Expert Systems
The Journal of Knowledge Engineering

Article

DOI: 10.1111/j.1468-0394.2010.00533.x

A case-based reasoning approach to derive object-oriented models from software architectures

German L. Vazquez, J. Andres Díaz-Pace and Marcelo R. Campo
ISISTAN Research Institute, UNICEN University, Campus Universitario, B7001BBO Tandil, Buenos Aires, and CONICET-Argentina, Argentina
Email: adiaz@exa.unicen.edu.ar

Abstract: *Software architectures are very important to capture early design decisions and reason about quality attributes of a system. Unfortunately, there are mismatches between the quality attributes prescribed by the architecture and those realized by its object-oriented implementation. The mismatches decrease the ability to reason architecturally about the system. Developing an object-oriented materialization that conforms to the original architecture depends on both the application of the right patterns and the developer's expertise. Since the space of allowed materializations can be really large, tool support for assisting the developer in the exploration of alternative materializations is of great help. In previous research, we developed a prototype for generating quality-preserving implementations of software architectures, using pre-compiled knowledge about architectural styles and frameworks. In this paper, we present a more flexible approach, called SAME, which focuses on the architectural connectors as the pillars for the materialization process. The SAME design assistant applies a case-based reasoning (CBR) metaphor to deal with connector-related materialization experiences and quality attributes. The CBR engine is able to recall and adapt past experiences to solve new materialization problems; thus SAME can take advantage of developers' knowledge. Preliminary experiments have shown that this approach can improve the exploration of object-oriented solutions that are still faithful to the architectural prescriptions.*

Keywords: architecture and object-oriented design, quality attributes, automated design assistance, case-based reasoning

1. Introduction

Software developers are compelled to make the right design decisions for a system in order to adequately fulfill the stakeholders' requirements. The software architecture prescribes a high-level structure for the system and a gross partitioning of functionality, independently from implementation issues (Bass *et al.*, 2003). Therefore, the architecture provides a convenient level of abstraction for developers to reason about quality attributes (e.g. modifiability, performance, security, availability, among others) and business goals. In later development stages, the developers usually refine the architecture into a more concrete design model, adding low-level information to produce a system implementation. Since the object-oriented paradigm

is a customary choice to realize the implementation, we have referred to this refinement as *object-oriented materialization of software architectures* (Campo *et al*., 2002). Conceptually, the concerns engineered at the architectural level (e.g. structural and behavioral features, and quality-attribute properties) should be preserved in downstream levels of the materialization, so as to ensure the benefits of software architectures through the life cycle. However, there are often mismatches between the architectural abstractions and the object-oriented counterparts used to realize them, which leads to multiple possible materializations for the same architecture. For example, the protocols of communication among architectural components (also called connectors) are not first-class citizens in a typical class-based language such as Java. Because of this, the communications captured in the software architecture must be mapped to a set of interactions between objects. The decisions that developers make regarding this mapping will affect properties of the resulting system such as modifiability, performance and other quality attributes.

Developing an object-oriented materialization for a given software architecture is a complex activity, because the links between architectural and object-oriented designs are not straightforward. Decision-making depends on the quality attributes involved, but it also depends on factors such as the developer's experience, infrastructure frameworks or application domain. Normally, developers reuse knowledge about proven design solutions when doing materializations, just as in other forms of pattern-driven development. A common practice is to base materializations on particular patterns called architectural styles (e.g. client–server, publish–subscribe, layers etc.) (Buschmann *et al*., 1996), as they usually come with implementation guidelines, even though materializations will still diverge from the intended architecture. These divergences are often caused by two forces: (i) the need to customize pattern-based solutions so that they can fit a particular problem; and (ii) a tendency of programmers to 'over-design' (when customizing) with little consideration for quality-attribute requirements. As the conformance of the implementation to the original architecture decreases, the ability to reason architecturally about the system gets diminished. This conformance problem is well known, although few approaches have proposed solutions for it (Medvidovic *et al*., 2002; Aldrich, 2008), and those solutions pay scant attention to quality attributes. In this context, knowledge-based tools are very helpful to address the challenges of object-oriented materializations. In particular, we think that such tools should assist developers to explore the space of 'allowed' materializations for an architecture, in which the stylistic and quality-attribute features of that architecture are preserved. However, to the best of our knowledge, assistive tools for quality-driven materialization are lacking.

In previous work, we developed a tool called ArchMatE (Díaz-Pace & Campo, 2005) to guide developers in the materialization of architectural styles with different balances of quality attributes. Although this approach proved useful for dealing with quality attributes, a drawback of style-based materializations is the coarse-grained granularity of architectural styles. ArchMatE generated object-oriented skeletons based on predefined styles, and the developer had little freedom to customize the skeletons. Reflecting on these issues, we realized that architectural styles are not always the best analogy for the materialization. We observed that the materializations are actually aggregations of fine-grained fragments that separately reify components and connectors of the architecture. In particular, the types of connectors have impact on the quality-attribute properties of both the architecture and the resulting object-oriented system (Medvidovic *et al*., 2002). Furthermore, connector types are materialized through object-oriented structures, which can be combined by the developer into an implementation for her (or his) architecture.

In this paper, we present a tool approach called SAME (Software Architecture Materialization Environment), in which the architectural connectors are the focus of the materialization process. The knowledge used by SAME comes essentially from *developers' experiences when materializing connectors*, rather than from the knowledge provided by architectural styles

(as in ArchMatE). That is, SAME supports the developer in the exploration of object-oriented solutions by mixing and matching various connector implementation options, which are then tailored by the architect's preferences and context. For instance, if the input is a client–server architecture, this architecture will not be materialized as a whole. Instead, the developer will identify the request-reply connectors present in the architecture and then s/he will adapt specific connector implementations in order to obtain a global object-oriented solution. In some cases, a simple HTTP implementation of the request-reply connector is enough. However, let us assume that there are modifiability requirements stating that the clients subscribe to topics and initiate requests upon notifications of those topics. If so, the developer should consider an event-based or peer-to-peer implementation of the request-reply connector. In most organizations, this kind of knowledge resides on 'gurus', who suggest the selection of connector implementations based on past experience and similar systems built (Mattmann *et al.*, 2007). This valuable materialization knowledge, although informed by architectural principles, is personal and memory-based in nature.

The experts' knowledge about connector-based materializations can be certainly captured, and quality-attribute heuristics can be developed to build an expert system. Along this line, frame-based codifications of the relationships between software development problems and architectural approaches have been proposed (Rapanotti *et al.*, 2004; Choppy *et al.*, 2005). Nonetheless, tool support and quality-attribute analyses are still pending issues. For these reasons, we have approached SAME as an expert system that relies on the case-based reasoning (CBR) paradigm (Kolodner, 1993). Each case represents a materialization experience for an architectural connector, along with predetermined quality-attribute levels. Structurally, a case describes the design context of a connector and its attached components (the problem), and proposes an object-oriented implementation that reifies that context (the solution). The SAME tool implements the exploration of materialization alternatives according to the typical CBR phases (Lopez De Mantaras *et al.*, 2005), namely retrieve, reuse, revise and retain. Initially, the developer describes the architecture by means of an editor for UML2 component diagrams (Ambler, 2005). The tool is equipped with a repository of cases that are potentially useful to materialize types of architectural connectors. After analyzing the connector instances of the input architecture, the tool selects from the repository the set of experiences that are most relevant for the current connectors. Then, these relevant experiences are adapted to fit the new problem specification, so that their design knowledge can be used to devise a new object-oriented solution for the input architecture. The developer can assess the quality of the derived solution, accepting or rejecting the experiences proposed by SAME, until a solution with acceptable quality-attribute characteristics is found.

The combination of connector-based materializations, quality-attribute information and CBR techniques proposed by SAME has four main advantages. First, it helps to control mismatches between a software architecture and its possible object-oriented implementation(s), particularly those mismatches involving the key qualities prescribed by the architecture. Second, it supplements style-based materializations by providing a more flexible materialization framework. Our case studies with SAME have shown that it outperforms ArchMatE regarding the number of materializations for a given architecture. Third, it leverages materialization knowledge from experts as well as from connector taxonomies. The experts' knowledge is vital here, because it captures how quality attributes are tied to object-oriented designs in practice, and thus it makes SAME customizable. The creation of the base of cases is a knowledge-intensive task (as in ArchMatE), but the fine-grained granularity of connector-based experiences permits fine-grained reuse of materializations. Fourth, the CBR tool described in the paper provides the computational support for realizing the three aspects mentioned above.

The rest of the work is organized into six sections as follows. Section 2 gives the foundations of our materialization approach, explaining the differences between SAME and previous

work through a motivating example. Section 3 is devoted to the representation, retrieval and adaptation of materialization experiences within the CBR paradigm. Section 4 describes the CBR engine used in SAME for the exploration of object-oriented implementations. Section 5 discusses results of case studies and main lessons learned. Section 6 analyzes related work. Finally, Section 7 presents the conclusions and directions for future research.

2. Background: the materialization problem

The software architecture is the primary carrier of the quality attributes (e.g. performance, modifiability or reliability) for a system (Bass *et al.*, 2003). At the architectural level, system quality is achieved by an adequate distribution of the system responsibilities among components and by the interactions of these components in order to realize their responsibilities. For instance, the attribute-driven design (ADD) method (Bass *et al.*, 2002) proposes a recursive process of decomposition, in which the developer constructs the architecture by applying tactics and patterns to the extent of satisfying the key quality-attribute requirements for the system. This architecture deliberately defers decisions about specific implementation technologies. Any implementation will conform to the architecture as long as it adheres to the design principles and constraints dictated by that architecture. However, unlike in architectural methods such as ADD, the transformations and decompositions applied by developers to arrive at a satisfactory implementation are mostly based on idiosyncratic and intuitive reasoning. Hence, an important research issue is how to guide developers to perform more systematic object-oriented materializations.

The classical approach to materialization is to define an object-oriented framework for a reference architecture (Tracz, 1995; Medvidovic *et al.*, 2002). Unfortunately, these materializations are limited regarding quality attributes other than modifiability, because the trade-offs are resolved and 'frozen' at the framework level. Recent advances in model-driven engineering (MDE) (Schmidt, 2006) have fostered automatic transformations from architectures to code, but the generation of implementations with known quality-attribute properties is still a hard problem. Instead of automatic generators, we believe that it is possible to have *design assistants* able to point out promising paths of materializations for a given architecture. Although this kind of assistant cannot explore the whole design space, they can certainly help developers to make informed decisions for the implementation and reason about the consequences of these decisions.

2.1. The base approach: exploring materialization alternatives for architectural styles

Architectural styles allow a developer to create a 'skeletal system' from the architecture (Bass *et al.*, 2003), in which s/he should first implement the object-oriented software dealing with the execution and interaction of architectural components, and then progressively implement the system functions. ArchMatE was a prototype of design assistant based on architectural styles (Díaz-Pace & Campo, 2005). In this approach, we analyzed the design knowledge contained in architectural styles and then linked variants of each style to small object-oriented frameworks, called *framelets* (Pree & Koskimies, 2000). When adding a particular style to ArchMatE and creating framelets for the style, we took advantage of general strategies to decode which elements of the style should be explicitly modeled through classes and which ones should be condensed or even skipped from the object-oriented solution. Examples of these strategies are constraint strengthening, constraint relation and direct mapping (Campo *et al.*, 2002). In this way, we preserved the architectural style intent, while admitting quality-attribute variations in the materializations.

In order to discuss the outcomes of the style-based approach, let us consider a simple blackboard architecture for a chess game system, as depicted in Figure 1(a). The architecture consists of three types of components: a *Board* component that is responsible for maintaining the internal state of the game and notifying the occurrence of

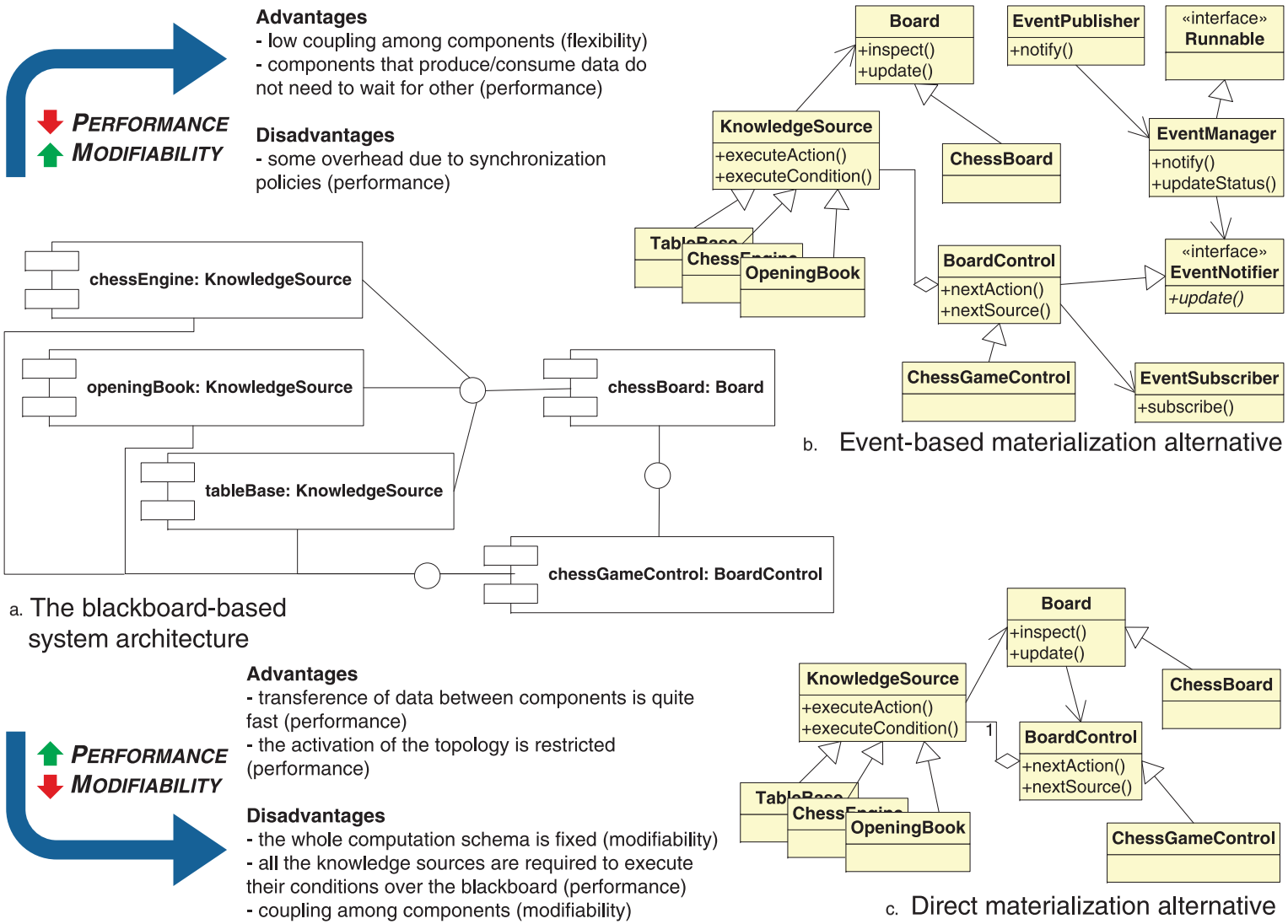

Figure 1: *Two object-oriented implementation variants for a blackboard architecture.*

changes in the state; various *KnowledgeSource* components that query and update the *Board*; and a *Control* component that is in charge of executing the *KnowledgeSources* in some order. Architecturally speaking, blackboards are designed to support modifiability, because their constituent components work in a decoupled fashion. Nonetheless, developers can tune the modifiability of a particular system (and therefore affect the system's performance), depending on the mechanisms implementing the components and connectors. The developer would feed the ArchMatE tool with a specification of the chess-system architecture, and would enter the preferences for her (or his) materialization. These preferences are what direct the search for possible instantiations of framelets, depending on the characteristic of the base style. The input architecture is specified via an architectural description language (ADL) (Medvidovic & Taylor, 2000).

Figure 1(b) shows a possible object-oriented solution for this example, as suggested by ArchMatE. This solution is structured around a framelet that captures certain features of the blackboard architectural style (e.g. repository, knowledge sources and control), and also permits the instantiation of application-specific responsibilities. In particular, note that the communication between components is achieved through an *Event* mechanism. Let us suppose now that the developer seeks an alternative materialization for the architecture of Figure 1(a), because the concurrency imposed by the *Event* mechanism may become a performance concern. This means that ArchMatE should consider a new trade-off between performance and modifiability.

A more satisfactory solution generated by the tool is presented in Figure 1(c). The new framelet performs a direct mapping of the style, as reported in Buschmann *et al.* (1996), and it implements all the components, ports and connectors via object-oriented classes. Note that performance is improved at the cost of fewer intermediaries between the repository and the control components. This exploratory modality of assistance was supported by a rule-based engine (Díaz-Pace & Campo, 2005). Nonetheless, the implementation of ArchMatE as a rule-based system exposed some limitations of the approach, namely:

- *Coarse-grained materializations*. The tool derives implementation alternatives of the same kind, because the relationships between a given style, its variants and framelets are fixed.
- *No support for heterogeneous architectures*. The generation of materializations alternatives for architectures comprising multiple styles was not possible, because ArchMatE cannot reuse the knowledge developed for individual styles.
- *Few opportunities for customizing implementations*. Despite the guidance of styles, developers have to (and want to) make some decisions at intermediate stages of the exploration process. However, the use of rules in ArchMatE for selecting materialization cannot easily incorporate the developer's feedback as the materialization proceeds.

When examining several framelets, we found that the object-oriented elements used for connectors occurred recurrently in many systems. Interestingly, the concrete implementation of connectors may differ (e.g. the specific communication protocols in a socket, or the particular method that may be invoked in an RPC connector), but the generic object-oriented structure that makes them work remains constant across implementations. This hint led us to an alternative approach for materializing architectures.

2.2. *A more flexible approach: exploring alternatives driven by architectural connectors*

The treatment of connectors as first-class citizens has been regarded as a key contribution to the study of software architectures (Shaw, 1996). Connectors commonly used in architectural descriptions include push/pull procedure call, producer–consumer connectors, event channel and observer connectors brokers, SOAP[1] connectors,

[1] *Simple Object Access Protocol: it is a protocol for exchanging XML-based messages over networks, normally using HTTP/HTTPS.*

acceptor/connector, semaphores and transporters, among others (Shaw, 1996; Fernandez, 1997; Mehta *et al*., 2000). Since connectors are the medium through which components realize their responsibilities, we argue that the object-oriented mapping of the connectors of an architecture confers to the resulting materialization its structural and quality characteristics. This constitutes the foundations of the SAME approach.

Coming back to the architecture of Figure 1(a), we have that the *Board* sends change notifications to the *ControlBoard* or the *KnowledgeSources* components. The connectors responsible for these interactions convey design information that influences the implementation alternatives. For instance, *conn1* could be a data-transfer connector that hides data representation through an abstract interface, in order to achieve modifiability. Also, *conn1* could be a control connector to schedule knowledge source according to a schema of priorities, in order to achieve performance. The same connector could even work as a proxy, allowing knowledge sources to interact remotely with the repository, in order to support interoperability with other systems. Based on these observations, let us disaggregate the framelets of Figures 1(b) and 1(c) and consider the two implementations given in Figure 2 for the connector between the *Board* and the *ControlBoard* components.

In the solution of Figure 2(a), the connector is materialized by means of a *Procedure call* implementation that makes the repository know about the type and number of elements that have to be notified. This object-oriented structure does not account for future changes (i.e. modifiability) but behaves well in terms of performance, due to the lack of intermediaries and synchronization in component interactions. The solution of Figure 2(b), in turn, presents an *Event channel* implementation, which promotes modifiability by explicitly decoupling components, at the cost of degrading performance. Note that these two examples also emphasize the features that characterize the protocols of interaction of the connectors (see the left side of Figure 2). Hence, the developer can explicitly influence the achievement of desired qualities by composing connector-based materializations with specific features. This line of reasoning is analogous to that of ArchMatE except that it considered global features as defined by style variants.

In the following section, we explain how SAME takes advantage of connector design blocks (like those of Figure 2) to generate object-oriented implementations for heterogeneous architectures.

3. Using CBR to support materializations of connectors

Developers normally use proven solutions to solve the same classes of design problems, when materializing either styles or connectors. In SAME, connector-based materializations (like those in Figure 2) can be seen as 'patterns' that combine experiences from human experts with general architectural guidelines. In order to build a design assistant for SAME, we need to operationalize the way developers recall and apply patterns of materialization for connectors to new architectures. The CBR paradigm (Kolodner, 1993) naturally fits the requirements of this memory-based design process. CBR is an artificial intelligence technique that emphasizes the role of prior experiences during future problem solving, by reusing and (if necessary) adapting the solutions to similar problems that were solved in the past. In this context, we have developed an Eclipse-based tool for SAME in which the design experiences related to connectors are codified as cases and consequently managed through a CBR engine.

The SAME tool is inspired by the model of assistance proposed for ArchMatE, although relying on a repository of cases (i.e. materialization experiences). A case consists of two sections: architectural problem and object-oriented solution. Initially, the architect specifies her (or his) architecture using a UML notation (Ambler, 2005) and sets her (or his) preferences regarding quality attributes for the materialization. The tool traverses the input architecture and, for each connector, gathers information

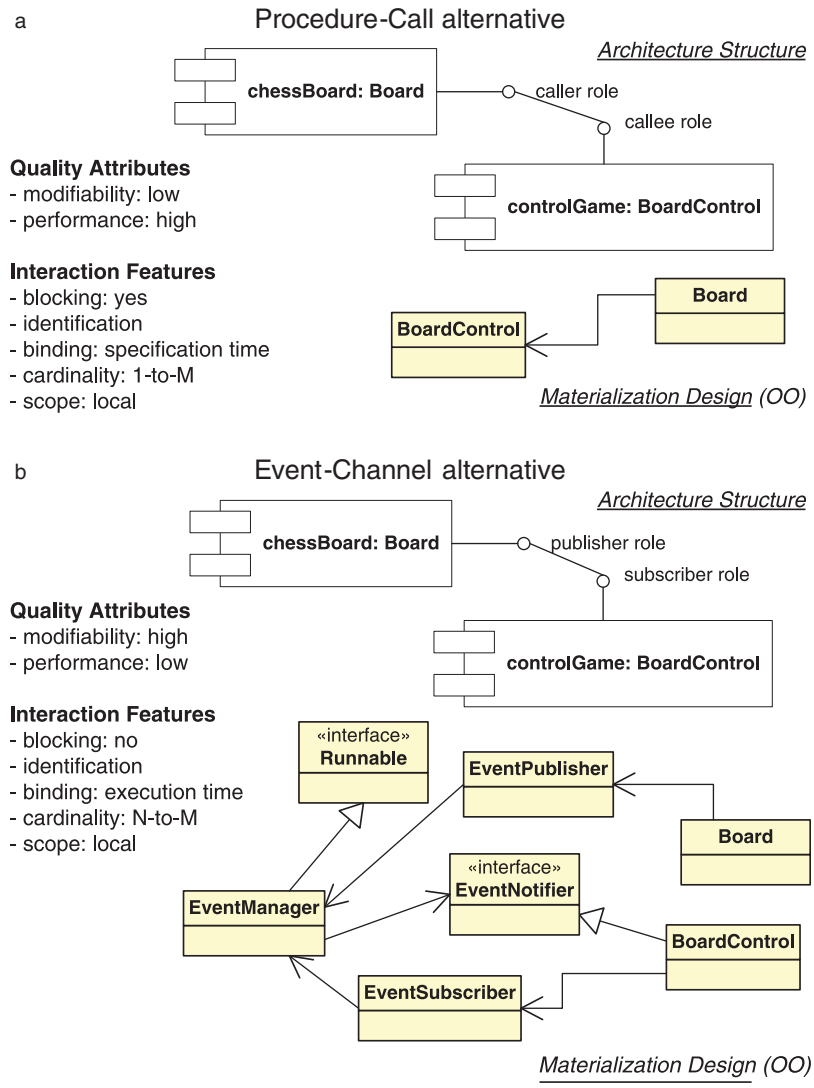

Figure 2: *Two implementation alternatives for a data-transfer connector.*

about its neighborhood. This neighborhood (or connector context) includes the target connector and its attached components. Then, the CBR engine systematically performs two tasks: retrieval and adaptation of cases.

For each connector, the engine retrieves a collection of relevant cases from the repository. A case is said to be relevant if (i) its problem matches the current architectural context, and (ii) its solution describes a useful implementation of the connector's interaction services. Once one or more cases are matched against the input architecture, the CBR engine adapts their object-oriented solutions and maps individual materializations for the connectors. These adapted materializations are put together to generate an object-oriented implementation that reifies the input architecture. Figure 3 shows a snapshot of the SAME tool at work. The user has edited the blackboard architecture for our chess

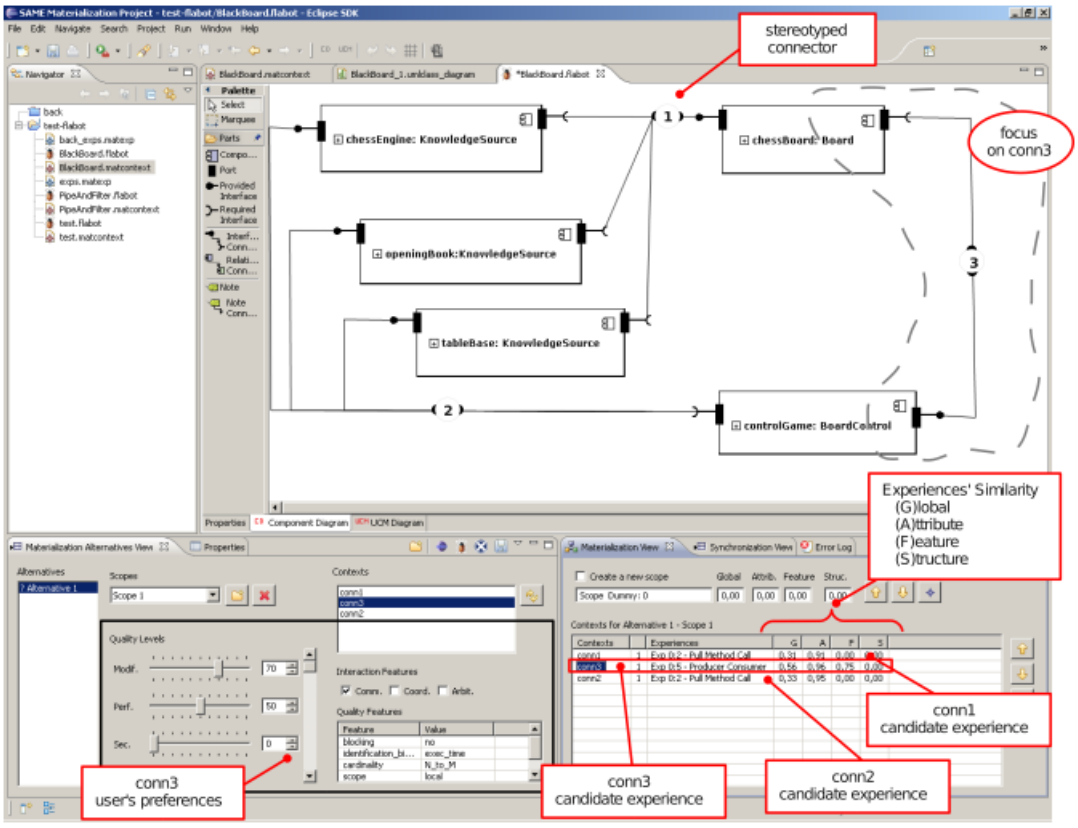

Figure 3: *Editing of input architecture and retrieval of connector contexts in SAME.*

game system in the top-level view (*Component Diagram* panel). The *Materialization Alternatives View* (at the left-bottom corner of the Eclipse perspective) shows the characterization of the connector contexts in the architecture. Each of these contexts (e.g. *conn1*, *conn2*, *conn3* in the *Contexts* combo) represents a problem whose materialization is pending. In the same view, we have sliders to set the quality-attribute levels of the current connector contexts. The values of the sliders are used by the CBR engine to compute the similarity of the cases with respect to the current connector contexts. The engine uses this quality-driven metric as part of the overall computation of case similarity (as we explain in Section 3.2).

The *Materialization View* (at the right-bottom corner of the Eclipse perspective) shows the list of cases retrieved by the CBR engine as candidates to materialize each connector context in the input architecture. At this point, the user gets engaged in an exploratory cycle, assessing the suitability of the solutions proposed by the tool. If a solution meets the user's expectations, the CBR engine will add the new (adapted) case to the repository for future use. Conversely, if the user considers the proposed solution as inadequate, s/he can ask the engine for a different solution. This will trigger the retrieval and adaptation of other cases. In case no satisfactory solution is found, the user could specify her (or his) 'personal' solution as a case, which will enhance the knowledge available in the repository. Let us assume that the CBR engine suggested the case of a *Producer–Consumer* materialization as solution for *conn3*. The snapshot

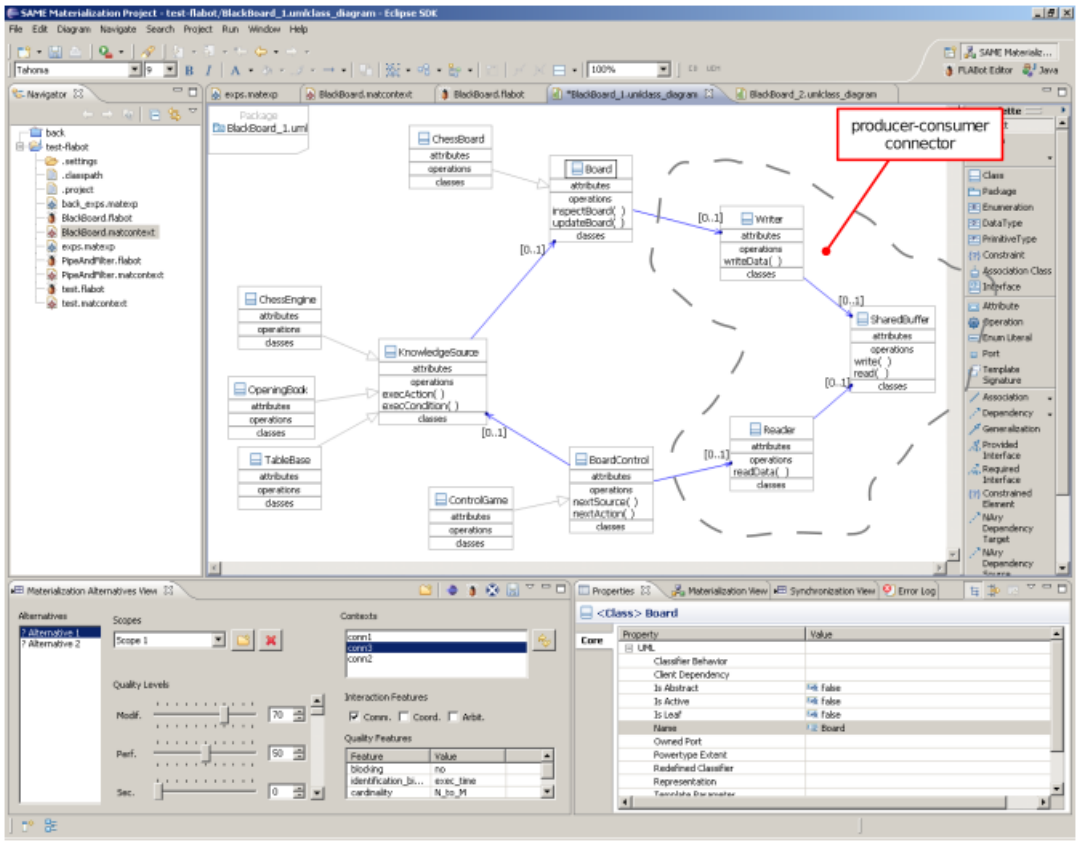

Figure 4: *Outcome of the materialization of the blackboard architecture using a producer–consumer case.*

of Figure 4 depicts a UML class diagram with the resulting implementation after adaptation. The adaptation procedure is supported by the so-called *interaction models*. Further details about adaptation of cases are given in Sections 3.1 and 3.3.

3.1. Case structure

As we mentioned before, a case defines a materialization experience in terms of a problem and a solution section. The problem section serves as the retrieval phase, whereas the solution section serves as the adaptation phase. This situation is exemplified in Figure 5. To retrieve candidate cases, the CBR engine needs to know the degree of similarity between the new design problem and the problems stored in the repository. The comparison of problems is achieved by looking at three dimensions of a connector context, namely the quality dimension, the feature dimension and the structure dimension. For instance, based on the values of these dimensions for the *Board-to-BoardControl* context, Figure 3 shows that the tool ranked the cases in the repository and selected the *Producer–Consumer* case as the most similar problem.[2] In the *Materialization Alternative View*, the quality-attribute levels controllable by sliders represent the *quality dimension* of the current case (*Board-to-BoardControl* context). The numerical scale of the sliders ranges from 0 to 1, where 0 means no fulfillment

[2]*The user might choose a particular case, less similar than the ones proposed by the tool, if s/he considers that case is more valuable for the current context.*

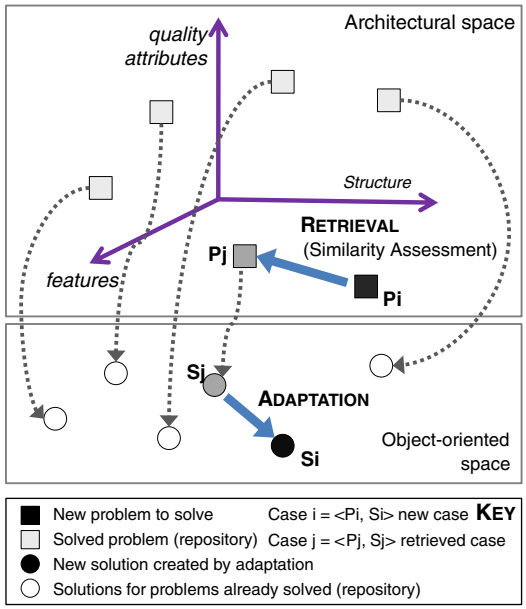

Figure 5: *Role of the problem and solution in the retrieval and adaptation of cases, respectively (adapted from Leake, 1996).*

and 1 represents full satisfaction of a quality attribute.

The *feature dimension* comprises what we call *interaction features*, which characterize the design problem with connector-specific properties. Specifically, the interaction features state how the connector's interaction services are mapped (or need to be mapped) to object-oriented elements. For example, some connector types block components as they interact with others ('blocking' feature), some connectors defer the binding among interacting components until runtime ('identification-binding' feature), or some others allow components to interact remotely ('scope' feature). In Figure 3, the feature dimension of the *Board-to-BoardControl* context characterizes the connector's interaction services with the following features and values: *blocking* = no, *identification binding* = execution time, *cardinality* = 1-to-*M*.

The *structure dimension* specifies the configuration of the components attached to the connector in a particular case. Connectors have interfaces[3] that identify components as participants of the interaction represented by the connector. Connector interfaces may have properties constraining the data flow among components and/or the control flow that moves through the connector implementation. We look at the interaction services of connectors (specified by the data/control properties of connector interface roles) to check whether connectors are compatible or not. Two connectors are compatible if one connector can be replaced by the other and the original connector's interaction services are still provided after the substitution (probably with different levels of quality, though). Connector substitutability allows the search engine to choose, from the repository, among a variety of compatible connectors that might solve a given materialization context.

Table 1 shows problem specifications for five types of connectors. We distilled this information from pattern catalogs, research literature and our personal experience (Buschmann *et al.*, 1996; Shaw, 1996; Fernandez, 1997; Kazman *et al.*, 1997; Mehta *et al.*, 2000; Spitznagel & Garlan, 2003). Developers can define variants of known connectors with subtle differences in their object-oriented materialization. For instance, a variation of the *Event channel* connector shown in Table 1 can be implemented with a blocking event publication mechanism or with a remote access to the publisher and subscriber components.

Coming back to the solution section of a case, it provides an interaction model that constrains the possible object-oriented solutions derivable from the case (assuming a high degree of similarity with the problem section). During the case adaptation, the solution embodies a specific collaboration of objects in such a way that the interaction features defined by a connector at the architectural level are preserved at the object-oriented level. Interaction models are an elaboration of the generic patterns used in ArchMatE. A generic pattern, as developed in

[3] *Also called roles or interface roles in the Architecture Description Language (ADL) jargon.*

Table 1: *Problem information of five types of connectors*

Connector	*Quality dimension*	*Feature dimension*	*Structure dimension*
Procedure call	Modifiability = 0.20 Performance = 0.90 ...	Blocking = yes Binding = compilation time Cardinality = 1-to-1 Scope = local	Caller interface: data-in/data-in Callee interface: data-out/data-out
Data access	Modifiability = 0.20 Performance = 0.90 ...	Blocking = yes Binding = compilation time Cardinality = 1-to-*N* Scope = local	Caller interface: data-in/data-in Callee interface: data-out/data-out
Producer–consumer	Modifiability = 0.20 Performance = 0.90 ...	Blocking = no Binding = execution time Cardinality = *N*-to-*M* Scope = local	Producer interface: data-in Consumer interface: data-out
Event channel	Modifiability = 0.20 Performance = 0.90 ...	Blocking = no Binding = execution time Cardinality = *N*-to-*M* Scope = local	Publisher interface: data-in Subscriber interface: data-out/data-out
Observer	Modifiability = 0.20 Performance = 0.90 ...	Blocking = yes Binding = execution time Cardinality = *M*-to-1 Scope = local	Observer interface: data-in/data-out Observable interface: data-out/data-out

Hammouda and Harsu (2004) and Hautamäki (2005), defines a collection of roles that, once instantiated according to a specific target problem, can generate a concrete object-oriented design. In our work, the roles materialize the interaction services provided by the given connector. Therefore, this mechanism of interaction models makes it feasible to 'generalize' the solutions of prior cases to arbitrary configurations of architectural elements (e.g. different number or types of components around a connector).

3.2. Retrieval and assessment of cases

In order to assess the similarity between cases, the retrieval proceeds as follows. Given a design problem, the CBR engine first looks for a subset of compatible cases. Then, the subset is ordered according to a similarity metric. At last, the engine chooses the case that maximizes that metric.

The similarity metric is a quantitative value ranging from 0 (for dissimilar cases) to 1 (for mostly identical ones). The overall similarity between two cases is computed on the basis of their quality-attribute, feature and structure dimensions. Each case is seen as a point in the problem space, whose coordinates are the values of these three dimensions. The retrieval algorithm applies a weighted sum of the similarities per dimension, where the weights represent the relative contribution of the quality attribute, feature and structure to the global metric. The similarity formula for two cases is as follows:

$$\begin{aligned} &\mathit{GlobalSim}(\text{target}, \text{source}) \\ &\quad = \omega_1 * \mathit{QualitySim}(\text{target}, \text{source}) \\ &\qquad + \omega_2 * \mathit{FeatureSim}(\text{target}, \text{source}) \\ &\qquad + \omega_3 * \mathit{StructureSim}(\text{target}, \text{source}) \end{aligned}$$

The *QualitySim*() function takes the Euclidean distance across the quality-attribute levels of the two cases. Let us consider the *Board-to-BoardControl* context and its quality characterization, as shown in Figure 3, with values *modifiability* = 0.8 and *performance* = 0.5. Let us compare that problem with two cases: a

Producer–Consumer and an *Observer* experience. The *Observer* has high levels of modifiability (0.8) and low performance (0.3). The *Producer–Consumer* experience has more balanced levels of modifiability (0.6) and performance (0.50). The similarity of these *Observer* (OBS) and *Producer–Consumer* (PC) experiences with respect to the quality dimension of the input materialization context (CTX) is computed as follows:

$$\begin{aligned}
&\mathit{QualitySim}(\mathrm{CTX}, \mathrm{OBS}) \\
&\quad = 1.0 - [\mathrm{abs}(\mathrm{CTX}_{\mathrm{modif}} - \mathrm{OBS}_{\mathrm{modif}}) \\
&\qquad + \mathrm{abs}(\mathrm{CTX}_{\mathrm{perf}} - \mathrm{OBS}_{\mathrm{perf}})]/2 \\
&\quad = 1.0 - [\mathrm{abs}(0.70 - 0.80) \\
&\qquad + \mathrm{abs}(0.50 - 0.30)]/2 \\
&\quad = 1.0 - [(0.10 + 0.20)/2] \\
&\quad = 1.0 - (0.30/2) \\
&\quad = 0.85
\end{aligned}$$

$$\begin{aligned}
&\mathit{QualitySim}(\mathrm{CTX}, \mathrm{PC}) \\
&\quad = 1.0 - [\mathrm{abs}(\mathrm{CTX}_{\mathrm{modif}} - \mathrm{PC}_{\mathrm{modif}}) \\
&\qquad + \mathrm{abs}(\mathrm{CTX}_{\mathrm{perf}} - \mathrm{PC}_{\mathrm{perf}})]/2 \\
&\quad = 1.0 - [\mathrm{abs}(0.70 - 0.60) \\
&\qquad + \mathrm{abs}(0.50 - 0.50)]/2 \\
&\quad = 1.0 - [(0.10 + 0.00)/2] \\
&\quad = 1.0 - (0.10/2) \\
&\quad = 0.95
\end{aligned}$$

The *FeatureSim*() function iterates through pairs of interaction features from the assessed cases and determines whether their values match each other. This function computes the feature d imension similarity as the ratio between the number of matching features and the number of evaluated features. The similarity of the previous *Observer* and *Producer–Consumer* experiences with respect to the feature dimension of the input context is given by

$$\begin{aligned}
&\mathit{FeatureSim}(\mathrm{CTX}, \mathrm{OBS}) \\
&\quad = [\mathrm{match}(\mathrm{CTX}_{\mathrm{blocking} = \mathrm{no}}, \mathrm{OBS}_{\mathrm{blocking} = \mathrm{yes}}) \\
&\qquad + \mathrm{match}(\mathrm{CTX}_{\mathrm{iden.bind.} = \mathrm{exec_time}}, \\
&\qquad\quad \mathrm{OBS}_{\mathrm{iden.bind.} = \mathrm{exec_time}}) \\
&\qquad + \mathrm{match}(\mathrm{CTX}_{\mathrm{cardinality} = \mathrm{N\text{-}to\text{-}M}}, \\
&\qquad\quad \mathrm{OBS}_{.\mathrm{cardinality} = \mathrm{N\text{-}to\text{-}M}}) \\
&\qquad + \mathrm{match}(\mathrm{CTX}_{\mathrm{scope} = \mathrm{local}}, \mathrm{OBS}_{\mathrm{scope} = \mathrm{local}})]/4 \\
&\quad = (0 + 1 + 0 + 1)/4 \\
&\quad = 0.50
\end{aligned}$$

$$\begin{aligned}
&\mathit{FeatureSim}(\mathrm{CTX}, \mathrm{PC}) \\
&\quad = [\mathrm{match}(\mathrm{CTX}_{\mathrm{blocking} = \mathrm{no}}, \mathrm{PC}_{\mathrm{blocking} = \mathrm{no}}) \\
&\qquad + \mathrm{match}(\mathrm{CTX}_{\mathrm{iden.bind.} = \mathrm{exec_time}}, \\
&\qquad\quad \mathrm{PC}_{\mathrm{iden.bind.} = \mathrm{exec_time}}) \\
&\qquad + \mathrm{match}(\mathrm{CTX}_{.\mathrm{cardinality} = \mathrm{N\text{-}to\text{-}M}}, \\
&\qquad\quad \mathrm{PC}_{\mathrm{cardinality} = \mathrm{N\text{-}to\text{-}M}}) \\
&\qquad + \mathrm{match}(\mathrm{CTX}_{\mathrm{scope} = \mathrm{local}}, \mathrm{PC}_{\mathrm{scope} = \mathrm{local}})]/4 \\
&\quad = (1 + 1 + 0 + 1)/4 \\
&\quad = 0.75
\end{aligned}$$

Finally, the *StructureSim*() function evaluates the compatibility of the graphs of components and connectors given by the cases. This function measures the ratio between the number of matching components from the assessed cases and the average numbers of components in those cases. That is, two components match each other whenever they belong to the same type and whenever they play compatible roles in the connector they are attached to. Since the *Producer–Consumer* and *Observer* experiences (in our example) are primitive experiences provided by the repository, their architectural structure is not defined in terms of specific component types (i.e. the components are 'untyped'). So, the structural similarity of

these experiences with respect to the input materialization context is 0.

The weights of the similarity formula have been set to predetermined values through experimentation. Currently, these coefficients are $\omega_1 = 0.4$, $\omega_2 = 0.3$ and $\omega_3 = 0.3$. The tool allows the developer to modify the weights in order to vary the result of the retrieval algorithm and get different rankings of cases. Along this line, the global similarity metrics for the *Producer–Consumer* and *Observer* experiences are computed as follows:

$$\begin{aligned}
&\mathit{GlobalSim}(\mathrm{CTX}, \mathrm{OBS}) \\
&\quad = 0.40 * \mathit{QualitySim}(\mathrm{CTX}, \mathrm{OBS}) \\
&\qquad + 0.30 * \mathit{FeatureSim}(\mathrm{CTX}, \mathrm{OBS}) \\
&\qquad + 0.30 * \mathit{StructureSim}(\mathrm{CTX}, \mathrm{OBS}) \\
&\quad = 0.40 * 0.85 + 0.30 * 0.50 \\
&\qquad + 0.30 * 0.00 \\
&\quad = 0.49
\end{aligned}$$

$$\begin{aligned}
&\mathit{GlobalSim}(\mathrm{CTX}, \mathrm{PC}) \\
&\quad = 0.40 * \mathit{QualitySim}(\mathrm{CTX}, \mathrm{PC}) \\
&\qquad + 0.30 * \mathit{FeatureSim}(\mathrm{CTX}, \mathrm{PC}) \\
&\qquad + 0.30 * \mathit{StructureSim}(\mathrm{CTX}, \mathrm{PC}) \\
&\quad = 0.40 * 0.95 + 0.30 * 0.75 \\
&\qquad + 0.30 * 0.00 \\
&\quad = 0.60
\end{aligned}$$

According to the results above, we see that the *Producer–Consumer* experience (similarity = 0.60) is a better match to our target context than the *Observer* experience (similarity = 0.49). Consequently, the former case is chosen for the adaptation phase.

3.3. Adaptation of cases to new problems

The idea behind an interaction model is to reuse solutions when applied to a connector context with a different number and types of components attached to the target connector. To do so, an interaction model specifies a set of *roles* and relationships among them. These roles are instantiated as in conventional generic patterns (Hammouda & Harsu, 2004; Hautamäki, 2005). Basically, a role supports two instantiation modes: (i) binding it to a newly created object-oriented element, or (ii) selecting an existing element in the derived object-oriented design

A special type of role, called *adaptation point*, is used to bridge between the materialization of a connector and the materializations of the concrete components around it. That is, an adaptation point is a placeholder from which the object-oriented elements that correspond to the components attached to a certain connector interface are hooked into the object-oriented elements reifying the interaction services provided by the connector. To clarify this mapping, Figure 6 shows what an interaction model for the *Producer–Consumer* experience looks like. This interaction model was actually applied to derive the materialization of Figure 4. The UML notation extended with stereotypes intends to highlight the relationships among regular roles and adaptation points. The edges denote relationships between roles.

The adaptation algorithm consists of five steps as follows. First, the CBR engine considers the interaction model of the best case selected by the retrieval phase. Second, the engine pulls from that case all the roles that are used to reify the components linked to each adaptation point of the new problem being solved. Third, the engine expands the adaptation points of the interaction model into regular roles ready to be instantiated by concrete elements of the new problem. This expansion is dependent upon the concrete configuration of components around a target connector. Fourth, the engine iterates over all the adaptation points in the interaction model and injects their owned roles (e.g. methods or attributes) into the roles used to materialize the associated components. As the last step, the engine replicates all the relationships that have an adaptation point as source or as destination, and creates new relationships that connect the materialization of the connector with the materialization of its attached

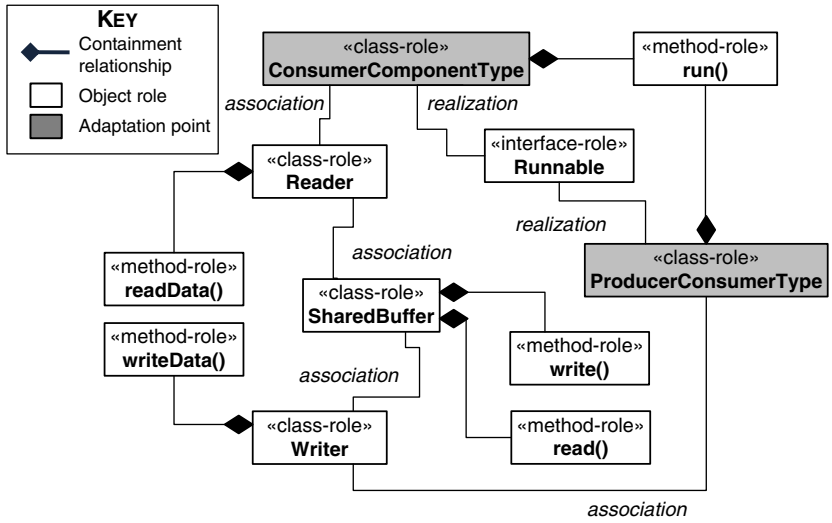

Figure 6: *The producer–consumer interaction model.*

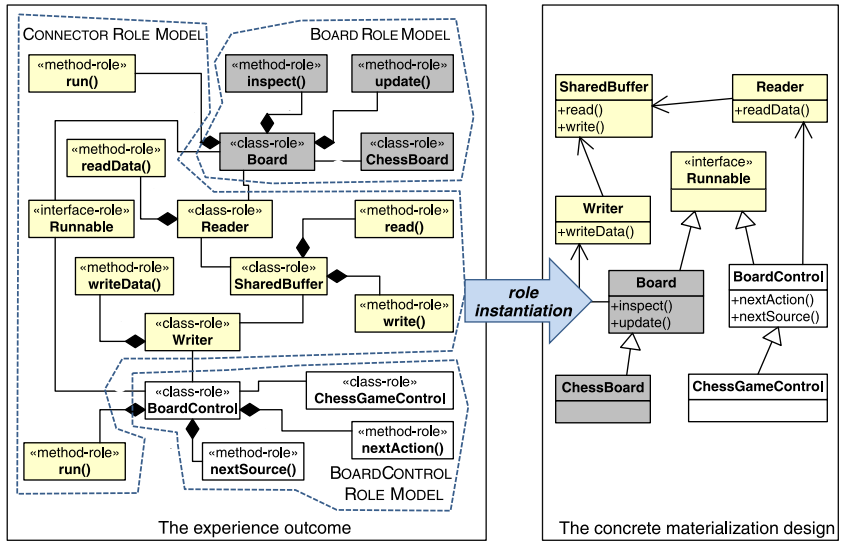

Figure 7: *Adapting the producer–consumer interaction model to the blackboard architecture.*

components. In this way, the adaptation procedure generates the outcome of a materialization experience which unifies the roles that reify the connector's interaction services and the roles reifying the involved components.

Figure 7 depicts how the adaptation procedure expands the adaptation points of the *Producer–Consumer* interaction model shown in Figure 6, generating the new model that serves to derive our pursued object-oriented materialization. The left side of Figure 7 shows the interaction model with its adaptation points expanded according to the architectural configuration involving the *Board* and *BoardControl* components. Note that

the *Board* class role contains the *inspectBoard( )* and the *run( )* method roles. The former method is part of the original materialization of the *Board* component that was pulled from the repository. The latter method, instead, is part of the materialization of the *Producer–Consumer* interaction model and therefore the adaptation procedure injects it into the *Board* class as an owned method. Note also that the realization relationship in that interaction model goes from the *ConsumerCompType* adaptation point to the *Runnable* interface role. This realization relationship is replicated in two relationships, as shown on the right side of Figure 7. This is necessary to make both the *Board* and *BoardControl* class roles implement the *Runnable* interface. Finally, the engine instantiates the outcome of the experience by binding each generic role to a concrete object-oriented element. These bindings lead to an object-oriented materialization of the original connector context, as illustrated in Figure 7. In our example, the role relationships enforced by the *Producer–Consumer* interaction model are satisfied in the derived object-oriented solution, so that the materializations of the components effectively interact via a shared-memory mechanism.

4. Design of SAME

The SAME tool is built on top of the Eclipse platform. The main goals of this tool are (i) manage a repository of architectural design knowledge, and (ii) provide reasoning capabilities to aid developers in the materialization of a software architecture. Figure 8 shows the architecture of SAME, which comprises five modules: the *Architecture Editor*, the *UML Editor*, the *Knowledge Casebase*, the *CBR Engine* and the *Experience Manager*. Figure 8 also shows two types of users: software developer and system manager. The software developer defines the input architecture, and interacts with *CBR Engine* to derive materialization alternatives. The system manager keeps the *Knowledge Casebase* updated and consistent.

The component *Architecture Editor* is an Eclipse plug-in that allows the user (i.e. software developer) to have architectural models. The architectural models are edited using a UML2 notation (Ambler, 2005). That is, the user can define configurations of components and connectors, and then specify information about types of elements, quality-attribute levels and

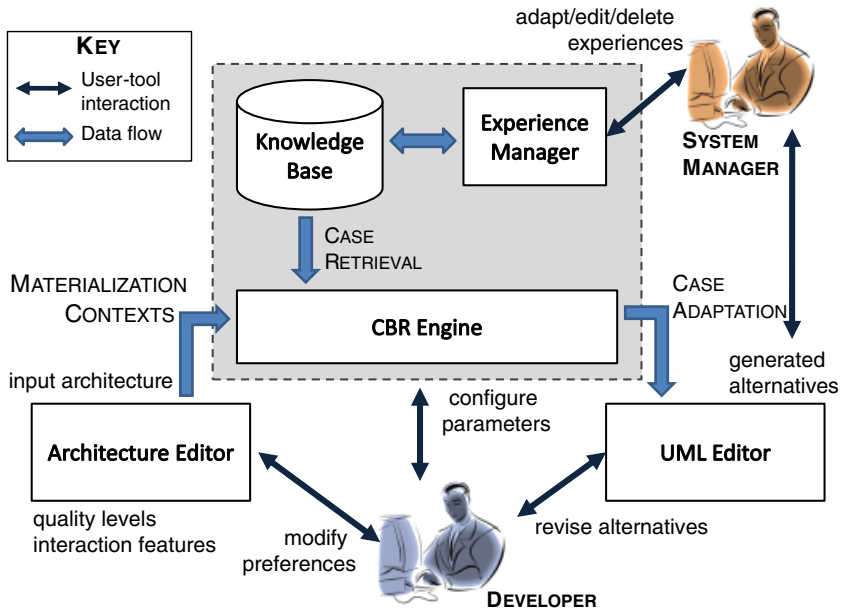

Figure 8: *Implementation of the SAME design assistant.*

inter-component interaction services. The component *UML Editor* is another Eclipse plug-in used to display the object-oriented alternatives with plain UML class diagrams. This editor is also used by the system manager to create interaction models. Both the architecture and the object-oriented models are modeled with the UML2.x meta-model facilities[4] of the Eclipse EMF framework.

The component *CBR Engine* is the core of the SAME tool, because it implements the retrieval and adaptation algorithms discussed in Section 3. *CBR Engine* interacts with *UML Editor* to analyze the structure and connector features of the input architecture. After generating a materialization using the cases stored in *Knowledge Casebase*, *CBR Engine* passes that solution to *UML Editor*. Before using SAME, we need to feed *Knowledge Casebase* with an initial set of experiences. To do so, we have identified a collection of primitive experiences, based on well-known connector types found in the literature (some examples were shown in Table 1). These connectors are procedure call, abstract server, event channel, observer pattern, producer–consumer, remote method invocation and SOAP connectors, broker, among others. The system manager imports primitive experiences into *Knowledge Casebase* by means of the *Experience Manager*.

In the previous section, we noted that end users can draw a personal solution when SAME does not propose a satisfactory alternative for a problem. However, the user's changes are subject to approval by the system manager before they can be committed to the *Knowledge Casebase*. This policy aligns with the 'maintenance phase' in the CBR cycle, and it allows SAME to ensure certain consistency and coherence of the repository of cases.

5. Results and lessons learned

In general, the evaluation of a quality-driven materialization assistant should ensure that the assistant delivers good recommendations. This would require having domain experts evaluate the SAME tool, as well as empirical comparisons of case studies in the literature with the actual tool outputs. In terms of artefacts, the evaluation would include the original architectural model, the quality-attribute requirements that drive the materialization, and a set of object-oriented implementations. Unfortunately, there is no published work about quality-driven design experiences integrating both the software architecture and object-oriented levels of design (see Section 6). For this reason, we decided to rely on architectural styles as a common practice for deriving object-oriented materializations, and we then performed a preliminary evaluation of SAME based on the results available from the ArchMatE work.[5] In this context, we carried out two experiments. In the first experiment, we reproduced the style-based materializations performed with ArchMatE for pipe-and-filter, blackboard and client–server architectures (Díaz-Pace & Campo, 2005). The goal was to analyze the number of alternatives generated by SAME and the characteristics of the style-based and connector-based approaches. In the second experiment, we trained SAME with additional cases, coming from the materialization of systems based on the styles mentioned above. The goal here was to analyze the effects of developer's knowledge in the materialization process. In both experiments, we also checked that the quality-attribute drivers were preserved in the different materializations.

For the first experiment, we atomized the different framelets for the client–server, pipe-and-filter and blackboard styles considered by ArchMatE so as to identify their connector types and model the connector contexts as cases. This led to nine types of connectors and a total of 29 cases, which were stored in the SAME repository. The connectors identified from these seed experiences were pull and push variants of procedure call, abstract server, event channel, observer, broker, producer–consumer and remote procedure call. Afterwards, we entered

[4] *www.eclipse.org/uml2/*

[5] *In the case of ArchMatE, we were lucky to find published case studies about materializations of architectural styles.*

Table 2: *Alternatives generated with ArchMatE and SAME*

Architectural style	*Quality attributes*	*ArchMatE*		*SAME*	
			Precision		*Precision*
Pipe-and-filter	Modifiability, reusability and performance	Six proposed solutions and four relevant	0.66	10 solutions and six relevant	0.6
Blackboard	Modifiability, scalability and performance	10 proposed solutions and seven relevant	0.7	14 proposed solutions and seven relevant	0.5
Client–server	Scalability and performance	12 proposed solutions and seven relevant	0.58	15 proposed solutions and five relevant	0.33

the architectures and initial conditions used in the evaluation of ArchMatE (Díaz-Pace & Campo, 2005), and we exercised SAME by retrieving candidate problems and producing solutions for those architectures. The input architectures and the proposed materializations were presented to four software design experts for an assessment. We asked these experts to evaluate the solutions given by SAME, using a qualitative scale to determine whether the object-oriented designs were in agreement with the architectural problem they were supposed to solve. In addition, we computed the precision of the assistance of the SAME and ArchMatE tools. This precision divides the number of relevant solutions (produced by the tools) by the total number of suggested solutions. The results are summarized in Table 2.

A positive result of this experiment was that SAME generated more materialization alternatives than ArchMatE, although with a little less precision regarding compliance of quality-attribute characteristics. We conjecture that the architectural knowledge (for client–server, pipe-and-filter and blackboard architectures) is partitioned into small design blocks when stored in the SAME repository. Thus, a developer (assisted by SAME) has freedom to assemble a variety of object-oriented solutions out of those blocks. Conversely, the solutions derived by ArchMatE are just instantiations of framelets defined in the ArchMatE repository, and they have few variability points. In the blackboard architecture, for example, we have 10 solutions proposed by ArchMatE against 16 solutions proposed by SAME. However, having many possible combinations of design blocks implies that the resulting materializations should be checked carefully. This can explain the drop in precision of SAME. We also observed a few situations in which SAME generated confusing solutions; for example, it proposed an *Event channel* experience when it should have proposed an *Observer* experience. This means that the case structure (along with the retrieval algorithm) is not always capturing the design intent of the human expert. This problem is less apparent in ArchMatE, because the architectural style chosen for the materialization somehow includes the design intent.

In the second experiment, we used SAME to materialize a heterogeneous system architecture consisting of a blackboard and a client–server style. We actually used our chess system with some additional functionality: 'the system should send every finished chess play to a server acting as a remote repository of plays'. The core functionality was allocated to the blackboard, and the additional functionality was allocated to the client–server style. Before conducting this experiment, we added the solutions generated during the first experiment to the SAME repository. With these settings, we specified combinations of quality-attribute preferences to produce a number of materialization alternatives for the chess system architecture. The SAME tool proposed six solutions, and five of them were judged as relevant by the experts. We observed an improvement in the precision of the CBR engine during the second experiment, due to the

fact that the added cases enriched the repository with application-specific materialization experiences. Actually, the analysis of the object-oriented solutions revealed that the CBR engine used a majority of cases from the first experiment to materialize blackboard-like and client–server-like sub-systems, while using the rest of the cases to derive the glue code for those sub-systems. This 'glue effect' has analogies with the way human experts adapt architectural patterns and implement object-oriented solutions when quality attributes are at work.

In sum, the two experiments allowed us to gain interesting insights about the SAME materialization approach. The claim that reducing the granularity of materialization knowledge (from architectural styles to connectors) improves the assisted generation of materializations was supported by the experiments. The main improvement is the proactive role of the user in the exploration of materializations, either as a developer or as a system maintainer. On one hand, a developer has a high degree of control over the outputs of the tool. For instance, s/he can decide to vary the characterization of the input materialization contexts, and interact with the tool by rejecting the experiences proposed by the engine and selecting alternative ones. On the other hand, the system maintainer can enrich (or refine) the base of materialization knowledge by adding particular instances of quality-attribute materialization strategies. Nonetheless, it should be noticed that the performance of the CBR engine depends on the number and quality of the cases, which required adequate maintenance.

6. Related work

The refinement and implementation of software architectures has been approached in different ways, including development of object-oriented frameworks (Johnson, 1997; Fayad *et al*., 1999; Medvidovic *et al*., 2002), architectural conformance tools (Aldrich *et al*., 2002; Tibermacine *et al*., 2006; Aldrich, 2008) and, more recently, MDE (Schmidt, 2006; Edwards *et al*., 2007).

Application frameworks (Fayad *et al*., 1999) were the first attempts to capture architectural abstractions of related systems, providing reusable object-oriented skeletons for building applications within particular domains. The instantiation of a framework through the standard object-oriented mechanisms (e.g. sub-classing, method overriding etc.) gives possible implementations of the underlying architecture (Johnson, 1997). Along this line, Medvidovic *et al*. (2002) described a family of frameworks implementing the C2 architectural style. C2 is intended for graphical user interfaces, and architectural models are specified using a C2-oriented ADL. The base 'architectural' framework derived from this ADL reifies the main concepts of C2 (e.g. components, connectors, buses) through specific classes and methods. Different versions of the base framework were developed, each of which includes specific optimizations in its design in order to support extra-functional properties such as efficiency, extensibility or distribution. Each framework version works similarly to a framelet in ArchMatE, and therefore the approach has many of the problems discussed in Section 2 and in Díaz-Pace and Campo (2005). In particular, SAME can do better than the C2 framework family, because SAME allows the same architecture (even a C2-based architecture) to switch from one implementation to another with better quality-attribute trade-offs, without the need of having a separate architectural framework supporting each implementation.

Nowadays, many of the concepts and techniques around ADLs (Medvidovic & Taylor, 2000) are being reconsidered under the general umbrella of MDE (Schmidt, 2006). MDE focuses on open standards and supporting tools for system modeling and transformation, usually based on UML profiles. In principle, MDE tools can generate (i.e. materialize) implementations from architectural models, in such a way that conformance between the architectural model and the implementation is given 'by construction'. However, when compared to SAME, current MDE tools still have limitations with respect to design exploration and quality attributes. First, MDE tools usually provide transformational rules that only support 1-to-1 mappings between

architecture and implementation. Second, these rules do not consider quality-attribute concerns, unless they are engineered through specific rule sequences of rules (as implemented by the rule-based engine of ArchMatE). Third, the transformational rules cannot account yet for developers' feedback or design experiences in the middle of the generation process. Anyway, since SAME generates implementations from architectural specifications, it can be seen as an MDE tool in a broad sense.

Conformance approaches such as ArchJava, ArchWare and ACL (Aldrich *et al.*, 2002; Morrison *et al.*, 2004; Tibermacine *et al.*, 2006) have sought to tame the co-evolution of the architecture and its implementation by providing mechanisms to enforce architectural prescriptions (or constraints) on the implementation. Similarly to what happens in SAME, the architectural prescriptions handled by these approaches refer mostly to structural design aspects (e.g. components, class diagrams, modules etc.). Unlike SAME, conformance approaches assume that the developer is responsible for manually specifying the object-oriented implementation of the architecture. So, we see conformance tools as a 'post-materialization check' that can verify architectural violations after the developer has explored a number of implementations using SAME.

ArchJava (Aldrich *et al.*, 2002; Aldrich, 2008) extends Java with constructs such as components, ports and connections, in order to enforce 'communication integrity' of function calls between components. ArchJava can be seen as a component-based materialization approach that allows developers to embed architectural principles in object-oriented implementations. Note that this approach does not have tool support for the generation of ArchJava implementations. Furthermore, quality-attribute issues are out of the scope of the ArchJava language. Another conformance approach that combines architectural constraints with model transformations is outlined in Tibermacine *et al.* (2006). Unlike ArchJava, Tibermacine's approach includes a tool that reminds the developer about possible loss of quality-attribute properties during evolution, akin to regression tests. The links between quality-attribute issues and architectural decisions is documented using contracts, which help to make the design rationale explicit and checkable. A special language called ACL based on the UML Object Constraint Language (OCL) is proposed for these contracts. Basically, a contract involves two parties: the developer of the base architecture who guarantees some quality-attribute scenarios; and the developer of a refined version of the architecture (e.g. an object-oriented design) who should respect the rules established by the former. The tool is able to point out inconsistencies in terms of constraints that are not satisfied by the current architecture. This tool approach differs from SAME in two aspects. First, ACL defines constraints for design models while SAME does not (the constraints are implicit in the interaction models for connectors). Second, the constraint-based approach is not concerned with the exploration of materializations as SAME is. As long as the ACL constraints are not violated, the refinement of the architecture and the exploration are left to the developer. We think that ArchJava or ACL can work as good complements for SAME, by formalizing the context for materialization experiences.

Regarding connectors, Allen and Garlan (1997) performed a formal study of the protocols of interaction between architectural components and several characterizations of architectural connectors have been reported since then. For example, Kazman *et al.* (1997) proposed a feature-based taxonomy of architectural elements – either components or connectors – commonly used in architecture composition; Mehta *et al.* (2000) classified common connector types according to the interaction services they provide; Fernandez (1997) described an exhaustive set of connectors used in the development of real-time systems. Also, Spitznagel and Garlan (2003) defined a set of high-level transformations that support compositional reasoning about connectors. The knowledge about connectors compiled by all these approaches constituted the basis for the development of SAME.

A connector-based materialization approach close in spirit to SAME is proposed by Heuzeroth

et al. (2001). The authors describe the so-called 'grey-box connectors' as static meta-programs for component interactions, which are implemented with aspect-oriented technology. The aspects actually provide the glue code to adapt component interactions and to compose together heterogeneous components. The main issue is that the aspects are directly applied to the object-oriented code, so neither architectural design aspects nor quality attributes are considered in this approach. We believe that the aspects can be captured as new types of interaction models for SAME.

On the other hand, there have been a few applications of CBR techniques to support reuse of object-oriented models. In Fouqué and Matwin (1993), an experience with a CBR tool called Caesar to reuse source code in Smalltalk is reported. This tool is able to locate program elements (e.g. classes) based on the functionality they realize, extending the implementation of these elements according to the particular requirements of the problem. The cases include a lexical specification of the element functionality, its implementation idiom and other code properties, which justify the use of the element in a particular context. ReBuilderUML (Gomes & Leitão, 2006) is another CBR tool to reuse fragments of UML class diagrams when addressing functional requirements. This approach relies on a domain-specific ontology to analyze the system requirements, and then searches in a repository of class diagrams for a case that is suitable for those requirements. As in SAME, the use of CBR techniques provides benefits for both novice and experienced developers. Inexperienced developers can profit from object-oriented patterns and avoid mistakes made in previous systems, while experienced developers can be assisted in the exploration of alternative designs. Furthermore, the two approaches support the construction of composite designs out of small design experiences. Nonetheless, the main drawbacks of these two approaches compared to SAME are that they can only deal with low-level object-oriented designs, and they can only derive implementations from functional requirements. Thus, the variability of object-oriented solutions imposed by quality-attribute requirements is not addressed.

7. Conclusions and future work

In this work, we have addressed the derivation of object-oriented implementations for software architectures from a connector-based perspective. In order to facilitate the exploration of conformant implementations, we have proposed a design assistant, called SAME, that relies on the CBR paradigm as computational support. Materialization strategies are modeled in terms of the problem/solution structure of the cases; thus, new strategies can be progressively added to the knowledge base. The SAME assistant is not intended to generate a running implementation of a system architecture; rather it guides developers in the construction of a blueprint from which they can start deriving the final implementation of the system. This approach represents a step towards a more systematic linkage between the architecture and the object-oriented worlds through the codification of design experiences.

As in ArchMatE, the quality-attribute focus of SAME contributes to keeping the conceptual integrity of the system (Bass *et al.*, 2003) during object-oriented design. Furthermore, from our preliminary evaluation results, we argue that SAME is more flexible than other tool-supported materialization approaches (including ArchMatE) with respect to managing quality-attribute knowledge. Nonetheless, more empirical studies are still needed. Recently, Sangwan *et al.* (2008) have discussed architectural and object-oriented design approaches based on an interesting case study. Pre-processing is necessary for this case study to fit the connector-based materialization format, so as to compare the results with those of SAME. A pre-processing of the case study is currently in progress as part of a graduate student project at UNICEN University.

A perceived limitation of the approach is that the developer can judge the solutions produced by SAME in a subjective way. Thus, the results may have variations from one developer to another. Here, it is desirable to automate the

quality-attribute criteria to assess the design. Along this line, we are investigating whether quality-attribute analysis models (Bass *et al.*, 2003) can be used to quantify some degree of conformance between architecture and object-oriented models. In addition, we are working on modeling the behavioral aspects of architectural designs, so that the SAME tool can complement the generated class diagrams with object interaction diagrams.

Finally, we believe that further developments of SAME should enable the construction of a 'design memory' for software organizations implementing architecture-centric practices. In that sense, SAME provides a shared repository of proven pattern-based solutions which amalgamates the design knowledge spread among the developers in the organization. Hence, we expect that the principles behind SAME will help organizations to standardize the transition from software architecture to object-oriented designs, reducing the complexity of the architectural design process.

Acknowledgments

The authors would like to thank the anonymous reviewers for their feedback, which contributed to improving the quality of the paper.

References

ALDRICH, J. (2008) Using types to enforce architectural structure, in *Working International Conference on Software Architecture*, Washington, DC: IEEE Computer Society.

ALDRICH, J., C. CHAMBERS and D. NOTKIN (2002) Architectural reasoning, in *ArchJava in ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, New York: Springer, 334–367.

ALLEN, R. and D. GARLAN (1997) A formal basis for architectural connection, *ACM Transactions on Software Engineering and Methodology*, **3** (6), 213–249.

AMBLER, S. (2005) *The Elements of UML 2.0 Style*, Cambridge: Cambridge University Press.

BASS, L.J., M. KLEIN and F. BACHMANN (2002) Quality attribute design primitives and the attribute driven design method, in *PFE '01: Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, New York: Springer, 169–186.

BASS, L., P. CLEMENTS and R. KAZMAN (2003) *Software Architecture in Practice*, 2nd edn, Boston, MA: Addison-Wesley.

BUSCHMANN, F., R. MEUNIER, H. ROHNERT, P. SOMMERLAND and M. STAL (1996) *Pattern-Oriented Software Architecture: A System of Patterns*, Hoboken, NJ: Wiley.

CAMPO, M., J.A. DÍAZ-PACE and M. ZITO (2002) Developing object-oriented enterprise quality frameworks using proto-frameworks, *Software: Practice and Experience*, **32** (8), 837–843.

CHOPPY, C., D. HATEBUR and M. HEISEL (2005) Architectural patterns for problem frames, *IEE Proceedings – Software*, **152** (4), 198–208.

DÍAZ-PACE, J.A. and M. CAMPO (2005) ArchMatE: from architectural styles to object-oriented models through exploratory tool support, in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, New York: ACM Press, 117–132.

EDWARDS, G., S. MALEK and N. MEDVIDOVIC (2007) Scenario-driven dynamic analysis of distributed architectures, in *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering (FASE)*, LNCS, Heidelberg: Springer, 125–139.

FAYAD, M., D. SCHMIDT and R. JOHNSON (1999) *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, Hoboken, NJ: Wiley.

FERNANDEZ, J.L. (1997) A taxonomy of coordination mechanisms used by real-time processes, *Ada Letters*, **XVII**, 29–54.

FOUQUÉ, G. and S. MATWIN (1993) A case-based approach to software reuse, *Journal of Intelligent Information Systems*, **2**, 165–197.

GOMES, P. and A. LEITÃO (2006) A tool for management and reuse of software design knowledge, in *Proceedings of the EKAW 2006 – 15th International Conference on Knowledge Engineering and Knowledge Management*, Lecture Notes in Computer Science 4248, Berlin: Springer.

HAMMOUDA, I. and M. HARSU (2004) Documenting maintenance tasks using maintenance patterns, in *Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR 2004), Tampere*, Washington, DC: IEEE Computer Society, 37–47.

HAUTAMÄKI, J. (2005) Pattern-based tool support for frameworks: towards architecture-oriented software development environment, PhD Thesis, Tampere University of Technology, Publication 521.

HEUZEROTH, D., W. LÖWE, A. LUDWIG and U. AßMANN (2001) Aspect-oriented configuration and adaptation of component communication, in *Generative and Component-Based Software Engineering: Third International Conference, GCSE*

2001, Lecture Notes in Computer Science, Berlin: Springer, 58.

JOHNSON, R.E. (1997) Components, frameworks and patterns, in *Proceedings of the 1997 Symposium on Software Reusability*, M. Harandi, ed., New York: ACM Press, 10–17.

KAZMAN, R., P.C. CLEMENTS, L. BASS and G.D. ABOWD (1997) Classifying architectural elements as a foundation for mechanism matching, in *Proceedings of the 21st International Computer Software and Applications Conference COMPSAC*, Washington, DC: IEEE Computer Society, 14–17.

KOLODNER, J. (1993) *Case-Based Reasoning*, San Francisco, CA: Morgan Kaufmann.

LEAKE, D.B. (1996) *Case-Based Reasoning: Experiences, Lessons and Future Directions*, Cambridge, MA: MIT Press.

LOPEZ DE MANTARAS, R., D. MCSHERRY, D. BRIDGE, D. LEAKE, B. SMYTH, S. CRAW, B. FALTINGS, M.L. MAHER, M.T. COX, K. FORBUS, M. KEANE, A. AAMODT and I. WATSON (2005) Retrieval, reuse, revision and retention in case-based reasoning, *Knowledge Engineering Review*, **20** (3), 215–240.

MATTMANN, C., D. WOOLLARD, N. MEDVIDOVIC and R. MAHJOURIAN (2007) Software connector classification and selection for data-intensive systems, in *Proceedings of the 2nd International Workshop on Incorporating COTS Software into Software Systems: Tools and Techniques (IWICSS'07)*, Washington, DC: IEEE Computer Society, 4–9.

MEDVIDOVIC, N. and R. TAYLOR (2000) A classification and comparison framework for software architecture description languages, *IEEE Transactions on Software Engineering*, **26** (1), 70–93.

MEDVIDOVIC, N., N. MEHTA and M. MIKIC-RAKIC (2002) A family of software architecture implementation frameworks, in *Proceedings of the 3rd IEEE/IFIP Working Conference on Software Architecture*, Deventer: Kluwer, 221–235.

MEHTA, N.R., N. MEDVIDOVIC and S. PHADKE (2000) Towards a taxonomy of software connectors, in *Proceedings of the 22nd International Conference on Software Engineering, ICSE '00*, New York: ACM Press, 178–187.

MORRISON, R., G. KIRBY, D. BALASUBRAMANIAM, K. MICKAN, F. OQUENDO, S. CIMPAN, B. WARBOYS, B. SNOWDON and R. GREENWOOD (2004) Support for evolving software architectures in the ArchWare ADL, in *Proceedings of the IEEE/IFIP Working Conference on Software Architecture (WICSA'04)*, Washington, DC: IEEE Computer Society.

PREE, W. and K. KOSKIMIES (2000) Framelets – small is beautiful, in *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, M.E. Fayad, D.C. Schmidt and R.E. Johnson (eds), Hoboken, NJ: Wiley.

RAPANOTTI, L., J. HALL, M. JACKSON and B. NUSEIBEH (2004) Architecture-driven problem decomposition, in *Proceedings of the 12th IEEE International Conference on Requirements Engineering*, Washington, DC: IEEE Computer Society, 80–89.

SANGWAN, R., C. NEILL, Z. EL-HOUDA and M. BASS (2008) Integrating software architecture-centric methods into object-oriented analysis and design, *Journal of Systems and Software*, **81** (5), 727–746.

SCHMIDT, E. (2006) Model-driven engineering, *IEEE Computer*, **39** (2), 25–31.

SHAW, M. (1996) Procedure calls are the assembly language of software interconnection: connectors deserve first-class status, in *ICSE '93: Selected Papers from the Workshop on Studies of Software Design*, New York: Springer, 17–32.

SPITZNAGEL, B. and D. GARLAN (2003) A compositional formalization of connector wrappers, in *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, Washington, DC: IEEE Computer Society, 374–384.

TIBERMACINE, C., R. FLEURQUIN and S. SADOU (2006) On-demand quality-oriented assistance in component-based software evolution, in *Proceedings of the 9th ACM SIGSOFT International Symposium on Component-Based Software Engineering (CBSE'06)*, LNCS 4063, New York: Springer, 294–309.

TRACZ, W. (1995) DSSA (Domain-Specific Software Architecture): pedagogical example, *SIGSOFT Software Engineering Notes*, **20** (3), 49–62.

The authors

German L. Vazquez

German L. Vazquez is a PhD student at the ISISTAN Research Institute, UNICEN University (Tandil, Argentina). He has several years of industrial experience in software development, and has been involved in technology transfer projects at UNICEN University. As a practitioner, Eng. Vazquez is currently applying his software architecture experience to the digital media industry with focus on video processing applications over cloud computing environments. His main interests are object-oriented frameworks, multi-agent systems, quality-driven architectural design and cloud computing. He can be contacted at gvazquez@exa.unicen.edu.ar or at gvazquez@mpoint.net.

J. Andres Díaz-Pace

J. Andres Díaz-Pace is a member of the technical staff at the Software Engineering Institute (SEI), where he works in the Research, Technology and System Solutions Program. His primary research interests are quality-driven architecture design, artificial intelligence techniques in design, architecture-based evolution and conformance. Currently, he works in the development of tool support for the exploration of architectural design alternatives. He has authored several publications on topics of design assistance and object-oriented frameworks. Prior to joining the SEI, Dr Díaz-Pace was a professor and researcher at UNICEN University (Tandil, Argentina) for more than 10 years. He also participated as architect in several technology transfer projects at UNICEN University. Dr Díaz-Pace received a PhD in computer science from UNICEN University in 2004. He can be contacted at adiaz@sei.cmu.edu.

Marcelo R. Campo

Marcelo R. Campo is a professor with the Computer Science Department and head of the ISISTAN Research Institute, both at UNICEN University (Tandil, Argentina). He received his PhD degree in computer science from Universidade Federal de Rio Grande do Sul (Porto Alegre, Brazil) in 1997. His research interests include intelligent tools for software engineering, software architectures and frameworks, agent technology and software visualization. Dr Campo has had several papers published in conferences and journals about software engineering topics. He is also a Research Fellow of the National Council for Scientific and Technical Research of Argentina (CONICET). He can be contacted at mcampo @exa.unicen.edu.ar.