Connecting Architecture and Implementation

Georg Buchgeher[1] and Rainer Weinreich[2]

[1] Software Competence Center Hagenberg, Austria
georg.buchgeher@scch.at
[2] Johannes Kepler University Linz, Austria
rainer.weinreich@jku.at

Abstract. Software architectures are still typically defined and described independently from implementation. To avoid architectural erosion and drift, architectural representation needs to be continuously updated and synchronized with system implementation. Existing approaches for architecture representation like informal architecture documentation, UML diagrams, and Architecture Description Languages (ADLs) provide only limited support for connecting architecture descriptions and implementations. Architecture management tools like Lattix, SonarJ, and Sotoarc and UML-tools tackle this problem by extracting architecture information directly from code. This approach works for low-level architectural abstractions like classes and interfaces in object-oriented systems but fails to support architectural abstractions not found in programming languages. In this paper we present an approach for linking and continuously synchronizing a formalized architecture representation to an implementation. The approach is a synthesis of functionality provided by code-centric architecture management and UML tools and higher-level architecture analysis approaches like ADLs.

1 Introduction

Software architecture is an abstract view of a software system. As such it abstracts from implementation details and data structures [1] and describes important elements, externally visible properties of these elements, and relationships among elements [1]. Different approaches for describing software architecture exist. Informal approaches and UML diagrams are typically used for architecture documentation. Formal approaches like *Architecture Description Languages* (ADLs) and architecture models [2] are used for automatically analyzing specific system properties.

Keeping an architecture description up to date and ensuring that the prescriptive (as-intended) architecture corresponds to the descriptive (implemented) architecture are still central problems in software development [2][3][4]. Approaches addressing these problems exist. For example, UML-tools may extract architectural abstractions from an implementation and thus provide a view of the currently implemented architecture. Architecture management tools like Lattix [5], SonarJ and Sotoarc [6] additionally support comparison and synchronization of

R. Meersman, P. Herrero, and T. Dillon (Eds.): OTM 2009 Workshops, LNCS 5872, pp. 316–326, 2009.

the intended and the implemented architecture. However, these approaches operate on the concepts at the low abstraction level provided by object-oriented programming languages (classes and interfaces). They fail to support higher-level abstractions like components, systems, and systems of systems. Typically they also work only for homogenous programs and do not inherently support heterogeneous software systems, like service-oriented software architectures. Approaches extracting architecture information from code also typically lack validation capabilities like ADLs. On the other hand, general-purpose ADLs like xADL [7] and ACME [8] provide only limited support for connecting architecture description and code and offer no synchronization support.

In this paper we describe an approach for connecting and synchronizing architecture description and system implementation. The approach is a synthesis of functionality provided by code-centric architecture management and UML tools and higher-level architecture analysis tools like ADLs. It supports both the description of high-level architectural elements like components and systems and of low-level concepts like classes and interfaces. Low-level abstractions are extracted from an implementation similar to reverse engineering and architecture management tools and can be synchronized with a prescribed architecture. High-level abstractions are either defined manually or extracted from a system implementation. In the latter case, the system analyses technology-specific component code, meta-data and configuration files to extract the needed information. Since we use an ADL for architecture representation, the approach also supports architecture validation based on constraints. Heterogeneous systems are supported through different technology bindings.

The remainder of this paper is structured as follows. In Section 2 we describe the basic elements of our approach, which are an ADL for architecture description and a toolset working on this ADL. The toolset provides visual editors for visualizing and manipulating an ADL-based architecture model, functionality for extracting information from an implementation, and functionality for synchronizing prescriptive and descriptive architecture. In Section 3 we provide an example to illustrate our approach. In Section 4 we describe related work in more detail. The paper is concluded in Section 5.

2 Approach

The main elements of our approach are an ADL, called LISA (*Language for Integrated Software Architecture*), and a toolkit working on LISA-based architecture models, the LISA-Toolkit (see Figure 1). The toolkit is implemented on the basis of the Eclipse platform and provides a number of plug-ins for architecture modelling, visualization, validation, and implementation synchronization. Some of the available editors and views for architecture modelling and visualization are shown in the next section.

The LISA-ADL is an extensible XML-based language for representing structural relationships of heterogeneous software systems. It is implemented by XML-based metamodels for describing different aspects of a software system. We

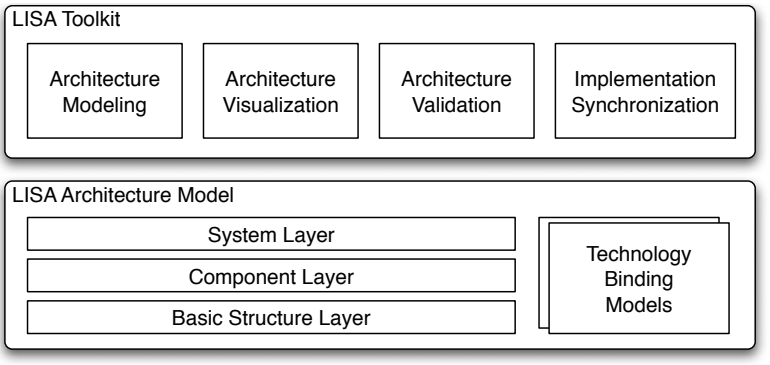

Fig. 1. LISA Overview

provide no dedicated alternative textual representation, since LISA-based architecture models are created and manipulated visually using the LISA-toolkit. As shown in Figure 1, LISA provides concepts and models for describing the architecture of a software system at different abstraction layers.

The basic structure layer is used for describing low-level abstractions like classes, interfaces, and packages, but also contains elements for defining elementary structures and constraints like modules, subsystems, and layers. Most architecture management tools are working at this layer. Some elements at this layer, like classes, can be extracted from an implementation through language connectors, other elements like layers need to be defined manually.

The component layer provides concepts for describing component-based architectures like components, ports, and contracts. In our model, components denote elements meant for late composition. Components may provide services to other components through service ports and they may reference other components through reference ports. Ports specify additional composition semantics. Examples are whether a connection is optional, whether it is created automatically or manually, and which connectors can be used for connections. Components and ports are independent from a particular implementation paradigm. If components are implemented through an object-oriented language, their implementation description can be refined through elements at the basic structure layer. However, LISA also supports bindings to other (non object-oriented) implementation technologies through technology bindings. Technology bindings are defined by means of technology-specific binding models (see Figure 1). Currently, we support technology bindings for EJB, Spring, OSGi, Spring Dynamic Modules, and SCA.

The system layer is used for describing the architecture of a whole system or application. A system consists of configured and connected component instances. Since components may be implemented differently, heterogeneous systems can be described. Hierarchical decomposition of systems is supported through composites. Even at this level, technology bindings are used for connecting and synchronizing architectural elements like systems and composites to technology-specific configuration files.

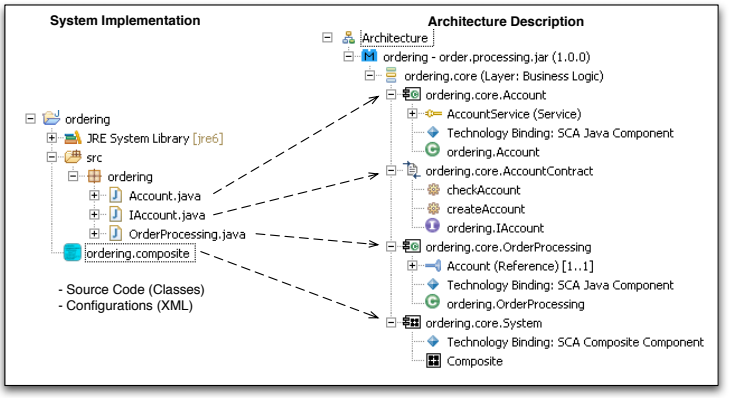

Fig. 2. Implementation/Architecture Mapping

Figure 2 shows the basic approach for mapping elements from a system implementation to the architecture description by means of actual views provided by our toolkit. The left side of Figure 2 shows the typical project structure of an eclipse based SCA/Java project. In the example shown, the project consists of one package with two Java classes, a Java interface, and of a configuration file describing an SCA-based system configuration (*ordering.composite*). The right side of Figure 2 shows the architecture decomposition editor of the LISA-toolkit. An architecture description (see top level element at the right side) is partitioned into modules and may contain layers (see *order.processing.jar* and *ordering.core* in Figure 2). The figure also shows components (*Account*, *OrderProcessing*), one contract (*AccountContract*) and one system definition (*System*). The dashed arrows in Figure 2 show the actual mapping of implementation artifacts to elements of the architecture model. The mapping is described using technology-specifc bindings, which are also shown in Figure 2. If possible, technology bindings are extracted from implementation artifacts by analyzing code, metadata, and configuration files. In cases where no implementation is available (yet) or when information can only partly by extracted from an implementation, technology bindings need to be created or completed manually.

The actual process of synchronizing architecture and implementation is shown in Figure 3. A system implementation is created and manipulated by editors that are part of the Eclipse IDE or of third party plug-ins like the Spring IDE. The architecture description is created and modified through architecture editors provided by the LISA toolkit. Both representations are continuously being observed for changes.

Changes of the system implementation are detected by extending the Eclipse project builder infrastructure. Whenever resources are changed an incremental build process is started. An *Architecture Project Builder* analyzes the changed resources. *Language Builder Extensions* are used for extracting information about low-level architectural elements like classes and interfaces from an implementation and for creating the architecture model at the *Basic Structure Layer*.

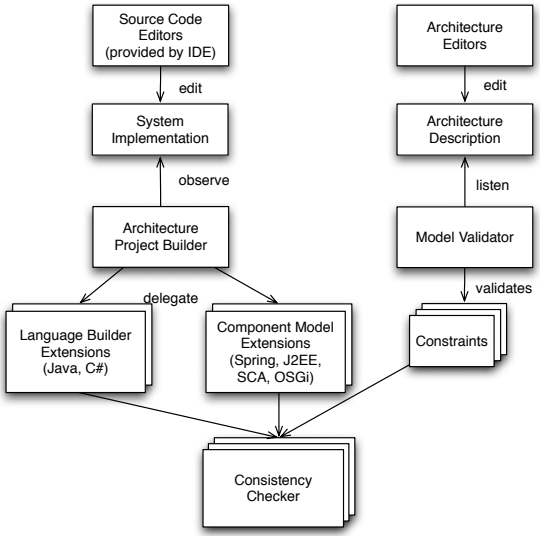

Fig. 3. Synchronization Approach

Technology-specific *Component Model Extensions* search the source code for component meta-data, analyze component configuration files, and create and validate architecture elements at the component layer.

The architecture model is observed by a *Model Validator*, which validates an extensible set of constraints whenever the architecture description is modified. *Consistency Checkers* are responsible for comparing architecture description and system implementation. Differences between both representations are visualized as validation problems and shown in both representations. Problems can then either be resolved by changing the system implementation, or by changing the architecture description.

3 Example

In this section we show some of the described concepts by means of a small example for order processing. The example is intended to illustrate three aspects of our approach. In a first step we show how components and systems can be modeled at a high-level of abstraction without any implementation binding. In a second step, we will show how technology-specific implementation bindings can be defined. The last step will illustrate how low-level architectural abstractions like classes and interfaces that are created as part of an implementation are connected to the high-level abstractions that were defined during system modeling. The presented scenario is only one of several possible ways to use our approach. We will discuss alternative ways at the end of this section.

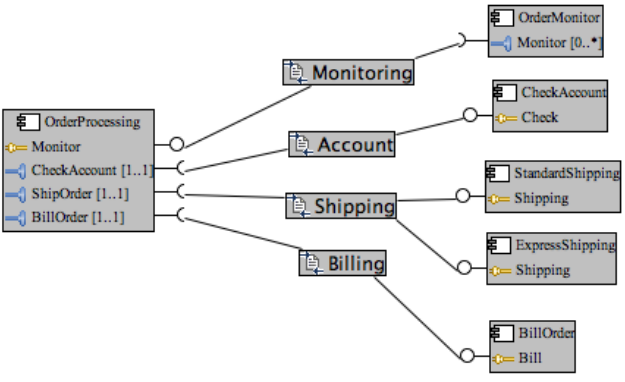

Fig. 4. Component Diagram

Step 1: Component and System Modeling
We start by modeling a system from scratch. This means we follow a top-down approach. First we define components and contracts for the order processing system. Modeling can be performed in different graphical diagram editors provided by the toolkit. The result of the modeling process is shown in the component diagram depicted in Figure 4.

The diagram shows components, their ports, as well as component/contract-dependencies (no connections). For example, the *OrderProcessing* component provides one service port (*Monitor*) and several reference ports, which can be connected to other components. The *ShipOrder* reference port has a multiplicity of 1..1 and must be connected to another component supporting the Shipping contract. The figure shows that currently two components (*StandardShipping*, *ExpressShipping*) support this contract and can be used in a concrete configuration of the *OrderProcessingComponent*. After defining components, instances of these components can be used for defining a system configuration[1]. Figure 5 shows the configuration of the *OrderProcessing* system. As shown in the figure, the system consists of two main parts which have been placed in different layers. The business logic layer contains an instance of the *OrderProcessing* component, which has been connected to instances of *CheckAccount*, *StandardShipping*, and *BillOrder*. The figure shows an additional aspect: the described configuration at the business logic layer forms the implementation of a newly defined composite component (*OrderingSystemComposite*). Since the *OrderingSystemComposite* is itself a component, the described model would allow defining multiple instances of this composite component. The *OrderMonitor* is part of the UI layer and has been connected to the ordering system. As described before, layers are used for

[1] Component instances are configured components, where values are supplied for properties, and connections. Multiple run-time instances of a component instance are possible.

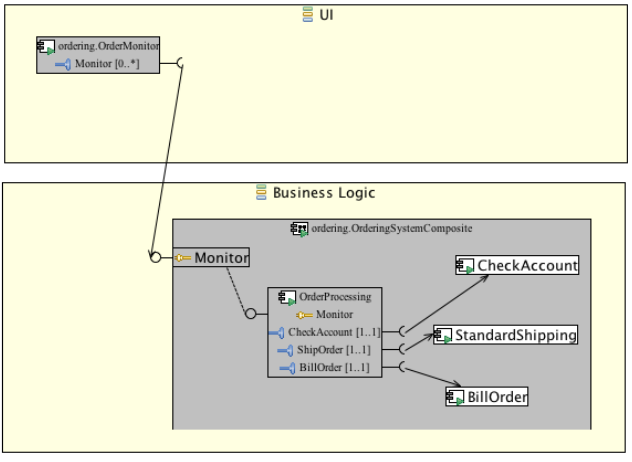

Fig. 5. System Diagram

constraining architectural structures. In our approach, layering constraints can also be used at the level of component configurations.

The validation engine of the LISA toolkit continuously validates the architecture description. Examples for problems that are detected at the level of component and system modeling are missing connections, unset property values, and layer violations.

Step 2: Definition of Technology Bindings
After modeling components and systems, the defined components can be bound to technologies and to component implementations. A component definition can be directly bound to an implementation in a specific technology. However, it is also possible to define the technology first and provide the implementation afterwards. This way it is possible to make technology-specific decisions and to perform technology-related validation before actually providing an implementation.

Technology decisions are modeled with technology-specific implementation bindings. Figure 6 shows the defined components in a different diagram. In this diagram, SCA bindings have been attached to the different components. Since SCA provides itself an implementation abstraction, the selected bindings are SCA Java (for binding to SCA components implemented in Java) and SCA Composite (for binding to SCA composites). As described in Section 2, bindings to different technologies are possible, even within the same system. The error annotations that are displayed on every implementation binding indicate that the implementation of the components is still missing, i.e., the binding is incomplete.

Step 3: Connecting to Implementation-Level Abstractions
In the final step of this scenario, the defined components are implemented and implementations are bound to component definitions. The toolset supports technology connectors for different component technologies and programming

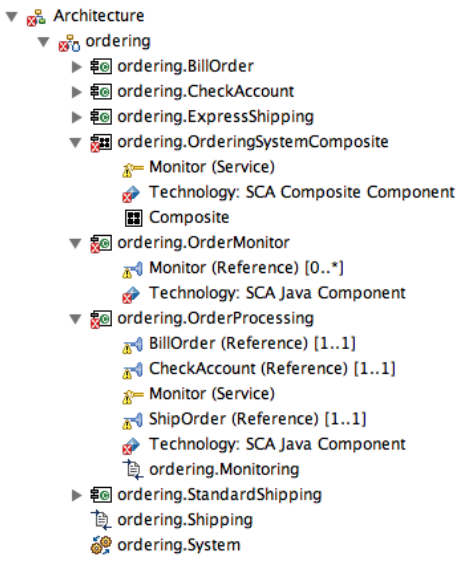

Fig. 6. Components with Technology Bindings

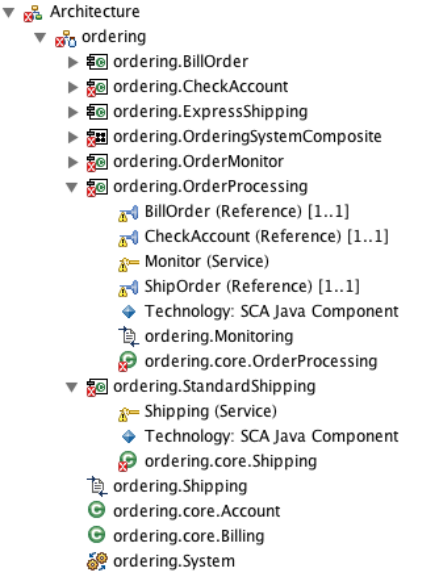

Fig. 7. Architecture Model with Language Abstractions

languages. In the case of the presented example, it uses technology connectors for SCA and Java to extract architecture-related information from an implementation. Architecture information at the component-level like SCA component definitions (derived from Java annotations in code) and SCA composite configurations (extracted from composite definitions) are used for validating and synchronizing architecture information with the implementation. For example, the toolkit detects when the component definition in an implementation differs from the component definition in the architecture model and vice versa.

Low-level architectural elements, like classes, interfaces, and static dependencies are automatically extracted by language connectors. The extracted elements have to be assigned manually to the concepts at a higher-level of abstraction. Figure 7 shows the model from Figure 6 with classes extracted from an implementation. Some of the extracted classes have been assigned to components (*ordering.core.OrderProcessing* and *ordering.core.Shipping*); others still need to be assigned (*ordering.core.Account* and *ordering.core.Billing*). The resulting architecture model now contains abstractions at higher and lower levels of abstraction and is continuously synchronized with an implementation.

The additional elements and dependencies at the language level can now be analyzed with regard to the higher-level elements as shown in Figure 8. The figure shows that the component implementations introduce static dependencies between components (shown as red lines in Figure 8). These are potentially unwanted dependencies because components may be directly affected through changes to the implementation of another component and components cannot

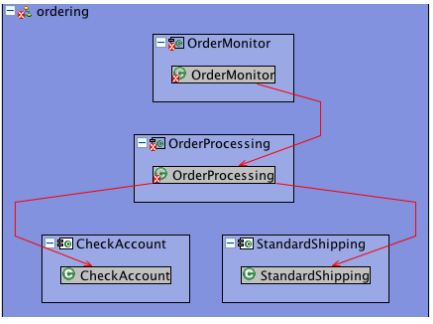

Fig. 8. Static component dependencies

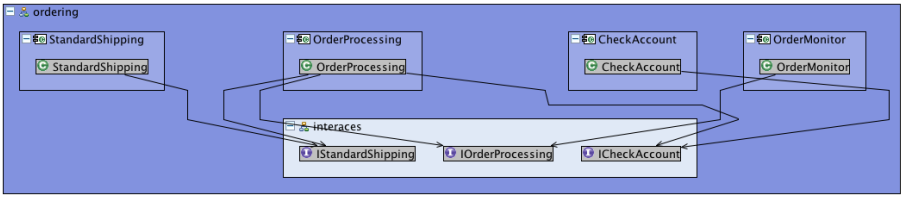

Fig. 9. Architecture without static component dependencies

be deployed independently from each other. The toolkit is able to detect such potential problems and an architect or developer may change the implementation to remove direct dependencies between component implementations. The result of such a refactoring is shown in Figure 9, where interfaces implementing a contract have been introduced and moved to a separate library.

In the example we used a top-down approach. We started by modeling at a high-level of abstraction and eventually provided technology bindings and component implementations. It would also be possible to start with an implementation and to define higher-level concepts later in the process. This way an architecture description can be provided for already implemented systems. Also, a combination of both approaches can be used. The elements of an architecture model may be partly implemented and partly specified. This is useful for modeling and analyzing the impact of extending an existing system.

4 Related Work

Our approach is based on an ADL, which is continuously synchronized with a system implementation and which supports the description of heterogeneous distributed system architectures. ADLs are primarily a means for system modeling and automated analysis. Connecting architecture description and code is a long known deficiency of ADLs [9].

xADL [7] and its related toolkit ArchStudio [10] allow specifying unidirectional connections from the ADL to implementation artifacts (by extending an abstract implementation model). However, architecture and implementation are not synchronized. Changes to the architecture model affecting the implementation are not detected. Equally changes to the implementation affecting the architecture description are not identified. Additionally, xADL provides no explicit support for connecting to component models and technologies used in practice. Finally, there is no mapping from low-level structural models (like class relationships) to high-level concepts like components. As shown in the last section such integration enables important structural analysis, continuous conformance analysis, and seamless modeling from high-level to low-level abstractions.

The Fractal ADL [11] has been designed as a configuration language for the Fractal Component Model. While theoretically possible, LISA has explicitly not been designed as a configuration model for a particular component technology. Rather LISA system models are mapped to and synchronized with system configurations of component technologies like Spring and SCA. Like in xADL, connections to an implementation are only one-way relationships in the Fractal ADL. Validation of consistency between architecture description and implementation, architecture synchronization, integration of low-level structural models, and seamless modeling as in our approach are equally not supported by the Fractal ADL.

ArchJava [9] is an extension to Java, which unifies architecture description and implementation in one representation. The focus of ArchJava is to ensure that components only communicate via defined ports. This is called communication integrity in ArchJava. Communication integrity can also be checked in LISA (see Figure 8). Main drawbacks of ArchJava are its support for a single language and a single JVM. It also requires a compiler and defines its own component model. As a result, ArchJava does not support existing component technologies and cannot be applied for describing heterogeneous systems. Since it is bound to an implementation language, it also does not support architecture modeling.

The Unified Modeling Language (UML) is a general purpose modeling language originally designed for the design of object-oriented systems. Improved support for the description of component-based systems and software architectures has been added as part of UML 2.0. Most UML tools offer reverse and forward engineering capabilities for creating UML diagrams from a system implementation and vice versa. However, this functionality is primarily available for class and sequence diagrams, i.e., for low-level architectural and implementation information. UML-based MDA tools like AndroMDA support code generation for specific component technologies. This is, however, restricted to forward engineering. Continuous and automated architecture analysis like in our approach is not supported by UML-tools [3].

5 Conclusion

We have presented an ADL-based approach for connecting architecture and implementation. The approach supports modeling of system architectures at

different levels of abstraction, binding architectural concepts to different technologies, immediate conflict detection, and continuous synchronization of architecture and implementation. Like other ADLs our approach supports system modeling and architecture analysis at a high-level of abstraction. Contrary to other approaches we focus on connecting and continuously synchronizing the architecture description with system implementation. This ensures that the architecture model always reflects an abstract high-level and automatically analyzable view of the currently implemented architecture. Finally we combine static analysis as provided by architecture management tools with ADL concepts and support heterogeneous systems through different technology-bindings, even within one system.

References

1. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edn. Addison-Wesley Professional, Reading (2003)
2. Garlan, D.: Formal Modeling and Analysis of Software Architecture: Components, Connectors, and Events. In: Bernardo, M., Inverardi, P. (eds.) SFM 2003. LNCS, vol. 2804, pp. 1–24. Springer, Heidelberg (2003)
3. Shaw, M., Clements, P.: The Golden Age of Software Architecture. IEEE Softw. 23(2), 31–39 (2006)
4. van Gurp, J., Bosch, J.: Design Erosion: Problems and Causes. Journal of Systems and Software 61(2), 105–119 (2002)
5. Sangal, N., Jordan, E., Sinha, V., Jackson, D.: Using Dependency Models to Manage Complex Software Architecture. SIGPLAN Not. 40(10), 167–176 (2005)
6. hello2morrow GmbH: Sotoarc and SonarJ (2009), `http://www.hello2morrow.com` (accessed: June 17, 2009)
7. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: A Comprehensive Approach for the Development of Modular Software Architecture Description Languages. ACM Trans. Softw. Eng. Methodol. 14(2), 199–245 (2005)
8. Garlan, D., Monroe, R., Wile, D.: Acme: An Architecture Description Interchange Language. In: CASCON 1997: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research. IBM Press (1997)
9. Aldrich, J., Chambers, C., Notkin, D.: ArchJava: Connecting Software Architecture to Implementation. In: ICSE 2002: Proceedings of the 24th International Conference on Software Engineering, pp. 187–197. ACM Press, New York (2002)
10. Institute for Software Research - University of California: ArchStudio 4 (2009), `http://www.isr.uci.edu/projects/archstudio/` (accessed: June 17, 2009)
11. ObjectWeb Consortium: Fractal ADL (2009), `http://fractal.ow2.org/fractaladl/index.html` (accessed: June 17, 2009)