

# Connectors for bridging Mismatches between the Components of a Software Engineering Environment

**Stephan Herrmann**  
Technische Universität Berlin,  
10587 Berlin, Germany  
stephan@cs.tu-berlin.de

**Mira Mezini**  
Universität Siegen,  
57068 Siegen, Germany  
mira@informatik.uni-siegen.de

## ABSTRACT

Software Engineering Environments (SEE) are complex systems, for which configurability is an important requirement. Constructing SEEs out of existing tools is evidently desirable. During such a composition mismatches in the data models of different tools need to be dealt with. The brute force technique by hacking data translators into the implementation of individual tools has severe drawbacks regarding modularity, maintainability and extensibility of the composed system. We propose a novel technique for designing SEEs that uses explicit language constructs for bridging the mismatches in the data models, called *dynamic view connectors* (*DVCs* for short). We show how the separation of tool functionality from the concerns of bridging data model mismatches imposed by *DVCs* improved the configurability and maintainability of an existing SEE.

### Keywords

Software Engineering Environments, Aspect Oriented Programming, Subject Oriented Programming, separation of concerns, Adaptive Plug&Play Components, roles, views, connectors, component-based development

## 1 Views in Software Engineering Environments

The design of software engineering environments (SEEs for short) involves very different concerns. Some concerns relate to the development of a *product*, e.g., constructing UML class diagrams, others to the *process* of the development, e.g., project management. Some tools act as transformers producing derived documents from source documents, others are interactive editors. During all activities version control, configuration management and access control should be used transparently for object access. Services can be implemented by the SEE framework itself or can be delegated to external tools. Tool integration may range from custom developed tools up-to third party tools, that should also be integrated as tightly as possible. Obviously, it is a challenging task to develop an SEE as a system of components (or modules) in which the concerns mentioned above are not terribly tangled which each other, since all of them actually operate on a common universe – some model of a software systems being developed.

A well-known model for integrating different tools and services of an SEE is the *repository* architectural style [15]. In this style, the representation of data play a predominant role. The repository is a central data storage and most communication among different tools in the SEE is driven by or relates in other ways to the central data. Tools in a repository-style SEE operate on different *views* of these data[2]. Some of these views are orthogonal, e.g., a project structure and the revision histories of single objects, others overlap in various ways, e.g., the UML class diagram and the source code representations of some piece of object-oriented software share the names of the classes that appear in them. Except for views of different types, also different view instances of the same type may overlap. The representation of a class may appear in different places in different class diagrams, since the class might participate in the different modules of the system whose structure is being shown in the class diagrams. Hence, some view related properties, like a class' position in a diagram, should not be modeled as an intrinsic property of the underlying repository object for the class, but rather live in some representation of the intersection of this shared object with a specific document (the class diagram). The same shared object has different values for this property in different contexts.

These observations suggest that the way tool integration is specified greatly impacts the modularity of SEEs. As a consequence, explicit support for separately specifying tool integration in languages that are used for constructing

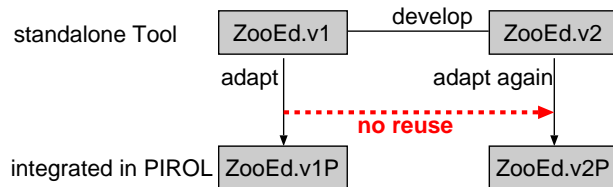


Figure 1: Evolution of the tool ZooEd

SEEs should result in more modular SEEs. When addressing tool integration into a repository based SEE, one should consider the following issues:

- Mismatches in data models of different tools should be tolerated.
- Tool specific models overlap in various fashions. We observe that the relation between repository objects and their different representations within tool specific models is isomorphic to the *played-by* relationship of *role* objects [17].
- Tools do not operate on isolated objects: tool specific documents are rather composed from a (large) number of (collaborating) objects. Integration of tools should therefore handle graphs of objects. This suggests that ideas from collaboration based modeling in general and *Adaptive Plug&Play Components* (AP&PC) [10, 9] in particular might be worth considering when designing mechanisms for tool integration in SEEs.

In the following we present the design of an SEE in a language which supports specifying tool integration concerns in first-class entities, called Dynamic View Connectors (DVC for short). We indicate the impact of DVCs on modularity of an existing SEE, called PIROL [3]. For doing so, in Sec. 2 we first shortly outline the maintainability problems we encountered with a previous design of PIROL, which did not include a separate integration layer. In Sec. 3, we present an improved design of PIROL in which DVCs are used to express integration and indicate how this improves modularity. Finally, Sec. 4 summarizes the paper and presents some related work.

## 2 Problems with Having Integration Concerns Tangled with Tool Functionality

PIROL is a three-tier system, whose bottom layer is a *repository* (we use H-PCTE[7] an implementation of the ISO standard PCTE[12]). This is where objects are stored persistently. The middle layer is a *workbench*, which represents a user’s session. The workbench provides an abstraction of the repository in which certain low level aspects are hidden away and methods are added to the repository types. Method definition happens in a special purpose language called LuaP (Lua [6] for PIROL) [5] which is interpreted within the workbench. *Tools* finally connect to the environment by means of a multicast messaging facility (this is based on MSG from the FIELD environment[13]).

An important feature of this messaging facility is the distribution of update messages that are issued by the workbench in order to keep the views of all tools consistent. A tool library transparently encapsulates repository objects by means of proxy classes. It is through this messaging facility, that a prerequisite to component integration is granted: tools may be developed as independent (binary) components, that are linked through some ‘middleware’, which provides the necessary communication patterns and also independence from any particular programming language. This property could be called “physical pluggability”. What remains, is a matter of reconciling the *logic* of different (independently developed) tools and the repository. In a first version of PIROL there was no support for defining “logical pluggability” separately from the tool implementation: the source code of tools whose model did not fit with the meta-model of the repository had to be modified in all places where data access occurred. This had severe impacts on tool maintainability and evolution as illustrated by the following example:

An editor for UML class diagrams, ZooEd[11], was first developed as a standalone tool (version *ZooEd.v1* in Fig. 1). The editor was later on adapted for integration into PIROL by inserting additional code into the source code, resulting in a new version (*ZooEd.v1P* Fig. 1). This adaptation was non-trivial, because it had to solve several problems: (a) duplication of actions of all user commands in order to write all data changes into the repository in addition to just modifying internal data, (b) translating incoming change notifications from the workbench into updates of the

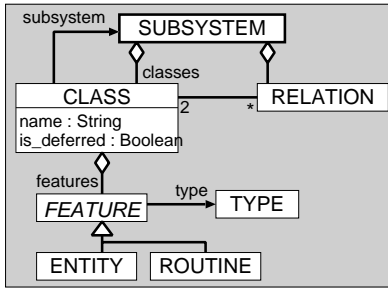


Figure 2: Extract from the repository’s meta model

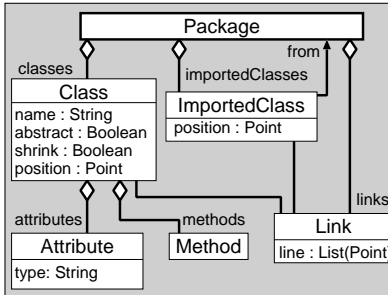


Figure 3: Meta model of a tool for UML class diagrams.

diagram view and (c) during both translations structural mappings had to be performed, because the tool’s internal data had a different structure than the repository model.

One can envisage an upgrade of the UML diagram editor *ZooEd.v2* that supports presenting the symbols for inner classes à la Java [1] as graphically contained within the symbol of the enclosing class. Evidently, the integration of *ZooEd.v2* into a repository could reuse much of the structural mappings of *ZooEd.v1*, since the data model of *ZooEd.v2* would be a “refinement” of *ZooEd.v1*’s model. However, if the adaptation code is hand-coded into the implementation of *ZooEd.v1P*, the structural mapping does not exist in one place and for this reason cannot be reused.

This motivated us to add special constructs, called *Dynamic View Connectors* (DVCs) to the language LuaP in order to ease the integration of independently developed tools into the SEE. A principal goal in this development was to decouple the meta-models of tools and repository: it should be possible to integrate tools that are implemented without (detailed) knowledge of the repository and its data model (the repository model). Furthermore, the repository and all tools should be allowed to evolve even with respect to their respective model without breaking the existing integration. Conceptually, DVCs share many features with AP&PC and their successor *Pluggable Composite Adapters* [9]: they can be considered as a concrete instantiation of these models in the domain of repository-based SEEs. At the implementation level, DVCs make use of *role* objects [17].

### 3 Dynamic View Connectors

We will present the main features of DVCs by means of a simple example. Consider the two models given in Fig. 2 and 3. Both capture roughly the same terse cut from a meta model for object-oriented software development. The first represents the “neutral” view of the repository: it focuses on abstractions and features that the repository designer has considered to be intrinsic for an object-oriented system, i.e., they are shared by different tools. On the contrary, the model in Fig. 3, looks at the object-oriented software development process from the specific perspective of an editor for UML class diagrams.

The Dynamic View Connector in Fig. 4 now brings these two separated worlds together by implementing the tool model (Fig. 3) on top of the repository model (Fig. 2). In other words: without changing the repository model the tool stores and manipulates objects of its private model which are then implemented by the connector as views of repository objects. For doing so, the DVC basically defines mappings between the two involved models concerning their classes, relations and attributes. Following the terminology of AP&PCs we call the classes of the tool model

*participant classes*, their instances are *participant objects*. *Repository classes* and *repository objects* refer to their counterparts at the repository level.

The example DVC in Fig. 4 defines four participant types (the names of their classes are put within boxes in Fig. 4), one of which is designated as the **root** of the model. The root participant class **Package** is mapped to the repository class **SUBSYSTEM**. Mapping participant classes to repository classes is specified by means of the **roclass** clause in the definitions of the participant classes. The outgoing links of a **Package** participant object, **classes** and **importedClasses**, are **filtered** views of the same association **classes** from the repository model. A filter **predicate** decides which objects of type **CLASS** contained in the repository association **classes** will be included in which of the resulting participant object lists, **classes**, respectively **importedClasses**. At runtime, those repository objects of type **CLASS** that pass the filter will automatically transformed to participant objects of type **Class**, respectively **ImportedClass**. This transformation will be the responsibility of connector instances created from our connector definition example (see below for more details on runtime dynamics). In this special case the filter simply inspects the defining context (the **subsystem**) of the target repository object in order to figure out which class object, contained in **classes**, is defined within the current context and which is imported from another subsystem.

Both participant classes **Class** and **ImportedClass** are mapped to the same repository class **CLASS**. However, they define two different views on the repository type **CLASS**, by making different sets of underlying **CLASS** attributes visible within the context of a UML tool. For some attributes the connector declares names and/or types that differ from those declarations in **CLASS**. Other attributes are added to the participant class without counterpart in the repository class. For instance, the attribute **abstract** in the definition of **Class** is identified by *renaming* with the attribute **is\_deferred** in the repository. That is, the repository-level attribute **is\_deferred** is made visible within the UML view under a different name, maintaining however the same type. The definition of the participant class **ImportedClass** contains an example of a repository-level attribute that is made visible within the view under a different name and type: the repository attribute **subsystem: SUBSYSTEM** (of repository type **CLASS**) is known within the view defined by the connector in Fig. 4 as the attribute **from: Package** (of participant type **ImportedClass**). Finally, attributes **shrink** and **position** in **Class** do not have counterparts in the core repository model. These attributes are **added** to the model (Note, that **position** is already defined in the abstract connector **GRAPH\_CONNECTOR** and made available in **CLASS** by inheriting from the participant class **NODE**).

The example connector in Fig. 4 contains also a situation where simple mappings are not sufficient. In the repository model (Fig. 2) the **type** of a **FEATURE** is represented by a repository object of class **TYPE**. However, the tool (Fig. 3) encodes the same information (the type of a feature) as a simple **String** in the repository class **Attribute** – the participant class corresponding to the repository class **FEATURE**. DVCs provide a construct to hand-code such mappings by implementing a **get** function and an **assign** function. These features, however, fall beyond the scope of this paper. The interested reader can find more details in [4].

In addition to participant definitions, a DVC may also define connector level attributes, e.g., **system** in Fig. 4. It also specifies the root participant of the view and defines the method to be used for applying the connector to repository objects – the **connect** method.

So far we have presented the static mapping defined by a DVC. Dynamic View Connectors are, however, first class objects that also contribute to the *dynamics* of the overall system. For each view of a graph of repository objects (**ROs**), a DVC instance is created – a connector object (**CO**) – and connected with (applied to) the root of the repository object graph, whereby lazily creating a corresponding graph of participant objects (**POs**). When examining the relationship between a **ROs** and its (possibly many) **POs**, two operations become necessary: *lifting* and *lowering*.

A **RO** is automatically *lifted* to a **PO** when it (re)enters the context of a **CO**. A **RO** may enter a connector as the result of invoking a method on a repository object within the context of the connector. The connector contains all those additional properties, that are attached to an object when it is seen as a participant object (cf. the attributes **shrink** and **position** in the example). Note that lifting requires the context of a connector in order to select from the set of participant objects that relate to the repository object to be lifted. Lifting an object for the first time corresponds to the creation of the respective participant object. This creation is lazy since it happens only when a repository object enters the scope of the connector and not when the connector is applied to the root **RO** from which the **RO** being lifted was reached. Lifting an object for the first time may need an initializer method for the added attributes playing a similar role to a constructor during object creation. A participant class that defines added attributes, also defines a method called **accept** that plays exactly this role.

```

Connector { GRAPH_CONNECTOR ;
  root = Node
  participants = {
    Node = {adds = {position : Point}},
    Edge = {adds = {line : List(Point)}},
  },
}
Connector { UML_CLASS_DIAGRAM_CONNECTOR ;
  inherit = GRAPH_CONNECTOR,
  attributes = {
    system : SYSTEM,
  },
  root = Package
  creation = connect,
  methods = {
    connect (root_ro : SUBSYSTEM)
      CONNECTOR:connect(root_ro)
      system = root_ro.get_system()
    end
  },
  participants = {
    Package = {
      roclass = SUBSYSTEM,
      filter = {
        classes : Class = { /* details omitted */},
        importedClasses : ImportedClass = { /* ... */},
        links : Link = { /* ... */},
      }
    },
    Class = { inherit = Node,
      creation = {place},
      roclass = CLASS,
      uses = {
        abstract = is_deferred
      },
      adds = {
        shrink : Boolean
      },
      filter = {
        attributes : Attribute = {
          base = {features : FEATURE},
          predicate (f : FEATURE)
            return f.conforms("ENTITY")
          end
        },
        methods : Method = {
          base = {features : FEATURE},
          predicate (f : FEATURE)
            return f.conforms("ROUTINE")
          end
        }
      },
      accept ()
        shrink = False
        ro.subsystem = co.root_ro
      end
    },
    ImportedClass = { inherit = Node,
      roclass = CLASS,
      uses = { from : Package = subsystem }
    },
    Attribute = {
      roclass = ENTITY,
      redirect = { type : String = { /* ... */ } }
    }
  }
  /* Participant Classes Method and Link omitted. */
}
}

```

Figure 4: Connector for UML class diagrams.

The reverse operation, *lowering*, requires no context, as each participant uniquely refers to one repository object. This operation is required, when passing a participant object down to the repository level. Finally issues related to object *identity* need to be reconsidered, since for some contexts, it is suitable to think of a participant object as being identical with the corresponding repository object, while in other situations that difference might be relevant. This distinction was introduced in [14]. Note that RO-to-PO mappings are transparent for the tools: view objects are read and written back just like ordinary objects, without the tool knowing about the RO/PO difference at all.

For the SEE PIROL it is crucial that the transparency between ROs and POs is not only a static relation, but change propagation to all relevant tools (through PIROL's messaging facility) also works for POs and between ROs and POs. No matter which view is changed all affected views are also notified. Only so, tools remain components in the full sense of the repository architectural style, or – in other words – the added property of “logical pluggability” does not interfere with the given “physical pluggability”.

## 4 Summary and Related Work

Our model supports separation of concerns in SEEs at two distinct levels. First, our *application domain* is that of software development using a variety of views. In fact, these views are concerns that can already be separated using different notations etc. All these views are stereotypic concerns that appear in most software models: static structure, dynamic behavior, functional decomposition and eventually source code. Allowing different views even within the same notation hints at the possibility to extend such an SEE towards techniques that support an arbitrary number of concerns being separated out into distinct documents.

Second, our technique maintains the separation of concerns even at the implementation level. Participant objects are distinguished from repository objects using the concept of role objects [14]. Graphs of participant objects are mapped to class graphs of repository object according to the AP&PC model [10]. This results in a great independence between the repository and its tools: both may evolve in terms of their meta models, without affecting the other, because most changes can be hidden from the other side by simply adapting the Dynamic View Connector.

Hence, Dynamic View Connectors share the same intention with SOP [16] and AOP [8]: development first produces separate concerns that later-on are combined in a (mostly) declarative style. A major difference between DVCs and SOP and also AOP lies on how different concerns are integrated. SOP and AOP basically perform integration at compile time by some sort of code transformation in order to melt new concerns into the definition of existing ones. With DVCs the integration is the responsibility of first-class objects during run-time. This allows to keep the concerns separated also after integration and facilitates dynamic reintegration. Independent evolution of tools is well supported, since the effects of evolution are mostly localized at the connectors, which exist outside the implementation of different concerns. Connectors are well maintainable, since they collect the mapping for all relevant classes of the participant class graph in one programming unit. Also inheritance can be applied to modularize connectors.

The current realization of the DVC model differs from current realizations of the SOP and AOP models with respect to the integration of concern-specific method definitions. Using DVCs, the integration is data driven, so methods remain untouched. To be more precise: an object in SOP or AOP may behave differently if a new Subject or Aspect is added to the system. In most cases, this is exactly what is intended, because all concerns should jointly contribute to a combined system behavior. For the moment being, DVCs only support context-based extensions that do not affect the intrinsic behavior of repository objects outside the context of a view. In other words, methods defined in a connector only affect the participant objects and not their corresponding repository objects. However, we are already working on an enhancement of the model that also allows to bind a method call on the repository level to a method in a view.

## 5 REFERENCES

- [1] K. Arnold, and J. Gosling. The Java programming language. Addison Wesley, 1997
- [2] D. Garlan. *Views for Tools in Integrated Environments*. PhD thesis, Carnegie Mellon University, May 1987.
- [3] B. Groth, S. Herrmann, S. Jähnichen, and W. Koch. Project Integrating Reference Object Library (PIROL): An object-oriented multiple-view SEE. In *Proceedings of the 7th Conference on Software Engineering Environments (SEE'95)*, Noordwijkerhout, Holland, April 1995.

- [4] S. Herrmann and M. Mezini. Modular design of software engineering environments with dynamic view connectors. Technical report, Technische Universität Berlin, 2000.  
<http://pirol.cs.tu-berlin.de/papers/DVC.ps>
- [5] S. Herrmann. Lua/P – a repository language for flexible software engineering environments. In *Procs. of CoSET Workshop at ICSE 2000*, Limerick.
- [6] R. Ierusalimsky, L. H. de Figueiredo, and W. Celes. Lua—an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.
- [7] U. Kelter. H-PCTE — a high-performance object management system for system development environments. In *Proc. COMPSAC '92*, pages 45–50, Chicago, Illinois, September 1992.
- [8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.M. Loingtier, and J. Irwin. Aspect Oriented Programming. In *Proceedings of ECOOP '97*, number 1241 in LNCS, pages 220–243.
- [9] M. Mezini, L. Seiter, and K. Lieberherr. *Software Architecture and Component Technology: State of the Art in Research and Practice*, chapter Component Integration with Pluggable Composite Adapters. Kluwer Academic Publishers, 2000 (to appear).
- [10] M. Mezini and K. Lieberherr. Adaptive Plug-and-Play Components for evolutionary software development. In *Proceedings of OOPSLA '98*.
- [11] A. Nordwig. Entwicklung einer Notation und eines grafischen Editors für den objektorientierten Entwurf hybrider Systeme. Master's thesis, TU Berlin, 1997.
- [12] ISO/IEC 13719-1: Portable Common Tool Environment (PCTE). Abstract specification, International Organization for Standardization (ISO), 1995.
- [13] S. P. Reiss. Connecting tools using message passing in the FIELD environment. *IEEE Software*, 7(4):57–66, July 1990.
- [14] 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*.
- [15] M. Shaw and D. Garlan. *Software Architecture: Perspectives of an emerging discipline*. Prentice Hall, 1996.
- [16] P. Tarr, H. Ossher, W. Harrison, and S. Sutton Jr. *N* degrees of separation: Multi-dimensional separation of concerns. In *Proc. of ICSE'99*.
- [17] R.J. Wieringa and W. de Jonge. Object identifiers, keys, and surrogates. *Theory and Practice of Object Systems*, 1(2):101–114, 1995.