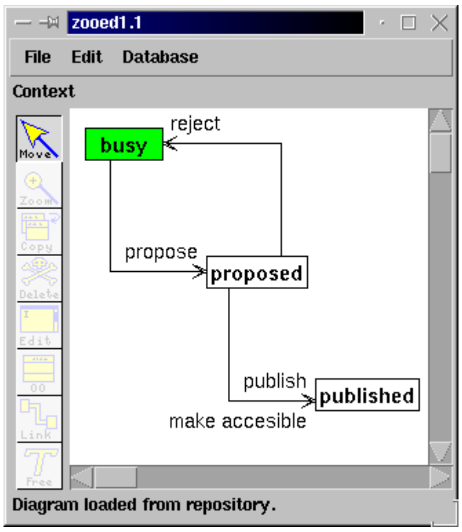


Figure 13.4: Editing a state machine using ZooEd

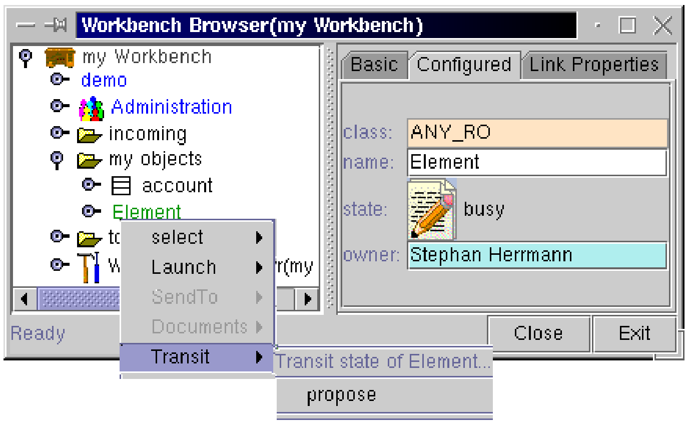


Figure 13.5: Invoking a state transition

`TRANSITION` etc., and without changing ZooEd, an editor should be realized for visually editing state machines.

This task was solved by mere 360 lines of connector code which basically maps classes and relations as seen by ZooEd to states and transitions as seen by the repository. The given number of lines even includes the observer mechanism by which ZooEd is used to visualize execution of the underlying state machine: a target object can be registered such that changing the state of this object (e.g., by use of the transit context menu) moves a highlight in ZooEd to the symbol representing the new active state. Fig. 13.4 shows a state machine being edited using ZooEd. Fig. 13.5 shows a sample invocation of a state transition using the context menu in PON. The green highlight in ZooEd automatically follows the document state of an object that has been registered with this view of the state machine.

Transit menu [←11.2.4]

The experiment shows how a given tool can be used to display and edit very different structures of ROs just by implementing a Dynamic View Connector

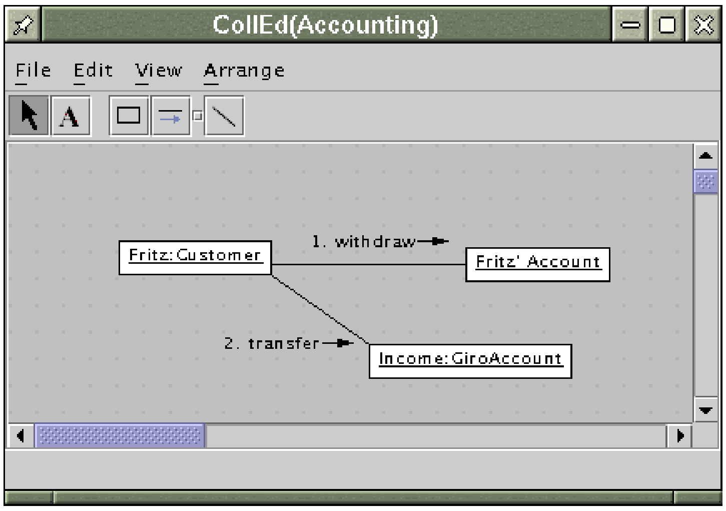


Figure 13.6: A sample instance of GEFTool: Collaboration diagrams

that maps specific RO classes and their relations to the classes and relations expected by the editor. This is well in line with, e.g., the strategy by which the UML specification [BRJ99] decouples the notation (chapter notation guide) from the semantics as defined by the UML meta model. The connection is given by a subsection for each diagram element, which specifies how the graphical element is mapped to elements of the meta model.

13.2.2 GEFTool — Graphical Editor Framework

In his diploma thesis, Burkhard Weber [Web01] integrated the GEF graphical editor framework into PIROL. This framework was selected against JHotDraw and Drawlets. An integration layer was developed using the Mediator and Observer design patterns [GHJV95]. The main task of this layer is to keep proxies for ROs and the internal representation of the editor consistent. This design is currently being improved by Florian Hacker [Hac02].

The goal is a framework by which it will be increasingly easier to create graphical editors for PIROL. While GEF is already quite complex by itself — the price by which powerful graphical capabilities are bought — the integrated version should not add to that complexity.

Example instances of the GEF-PIROL integration are still at an experimental state with functionality far from being complete for any one kind of diagram. Still, in a chain of several attempts of building graphical editors for PIROL (cf. App. B) this framework seems to provide maximum productivity during development. Simple applications can be constructed quite quickly. Some frameworks that share this property fail at more complex requirements unless one deliberately changes the fundamental design. With GEF, more complex requirements, while requiring a deep knowledge of the framework, can usually be implemented within the given design.

13.3 Text editors

Browsers and diagram editors naturally operate on structured data. Editors for HTML contents were the first text editors integrated into PIROL. Three approaches have been taken to HTML editing in PIROL. In the ToolTalk based version of PIROL an integration of *XEmacs* has been implemented. Relying on XEmacs' integration of ToolTalk and the w3 module for HTML display, XEmacs was able to query HTML texts from the repository and display this text with layout. It was easy to include resolution of pirol style URLs to enable browsing HTML pages from the repository.

ToolTalk as alternative to MSG [←7.3]

pirol URLs [←11.3.8]

After replacing ToolTalk with MSG, XEmacs could no longer be integrated with the same ease. Another gateway between XEmacs and PIROL has been built in the ESPRESS project using a Unix pipe between XEmacs and a Java program with workbench access. This seemed, however, impractical for HTML editing. Instead, the pirol URL protocol was integrated in the Java framework provided by the java.net package.

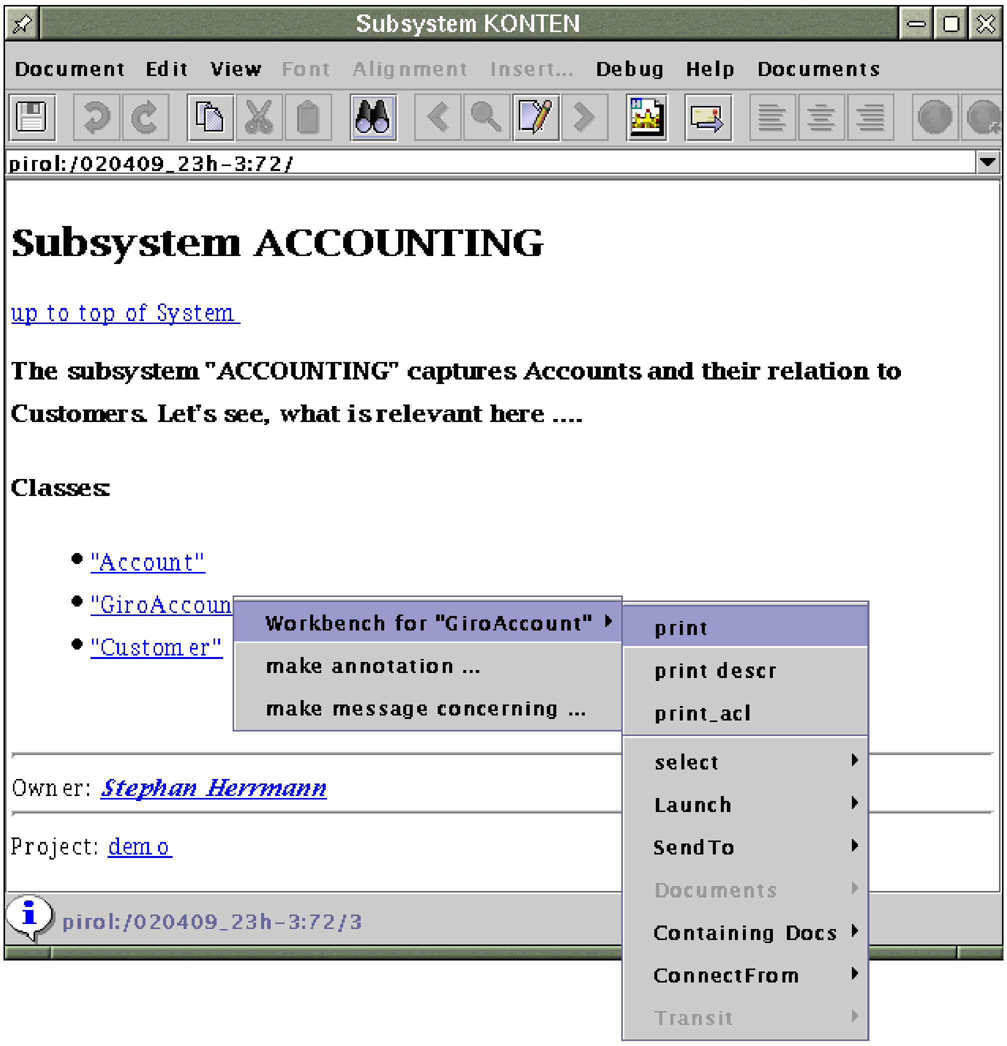
Built on this bridge and on a framework for HTML editors in Java (package javax.swing.text.html) two tools have been developed: *PHED* (PIROL HTML EDitor) was developed by Doris Fähndrich at a time when the editor framework was still quite unstable. The third tool in this sequence is *MESSED*, the Message Editor developed by Ralf Kruber [Kru00], which is to be presented here.

13.3.1 MESSED — Message Editor

MESSED has been developed as an HTML editor with a special focus on user-to-user communication. In this setting HTML pages are understood as either messages or comments, which users exchange by a mail mechanism or as annotations.

Editing HTML

MESSED generally works in one of two modes: *editing* or *browsing*. Both modes present a given HTML page in the same lay-outed way, but behavior differs between both modes. During editing, menus for formatting and insertion are active ("Font", "Alignment", "Insert...") and mouse clicks simply position the cursor. In browsing mode the page can not be modified but hyper links are active instead. In addition to following a hyper link by a left click, this has the effect that pirol links can be used to open the PIROL context menu regarding the referenced object. Thus a hyper link to an RO behaves in the same way as any occurrence of an RO within a PIROL tool does. Fig. 13.7 shows a snapshot of MESSED in browsing mode. The right mouse button has been clicked on the link "GiroAccount". The status line displays the target URL (in this case a pirol URL denoted relative to the current page) and the context menu for the referenced object is presented. The workbench sub-menu should be familiar to the reader by now. Additionally, tool specific options are offered whose intention is to be shown below.


Subsystem KONTEN

Document Edit View Font Alignment Insert... Debug Help Documents

pirol:/020409_23h-3:72/

Subsystem ACCOUNTING

up to top of System

The subsystem "ACCOUNTING" captures Accounts and their relation to Customers. Let's see, what is relevant here ....

Classes:

- "Account"
- "GiroAccoun
- "Customer"

Owner: *Stephan Herrmann*

Project: demo

pirol:/020409_23h-3:72/3

Figure 13.7: MESSED as HTML editor

Tool docking works in both directions for MESSED: MESSED can follow the focus of selected objects within any other tool and vice versa.

Annotating documents

Menu action "make annotation ..." creates a new HTML page as annotation to the selected RO. This is most useful if the context menu was invoked on a non-link area of the current page, where the context menu refers to the currently displayed RO instead of a referenced object (link target).

MESSED reflects the two levels of adding documentation to an RO: opening an RO in MESSED generally means to display the `text` of its `DESCRIPTION`, which is regarded as the HTML view of the RO. Additionally, any number of annotations can be attached to each RO. Note, that attaching an annotation does not require write permissions concerning the RO. Navigating from an object to any of its annotations should definitely be supported by MESSED, too.

Class ANNOTATION [←9.1.1]

This should include some visual clue whether the current page has annotations or not. This additional navigation support is, however, not yet implemented.

Sending messages

Sending a message using MESSED happens similar to creating an annotation, i.e., messages can be created as comment to an existing object. Also, message creation from scratch is possible. Currently, MESSED does not support selection of recipients of a message, but PON has to be used for this purpose by appending recipient links to the message's ENVELOPE object. Integration of address management would be easy, if a simple address book mechanism was included in the ROCM. It would in fact suffice to augment class WORKBENCH by an attribute address_book: GROUP, which would allow to maintain an arbitrary hierarchical directory of groups and persons. A fictitious "add recipient" menu in MESSED would simply present the tree structure of this "address book" for selection. Finally, the "send" button in MESSED sends the current page to all recipients recorded in the associated envelope using the mechanisms presented in Sect. 9.1.4.

Class ENVELOPE [←9.1.1]

13.3.2 pjEdit — Source Code Editor

HTML editors were said to be the first text editors integrated in PIROL. Their integration posed no conceptual problems aside from designing the pirol URL protocol. HTML pages are structured in a way that is already suitable for storing in the repository, given that no other tool needs to know about the internal structure of each page.

The situation changes drastically when moving to program source code. For good discussions on structure oriented versus text oriented editing and some interesting combinations see [Rei95] and [VB00].

A first work on source code editing in PIROL has been made by Jan Peter [Pet00]. The contribution of that thesis was mainly in designing a general meta model for object-oriented programming languages. His example application *OOEd* has been implemented from scratch and stayed at a prototypical state that can not be compared with modern text editors.

In contrast, *pjEdit* (developed by Christian Mattick [Mat02]) is based on the powerful open editor jEdit. A major contribution of pjEdit is a practical approach to maintaining several representations of source code and supporting their consistency during development. In the context of pjEdit these representations are used:

- **PRODUCT**
  ROs from this package still capture the structural information about the software under development. These classes are extended with properties of an abstract class SOURCE_PRODUCER in order to generate SOURCECODE instances.

- **SOURCECODE**
  This ROCM package implements the composite pattern where leaves of

the resulting tree contain concrete pieces of source code. The tree structure represents the syntactic structure of the source code but its nodes are not typed according to the source language grammar. Instead, nodes are named as to reflect this grammar.

- **SourceFragment**
  Java classes exist that build a tree parallel to the `SOURCECODE` tree in the repository. Each `SourceFragment` is associated to a `SOURCECODE` RO and stores editor internal information.

- **Buffer**
  Within jEdit (and pjEdit) editing happens in buffers that contain the actual sequence of characters. `SourceFragment`s refer to positions in the associated buffer and control the layout of the display.

Several mechanisms like observers and queues of invalidation events are used to keep these representations consistent without disturbing the process of editing in undue ways. Some elements can be edited in pjEdit textually and will be parsed by Lua/P methods. Other structural elements are inserted only using indirect modification: macros[2] exist, that create elements in the repository and textual skeletons of these elements are inserted to the editor's buffer in response to the `changed` message from the workbench.

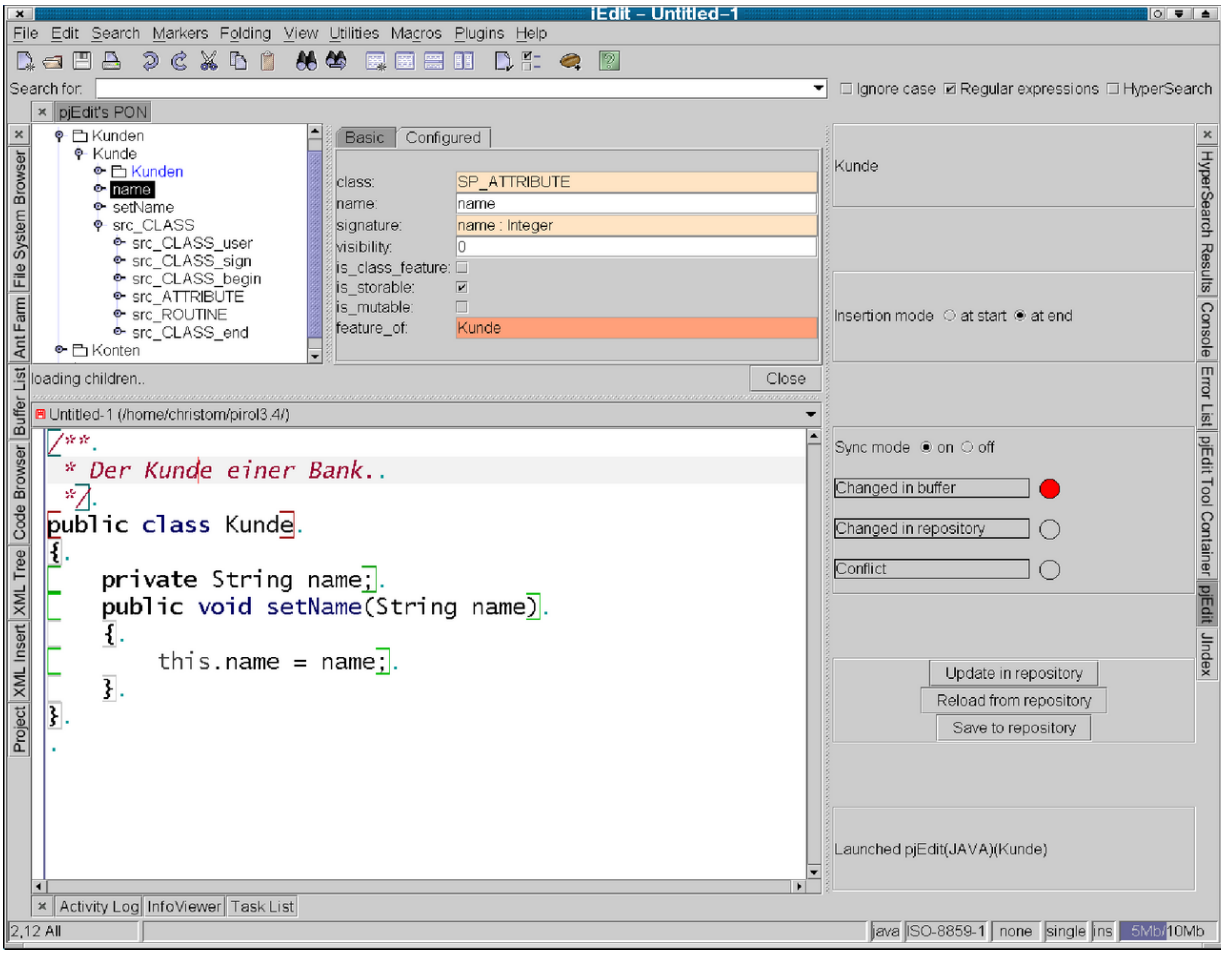
All this results in an editor that is aware of the structure of the source code being edited and its relation to the repository. This can be used in conjunction with other tools as it is demonstrated by a close cooperation with PON. pjEdit can, e.g., easily follow the focus of a specified instance of PON and highlights the text position that corresponds to the RO that is selected in PON. Furthermore, presentation integration was improved by integrating PON as a plugin into the jEdit application. This integration has motivated the current structure of the `TOOLS` package. As a plugin PON has to run in the same JVM as jEdit, which also has to host the pjEdit plugin. This led to the design of `TOOL_PROCESS`es and `TOOL_PROCESS_MANAGER`s.

Fig. 13.8 shows a typical situation of pjEdit containing the following elements: The upper left part of the window shows an integrated instance of PON, which shows the structural view of the class being edited. Below is a source code text buffer. The brackets indicate that the editor is aware of the structure of this text. It has been criticized that these brackets disturb the user and in future versions only the delimiters of the current text block will be shown. At the right hand side additional control elements are shown. In this section several "lamps" signal the need of synchronization between editor and repository. Buttons are provided for performing such updates in different ways.

In his thesis [Mat02] Mattick identifies some problems with this integration. With respect to PIROL the major problems are related to the lack of transactions for grouping several requests of a tool. He gives good examples of possible sources for inconsistencies due to this lack of transactions. Other integration problems relate to the jEdit API and will not be discussed here.

Discussion of tool transactions [←9.1.7]

[2]These macros are written in the BeanShell language and can be bound to specific key shortcuts.



Kunden
Kunde
Kunden
name
setName
src_CLASS
src_CLASS_user
src_CLASS_sign
src_CLASS_begin
src_ATTRIBUTE
src_ROUTINE
src_CLASS_end
Konten

loading children..

class: SP_ATTRIBUTE
name: name
signature: name : Integer
visibility: 0
is_class_feature: ☐
is_storable: ☑
is_mutable: ☐
feature_of: Kunde

Kunde

Insertion mode ○ at start ● at end

Sync mode ● on ○ off
Changed in buffer ●
Changed in repository ○
Conflict ○

Launched pjEdit(JAVA)(Kunde)

Untitled-1 (/home/christom/pirol3.4/)

```
/**
 * Der Kunde einer Bank.
 */
public class Kunde
{
    private String name;
    public void setName(String name)
    {
        this.name = name;
    }
}
```

2,12 All | java | ISO-8859-1 | none | single | ins | 5Mb/10Mb

Figure 13.8: A snapshot of pjEdit

13.4 Gateways to the outside

PIROL is about *integrating* tools. Such tools have to be aware that they live within an environment. Sect. 10.2.6 has discussed the preconditions that a tool must meet in order to be integrated. A realistic approach must acknowledge that there are techniques beyond the specialized integration mechanisms used by the environment. In this section interfaces to two important standards are presented: file systems and the world wide web.

13.4.1 COFS — Conceptual Object File System

Objects versus files and documents [←1.1]

It is a fundamental design decision of PIROL to store artifacts as ROs not files. We have discussed that users should still have the impression of working on documents. In this conflict COs mediate between both worlds and DVCs finally provide a clean encapsulation of this mapping. This works fine for all tools that are aware of PIROL. At least for one category of tools PIROL-awareness is, however, not a realistic assumption: compilers and other sorts of translators.

For all these tools the one and only standard interface for accessing data is the file system. In fact the interface for accessing files as part of the POSIX standard might be considered one of the most valuable standards of information technology.

Discussion ▽

Nowadays, the uniqueness of this standard should be questioned in favor of a superset, as defined by uniform resource locators (URL). URLs unify the access to data via different protocols like `ftp` and `http`. Libraries exist for all relevant programming languages that resolve arbitrary URLs (as absolute references, or relative to a given base URL, just like relative paths in a file system) and provide a stream from which data can be read in the sequel.

Once compilers are able to read their input from arbitrary URLs it should be easy to add new protocols to the respective library and thus feed `pirol` URLs into a compiler. Given that the `pirol` protocol reads directly from the repository this would enable any URL-aware compiler to read data that is assembled on the fly from RO attributes.

Already in [BGHHm98] we have used a URL-like technique of referring to repository contents, which is called model resource locators (MSL) in [BGHHm98].△

The conservative solution would be to use an explicit export function for creating the files needed, e.g., by a compiler. This has several disadvantages.

- Exporting requires an extra user interaction, i.e., before starting a tool in file system world, a PIROL tool must be used to trigger exporting.
- It might be difficult or impossible to know ahead of time all files that need to be exported. Many tools start by reading a root file and from there find many more files that need to be opened, too. These dependencies might or might not be reproducible within PIROL.
- Once files have been exported they introduce redundancy and it is difficult to maintain their validity by keeping track of dependencies between files and ROs.

If exporting is not a preferable option and if flexibility concerning their input is not built-in to compilers, the interface that they expect must be simulated by the environment. The solution is to implement a *virtual file system.*

Technology of `userfs`

In PIROL we used `userfs` [Fit00] as a framework for arbitrary virtual file systems. `userfs` is limited to Linux and its architecture is as follows.

A kernel module registers the new file system type in the Linux kernel and allows to use this type in `mount` system calls. The module does not implement a specific file system but establishes a socket connection to a separate program that runs outside the kernel, i.e., in user space. This separation is motivated by kernel safety: while the kernel module is fairly good tested, the program that will be responsible for a concrete file system should not compromise the kernel in case it contains any bugs. By means of the separation the user space part may even dump core without affecting kernel stability.

User space programs exist for examples like `arcfs` (providing access to files packed in a tar archive) and `ftpfs` (allowing to mount remote file systems via ftp). Some common abstractions for all user programs like *inodes*, *directories* and *files* are put into a C++ library. Library classes define hooks (abstract methods) by which a concrete file system implementation must realize virtual files and directories.

Using a virtual file system requires a special mount command using the `muserfs` utility. This utility starts the user space program and registers the program at the kernel module using the said `mount` system call. Additionally, a mount point (an empty directory in the real file system) must be given. As a result, any file access below the mount point will be redirected within the Linux kernel to the `userfs` module, which will request all information (file attributes and file contents) through the socket connection from the user program. An example invocation might look like this:

```
$ muserfs arcfs /mnt/archive some_tar_file.tar
$ cd /mnt/archive
$ ls
  ⇝ README ...
$ cat README
  ⇝ ...
```

COFS

Using the technology of `userfs` and PIROL, COFS provides file access to conceptual objects (CO). This is how the contents of a COFS mount is determined: Each file is represented by a CO. It has registered an object of type `UNPARSER` that is able to proved the file contents by invoking method `get_text` passing the CO's leading RO as argument. Unparsers exist for C++ headers and C++ source code, which flatten the RO structure to a textual representation. Directories can be created explicitly as a CO of document type "directory". The list `contained_objects` contains all COs that represent files in the directory. It should also be possible to derive the list of contained files (represented by COs) from any relation in the repository. E.g., should the list `SUBSYSTEM.classifiers: List(CLASSIFIER)` setup a directory containing all classes of a given subsystem. This is, however, not implemented.

Discussion ▽ Design and implementation of COFS have again recognized the need for tool transactions in PIROL. Problems with respect to consistency might occur if in the middle of a compiler run (which might be a long running task) ROs are changed that lead to invalidating and recomputing some file contents that is currently used.

Other nice features can be realized using a virtual file system, once the file system is made read/write. One could, e.g., easily observe write access to a `.log` file that a LaTeX run creates in order to pop-up a PIROL window showing the diagnostics parsed from the log file. The basic idea is to translate file operations to RO modifications: creating a file results in appending an object to a list of contained objects, appending data to a file changes a "size" property of a CO representing the file etc. Based on such translations, attribute guards can be employed to use such file events as trigger for arbitrary actions within PIROL, including parsing of the file contents into a structure of ROs.

Status and future of `userfs` and COFS

Discussion ▽

Unfortunately, `userfs` has not been updated for current Linux kernels. For one reason this effects the kernel module, which exists only for 2.2.x kernels. There is another weak spots: the tool for generating (de-)serialization code for the socket connection is highly fragile. For the sake of generating wrapper code for all types used by a program it *parses* all system include files (C header files), which is known to be a extremely difficult task.

Meanwhile, an alternative to `userfs` exists. It was developed by the name `podfuk` and has later been renamed to `uservfs`. This package is derived from the integration of samba support in the midnight-commander file manager.

The transition to `uservfs` has not yet been made, and therefore COFS is currently discontinued. Also, the findings of [Mat02] should be integrated into COFS with respect to synchronizing different data representations. Using the hierarchical structure of `SOURCECODE` ROs it should be significantly easier to create file contents on the fly.

Summarizing, COFS can be evaluated to be an promising demonstration of how the gap between the very different worlds of files and repository objects can be bridged. It just needs to be re-built using newer technology. △

13.4.2 PIROLWEB — PIROL–WWW gateway

We have already seen how PIROL can be integrated with standard web technology by adding a new URL protocol. Unfortunately, this does not work with standard browsers, unless a browser is designed for easily adding more protocols at the client site.

The tool to be presented here, takes a different approach: ROs from the PIROL repository are to be converted to HTML pages which are made available on the web using a servlet. This way, PIROL web content can be viewed by any browser using the `http` protocol.

A framework is currently under development, which should allow easy configuration of a processing chain that starts at a set of ROs, constructs an XML dom tree, processes this tree using an XSL stylesheet and sends the result to the client. This chain is controlled by a servlet. Such a framework significantly resembles a standard content management system (CMS), which in our case is based on an object-oriented repository in contrast to the commonly used relational databases.

13.5 Small Lua/P tools

PIROL is to a large extent a data-centric environment. Attributes are considered public properties of ROs, DVCs define class mappings largely in terms of attribute mappings. PON can be used to browse arbitrary data, provided a suitable configuration exists.

Compared to pure data access, generic support for providing access to the RO *behavior* at the user interface is comparably difficult. The reason for this difference lies in different method signatures which usually require a tighter embedding into a tools user interface.

An early day browser of PIROL had offered the option to execute any defined method on any RO presented by the tool. While selecting a method name from a list that is generated by reflection is not a problem, we found no really useful way of providing the arguments of methods in a generic way. Generic dialogs created via reflection proved more or less impractical.

Creating ROs falls into the same category because it includes a call to a creation method. In many cases, we found that creation of objects is indeed an intrinsic duty of a specialized tool. E.g., ZooEd may create all objects relevant to a class diagram.

Method invocation should be integrated into specific tool GUIs in cases like a "send" button in MESSED for sending a message.

Lua/P provides another option by which data manipulation (e.g., using PON) can be used to drive method invocation. The trick lies in attribute guards that have been used for the following experiments.

An experimental RO class `MAKER` has been developed that realizes the model of a simple RO creation dialog. This class has attributes like `target_name` and `target_class` into which parameters for creation are entered. An attribute guard of the `target_class` attribute triggers creation of a new RO with the given parameters and store this object in another attribute called `target`. A simple configuration of PON for class `MAKER` defines the presentation of the said creation dialog. Entering a class name into this dialog after other parameters have already been set triggers the desired creation.

Another more realistic experiment concerns the list of recipients of a message. This list is declared in Lua/P as:

```
recipients : List {
   agent : AGENT,
   message_work_state : MESSAGE_WORK_STATE,
   is_sent : Boolean,
}
```

Because entering tuples to lists like this is not supported by PON, a method exists, that allows insertion of recipients and initializes the extra tuple components to reasonable values. In order to allow associating new recipients to this list by means of existing techniques, a dummy attribute

```
new_recipient: Transient(AGENT)
```

is added to class `MESSAGE`. This attribute is transient, because it carries no valuable information. An attribute guard, however, is attached to the attribute which intercepts each assignment to the attribute and calls method `add_recipient` according to the above explanation. This dummy attribute could be called an active port, that is an interface were sending a value to triggers an action.

Data driven actions can at a small scale add to the user interface by providing a means to invoke methods or create objects within an otherwise data-centric GUI. This should, however, not be taken to the extreme. User friendly interaction is still best implemented by specialized buttons and dialogs of specialized tools.

13.6 Summary

Some tools presented in this chapter may contribute in one way or other to other concerns. The most obvious example might be MESSED contributing to multi-user capability with respect to user-to-user communication. Of course, COFS contributes to the file-versus-object issue by providing a gateway between both worlds.

Generally, however, tools sit on top of the infrastructure and implement little more than user interaction (in the case of interactive tools, editors that is) or analysis and transformation of repository contents as it was the purpose of many tools in the ESPRESS project.

The architectural idea is to have each tool “do one job good”, and only this one job. Integration of such specialized tools is achieved by an infrastructure that must be aware of much more than “one job”. The complexity presented and discussed throughout this thesis can be seen as the “once-and-for-all” attempt to cover different integrational and fundamental issues, *in order* to enable cleanly specialized tools to smoothly co-operate.

It has been shown that all tools benefit from the workbench provided context menu, ensuring some level of uniform look&feel. Also tool docking works in the same direction.

The tools that have been integrated, both in the ESPRESS variant and in the current state of PIROL, demonstrate for a wide range of kinds of tools that integration is feasible and that complexity of this integration does not depend on the number of tools already integrated, but that only one integration task — tool-to-workbench — has to be performed for each tool. This suffices for all tools to co-operate which is mediated effectively by the workbench.

Examples have been shown how the combined use of tools and ROCM extensions facilitates a semantically meaningful and yet flexible integration. PON relies on representative objects for its own configuration. ZooEd shows the usefulness of DVCs even for unexpected underpinning of semantics in terms of repository classes (see the state machine example). MESSED uses the `COMMUNICATION` ROCM package for creating and sending messages. The idea of address books shows how a tiny extension to the ROCM may contribute significant value to the environment. pjEdit and COFS demonstrate how functionality can be split between a tool and methods of the ROCM: In the case of pjEdit, class `LANGUAGE` defines an interface for small parsers that are capable of creating structured objects from a flat text. Conversely, sub-classes of `SOURCE_PRODUCER` create textual skeletons from structured information. Similarly, COFS uses sub-classes of `UNPARSER` to generate text from structured objects.

Chapter 14

Miscellaneous and Summary so far

This chapter summarizes the presentation of PIROL.

At the concrete level, PIROL is evaluated with respect to run-time performance (Sect. 14.1) and suitability of the technology used (Sect. 14.2). In a concluding evaluation of PIROL, Lua/P is compared to related work, and the internal structure of PIROL is discussed with respect to evolvability.

Secondly, the presentation of concerns and concern interactions is summarized and some generalizations and conclusions are drawn (Sect. 14.4).

14.1 Performance

Considerations about performance have been guiding for several aspects during the development of PIROL. While some of these discussions are evident without measurement, other decisions require a closer look. There is no doubt, that manipulating objects in PCTE costs more than the corresponding operations performed solely within Lua. Concrete caching strategies, however, which avoid repeated lookup in PCTE have to pay for this by a computational penalty and increased memory usage. This trade-off needs detailed performance figures. PIROL is not optimized to the last details but some measurements have been performed with according action.

14.1.1 Profiling technique

The task of profiling PIROL is difficult because of PIROL's heterogenous nature. Control flow crosses several language and even process boundaries. For instrumenting the system, different styles of meta–programming have been employed. For the Lua part, it is easy to wrap selected functions by profiling functions, which accumulate the times of function execution. Functions `profile_reset` and `profile_print` allow to constrain the measurement to relevant sections during execution. Tools where not patched, but an attribute guard was added that used the attribute `TOOL_INSTANCE.message` as a trigger for starting and stopping measurement. Tools are expected to indicate their current state by writing to this attribute, so that the attribute guard indeed catches relevant points during the execution of a tool.

Calls to the PCTE API were effectively wrapped by C macro technique. Each function that is exported from C to Lua is declared by macros that map the different disciplines of parameter and result passing. Into these macros additional calls to Lua were inserted that record each function's timing data. Also this macro technique can be considered meta-programming, as it changes the semantics of a function definition. This is instrumentation without additional tool support and without the need to manually select or even modify all functions that should be observed.

Each measurement was performed repeatedly on a Pentium III/1000 system with 256MB RAM, running Linux with no other activity. From these repeated runs, the fastest run was selected. Some details of comprehensive material from such fastest runs are collected in the discussion below. For each run the total time, the time spent in the PCTE subsystem and detailed figures for the most relevant functions are captured. In special cases details about actual parameters and call chains are also collected as context for a given function. While some measurements involved the execution of typical tool functionality, others were implemented as Lua/P scripts. For the case of scripts, the total time spent in Lua is computed as the difference between the total execution time and the time spent in PCTE. Conversely, the time spent in the tool could not be measured easily, because this would either require modification of the tool or profiling of MSG, which both seemed not practical. However, indirect measurement of MSG was performed.

14.1.2 Concrete measurements

The following questions were investigated by measurement:

1. Is the number of requests relevant, by which a tool obtains its data? If this is a bottleneck, combining several requests into one should show a considerable gain in performance.
2. How expensive is creation of many objects?
3. Is creation of dangling objects tolerable, or should linked creation be enforced throughout?
4. Does the effort for retrieving an RO by its ROID justify a cache of ROs by ROID?
5. Does the effort for retrieving the ROID of a given RO justify a cache of ROIDs by RO?
6. Are keep–alive links a tolerable technique?
7. Is access control on every object access tolerable?
8. Is launching a new Java virtual machine tolerable each time a window is opened?

Most questions regard the potential for optional optimizations. Only (6) analyzes the impact of a technique that is required for data integrity in the given architecture. Most measurements were performed in the same version of the system. Only for (7), it was too difficult to implement both variants in the same version with only a runtime switch.

	no folding		with folding		improvement	
Number of requests	169		129		40	23.7%
Workbench	172 ms	3.4%	155 ms	3.0%	17 ms	0.3%
Java + MSG	4957 ms	96.6%	4595 ms	89.6%	362 ms	7.1%
Total response time	5129 ms	100.0%	4750 ms	92.6%	379 ms	7.4%

Figure 14.1: Folding list messages

Results at the interface to MSG

When reading a list of objects, a tool first reads the list of ROIDs and then has to query detailed information for each object separately. At least the dynamic type (attribute `class`) of an object is needed for creating an appropriate proxy object. In this experiment **(1)** the result of querying a list was extended to be a pair of lists: a ROID list and a list of class names. Now the JAVA library was modified to read the type information for proxy creation from the second list instead by separate queries. Thus, one query for each element in a list was saved. As a test driver PON was used, and the action to be performed was opening the base folder containing 40 objects. Thus, the interesting section was reading this list of 40 objects. Only 3.4% of the total time of about 5 seconds was consumed by the workbench, the rest is due to MSG and the tool itself. Tab. 14.1 shows the key data where percentages are relative to the total time in the original version. Reducing the number of requests by 23.7% yielded an improvement in Java plus MSG of 7.1%. Also the workbench consumes more time, if invoked in many small requests instead of the one compound request. This yields an overall improvement of 7.4% of the modified variant, which should be considered significant.

A previous setting of the same experiment showed a larger difference also on the workbench side. After changing the experiment, impact of the workbench is now quite small. Since Java and MSG were not measured separately it can be discussed only informally, that the tool did not behave significantly different in both settings. The difference in timings must indeed relate to messaging — including of course the invocation of MSG functions from Java.

The numbers further-on show, that the actual performance problem is the tool PON. It is unclear where the actually intolerable amount of time is spent. One might suspect the Swing framework of JTree [Suna] to have a considerable share, but measuring PON is beyond the scope of this analysis. The important facts are, that the workbench is already comparably fast and is further improved by folding many messages into one.

Results at the interface to PCTE

Object creation. Creating objects in PCTE **(2)** was said to be an expensive task which has motivated some optimizations in the design of PIROL. Precise measurements surprisingly revealed that an even greater performance penalty stems not from creating PCTE objects, but from their initialization in Lua/P. In fact, object creation spends most time in the API function `pcte_object_get-`

`string_attribute` while reading the ROID of objects. Closer analysis showed that during each object creation some objects are accessed that reside on a global segment that cannot be loaded to the workbench. Reading a string attribute from such objects executes roughly 100 times slower than for objects on a segment that is loaded to the workbench. This underlines the importance of loading segments to the workbench and keeping relevant objects on such segments. For the ideal case of all objects residing on loaded segments, only performance estimations exist. Estimation suggests that object creation is about ten times more expensive than most other PCTE operations.

Dangling objects. The difference between linked and dangling creation (**3**) was surprisingly not significant. Three experiments were performed creating a large number of objects with each style of creation. The results ranged between a penalty caused by dangling creation of 3.8% up-to even a performance gain of 0.89%. It should be noted, that dangling creation might cause a significant penalty, if the graph of dangling objects and their dependencies becomes large and complex. In that case, deleting a dangling object may require some effort in traversing that graph and deleting objects that become unreachable. Such efforts of maintaining a complex graph of dangling objects are not covered by these experiments and don't seem to be typical scenarios.

Caching ROs and ROIDs. As reported, during these measurements a large number of calls to the API function `pcte_get_string_attribute` was observed and could be traced back to accesses to the `exact_identifier` attribute, which is used as the ROID. These retrievals of ROIDs from the repository consumed about half of the time of some of the experiments. By these observations the relevance of caching ROIDs has been discovered.

Speed improvement by caching ROIDs (**5**) was observable, precise data, however, are not available. ROID caching only achieved its full effect after also RO caching was introduced, because PCTE hands out different handles to the same RO, that can not efficiently be mapped to the same ROID. RO caching cures this for the case of accessing objects via MSG. If an RO is reached by different PCTE links, there is no efficient way of detecting that the same object has already been accessed from Lua.

During experiments, caching ROs (**4**) saved between 6% and 32% of the time spent in the workbench. Maintaining keep-alive links (**6**) costs between 0.1% and 10%. All numbers are derived from a scenario, where a large number of objects is retrieved by a tool, which is the worst case for both questions. Once a tool has loaded most of its objects, keep-alive links will make no significant difference, but the gain of caching ROs will grow.

Compound change notifications. An unexpected anomaly has been discovered during profiling: deleting all elements from a list using method `List:wipe()` works unacceptably long on lists with many elements. More than 80% of that time is spent on preparing data for change propagation. More precisely, incremental change notifications are assembled element by element. The algorithm for compacting these notifications is run for each element performing $O(n^2)$ calculations for a list of length n. This calculation calls no PCTE functions and was therefore not considered harmful. Measuring proved this assumption wrong. Compacted change notifications reduce the effort that tools must perform in order to react to compound modifications to a list. However, the algorithm for compacting needs further optimization especially for degenerate cases like wiping a list.

Access control. Although no precisely comparable figures exist for access checking, experiments before optimization showed that about half of the time spent in PCTE was used for checking the access to an object before actually performing the access (7). The strategy was then changed as to try access unchecked and catch access errors implicitly by inspecting the return code of all access operations. This inspection is said to be implicit, because it is done by the macro that wraps all PCTE functions for access from Lua, i.e., this is not visible in client code. After this optimization, overall performance was significantly better. Direct comparison was, however, not feasible due to major restructurings in the workbench.

Java startup

Starting a Java/Swing based interactive tool (8) is one of the most expensive operations in PIROL. To what extent this cost is simply due to Java technology can be estimated by the following experiment: Two applications that just open a window, one written in Java using Swing the other in C++ using the QT toolkit. These are the figures:

language/toolkit	typical time for application launch (msec.)
Java/Swing	> 2000
C++/QT	360

Since this time is the lower bound for displaying a window that resides in a separate OS process, it can be concluded, that for Java/Swing the technique of multiple OS processes in an integrated environment is broken.

14.1.3 On the role of performance optimization

Measurements and discussion should have shown, that design and implementation choices have significant impact on the performance of PIROL. It was not the primary goal of this development to provide a fully optimized implementation, but in a field where some former research was canceled because of unacceptable performance, such issues can certainly not be ignored. Improved computer hardware has of course made a significant difference during the development of PIROL[1], but a design completely ignorant of performance issues would still be useless even on todays computers. During the development efforts were invested not to repeat the same performance problems that previous projects had.

Measurements were only performed in the late phase of preparing this thesis. Some of the measurements confirmed previous expectations. Some mea-

[1]CPU speed of machines used for the development of PIROL ranged from 50 to 1000 MHz.

surements revealed unexpected performance penalties. All these data are not really suitable for comparison with other projects. The main value of profiling was in incrementally improving the performance of PIROL. There is no easy way, to reproduce a variant without these improvements against which precise comparison could be performed.

Actual bottlenecks. Measurements show at an informal level, that the main bottlenecks are Java (Fig. 14.1) and Lua/P (`List:wipe()`). Java is beyond scope for evaluating the PIROL infrastructure. Lua/P is admittedly a prototype language, which could for a production environment be transformed into a compiled language with many options of classical optimization. Note, that the worst case measurement relates to an algorithmic problem and to a programming technique, that is well explored for functional languages (type based pattern matching). The uses of PCTE and MSG have been demonstrated to perform well enough. Only two restrictions apply: PCTE segments not loaded to the workbench might be a problem (factor 100) and concurrent access by many users has not been measured. [EKS93] reports performance problems for that kind of scenario. Lua/P has the capability to provide better results by the use of terms, but sufficient measurements of scenarios using considerable sized term structures are missing.

14.2 Suitability of Lua

PIROL has seen two very different implementations. The first incarnation [Her94] was implemented mainly in Eiffel [Mey92] and used Tcl [Ous94] for interpretation of method bodies. Tcl proved very difficult to manage because of the lack of a real syntax. In Tcl every data is strings and every statement is a command. This was found not to be a good starting point for the design of a programming language. Also the language barrier between Eiffel and Tcl (mediated of course by C) was suspected to be a bottleneck, because all control structures where interpreted in Tcl and each access to an object had to be delegated to Eiffel, including even method calls, that via Eiffel were dispatched back to Tcl.

Within a German Brazilian co-operation the author had the chance to directly learn from the developers of Lua [IdFC96] and discuss many language issues with them. Early experiments with Lua for PIROL had their influence on the transition from fallbacks to the more powerful tagmethods. The second incarnation of PIROL started out as integrating MSG, a simple concept of classes and a simple Tk [Ous94] interface into Lua. This integration experiment was very successful and after that, Lua/P evolved over time up-to the current state. This section is now to evaluate the suitability of Lua for the tasks at hand. Criteria to be used in this evaluation are:

- Could desirable features be implemented or did Lua impose significant limitations on the design of PIROL?

- Is the syntax (which by and large adheres to the general syntax of Lua) suitable?
- Could an implementation be achieved whose structure is comprehensible and apt to evolution?

The performance of Lua has already been discussed above. During development of PIROL it has met the expectations, for a production environment a compiled, optimizable language would be required.

14.2.1 Syntax

For a prototypical environment the built-in capabilities of Lua's syntax were quite satisfactory. The relevant constructs could be introduced without modifying the Lua parser.

Syntax variants of Lua tables [← Fig. 1.2 on page 24]

Associative arrays with their different syntactical variants are used to denote structure definitions of classes and grammars. Anonymous functions are used in the `t_select` control structure, as access functions of attribute guards, and within a DVC for predicates, accept functions and redirect functions. In some of these cases the original Lua syntax appeared slightly unsuited. In these cases a simple pre-processor enhances the syntax, translating, e.g., the predicate syntax

predicate (arg) *stmts* **end**

to real Lua syntax

predicate = **function** (arg) *stmts* **end**

In addition to hiding the assignment in the above function definitions, method declarations had to be simulated for some cases. While regular (named) function definitions allow to create an implicit parameter "self" by using a colon delimiter instead of a period, this cannot be used for anonymous functions. Also this "self" parameter is generated by the pre-processor regarding accept functions and access functions of attribute guards and attribute redirections.

Anonymous functions can be compared to blocks or chunks in languages like Smalltalk [GR83]. With functions the interface between different blocks is clearer than with blocks, but this comes for the price of a little more verbose syntax.

The pre-processor is a simple Lua function based on regular expression matching and substitution. Another task of this preprocessor relates to type declarations. Plain Lua syntax would require attribute declarations of this syntax:

```
attributes = {
   attr_name = attr_type, ...
}
```

In this place the equality symbol is misleading and thanks to the pre-processor a colon can be used instead. For method definitions Lua would not allow any type annotations to arguments and return type. Using the pre-processor, such type annotations are possible and translate into additional function calls that are generally ignored but the bootstrap process uses this information for building the reflective representation of methods including typing information.

To summarize, plain Lua syntax was sufficient for all requirements. Types in method signatures would have to be annotated separately and several minor inconveniences would have to be tolerated, that are, however, alleviated by means of the pre-processor.[2] This holds for the prototypical environment. For a production environment, a custom parser with no such restrictions would probably be more suitable. But during the development of Lua/P a more rigid approach might have hindered evolution.

14.2.2 Program structure

Much of this thesis was dedicated to the question how different concerns interrelate at the conceptual level. Using a flexible language like Lua with little support for encapsulation, the resulting program structure deserves a closer look. Can different concerns be localized in the source code? Are the existing means for structuring, generalization and re-use powerful enough for building complex systems? We have argued for evolvability of the ROCM and tools, but is the workbench itself evolvable?

The following techniques of Lua have been employed.

Tagmethods perform much of the dispatching, and different tags are used for ROs, VOs, transient objects, different types of lists and terms. Whenever the tagmethods need meta information for dispatching, *tables of access functions* are used. This has several advantages.

- The main dispatch methods are kept separated from individual strategies that relate to specific types.
- Access methods can be implemented as *function closures* storing additional context data. Thus generic access functions can be customized during initialization.
- Access methods can later-on be replaced by *wrappers*. Guards are an example for this technique. Usually, the wrapper function is again a closure that stores the original function for chaining.

Such a touch of higher-orderedness — in conjunction with the powerful concept of meta programming with tagmethods — gives a very good separation of certain concerns for a low price. Several changes to the workbench have been really simple tasks thanks to this structure. For example, allowing writable derived attributes by use of a client provided assign function was a matter of adding a handful of lines to the workbench.

Bootstrapping PIROL [←4.2]

Function values play a role also for different modes, by which a Lua/P file can be read. According to the different phases of the bootstrapping process, reading the same file has quite different effects. This is easily achieved by assigning different functions to commands like `Package` and `Class`.

Lua has other *reflective* capabilities in addition to tagmethods.

[2]Lua's openness towards semantic extensions seems to call for equivalent means for syntactic extensions, like some kind of annotation technique, which is ignored by the interpreter, but could be used via the AST by external tools. Syntactic extensions for Lua are, however, not supported.

- The ability to query the type and tag of each value,
- Functions for accessing data by passing variable names as strings,
- Invoking functions with explicitly constructed argument lists (encoded as table, of course) using builtin function `call`.

All these capabilities were valuable for PIROL. Type inspection is used to pack values for sending via MSG. The explicit `call` function can also be used for exception handling by means of a protected mode of invocation. For generic error handling this is valuable. This does, however, not reach the convenience of application level support for "try-catch" blocks.

For one specific purpose, the object-oriented *Template&Hook* meta pattern [Pre95] is simulated using function parameters: internal runtime structures for the different kinds of classes for ROs, VOs and transient objects are built using the same function (template), but the specific parts for each kind are implemented by functions that are passed as parameters (hooks). This illustrates the similarity between inheritance and parameterization.

Caveats

Lua does not provide a *module* concept. A very rudimentary file based module concept has been implemented using `require` and `provide` functions, that simply ensure that each module is loaded only once, while allowing their import at any location that depends on the given file. With Lua, the real purpose of separation into different files can, however, not be encapsulation but just comprehensibility.

Several aspects of a complex Lua program need very good *documentation* and conventions to alleviate missing support for safety. Firstly, global variables fall in this category. Also, the flexibility of Lua tables requires cautious documentation. Especially nested tables can easily get out of control, if documentation is lax.

After all, the amount of Lua code used for implementing the workbench[3] does not require strict modules. This part of the architecture amounts to 10 KLOC spread over 31 files. This is well manageable in Lua.

Inter language working

The integration of libraries written in C is quite easy and proved to be a very useful technique. Normal integrations of this kind simply provide an interface by which a Lua program can invoke C functions. It is also not a problem, to register Lua functions for a dispatcher written in C. This style is used for registering callbacks written in Lua for MSG patterns which are dispatched in C code.

Integrating garbage collection across this language border is more intricate, but registering a C function as tagmethod for the `gc` (garbage collect) event is a perfect hook for having both worlds co-operate also with respect to garbage

[3]This is obviously not counting code written in C, which is cleanly decoupled from Lua code by the language barrier.

collection. The term–library uses a simple mechanism of reference counting for memory management. This mechanism had to be extended to count also inter-language references. Once the Lua API for terms had been constructed with disciplined maintenance of this reference count, the joint garbage collection works well also in this setting.

Exceptional control flow [←7.4.5]

The most intricate issue in inter language working was the implementation of control flow of exceptions across the language border. In order to obtain a good combination of convenience and performance a set of callbacks in different directions co-operate for catching and reporting common errors. The author considers the given solution close to optimal, since it combines a good performance with little burden in client code. Most of the complexity is encoded in the generic code of the dispatcher for MSG requests, macros that wrap PCTE functions and the Lua module `error.lua`.

Remaining complexity

A piece of code, which is naturally complex, no matter which language is employed, concerns the maintenance of consistency between lists that are connected by the filter construct of DVCs. Difficulty arises from the one-to-many relationship, from the fact that each of these interconnected lists may initiate a modification, and from separate sets of indices[4], not to forget renaming. Such complexity is once more the price for requiring transparency of filtered lists. The infrastructure solves this issue in order to unburden client code.

Object Teams [17.6→]

A difficulty, that may not be worth its benefit, regards lazy lowering and the many places which need to check whether lifting or lowering should occur. The experiment was quite instructive, but follow-up models work with well defined places of lifting and lowering, introducing so called *externalized roles* instead of lazy lowering. Of course, such static placement of lift/lower calls requires static program analysis, which leads to our next issue.

14.2.3 Static correctness

A major deficiency of Lua/P is of course its lack of static support for correctness. More specifically, Lua/P has no type checker and cannot statically recognize further errors like wrong uses of colons or periods in method calls. This is the backside of Lua's great support for flexible language evolution. It wouldn't even require a change of the language proper. A separate toolkit, providing a parser that generates a standard AST, would help significantly. Such a toolkit should by a clean API or even framework design enable the construction of diverse analysis tools. I.e., a separate type checker, which would be run independently of the interpreter, would have saved considerable efforts of debugging.

For PIROL such a tool for static analysis could also verify that only authorized parts of the code access certain low-level functions like the PCTE API.

At a first glance, Lua might lack some support for scoping, leading to excessive and uncontrolled use of the global name space. Actually, tables create

[4]Recall, that each filtered view locally uses consecutive indices $1..n$.

name spaces, which made implementing classes as a special kind of table relatively easy. This is restricted only syntactically: Lua does not natively support entering and leaving a name space, thus, the current object has to be applied explicitly (`self`) within method bodies. It is, however, possible to simulate local visibility of certain names by executing certain functions in a temporarily modified global environment. Attribute guards apply this technique in order to provide access to functions like raw_assign, which are inaccessible outside guards.

If the technique of simulating name spaces by changing the global environment would be used excessively — e.g., for accessing attributes and methods without `self` — this would certainly conflict with performance.

It would have been an interesting research to combine all special language features of Lua/P with a state-of-the-art type system including type parameters etc. Unfortunately, most of this is meaningless without a type checker.

14.2.4 Things that could not be done with Lua

Two issues remain, where the available version of Lua imposed notable restrictions. A minor problem remained with the syntax of linked creation. It would have been desirable to keep the syntax of dangling creation and only by static analysis achieve an order of evaluation that could combine creation and assignment into one PCTE call. Of course the order of execution could not be influenced in Lua. After finding out, that dangling creation is in fact tolerable, this is not a relevant problem.

More seriously, Lua as of version 4.0 does not support multi-threading. This rendered impossible all experiments of transaction support, in which each tool could start a new thread in the workbench with which it communicates directly. As discussed in Sect. 9.1.7 this would be the precondition to using PCTE transactions for tool synchronization.

14.3 Evaluation

In the early phase of research concerning SEEs, Osterweil observed,[5] that a software environment must be broad in scope, highly flexible and extensible and very well integrated. This illustrates a tension, in which SEEs may appear as killer applications for technology in OODBMS and multi-dimensional separation of concerns. Both developments have largely been influenced by SEE research.

The integrational aspect is covered by PIROL's architecture. Both broadness and flexibility can be applied as criteria to Lua/P.

In Chap. 2 a comprehensive list of requirements is given according to [ESW93]. From these requirements, only one has not been dealt with appropriately: versioning. Sect. 9.1.7 has unfolded the discussion and explained why versioning issues are especially hard in the given setting. Other than that, PIROL meets all requirements of [ESW93] and also some requirements not listed there.

[5][Ost81] quoted according to [TBC$^+$88].

14.3.1 Lua/P

In previous chapters, three repository languages other than Lua/P have been presented: Pleiades, APPL/A and GTSL. Of these, only GTSL is object-oriented. All languages provide support for consistency management, be it rules or event-action pairs. Guards in Lua/P roughly fall in the second category. Deriving data is supported by APPL/A (`out` attributes of relations) and GTSL (methods). Derived attributes of Lua/P improve on these techniques by automatic updating and notifications. The special support for unparsing in GTSL falls beyond general purpose language features. In PIROL such capability was not considered a requirement needing special language support. Sect. 16.1.1 will discuss a more detailed approach to that issue.

Pleiades and GTSL distinguish aggregation and referencing, a distinction that is also part of PCTE's much richer concept of link categories. Efforts to exploit this difference also in Lua/P have been started, but conceptual difficulties as discussed in Sect. 9.1.7 have rendered such efforts useless. To some extent, COs replace the containment relation of other approaches, even allowing overlapping containment of ROs.

To the best of the author's knowledge, no existing repository language before Lua/P supports a combination of objects and efficient very fine grained terms. Pizza [OW97] is related in concept and there has also been a recent discussion about introducing tuples to Java, but the contribution of Lua/P goes beyond mere language design as it integrates efficient persistence for this hybrid setting.

Furthermore, Dynamic View Connectors provide support for flexible views that is unknown for other repository languages. Sect. 16.1 will give some more details on comparison.

14.3.2 Multi-paradigm modularization and re-usability

Lua/P integrates different paradigms and programming concepts. However, Lua/P is embedded into an even greater framework, where very different concepts are used for different aspects of modularization and flexibility.

Different levels of encapsulation exist. The weakest level concerns the Lua part of the workbench. Language borders provide much stricter encapsulation, process borders, finally, strongest encapsulation.

On the other hand, many different extension points exist, most notably by the framework-like design of large parts of the meta model. Of course, Dynamic View Connectors and upgrading are essential facets concerning re-use and evolution as discussed in Chap. 12.

In retrospect, the most valuable method for achieving just all the flexibility that is needed, was the *co-design* of the *concrete system* PIROL populated with concrete tools together with all its *infrastructure*. In the end, only the combination of so many techniques ranging from macros, over tagmethods, all the way up-to a three–tier component architecture made the development of PIROL possible, which by now covers so many concerns, is a powerful and flexible platform for tool integration and even after several years of development exhibits no decay of internal structure, but preserved its maintainability through many iterations of development.

14.3.3 An example maintenance task

One success story can be reported from the late phase of preparing this thesis. This is reported here, to underline both maintainability and also the benefit of documentation at that level that has dominated large parts of this thesis.

An outstanding bug had to be removed, which had been identified as the invalid access to an object that had been created within a transaction, which was later-on aborted. Rolling-back the transaction made the newly created object invalid. Further analysis showed, that a reference to this invalid object was still accessed via a list, i.e., the list was discovered to be unaffected by the roll-back.

At this spot, a brute–force approach might have tried to 'manually' update the list representation when aborting a request. For this task almost all needed information is readily available because change propagation incrementally collects all modifications during each request.

Further analysis revealed the reason, *why* the list was not updated, and, *why* it exists in memory, to begin with. Lists (i.e., their index structure including object references) are cached in the workbench for efficient access using consecutive indices, which are not guaranteed in PCTE. Such caches are not affected by roll-back in PCTE.

Only going back to the very nature of lists in Lua/P, which are just cached images of underlying repository data, it became clear, that simply dropping such inconsistent cached lists is a much cleaner solution.

Remains to be pointed out a location in the code, that identified which list would have to be invalidated (deleted) at which time during execution. The answer is of course: delete all lists that have been modified in a request that is later-on being rolled-back. The location that triggers the roll-back is easily identified, but how to detect arbitrary modifications to lists? All lists declare all their modifications for the purpose of change propagation. Fortunately, even the four hooks that collect update information for the list operations *append, insert, replace, delete* go through just one common hook. As a result, in order to remedy the bug, five lines of code had to be inserted to maintain a list of dirty lists per request and four lines of code for dropping all dirty cached lists when a request is aborted. That is nine lines of code for a quite hairy bug.

This example demonstrates two kinds of success. Firstly, a good documentation of concerns at different levels helps in finding the *real cause* of a problem. In this case the caching nature of lists is not visible from looking at the list implementation. Only from the bigger context the intention of internal list representations can be understood as a *cache* of underlying repository data. Secondly, the workbench code, which has gone through many iterations of restructuring, allows to identify very specific conditions with a small number of locations in the source code. In other words, many situations during program execution that can be expressed in simple words can be located in the source code in very few places.

While re-structuring involves a fair amount of intuition to 'put together what belongs together', maintenance tasks as the one presented here, help to evaluate the structure that has been reached. If simple corrections would involve tedious modification of many parts of the code (which might only exhibit faint similarities), the structure should be called inappropriate. If doing simple corrections is in fact simple, the structure can be said to match the underlying concepts. Thus, being able to correct a bug with very few lines of code, proves the given design suitable and helps to maintain such clean design. In other words, striving for the simplest possible solution to a problem, is not an end by itself, but guides towards least possible pollution by work-arounds and patches.

In fact, localizing errors was in some cases quite difficult due to the number of different techniques involved. Error correction, however, hints at a good matching of structure and concepts.

14.3.4 Object-oriented SEEs

PIROL is an effort at building an SEE framework using the object-oriented paradigm. There have been comparable efforts before. The following quotation is from an article titled "*Good News, Bad News: Experience Building a Software Development Environment Using the Object-Oriented Paradigm*" [HSS89]:

> "Our experience has taught us that object instances provide a natural way to model program constructs, and to capture complex relationships between different aspects of the software system being constructed. The object-oriented paradigm can be efficiently implemented on standard hardware and software, and does provide a degree of extendibility without making major modifications to the existing implementation.
>
> Unfortunately, our experience has also demonstrated that some natural extensions to the software development environment are difficult to incorporate into the system. We argue that the lack of extendibility is due to the object-oriented paradigm's lack of support for providing modifications and extensions of the object-oriented paradigm itself."

On principle, these observations — originally referring to the much more limited setting of programming environments — hold for PIROL, too. Object-oriented meta modeling is in fact a good starting point, but SEEs need more than that, they need multi-paradigm languages, like Lua/P.

With this statement, we will leave the concrete considerations concerning PIROL and will conclude on the method of a Concern Interaction Matrix below, using of course the experience from developing and documenting PIROL.

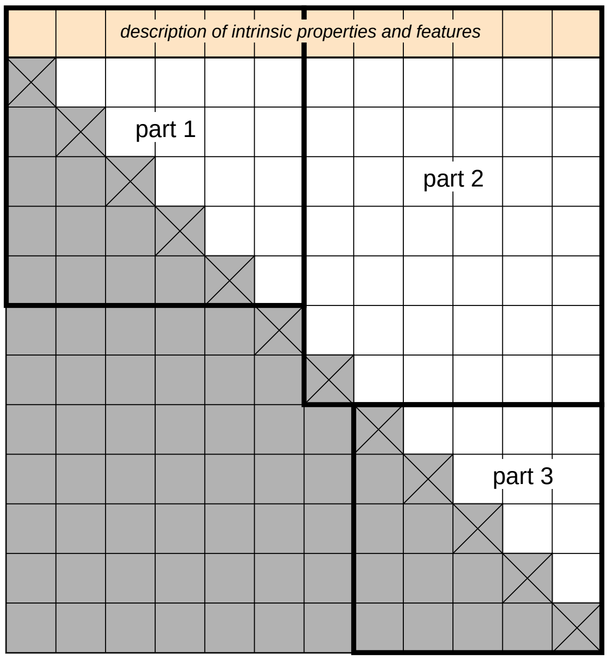


Figure 14.2: Overview of partitioning the concern interaction matrix

14.4 Concern interaction

Throughout this thesis little pieces have been collected on how the different concerns of PIROL interact. The chapter summaries have started to draw conclusions from these findings. Still the pieces remained more or less isolated. This section is an attempt to draw a very broad overall picture of PIROL in terms of the concerns that have structured the text. During this process, constellations are compared and more fundamental patterns of concern interaction are sought.

As a guide through the following discussion the central keywords from the presentation of PIROL are collected into an abstract view of the concern interaction matrix as introduced in Sect. I.5.1. This matrix is split into three tables Tab. 14.1–14.3. Fig. 14.2 shows the location of these parts in the global matrix. The order of concerns is the same as it was in the previous chapters.

Cells of the matrix contain keywords prefixed by a rough classification:

B Interface to an existing *base technology.*

F Introduction of a new *feature.*

U The development of a new feature was guided by the desired *uniformity* with an existing feature.

C Two features complement each other.

A One concern *applies* a feature from another concern.

I One concern implements a requirement from another concern.

IF A new *feature* is *implemented* by the combination of two concerns.

IT A new mechanism *implements transparency* between two concerns.

T A specific aspect is conceptually *transparent* to another concern.

— Two concerns appear orthogonal.

?? A combination of concern raises a new issue.

14.4.1 Concern characteristics

We will now once more iterate through the 12 fundamental concerns but this time from the perspective as to find out what *kind of concern* we have at hand, and what its general relationship to other concerns is. This will also give explanations to some entries of the matrix.

Meta modeling

For PIROL, meta modeling is the most fundamental design decision. Without meta modeling there would be no PIROL. It is the basic terminology in which many of the other concerns are spelled out. Many packages and classes of the ROCM are dedicated to specific concerns, and the most central class `ANY_RO` is the pivot by which many concerns are connected. The rôle of meta modeling is close to being a *paradigm* for PIROL.

Persistence

This concern comes as a *requirement*: (almost) all data must be persistent. It connects the base technology PCTE to PIROL. In this connection several *mappings* had to be developed. Goals of these mappings were transparent encapsulation, uniformity and efficiency. Also, some non-standard functionality of PCTE should be made available to higher levels of the architecture. This is, because PCTE is more than a passive data storage, but provides many services from schema management up-to transactional protection. Most other concerns had to find specific *translations* into the world of persistent data.

Finally, having persistence also opened the *choice* between persistent and transient data. In order to achieve best possible transparency of this choice, other concerns had to be *duplicated* as to perform the same operations in the same way for both kinds of data. In this respect, persistence can be considered a *multiplier*.

Granularity

In principle, granularity is another fundamental requirement of PIROL but at a closer look, it spans a scale on which many options can be picked. On this scale, data in PIROL can be either objects, lists or terms. Also, changes and change notifications can be of different granularity and PIROL chooses quite a fine level of increments. In this respect, also granularity is a *multiplier*.

	meta modeling	**persistence**	**granularity**	**behavior**	**exceptions**	**integrity**
intrinsic	**B**: Lua **F**: CO	**B**: PCTE	**F**: terms grammars	**B**: standard OO		**F**: guarded attribs. **F**: derived attribs.
meta modeling	×	**IT**: type mapings PCTE ↔ Lua/P **F**: object inlining **F**: reverse links **F**: transient data	**U**: syntax	**F**: GENERAL PRODUCT PROCESSES	**I**: report PCTE errors	**I**: meta model semantics
persistence	×	×	**I**: pack as content	—	**I**: transaction	**F**: orphanage
granularity	×	×	×	**F**: `t_select`	**I**: `ERROR`	**F**: hybrid garbage collector
behavior	×	×	×	×	**A**: many	**C**: guard+method
exceptions	×	×	×	×	×	**I**: guard exception

Table 14.1: Concern interactions (part1)

	architecture	**control integration**	**multi-user**	DVC	**services**	**evolution**
intrinsic	**F**: 3–tier **F**: MSG	**F**: wb→tool messages **F**: change propagation **F**: representatives	**F**: access coordination **F**: user context **F**: communication	**F**: connector, VO **F**: lifting/lowering **F**: feature mapping	**F**: tool adminis-tration **F**: context menu **F**: state machine	**F**: upgrading

meta modeling	**F**: proxy classes	**F**: `TOOLS`	**IF**: `PROCESSES`
			IF: `PERMISSION`
			IF: `MESSAGE` et al
				U: VO $\simeq$ RO
				IF: two level meta model
					I: `TOOLS`
					I: `MENU`
					I: `STATE` et al
						IF: two level meta model

persistence	**IT**: ROID	**IF**: represenative as configuration	**I**: permissions **I**: annotation	**T**: mapped to persistent CO	**I**: transient menu	**I**: upgrade (convert)
granularity	**F**: point-to-point messages **IT**: list operations	**IF**: incremental change propagation	**I**: terms are atomic wrt. access control	—	—	—
behavior	**F**: proxy generator	**F**: utilize tool behavior	**IF**: mail delivery	**F**: accept **F**: delegation	**I**: throughout	**F**: `upgrade`–methods
exceptions	**F**: request= transaction **IT**: error propa-gation	**IT**: no `changed` message on abort	**IT**: `AccessException`	—	—	**F**: `UpgradeError`
integrity	**F**: transaction **T**: guards **IT**: keep-alive-links **IT**: distributed GC	**IT**: `changed` for derived attribs. **F**: guards trigger external behavior	**A**: access control **F**: synchronize meta data by guards	**T**: change propa-gation RO ↔VO	—	—
architecture	×	**F**: tool-to-tool communication	**F**: workbench–to–work-bench communication	**F**: tool=context	**F**: distributed implementation of context menu	**I**: proxy=interface

Table 14.2: Concern interactions (part2)

	multi-user	DVC	**services**	**evolution**
control integration	**F**: user representative **F**: workbench representative	**IT**: lifting location	???	**??**: compatibility
multi-user	×	—	**F**: communication tool	—
DVC	×	×	**F**: document handling	**I**: decoupling
services	×	×	×	**I**: loose coupling by context menu

Table 14.3: Concern interactions (part3)

On the other hand, granularity introduced the mechanism of terms, which in turn helps to effectively and efficiently implement several other concerns. Thus terms are also a *facilitator* for other concerns.

Behavior

In the very early days of PIROL, the author opposed to the introduction of methods to the concept of ROs. Fortunately, his fellows Olaf Bigalk and Boris Groth persuaded him to include behavior into the ROCM. From todays perspective, behavior modeling is a golden choice to include. This concern, like no other concern, operates as a *facilitator* for the implementation of other concerns. Methods in the ROCM are a well chosen location for implementing much of the functionality of PIROL. Even that functionality that is not *implemented* in Lua/P is mostly *encapsulated* as to be available in Lua/P.

Methods complement well with the mechanism of guards. Guards profit from calling methods and extend the behavioral part of the ROCM towards access oriented programming.

Other concerns motivated specific *contributions to* behavior modeling: creation, upgrade and accept methods attach behavior to different transitions in an object's life cycle. Terms suggested adding a touch of functional programming, e.g., by means of the t_select pattern matching function. Proxies extend method accessibility across process and language boundaries. Control integration, finally, made even parts of a tool's implementation available to behavior execution in the workbench.

Exceptions

This concern has little life on its own. The underlying requirement is robustness, but exceptions *mediate* between many more concerns and requirements. No other concern cuts across so deeply. The more it surprises that exceptions seem to be orthogonal to DVCs and common services.

It is yet unsolved, whether exceptions, operating within the control flow of methods and control integration, and acting on behalf of data integrity, and paying attention to distributed architecture and change propagation, whether exceptions really are the fundamental concern, or whether transaction management and related issues should be raised to the degree of a "concern".

For the time being, an explicit concept of exception handling was a crucial ingredient to the description of PIROL. Finally, it is a good thing to know, that controls flows like the one shown in Fig. 7.8 on page 96 are unsurpassed in complexity. All other mechanisms in PIROL operate in simpler ways.

Integrity

This is a very *general requirement.* Different interpretations of integrity where used throughout the description, the distinction into technical and semantical integrity being the most prominent one. Many other concerns have to pay attention to integrity. If it were only for that unspecific requirement, integrity might have stayed at the same level as *performance*, which is omnipresent but seldomly precisely explicit.

Integrity has, however, produced some mechanisms, that *facilitate* some other concerns. In parts of this thesis, integrity has been identified with guarded or derived attributes. This is of course not precise, as it mixes the requirement with mechanisms to its solution. Still the author considers this kind of grouping helpful for the purpose of documenting concepts, rationales and connections thereof.

Client server architecture

This concerns is clearly another *multiplier*. Many mechanisms had to be extended to work not only in a local setting but also in a distributed fashion. Yet, this architecture is a most fundamental *facilitator* to at least evolution. As such, it is of very great value, because only the architecture-based interfaces between components made the independent development of different tools for PIROL possible. The decoupling achieved by this architecture is irreplaceable.

Control integration

Control integration has a rôle similar to that of behavior modeling. Both *facilitators* keep things going in PIROL. Not only do these two concerns resemble each other, but also both concerns appear as a complement to a more structural concern. In much the same way as methods complement the structural part of the meta model, does control integration complement the client server architecture. This could be our first *concern interaction pattern*: concerns have a tendency of appearing as a *pair of a structural concern and a behavioral concern*. The experience of this thesis suggests the conclusion that identification of such a pair of concerns is a proper hint at having found very suitable and helpful abstractions.

Control integration hooks into the rest of the architecture mainly by the use of representative objects of different kinds. These representatives finally allow for a logical communication between all components of the environment, which is much more flexible than the physical architecture by which these components are actually interconnected. Thus, representatives introduce an *additional abstraction layer* to the system.

Multi-user capabilities

The *requirement* to support many co-operating users splits into a low level concern and two higher level concerns. Access coordination (permissions and synchronization) has to be implemented at the level of the repository. Many other concerns need to pay attention to this requirement. The other sub-concerns, user context and user communication, are built on top of the existing meta model and architecture. They *apply* other mechanisms for completely new scenarios.

Dynamic View Connectors

At some levels of the architecture, DVCs *multiply* quite a few operations, because ROs and VOs must internally be treated differently. New mappings are needed and behavior is augmented by a second kind of inheritance: the delegation between VOs and ROs. The most intricate part of implementing DVCs was the management of different views on lists: VO lists as partial views on RO lists. But the gain of DVCs lies in the fact that such multiplication is limited to the core of PIROL and that outside that core, the RO-VO difference is invisible. The uniformity of ROs and VOs decouples tools from the actual ROCM and thus is a major gain for evolvability. DVCs achieve this evolvability by introducing an intermediate level of abstraction.

Common services

Common services have had very *little influence* on the PIROL core. More so, they *expose* core functionality to tools and the user interface. These services are mostly implemented in the ROCM. A major issue for services is to reconcile openness and extensibility. This happens by specific hooks in the meta model that declare services in a way that allows flexible extension during environment customization.

Evolution

This concerns is an important criterion when evaluating the quality of the system. Meta modeling is maybe the central of PIROL. The mechanism of upgrading allows dynamic reclassification thus permitting a very flexibly usage of inheritance for extension of the environment after its initial deployment. Upgrading is easily integrated in PIROL because PCTE already supports such conversions. Only the behavioral part of Lua/P had to be extended in order to integrate upgrade methods. Other than that, all mechanisms that helped decoupling different parts of the environment helped for its evolvability, most notably: DVCs.

14.4.2 Concern kinds

We can now collect a rough classification of the major concerns that have been elaborated. These kinds are put forward also with respect to observed relations to other concerns. A concern may belong to more than one of these kinds. The list starts with *restrictive* concerns gradually moving over to *enabling* concerns.

paradigm Such concerns predetermine many choices throughout the system (meta modeling).

requirement Potentially setup obligations for many other concerns (persistence, integrity, evolution, performance).

multiplier Introduces an option or choice such that many concerns need to be multiply realized with respect to these options (persistence, granularity).

projection	Other concerns need to be projected, mapped or translated into the technology of this concern.
encapsulator	A concern provides restricted access to the features in another concern (behavior).
exposure	A concern makes features from other concerns available where they otherwise would not be (common services).
layer	A concern may introduce a layer both in the physical and logical architectures (behavior, client server architecture, control integration).
complement	A concern may complement another concern. The effect may be concerns occurring as pairs like *metamodeling* × *behavior*. Also different paradigms (object-oriented, functional, access oriented . . .) may complement each other.
mediator	Communicates between different concerns, possibly reifying concern interaction with more than two concerns involved (exception handling).
facilitator	Helps in realizing another concern.
application	A concern may apply one or more other concerns as to implement its functionality (multi user capability, common services).

Two additional distinctions are possible that are not reflected by this rather pragmatic classification. First, concerns and concern relationships could be classified separately as it is done in Cosmos [SR02]. Second, requirements and mechanisms could be distinguished throughout.

Concern space modeling [15.2.4→]

14.4.3 Towards a meta language for concerns

The above list is a shy attempt at making the findings from the development and description of PIROL available beyond the original context. Good documentation — be it before, during or after the actual development — requires experience, a lot of experience. Descriptions should not satisfy, what a textbook requires us to specify. Good descriptions provide a means to find out all you need for understanding a given aspect of a system and find out, all that needs to be known in order to safely change that aspect without breaking other aspects. A very instructive field study is presented in [BM02]. The importance of this field study lies in connecting AOSD concepts to the actual, pressing problem of maintaining software, of which we only understand small parts. There is no time to read huge piles of documentation, when a maintenance task is to be performed within a tight time frame.

Documentation must enable "just-in-time competence" [WHS01]. To come back to Concern Interaction Matrix and concern classification: The first helps to achieve completeness and high connectivity of different parts of documentation, whereas the second establishes a language by which a high-level understanding of concern relationships can be reached considerably faster, than by digging through all the technical details. The title of this section speaks of a meta language, because not only capturing each concern in isolation is needed, but classification is needed and abstract description of interrelationships; descriptions that give a high level understanding without referring to the internal structure of all concerns.

The vision is a common concept of concern management in conjunction with high-level classifications as part of normal language of software developers. This given, developers can start learning about a system by descriptions at the level as demonstrated by this summary chapter. From there, essential interrelations are understood and after that, details are retrieved by jumping into the Concern Interaction Matrix at exactly those points that are of interest for a given maintenance task. The difficulty lies in providing structure that is obvious to outsiders and communicating as much meaning as possible concerning what, how and most importantly: *why.*

The lesson that this thesis tries to convey is: looking at concerns in isolation only gives a thin slice of a densely populated space. Essential aspects are best explained by focusing on the interactions between concerns. Feature interaction is not a troublesome exception, it is the rule. More than that: concern interaction is a very important origin from which system design is motivated.

Part III

Views

Chapter 15

Views in Software Engineering

The previous part of this thesis argued on two levels. The PIROL system was presented as a multi–view software engineering environment[1]. Secondly, it was discussed how such a complex system can be implemented and documented in a modular way, which simultaneously respects many relevant concerns.

This part now tries to extract the common ideas of both levels. The concept of *views* is elaborated as one of the most central notions of software engineering. It is argued that this concept is of similar importance to software development as abstraction and decomposition/composition. Reasons are sought, why for views there is still less common understanding than for the other two. Some roots of the uses of views in software are explored and the next chapter will elaborate on selected technologies at different levels that support some notion of views.

15.1 Views, Abstraction and Decomposition

Views are a very general concept of software engineering. In order to grasp this concept more crisply, it is first related to two other fundamental concepts: abstraction and decomposition.

15.1.1 Abstraction

Abstraction denotes the relationship between something conceptual and something real (concrete). Thus the inverse relationship is that of realization. The value of such relationship lies in the fact that all properties that can be ascribed to the abstract thing are known to also hold for the concrete thing. For this reason, abstraction is one of the most powerful concepts in computer science since it allows to handle a supposedly simpler — abstract — representation and still make valid statements for the more complex — concrete — thing. Examples of this relationship are design↔implementation, type↔instance, and also type↔sub-type.

Working with abstractions can take either direction. Communication among developers regarding complex programs benefits from abstract descriptions of

[1]Cf. the title of [GHJK95] containing the phrase “a Multi–View SEE”.

the existing program. Software development to a large extent tries to go the opposite direction: the traditional top–down development method tries to create an abstract specification from which a concrete system is derived by stepwise refinement/realization. The problem with top–down development lies in interactions between different levels of refinement: more often than not, only refinement reveals that decision made at an abstract level need to be revised before refinement is feasible. Parnas stresses that top-down (or as he prefers: outside-in) as a sequence of development steps has many problems, but a layered structure of the resulting product should still be aspired[Par75]. For documentation purposes it might well be useful to pretend that a process of stepwise refinement had been used, but actual development — especially in the presence of COTS components — takes different roads [PC86].

From a mathematical point of view, abstraction is a selection of information, omitting other information that is considered irrelevant (in many situations this means to omit almost everything). The abstraction relation is a partial surjective function that need not be injective. It is a function, because each origin element is mapped only once. It is partial, because it omits details from the origin. It is surjective, because the abstract level only contains elements that are related to the origin. It may be not injective, because one abstract element may subsume several origin elements.

15.1.2 Decomposition and composition

If complexity needs to be handled, a more pragmatic skill as compared to abstraction is decomposition. Large problems are supposed to be solved according to the ancient "divide et impera" (divide and reign — commonly changed to divide and conquer). It is the "reign" part that suggests to look not only at decomposition. Perhaps composition or integration of disparate parts is the larger problem (cf. [MH01]). Decomposition and composition create a relationship of containment which is to indicate a hierarchical tree structure (since physical containment does not allow overlap). It should be obvious, that for complex problems decomposition should be more than just one step from complex to atomic parts, but rather proceeding in several steps (as with stepwise refinement) should again be preferred. Already in the 1970s, this obviously yielded a proliferation of claims of hierarchical (layered) structure of systems.[2] Parnas [Par74] has some critical thoughts on the use of the "buzzword" hierarchical structure. He questions, whether the uses of this buzzword in computer science actually refer to some meaningful and comparable relation. Firstly, Parnas ridicules such statements by the observation that "any system can be represented as a hierarchical system with one level and only one part". It is the relation defining the hierarchical structure that must be defined. Pure containment is first of all difficult to establish for software systems, and, more important, static containment of parts does not necessarily say anything about

[2]Of course, layered systems may break the tree structure. The discussion frequently uses the concept of abstract machines to denote such layers. Still such system structure is more a matter of decomposition than proper abstraction as can be deduced from Parnas' critique.

the behavior or functionality of software, which might follow a completely different structure.

Parnas points out, that decomposition into modules is of value mainly for two reasons: division into work assignments and anticipation of change [Par72, Par78]. Both issues require narrow and abstract interfaces, which should be based on "assumptions believed unlikely to change" [Par78]. At least for the data-flow based approach, Parnas argues that top-down decomposition is not likely to achieve good interfaces in the sense given. Work on aspect-oriented programming[KLM+97] is in part motivated by the observation, that *no* decomposition hierarchy is able to prepare for changes of *arbitrary* nature. Only one decomposition must be selected among different possible solutions, which Tarr et al. call the "tyranny of the dominant decomposition" [TOHS99].

As stated above, the mathematical model behind decomposition and composition is hierarchical structure, and even more restrictive than that: it is a tree.

15.1.3 Views

The range of research applying the notion of views is extremely broad. Yet it is difficult to find a definition of what a view is. It seems that authors call much to the intuition of readers. An attempt for a definition can be found in [SS89]: "A view is a simplifying abstraction of a complex structure. It is useful because it emphasizes a single aspect of the structure, suppressing information not relevant to the current focus". On a general level little more can be said about views, and even the above definition might be debatable. Is a view an abstraction? Some views seem to be even more concrete, than what is being looked on. E.g., in the relation between an abstract syntax tree (AST) and a source file representing the same program, the view (source file) is more concrete than the underlying information (AST).

It is a valuable approach to define a notion not by its constituents and properties, but in terms of why it is useful. In fact a "current focus", which emphasis one aspect while suppressing other information is at the heart of views. But why is it so hard, to say more about views in general? In the light of the claim that views are a central concept of computer science, a unified theory of views might provide a valuable foundation for many aspects of software engineering. Is there a mathematical model behind this notion?

Views as a concept in computer science is so problematic because it is a set of homonyms. There is not one notion of views but quite a range of them. The task of this section is to find the differences between different application of the notion yet elaborating the commonalities that may still justify to see all this as variations of a common theme or rather specimen of one concept that can only be approximated by definitions.

Let's have a look at who is using the term "views", and what is the intention behind this usage. This will only be a first iteration. Sect. 15.2 will investigate the perspectives of different phases in the software life cycle.

Requirements Engineering. At the far end, requirements engineering tends to speak of viewpoints or perspectives of different stakeholders. Here, the common part behind all views is a system that is still being planned and designed. Differences between views originate from different backgrounds and interests of stakeholders. As a general refrain of all occurrences of "views" we identify the obligation to *reconcile* different views in one way or other. Regarding the viewpoints of different stakeholders this is performed by *negotiation*. Negotiation starts at a state of inconsistency, at which no system can be build that conforms to the all requirements from all perspectives and should establish a commonly agreeable view of the system, which is ready for development.

Thus, views are directly associated with the possibility to introduce inconsistencies. Inconsistency, furthermore, does not prove any of the involved views wrong. The requirements engineering community has reacted by performing research on how to "live with inconsistencies"[3]. Strangely enough, this "movement" was initiated by a quite technical paper, recently awarded as most influential ICSE paper [Bal91]. The original work mostly dealt with database issues in the presence of constraints that may temporarily be violated by some data in the database. But databases will be examined further later.

In the sequel we will not follow on the issue of viewpoints of different stakeholders but focus on more technical aspects of developing a system whose requirements are put forward uniformly, i.e., after stakeholder negotiation has taken place.

Concerns. Here we have another very important yet weakly defined notion. Instead of trying to define "concern", only a very pragmatic statement shall be given: for the following discussion it suffices to regard concerns as arbitrary mental foci, from which a complex system may be viewed at, in order to analyze or somehow understand a specific aspect of the system. The author is well aware, that this means to explain one notion by two even vaguer notions. The list of concerns used for structuring the previous part of this thesis illustrates the broad applicability of the term "concern". The fuzziness is intended and all that counts is, whether a prospective concern helps in structuring explanations of what a system is like or how it should be.

Apart from a rigorous definition, concerns may be classified according to some taxonomy, and relations between concerns can be captured. A special value of such models is in predicting all required changes for a planned maintenance task (cf. [SR02]). A concern model allows to analyze how changes *propagate* through the concern space.

Concern modeling [15.2.4→]

Design dimensions. This notion is used deliberately for concepts that are originally also called views: in OMT [RBP+91], design is supposed to produce a structural view, a functional view and a behavioral view. UML has adopted and complemented this concept to more than three different views [BRJ99]. The intention of the original OMT views is to separately define those fundamental aspects, which each system has. Here different views do not express the interests

[3] See the workshop series of this name at the ICSE conferences of the last years.

of different stakeholders, but views are supposed to be orthogonal, in a way that only a definition of all three views gives a real definition of the shape of the system being developed. The relation between different design dimensions is determined by the goal to *blend* all views into one model that can finally be realized by software.

Documents. While developing software, views of many different kinds end up being written down, or drawn or somehow cast into the shape of documents. Documents are no specific kind of views, but documents realize, e.g., the dimensions that were discussed above. When thinking about OMT views as documents it is an easy transition to think of a technical realization, where the blended model is stored in some repository and all views are derived from this shared data. This is the idea of PIROL and many previous works on software engineering environments.

Database Views. Research on software engineering environments (SEE) has given significant momentum to the development of object-oriented databases [AB91, EKS93, ECM90][4]. Some of these OODBMS have unique facilities for working with views for the reasons just given above. When looking at the relationship between a shared, blended model and its different views, two activities come to mind: *Merging* all partial data models to the one shared (possibly hidden) data model. Secondly, views are *derived* from shared data. Both operations may include restructuring, but such restructuring must still allow to store and retrieve any view without loss of information.

At this level it becomes clear, that such views do not have any *direct* relationship but all relations are mediated by the common repository. Still consistency between views is an important issue, or more precisely, consistency of the blended model is an issue. This is where constraints of data stored in the repository come into focus.

Model View Control. The most detailed and technical concept of views to be presented in this list stems from the Model-View-Control “paradigm”. It is related to the previous discussion, since different database views are commonly presented to the user by tools which play a View–Control rôle in the overall environment architecture. Two issues are important here: tools do not separately store their “model”, but operate on the common model in the repository. Secondly, changes in the repository must be reflected by updates in the tools, which are triggered by some observer mechanism. This is the general architecture of multi–view environments, as it has been discussed in depth throughout the presentation of PIROL. We can identify the inter–view dynamics of the MVC architecture with *change propagation* as discussed in Part. II.

Presentation views à la MVC extend the notion of views as discussed so far. Only now it becomes evident, that views not only extract information.

[4]Interestingly, some very similar concepts have been developed under the label of artificial intelligence or even in close cooperation of both research fields (cf., e.g., [GB80]). Knowledge representations require structures, languages and database capabilities that seem to be quite similar to findings in developing SEEs.

Views are not pure abstractions, but they also *add* information which is needed for presentation. It is this layout information and syntactical sugar that distinguishes views from abstractions. Views in MVC are to some aspect more concrete than the data to which they relate. Similar findings might also occur with respect to concerns and stakeholders viewpoints. Also such views might contain information that is not important to the whole. A repository based implementation still needs to store this information. However, presentation information should not be made available to all other tools, it should in general be considered private data of a tool.

15.1.4 Relating different view concepts

Along the chain of view concepts there is at least pairwise overlap. Stakeholder viewpoints define requirements for the system which can be grouped to concerns. Some concerns may map to a specific design dimension, e.g., meta modeling refers directly to the structural view. Design views can be realized by documents which in turn are stored as persistent views in a database. A tool finally presents a document, based on the database view, using the model view architecture for presentation and updating. It should be noted, however, that this interpretation restricts each kind of views to a very specific application. Many different applications of and relations between views can be thought of.

As it is difficult and maybe useless to precisely define views as such, the interaction between different views gives more information. In this case we focus on interaction between views of the same kind, and simply collect what as been said above. Views require reconciliation of disparate but possibly overlapping information, which comprises:

1. Negotiation (in order to achieve consistency)
2. Propagation of changes (in order to maintain consistency)
3. Blending[5](in order to achieve a common model)
4. Merging[5] (in order to achieve a common meta model)
5. Deriving (in order to extract a view from shared data)
6. Change propagation (in order to update presentation views)

Please note, that items (2) and (6), though sharing a common idea, are quite distinct, since (2) argues at a conceptual level of a concern model while (6) relates to a technical mechanism. It is the intention of such a list to help identify commonalities *and* differences between the different uses of the word "views".

Far remote from all precise definitions, views can now be paraphrased as

> ... something, of which multiple instances relate to some shared whole, such that instances may overlap and inconsistencies might occur, which need some provisions for reconciliation.

[5]It is not the difference between the words "blending" and "merging" that makes this point, but the difference between the levels of model and meta model.

Furthermore,

> ... mappings play an important role by which views are related to the whole and vice versa.

Such mappings may be needed both at the type and the instance level and may be of very different nature.

A word on terminology. Throughout this thesis, difficulties persist in combining the terminology of different research communities without causing undue confusion. As an example consider the notions “change” and “update”: during software maintenance change propagation refers to modifications of one part of software, which entail further modifications in other parts. If such propagation is performed automatically, the MVC architecture speaks of “notifications” resulting in an “update” operation to be performed in each view. I.e., an update is a reaction to a notification and should re-establish a state of consistency. Database terminology on the other hand calls “updates” all modifications that a client program performs regarding data in the database. Thus, an “update” (database) may, via triggers and notifications cause an “update” (MVC). Are updates cause or effect? What is an updatable view? Is it writable, or reactive? More examples could be given.

Towards behavior views. The commonality of all view concepts discussed so far relates to shared *data*. Matters will grow considerably more complex, once views are understood also as partial *behaviors* that relate to shared assets of potential behaviors. Database views are naturally focussed on data. Dynamic View Connectors stand on the borderline to programming models that introduce view concepts into implementation of behavior. Follow-up research in that direction will be mentioned below (Sect. 17.6).

Please note the difference between the *behavioral* view of OMT (which is one view beside other views) and *behavior* views (where the behavior itself is split into many views).

15.1.5 Relating abstraction, decomposition and views

Some differences between the three central notions of this section have already been discussed. These differences are now further illustrated by a few examples that can be ascribed to different intersections of abstraction, decomposition/composition and views.

An example for a strong combination of all three concepts is the Façade design pattern [GHJV95]. It is used to decompose a model into a set of classes constituting a package, such that a single class represents the functionality of the package. Thus, Façade is motivated by decomposition. The interface of the Façade is internally mapped to different contained units. Thus it can be considered a view of that internal structure. This view is at the same time an abstraction, because in proper uses of the design pattern the Façade has no functionality of its own but only provides selective access to internals, omitting most of its details.

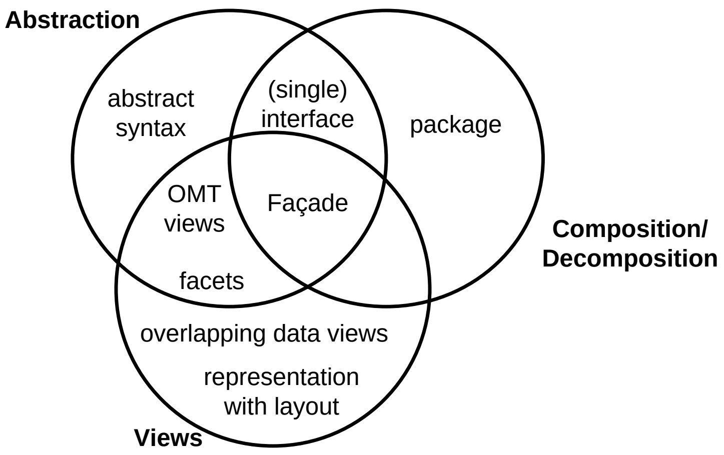


Figure 15.1: Relations between abstraction, decomposition and views

In contrast, a pure package, e.g., in Java is too weak for proper abstractions, which only motivates the use of the Façade design pattern. After all, design patterns quite commonly pop up for situations that given a “better” language might simply be solved by a language feature. Thus, packages give little more help than just for decomposition of a system into sets of classes. Due to the weak semantics of packages their actual value relates more to configuration management than the issues discussed here.

Pure encapsulation of a class using its (single) interface is very similar to the Façade pattern, but it is not considered a view, because there is neither a choice between different views nor does the interface introduce any mapping between internal and external elements. These criteria are not derived from any definition, but relate to the intuition of the notion “views”, which seems to imply, that either different views can be chosen from, or the view differs in structure from the underlying entities.

Abstract syntax has already been given as an example where a corresponding view (the actual source file) would be more concrete than the abstract syntax. No decomposition is involved in obtaining either abstract nor concrete form.

Design dimensions like OMT views are abstractions and views, but they involve no decomposition in terms of a containment hierarchy. Note, that within each view decomposition may very well be used, but this does not refer to the relation between different views. Another example in this category are facets in the CORBA component model [OMA99]. Facets are different (abstract) interfaces, which define alternative ways of accessing a component. The internal component realization is not forced to strictly follow the same structure.

Overlapping data views can not clearly be associated with abstraction nor do they relate to proper decomposition. Even clearer, view representations with layout to not belong to either of the other concepts.

Decomposition in its strong meaning enforces a strict hierarchy of elements. Abstractions do not make this commitment. Several abstractions may exist side–by–side. Such abstractions may be called views like OMT’s structural, behavioral and functional view. This suggests to divide abstractions into hierarchical ones (decomposition) and those that involve more than one dimension (views). The latter case includes true orthogonality as well as any degree of skewed interrelationships. The aspect of multi–dimensionality is captured by the notion of perspectives. This borrows from the analogy to three-dimensional space, where a perspective induces a two–dimensional image by abstracting from the third dimension. Such an understanding of views stresses the fact that certain information is omitted or hidden, just like with abstraction.

Then again, views may also be quite the contrary to abstractions. While abstractions reduce a thing e.g. by discarding its concrete representation, views may in fact *add* a representation to an abstract thing. In the Model–View–Control architecture, the abstract model is invisible and only through UI elements of a view it can be represented on the screen. Regarding the concept of views it is not an exclusive alternative, whether details are added or omitted, but a view may at the same time ignore information of its model *and* adorn the model with presentation details, not present in the model.

Obviously, from these three notions, views are defined with least precision. Therefore, it should not surprise, that this notion is more popular in the early phases of software development, than its sibling notions. For similar reasons view concepts found in implementation are quite restricted versions of the general concept.

Among the relations between the three concepts of this section, views and decomposition seem to harmonize least. Views question the goal of decomposition without overlap. Obviously, files of source code, by which software is made manifest in the end, cannot overlap, but it is possible to use *overlapping concepts* as decomposition criteria. This has strong impact on interfaces between modules, which will be more open than the highly aspired black box. In his presentation titled "why are black boxes so hard to reuse", Kiczales argued that a neat black box abstraction is only one possible view of a piece of software[Kic94]. Especially for the task of software reuse, a second way of looking at a module must be provided, which Kiczales calls the "meta–interface". His subtitle "towards a new model of abstraction in the engineering of software" may be used for assessing recent approaches and statements. Chapter 16 will present some approaches that strive for decomposition in the vein of views. All these approaches try to escape the "tyranny of the dominant decomposition"[TOHS99].

15.2 Views in the Software Life Cycle

In order to understand why views are used in software engineering and why they are so important, it is necessary to differentiate further. Different phases and activities of software development have very different reasons for working with views. Of course, managing complexity is always a driving force. Other forces have to be analyzed in the context of specific applications of views.

15.2.1 Loose coupling in the early phases.

Perspectives in requirements engineering have already been discussed. Even after the negotiation phase requirements can be defined which very little coupling. It is a significant advantage of (possibly formal) *specification* over implementation, that different aspects can be described in isolation with only one kind of relation: the model is composed as the *conjunction* of all predicates.

Views for Z specifications

The above statement has been made explicit by Jackson [Jac95]. He proposes views for structuring Z specifications. His proposal is quite straight forward, because implicit specifications in Z, which constrain the system state only by predicates not assignments, naturally support this style of separation of concerns. Actually, Jackson also discusses the two different styles of combining views that have been identified in the previous section: a data oriented approach and one based on synchronizing operations, i.e., a behavioral approach. The great ease of view-oriented specification lies in specifying different operations in completely disjoint ontologies and adding special inter–view invariants that translate sharable state between different views/representations. The connection between views is based on the advice to admit redundancies, such that consistency rules for redundant data capture the relationship between views. In a concluding criticism of Z, Jackson discusses the issue of open versus closed specifications. In Z, new specification objects need to be created to compose different views. He suggests that it might be far more convenient to just add new properties to existing units in order to combine different views. This discussion follows very much the same pattern as the issue of open classes in CLOS [Kee89] and AspectJ [KHH+01].

Structuring specification with views is feasible because an implicit specification only defines the constraints that must be fulfilled by the system, ignoring completely all difficulties that arise when constructing an implementation that conforms to the specification. It may even be impossible to implement a given specification. This doesn't necessarily prove the specification “wrong”, but only means that it has an empty model set. While the impossibility to implement a specification can — within limits — be detected automatically by the aid of verification tools, our interest lies on those specifications that can be implemented, yielding, however, an implementation that is far more complex than the specification.

It is the goal of research concerning aspect-oriented software development, to allow developers to operate with locally restricted focus. The necessity of paying attention to a large number of requirements during the implementation of each single piece of code is to be reduced as far as possible.

15.2.2 Implementation: weaving separate views and aspects

In order to come to an executable program, the subjective views from the requirements analysis have to be reconciled. Ambiguities have to be eliminated from specifications, and also the loose coupling between views must be resolved into a more or less linear form. Striving for deterministic programs means to construct chains of instructions that in a single thread of control modify data and calculate results that fulfill all relevant requirements. This transformation commonly destroys the independence between requirements.

Current research in generative and aspect-oriented programming strives for automation of the last step, where different aspects are woven to instruction chains, allowing the programmer to operate on an intermediate form, in which independence between aspects is to be pushed to the limits. The generality of this effort is demonstrated by the COMPOST project [ALN00] which reduces different AOP approaches to static meta programming. The final executable program is generated by tools from the input of source code, that exhibits better modularity than conventional source code.

Of course not every program is a linear chain of instructions. True parallelism is commonly used for improving system performance. Little is known about the usefulness of parallelism regarding semantical modularization of problems that do not naturally involve parallelism. Regardless of the difference between true and simulated parallelism, *event based* programming is an alternative to the strict sequential paradigm using procedure calls. Communication via events in fact helps to decouple modules [SN92].

15.2.3 Design: mediating between specification and implementation

It is the goal of design to mediate between the different preferences of specification and implementation. As a consequence, design has few intrinsic criteria, but development of design techniques and methods depends on a good knowledge of the two adjacent phases. If seamlessness of software development is to be improved, design techniques should closely adhere to the conditions of requirements engineering. The gap that results between design and implementation should preferably be bridged by improved implementation technology, because at this level, technology is not restricted by the needs of (communicating with) non-experts. New formalisms and techniques for implementation have only one grave restriction: skills and training of developers, which is, however, more a matter of time than of anything else.

While experience concerning multi–dimensionality in source code is growing, the transfer of these new concepts to the pure design level needs to catch up some.

Several extensions and modifications of the UML exist for AOSD. Low level approaches try to diagrammatically visualize e.g. AspectJ constructs [SHU02]. UMLAUT focuses more on design patterns and uses AOP techniques to perform weaving at the level of designs [HJPP02]. Theme/UML [CW02] and UFA [Her02a] take a middle road. Both strive for design modules at the level of packages, which group a set of roles and their collaboration. In contrast to earlier work on collaboration based design [DW98], these approaches include binding relationships, which should map to corresponding relationships in the implementation, making use of AOSD technology for model composition.

Only through large scale aspect-oriented design, the seamlessness reached to-date can be assessed and directions for further research in programming languages and tools can be determined.

15.2.4 Concern Modeling

It should be clear by now, that software development has to deal with different facets. For complex systems, management of these facets is essential. As the number of concerns rises, additional effort is needed, just to organize all concerns and their relationships. Views as discussed throughout this chapter are just one dimension in a larger concern space. To the best of the author's knowledge, only one approach has yet addressed concern modeling in all its generality: Cosmos [SR02].

Technically, Cosmos is a data schema, providing types for concerns, relationships and predicates. An initial, extensible classification starts with distinguishing logical and physical concerns, and splitting relationships into categorical, interpretive, mapping and physical. Logical concerns are sub-divided into classifications, classes, instances, properties and topics. Similar sub-divisions exist for the other categories.

The Cosmos schema is backed by rich experience with developing complex re-usable software. Case studies and industrial experience [Mem02] exist, that demonstrate how a Cosmos model can be used as a "semantic hyper-index into work products and other resources" [SR02]. More specifically, Memmert reported [Mem02], how concern space modeling supports software evolution. The model was used for maintaining dependencies and enabled a high-level impact analysis which pointed out, how planned modifications of the system would propagate through concerns and artifacts of different development stages. A simple fixpoint analysis determined a minimal set of consistent changes.

In the future, Cosmos should be compared to our approach using a Concern Interaction Matrix. The intention behind both methods is very similar. From looking at examples, Cosmos seems to suggest finer grained concerns that are connected in a hyper-graph. In comparison, the structure of a Concern Interaction Matrix is simpler, yet connectivity between concerns is higher in this matrix than in a Cosmos model. The concern classification presented in Sect. 14.4.2 combines concern properties and their contribution to relationships. In contrast, Cosmos strictly separates concern classification and relationships.

A thorough comparison would require to apply both methods to the same case study. Such experiment has not yet been performed. It seems that both methods represent valuable experience, which in the future should be merged in one way or another.

Chapter 16

Technology for Views

Development of software engineering environments has largely motived the development of technology for supporting views on different levels. In this chapter, examples of such technology are presented. In the late 1980s and early 1990s the concept of views was defined mostly with respect to database technology for the integration of tools. Important steps on this road are

- The Ph.D. thesis of David Garlan ("views for tools in integrated environments" [Gar87])
- The standard PCTE [ECM90]
- The O_2 OODBMS [SAD94]

Other OODBMSs followed with different flavors of view systems.

Programming languages have had view-like features starting from at least 1989 [SS89]. Many approaches of this field used some kind of a role concept. Many different ideas exist on how to treat object identity in the presence of roles [WdJ95]. Many of these approaches suffer from the lack of modules for encapsulating sets of interacting roles, as already present in the concept of [SS89].

Apart from roles, starting in 1993 ([HO93]) a different branch of research transferred some experience from multi–view environments to programming in general. Multi–dimensional separation of concerns and aspect–oriented programming are some of today's buzzwords in this field. Examples will be given in Sect. 16.3.2.

Sect. 17 relates different uses of view concepts in PIROL to other approaches discussed in this chapter and to further related work.

As an outlook of this chapter, Sect. 17.6 outlines current work by the author on a programming model that combines the lessons learned from many approaches that are subject of this chapter.

16.1 Database related views

16.1.1 "Views for Tools in Integrated Environments"

In his Ph.D. Thesis, David Garlan elaborates what seems to be the first notable and comprehensive technology for views [Gar87]. For the purpose of easing the integration of new tools into an environment, he develops three concepts of views which he calls *display views*, *basic views* and *dynamic views*, of which the first has been implemented in the Genie system[1]. Also for basic and dynamic views quite some details for implementation are given.

Display views

Display views serve the seemingly simple purpose of *deriving* screen presentations in some concrete syntax of a program that is stored using an annotated syntax tree. Issues which complicate this task include:

- New unparsers should be created by a simple declaration of an unparsing scheme.
- Conditional unparsing should be supported, by which a variety of context information should determine the unparsing result.
- Whenever data in the repository is changed, views should be updated incrementally with minimum effort.

The first two requirements are fulfilled by Garlan's unparse language VIZ. The third requirement led to a design of a three–phases transformation. *Unparsing* transforms the abstract syntax tree into another tree carrying elements of concrete syntax. On this tree, *formatting* is performed in a separate phase. Finally, a tool is used for *displaying* the formatted view. Except for the last step, everything happens in the repository, and display views can be cached persistently.

The API, by which tools access a display view, has functions of two categories. The first category allows to handle a display view as a list of text lines, while the second category reflects the internal tree structure.

Basic views

Basic views relate to how tools store their data in the repository. This is done in an object-oriented fashion, i.e., types are defined with attributes and methods. Basic views address the issue, how different tools can share information through a common database yet remain independent. Independence is taken to be important for modularity (for the sake of comprehensibility) and environment evolution (in terms of integrating new tools). Basic views are grouped to *features* and from these modules an environment is built by *merging* all desired definitions.

[1]Publications on Genie seemingly don't exist.

Basic views roughly correspond to classes in object-oriented languages and features are encapsulations of a set of views corresponding to, e.g., Ada packages. The importance of a feature lies in grouping a collection of types such that all type references can be resolved.

The interesting part is the merging of different features which supports the integration of a new tool. During merging, sharing of fields is determined by name–equality, but renaming may also be employed to create correspondence. Merging distinguishes three cases:

- Unshared fields remain local to one view.
- Fields that have the same type in two views may be shared.
- Fields of a collection type my share the contained references but organized in different containers.

Garlan presents field sharing only as an optimization of a more fundamental technique for maintaining inter–view consistency. The essence is a model where an object is composed of different facets according to the views involved. Each facet is an instance of a basic views. In this model some updates remain local to a facet while for other updates events are sent to several facets. A tool always executes in the *context* of a feature which selects by its contained basic views the facet of a given object that should be operated on. For the sake of view consistency, however, the same operation may by a single trigger be executed on different facets, i.e., operations are merged, too. The translator that performs view merging statically coordinates how an event is distributed to several facets.

Features can be parameterized by basic views in order to improve re-use.

This model seems to be sound (with respect to view consistency) and powerful (with respect to integration of independent tools). The major restriction lies in the fact, that all facets of an object in the database are always created simultaneously. In other words, the flexibility of composing views and features is available only during development of an environment. Once an environment has been configured and a database is setup, the number and types of facets for each object type are fixed.

Basic views assume an architecture in which all functionality is always available. A component based architecture might raise difficulties regarding the update of a facet that needs functionality from a tool that is not currently running.

It is also worth noting, that coordination of different views is performed in an event–based style. Events in this concept subsume normal method calls and predefined events like Create, Insert, which are triggered automatically when an instance of the type is modified by a primitive operation. For any event that is in some way shared between two views, the system kernel is said to avoid infinite loops by triggering the event only once for each view.

Dynamic views

The third view concept introduced by Garlan, resembles a general query facility. He presents a language for dynamic view specifications that uses patterns, predicates and selectors to compute sets of objects. The important issue here is that dynamic views are not one-time snapshots of the database, but they are automatically updated, whenever relevant changes in the database occur.

Technically, dynamic views are similar to display views, both perform some derivation in order to produce a new representation of existing data. Both concepts include a mechanism for updating. The main difference lies in the style of the derivation specification and the structure of the resulting view. Display views are based on transformation rules while dynamic views use patterns and predicates, i.e., the former are constructive while the latter are selective. The structure of a display view is a tree while dynamic views are flat collections.

Garlan compares his language for the specification of dynamic views with some tree manipulation languages of that time. It would certainly be interesting to repeat this comparison today against XSL. Maybe, XSL would come closer for defining display views than dynamic views. An obvious problem of XSL in this context is the lack of incremental processing which has been addressed only recently [LN02].

Comparison to concepts of PIROL

Garlan's thesis makes some fundamental contributions but it is hardly cited in recent research. For this reason, the author learned about those concepts only *after* developing the larger part of PIROL. Thus, any similarity between Garlan's work and PIROL is incidental, which makes comparison even more interesting.

For the sake of display views, Garlan represents not only abstract syntax but also concrete syntax in a tree structure. A very similar solution has also been developed by Christian Mattick [Mat02] (cf. the class `SOURCE_CODE`, 13.3.2).

Basic views have a capability similar to Dynamic View Connectors in PIROL. The major difference lies in the dynamic character of DVCs. Clearly, basic views do not require a mechanism like lifting: all facets are created simultaneously. The more static approach has three drawbacks:

1. Facets have to be allocated even if they are never needed.
2. An object can not acquire a facet at a time later than its creation.
3. An object can not have several facets of the same type.

The result may be a bloated database (1), that is not prepared for evolution (2). Another difference between items (1) and (2) concerns the initialization of a facet which may not be possible with meaningful data at object creation time. Basic views enforce all facets to be updated throughout the life-cycle of an object. In PIROL a VO may come to life at any point in time. The accept function may perform some delayed initializing of a newly acquired "facet".

Also item (3) originates from the static translation scheme of basic views. The capability of attaching several VOs of the same type to one RO is regarded as an indispensable feature in PIROL.

The event–based coordination between different basic views can be compared to PIROL's attribute guards, which use the same concept of predefined events. It might also be interesting to analyze the algorithms behind event dispatching in basic views with PIROL's change propagation. From the description in [Gar87] it is not clear, how actually update recursion is avoided.

Update recursion [←8.2.3]

Finally, dynamic views can be compared to derived attributes in PIROL. Both combine properties of a database query with a mechanism for updating. It is not clear, whether Garlan's mechanism allows to attach observers to a dynamic view, i.e, whether a tool can react to changes in a dynamic view. The technique for PIROL's derived attributes is simpler and at the same time more powerful than dynamic views, because it relies on a full programming language (Lua/P) rather than a specialized query language. Dynamic views, furthermore, can have different update strategies: *always*, *on-access* and *once*. A permanence modifier qualifies a dynamic view as *volatile* or *non-volatile*. All derived attributes in PIROL are *volatile* and updated *always*. It might be reasonable to add these choices to PIROL. Lists that are imported in a DVC using the filter construct are a special form of (predicate-based) derived attributes. Filtered lists can be modified directly (presuming consistence with the filter predicate). This is not possible with dynamic views, which are strictly read-only and can be modified only by modifying the underlying data, from which they were derived.

16.1.2 Portable Common Tool Environment

The Portable Common Tool Environment (PCTE) plays an important role within this thesis due to several circumstances: First, PCTE was formed as a standard[2] during the most flourishing era of SEE research. It was an important milestone at the transition from relation databases towards object-oriented databases. PCTE stands in the middle of this transition: it supports a notion of objects with (simple) attributes and typed references, and also allows to use inheritance between object types. Yet, it gives no support for attaching methods to object types, i.e., the object model is purely static. Implied by the devotion to objects and references (in PCTE: links) the primary access to a PCTE object base is by navigation (along links) rather than by filtered access to all instances of a type as it would follow from a relational understanding.

Second, as part of the object management system (OMS) of PCTE a simple but effective view mechanism is included that is based on types or even slices of types. This view mechanism will be described in depth in Sect. 16.1.2.

Third, PCTE is a very comprehensive standard closely related to the ECMA reference model [ECM93] and thus covers more aspects of integration than many other systems[3].

Last but not least, an implementation of PCTE, namely H–PCTE[Kel92], is used as the OMS for PIROL. The reasons for that choice have been given in part II.

However, within PIROL PCTE is used in a way, that slightly deviates from

[2]Firstly only a European standard (ECMA-149), later-on also as an international standard (ISO/IEC 13719-1).

[3]Cf. [NJB97] for an evaluation of different repositories, where PCTE is given very good grades.

the original intention. The most obvious change is an additional architectural layer, the PIROL workbench, that mediates between tools and the OMS. Due to this architectural change the original view mechanism was of no use for the implementation of PIROL. However, for completeness of this elaboration a discussion of the mechanisms provided by PCTE may not be omitted.

Incremental Schema Definition

In Sect. 15.2.1 we have mentioned the issue of open classes. PCTE applies this concept to database technology. PCTE allows to define types in an incremental way. This is to say, that schema definition sets (SDS), although being the capsule of introducing any new types, are not strictly closed: a type defined in one SDS can be *extended* in any other SDS importing this type. This results in different views of the same type, as the original SDS is not at all effected by the extensions, that exist only in the context of the new SDS.

Obviously such distributed definitions are prone to introducing compatibility problems, because no single place exists where all extensions of a given type can be seen. Here SDSs help as scoping construct in that way, that the uniqueness of attribute types is required only within an SDS. Different SDS may introduce equally named types with different specifications. In a context where definitions from different SDS would induce a name clash, a qualified name must be used that consists of an SDS name and the type name.

Aside from name clashes, extending type definitions is unproblematic, as only additive changes are allowed. The question of *when* existing objects will *gain* newly defined attributes has to be investigated in conjunction with PCTE's view mechanism of so-called *working schemas*. Note, that for link types this is not an issue. Applying a link type to an object type does not by itself add anything to objects of that type. It only yields the *capability* of an object to get a link of this type attached at runtime. Instead of supporting a notion of a void reference, links can be either present or completely absent from an object.

Type based Views

When accessing a PCTE objectbase, each process has to identify its working context called its *working schema* (WS). A WS is simply a list of SDSs, whose type definitions should be visible for the calling process. This has a twofold impact: firstly only those types defined in one of the named SDSs are visible. Invisibility of an object type means that any object of an invisible type is itself invisible, unless a visible super-type of the object exists. In that case, the object is seen as being of that super-type, which is just a special case of polymorphism. Note, that PCTE subtypes are strictly additive, as no methods are attached to object types that could be redefined, and also attribute and link types cannot be redefined, either. Any link leading to an object that is invisible within the current context, will itself be invisible within that context. Also note, that the given definition of visibility will never render a link type visible such that a link will point to an invisible object, as a link type cannot be defined if the destination type is not visible in the defining SDS. However, object visibility also depends on access permissions. Only in that case a link can become invisible because the target is invisible. This is one instance of the intricate interactions between different concerns of PCTE: the view mechanism vs. access control.

Second, for an object that is determined to be visible, the set of visible attributes and links still depends on the active SDSs. There might be another SDS that extended the given object type with additional properties, but if this SDS is not listed in the current WS, these properties will not be visible.

The combination of incremental schema definitions and type based views allows tools to add their own data model to the object base and operate only on those values, that can be seen through the corresponding WS. For regular tools in fact the incremental schema definition is more important than the view mechanism, because the tool will naturally query only those properties, that it knows about.

It is a special class of tools for which this mechanism really helps: generic tools that retrieve their data model via the meta data API from the object base. An example of this class is given in [DK95]. Each instance of the tool described in that paper is defined by a WS and some additional parameters. The WS selects the view of the repository to be visible to the tool. By means of reflective type inquiry the tool may traverse and display object structures of types that where not known to the tool implementor. The view mechanism ensures that no irrelevant information clutters the presentation.

PCTE supports two concepts, which deserve comparison. Types can be augmented either by sub-typing (`child type of ...`) or extension (`extends ...`). First of all, the use of the keyword `extends` seems better motivated than its interpretation in Java, where it denotes inheritance. Second, this introduces an interesting choice, as a schema developer may decide, whether a given attribute should be applied to all instances of a given type (`extends`). Alternatively, objects must be created explicitly of the new sub-type, in order to gain the new attributes. An interesting middle road is given by the ability to use a super-type for creation and convert to the sub-type later-on.

16.1.3 O_2 views

Motivated by SEE research, the next step in database technology after PCTE was a series of object-oriented databases, of which O_2 [SAD94] will be presented as an example. Unlike PCTE, O_2 is fully object-oriented. A type in O_2 defines attributes and operations. Views in O_2 are defined at three levels: virtual schemas, virtual classes and virtual attributes.

A *virtual class* is defined by a query over the actual database, or relative to an existing type by a characteristic function. Thus conceptually, the extent of a virtual class is defined relative to the extent of a real class, although notation differs slightly if extents are explicitly available or not[4].

A *virtual schema* is a collection of virtual classes such that a virtual schema applied to a real (data) base will yield a so-called virtual base, a space that is

[4]In O_2 extents exist at the physical but not at the logical level.

dynamically populated with objects from the real base according to the definition of virtual classes.

A *virtual attribute* is a property of a virtual class that is specified by a derivation function.

Virtual schemas preserve object identity, i.e., an object in a view has the same identity as the underlying real object. Virtual schemas can be dynamically activated and deactivated, which switches the context for object structure and object behavior from a real base to the derived virtual base. Virtual schemas can be nested, but only one most specific view can be active at a time.

While a virtual schema is active every object will be classified by the virtual classes of this schema. In case of ambiguity general rules of priority are used for resolution.

Special attention is paid to different forms of updating between a view and the real base. Virtual attributes are re-calculated when their derivation source is modified. Virtual objects may even be re-classified dynamically when the query defining the virtual class yields a different result. Conversely, it is desirable to directly modify (database jargon: update) virtual objects. This was not realized in early versions of O_2, but [AYBdS96] discusses this issue, which has been prototypically implemented in a Master's thesis.

An ambitious setting of database integration is mentioned in [SAD94]. A combination of views, a-posteriori generalization and a role relationship are claimed to allow the integration of databases with different schemas. Unfortunately, essential details of this concept are explicitly ignored in the paper.

History of O_2. First publications on O_2 date 1991 [AB91]. In 1996, [KR96] reports that O_2 be the first and only commercial implementation of an object-oriented view management system. The last published use of O_2 is in [EAMP97]. As of today the company that sold O_2 has disappeared from the market.

Comparing O_2 and PIROL

The intention behind the view concepts of O_2 and DVCs are remarkably similar. Still the realization differs significantly. O_2 is much more determined by database concepts, while PIROL focuses on component technology and object-oriented programming in Lua/P.

More specifically, the mapping between real classes and virtual classes in O_2 is defined by *queries* or characteristic functions. In PIROL *navigation* along object references prevails. Within a DVC the view class to which a given RO is lifted is determined by the static type that is expected in the DVC. In other words, O_2 focuses on mappings between classes only, while DVCs also map associations. The type of a VO is thus determined by the path through which it is reached. predicates that are attached to a view class in PIROL are comparable to the characteristic function of an O_2 view, but in PIROL predicates only play a secondary role. As a result, PIROL can with greater ease admit multiple view classes for the same base class and still resolve view classes unambiguously for common cases.

Virtual attributes in O_2 compare to PIROL's derived attributes. Three policies for modifying virtual attributes are presented in [AYBdS96]: (1) forbid, (2) derive reverse function automatically, (3) let the user supply a reverse function. Option (3) corresponds to the redirect construct in Lua/P. PIROL supports (2) for all variants of uses and filter.

The most restricting difference is the lack of added attributes in O_2 views. All persistent data must be defined in a real schema. In contrast, Lua/P provides the adds keyword for the purpose of introducing new attributes in a view, which are not derived from RO attributes. By means of added attributes, a VO becomes a first class entity instead of being merely derived from underlying data. With added attributes, it makes sense to maintain several VOs relating to the same RO either within the same CO or in difference COs, even several COs of the same type are very useful, as we have seen.

For virtual objects that can not add data to existing objects all this is of little use. In fact, O_2 does not supports multiple instances of view objects per real object nor multiple view instances.

16.1.4 MultiView

The goal presented in [KR96] could not be phrased better for PIROL:

> "...achieving interoperability by hiding the idiosyncrasies of component systems to be integrated into one unified, yet federated system."

Superficially, the concepts of MultiView and O_2 are very similar. The first noticeable difference is MultiView's capability of capacity–augmenting virtual classes. This corresponds to our criticism of O_2 concerning the addition of attributes. Internally, MultiView is built quite differently from O_2. MultiView creates a single inheritance hierarchy of all classes, real and virtual. Objects are implemented using a technique of object slicing, where a *conceptual object* serves as a point of reference ensuring a single unique OID. Quite similar to PIROL's COs a conceptual object in MultiView also groups a set of implementation objects. This allows multiple classification of objects. Feature access is then dispatched between the different facets of an object coordinated by the conceptual object.

Also like PIROL, MultiView supports incremental propagation of updates. Just like O_2, MultiView differs from PIROL by its focus on queries instead of navigation. More attention is paid on automatic acquisition and dropping of additional facets based on state changes, which are detected by view defining queries. Instead, MultiView lacks the grouping capabilities of DVC which reify sets of collaborating objects.

In [KR96] the deputy mechanism of [PK95] is quoted and from that comparison it might be a step closer to PIROL because it uses a kind of role objects rather than unifying all classes into a global inheritance hierarchy. However, within this thesis, the deputy mechanism is not further examined.

16.1.5 Views in Chimera

The presentation of view concepts for object-oriented database systems is concluded by a presentation of a model that is not only promising in functionality but also well-founded by a formal definition [GBCGM97]. That paper also gives a good overview of the design dimensions of views for object-oriented databases.

The Chimera database system is based on three programming models: an object-oriented data model, deductive rules and an active language for reactive processing.

Views in Chimera are distinguished into object–preserving views, that contain a selection of existing objects maintaining their original identity, object–generating views that create new persistent objects and set–tuple views containing transient derived data with no persistent identity.

In Chimera, just like MultiView, view classes may introduce additional non-derived attributes. Unlike MultiView, Chimera keeps distinct inheritance hierarchies of classes and views. Both hierarchies are connected by a view derivation hierarchy. The difference between the relationships inheritance and view derivation is clearly stated: for inheritance, signature sub-typing must go together with forming subsets of extents. This constraint is not enforced for the view derivation relationship.

Objects can be members of several most specific classes including view classes, which facilitates the separation of inheritance hierarchies.

Just like classes are combined to schemas, views are combined to view schemas. This allows applications to operate on view schemas and views with no difference to schemas and classes. Schemas must be closed with respect to referenced types. Closures of view schemas are obtained by including identity views (views that map classes without modification) of (transitively) referenced types.

For selecting the appropriate facet of an object Chimera considers the context of a reference. Not only the active view schema is used, but class selection and feature dispatch also takes into account the static type of a reference. This allows the same object to appear in different roles, i.e., as an instance of several views without reverting to dubious priority rules like in O_2.

Comparing Chimera views to PIROL

From the models investigated in this chapter, Chimera is probably closest to PIROL. Apart from single features, the very nature of combining different paradigms into one model speaks of a similar spirit.

Partly shared identity between ROs and VOs, added attributes, a distinct role-of relationship, grouping view classes to modules (DVC), lifting with respected to the static type of reference, all these are PIROL concepts that correspond also to Chimera views.

PIROL does not support object–generating views, although a role object that uses no features of its base RO may simulate this concept. Transient objects are also handled slightly different. PIROL does not require explicit schema closure, allowing a VO to refer directly to an RO. Explicit schema closure might help disambiguate some cases of lifting and lowering, but this would come for a price of drawing an explicit border around reachable classes. Naturally all classes in the PIROL meta model are connected and especially generic tools like PON would suffer in flexibility if visible classes would have to be determined in advance.

On the back side, at the time [GBCGM97] was published, some essential features had only been planned for Chimera, with no concrete design yet. Among these are: update propagation and the use of triggers for view coordination, especially with respect to integrity constraints. It is unclear, whether callbacks into the client program are supported as a means for update propagation. Update propagation including notification to the client program is central in PIROL and some form of triggers for integrity constraints exists as attribute guards. A final decisive difference is the lack of instantiability regarding view schemas in Chimera, which would correspond to instantiating DVCs in PIROL. It has been underlined repeatedly that this feature is indispensable for PIROL.

16.1.6 Tolerating Inconsistency

Several techniques have been developed to explicitly tolerate inconsistency in a database [Bal91]. Rules in GTSL [Emm96], predicates in APPL/A [SHO95], and especially the possibility to temporarily relax predicate enforcement (suspend and allow) are examples of such technology. PIROL's primary goal is to avoid redundancy, which could possibly lead to inconsistency. Experience with strictly structure oriented environments showed that editing naturally goes through many stages of inconsistency. Yet, PIROL tends to solve such conflict already at the object level, avoiding complex consistency constraints to be checked across large documents.

Managing source code [←13.3.2]

16.2 Views and Subjectivity in Programming

Apart from database technology, SEE research has also stimulated advances in programming languages. This also relates to views as it can be demonstrated by the example of RPDE^3. This environment framework was designed with the goal of supporting changes concerning function, data, supported languages and hardware/operating systems [OH90]. In an evaluation [HSS89], the authors conclude that the object-oriented paradigm is indeed well suited for modeling the entities of a programming environment — RPDE^3 operates on an object-oriented AST. However, to obtain the desired flexibility they add some concepts to the object-oriented paradigm (the papers give no hint at which language was actually used). Some of these extensions are:

1. Fragments — allowing refinement below the level of methods without modifying original sources.
2. Structure-bound messages — these are dispatched by the run-time system by traversing the AST up-to a node with the desired capability.
3. Roles — different interfaces for a class.

4. Envelopes — wrapper objects roughly comparable to the composition filter approach [BAWY95].

Two authors from the RPDE[3] context, Shilling and Sweeney, propose to apply view concepts to object-oriented programming in general [SS89]. They postulate these three steps:

1. Objects have multiple interfaces

2. For different facets visibility of instance variables is explicitly controlled with respect to the three options *hidden*, *read–only* and *read–write*. Such controlling includes resolution of name clashes.

3. Multiple copies of instance variable sets are allowed.

Step (1) should be quite familiar to the reader by now. This also connects to current component technology à la CCM [OMA99] (facets). Step (2) transfers findings regarding multiple inheritance to multiple view–base relationships. Finally, step (3) transcends most view models from object-oriented databases and paves the way to models that are based on distinguishable role objects. This adds dynamics to the model, because roles may explicitly *join* and leave a view.

16.2.1 Roles

The mechanism of role objects dates back to at least 1986 [Lie86]. Only a year later Stein proved the congruence between delegation and inheritance [Ste87], and Ungar and Smith presented their prototype based language Self [US87]. Authors from these three works have gathered in 1989 to write the "Treaty of Orlando"[SLU89], which makes explicit the parameters of language design concerning different mechanisms of sharing such as inheritance and delegation.

After a mostly technical discussion, the early 1990's introduced many names that should give meaning and intuition for the given techniques. An interesting, though seldomly cited paper, is [RS91] (1991) which surprises by its title "Aspects: extending objects to support multiple, independent roles". This happened clearly before the rise of AOP [KLM+97].

Other works discuss roles from the viewpoint of database technology, like [WdJ95, WdJS95](1995), which shift the focus to issues of object identifiers (should a role and its parent object be considered the same object?) and database schema evolution. A good overview of roles and related concepts can be found in [Bar98].

Role objects provide expressiveness and flexibility. Language support for roles is easily justified by considering the extra complication that is introduced if role concepts are simulated in classical, class based languages (cf. [Fow99a, DBW00]). On the other hand, some new problems arise from role objects:

1. If role objects share properties of their base object, changing the base (or parent) link might change the interface of the role object.

2. Different authors disagree whether roles should be allowed to modify properties of their base object or just add new properties.

3. How can a role-base relationship be qualified to have access to a specialization interface which is otherwise hidden to normal clients?

4. If a base object may have attached several roles, how (if at all) is navigation from a base to a specific role realized?

A solution to items (1) and (2) is, for instance, given by the programming language Lava [Kni99]. In order to guarantee presence of a parent object to which methods can be delegated, Lava introduces the modifier `mandatory` which ensures, that at run-time such an attribute can never become a null reference. Modification or pure addition can be controlled by choosing either of the modifiers `delegatee` or `consultee`. The former introduces a parent link that allows overriding of methods by a role, while the latter introduces pure method forwarding, where methods are executed in the context of the base object without redirection of self calls. Of course, for both styles of role-base relation, the base object when accessed directly always remains unchanged.

Item (3) is motivated in [SM95] and a conceptual solution is presented. The author is not aware of any current implementations that truly address this issue. The problem may, however, be more than just a technical issue. As Steyaert and De Meuter state, object–based inheritance breaches encapsulation [SM95]. The problem might be, that some weakening of encapsulation is exactly wanted, but we are unable to technically define, *to which extent* this should be allowed.

Item (4) relates to the fact, that the use of role objects may yield a proliferation of identifiable objects, which adds new complexity to software designs. [BD96] contains a discussion of secondary keys that may select roles from a given base object. This hints at the observation, that roles discussed in isolation — i.e., as single object views — are not particularly helpful for structuring systems, but roles should always be seen as *participants in a collaboration*.

16.2.2 Collaborations and Subjects

Starting in 1992 [AR92], Reenskaug developed his set of methods for role modeling, culminating in OORAM (1996) [Ree96]. In this concept, roles are seen as collaborators in partial models of system behavior, which are called role models. The system is then composed of several role models, a translation that synthesizes composite roles from elementary roles.

Role modeling can be mapped to implementation in at least three different ways:

1. Perform the role synthesis manually and implement the final composite roles a classes.

2. Use special language features to let a compiler or translator perform the synthesis at compile time.

3. Use a more dynamic language such that roles exist at runtime (as role objects according to the previous discussion).

Solution (1) is certainly cumbersome and impedes maintenance.

Static role model composition

Two approaches shall be presented that realize solution (2) of the above list: generative programming and subject-oriented programming.

Generative programming. For some time, techniques using C++ templates [VN96, SB98] appeared as the best match to keep the structure of role models also for the implementation. These techniques synthesis collaborations using template parameters and classes using inheritance. The major drawback is the lack of independence between different role models. If different role models are supposed to share methods of a common class, names and signatures have to be identical. From this follows, that such solutions require an global agreement on sharable features, which is contrary to the desired independence.

Subject-oriented programming. Independence is significantly improved by subject-oriented programming [HO93] and its successor, the hyperspace approach [TOHS99, TO00]. Subjects are (partial) models that are developed independently. Subject composition uses explicit composition rules in order to define correspondence and combination of classes and features. Subject-oriented programming supersedes the above generative approach by providing means for adaptation. Adaptation is crucial if mismatches shall be anticipated that inevitably result from independent development.

Subject-oriented programming also fixes a typing problem of the generative approach, which detects certain typing errors only when compiling the whole application. In contrast, each subject is *declaratively complete*, i.e., it has to declare all necessary features at least as abstract methods, such that static type checking can be performed on each subject in isolation. Composing an application only requires to check type correctness of feature composition.

The hyperspace approach advances independence even further, by splitting composition into two conceptual phases: first, existing modules (Java packages) are restructured into concerns and features. This extracts and partitions elements of existing software. Second, the application is composed of these concerns and features. Obviously, such on-demand re-modularization gives even more flexibility for reuse of existing parts.

These approaches have in common that several views of a system can be expressed, which contain a set of collaborating (partial) classes. The application is generated at compile time by synthesizing partial classes into effective classes. The hyperspace approach provides views not only at the level of independent collaborations, but also as intermediate, conceptual system structure that is defined by mapping units of implementation to concerns and features.

Partial classes can in these models be interpreted as a restricted form of roles. The restriction lies in their static composition, which limits their use to development time, as roles don't exist at run-time.

On the notion "Subject". In a footnote of [HO93], Harrison and Ossher give an explanation, why they chose the term "subject", which is reproduced here in full:

> "The term subject differs somewhat from its use by Coad and Yourdon [CY91], although both usages share the idea or reflecting a smaller, more focussed perception of a complex shared model. We avoided the similar term view in order to emphasize the stronger philosophical similarity with non-classical philosophical trends that emphasize the idea that subjective perception is more than just a view filtering of some objective reality. The perception adds to and transforms that reality so that the world as perceived by a body of perceptive agents is more than the world in isolation."

Despite this disclaimer, the notion can well be understood as a play on the double meaning of "subject": as an opposite to object and in the sense of topic, the latter being close to Coad and Yourdon's concept of grouping classes and objects to a common theme). In this thesis, the term view is seen in a broad interpretation which includes the ideas quoted above.

Dynamic role model composition

Implementation support that fits into category (3) of the above list is only now emerging. In particular the work on Adaptive Plug&Play Components [ML98] and Pluggable Composite Adapters [MSL01] has already been cited, because these have directly influence the development of the Dynamic View Connector model. These approaches introduce instantiable collaborations (also called: adapters, connectors) that have the responsibility of maintaining the relationship between separate objects that can be interpreted as role-base pairs. In [MSL01] the notion of *lifting* is introduced, which refers to the context specific retrieval of a role object for a given base object. Lifting gives an answer to the problem of role proliferation and identification, stated above.

A successor of PCA, which is currently being developed, will be presented briefly in Sect. 17.5.

16.2.3 Roles and collaborations in PIROL

Dynamic View Connectors define a disciplined model of roles.

View objects (VO) are implemented as role objects of their respective RO. Import of features from the RO is controlled by the uses and filter constructs. Therefore, a VO may share features with its base, but it need not conform to the interface of its base class, since features may be hidden (not imported) and renamed. VOs may have additional attributes declared by the adds construct. Many other effects, like e.g., read-only import can be achieved by the redirect construct. Methods are dispatched from a VO to its base using true delegation, thus allowing a template and hook style, where a template method is shared from the base yet overriding a hook that is called by the template method.

Delegation in DVC [←10.3.4]

In Lua/P, the role-base relationship is mostly under the control of the run-time system. The only means for attaching a role to a base is by the lifting operation which is invoked implicitly, whenever a base object enters the scope of a connector that has a matching role definition for the base object. The base link of an existing role object is never changed. Still the capability of dynamically attaching new roles to a base is of decisive importance. Note, that no type checker exists for Lua/P, but still, conceptually, Lua/P is a statically type safe language. The interface of an existing role object never changes.

The major difference between PCA and DVC lies in the focus on method based composition (PCA) versus data-centric composition (DVC). Furthermore, DVCs have the following restrictions: there is no way of staged connectors, that is, connectors on top of connectors. Second, a DVC is meant for connecting an external tool to the repository. Therefore, it cannot connect a collaboration written in a separate Lua/P module to the repository. In contrast, the JADE preprocessor for PCA [Hau00] can even combine several collaborations using a single adapter.

Views as defined by Dynamic View Connectors

Returning to our overall theme of views, DVC as an extension of an object-oriented programming language can now be summarized as follows.

A VO is a view of an RO, which may share, hide, rename and add properties. Method execution on a VO preserves the VO identity throughout self calls thus supporting method overriding. Accessing a pure RO is ignorant of any VOs that might exist. However, accessing an RO within the context of a DVC automatically lifts the RO to the corresponding VO. Object identity of a VO is a triple of ROID, connector ID and view class. Thus a connector defines a compound view in the sense of providing the context that identifies role objects for a given base object.

It has been a principle design guideline throughout, to concentrate knowledge about the RO-VO binding to the connector and to leave all actual translations to the run-time system. Thus, a tool runs in the context of a view as defined by a DVC instance without perceiving the difference between the virtual repository, on which it operates, compared to a real repository.

Connectors can be used either for connecting a tool to the environment, or for implementing collaborative behavior of repository objects while restructuring the repository model to a collaboration model that might be more suitable for the behavior. Also re-use of complex behavior benefits from such structure mapping. Connector inheritance may be used to separate abstract behavior, structure mappings and behavior refinements.

16.3 Towards Views for Improved Modularity of Behavior

16.3.1 Implicit invocation

Another technique that has in part evolved from experience in SEE research concerns implicit invocation. There is a common agreement that the FIELD environment pretty much laid the grounds for connecting components using synchronous and asynchronous messages [Rei90]. In this approach, pattern registration is used for subscribing to certain sets of events. In 1992, Garlan, Kaiser and Notkin suggested to generally adopt that technique for decomposition and composition of software [GKN92]. They coin the notion of "toolies" for pieces of system function. Toolies operate on shared abstract data structures. Operations on the data structure correspond to events to which any toolie in the system may react. In their paper, the authors give no precise model of the interaction between data structures and toolies. In particular, it is not perfectly clear, how subscription to events takes place and what mechanisms are to be provided by the run-time system.

Instead, they take on the discussion of Parnas in [Par72] concerning system modularization, comparing his preferred solution with another one based on toolies. Their argument is quite convincing, that toolies improve the evolvability of a system, since they reconcile loose coupling with an effective integration that enables an acceptable performance.

Conceptually, the work on toolies resembles later works on a tool–automaton–material metaphor [Zül98][5]. However, this metaphor is only a special style of classical object-oriented design. In contrast, toolies extend the object-oriented model to also include event based communication. In [GKN92], the authors also discuss uses of observer mechanisms, e.g., in the Smalltalk-80 environment [KP88]. They come to the conclusion, that the run-time system can cover quite some complexity if trigger based programming is embedded into the programming language, but the fundamental mechanisms can also be simulated using a traditional object-oriented language. It is also a matter of encouraging developers to apply the trigger based style, which is more likely with special support from language and development environment. LOOPS is mentioned as a programming language that directly supports "access oriented programming" [SBK86], which is in fact based on events.

In the same year as the work on toolies, Sullivan and Notkin propose the Mediator style for environment integration [SN92]. Also mediators rely on implicit invocation using events. Still the proposed system structure differs slightly. Toolies are suggested as direct counterparts to active data structures. The mediators approach does not distinguish data and tools but introduces explicit mediator components, which coordinate communication.

[5]The WAM metaphor, according to the German "Werkzeug, Automat, Material", was initially devised with "aspect" instead of "automaton" (cf. [BZ89, Her93]). Here, "aspect" (also called "property class") referred to interfaces that defined an abstract view of how certain materials could be handled by tools. A concept that is also suggested by [GHJV95] as "abstract coupling".

Method-call interception

In [GKN92], a very decisive point has to be read "between the lines": when presenting an Omit toolie, which is to prevent the action performed by another toolie, the authors imply, that an event, which is supposed to cause an action, can also be *canceled* by any toolie in the system. This speaks of a very open system design, where calls to a method can be *intercepted* by another component. As recently pointed out by Lämmel [Läm02], method–call interception is the fundamental mechanism underlying the model of aspect-oriented programming.

Another early approach to method interception originates from Myers' Artists [Mye83]. According to [TBC+88], Loops [SBK86] "binds the equivalent of artists to objects using a specialized form of inheritance called *annotation*". This underlines, how implicit invocation mediates between architecture and multi-paradigm language design.

16.3.2 Aspects

Toolies suggest two universal views on a system: a data structure view and a function view. Aspect-oriented programming (AOP) generalizes this to the promise, that many different views can be modularized as aspects, that in other approaches *cut across* the system structure. The most prominent language for aspect-oriented programming is AspectJ [KHH+01]. AspectJ supports the open class concept discussed in Sect. 15.2.1. More importantly, it features method–call interception using the concepts of pointcuts and advice. There is a shift in focus from prior approaches, as pointcuts abstract over certain points in the run-time call graph of an application. By giving names to these points, it is now possible to add pieces of code — advice — to these points.

At a first glance, AspectJ has the same capabilities as implicit invocation with call interception. The special power of AspectJ is, however, based on the abundant features for specifying pointcuts. A continuously growing set of predicates and modifiers exists, by which pointcuts can be defined for different kinds of events, for sets of classes and methods and many more conditions. This set of features has grown according to the needs of a notable community of AspectJ programmers. The author considers this also the weak spot of AspectJ, since this growing set of features speaks against a clear and orthogonal language design and has confused people who have been looking at AspectJ from time to time.[6]

Instead of iterating through all those specifiers, the fundamental contributions shall be discussed. The control flow of object-oriented programs tends to become really complex[7]. Being able to explicitly refer to points and situations in this control flow is a value in its own right. The work on implicit contexts is even more explicit concerning the history of interactions at any given point during program execution [WM00].

[6]The approach of Gybels [Gyb02] seems to provide better orthogonality. He proposes logic meta programming for specifying pointcuts.

[7]Unfortunately, the author lost the source of this appropriately pointed quotation: "Reading the control flow of an object-oriented program is like reading a map through a straw".

More importantly, all that discussion about aspect-oriented programming has finally lifted a large number of concerns to the level of notable problems of design and implementation for which modularity is extremely difficult to achieve, yet vital for system evolution. Aspects like persistence, logging, synchronization, replication, security, to name only the most prominent examples, had before AOP to be coded "between the lines" of the actual algorithms.

The presentation in Part. II may have given an idea, how some of these aspects lead to a program structure that in no way resembles those nice textbook examples, by which students shall learn "good" design. Such problems are hard, and any help is welcome for improving modularity for the listed aspects. It is an observation by several authors, that such complexity, which justifies the use of AOP, is mostly found in infrastructure software (cf. [ACP02, GR02]).

Technically, AOP is realized using a so-called weaver, a tool that merges code fragments from different modules into the final program, which is usually done statically at compile time. The details of this process are, however, of little interest for the discussion at hand. What is more important here, is the fact, that AOP helps to identify many of concerns, which arise during requirements elicitation, and to implement those concerns as modules with well defined borders, which was not possible before.

And yes, these concerns are views, too (which should already be clear by the terminology: "aspect", the Latin `aspectus` translating directly to look, sight, view). In fact, there is a rather broad understanding of the word aspect. The term "aspect-oriented software development" [AOS02] is meant to subsume also subject-oriented programming and many more approaches. A more focussed interpretation is usually given to the notion aspect-oriented programming, in which aspects more often than not refer to so-called non-functional requirements. Any notion of "non-functional" something in software engineering is quite debatable, because many of these properties directly relate to system functions, too. The point is, this class of aspects does not describe the main concept of *what* a system is doing, but rather different facets of *how* this is performed, additional properties, one could say.

Apart from this picking on words, aspects denote a third kind of views beside data views and function/behavior views. Aspects try to make explicit within source code, many of those properties that used to be scattered, i.e., spread around many different modules of the system.

16.3.3 Views need context

One way of describing a view concept is defining what makes a context that discriminates one view from others.

In an SEE a view context can be a tool or a user role. Such a context is comparatively static and selects a quite stable view of the repository. Objects are identified by a tuple of view identifier and object identifier.

In the hyperspace approach, context is defined by a feature, a concern, or a hyperslice, all of which are static perspectives.

Collaborations can be more dynamic and here a context may be a scenario that is currently executing.

Aspect-oriented programming supports explicit referral to a control flow, within which behavior will be different than outside that control flow.

All concepts of context share the capability to avoid explicit conditional programming[8], because the program is "aware" of its current situation. Lifting and lowering according to PCA, DVC and Object Teams implement a style of *contextual dispatch* that complements dynamic binding in standard object-oriented programming. Active connectors (adapters, teams...) determine the choice of role, to which an object should be lifted. This gives further strength to Meyer's "single choice principle" [Mey97]: Any alternative that exists in a program and can be chosen at run-time should only affect the single location in source code, that actually decides. After this decision, the run-time system should be responsible for executing behavior according to the choice taken. In standard object-oriented programming the choice is in creating an instance of a concrete class. From there on, dynamic binding will dispatch to methods corresponding to that choice. Using DVCs or a related model, the choice is in entering or activating a connector and subsequent dispatch to appropriate roles is done by lifting to the context of that active connector.

Object Teams [17.6→]

Views can be made manifest in programming, by first-class entities, which represent a context. Activation of a context has impact on the behavior of the system. Views and explicit context in concert provide a powerful concept for modularization beyond standard object-oriented techniques.

[8]Cf. [Orl01] with the wonderful sub-title "If Statement Considered Harmful".

Chapter 17

Views in PIROL and beyond

Part II has presented PIROL based on those high-level views that were called "concerns". It is to be shown now, that also the implementation supports views in very many ways. These views are presented according to the following two categories:

1. Mappings and translations between different representations and "physical state" of common concepts.
2. Contexts that determine visibility, appearance and behavior of contained elements.

17.1 Mappings and Translations

17.1.1 Inter-language working

In Sect. 14.2.2 the co-operation between parts of the system written in different programming languages has been evaluated. In this architecture, Lua and C have quite strong coupling. Mostly, the Lua encapsulation of H–PCTE could be called an abstract view of that subsystem. Translations concern parameter conversions and the different call disciplines (Lua passes parameters on an explicit stack).

Marshalling [←7.3.2]

Marshalling is a central concept, by which interoperability is achieved, which is the capability to access the workbench from any programming language that has a mapping for the intermediate representation used across MSG. For simple data, marshalling is straight forward. Transmitting, references through marshaled data requires extra effort. Sect. 7.4.2 presented the interplay of PCTE references, ROIDs and keep-alive links. Motivated by performance issues, Sect. 14.1.2 showed, that transmitting pairs of ROID and class name is a reasonable, though not crucial concept. Demarshalling, which is done in a library for tool implementation, relies on reflective access to class names and employs a proxy cache for reference resolution.

Proxy classes [←7.3.2]

Proxies provide a transparent view on remote objects. As discussed in the context of OODBMS, updatability of views is a decisive issue. Proxies can be modified directly and react to changes in the repository. Thus, the claim of transparency is justified.

17.1.2 Representatives

Marshalling and proxies are standard techniques in component based systems. The application of this technique in various directions is an architectural concept, which goes beyond technical issues.

Tool representative [←8.2.3]

A tool representative is a local view for an external tool. Methods can be delegated from a representative to a running tool process. It is the tool's responsibility to maintain consistency between the state of its representative and its internal state, because this is a means of exporting state from the tool to the repository. Most ROs in the repository are intrinsic objects, of which other representations and views can be derived. A tool representative — in part — is a view of an external entity, the tool. The restriction is to say, that some attributes even of a representative may be intrinsic, too. Certain configuration information is stored under the responsibility of the workbench. This leads to an interpretation of representatives as ports for two–way communication between workbench and tool.

User representative [←9.1.1, 9.1.8]

Within the workbench, tool representatives refer to a higher architectural layer. User representatives provide an interface to PCTE's access control and to inter–workbench communication.

A `WORKBENCH` instance, finally, serves as an interface to the enclosing workbench context.

All representatives involve some kind of translations. Tools, users, groups and workbenches are identified by different IDs at different levels. E.g., a tool has a PCTE reference, ROID, a OS process ID, a MSG client ID, and a language specific ID for each proxy within a client. For the sake of transaction support, a tool should also have a PCTE process associated (cf. 9.1.7 on page 131). All these identities have to be kept consistent in order to achieve greatest possible consistency.

17.1.3 Derived data

Domain specific mappings can be integrated by means of derived attributes and Dynamic View Connectors. The latter will be discussed further below. Derived attributes can be used to perform arbitrary computations, but the example of a routine signature implemented as a derived attribute shows that this mechanism is well suited for translating between different representations like a structured and a string based representation.

17.2 The Software Process as Context

Process integration [←8.1.1]

User and workbench representatives gave hints at a dimension of PIROL that tends to stand back in technical discussions. Process integration originally motivated the multi–view capability of PIROL. From [GHJK95]:

> "The fundamental concept of PIROL might be the concept of distinct **views onto the repository**.
>
> Different project members shall . . .

> - when carrying out different activities
> - incorporating different roles within the project
> - using different tools
> - and different tool-modes
>
> . . . always be presented exactly that information in the most suitable visualization as it is needed to do their job."

Support for views in this sense is implemented — or encapsulated from other layers — by the ROCM package PROCESSES. This comprises

Package PROCESSES [←4.1.1]

- Access control
- A generalized mail service
- Association to a current PROJECT
- A document state machine
- A customized set of available tools (via class WORKBENCH)

In this interpretation, a view is a *configuration* of the environment. Such views define, what objects and documents are visible, and what can be done to them.

17.3 Other representations

Structured data in the repository can also be translated into several external representations.

Source code. Subclasses of SOURCE_PRODUCER derive a textual software representation, which is stored in SOURCECODE instances and used for source code editing. Maintaining consistency between structured data and source code involves several techniques and also user interaction as presented in Sect. Sect. 13.3.2.

Class SOURCE_PRODUCER [←13.3.2]

ROCM. A special case of source code are different representations of PIROL's meta model, the ROCM. HTML documentation for the ROCM is generated in an ad-hoc style by simple text processing of Lua/P input files using pattern matching. This could be replaced by a more structured approach, which uses the reflective definition using ROs of type ROCM_CLASS and related. While HTML generation and export of documentation from these objects is implemented it currently has a significant problem: comments from Lua/P files are not stored in the repository. Thus, documentation only shows interfaces, not comments.

This technique is, however, used for generating Java proxy classes for repository classes. For this task, all necessary information is available in the repository and mentioned proxygen script is a very simple yet effective proxy generator.

Proxy generator [←7.4.4]

COFS [←13.4.1] **Virtual files.** The CO file system (COFS) allows to access repository data as regular files. This technique is a technical mapping between two disjoint worlds. Also the contents of virtual files requires some mapping like, e.g., the structure to source code mapping mentioned just above.

PIROLWEB [←13.4.2] **XML and HTML representations.** Work on exporting repository data via a XML–HTML transformation has just started. As a result, Web–views will be easily generated from repository data. Supporting updates from the Web is currently not planned.

17.4 Documents and Virtual Repositories

Conceptual objects (CO) were the first step towards reconciling fine grained objects and documents — two fundamentally different views. Plain COs rely on a tool to interpret its information. This is largely automated by DVCs. It should be noted, that plain COs are still useful for certain tools. COFS is an example that needs no sophisticated support from DVCs but works fine with COs. A reason for using plain COs is of course performance, since COs don't require automatic management of view consistency, as it is performed by a DVC.

17.4.1 Dynamic View Connectors

In Chap. 10, Dynamic View Connectors have been introduced as a means to decouple the components of PIROL. This can be seen as a first step on the road of unifying several concepts of views.

DVCs unite ideas from database views [ECM90, Gar87, SAD94] and the AP&PC programming model [ML98]. Due to their primary goal of tool integration in a repository based environment, DVCs put more emphasis on structural relations than AP&PCs do. But despite their data-centric nature, DVCs exhibit the following dynamic properties:

1. Notifications propagate changes from an RO to each view it is contained in.
2. VOs may be modified resulting in changes in the underlying ROs.
3. Views can be created at runtime creating also new scopes for added attributes.
4. VO methods add new behavior to the meta model.

In comparison to some of the view concepts in database technology, DVCs put more emphasis on navigation between objects than on queries. Query-like capabilities are, however, included with both uses of predicates: class predicates and filter predicates.

The expressiveness of DVCs is optimized for adaptations of mismatching structures. Such adaptations may re-arrange class structures regarding attributes and aggregations (including special treatment of lists). Also the inheritance structure of a model can be re-arranged in multiple ways. On class level, these mappings may even be ambiguous, as long as ambiguity can be resolved at runtime using either (1) the static type that is requested within the DVC or (2) a class predicate. Concerning the expressiveness for structural mappings, DVCs are unique among the mentioned concepts.

DVCs differ from any pure programming model by their close integration with other properties like persistence and change propagation. This is crucial for achieving connector transparency required by the concept of "virtual repositories".

When comparing DVCs to programming models from the field of aspect-oriented programming, the most obvious difference is the disability of DVCs to externally *modify* the behavior of an existing implementation. While this simply was not needed for tool integration via DVCs, inclusion of such mechanisms is the natural next step after AP&PC and DVC.

17.5 LAC

The direct successor of Adaptive Play-and-Play Components [ML98] was Aspectual Components as described in [LLM99]. The main advance of Aspectual Components was to add to the AP&PC model a special style of advice weaving similar to AspectJ[Asp]. Within a strict separation of base, collaboration and connector, the main constructs are expected methods and replacements. The declaration of expected methods guarantees that modules are declaratively complete. Method replacements allow to override base methods by collaboration methods, while only the connector knows both sides.

"declaratively complete" [←16.2.2]

For Aspectual Components as described in [LLM99] no compiler has ever been written. The experience from developing the DVC interpreter, however, paved the road for the development of a prototypical interpreter for a variant of Aspectual Components, that differs in its concrete syntax, because it is based on Lua instead of Java, but semantically adheres to the model of [LLM99]. The resulting language LAC (Lua Aspectual Components) [HM01] elaborates and extends some issues that have only been touched marginally in [LLM99].

The new feature — as compared to DVC — is advice weaving which in its Lua-based implementation is a small meta program: a base class and a participant class (the latter from an aspectual component) are integrated by matching all method names of the base class against a given name pattern. Each matching method is then replaced by the participant method given in the binding specification. An important detail concerns the ability of a replacement method to invoke the original method which it replaces. This is further complicated by the fact, that one replacement may replace several base methods. The solution in Lua is, to wrap the participant method each time it replaces a base method, using a function closure that stores a reference to the original method. By this technique each original method is accessible only for its specific replacement wrapper.

A special feature of LAC is its distinction of static, dynamic and singleton connectors. Static connectors are most similar to AspectJ as they permanently modify base classes for a given application. In such an application, base classes only exist in their modified versions. The other extreme, dynamic connectors, are closer to DVCs. They keep the base unmodified, creating only a view that incorporates the changes. Unlike DVCs, a dynamic connector in LAC can, however, be activated. Activating a connector has the effect, that all affected base classes are temporarily replaced by their modified version. This cannot be achieved with DVCs, which have effect only when using VOs. Any RO that is used as an RO is totally unaffected by and DVC. Singleton connectors, finally, are provided for convenience. They have the same dynamic semantics as dynamic connectors. However, dynamic connectors are a special form of classes, that need to be instantiated prior to usage. Singleton connectors already define exactly one instance. They neither need nor must be instantiated.

DVC and LAC base upon the technique of role objects with delegation. In both models a role object may override any base method without the need of declaring so. Some details of LAC can be found in [HM01].

17.6 Object Teams

After demonstrating the soundness of the Aspectual Components model using LAC, its concepts have been reworked with the goal of a better integration with standard object-oriented concepts. The model has been renamed to Object Teams and is currently being integrated into Java.

Also the terminology is changed such as to reduce the usage of methodological notions like participant and expected method in favor of a minimal set of technical notions. Now certain classes *can be used as* participant classes, and certain methods can be interpreted as expected methods, but a better orthogonality is achieved without notions, that already carry too much meaning. For the integration of specific features into the Java language, a concern interaction matrix, as proposed in this thesis, is being elaborated for analysis of interaction between existing features of Java and those of Object Teams.

The central concepts are:

- Role-objects (declared as `class R1 playedBy B1`).
- Open classes/packages: incomplete units that obtain their missing parts via object inheritance.
- Collaborations and connectors are packages with similar properties. Such packages are subject to refinement by child–packages. The distinction collaboration/connector is no longer explicit in the language. In Object Teams the distinction is only of methodological value. Technically, concepts can be mixed.
  In Java, these packages are mapped to classes with inner classes.
- Two kinds of binding are supported: callout (delegation), callin (advice weaving).
- Callin are further specified as one of:
  - before, after, replace

Object Teams support different mechanisms for extending a class. Further analysis should, e.g., compare these mechanisms to the choice between extending and inheritance in PCTE.

Extend vs. inherit [←16.1.2]

17.7 Lessons learned and the Future

The development of PIROL has been highly explorative. Techniques have been developed, which are highly customized for open integrated environments. The development of Object Teams exploits much of the experience from PIROL's development and from many other approaches to aspect-oriented software development. In PIROL new features were introduced when needed without much ado. Object Teams are a concept, which is being consolidated and will be defended against such "featurism" in order to keep language design clean. Both kinds of development are needed: Object Teams will make collaboration based separation of concerns available to programmers and bear the potential to attract a significant community of developers. PIROL had a different goal. PIROL proved the feasibility of an SEE based on a fine grained object-oriented meta model, supporting many different views. Development of PIROL was a value in its own right, for all students who wrote the diploma theses in this field. Development of PIROL served as a case study regarding managing software complexity. Finally, the resulting document, Part II of this thesis, is a comprehensive documentation of concerns and forces in building an SEE, which can be taken as a blue print for a future industrial-strength software engineering environment, which would exploit findings from SEE research far better than any of today's development environments.

Chapter 18

Acknowledgements

Much as all this thesis is concerned about structuring tangled concerns, also tangled relationships to institutes and people are now *untangled*, which is done within the *view* of contributions to this thesis. This view has abstraction properties, as it omits almost all details. Contributions are furthermore classified into three categories.

Institutional support comprised the German Brazilian co-operation, which funded a one-month stay in Rio de Janeiro planting the seed of Lua's role for PIROL. Also the ESPRESS project willingly/un-willingly supported the development of PIROL of which it used an early prototype for its show-case development environment. Other funding did not exist.

On the far end, *personal* relationships could be mentioned, which had impact on my development. Here and now I will leave such impact implicit in this writing. I do hope, the persons in question know better than I could express in this place.

My honest gratitude concerns all those *colleagues, supervisors and especially students*, who throughout the years — in changing constellations — supported the PIROL project in very different ways.

Michaela Reisin stimulated the birth of PIROL by her research on reference glossaries. Wilfried Koch established a niche for the early PIROL project to flourish. Doris Fähndrich contributed a tool and maintained PIROL Web pages over some years. Stefan Jähnichen showed great patience with this long running project far away from all hypes and fast successes. Prof. Kelter and his staff have been helpful with the usage of H-PCTE and fixed some bugs that occurred in no other application but PIROL. Mira Mezini saw a special value in PIROL as a realistic case study of new technology for separation of concerns. She helped me writing about Dynamic View Connectors and gave inspirations for exciting research beyond PIROL. All the students listed in App. B have contributed considerable shares by what for each of them was their one and only diploma thesis in computer science.

Most of all, Boris Groth has never ceased to believe in PIROL. As a co-founder of this project he contributed important ideas and inspired many students for the work in PIROL. While he was present and during his time travelling around, his enthusiasm for PIROL has always given strength to the little yellow bird ...

Part IV

Appendices

Appendix A

Definition of Lua/P

A.1 Syntax of Lua/P

(1)	class	::=	class_structure method* guard*
(2)	class_structure	::=	**Class{**classname;
			inherit,
			[creation,]
			[upgrade,]
			[attributes,]
			[class_attributes,]
			[connector_part]
			}
(3)	inherit	::=	**inherit =** (classname | classnames)
(4)	creation	::=	**creation =** (methodname | methodnames)
(5)	upgrade	::=	**upgrade =** (methodname | methodnames)
(6)	classnames	::=	**{**classname (**,** classname)***}**
(7)	methodnames	::=	**{**methodname (**,** methodname)***}**
(8)	attributes	::=	**attributes = {** attr_decls }
(9)	attr_decls	::=	attr_decl (**,** attr_decl)*
(10)	attr_decl	::=	attrname **:** attrtype
(11)	attrtype	::=	simpletype | listtype | termtype | **Binary**
(12)	simpletype	::=	basictype | classtype
(13)	basictype	::=	**String** | **Integer** | **Boolean**
(14)	classtype	::=	classname
(15)	listtype	::=	**List** listelemtype
(16)	listelemtype	::=	**(** simpletype **)** | tupeltype
(17)	tupletype	::=	**{** simple_attr_decl (**,** simple_attr_decl)* **}**
(18)	simple_attr_decl	::=	attrname **:** simpletype
(19)	termtype	::=	grammarname**.**typename
(20)	class_attributes	::=	**class_attributes = {** lua_attr_decl* }
(21)	lua_attr_decl	::=	attrname **:** (**table** | **number** | **string**)
(22)	guard	::=	simple_guard | list_guard
(23)	simple_guard	::=	**AttributeAccess** classname**.**attrname **{**
			[assign_func,]
			[get_func]
			}
(24)	get_func	::=	**get = method ()** functionbody **end**
(25)	assign_func	::=	**assign = method (** valueparam **)** methodbody **end**
(26)	list_guard	::=	**ListAccess** classname**.**attrname **{**
			[adding_func,]
			[removing_func,]
			[list_func,]
			}

(27)	adding_func	::=	**adding = method** (indexparam, valueparam)
			methodbody
			end
(28)	removing_func	::=	**removing = method** (indexparam, oldvalueparam)
			methodbody
			end
(29)	list_func	::=	*any method overriding one of those listed in App. A.2*
(30)	connector_part	::=	**root** = classname, view_classes,
(31)	view_classes	::=	**viewclasses = {** view_class (, view_class)* **}**
(32)	view_class	::=	classname = **{** roclass**,**
			[inherit ,]
			[creation ,]
			[cpredicate ,]
			[accept ,]
			[uses ,]
			[adds ,]
			[filter ,]
			[redirect]
			}
(33)	roclass	::=	**roclass** = classname
(34)	cpredicate	::=	**predicate = function (co, ro)** bool_func_body **end**
(35)	accept	::=	**accept = method()** methodbody **end**
(36)	uses	::=	**uses = {** featurelist **;** renamelist **}**
(37)	featurelist	::=	featurename (, featurename)*
(38)	renamelist	::=	attr_decl = featurename
(39)	adds	::=	**adds = {** attr_decls **}**
(40)	filter	::=	**filter = {** filter_decl (, filter_decl)* **}**
(41)	filter_decl	::=	attr_decl **{**
			base = { attr_decl**}**,
			fpredicate
			}
(42)	fpredicate	::=	**predicate (elem)** bool_meth_body **end**
(43)	redirect	::=	**redirect = {** attr_redir (, attr_redir)***}**
(44)	attr_redir	::=	attr_decl = **{** get_func [, assign_func] **}**
(45)	method	::=	**function** classname**:**methodname signature methodbody **end**
(46)	signature	::=	**(** attr_decl (, attr_decl)* **)** [**:** attrtype]
(47)	methodbody	::=	lua_stmt | luap_stmt
(48)	luap_stmt	::=	methodcall | classmethodcall | creationcall | upgradecall
(49)	methodcall	::=	objvar**:**methodname **(** expr (, expr)* **)**
(50)	classmethodcall	::=	classname**:**methodname **(** expr (, expr)* **)**
(51)	creationcall	::=	varname = classname**:**creator **(** expr (, expr)* **)**
(52)	creator	::=	**New** | methodname
(53)	upgradecall	::=	classname**:**methodname **(**objvar (, expr)***)**

A.2 Interface of builtin class List

Class List($\mathcal{T}$);

−−NOTE: Lists are assumed to be indexed contiguously starting at 1.
−−Instance fields are:
−−1.. : The elements
−−length : number of elements

- function List:**valid_index** (i)
  −−Is *i* a valid index into this list?
  −−**param** i: Integer
  −−**result** Boolean

- function List:**is_element** (el)
  −−Does this list contain an element equal to *el*?
  −−**param** el: $\mathcal{T}$
  −−**result** Boolean

- function List:**append** (elem)
  −−Append *elem* at end of list.
  −−**param** elem: $\mathcal{T}$

- function List:**concat** (other)
  −−Append all elements of *other* at end of list.
  −−**param** other: List($\mathcal{T}$)

- function List:**insert** (ind, elem)
  −−Insert *elem* at position, moving all elements at *i* >= *ind* up one.
  −−**param** ind: Integer
  −−**param** elem: $\mathcal{T}$

- function List:**first** ()
  −−Give index and value of the first element.
  −−**result** Integer, $\mathcal{T}$; Two values! First result is always 1.

- function List:**next** (i)
  −−Give index and value of the element following index *i.*
  −−**param** i: Integer
  −−**result** Integer, $\mathcal{T}$; Two values!

- function List:**last** ()
  −−Give the last element.
  −−**result** $\mathcal{T}$

- function List:**foreach** (func)
  −−Iterate *func(i,v)* over all elements or until *func* returns non-nil.
  −−Return first non-nil result of *func(i,v)*
  −−**param** func: function(Integer, $\mathcal{T}$) $\rightarrow \mathcal{X}$
  −−**result** $\mathcal{X}$

- function List:**map** (func)
  −−Map *func* over all elements and collect the non-nil result in a Lua table.
  −−Return the number of results and the result table.
  −−**param** func: function(Integer, $\mathcal{T}$) → $\mathcal{X}$
  −−**result** Integer, table($\mathcal{X}$)

- function List:**foldl** (start, func)
  −−Call *func* for each element of the List.
  −−Pass as second argument the result of the previous call.
  −−First call uses *start* instead.
  −−**param** start : $\mathcal{X}$
  −−**param** func : function($\mathcal{T}$, $\mathcal{X}$) → $\mathcal{X}$
  −−**result** $\mathcal{X}$

- function List:**search** (val, start)
  −−Search for an element equal to *val*.
  −−If *start* is non-nil it specifies the position, where to start searching.
  −−**param** val: $\mathcal{T}$
  −−**param** start: Integer or nil
  −−**result** Integer, $\mathcal{T}$

- function List:**find** (test_func, start)
  −−Find an element for which *test_func(elem)* evaluates to non-nil.
  −−If *start* is non-nil it specifies the position, where to start searching.
  −−**param** test_func: function($\mathcal{T}$) → Boolean
  −−**param** start: Integer or nil
  −−**result** Integer, $\mathcal{T}$

- function List:**remove** (ind)
  −−Remove the element at position *ind*. Move following elements up one.
  −−**param** ind: Integer

- function List:**remove_first** (test)
  −−Remove the first element satisfying function *test(elem)*.
  −−**param** test: function($\mathcal{T}$) → Boolean

- function List:**remove_all** (test)
  −−Remove all elements satisfying function *test(elem)*.
  −−**param** test: function($\mathcal{T}$) → Boolean

- function List:**range** (from, to)
  −−Create a new list of elements ranging from *from* to *to*.
  −−**param** from: Integer
  −−**param** to: Integer
  −−**result** List($\mathcal{T}$)

- function List:**wipe** ()
  −−Reset this list to an empty list.

- function List:**copy** (other)
  −−Make this list a copy of *other*
  −−**param** other: List($\mathcal{T}$)
- function List:**tostring** ()
  −−Convert list to String representation.
  −−**result** String

Documentation of builtin class List
Generated on Sun Nov 5 19:16:18 CET 2000
by *stephan@cs.tu-berlin.de*

Appendix B

Diploma theses related to PIROL

Several diploma theses have been cited throughout this work. Actually, the author started the PIROL project by his diploma thesis which was written in coordination with diploma theses by Boris Groth and Olaf Bigalk. Over the years, many students have contributed concepts and software to the PIROL project. This appendix lists all relevant diploma theses in chronological order and briefly sketches their contribution. Naturally, not all these theses have manifested themselves in the PIROL system as it exists today. Comments are only given were considered relevant for the state reached today. It turned out, that a tool can very well be developed by a student provided a suitable basis (as a framework) is available. Meta model extensions are also a realistic task. Extending the workbench has been tried once, which was maybe a bad idea in the beginning since dependencies are very high.

1994

Boris Groth

> Project Integrating Reference Object Library (PIROL):
> Concepts for Integrating an Object Oriented Generic Process Model
> into a Software Development Environment

Initiated process modeling in and for PIROL.

Stephan Herrmann

> Project Integrating Reference Object Library (PIROL): Development of a Workspace for Integration of Tools into a Software Development Environment for Consistent Object Oriented Modelling

Basic concepts, architecture and infrastructure of PIROL. This thesis also describes an implementation using Eiffel and Tcl.

1995

Olaf Bigalk

> Eine plattformunabhängige wiederverwendbare objektorientierte Klassenbibliothek für ein mehrschichtiges Kommunikationsprotokoll in der Software-Entwicklungsumgebung PIROL
>
> ——
>
> A platform independent re-usable object-oriented class library for a multi-layered communication protocol in the SEE PIROL.

Fundamentals of PIROL's middleware. At that time implemented using ToolTalk and by interfacing ToolTalk to Eiffel.

Stefan Brauer

> Project Integrating Reference Object Library (PIROL):
> Entwurf und Teilimplementierung eines Repositories für die Software-Produktionsumgebung PIROL
>
> ——
>
> Design an partial implementation of a repository for the SEE PIROL.

1996

Matthias Bienert ("student thesis")

> Konzeption und Prototyp-Realisierung eines graphisch orientierten Werkzeuges zur objektorientierten Analyse in PIROL
>
> ——
>
> Concepts and prototypical realization of a graphical tool for object-oriented analysis in PIROL.

The first graphical tool in PIROL based on [incr tcl], an object-oriented flavor of Tcl.

Jörg Buchwald

> Ein allgemeiner Generator zur Übersetzung von Klassen der objektorientierten Erweiterungssprache DROSSEL in objektorientierte Programmiersprachen.
>
> ——
>
> A general generator for translating classes of the object-oriented extension language DROSSEL into object-oriented programming languages.

A predecessor of the `proxygen` script (see Sect. 7.4.4). At that time no reflexive definition of the meta model was available in the repository.

Stefan Schuster

> Project Integrating Reference Object Library (PIROL):
> Eine objektorientierte Methode zur Visualisierung von Prozeßmodellen im Projekt PIROL auf Basis der Analyse existierender Methoden.
> ——
> An object-oriented method for visualizing process models in the PIROL project based on the analysis of existing methods.

Elaborated on process modeling and languages.

Ronald Melster

> Visualisierungstools zur Prozeßmodellierung auf Basis des PIROL-Metamodells
> ——
> Visualization tools for process modeling based on the PIROL meta model

The first framework based approach to graphical tools in PIROL. Based on the ET++ framework and applied the technique to process models.

Alexander Onnasch

> Project Integrating Reference Object Library (PIROL):
> Konzeption eines erweiterbaren Frameworks in PIROL und dessen Einsatz bei der Erstellung eines Designwerkzeuges
> ——
> Concepts for an extensible framework in PIROL and its application for building a design tool.

Carries on the previous work.

1997

Asuman Sünbüll

> Entwicklung eines Werkzeuges für die Verwendung des CORBA-Standards in einer objektorientierten Softwareentwicklungsumgebung.
> ——
> Development of a tool for applying the CORBA standard in an object-oriented SEE.

Developed a gateway between PIROL and CORBA.

Patrick Grüger

> Konzeption und prototypische Realisierung der Version- und Konfigurationsverwaltung in einer Software-Entwicklungsumgebung
>
> ——
>
> Concepts and prototypical realization of the version and configuration management in a SEE

An analysis of difficulties in versioning fine grained objects and their relations. Devised an architecture using filters for version selection. The first thesis to build on the new version of PIROL using Lua. Attempted to integrate functionality into the PIROL workbench while this already grow more complex than can be handled in a diploma thesis.

Michael Freitag

> Entwicklung einer genereischen Semantik für die Prozeßmodellierung und deren Realisierung in einer Prozeßmaschine.
>
> ——
>
> Development of a generic semantic for process modeling and realization by a process engine.

1998

Bertram Stahl

> Komponentenbasierte Entwicklung eines Browsers für das Repository einer Softwareentwicklungsumgebung
>
> ——
>
> Component based development of a browser for the repository of a SEE.

Initial development of PON. Much of the functionality existed in that version. Extensibility required quite some refactoring, though.

2000

Jan Peter

> Enwicklung eines Repository–fähigen mehrsprachigen Quelltexteditors für objektorientierte Programmiersprachen
>
> ——
>
> Development of a repository-capable multi language source code editor for object-oriented programming languages

Mostly contributed a thoroughly revised ROCM package `PRODUCT`. The prototypical application only handled high level language constructs.

Frank Bilgi

> Konzepte und Sprachen für die Software-Architektur:
> Vergleichende Anwendung auf die Software-Entwicklungsumgebung PIROL
>
> ——
>
> Concepts and languages for software architecture:
> comparative application to the SEE PIROL

This theoretical thesis used PIROL as a case study for architecture specification modeling. It helped to identify some problems in PIROL's protocols.

Ralf Kruber

> Werkzeugunterstützung für prozeßmodellbasierte Kommunikation in der Software-Entwicklungsumgebung PIROL
>
> ——
>
> Tool support for communication based on a process model in he SEE PIROL

Development of MESSED embedded into a larger concept of CSCW in PIROL.

2001

Burkhard Weber

> Graphische Editoren für die repository-basierte Softwareentwicklungsumgebung PIROL durch Erweiterung eines bestehenden Frameworks
>
> ——
>
> Graphical editors for the repository based SEE PIROL by extending an existing framework.

Comparison of three Java based frameworks for graphical editors and integration of the framework GEF into PIROL. Example application was an initial version of CollEd.

Frank Tscheuschner

> Einsatzmöglichkeiten eines virtuellen Dateisystems zur Werkzeugunterstützung für zyklische Software-Entwicklung
>
> ——
>
> Fields of application for a virtual file system for tool support for cyclic software development

Prototypical implementation of COFS.

2002

Christian Mattick

Editieren von Quelltexten in einer Softwareentwicklungsumgebung (PIROL) mit einheitlichem Repository

——

Source code editing in a SEE (PIROL) with a common repository.

Development of pjEdit.

Florian Hacker

Aspektorientiertes Entwerfen mit "Aspectual Collaborations"-Entwicklung eines grafischen Editors für die repository-basierte Entwicklungsumgebung PIROL

——

Aspect-oriented design with “Aspectual Collaborations” – Development of a graphical editor for the repository based SEE PIROL.

Appendix C

List of Figures

Appendix D

Index

Page numbers in **bold** face refer to a chapter dedicated to the given notion.
Page numbers in *italics* refer to a definition.
Roman numbers refer to a notable use of that notion

Appendix E

Bibliography

[AAA$^+$94] S. Abiteboul, M. Adiba, J. Arlow, P. Armenise, S. Bandinelli, L. Baresi, P. Breche, F. Buddrus, C. Collet, P. Corte, Th. Coupaye, C. Delobel, W. Emmerich, G. Ferran, F. Ferrandina, A. Fuggetta, C. Ghezzi, S. Lautemann, L. Lavazza, J. Madec, M. Phoenix, S. Sachweh, W. Schäfer, C. Santos, G. Tigg, and R. Zicari. The GOODSTEP project: General object-oriented database for software engineering processes. In *Proc. of the 1st Asian Pacific Software Engineering Conf*, pages 10–19. IEEE Computer Society Press, 1994. 44, 129

[AB91] S. Abiteboul and A. Bonner. Objects and views. In *Proc. ACM SIGMOD Conference on Management of Data*, pages 238–247, 1991. 233, 248

[ACF97] V. Ambriola, R. Conradi, and A. Fuggetta. Assessing process-centered software engineering environments. *Transactions of Software Engineering Methodology (TOSEM)*, 6(3):282–328, July 1997. 58

[ACP02] Proc. of Workshop on Aspects, Components, and Patterns for Infrastructure Software, held at [AOS02]. Technical Report 2002-02, University of British Columbia, 2002. 259, 299

[AG97] K. Arnold and J. Gosling. *The Java programming language.* Addison Wesley, 1997. 179

[ALN00] U. Aßmann, A. Ludwig, and R. Neumann. COMPOST project home page. `http://i44w3.info.uni-karlsruhe.de/~compost`, March 2000. 239

[AOS02] *Proc. of First International Conference on Aspect Oriented Software Development*, `http://trese.cs.utwente.nl/aosd2002/index.php`, 2002. ACM Press. 259, 295, 296, 297, 299, 300, 301, 302, 305

[AR92] E. Andersen and T. Reenskaug. System design by composing structures of interacting objects. In *Proc. of ECOOP'92*, number 615 in LNCS, pages 133–152. Springer Verlag, 1992. 253

[Asp] PARC Xerox. *AspectJ Language Specification.* available from `http://aspectj.org`. 265

[AYBdS96] S. Amer-Yahia, S. Brèche, and C. Souza dos Santos. Object views and updates. In *Proc. of Journées Bases de Données Avanceées (BDA'96)*, 1996. 248, 249

[Bal91] Robert Balzer. Tolerating inconsistency. In *Proc. of the 13th ICSE*, pages 158–165, Austin, Texas„ 1991. IEEE Computer Society Press. 232, 251

[Bar98] Daniel Bardou. Roles, Subjects and Aspects: How do they relate?, July 1998. Position paper at the Aspect Oriented Programming Workshop, ECOOP'98, Brussels, Belgium. Extended abstract published in ECOOP'98 Workshop Reader, Serge Demeyer and Jan Bosch, editors, Lecture Notes in Computer Science (LNCS), vol. 1543, Springer, 418–419, December 1998. 252

[Bar02] Joachim Barheine. Strategien zur transparenten Optimierung verteilter Komponentensysteme am Beispiel von Enterprise Java Beans. Diploma thesis, Technical University Berlin, 2002. 158

[BAWY95] L. Bergmans, M. Aksit, K. Wakita, and A. Yonezawa. An object-oriented model for extensible concurrent systems: The composition filters approach. Dept. of Computer Science, University of Twente, 1995. 106, 252

[BD96] D. Bardou and C. Dony. Split Objects: a Disciplined Use of Delegation within Objects. In *Proceedings of the 11th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)*, pages 122–137, San Jose, California, USA, October 1996. Published as ACM SIGPLAN Notices 31(10). 253

[BFW91] A. Brown, P. Feiler, and K. Wallnau. Understanding integration in a software development environment. Technical Report 31, CMU/SEI, 1991. 103, 107

[BGHHm98] R. Buessow, W. Grieskamp, W. Heicking, and S. Herrmann. An open environment for the integration of heterogeneous modelling techniques and tools. In *Proc. of the International Workshop on Current Trends in Applied Formal Methods*, number 1641 in LNCS. Springer, October 1998. 8, 49, 89, 151, 183, 184, 197

[Bil00] Frank Bilgi. Konzepte und Sprachen für die Software-Architektur: Vergleichende Anwendung auf die Software-Entwicklungsumgebung PIROL. Diploma thesis, Technical University Berlin, Fachbereich Informatik, Sekr. 5-6, 2000. 113

[BM02] E. Baniassad and G. Murphy. Managing crosscutting concerns during software evolution tasks: An inquisitive study. In AOSD'02 [AOS02], pages 120–126. 225

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999. 191, 232

[Bro92] Alan W. Brown. Control integration through message passing in a software development environment. Technical Report 35, CMU/SEI, 1992. 104, 135

[BZ89] R. Budde and H. Züllighoven. *Software-Werkzeuge in einer Programmierwerkstatt.* PhD thesis, Technical University Berlin, D83, 1989. 257

[C#01] *C# Language Specification.* Microsoft Press, 2001. 69

[CAB$^+$94] D. Coleman, P. Arnold, St. Bodoff, Ch. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object–Oriented Development: The Fusion Method.* Prenctice–Hall, 1994. 19

[Cag90] M. Cagan. HP SoftBench: An architecture for a new generation of software tools. *Hewlett-Packard Journal*, 41(2), 1990. 104

[CGL$^+$94] R. Cleaveland, J.N. Gada, P.M. Lewis, S.A. Smolka, O. Sokolsky, and S. Zhang. The concurrency factory — practical tools for specification, simulation, verification, and implementation of concurrent systems. In *Proceedings of the DIMACS Workshop on Specification of Parallel Algorithms*, Princeton, NJ, May 1994. 184

[CKM$^+$02] P. Costanza, G. Kniesel, K. Mehner, E. Pulvermüller, and A. Speck (eds.). Proc. of 2nd workshop on aspect-oriented software development. IAI-TR 2002-1, Rheinische Friedrich-Wilhelms-Universität Bonn, 2002. 299

[Cop99] James Coplien. personal communication, 1999. 10

[CoS00] University of Wollongong, Australia. *Proc. of CoSET workshop at the 22nd ICSE*, 2000. 299, 306

[CRI97] R. Cerqueira, N. Rodriguez, and R. Ierusalimschy. Binding an interpreted language to CORBA. In *II Simpósio Brasileiro de Linguagens de Programação*, pages 23–36, Campinas, September 1997. 30

[CW02] S. Clarke and R. Walker. Towards a standard design language for AOSD. In AOSD'02 [AOS02], pages 113–119. 239

[CY91] P. Coad and E. Yourdon. *Object-oriented Design.* Yourdon Press (Prentice Hall), New Jersey, 1991. 19, 255

[DBW00] W. Siberski D. Bäumer, D. Riehle and M. Wulf. *Pattern Languages of Program Design 4*, chapter Role Object, pages 15–32. Addison-Wesley, 2000. 252

[DK95] D. Däberitz and U. Kelter. Rapid prototyping of graphical editors in an open SEE. In SEE'95 [SEE95], pages 61–72. 131, 247

[DoD80] Requirements for Ada programming support environments — Stoneman. Technical report, Department of Defense, 1980. 33

[DW98] D. D'Souza and A. Wills. *Objects, Components, and Frameworks with UML – The Catalysis Approach.* Addison-Wesly, 1998. 239

[EAMP97] W. Emmerich, J. Arlow, J. Madec, and M. Phoenix. Tool construction for the British Airways SEE with the O_2 OODBMS. *Theory and Practice of Object Systems*, 3(3), 1997. 45, 77, 105, 129, 248

[Ecl] Eclipse project home page. `http://www.eclipse.org`. 112, 169

[ECM90] Portable Common Tool Environment (PCTE) ECMA-149. Abstract specification, European Computer Manufacturers Association (ECMA), June 1990. 33, 233, 241, 264

[ECM93] Reference Model for Frameworks of Software Engineering Environments – ECMA TR/55 3rd edition. Technical report, European Computer Manufacturers Association (ECMA), June 1993. 8, 101, 102, 104, 245

[EKS93] W. Emmerich, P. Kroha, and W. Schäfer. Object-oriented database management systems for the construction os CASE environmnets. In V. Marik, J. Lazansky, and R. R. Wagner, editors, *Proc. of the 4th Int. Conf. DEXA '93*, volume 720 of *LNCS*, pages 631–642. Springer Verlag, 1993. 44, 45, 208, 233

[Emm96] Wolfgang Emmerich. Tool specification with GTSL. In *Proc. 8th Int. Workshop on Software Specification and Design*, pages 26–35, Schloß Velen, Germany, 1996. IEEE Computer Society Press. 44, 45, 68, 78, 80, 251

[ESW93] W. Emmerich, W. Schäfer, and J. Welsh. Databases for software engineering environments - the goal has not yet been attained. In I. Sommerville and M. Paul, editors, *Proc. of the 4th European Software Engineering Conference*, volume 717 of *LNCS*, pages 145–162. Springer, 1993. 33, 34, 45, 129, 213

[Fit00] Jeremy Fitzhardinge. userfs project home page. `http://www.goop.org/∼jeremy/userfs`, 2000. 197

[Fow99a] Martin Fowler. Dealing with roles. Working draft: `http://www.martinfowler.com/apsupp/roles.pdf`, April 1999. 252

[Fow99b] Martin Fowler. *Refactoring: Improving the Design of existing Code.* Addison-Wesley, 1999. 174

[Fro89] B. Fromme. HP encapsulator: bridging the generation gap. *Hewlett–Packard Journal*, 41(3):59–68, 1989. 104

[Gar87] David Garlan. *Views for Tools in Integrated Environments.* PhD thesis, Carnegie Mellon University, May 1987. 137, 180, 241, 242, 245, 264

[GB80] I. Goldstein and D. Bobrow. Extending object oriented programming in smalltalk. In *Proceedings of the Lisp Conference*, Stanford, CA., 1980. 233

[GBCGM97] G. Guerrini, E. Bertino, B. Catania, and J. Garcia-Molina. A formal model of views for object–oriented database systems. *Theory and Practice of Object Systems*, 3(3):157–183, 1997. 250, 251

[GHJK95] B. Groth, S. Herrmann, S. Jähnichen, and W. Koch. Project Integrating Reference Object Library (PIROL): An object–oriented multiple–view SEE. In SEE'95 [SEE95], pages 184–193. 229, 262

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995. 38, 57, 87, 92, 151, 191, 235, 257

[GKN92] D. Garlan, G. Kaiser, and D. Notkin. Using tool abstraction to compose systems. *IEEE Computer*, pages 30–38, June 1992. 257, 258

[GR83] A. Goldberg and D. Robson. *Smalltalk 80: The Language and its Implementation.* Addison–Wesley, 1983. 209

[GR02] R. Green and A. Rashid. An aspect-oriented framework for schema evolution in object-oriented databases. In *Proc. of Workshop on Aspects, Components, and Patterns for Infrastructure Software, held at [AOS02]* [ACP02]. 176, 259

[Gro94] Boris Groth. Project Integrating Reference Object Library (PIROL): Concepts for integrating an object oriented generic process model into a software development environment. Diploma thesis, Technical University Berlin, Fachbereich Informatik, Sekr. 5-6, April 1994. 20, 58

[Grü97] Patrick Grüger. Konzeption und prototypische Realisierung der Version- und Konfigurationsverwaltung in einer Software-Entwicklungsumgebung. Diploma thesis, Technical University Berlin, Fachbereich Informatik, Sekr. 5-6, Aug. 1997. 129

[Gyb02] Kris Gybels. Using a logic language to express cross-cutting through dynamic joinpoints. In *[CKM+02]*, pages 49–54, 2002. 258

[Haa] Oliver Haase. Ntt, an algebraic query language for H-PCTE. `http://pi.informatik.uni-siegen.de/ntt`. 43

[Hac02] Florian Hacker. Aspektorientiertes Entwerfen mit ”Aspectual Collaborations”- Entwicklung eines grafischen Editors für die repository-basierte Entwicklungsumgebung PIROL. Diploma thesis, Technical University Berlin, Fachbereich Informatik, Sekr. 5-6, August 2002. 191

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987. 183

[Hau00] Michael Haupt. JADE: Entwurf und Implementierung eines Sprachkonstruktes zur dynamischen Komposition wiederverwendbarer Softwaremodule als Erweiterung der Programmiersprache Java. Diploma thesis, Universität-Gesamthochschule Siegen, `www.st.informatik.tu-darmstadt.de/projects/JADE/`, December 2000. 256

[Hen95] Andreas Henrich. P-OQL: and OQL-oriented query language for PCTE. In SEE’95 [SEE95], pages 48–60. 43

[Her93] Stephan Herrmann. Objektorientierter Entwurf und Implementierung eines Systems z ur Verzeichnisverwaltung anhand der Kriterien von STEPS. Student thesis, Technische Universität Berlin, Fachbereich Informatik, Sekr FR 5-6, February 1993. 257

[Her94] Stephan Herrmann. Project Integrating Reference Object Library (PIROL): Development of a workspace for integration of tools into a software development environment for consistent object oriented modelling. Diploma thesis, Technical University Berlin, Franklinstr. 28/29 D-10587 Berlin, Germany, june 1994. 23, 208

[Her00] Stephan Herrmann. Lua/P – a repository language for flexible software engineering environments. In CoSET 2000 [CoS00]. 23

[Her02a] Stephan Herrmann. Composable designs with UFA. In *Workshop on Aspect-Oriented Modeling with UML at [AOS02]*, 2002. 239

[Her02b] Stephan Herrmann. Object teams: Improving modularity for crosscutting collaborations. In *Proc. Net Object Days 2002*, www.netobjectdays.org, 2002. 157

[HJPP02] W.-M. Ho, J.-M. Jézéquel, F. Pennaneac'h, and N. Plouzeau. A toolkit for weaving aspect oriented UML designs. In AOSD'02 [AOS02], pages 99–105. 239

[HLN$^+$90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4), April 1990. 19, 183

[HM00] S. Herrmann and M. Mezini. PIROL: A case study for multidimensional separation of concerns in software engineering environments. In *Proc. of OOPSLA 2000*. ACM, 2000. 137

[HM01] S. Herrmann and M. Mezini. Combining composition styles in the evolvable language LAC. In *Proc. of ASoC workshop at the 23nd ICSE*, 2001. 265, 266

[HO93] W. Harrison and H. Ossher. Subject-oriented programming: a critique of pure objects. In *Proc. of OOPSLA'93*, pages 411–428. ACM, 1993. 180, 241, 254, 255

[HOT97] W. Harrison, H. Ossher, and P. Tarr. Using delegation for software and subject composition. Technical Report RC 20946 (92722) 5AUG97, IBM Research Division, 1997. 153

[HSS89] W. Harrison, J. Shilling, and P. Sweeney. Good news, bad news: Expersience building a software development environment using the object-oriented paradigm. In *Proc. OOPSLA '89*, pages 85–94. ACM, 1989. 216, 251

[HSSS96] F. Huber, B. Schätz, A. Schmidt, and K. Spies. AutoFocus - a tool for distributed systems specification. In *FTRTFT'96 – Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1135 of *LNCS*, pages 467–470. Springer Verlag, 1996. 184

[IdFC96] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. Lua—an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996. 23, 24, 28, 208

[Jac95] Daniel Jackson. Structuring Z specifications with views. *Transactions on Software Engineering and Methodology*, 4(4):365–389, 1995. 238

[KBPO$^+$95] B. Krieg-Brückner, J. Peleska, E.-R. Olderog, D. Balzer, and A. Baer. Universelle Entwicklungsumgebung für Formale Methoden (UniForM Workbench). Informatik Bericht 8/95, Universität Bremen, 1995. 170, 183

[Kee89] S. E. Keene. *Object-Oriented Programming in Common LISP: A Programmer's Guide to CLOS*. Addison–Wesley, Reading, 1989. 20, 78, 238

[Kel92] Udo Kelter. H–PCTE — a high–performance object management system for system development environments. In *Proc. COMPSAC '92*, pages 45–50, Chicago, Illinois, September 1992. PCTE product repository. 35, 245

[KFS95] M. Klose, V. Friesen, and M. Simons. Smile — a simulation environment for energy systems. In A. Sydow, editor, *Proc. of 5th International IMACS-Symposium on Systems Analysis and Simulation (SAS'95)*, pages 503–506. Gordon and Breach Publishers, 1995. 189

[KHH^{+}01] G. Kiczales, E. Hisdale, J. Hugunin, M. Kersten, and J. Palm. An overview of AspectJ. In *Proc. of 15th ECOOP*, number 2072 in LNCS, pages 327–353. Springer–Verlag, 2001. 78, 238, 258

[Kic94] Gregor Kiczales. Why are black boxes so hard to reuse? Transcript from Presentation at OOPSLA'94, 1994. 12, 237

[Kla00] Marcus Klar. *A Semantical Framework for the Integration of Object-oriented Modeling Languages*. PhD thesis, Technical University Berlin, July 2000. 19

[KLM^{+}97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.M. Loingtier, and J. Irwin. Aspect Oriented Programming. In *Proceedings of ECOOP '97*, number 1241 in LNCS, pages 220–243, 1997. 231, 252

[Kni99] Günter Kniesel. Type-safe delegation for run-time component adaptation. In *Proc. of ECOOP'99*, number 1628 in LNCS, pages 351–366, 1999. 253

[KP88] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *JOOP*, Aug./Sept. 1988. 105, 106, 257

[KR96] J. Kuno and E. Rundensteiner. The MultiView OODB view system: Design and implementation. *Theory and Practice of Object Systems*, 2(3):203–225, 1996. 248, 249

[Kru00] Ralf Kruber. Werkzeugunterstützung für prozeßmodellbasierte Kommunikation in der Software-Entwicklungsumgebung PIROL. Diploma thesis, Technical University Berlin, Fachbereich Informatik, Sekr. 5-6, December 2000. 121, 167, 192

[Lam94] Andreas Lampen. *Attributierte Softwareobjekte als Basis zur Datenmodellierung in Software-Entwicklungsumgebungen*. Dissertation, Technische Universität Berlin, D83, 1994. 42, 58

[Läm02] Ralf Lämmel. A semantical approach to method-call interception. In AOSD'02 [AOS02], pages 41–55. 258

[Lie86] Henry Liebermann. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proc. of OOPSLA 86*, pages 214–223. ACM Sigplan Notices, 1986. 252

[LLM99] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with aspectual components. In *Technical Report*, Northeastern University, April 1999. 265

[LN02] W. Löwe and M. Noga. Scenario-based connector optimization. In J. Bishop, editor, *Proc. 1st International Working Conference on Component Deployment (CD 2002)*, number 2370 in LNCS, pages 170–184. Springer–Verlag, 2002. 244

[Mat02] Christian Mattick. Editieren von Quelltexten in einer Softwareentwicklungsumgebung (PIROL) mit einheitlichem Repository. Diploma thesis, Technical University Berlin, Fachbereich Informatik, Sekr. 5-6, July 2002. 110, 194, 195, 199, 244

[Mem02] Juri Memmert. Employing AOSD technologies in large companies. Presentation in a session on "Early Industrial Experience With AOSD" at [AOS02], 2002. 240

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall International, New York, 1992. 38, 55, 63, 77, 208

[Mey97] Bertrand Meyer. *Object oriented software construction*. Prentice Hall International, New York, second edition, 1997. 9, 12, 19, 69, 260

[MH01] M. Mezini and M. Haupt. Neue Paradigmen des Softwareengineering: Integrationsorientierte Programmierung. *ObjektSpektrum*, Feb. 2001. 230

[ML98] M. Mezini and K. Lieberherr. Adaptive Plug-and-Play Components for evolutionary software development. In *Proc. OOPSLA'98*, volume 33 of *SIGPLAN Notices*, pages 97–116. ACM, 1998. 139, 140, 255, 264, 265

[MMD$^+$99] H. Mili, A. Mili, J. Dargham, O. Cherkaoui, and R. Godin. View programming: Towards a framework for decentralized development and execution of OO programs. In *Proc. of TOOLS'99*, 1999. 158

[MSL01] M. Mezini, L. Seiter, and K. Lieberherr. *Software Architecture and Component Technology: State of the Art in Research and Practice*, chapter Component Integration with Pluggable Composite Adapters. Kluwer Academic Publishers, 2001. 137, 139, 255

[Mye83] B. A. Myers. Incense: A system for displaying data structures. *Computer Graphics*, 17(3):115–125, July 1983. 106, 258

[NJB97] J. Neuhaus, W. Janzen, and A. Bäcker. A Case Study in Repository Selection for a distributed Software Engineering Environment. In *Proceedings of the 8th Conference on Software Engineering Environments (SEE'97)*, Cottbus, Germany, April 1997. 245

[Nor97] A. Nordwig. Entwicklung einer Notation und eines grafischen Editors für den objektorientierten Entwurf hybrider Systeme. Master's thesis, TU Berlin, 1997. 135, 178, 188

[Obj98] Object Design, Inc, Burlington, MA. *ObjectStore Advanced C++ API User Guide*, March 1998. 180

[OH90] H. Ossher and W. Harrison. Support for change in RPDE^3. In *Proc. 4th ACM SIGSOFT Symposium on Software Development Environments*, 1990. 251

[OKK$^+$96] H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal. Specifying subject-oriented composition. *Theory and Practice of Object Systems*, 2(3):179–202, 1996. 137

[OM01] K. Ostermann and M. Mezini. Object-oriented composition untangled. In *Proc. of OOPSLA 2001*, volume 36 of *Sigplan Notices*, pages 283–299. ACM, 2001. 79

[OMA99] CORBA Components – volume I, joint revised submission. OMG Document orbos/99-07-01, Object Management Group, August 1999. 236, 252

[OMG97] The Common Object Request Broker: Architecture and Specification, revision 2.1. TC Document formal/97.9.1, OMG, 1997. url: http://www.omg.org/cgi-bin/doclist.pl. 104

[OMG99] *UML Semantics*, chapter 2. OMG RTF committee, `http://www.rational.com/uml`, version 1.3 edition, June 1999. 19, 68

[Orl01] Doug Orleans. Separating behavioral concerns with predicate dispatch, or, if statement considered harmful. In *Workshop Advanced Separation of Concerns in Object-oriented Systems at OOPSLA'01*, 2001. 260

[Ost81] Leon J. Osterweil. Software environment research: Directions for the next five years. *IEEE Computer*, 14(4):35–43, 1981. 213

[Ous94] J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994. 23, 26, 208

[OW97] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, January 1997. 49, 89, 94, 214

[Par72] David L. Parnas. On the criteria to be used in decomposing systems in modules. *Communications of the ACM*, 15(12), 1972. 5, 231, 257

[Par74] David L. Parnas. On a 'buzzowrd': Hierarchical structure. In *IFIP Congress 74*, pages 336–339. North Holland Publishing Company, 1974. 230

[Par75] David L. Parnas. Use of the concept of transparency in the design of hierarchically structured systems. *CACM*, 18(7):401–408, 1975. 230

[Par78] David L. Parnas. Some software engineering principles. State of the art report on structured analysis and design, Infotech Internation, 1978. 231

[Pas89] W. Paseman. Tools on a new level. *Unix Review*, 7(6):68–77, June 1989. 104

[Pau96] L. C. Paulson. *ML for the working programmer.* Cambridge University Press, 2nd edition edition, 1996. 61

[Pau94] L. C. Paulson. *Isabelle – A Generic Theorem Prover*, volume 828 of *LNCS*. Springer-Verlag, 94. 183

[PC86] D. Parnas and P. Clements. A rational design process: How and why to fake it. *Transactions on Software Engineering*, SE-12(2):251–257, 1986. 15, 230

[PCT95] Amendment 1 to ISO/IEC 13719-1: Fine-grain object extensions. Technical report, International Organization for Standardization (ISO), 1995. 49

[Pet00] Jan Peter. Enwicklung eines Repository–fähigen mehrsprachigen Quelltexteditors für objektorientierte Programmiersprachen. Diploma thesis, Technical University Berlin, Fachbereich Informatik, Sekr. 5-6, 2000. 20, 194

[PK95] Z. Peng and Y. Kambayashi. Deputy mechanisms for object-oriented databases. In *Proc. IEEE International Conference on Data Engineering*, pages 333–340, 1995. 249

[Pla99] Dirk Platz. *Ein Werkzeugtransaktionskonzept für Objekt-Managementsysteme als Basis von Software-Entwicklungsumgebungen.* PhD thesis, Siegen University, 1999. 131

[Pre95] Wolfgang Pree. *Design patterns for object–oriented software development.* Addison–Wesley, New York, 1995. 211

[RBP$^+$91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, New Jersey, 1991. 232

[Ree96] Trygve Reenskaug. *Working with Objects – The OORAM Software Engineering Method.* Prentice Hall, 1996. 253

[Rei90] Steven P. Reiss. Connecting tools using message passing in the FIELD environment. *IEEE Software*, 7(4):57–66, July 1990. 85, 104, 257

[Rei92] Fanny-Michaela Reisin. *Kooperative Gestaltung in partizipativen Softwareprojekten*, volume 7 of *XLI Informatik.* Peter Lang, Frankfurt/Main; Berlin; Bern; New York; Paris; Wien, 1992. 19

[Rei95] Steven P. Reiss. Program editing in a software development environment. Technical report, Brown University, 1995. 194

[RIC] L. H. de Figueiredo R. Ierusalimschy and W. Celes. *Reference manual of the programming language Lua 4.0.* `http://www.tecgraf.puc-rio.br/lua/manual`. 30

[RS91] J. Richardson and P. Schwarz. Aspects: extending objects to support multiple, independent roles. In *Proceedings of the 1991 ACM SIGMOD international conference on management of data*, 1991. 138, 180, 252

[RV98] D. Rémy and J. Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Objects Systems*, 4(1):27–50, 1998. 20

[S$^+$00] David Sweet et al. *KDE 2.0 Development.* Sams Publishing, `http://www.andamooka.org/index.pl?section=kde20devel`, 2000. 170, 306

[SAD94] C. Santos, S. Abiteboul, and S. Delobel. Virtual schemas and bases. In *Proc. of the International Conference on Extending Database Technology*, volume 779 of *LNCS*, pages 81–93. Springer–Verlag, March 1994. 241, 247, 248, 264

[SB98] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *Proc. of ECOOP'98*, number 1445 in LNCS, pages 550–570. Springer Verlag, 1998. 254

[SBK86] M. Stefik, D. Bobrow, and K. Kahn. Integrating access-oriented programming into a multiparadigm environment. *IEEE Software*, 3(1):10–18, Jan. 1986. 69, 257, 258

[SEE95] IEEE Computer Society Press. *Proc. of SEE'95*, Noordwijkerhout, Holland, April 1995. Malcolm S. Verrall. 297, 298, 299, 305

[SHO95] S. Sutton, Jr., D. Heimbigner, and L. Osterweil. APPL/A: a language for software process programming. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 4(3):221–286, 1995. 68, 78, 251

[SHU02] D. Stein, S. Hanenberg, and R. Unland. A UML-based aspect-oriented design notation for AspectJ. In AOSD'02 [AOS02], pages 106–112. 239

[SLU89] L. A. Stein, H. Lieberman, and D. Ungar. A shared view of sharing: The Treaty of Orlando. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 31–48. ACM Press/Addison-Wesley, Reading (MA), USA, 1989. 252

[SM95] P. Steyaert and W. De Meuter. A marriage of class- and object-based inheritance without unwanted children. In *Proc. of ECOOP 95*, number 952 in LNCS, pages 127–144. Springer Verlag, 1995. 253

[SN92] K. Sullivan and D. Notkin. Reconciling environment integration and software evolution. *ACM Transactions on Software Engineering and Methodology*, 1(3):229–268, 1992. 78, 239, 257

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. Series in Computer Science. Prentice Hall International, 2nd edition, 1992. 183

[SR02] S. Sutton, Jr. and I. Rouvellou. Modeling of software concerns in Cosmos. In AOSD'02 [AOS02], pages 127–133. 225, 232, 240

[SS89] J. J. Shilling and P. F. Sweeney. Three steps to views: extending the object-oriented paradigm. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 353–361. ACM Press, 1989. 231, 241, 252

[SS95] S. Sachweh and W. Schäfer. Version management for tightly integrated software engineering environments. In SEE'95 [SEE95], pages 21–31. 129

[Sta98] Bertram Stahl. Komponentenbasierte Entwicklung eines Browsers für das Repository einer Softwareentwicklungsumgebung. Diploma thesis, Technical University Berlin, Fachbereich Informatik, Sekr. 5-6, Nov. 1998. 184

[Sta99] Victoria Stavridou. Integration in software intensive systems. *Journal of Systems and Software*, 48(2):91–104, October 1999. 103

[Ste87] Lynn A. Stein. Delegation is inheritance. In *Proc. OOPSLA'87*, pages 138–146, 1987. 252

[Su91] J. Su. Dynamic constraints and object migration. In Guy M. Lohman, Amílcar Sernadas, and Rafael Camps, editors, *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*, pages 233–242. Morgan Kaufmann, 1991. 181

[Suna] The Swing Connection. `http://java.sun.com/products/jfc/tsc`. 205

[Sunb] Sun Microsystems. Java-beans specification. Web Page `http://java.sun.com/beans`. 188

[Sun93] SunSoft. The tooltalk service — an inter–operability solution. Technical report, SunSoft Press and Prentice Hall, 1993. 85, 104

[Sut90] Stanley M. Sutton, Jr. *APPL/A: A Prototype Language for Software-Process Programming.* PhD thesis, University of Colorado, Boulder, Aug 1990. 43, 78

[Szy98] Clemens Szyperski. *Component Software, Beyond Object-Oriented Programming.* Addison-Wesley, 1998. 103, 135

[TBC$^+$88] R. Taylor, F. Belz, L. Clarke, L. Osterweil, R. Selby, J. Wileden, A. Wolf, and M. Young. Foundations for the Arcadia environment architecture. In *Proc. of the Software Engineering Symposium on Practical software development environments*, pages 1–13. ACM SIGSOFT/SIGPLAN, Nov. 1988. 43, 78, 106, 170, 213, 258

[TC93] P. Tarr and L. Clarke. PLEIADES: An object management system for software engineering environments. In *ACM SIGSOFT '93 Symposium on Foundations of Software Engineering*, pages 56–70, Los Angeles, Dec. 1993. 43, 78

[Tib00] Cristian Tibirna. *KDE 2.0 Development*, chapter DCOP: Desktop Communication Protocol. In [S$^+$00], 2000. 104, 170

[TN92] I. Thomas and B.A. Nejmeh. Defintions of tool integration for environments. *IEEE Software*, pages 29–35, March 1992. 103

[TO00] P. Tarr and H. Ossher. *Hyper/J User and Installation Manual.* IBM Corporation, 2000. 254

[TOHS99] P. Tarr, H. Ossher, W. Harrison, and S. Sutton, Jr. *N* degrees of separation: Multi-dimensional separation of concerns. In *Proc. of the 21st ICSE*, 1999. 6, 231, 237, 254

[US87] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Proc. of OOPSLA'87*, 1987. 20, 32, 252

[VB00] M. Van De Vanter and M. Boshernitsan. Displaying and editing source code in software engineering environments. In CoSET 2000 [CoS00]. 194

[VN96] M. VanHilst and D. Notkin. Using role components to implement collaboration-based designs. In *Proc. of OOPSLA'96*, 1996. 254

[Was89] Anthony Wasserman. Tool integration in software engineering environments. In F. Long, editor, *Software Engineering Environments*, number 467 in LNCS, pages 137–149. Springer Verlag, 1989. 33, 102, 103, 104, 170

[WdJ95] R.J. Wieringa and W. de Jonge. Object identifiers, keys, and surrogates. *Theory and Practice of Object Systems*, 1(2):101–114, 1995. 138, 241, 252

[WdJS95] R.J. Wieringa, W. de Jonge, and P.A. Spruit. Using dynamic classes and role classes to model object migration. *Theory and Practice of Object Systems*, 1(1):61–83, 1995. 181, 252

[Web96] Matthias Weber. Combining statecharts and Z for the desgin of safety-critical control systems. In Marie-Claude Gaudel and James Woodcock, editors, *Industrial Benefits and Advances in Formal Methods*, volume 1051 of *LNCS*, pages 307–326. Springer-Verlag, 1996. 183

[Web01] Burkhard Weber. Graphische Editoren für die repository-basierte Softwareentwicklungsumgebung PIROL durch Erweiterung eines bestehenden Frameworks. Diploma thesis, Technical University Berlin, Fachbereich Informatik, Sekr. 5-6, 2001. 191

[WF91] K. Wallnau and P. Feiler. Tool integration and environment architectures. Technical Report 11, CMU/SEI, 1991. 101, 102

[WHS01] K. Wallnau, S. Hissam, and R. Seacord. *Building Systems from Commercial Components*. SEI Series in Software Engineering. Addison-Wesley, 2001. 225

[WM00] R. Walker and G Murphy. Implicit context: Easing software evolution and reuse. In *Procs. of FSE 2000*, 2000. 258

[WWRT90] J. C. Wileden, A. L. Wolf, W. R. Rosenblatt, and P. L. Tarr. Specification level interoperability. In *Proc. of the 12th ICSE*, pages 74–85. ACM press, 1990. 101

[Zül98] Heinz Züllighoven. *Das objektorientierte Konstruktionshandbuch nach dem Werkzeug & Material-Ansatz*. dpunkt-Verla, Heidelberg, 1998. 257