PIROL: A Case Study for Multidimensional Separation of Concerns in Software Engineering Environments

Stephan Herrmann
Technical University Berlin
D-10587 Berlin, Germany
stephan@cs.tu-berlin.de

Mira Mezini
Darmstadt University of Technology
D-64283 Darmstadt, Germany
mezini@informatik.tu-darmstadt.de

ABSTRACT

In this paper, we present our experience with applying multidimensional separation of concerns to a software engineering environment. By comparing two different designs of our system, we show the importance of separating integration issues from the implementation of the individual concerns. We present a model in which integration issues are encapsulated into first–class connector objects and indicate how this facilitates the understandability, maintenance and evolution of the system. We identify issues of binding time, binding granularity and binding cardinality as important criteria in selecting an appropriate model for separation of concerns. We finally show how a good choice following these criteria and considering the requirements of software engineering environments leads to a system with dynamic configurability, high–level component integration and support for multiple instantiable views.

Keywords

Separation of concerns, software engineering environment, domain–specific language, component integration.

INTRODUCTION

The principle of separation of concerns [7 36] has made its way into all modern software formalisms, bringing about many different concepts for breaking a system into parts, with the object concept being the most recent one. In the last decade, a "post–objective" era has emerged represented by role and collaboration based designs [39 18 46 51], reflective architectures [25 21 26 27], and more recent approaches such as aspect oriented programming [22], programming with reusable collaborations [28], and the multidimensional separation of concerns model [49]. The main message of this "post–objective" era [15] is that (a) existing software formalisms support separation of concerns only along a "predominant dimension" (e.g., the data type axis in object–oriented languages as opposed to functions in procedural programming) while neglecting other dimensions, (b) focusing on only one dimension causes the definitions of concerns in other dimensions to be tangled with the definition of the concerns in the main dimension with negative effects on reusability, locality of changes, understandability, hence maintenance, and (c) the principle of separation of concerns becomes really effective when it is simultaneously supported along multiple dimensions, with concerns in different dimensions overlapping each other.

While the idea of multidimensional separation of concerns (MDSOC) is more than intuitive, and programming models for supporting it are emerging [22 28 49], an investigation of the design space of such models is still missing, partly due to the lack of experience reports on applying MDSOC to the construction of moderately sized software. In this paper, we make a modest effort to fill this gap by reporting our experience with applying the principle of MDSOC to a real software engineering environment (SEE). SEEs are integrated environments consisting of a collection of software engineering tools that work together, freeing the user from the need of manual coordination [48]. Hence, although our case study is an SEE, the results of the work presented here apply to any integrated environment, for which SEEs are just one example.

As any other integrated environment, SEEs are complex systems that support very different activities. Different team members act in different roles such as developer, administrator, project manager, etc. On the other side, development goes through different stages. Both dimensions refer to and manipulate the same product: a model of some problem domain — to be refined up to executable software — which is itself split into different stage–specific artifacts. Due to this variety of concerns and facets involved, the design of an SEE provides a good case study for applying multidimensional separation of concerns. Our case study is a medium sized SEE, called PIROL, developed at the Technical University of Berlin, Germany.

The contribution of the paper is twofold. *First*, it presents a model for MDSOC in integrated environments with a strong support for evolution. Not only does the model allow different tools to be developed independently of each other; more importantly, it preserves this independence beyond integration time. Obviously, software evolution is easiest when system components are independent. However, independence is a challenging goal in integrated environments, where integration requires fine–grained coordination of the involved components. The design presented in this paper solves this trade–off by separating the definition of tool integration relationships from the tools themselves.

Our design is based on the repository architectural style [45], where all tools are to be integrated with the shared repository. We distinguish between two levels of integration. At what we call the "physical" level, a basic "wiring" between decoupled tools is enabled by means of a uniform communication protocol. This wiring merely allows components to exchange messages, implying, however, that the participants exchange data of the same type. However, if tools are developed independently from their context of use, mismatches in their domain models are very likely. We call "logical" the level of integration responsible for handling these mismatches. We compare a design in which logical integration is mainly a matter of tools, i.e., each tool is responsible for the conversion of the data it exchanges with the repository, with a design in which logical integration is separated out of the tool definition. We show the superiority of the latter approach with respect to maintainability, reusability and evolution.

The *second contribution* of the paper consists of an evaluation of PIROL's design along a set of questions that can be used as a basis for comparing the chosen approach with other MDSOC models. The questions we consider are:

Q1: Specialized language support: *Is specialized (domain–specific) language support beyond the standard object–oriented model needed for enabling separate definition of concerns?*
For instance, the aspect language *D* [24] provides highly specialized constructs for the synchronization and distribution concerns. In most other approaches the actual implementation of concerns relies mostly on standard object–oriented notions.

Q2: Explicit integration: *Is concern integration defined separately or is it part of the concern definitions?*
The models of subject oriented programming (SOP [15]) and Pluggable Composite Adapters (PCA [29]), allow to define integration in separate units whereas, e.g., in AspectJ [35] concerns and their integration are defined in the same place.

Q3: Binding time: *Are concerns bound at compile, link or load time or can binding be delayed until run–time?*
The role objects in delegation based approaches (e.g., [42]) still exist at run–time and can by bound dynamically. On the contrary, in SOP different concern definitions are merged together at compile–time.

Q4: Binding granularity: *Which entities in concern definitions can be mapped to each other ?*
PCA's constructs of collaborations and composite adapters are examples for higher–level units of concerns and their integration. All relevant approaches allow mapping at the level of classes and their features.

Q5: Binding cardinality: *Can concerns be applied multiply? Can elements of a concern be mapped to multiple elements of another concern?*
Again PCAs are an example for flexibility, whereas, e.g., in SOP each element from one concern can only be bound to (at most) one element from each other concern.

As we see, emerging MDSOC models answer these questions differently. Also our experience shows that when applying MDSOC to complex systems such as SEEs, there is indeed no single answer to most of the questions above. Due to the many trade–offs that need to be solved in such a system, several co-existing alternatives should rather be supported. Our answer to *Q1* will be that standard object–oriented concepts enhanced by some semantically useful concept of modules actually suffice for the separate definition of tools and repository. However, domain–specific language features and constructs appear to be better suited for expressing lower–level concerns related to system properties, such as synchronization, persistency, etc., since these interact with the other concerns at a very fine–grained level. Similar trade–offs have to be solved with respect to *Q2*. While the explicit integration layer, supported by advanced language constructs, is the key to enabling true independence of tools, other concerns, for which this independence is not an issue, should be integrated more directly because the additional layer would unnecessarily increase the system complexity in terms of the number of units to be developed and might affect performance, as well. It is obvious that run–time binding (*Q3*) yields a greater flexibility, which has to be balanced, however, against the cost in code complexity and execution time. Possible answers to *Q4* also range from single attributes up to complex modules. We will demonstrate the benefit of allowing all levels of entities in a concern mapping, especially including a kind of module level that encapsulates a set of collaborating objects. In a similar vein, we will show examples where flexibility in binding cardinalities (*Q5*) is crucial.

To summarize, our experience with PIROL suggests that some form of "unification" of the MDSOC models is needed that supports alternative answers to the above questions (and others not considered here) regarding concern decomposition and composition. This requires a systematic investigation of the design space of MDSOC models, which would result in the definition of a "meta–model" for MDSOC software formalisms in order to facilitate the combination of several MDSOC techniques. This paper does not conduct such an investigation. With the evaluation of PIROL's design along the questions posed above, we only give first hints about the direction of this investigation and provide some evidence for its usefulness.

The remainder of the paper is organized as follows: In Sect. 1, we present the initial design of PIROL based on a MDSOC model without explicit support for integration of tools with mismatching data models, and indicate the shortcoming of this approach with respect to reusability, maintenance and traceability. In Sect. 2, we introduce an explicit integration dimension and use class graphs as the top–level binding unit. Sect. 3 evaluates this design along the questions posed above. Sect. 4 presents related work, while Sect. 5 concludes the paper and outlines areas of future work.

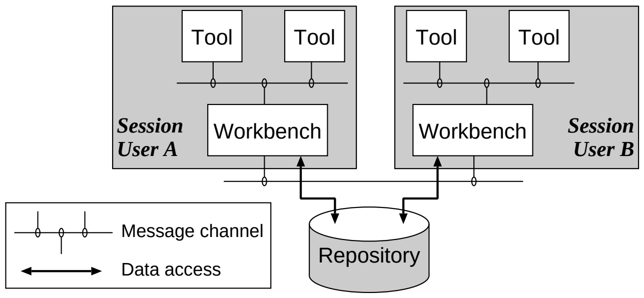


Figure 1: Overall architecture of PIROL

1. FIRST DESIGN OF THE SEE PIROL

From the very beginning, separation of concerns has been an important rationale in the design of PIROL [14 38]. This is due to the goal to build a generic SEE framework that allows *independent tools* to be closely integrated. *Independent* means that (a) there are no explicit inter–tool invocations, and (b) tools are developed for their own data model. Tools may be delivered as pre-compiled units, which run as separate processes, possibly across machine boundaries.

Yet, within the integrated environment tools (a) share a common universe of basic abstractions — the intrinsic model of a piece of software under development as well as the development process itself – and (b) interact with each other in performing some functionality, while (c) still preserving their independence. As pointed out in [48], preserving tool independence is crucial for facilitating evolution in an integrated environment, implying both (a) replacement of a tool with a new version of it and (b) a-posteriori integration of new tools. Our experience suggests that tool independence beyond integration time is crucial also for enabling *flexible binding cardinalities*, allowing e.g., shared data to be simultaneously displayed and manipulated by multiple tools and/or multiple co-existing documents belonging to the same tool, whereby the involved tools or documents may have specific, partially overlapping "perceptions" of the shared data.

In this section, we show how a set of techniques for separation of concerns, most of which are well explored, was combined into our first design of a framework for integrated environments, that partially met our goals, but still suffered from some shortcomings. An analysis of these shortcomings motivates the introduction of another level of separation of concerns into our design to be presented in Sect. 2.

1.1 Achieving "physical" independence

A first requirement was to provide a basic generic infrastructure that enables independent tools to "talk" to each other even if they do not explicitly know about each other. For this purpose, PIROL was designed with a three–tier architecture (Fig. 1). The *repository* provides a central data storage including basic services such as schema management, access control, and transactions. The repository schemas define the static part of PIROL's meta model. The *workbench* defines a user's session. It contains an interpreter for the object–oriented repository language Lua/P [16], that fully encapsulates the repository in that data from the repository are lifted to dynamic Lua/P objects, whose methods are executed within the workbench. Hence, Lua/P classes define PIROL's full meta model, referred to as the *repository model* throughout this paper and representing the universe of abstractions shared by all tools. For client processes the workbench very much resembles an OODBMS, with some additional features that are not discussed here. Finally, *Tools* provide the only means for user interaction.

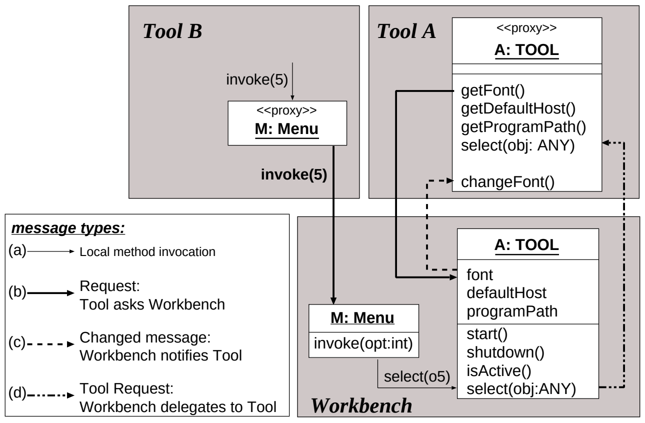


Figure 2: Messages between tools and workbench

As shown in Fig. 1, the communication between tools and the workbench (as well as workbench–to–workbench messages) happens via an explicit multicast messaging facility, that is based on the MSG package of the FIELD environment [40]. In order to communicate, tools and the workbench connect as clients to the messaging facility and register as observers of certain events. Due to this decoupled communication, tools may be implemented independently and in any language (that can interface C — the language in which MSG is written). They access objects in the repository by means of proxy classes, that (via MSG) transparently delegate all accesses to the workbench by means of `set` and `get` methods as well as method wrappers (cf. message type (b) in Fig. 2)[1].

The repository architectural style [45] also requires a mechanism for *automatic change propagation*. The workbench issues `changed` messages for each value that is changed within the repository ((c) in Fig. 2). Tools register with the messaging service as observers of such messages installing the necessary callbacks for updating the tool's display and internal data whenever a relevant `changed` message is issued to the message channel. The mechanism of change propagation is thus decomposed into three concerns: (1) the workbench is responsible for *generating* `changed` messages, (2) the message facility *dispatches* these messages to the registered observers and (3) tools only need to implement their specific *reaction* (redisplay etc.).

In addition to this data driven communication, PIROL also allows one tool to (indirectly via the workbench) invoke methods of another tool. For this purpose, the workbench represents each tool by an object of the Lua/P class `TOOL`.

[1]Most tools are currently implemented in Java, for which a library of proxy classes exists.

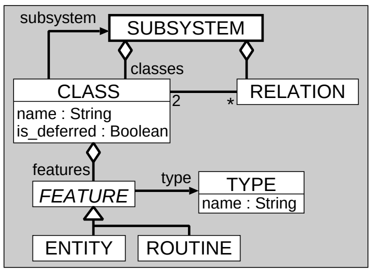


Figure 3: Extract from the repository's meta model

This class has methods that delegate execution (via the messaging facility) to the running tool process ((d) in Fig. 2). This feature is used e.g., to implement the notion of a context menu that enables the user of a tool to start or otherwise control other tools. When a tool (say, Tool B in Fig. 2) requests the context menu for a given object, the workbench configures a menu definition in the shape of a repository object based on the knowledge about currently installed tools and their features. The tool displays the menu and lets the user select an option. Now, the method `invoke(optionIndex)` (called on the menu proxy and passed to the menu repository object) may, e.g., result in a call of the method `select` to the `TOOL` object that represents `A` in the workbench. The latter call is finally forwarded to the process running tool `A`.

To summarize, the message channel provides a generic "wiring" which enables a *physical decoupling* between tools and the workbench. It is implicit with the message channel that tools are not only separate processes but they may be distributed all over the internet. Deploying a tool with special resource requirements to a dedicated host is simply configured by means of the `TOOL` object.

1.2 Striving for "logical" independence

The infrastructure presented so far enables "physical" "wiring" of tools and repository. As also pointed out in [4], this basic 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.

Meta model mismatches

The repository model includes only elements that contribute to the structure and semantics of the system under development. Classes in the repository model are, e.g., `SUBSYSTEM`, `CLASS`, `FEATURE` etc. (cf. Fig. 3). Different tools have, in general, their own specific abstractions. For instance, the meta–model of a concrete graphical editor for (a variant of) UML class diagrams, called *ZooEd* [31], which we will use as an example throughout this paper, is presented in Fig. 4. This model defines the abstractions on which the editor operates and which it needs to store in the repository.

Note, that the two models in Fig. 3 and Fig. 4 are completely independent of each other. For instance, when comparing the class abstractions in both meta models, namely `CLASS` in Fig. 3, and `Class` in Fig. 4, it appears that the

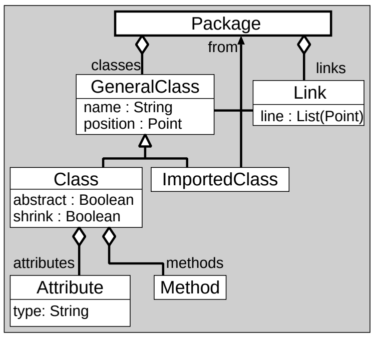


Figure 4: Meta model of a tool for UML class diagrams.

latter is mainly a graphical abstraction, defined in terms of its position and appearance (the latter is determined by the attributes `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. 3), 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. 3, the type of a `FEATURE` (and its subclasses) 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. 4). 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`.

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. Instead, we distinguish two kinds of persistent objects: the regular *repository objects* (RO) are defined by PIROL's basic meta model and carry all information that is to be shared between tools. In addition, documents that are manipulated by tools are modeled as a special type of persistent objects, called *conceptual objects* (CO). A CO defines a specific view onto a set of ROs: it encapsulates additional properties of ROs, relevant for the view at hand. For instance, a CO for a UML diagram encapsulates graphical information about the (`CLASS`) ROs included in the diagram such as their positions, fonts, expand/collapse flags, etc. Document specific properties are stored in a CO basically as a hash table; each entry in the table associates a value to a key–pair consisting of the name of a tool–specific property and the ID of the RO to be decorated with this property. COs are persistent objects with an open structure: a tool may store arbitrary properties for any RO without prior declaration.

For illustration consider the scenario presented in Fig. 5. In the upper part of the figure two documents are shown as they are displayed to the user of the graphical editor, represent-

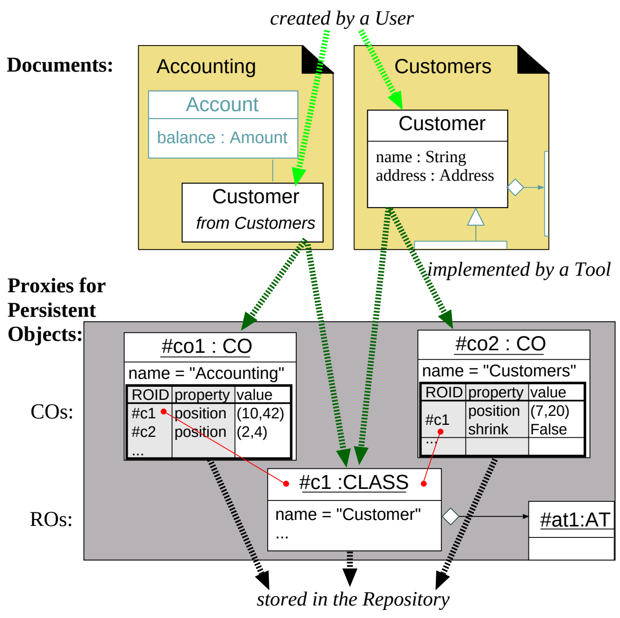


Figure 5: Documents are implemented as CO**s**

ing 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.

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. This is what we referred to as *flexible concern binding cardinality* in the previous section. If integration of different concerns was based on compile–time weaving of all partial definitions into one whole, as e.g., with IBM's MDSOC model [34] or Garlan's model of basic views [9], 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.

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* (see [16] for details), which allows *access–oriented* programming comparable to LOOPS' *active values* [47], comes into play here. Attribute guards introduce an additional layer between regular manipulation of ROs and their persistent representation, intercepting *all* access to an RO attribute and implementing appropriate policies of enforcing constraints over repository data. For instance, by attaching a guard to the `is_deferred` flag of a `ROUTINE`, the following constraint can be enforced: "each class that has at least one deferred routine is deferred, too". So when changing a `ROUTINE` to deferred, it is ensured, that also the `CLASS` will be set to deferred, if it was not already deferred before. The change is *propagated* from one RO to another. As attribute guards sit below every view definition, 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.

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. For illustrating some advantages and pitfalls of PIROL's design, as presented so far, we briefly report the experience with integrating *ZooEd* [31], whose source code was available to us.

We integrated *ZooEd* into our SEE framework by following three steps: (1) defining a structural mapping between *ZooEd*'s objects and their representations in the repository, (2) ensuring that user interactions trigger the appropriate repository updates using the defined structural mapping and (3) making sure that the defined structural mapping was also used to translate messages from the workbench to appropriate updates within the tool.

The first step was realized by defining *adapter* classes for all classes in *ZooEd*'s meta–model (cf. Fig. 4). The object structure in Fig. 6 shows how an adapter object mediates between a tool object and proxies to repository objects that store the data for the tool object[2].

[2]The figure shows the simplest case of mapping exactly one

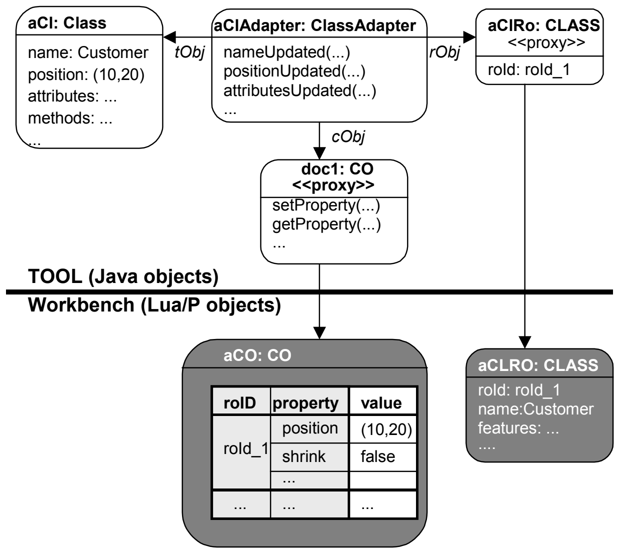


Figure 6: Structure of adapter objects.

For the second step, it was necessary to find all places in *ZooEd*'s code where relevant changes to the tool's internal data happened, so that actions on the appropriate adapters could be *triggered*. Fortunately, the design of *ZooEd* consequently used the command pattern [8] to model all "relevant" user interactions with the tool. For all command classes, the relevant adapters had to be identified (this required some administrative functions for adapter retrieval), and changes had to be classified such that each command could trigger appropriate updates of the repository via the adapter (cf. the methods `nameUpdated` etc. in Fig. 6). Note, that the command classes had to be modified in place: subclassing was not feasible in this case, because creation of command objects (i.e., references to command classes) is spread all over the tool.

For realizing the third step above, in addition to translating changes from the tool to repository changes, each adapter object is also responsible for *listening* to changes in the repository and reverse–translating them to changes in the state of tool objects. This could involve directly manipulating one or more tool objects (possibly repeating code very similar to that of some command class) or creating and configuring a command object that could perform the changes. In the latter case, great care had to be taken in order to avoid infinite change loops that would occur, if commands triggered by user interaction were indistinguishable from those invoked in response to a notification from the workbench.

Apart from these three steps, only the top–level class had to be modified, in order to connect to the workbench and install the initial adapters. No changes to the repository were required except for creating a new instance of type `TOOL`, that represents ZooEd and maintains its basic configuration. Being able to integrate a new tool without changing the repository model is what we gained from the separation of concerns within the meta model encoded by the RO–CO distinction along with tool encapsulation using the class `TOOL`.

tool object to one RO (plus the CO representing the document). Generally, an adapter mediates between more objects on either side and manually implements the knowledge of how data interrelate in both directions

1.3 Drawbacks of the first design

The ZooEd example reveals several drawbacks of encoding logical integration concerns into the tool's implementation[3].

First, the structural mapping between both models is hand–coded multiply. For instance, all three following adaptations implement essentially the same structural mapping (if we ignore the direction of mapping):

- Each command object needs to retrieve one or more appropriate adapter objects and call the appropriate update method(s) in order to write relevant changes to the repository.
- Each adapter object needs to react to `changed` messages from the workbench. For this task, it needs to know (a) which tool–internal objects should be modified and (b) how to trigger the appropriate redisplay within the editor. Especially difficult is the tracking of changes of CO properties: because all properties are stored in just one table it is not trivial to find the appropriate adapter object that is able to propagate the change to the right objects.
- To display a class diagram from the repository, each adapter implements a traversal over all components of its adapted RO. During this traversal, new tool objects, to be included within the drawing area, as well as proxies to the corresponding ROs and corresponding adapters are created.

Hence, structural mapping is spread over the tool implementation, tangled with code that implements other concerns such as change propagation (adapters) and basic tool functionality (commands, iteration of object structures etc.). Changing the implementation of one aspect will probably unnecessarily affect the others: they can not be reused independently of each other. For instance, after the in-place modification it is impossible to reuse the implementation of the tool's basic functionality with another repository model (or with a modified model of the same repository). On the other hand, different versions of a tool cannot reuse (parts of) the structural mapping aspect. One can envisage an upgrade of the UML diagram editor *ZooEd.v2* that supports presenting the symbols for inner classes à la Java [2] as graphically contained within the symbol of the enclosing class. Evidently, the integration of *ZooEd.v2* could reuse much of the structural mappings of *ZooEd.v1P* (the old version with adaptation for PIROL), 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 cannot be reused. This is schematically represented in Fig. 7.

[3]Note, that the drawbacks discussed here apply to both the case when structural mapping is included in the tool implementations from the very beginning, and the case, when it is inserted later via in-place code modifications.

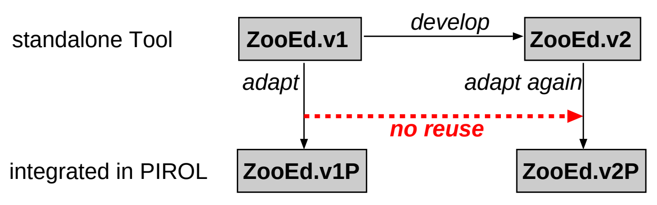


Figure 7: Adapting consecutive versions of ZooEd

A second design of the current design is that the properties stored in COs are handled as improper attributes: explicit queries to the CO are needed instead of direct attribute access at the RO. Also, no type information for CO–properties exists. As a result, no type checking is possible on these properties, and their correct use depends solely on the programmer's discipline.

Last, but not least, the design presented so far assumes that the source code of tools is available. This is not always the case, since tool providers want to protect their intellectual property. This is where many projects of tool integration fail and it may in fact be the most important reason for striving for a separation of tool implementation and its structural mapping to the repository.

1.4 Summary so far

We distinguished between two facets of tool independence to be considered by tool integration: *physical* versus *logical*. Two tools are considered as "physically" independent, if they do not explicitly invoke or otherwise reference each other. PIROL supports the integration of "physically" independent tools by means of: (a) its three–tier architecture, where tool interaction with the repository and other tools is explicitly modeled by the messaging facility and the workbench, and (b) its *implicit invocation* [48] mechanism based on the messaging facility[4]. As argued in [48], a design based on implicit invocation via an intermediate unit satisfies the requirement for "physical" independence[5].

Two tools are *logically independent*, if their data models are independent. The design of any integration framework has to deal with this kind of independence, as well. The question we pose here is how to do so without compromising the ease of evolution in an integrated environment. We claim that explicit support is required for this kind of logical interaction between tools justifying this claim by the drawbacks of the design of PIROL as presented so far, where independence is achieved at the cost of a clear and maintainable structure. While at the repository level, intrinsic and tool–specific abstractions are clearly separated into different persistent objects, mapping between tool and repository models is not explicit, but rather in-lined into the tool's implementation: a design that impedes evolution.

[4]Note, that also attribute guards are invoked implicitly, i.e., without the caller's knowledge.

[5]There is no distinction between "physical" and "logical" independence in [48]: what they deal with is actually only the physical aspect of tool independence as considered in this paper.

2. DYNAMIC VIEW CONNECTORS

The shortcomings of the first design have motivated the introduction of an explicit construct into Lua/P where to define the structural mapping necessary for integrating a tool into the environment. 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. 2.1) before going into the details of how to define it (Sect. 2.2).

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, define *virtual* repository *classes* (VC) 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.

For illustrating the idea, Fig. 8 redraws Fig. 5 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. 5, as it is not localized in a single place within the tool). The additional layer in Fig. 8 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.

One can think of VOs as implementing *roles* [42 52] 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 means eventually to gain new properties (e.g.,

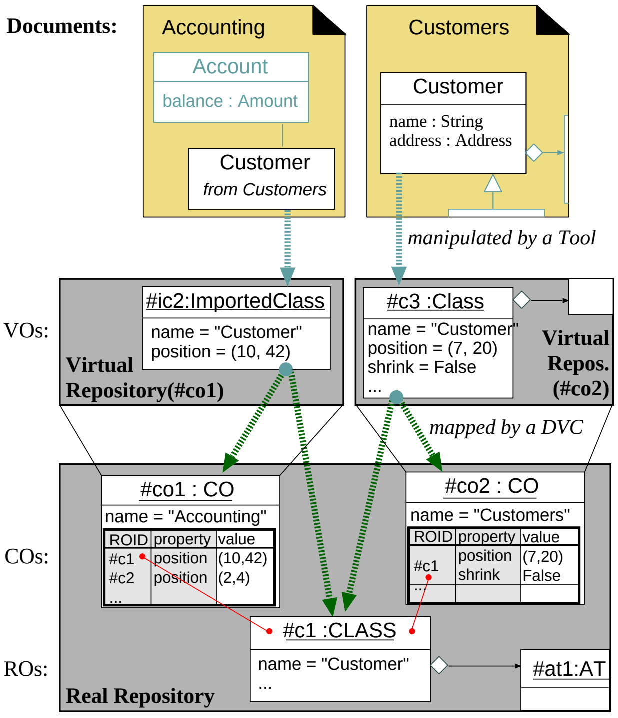


Figure 8: A virtual repository simulates tool objects

`position`). Note that the same base object may play different roles in different contexts. For instance, `#c1:CLASS` in Fig. 8 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* which *comprises a set of collaborating 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. 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*[28] and *Pluggable Composite Adapters*[29].

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. 9 shows part of a textual representation of the model in Fig. 4 — this is the expected interface to which the UML–diagram

```
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 9: Expected interface for UML class diagrams.

editor is implemented without any knowledge of the repository model.[6]

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 [28] 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*.

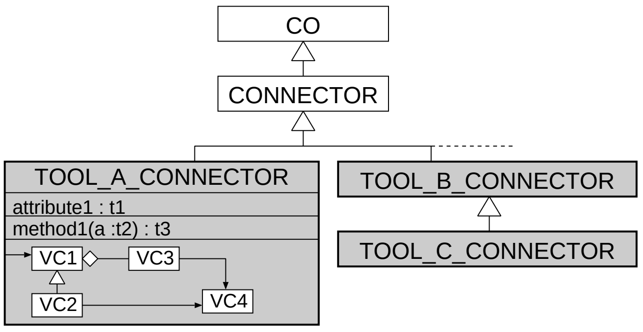


Figure 10: Hierarchy of connector classes

2.2 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 *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.

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

Connectors supersede the conceptual objects mentioned in Sect. 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 Fig. 11, which implements the `UML_CLASS_DIAGRAM` tool model of Fig. 9 on top of the repository model of Fig. 3. Within a **Connector** definition (10)[7] there are five constructs that define structure and behavior of the connector objects (beside those that define view classes nested in the connector): inherit, attributes, methods, creation and root. The keyword **inherit** serves to define a new connector class by inheriting from an existing one, the same way, e.g., extends does for classes in Java. For instance, `UML_CLASS_DIAGRAM_CONNECTOR` inherits from `GRAPH_CONNECTOR` (11) which provides abstractions such as `Node` and `Edge`: a UML class diagram is actually a graph. The keyword **attributes** serves to declare attributes of connector instances. Each instance of the connector class in Fig. 11, e.g., refers to the top–level `SYSTEM` RO (13) containing the package that is being shown in the UML diagram at hand. Similarly, the keyword **methods** introduces methods that can be applied to connector instances. The **creation** clause is used to declare the instantiation method for the connector (16), since Lua/P has no constructor naming convention à la Java. In our example, it is the `connect` method (18) that should be called in order to create instances of `UML_CLASS_DIAGRAM_CONNECTOR` (the usage and implementation of this method will be discussed in some more detail in Sect. 2.3). 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 type of the root VO is declared in the **root** clause of a connector. In Fig. 11, the root view class is `Package` (15).

Connector–level definitions are followed by nested view class definitions. First, for each view class `VC` used by the tool, the **roclass** clause declares the repository class `RC` to which `VC` is mapped. 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**. Three constructs (uses, filter and redirect) define how virtual objects are constructed out of the data stored in ROs. New attributes and/or methods of a view class that have no correspondence in the RO class are defined within the adds construct. Mappings as well as added attributes/methods are also inherited from parent view classes (possibly defined in a parent connector class). For instance, `GeneralClass` (30) inherits the attribute `position` from `Node` in `GRAPH_CONNECTOR`.

uses. This construct specifies those features of a view class that are *identified* with features of the corresponding RO class. This clause may optionally rename the repository feature and/or specify the view type to which the value of the feature ought to be converted to. For illustration consider the uses clause in the definition of `ImportedClass` (38). It specifies that (a) the feature `subsystem` from the underlying RO class `CLASS` (cf. Fig. 3) is visible in the `ImportedClass` view under the name `from`, and (b) when used in this context the value of this feature is to be automatically (by the

[7] Throughout this subsection, numbers in parentheses denote line numbers in Fig. 11.

```
1   Connector GRAPH_CONNECTOR {
2       root = Node
3       class Node {
4           adds = {position : Point}
5       },
6       class Edge {
7           adds = {line : List(Point)}},
8       },
9   }
10  Connector UML_CLASS_DIAGRAM_CONNECTOR {
11      inherit = GRAPH_CONNECTOR,
12      attributes = {
13          system : SYSTEM,
14      },
15      root = Package
16      creation = connect,
17      methods = {
18          connect (root_ro : SUBSYSTEM)
19              CONNECTOR:connect(root_ro)
20              system = root_ro:get_system()
21          end
22      },
23      class Package { roclass = SUBSYSTEM,
24          uses = {
25              classes : List(GeneralClass),
26              links : List(Link),
27              findClass (name: String) : GeneralClass,
28          },
29      },
30      class GeneralClass { roclass = CLASS,
31          inherit = Node,
32      },
33      class ImportedClass {
34          inherit = GeneralClass,
35          predicate ()
36              return not co.root_ro:eq(ro.subsystem)
37          end,
38          uses = { from : Package = subsystem }
39      },
40      class Class {
41          inherit = GeneralClass,
42          predicate ()
43              return co.root_ro:eq(ro.subsystem)
44          end,
45          creation = {place},
46          uses = { abstract = is_deferred },
47          adds = { shrink : Boolean },
48          filter = {
49              attributes : List(Attribute) = {
50                  base = {features : List(FEATURE)},
51                  predicate (f : FEATURE)
52                      return f:conforms("ENTITY")
53                  end
54              },
55              methods : List(Method) = {
56                  base = {features : List(FEATURE)},
57                  predicate (f : FEATURE)
58                      return f:conforms("ROUTINE")
59                  end
60              }
61          },
62          accept ()
63                  shrink = False
64                  ro.subsystem = co.root_ro
65          end
66      },
67      class Attribute { roclass = ENTITY,
68          redirect = { type : String = { /* see Fig. 12 */ }}
69      }
70      class Method { roclass = ROUTINE
71          adds = { body : String }
72      }
73      class Link { roclass = RELATION,
74          inherit = Edge,
75      }
76  }
```

Figure 11: Connector for UML class diagrams.

run–time machine) converted to a VO of type `Package` as defined within the same connector. Details of converting an RO to a VO (we also use the term lifting for this process) will be given later in this section. Note that the uses declaration for `findClass` in `Package` (27) 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`.

adds. A view class may *introduce new* attributes that do not have counterparts in the repository. E.g., attributes `shrink` (47) and `position` (from `Node` (4)) of `Class`, declared using the `adds` keyword, do not relate to any attributes in the repository model.

filter. This construct may be applied to 1:n repository level associations (list–attributes), in order to produce *selective views* of a them. The repository list–attribute to be filtered is specified by the keyword **base**. A **predicate** function determines the selection criterion. It is applied to each element of base to decide if it should be included in the corresponding tool–specific list–attribute. Elements of the base list that pass the test are converted to a view class when included into the resulting list. In Fig. 11, `Class` defines two selective views on the `features` attribute of `CLASS`. The filters `attributes` (49) and `methods` (55) split the base list attribute `features` according to the type of each element into a list of `ENTITIES` that are lifted to `Attributes` and a list of `ROUTINES` that are lifted to `Methods`.

A **predicate** may also be associated with a view class. For instance, both classes `Class` and `ImportedClass` define views on the same RO class `CLASS`. Whether instances of `CLASS` should be seen as `Class`, respectively `ImportedClass`, depends on the result of evaluating the respective class predicates (35,42) at run–time. So, when reading a `Package`'s list of `classes` (25) within the context of a connector, each RO contained in the base list attribute (`classes` of `SUBSYSTEM` in Fig. 3) 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[8].

redirect. Any other mapping that cannot be declaratively expressed with the constructs discussed so far can be hand–coded by defining two functions: **get** and **assign** for reading, respectively writing the value of an attribute. For illustration, recall the structural mismatch between the representation of `Attribute` types in the graphical editor — a `String` — as opposed to the representation of `FEATURE` types in the repository — an instance of the repository class `TYPE`. The `uses` construct is of no help here. Since `String` is a Java predefined rather than a view type whose definition is available within the connector, the Lua/P VM has no idea of how to convert `TYPE` to `String` and back. The programmer of the connector provides the VM with this knowledge by hand–coding the conversion process. When going from the repository model to the tool model, the name of the `TYPE`

[8] For the time being, we restrict class predicates to be based on constant attributes only, although object migration, i.e., dynamically changing the type of an object, might become a relevant topic here.

```
  ...
  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 12: Manual redirection of attribute `type`

RO will do perfectly, as shown in the implementation of get in Fig. 12. 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.

2.3 Repository, view, and connector objects

Fig. 13 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. 8 (`#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.

Let us now consider how the relationships between ROs, VOs and COs, are established and maintained. Fig. 14 illustrates how root ROs enter a connector when the latter is created. As defined in Fig. 11, `connect` expects the root RO (in this case of type `SUBSYSTEM`) to be passed as a parameter. In Fig. 14, we assume that `s1` is the RO for the `Accounting` subsystem (`#s1` in Fig. 13). The execution of the first line in Fig. 14, will

(a) create an instance of `UML_CLASS_DIAGRAM_CONNECTOR` (`#co1` in Fig. 13),

(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. 13) 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 19 in Fig. 11) and only (c) is explic-

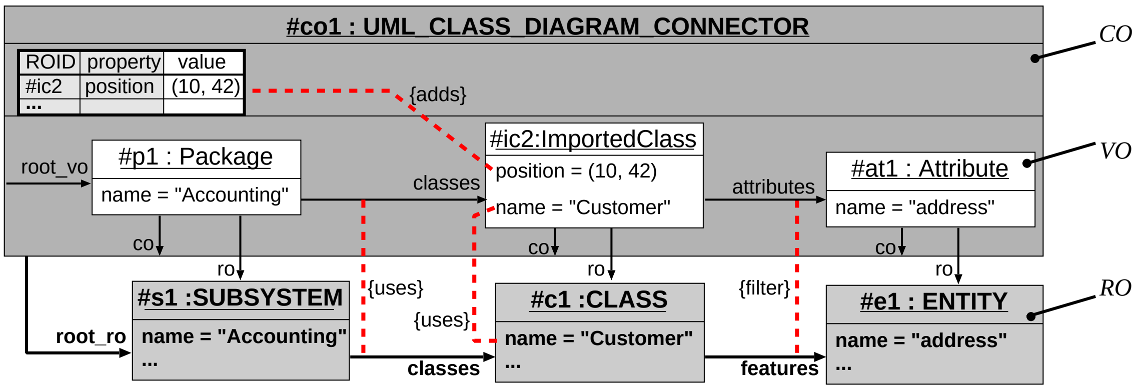


Figure 13: Run–time relations between ROs and VOs in a Connector.

itly 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 14: Connector instantiation

In Fig. 14, `#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. 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.

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 62–65 in Fig. 11 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.

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`.

2.4 Hand coded mappings versus specialized constructs

As indicated above, actually any attribute mapping can be expressed by the redirect construct. So, why do we then need the other constructs, uses, adds, filter besides the class level mapping constructs roclass and predicate? There are two main reasons for providing specialized constructs for recurring patterns of mapping.

First, the declarative style achieved through high–level constructs improves the readability and maintainability of the connector code. Without specialized constructs, connector classes will very likely contain a lot of stereotyped code duplication and unnecessarily include too much detail about the repository model. This results in connectors that are difficult to read, evolve and maintain.

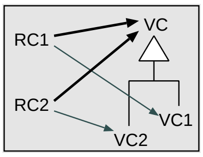
Second, using only redirect mappings could severely impact the performance of the system. To understand why, one has to reconsider change propagation in the presence of virtual repositories. Before the introduction of VOs, every data change in the repository unambiguously concerned a single RO. Hence, it was easy to deliver change notifications on the basis of ROs and their attributes. With VOs, it is, however, common that one atomic change in the repository affects several objects: one RO and all VOs that are views of that RO. It is crucial that VOs are also notified in order to preserve consistency between *all* documents in the environment. Change notifications concerning VOs need to be generated automatically just like for ROs to ensure that the difference between ROs and VOs remains transparent for the tools. They are implemented only against their expected interface from which proxy classes are generated, without knowing, which proxy objects stand for ROs and which for VOs. In fact, only this seamless access of ROs and VOs — including change propagation — enables us to use the term *virtual repository*.

The dependencies between ROs and VOs as established using built–in mapping constructs are known to the workbench (the Lua/P VM): for each changed RO, it knows exactly which attributes of which VOs need to be changed and can broadcast this change efficiently. This is not true for hand–coded mappings (using redirect). The dependencies established by this kind of mapping relationship are encoded in the programmers code. This makes it hard for the VM to determine when a change in the repository should result in a mapped attribute to change its value accordingly. The re–computation of redirected values is often triggered by "false alarm" because the interpreter lacks detailed knowledge about the data dependency. These excessive re–computations cause performance penalties.

Providing predefined specialized constructs always comes with the danger of arbitrariness. The question is how to define a set of standard mappings independently of a particular application, i.e., with enough expressive power to cover the needs of most applications. A combination of theoretical analysis and empirical evaluation should eventually yield a set of constructs that is satisfactory for all relevant problems. While not apt for proving completeness in a rigorous sense, a preliminary analysis [17] indicates that the DVC mapping constructs presented here do cover a wide range of relevant situations.

For illustration, let us discuss the coverage of relevant situations that occur when mapping RO *classes* to VO *classes*. In [17], similar considerations are given for the mapping of instances, attributes and lists. Three cases are trivial: $0 : n$ (new classes introduced in a connector), $1 : 1$ (direct mapping, set equality) and $n : 0$ (no mapping for an RO class). We also saw the $1 : n$ case in the example. Class `CLASS` from the repository model is *split* into classes `Class` and `ImportedClass`, which is discriminated by a class predicate (alternatively the declared type of a reference through which an RO is reached might also decide, which view class to use for lifting).

In order to complete this picture for class mappings, the only remaining situation is when several (unrelated) RO classes, e.g., `RC1` and `RC2`, are mapped to (merged into) one virtual class `VC` ($n : 1$ mapping). This case needs no special syntax, since it can be implemented by one indirection: First, the repository classes are mapped to corresponding view classes (`VC1, VC2`). Then a *supertype* `VC` for both view classes is *introduced*. View classes `VC1` and `VC2` implement the interface of `VC` in terms of their repository classes `RC1` and `RC2`.



The mapping constructs implemented so far do not cover, e.g., the case when a whole *path* in the repository class graph (a derived association between two repository classes) is mapped to a direct association between to view classes, or when an object is mapped to an atomic value. These are two examples where we would suggest the use of `redirect`. In case certain patterns of redirection like that occur over and over again, we might, however, consider adding a new keyword as a shorthand for such stereotyped mappings in the future. Addressing the mapping of paths with explicit constructs is under consideration for future extensions. This would bring forward the benefits of Adaptive Programming [23] also for Lua/P. It would make connector code more robust w.r.t small structural changes in the repository model.

2.5 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. Similarly, we will provide a generator for PIROL proxies (cf. Fig. 15). Deploying a tool is then a matter of furnishing also the set of generated proxy classes for the chosen middleware.

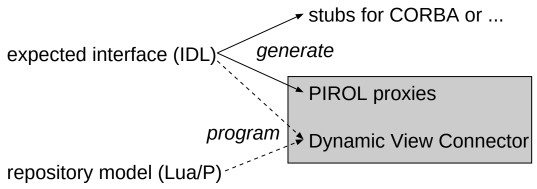


Figure 15: Tasks of integrating a tool

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 and the concrete repository model, the integration of tools now includes an additional step. A dynamic view connector has to be developed (cf. Fig. 15), 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.

3. EVALUATION

Our focus has been mainly on high–level dimensions of concerns involved in the "application logic" of an SEE: (1) the basic data abstractions of the SEE (the repository) (2) the features of an SEE (as represented by the tools), and (3) what we called the logical integration of concerns from the above two dimensions. Other dimensions, that do not directly contribute to the application logic but rather define *system properties*, such as transaction management, concurrency control or distribution aspects, orthogonally apply to all elements of the workbench interface. Most concerns in these dimensions are already covered by the functionality of the underlying middleware or repository and were not effectively elaborated in this paper. We did, however, mention two dimensions in the system properties category: (4) data synchronization (implemented by the change propagation mechanism) and (5) semantical integrity of repository data. Data synchronization includes concerns such as triggering, distributing and reacting to change notifications. In the semantical integrity dimension, we considered the concern of maintaining constraints over repository data by implementing constraint–preserving policies as attribute guards. Another concern in this dimension is avoidance of data redundancy, which is also an issue in PIROL, but falls beyond the scope of this paper.

Using the dimensions (1 – 5) mentioned above as our "test bench", in this section we discuss the design of PIROL along the questions concerning the design space of MDSOC models posed in the introductory section.

Q1: Specialized language support

It appears that the standard object–oriented model together with a long–term commitment to architecture do a good job in achieving modularity in the definitions of separated concerns in the data and feature dimensions. Provided the programmer of a tool agrees on (a) declaring the expected interface of the tool and (b) implementing the tool's functionality to this interface, the tool can be written as a stand–alone object–oriented application. Similarly, the repository meta model is written as a standard (certainly well–designed) object–oriented program in Lua/P, isolated from all other concerns. The designer of the meta–model only needs to focus on the basic abstractions involved in the process of software development.

On the other hand, special language support (combined with a commitment to a well–defined system architecture) comes quite handy for separating concerns related to system properties. Consider e.g., change propagation. Separating different concerns of change propagation from each other and from the application logic was achieved by (a) having change notification as a built–in feature of the workbench and (b) by conformance to the multicast–based architecture of PIROL. *Triggering* change notifications happens at the meta–level (within the Lua/P interpreter) orthogonally to application logic. The *dispatching* of notifications is the responsibility of the message channel, implemented by a separate server process that is loosely coupled to its clients — the workbench and the tools — via the mechanism of message–pattern registration. Tools define their reaction to change notifications by means of *callbacks* registered with the messaging channel. These are well–separated from other concerns, such as tool functionality or the definition of the repository data abstractions.

Maintaining the semantical integrity of shared data also makes use of a special construct of Lua/P, the attribute guards, which allows the programmer to plug additional behavior into the meta–level semantics of attribute access. Due to these built–in "hot spots", the integrity preservation concern can be defined well separated from the application–level code in the policies associated with attribute guards.

So, based on an architecture suitable for the given domain, we enriched the object–oriented model by (1) an infrastructure of built–in system services, that transparently apply to all requests to the workbench, (2) object–oriented techniques for filling in the hot–spots in such system services (e.g., tool callbacks within the mechanism of change propagation) and (3) a special language construct (attribute guards) to "decorate" the default semantics of attribute access with additional behavior, as needed.

Note, that the choice for (1) would change if variability concerning the system architecture were to be supported. An example of an alternative architecture would be a "poor–man's" implementation, where the repository is replaced by plain file storage, all tools were integrated to a single process and change propagation had to be re-implemented by simple method invocations instead of the server based message dispatching. If variability with regard to system architecture is needed, dispatching change notifications would have to be defined as an explicit concern rather than a transparent system service, so that it can be replaced by different implementations appropriate for different architectures. It would be pure speculation, to elaborate on the language constructs, that would be needed in order to define arbitrary implementations for dispatching notifications. However, as long as this variability is not needed, a transparent implementation of system properties can be given with less coding effort and providing a better performance than in the more general case.

Q2: Explicit integration

In answering this question, one has to trade off understandability, flexibility, reusability against performance. Our experience shows that the integration of high–level concerns that are involved with the application logic should be defined in explicit constructs, separately from the concern definitions. High–level concerns will very likely be subject of reuse and mixing their definition with integration issues hinders their reusability.

To support this claim, we presented two approaches to the definition of the "logical" integration between the core dimensions: features and basic data abstractions. In the first approach, *bridging data mismatches* during integration was mainly a matter of the adapter code spread around the tool's code. There was no distinct, separable integration layer, with the reported negative impact on maintainability. By extending Lua/P with DVCs, we separated the "logical" integration from tool definition which improved comprehensibility, reuse, and maintainability. The additional adaptation layer decouples the repository and its tools in multiple ways. First, the repository model may evolve without affecting existing tools, which are protected against these changes by the connector. Only the connector needs to be changed. Second, tools may be enhanced with no need to change the repository model, as long as other tools depend on the old model. Third, other tools may immediately benefit from the enhanced model by also using the new connector. Fourth, the same tool may be used to manipulate different types of objects: plugging a tool to a set of RO classes, defines its semantics in terms of the repository model. Different configurations of the same tool may be plugged in at different spots of the repository model providing different semantics. Other combinations of evolution and reuse can be thought of. All this comes with little effort: with its mostly declarative style, the definition of a DVC will for most applications be small and easy to read and maintain. The code for a connector is a succinct specification of the required adaptation. In fact, the example presented in this paper is only slightly modified from the result of a real case study. We are convinced that implementing a DVC is not much additional effort after understanding the required mapping in the first place.

On the other hand, reusability is probably less an issue for a considerable number of system properties. In analogy to the above scenario of a poor–man's version of PIROL, concern reuse by explicit integration would imply to exchange, e.g., the MSG–subsystem by a third party communication service without modifying any other concern. This would have to be achieved by an explicit gate–way between PIROL's components and the new communication software. Obviously, this kind of flexibility is even harder to achieve than pluggability of tools. I.e., for certain integral parts of the environment transparent integration may be more suitable.

When introducing an explicit integration layer, *scalability* issues have to be considered, as well. The introduction of DVCs in our architecture only slightly affects scalability. This is because of the three–tier architecture, where the workbench decouples the sessions of different users. Since DVCs live within a workbench, and not within the repository, only the following issues have to be investigated with respect to scalability: the number of DVC instances within one workbench, the number of ROs to be lifted to VOs within a single operation (say: loading a diagram) and the complexity of computations performed on a graph of VOs. Such performance measurements are still to be carried out. However, the number of users is no issue for DVC scalability.

Q3: Binding time

The high–level concerns are bound at run–time in the current design. A tool is bound to instances in the repository by instantiating a connector and applying it to a root RO. The resulting graph of virtual objects is lazily generated on demand. Of course, a connector definition must exist which matches the expected interface of a tool. But this definition can be added to the workbench at any time during a running project, which could also be seen as a run–time integration of a new type of virtual repository.

The late binding of VOs to ROs is not paid for at the source code level, because lifting of the sub–components of a root ROs happens automatically as they come into the scope of a connector. That is, the programmer never has to explicitly call the lift function. However, there will obviously be impacts on performance due to (a) run–time lifting and (b) the delegation from VOs to the appropriate ROs respectively COs. Although we cannot for the moment back our claims by empirical measurements, we expect *delegation* from a VO to the appropriate RO resp. CO to have little impact on the overall performance, because this delegation is hardly more then dereferencing one link per access; the appropriate access strategy for each declared feature is already determined at load–time of a connector definition. This overhead can be ignored when compared to the overhead due to inter–process communication between tools and workbench. Dynamic *lifting* may be more expensive especially in those cases where the choice of the appropriate virtual class to use for lifting depends on run–time state of the repository object to be lifted. Analyzing the connector structure and caching lifted objects are possible ways to reduce performance penalties.

However, there is no doubt that in answering the question about binding time one often has to trade off flexibility against performance. "Often" is to emphasize that in some cases, there is no option, i.e., binding must be dynamic. One of the arguments for runtime binding is that tool–specific data has to be represented as first–class objects in order to enable flexible binding cardinality, allowing attributes of an object or graph of objects to appear simultaneously in different views with different values of the same property. As already pointed out in Sect. 1, with compile–time weaving of all partial definitions into one whole, as in [34 9], it would be e.g., 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.

Another argument for dynamic binding is dynamic re-configurability, which implies that a tool can be replaced by another, or new tools can be integrated on–the–fly, i.e., without recompiling the repository. A third scenario, where dynamic binding is needed, is for attaching additional behavior to repository objects, whereby several variants of the additional behavior and the ability to dynamically exchange these variants should be supported. For instance, we might want to enhance certain ROs with the ability to write themselves out into a file, supporting several different storing formats and the ability to dynamically choose the format depending on the context of use.

If, however, flexible binding cardinalities or dynamic evolution are not an issue, static binding may be preferable for the sake of performance. There are certainly cases, where binding tools at *link–time* would be flexible enough, provided that integration can still be performed without access to the tool's source code. Link–time binding could be considered as an alternative to dynamic binding for all those cases where only one connector of each connector type would ever be created for a given graph of repository objects and also objects are mapped only on a 1:1 basis. This could be seen as a performance optimization of our model, but it may also bloat the underlying database. Object fields from all views of all relevant tools would have to be allocated for all objects of a given type, even if most of these objects may never actually appear within certain views of certain tools.

Q4: Binding granularity

Our experience with PIROL shows that it is important to support concern binding at different levels: features, classes, associations and even whole collaborations of classes. We emphasize support for binding at the level of collaborations of classes, since this has been ignored by most of the MDSOC models so far. Before introducing DVCs, PIROL's design did not support binding at the level of a set of collaborating classes. Our experience taught us that supporting this level is crucial in real–life complex systems. The idea is that within an integrated environment mapping different context specific definitions to some shared abstractions will very likely not happen at the level of isolated elements.

The adaptation process has, in general, to satisfy constraints such as "if the abstraction A has to be adapted to a view–specific definition A' then the abstraction B associated with

A has to be adapted to a view–specific definition B', with the A–B relation being mapped to some A'–B' relation". If there is no language support for gluing at the level of class graphs as provided by DVCs (and *connectors*, respectively *adapters* in [28 29]), these constraints have to be manually programmed mixed within the code for the A–to–A', respectively B–to–B' mappings. This was basically the approach in the first design of PIROL, where each adapter object within a tool implementation was responsible for triggering the adaptation of all relevant parts of its tool object, once the tool object was adapted itself (cf. Sect. 1.3). This approach is error–prone, since there is no guarantee that constraints are properly maintained. Furthermore, the gluing code becomes tangled, hard to read and fragile with respect to changes in the structural relationships.

With DVCs, a whole graph of mutually referring view classes is bound to a graph of mutually referring repository classes. Of course, this mapping is defined in terms of mappings between the lower–level constructs. A connector instance defines an encapsulation of an object graph ensuring that all objects reachable within a connector are lifted to the appropriate view classes defined in the connector. In a similar vein, the advantages of supporting extension at the level of *mutually recursive types*, i.e., types that refer to each other, such as A and B in previous paragraph, has been acknowledged in [3]. In [3] inheritance is used as the extension mechanism for defining mutually recursive extensions of the base types (corresponding to A' and B' in our example scenario). The adaptation of repository classes by view classes actually follows the same pattern, except for using aggregation rather than inheritance as the composition technique.

Q5: Binding cardinality

In our design binding cardinality is flexible in that no restrictions exist on the number and type of connectors instantiated for the same or overlapping set of base ROs. For the reverse case uniqueness is of course indispensable: each graph of VOs is uniquely bound to a graph of ROs. When focusing on the elements within a connector, the flexibility is given for having one RO playing different roles even within the same virtual repository. Flexible cardinalities were already supported by the first design of PIROL: conceptual objects (COs) defined views, that could be instantiated multiply for the same or overlapping sets of base objects. This was, however, lacking an elegant and type safe encapsulation.

Lessons learned

Within the given context of designing an integrated software engineering environment, we took a mixed approach to MDSOC by combining the advantages of (a) a well–designed architecture for a generic "wiring" mechanism between different concerns involved in the environment with (b) the strength of a hybrid domain–specific language, that

1. builds upon standard object oriented notions,
2. supports certain system properties crucial for the domain transparently as built–in features, (e.g., persistence, data synchronization), while providing some hot spots, where client code may plug into these system services (e.g., callbacks for handling change notifications),
3. provides specialized constructs for allowing the programmer to influence the semantics of the language, which come handy for defining certain concerns of the system property dimension (e.g., attribute guards as policies for semantical data integrity), and
4. introduces a high–level construct for the dimension of integrating the data and feature dimensions.

Thus, our approach indeed combines features from several MDSOC models. With PCA and SOP approaches it shares the explicit constructs for defining integration, as well as the use of the standard object-oriented model for the definition of high-level concerns. Additional constructs such as guarded attributes resemble the *advice* construct of AspectJ, while still other issues like change propagation are mainly solved by architectural design. In order for such a "supermarket" approach ("buy what you need") to work, it is desirable to have (1) a common foundation of MDSOC techniques (probably based on a language meta–model as we mentioned in our introduction), (2) a catalogue of questions to guide the process of selecting MDSOC techniques, (3) a language framework which allows to flexibly integrate the selected techniques into an operational programming infrastructure. At the experimental stage of applying MDSOC to PIROL, Lua/P (and its host language Lua [19]) proved to be well suited for this kind of evolutionary domain–specific language extension. This is due to its dynamic meta–level features that they share with other dynamic languages such a Smalltalk [11] or Self [50]. We can envision toolkits for modular language construction, that provide the power of a specialized combination of MDSOC techniques even to ordinary development projects.

Another insight form our case study that might be valuable for the MDSOC community concerns the issue of a distinguished "base" concern. Currently, there is no agreement in the MDSOC community, as whether there is a "base" concern that plays a distinguished role for the composition or not. The Xerox PARC approach to MDSOC [22] seems to favor the "base" concern idea, while other approaches [49 29] tend to consider all concerns as independent. As far as our experience with the design of PIROL tells, the answer seems to be in the middle. It appears that, at least in the context of integrated environments based on the repository architectural style, the data abstraction dimension indeed plays a distinguished role. All other concerns in one way or other relate to this dimension, *but* it is important that other concerns can still be developed independently. It is again the DVC layer that makes explicit, by which mapping other concerns relate to the base dimension. Change propagation, e.g., is coordinated at the level of RO attributes, but is automatically mapped to the concerns of tools by DVCs.

Our experience also shows the advantages of expressing the "base" concern in a domain–specific language, which transparently supports certain (for the domain important) system properties as built–in features complemented with explicit support for expressing other system properties. This level of specialized support is not needed for the implementation of tools as kind of "secondary" concerns.

Finally, the improvement in modularity as introduced by DVCs gives an encouraging example of how any component based system can be built with two seemingly contradictory goals reconciled: The flexible decoupling of components to be integrated a–posteriori and the tight integration of components that as an ensemble perform a complex set of functionalities.

4. RELATED WORK

Research about integrating software engineering tools always had a focus on supporting multiple views. As an early example, Garlan's [9] *basic views* can in fact be seen as an anticipation of subject–oriented programming: the actual model of a central database is an automatic merge of the required models of different tools. The mechanisms supported by basic views correspond to our uses and adds clauses. Objects are always manipulated within the context of a given view. This, in a way, resembles our way to identify a virtual object as the pair `(roid x coid)`, but Garlan's views are only identified by the view *type*, since basic views cannot be instantiated. With DVCs on the other hand, view instantiation is an essential technique. Furthermore, Garlan's *dynamic views* generalize over our filter construct, supporting a universal query language, but these views are *not updatable*, only the objects contained in such a view (objects of some basic type) are updatable. Dynamic view connectors on the other hand are carefully designed to ensure that all mappings are reversible yielding views that cannot be distinguished from collaborations of base objects.

The following two themes can also be found in many successor approaches concerning repositories for SEEs: (a) the global schema can be **combined** from different partial schema definitions and (b) the view on which a tool operates is defined as some kind of **filter** on shared data. *PCTE* [37], the repository used in PIROL, allows a "schema definition set" to extend existing object types (a). Tool views are defined by a "working schema", that simply enumerates schema definition sets, such that only values (objects, links and attributes) defined in these definitions are visible to the tool (b). The latter technique has been used, e.g., to implement generic tools [5]. These tools are configured by providing a working schema (filter) and exploit meta data (i.e., type information) for traversing and displaying the given view.

Sullivan and Notkin [48] very clearly state the problems with software evolution in an integrated environment. They show the shortcomings of three basic designs for integrated environments: (a) employing explicit invocations between independent tools, (b) employing implicit invocation via an event model, and (c) providing an explicit component that encapsulates the individual tools and realizes their integration relationships. Their approach combines the advantages of the second and third basic designs, in which (a) the integration relationship is modeled in an explicit unit, the *mediator*, and (b) individual tools are connected to an integration unit via implicit invocation. The design of PIROL presented here can be seen as an important improvement on the *mediators* approach to integrated environments. In the first design of PIROL presented in Sect. 1.2, it is the workbench that plays the role of a mediator and implements the integration relations between individual tools which communicate with the workbench via implicit invocation. In fact, even this first design of PIROL is an improvement on *mediators* because it (a) allows tools to share single copies of overlapping definitions (distinction between ROs and COs), which multiply exist in *mediators*, without, however, sacrificing tool independence as in [9], and (b) deals with datatype mismatches (through the adapters) which are completely ignored in [48]: their implicit invocation mechanism requires that events and methods associated with them conform on their signatures.

In the *GoodStep* project[12], O_2Views[43] was developed as an extension of the OODBMS O_2. Similar to our approach, objects in a view may "inherit" features from their base class by delegation. Views are again filters on shared data, which are in this case expressed as database queries. However, a view cannot add values, that are not present in the base (i.e., no schema combination). The extension (the set of contained objects) of a view has to be recalculated by an explicit command and updating through a view is not consistently solved. It follows that objects in a view behave noticeably different from base objects and the generation of a "virtual base" (i.e., a real base as seen through a view) from a base remains a heavy weight operation that prohibits fine grained integration. With its support for materializing a virtual base, O_2Views aims at flexible schema evolution, which has also been pursued in different projects for stepwise integration of new tools.

The OODBMS *ObjectStore*[33] allows to define "transformer functions" for performing data modification in the course of schema evolution. These directly correspond to the transformations implemented with our accept construct as a means for lazily adapting objects to new types.

All these approaches rely on the assumption, that at any given point in time, there exists *one global repository schema* as a union of all involved partial schemas. Each object has exactly one identity and the overall structure of all views is either identical (PCTE) or diverging structures are generated as second–class entities, that are not fully updatable (O_2). If tools are to be decoupled completely, as it is needed for a-posteriori integration, it is necessary to allow *mismatching schemas* to exist *simultaneously*, which in turn requires on–line bidirectional translation between different structures. This is a major novelty of dynamic view connectors.

Also, most repository data models do not allow view objects of the same base object to have different values for a given property (context dependent duplication of attributes). This is needed, e.g., for attaching different position values to a base object that appears in different diagrams of the same type. Surprisingly, already PCTE provides an elegant solution to this. A diagram can be modeled as an object whose *links* to contained symbols (base objects) can carry attributes that are specific to a base object within the context of a given diagram. Unfortunately, link attributes have no direct mapping to object oriented programming languages. Thus, a tool implementor is again faced with the problem of translating this information into the given programming language. In fact, link attributes serve as a low level implementation technique for added attributes of VOs in PIROL. Consequently combining the concept of role objects with repository views is a contribution of dynamic view connectors.

Finally, older approaches like PCTE work on pure data objects. Dynamic view connectors naturally integrate *methods* into the concepts of views, roles and collaborations. Not only can views access the methods of the repository model; this also allows to implement (certain services of) tools in Lua/P, granting for execution in the workbench. Functions implemented in Lua/P can be invoked by any tool in the environment, and thus directly contribute to the set of services which the workbench provides to all tools.

More recent work on SEEs pays less attention on view definitions, but aims at optimizing other aspects. Most importantly, performance and reuse of existing (monolithic) tools (commonly bringing their own data stores) has forced many authors to compromises regarding their initial concepts. E.g., Reiss [41] describes a system called DESERT with many ideas that are similar to PIROL. Yet, only two fairly weak concepts are included for integrating tools with structural *mismatches*: at the level of control integration a message mapper interprets simple rules of translating messages. Translation only allows renaming and rearranging of message arguments. Secondly, for the sake of data integration, "virtual files" may be defined as a collection of "fragments" (identifiable portions of files). In this concept, data integration is based on plain files containing source code, rather than on a repository with a fine–grained meta model. In source code centric environments, a tight semantical integration is much harder to achieve across development phases and notations. While approaches that build on existing complex tools provide a good amount of integrated functionality to the developer, our approach is more visionary, as it clearly defines the conditions, under which third party tools can be integrated much more efficiently in terms of (a) a greater variety of tools that can be integrated (not only those, that work on source code files), (b) a closer (semantical) integration across notations, and (c) greater ease of integration by means of a succinct connector specification.

Apart from research in the domain of SEEs, several general purpose concepts have more or less influenced the development of dynamic view connectors.

Views as a concept are also used in requirement analysis methods. For instance, in [32] *ViewPoints* are used to model different analysis artifacts, each specifying partial requirements for a system to be developed. *ViewPoints* also include so–called *inter–ViewPoint rules* expressing the relationships between separated requirement slices. These rules are checked for consistency during the integration phase and eventually a transformation is performed to handle inconsistencies. This transformation has the flavor of DVCs but the purposes are quite different: consistency checking of partial overlapping requirements versus integration of independent components.

The work on *collaboration–based design* [18 51 46 28] focuses on developing language support for defining collaborations between several objects separately of the static object model of the system. Our work is related to this research in that we allow tools — which can be seen as large–scale collaborations — to be written separately of the static structure of the repository objects that are involved in the collaboration.

A key theme of the work described in this paper is separation of concerns to avoid software tangling. This is also the motivation behind both *aspect oriented programming* [22] and *HyperSpaces* [49]. Our work developed special–purpose language technology for supporting separation of some concerns in SEEs, and applies it to the design of a real system. Another case study for separation of concerns can be found in [20], where AOP is applied to the design of a web–based learning system. Separating structural and behavioral aspects of an object–oriented system is the focus of the *adaptive programming* [23] approach. The static structure of a system is defined in a class dictionary — essentially a textual representation of the class graph. The behavioral aspects are written in terms of so–called traversal strategy specifications — essentially succinct descriptions of paths in the class graph — and do not embed detailed knowledge about the structure. Behavior written in this way is more robust against structural changes.

Several works in the area of object–oriented language design on *roles* [13], *composition filters* [1], *context objects* [44], *variation–oriented programming* [27], etc., also propose language support for separating the definition of different aspects. However, they focus on the separation of concerns at the level of a single, isolated object, and ignore important issues that need to be addressed when dealing with sets of collaborating objects.

Garlan et al. [10] identify categories of mismatches that occur when integrating components into a system. Their analysis of the "nature of connectors" with sub-issue "data model" comes closest to this work. Their focus is, however, much more technical: their example addresses the issue of inter language working, which is given for free in PIROL by decoupling components by means of a language independent communication layer. The components that were integrated in the Aesop system were generic components, so no concrete data models, that had to be reconciled, existed.

DeLine's *flexible packaging* [6] separates a component core functionality from its interaction with a certain architectural style or component interface standard (such as pipes and filters, ActiveX, or relational databases) within a certain context of use into two separate modules: the *ware*, respectively the *packaging description*. Flexible packaging helps to automate the build process of integrated components including all supplementary resources needed for this purpose. DeLine's wares and packaging description roughly correspond to tools, respectively repository meta model in PIROL with the meta model seen as one particular packaging description for the underlying PCTE. DeLine also recognizes that name, datatype, order, and aggregation mismatches between the packager and the ware should be dealt with and exploits explicit *maps* for this purpose. Yet, the mapping language is very primitive including, roughly speaking, only a restricted version of our redirect construct for primitive data types. Bidirectional conversions are not covered, nor are *aggregation mismatches*, i.e., mismatches concerning the structure of data. Automatic conversion of graphs of user defined data types are not supported at all. That is, as compared to the succinct and declarative nature of DVCs mapping in the *flexible packaging* approach is "manual", which might turn out to be a tedious and and hard to get right job in a real–life setting. Another difference concerns binding time. In the *flexible packaging* approach fix-up code from maps is inlined at compile–time. As the result, only a 1:1 binding cardinality is possible.

There is an analogy between DVCs and the use of deployment tools to map object (bean) attributes to data in a (relational or object) database in the EJB component model [30]. Due to the explicit deployment process, a bean's definition contains only the business logic, free of integration issues, and can for this reason be reused with different databases. This is similar to the decoupling of tool definitions from the repository model achieved with the introduction of DVCs, protecting tools against changes in the repository. DVCs are higher–level and allow for declarative specification of more complex mappings. Furthermore, by being first–class entities, DVCs can be reused with different versions of the same tool or even with different tools, eventually by being layered on top of each other.

5. SUMMARY AND FUTURE WORK

In this paper we presented our experience with applying multidimensional separation of concerns to a software engineering environment. The main focus of the paper was on requirements that this domain poses on the design space of models for MDSOC. By comparing two different designs of our system, we showed the importance of separating *integration* issues from the implementation of the individual concerns. We presented a model in which integration issues are encapsulated into *first–class connector objects* and indicated how this facilitates the understandability, maintenance and evolution of the system. Furthermore, we identified as crucial features for SEEs: (a) the postponement of the binding time of concerns until runtime — made possible by having connectors as first–class objects, (b) supporting the binding granularity of concerns at the level of *collaborating objects*, and (c) enabling flexible binding cardinalities with no restrictions on the number of connectors instantiated for the same or overlapping sets of objects. These features enable flexible configurability of SEEs including distribution across machine boundaries and a-posteriori integration of third party tools without source code modifications.

In the future, it would be interesting to explore more complex situations of mismatches in models, in order to gain a better understanding of what cannot be bridged by our connectors and extend the usefulness of dynamic view connectors towards even more complex situations. This may eventually lead to additional mapping constructs, of which the mapping of paths à la AP has already been mentioned. Furthermore, we plan to generalize over different techniques for "physical pluggability" (i.e., middleware like CORBA), in order to show how the concept of "logical pluggability" can also be applied to these techniques, in order to obtain a fully pluggable component infrastructure.

6. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their very helpful comments.

7. REFERENCES

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting Object Interactions using Composition-Filters. In *Object-Based Distributed Processing*, LNCS 791, pp. 152–184, 1993. 17

[2] K. Arnold and J. Gosling. *The Java Programming Language.* Addison Wesley, 1997. 6

[3] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *European Conference on Object-Oriented Programming (ECOOP)*, Brussels, July 1998. 15

[4] Clemens Czyperski. *Component Software, Beyond Object-Oriented Programming.* Addison-Wesley, 1998. 4

[5] D. Däberitz and U. Kelter. Rapid Prototyping of Graphical Editors in an open SEE. In *Proc. of 7th Conference on Software Engineering Environments (SEE'95)*, pp. 61–72. IEEE Computer Society Press, 1995. 16

[6] R. DeLine. Avoiding Packaging Mismatch with Flexible Packaging. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*, 1999. 17

[7] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976. 1

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995. 6

[9] D. Garlan. *Views for Tools in Integrated Environments.* PhD Thesis, Carnegie Mellon University, May 1987. 5, 14, 16

[10] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch, or, Why it's Hard to Build Systems out of Existing Parts. In *Proceedings of the 17th Internation Conference on Software Engineering (ICSE '95)*, pp. 179–185, April 1995. 17

[11] A. Goldberg and D. Robson. Smalltalk 80: The Language and its Implementation. Addison-Wesley, 1983. 15

[12] The GoodStep Team. The GOODSTEP Project: General Object-Oriented Database for Software Engineering Processes. In *Proc. of the 1st Asian Pacific Software Engineering Conf*, pp. 10–19. IEEE Computer Society Press, 1994. 16

[13] G. Gottlob, M. Schrefl, and B. Roeck. Extending Object-Oriented Systems with Roles. *ACM Transactions on Information Systems*, 14(3):268–296, 1996. 17

[14] B. Groth, S. Herrmann, S. Jähnichen, and W. Koch. Project Integrating Reference Object Library (PIROL): An Object–Oriented Multiple–View SEE. In *Proc. of 7th Conference on Software Engineering Environments (SEE '95)*, pp. 184–193, IEEE Computer Society Press, Apr. 1995. 3

[15] W. Harrison and H. Ossher. Subject-Oriented Programming: a Critique of Pure Objects. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '93)*, Sigplan Notices 28(10), pp. 411–428, 1993. 1, 2

[16] S. Herrmann. Lua/P – A Repository Language for Flexible Software Engineering Environments. In *Proc. of The Second International Symposium on Constructing Software Engineering Tools*, pp. 78–86, ISBN 0 86418 725 4, 2000. 3, 5

[17] S. Herrmann and M. Mezini.*Dynamic View Connectors*, `http://pirol.cs.tu-berlin.de/papers/DVC.pdf`, Technical Report, Technical University of Berlin, 2000. 12

[18] I. Holland. *The Design and Representation of Object-Oriented Components*. PhD Thesis, Northeastern University, Computer Science, 1993. 1, 17

[19] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes, "Lua—an extensible extension language," *Software: Practice and Experience*, 26(6):635–652, 1996. 15

[20] M. Kersten and G. Murphy. Atlas: A Case Study in Building a Web-Based Learning Environment Using Aspect-Oriented Programming. In *Proeedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, Sigplan Notices 34(10):340–352, 1999. 17

[21] G. Kiczales, J. des Rivières, and D. G. Bobrow. The Art of the Metaobject Protocol. The MIT Press, 1991. 1

[22] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.M. Loingtier, and J. Irwin. Aspect Oriented Programming. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP '97)*, LNCS 1241, pp. 220–243, 1997. 1, 15, 17

[23] K. Lieberherr. *Adaptive Programming: the Demeter Method*. PWS Publishing Company, 1996. 12, 17

[24] C. Lopes. *D: A Language Framework for Distributed Programming*. PhD Thesis, Northeastern University, Computer Science, Nov. 1997. 2

[25] P. Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '87)*, Sigplan Notices, 22(12):147–155, 1987. 1

[26] J. McAffer. Meta-Level Programming with Coda. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP '95)*, LNCS 952, pp. 190–241, Springer Verlag, 1995. 1

[27] M. Mezini. *Variational Object-Oriented Programming Beyond Classes and Inheritance*. Kluwer Academic Publisher, 1998. 1, 17

[28] M. Mezini and K. Lieberherr. Adaptive Plug-and-Play Components for evolutionary software development. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*, Sigplan Notices, 33(10):97–116, 1998. 1, 8, 15, 17

[29] M. Mezini, L. Seiter, and K. Lieberherr. Component Integration with Pluggable Composite Adapters. In M. Aksit (ed.) *Software Architecture and Component Technology: State of the Art in Research and Industry*, Kluwer Academic Publishers, 2000. 2, 8, 15

[30] R. Monson-Haefel. *Enterprise Java Beans*. O'Reilly & Associates, Inc, 1999. 18

[31] A. Nordwig. Entwicklung einer Notation und eines grafischen Editors für den objektorientierten Entwurf hybrider Systeme. Master's Thesis, TU Berlin, 1997. 4, 5

[32] B. Nuseibeh, J. Kramer, and A. Finkelstein. A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification. *IEEE Transactions on Software Engineering*, 20(10):760–773, Oct. 1994. 17

[33] Object Design, Inc, Burlington, MA. *ObjectStore Advanced C++ API User Guide*, March 1998. 16

[34] 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. 5, 14

[35] PARC Xerox, available from `http://aspectj.org`. *AspectJ Language Specification*, Aug 1999. 2

[36] D. L. Parnas. On the Criteria to be Used in Decomposing Systems in Modules. *Communications of the ACM*, 15(12), 1972. 1

[37] ISO/IEC 13719-1: Portable Common Tool Environment (PCTE). Abstract Specification, International Organization for Standardization (ISO), 1995. 16

[38] PIROL Web-page. `http://pirol.cs.tu-berlin.de`. 3

[39] T. Reenskaug, E. P. Andersen, A. J. Berre, A. Hurlen, A. Landmark, O. A. Lehne, E. Nordhagen, E. Ness-Ulseth, G. Oftedal, A. L. Skaar, and P. Stenslet. OORASS: Seamless Support for the Creation and Maintenance of Object Oriented Systems. *Journal of Object-Oriented Programming*, Oct. 1992. 1

[40] S. P. Reiss. Connecting Tools Using Message Passing in the FIELD Environment. *IEEE Software*, 7(4):57–66, July 1990. 3

[41] S. P. Reiss. The Desert Environment. *ACM Transactions on Software Engineering and Methodology*, 8(4):297–342, Oct. 1999. 17

[42] 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. 2, 7

[43] C. Santos, S. Delobel, and S. Abiteboul. Virtual Schemas and Bases. In *Proceedings of the International Conference on Extending Database Technology*, LNCS 779, 1994. 16

[44] L. Seiter, J. Palsberg, and K. Lieberherr. Evolution of Object Behavior using Context Relations. *IEEE Transactions on Software Engineering*, 24(1):79–92, Jan. 1998. 17

[45] M. Shaw and D. Garlan. *Software Architecture: Perspectives of an Emerging Discipline*. Prentice Hall, 1996. 2, 3

[46] Y. Smaragdakis and D. Batory. Implementing Layered Designs with Mixin Layers. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP '98)*, LNCS 1445, pp. 550–570, Springer Verlag, 1998. 1, 17

[47] M. Stefik, D. Bobrow, and K. Kahn. Integrating Access-Oriented Programming into a Multiparadigm Environment. *IEEE Software*, 3(1):10–18, Jan. 1986. 5

[48] K. Sullivan and D. Notkin. Reconciling Environment Integration and Software Evolution. *ACM Transactions on Software Engineering and Methodology*, 1(3):229–268, 1992. 1, 3, 7, 16

[49] P. Tarr, H. Ossher, W. Harrison, and S. Sutton Jr. *N* degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*, 1999. 1, 15, 17

[50] D. Ungar and R. Smith. Self: The power of simplicity. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '87)*, Sigplan Notices, 22(12):227–242, 1987. 15

[51] M. VanHilst and D. Notkin. Using Role Components to Implement Collaboration-Based Designs. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '96)*, Sigplan Notices, 31(10):359–369, 1996. 1, 17

[52] R.J. Wieringa and W. de Jonge. Object Identifiers, Keys, and Surrogates. *Theory and Practice of Object Systems*, 1(2):101–114, 1995. 7