The Journal of Systems and Software 83 (2010) 2513–2527

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Enhancing middleware support for architecture-based development through compositional weaving of styles

Sam Malek*, Harshini Ramnath Krishnan, Jayalakshmi Srinivasan

Department of Computer Science, George Mason University, 4400 University Drive, MS 4A5, Fairfax, VA 22030, United States

ARTICLE INFO

Article history:
Received 14 April 2010
Received in revised form 21 July 2010
Accepted 22 July 2010
Available online 30 July 2010

Keywords:
Software architecture
Architectural style
Aspect-oriented programming
Middleware

ABSTRACT

Architecture-based software development has been shown as an effective approach for managing the implementation complexity of large-scale software systems. Architecture-based development is often achieved with the help of a middleware, which provides implementation-level counterparts for the architectural modeling constructs. Such a middleware automatically ensures that implemented system accurately embodies the properties encoded in its architectural models. However, existing middlewares do not provide sufficient support for architectural styles. This is due to the crosscutting structure of styles that impacts the behavior of every other architectural construct, and hence the corresponding middleware facilities. We present an aspect-oriented approach that alleviates this problem by weaving the stylistic concerns with the rest of the middleware. The approach decouples stylistic concerns from other middleware facilities, which in turn improves the middleware's understandability and flexibility, and enables rapid composition of hybrid styles. We evaluate the approach and describe our experiences by providing support for several well-known styles using two open-source middleware platforms.

1. Introduction

Software engineering researchers and practitioners have successfully dealt with the increasing complexity of software systems by employing the principles of software architecture. *Software architectures* provide design-level models and guidelines for composing software systems in terms of components (computational elements), connectors (interaction elements), and their configurations (also referred to as topologies) (Shaw and Garlan, 1996). Software *architectural styles* (e.g., Publish–Subscribe, Client–Server, Pipe–Filter) further codify structural, behavioral, interaction, and composition guidelines that are likely to result in software systems with desired properties (Fielding, 2000; Shaw and Garlan, 1996).

For the software architectural models and guidelines to be truly useful in a development setting, they must be accompanied by support for their implementation (Shaw et al., 1995). However, there is a gap between the high-level architectural concepts and the low-level programming language constructs that are used for the implementation. This gap requires engineers to maintain a (potentially complex) mental map between components, connectors, communication ports, events, etc. on the one hand, and classes, objects, shared variables, pointers, etc. on the other hand.

A more effective approach for architecture-based software development is to leverage a middleware solution that provides native implementation-level support for the architectural concepts. Such a middleware platform automatically reduces the possibility of *architectural drift* and *erosion* (Perry and Wolf, 1992). It alleviates the developers from resorting to manual techniques for verifying the fidelity of the implemented system with respect to the architectural models.

Unfortunately, the state-of-the-art middlewares provide implementation support for some architectural concepts (e.g., components, ports, events), but not adequate support for others (Malek et al., 2007), such as explicit connectors (Mehta et al., 2000), which are usually distributed across the different implementation-level modules as combinations of method calls, shared memory, network sockets, and other facilities provided by the middleware. Also, modern commercial middleware platforms do not fully support architectural styles. They ignore, mimic, or at best assume a particular style. In turn, instead of streamlining the architecture-based development of software systems, they form an obstacle—the software architect is forced to choose a style that is best supported by a given middleware platform, as opposed to a style that suits the requirements of a particular software system.

The lack of support for architectural styles stems from the difficulties associated with their implementation. Often architectural styles prescribe rules and constraints that impact (crosscut) the behavior and structure of all the other architectural constructs, and hence the corresponding middleware facilities (Malek, 2008a). This in turn makes it difficult to realize the stylistic concerns in the

* Corresponding author.
E-mail addresses: smalek@gmu.edu (S. Malek), hramnath@gmu.edu (H. Ramnath Krishnan), jsriniva@gmu.edu (J. Srinivasan).

0164-1212/$ – see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2010.07.050

traditional Object-Oriented Programming (OOP) paradigm (Malek, 2008b). At the same time, there is a lack of consensus among the software engineers on the exact specification and semantic of many well-known architectural styles, forcing the middleware designer to realize an interpretation of a style that is not acceptable by others. Finally, there are many architectural styles that an architect may opt to use, making it impossible for the middleware designer to provide support for all of them.

We present an approach for implementing architectural styles that is based on the *Aspect-Oriented Programming* (*AOP*) paradigm (Kiczales et al., 1997). We provide an overview of our approach on top of an existing middleware platform, called Prism-MW (Malek et al., 2005). Our approach allows for modularized implementation of stylistic concerns, which are weaved into the middleware's core facilities to generate style specific versions of the middleware.

Separation of the stylistic concerns from the middleware core facilities has several benefits, including the ability to hierarchically compose hybrid styles by reusing the existing styles, and the flexibility of changing a system's architectural style without impacting the rest of the system. Our approach shifts the responsibility of making stylistic decisions from the middleware designer to the software engineer. It allows the engineer to implement support for a new, potentially domain specific, style in a given middleware platform.

We also provide an overview of our experience with the development for supporting several styles in two middleware platforms, and their application in the context of a real-world software system. Our experience corroborates that by providing extensive support for styles in middleware, it is possible to automatically ensure that the implemented system accurately embodies the properties encoded in its architectural models.

The remainder of the paper is organized as follows. Section 2 motivates the research. Section 3 describes our overall approach. Section 4 provides an overview of Prism-MW. Section 5 demonstrates the crosscutting structure of styles using Prism-MW. Section 6 details support for several styles on top of Prism-MW using the proposed methodology. Section 7 shows how the resulting middleware is used. Section 8 describes the composition of hybrid styles. Section 9 evaluates the approach. Finally, the paper concludes with a summary of the related work and avenues of future research.

2. Problems and challenges

Ideally, when fronted with the complexity of designing a large-scale software system, the architect should have the flexibility to construct a software architecture that best satisfies the system's requirements, irrespective of the candidate middlewares that could be used for the implementation. However, as observed by other researchers, including Gorton (2006) and Medvidovic et al. (2003), this is rarely the case. The ability to go from the system's architectural models to their implementation is consistent with the Object Management Group's Model-Driven Architecture methodology that consists of two phases: Platform Independent Model (PIM) and Platform Specific Model (PSM) (MDA). Unfortunately, deriving PSM from PIM is often extremely challenging and in some cases not even feasible (Jean et al., 2002). We believe the existing middlewares' rudimentary support for styles to be one of the key culprits in complicating the transformation of PIM to PSM. As a result, in practice, the engineers are constrained by a subset of architectural choices, including styles, effectively supported by available middleware platforms.

In general, existing middleware solutions either do not support styles at all or fall into one of the following two categories:

- *Style supposition*: The majority of middleware platforms are developed with certain intrinsic assumptions about the structural and behavioral characteristics of the applications that could be deployed on top of them. These presumed characteristics determine the style of systems that could be effectively deployed on top of such platforms. For instance, Java RMI and CORBA make certain assumptions on the nature of interaction among software components (e.g., remote method invocation) that make them suitable for the development of Client–Server, but not Publish–Subscribe systems.
- *Mimicking styles*: Even the middlewares that presuppose styles (e.g., Java RMI, CORBA, and DCOM) lack a sophisticated support for them. They provide support for some aspects of architectural styles, such as the *communication style* (e.g., synchronous versus asynchronous), but do not support others. For instance, consider the fact that stylistic rules and constraints on the valid configurations of a software system are rarely ensured and enforced by these middlewares. In turn, making it extremely difficult to verify the fidelity of the constructed software system with respect to its intended architecture. As another example, consider the fact that the majority of middlewares do not provide explicit support for architecture-level connectors, which are usually distributed across different implementation-level modules as combinations of method calls, shared memory, network sockets, and other facilities in the middleware (Mehta et al., 2000). As a result, such middlewares cannot provide effective support for the numerous styles (e.g., C2, Taylor et al., 1996; Pipe–Filter, Shaw and Garlan, 1996) that rely on the existence of explicit connectors.

The current shortcomings stem from challenges the developers of middleware systems face in realizing support for styles:

- *Fragmented implementation*: Architectural styles often prescribe rules and constraints that impact the behavior and structure of all the other architectural constructs. Therefore, unlike any other architectural concept, architectural styles cannot be effectively abstracted and implemented using the traditional Object-Oriented Programming constructs. In fact, if a middleware provides support for the stylistic concerns, they are often implemented as dispersed code snippets, and thus lost in the final product (Malek et al., 2005).
- *Lack of consensus*: As acknowledged by Fielding (2000), there is a lack of consensus among software engineers on the exact specification of some commonly used architectural styles. For instance, consider the Client–Server style, which is one of the most widely used styles in distributed systems. There is no general agreement in the software engineer community on whether the Client behavior of blocking the thread of execution while a request is processed forms an essential characteristic of this style or not (Fielding, 2000). As a result, middleware developers provide support for some of the conventional and commonly accepted aspects of a style, and ignore the others.
- *Many styles*: An approach pioneered in our previous work (Malek et al., 2005) provides support for multiple styles in a middleware by parameterizing it, such that it can be configured to behave according to the rules of a style at boot up time. However, we argue that this is not a viable solution for a general-purpose middleware. There are many well-known styles and even more domain specific and hybrid styles that an architect may opt to use. Predicting all of those a priori (i.e., during the construction of the middleware) is infeasible. Furthermore, providing implementation support for any substantive set of styles inevitably makes the middleware bulky and hinders its performance.

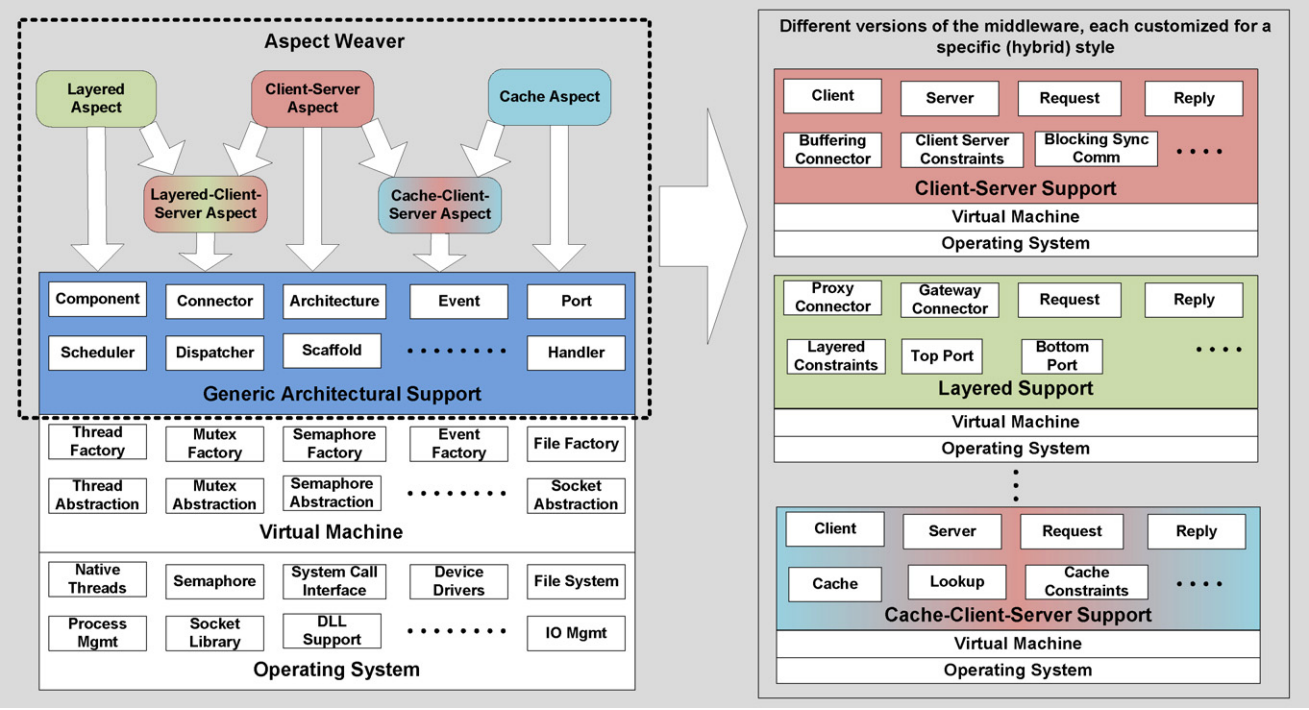


Fig. 1. Injecting support for styles into middleware.

3. Injecting architectural style

The approach presented in this paper builds on our earlier work (Malek, 2008a), which as detailed below has been extended to accommodate composition of multiple architectural styles. Fig. 1 shows an overview of our approach. A typical structure of the middleware stack is shown on the left side of the figure. We have distinguished between two types of facilities that a middleware may provide on top of the Operating System: at the bottom is a *virtual machine layer* that allows the middleware to be deployed in heterogeneous settings; the abstraction facilities provided by the virtual machine are leveraged by the middleware's *architectural constructs* that lay on top of it.

In this research, we are interested in the top layer, i.e., *architectural support* layer. As noted earlier, the level of architectural support provided by middlewares vary. In this figure, we are depicting the typical facilities an *architectural middleware*, which provides extensive support for architecture-based software development, may provide. We provide a detailed overview of an architectural middleware in the next section.

In our approach, we assume that the architectural facilities provided by the middleware are generic, i.e., they are not stylistically constrained. This assumption does not impede the applicability of our approach, since as you may recall from the previous section most middlewares do not provide sophisticated support for architectural styles. We implement the stylistic concerns in one or more aspects, which when weaved with the middleware's generic constructs result in a middleware that supports the implementation of one or more styles.

For example, as depicted in Fig. 1, to provide support for the Client–Server style, the *Client–Server Aspect* is weaved with the middleware to generate a specialized version of the middleware with typed components (*Clients*, *Servers*), typed events (*request*, *reply*), modified component behavior (*Client* blocks after making a request), new connector functionality (server connector buffers incoming requests), and strict configuration rules (e.g., preventing a *Client* from connecting and making requests to other *Clients*). Similarly, the *Layered Aspect* and *Cache Aspect* enable the style specific characteristics of the Layered and Cache styles, respectively.

The contributions of our approach can be summarized as follows:

- *Locality*: All the code that implements a style is defined in an aspect, and not scattered in the middleware's core facilities. The middleware facilities are free of the style concerns, and as a result there is no style specific coupling. The locality of stylistic concerns has three advantages: (1) aids the system understandability, and hence improves the middleware's maintenance; (2) allows a user of the middleware to refine the realization of a given style, if the user has a different interpretation of that style; and (3) enables a user of the middleware to develop support for new domain-specific styles.
- *Compositionality*: Several basic style aspects can be weaved together to provide support for hybrid styles. A hybrid style is an architectural style that inherits the properties of two or more parent styles. An example of a hybrid style is the Cache–Client–Server that extends the Cache and Client–Server styles (shown in Fig. 1).
- *(Un)Pluggability*: Since the middleware facilities are not aware of their role in architectural styles (i.e., there is no dependency from the middleware to the style aspects), it is possible to change the style of an application, potentially at run-time via dynamic AOP, without impacting either the application or the middleware facilities.
- *Automatic architectural conformance*: A generated style specific version of the middleware constrains the application developer to a subset of implementation choices allowable in that style. As a result, it automatically prevents implementation choices that could result in architectural drift and erosion.

In the remainder of this paper, we describe the details of the approach using an existing middleware platform and three well-known styles, which are composed to support several hybrid styles.

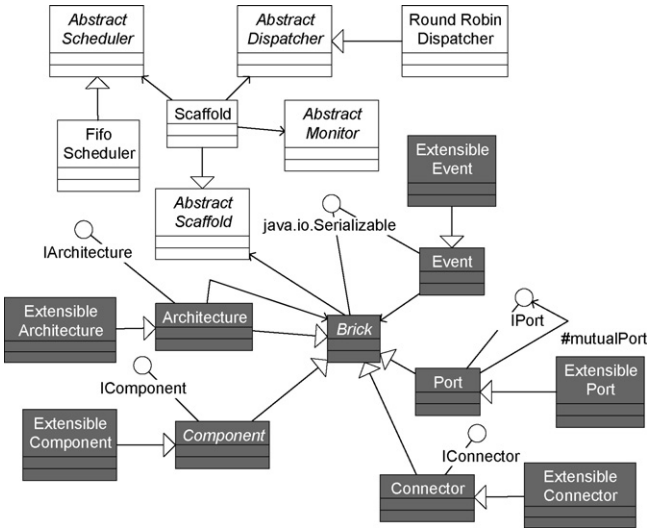


Fig. 2. Abridged UML class design view of Prism-MW. Classes impacted by style are in gray.

4. An illustrative middleware

In this section, we provide an overview of Prism-MW (Malek et al., 2005), an architectural middleware platform that we have used extensively in this research. Prism-MW is an architectural middleware platform that provides implementation-level support for architectural constructs in an extensible, efficient, and scalable manner (Malek et al., 2005). Prism-MW is a suitable platform for describing and applying our approach: (1) Prism-MW supports straightforward one-to-one mapping of architectural constructs to their implementations, which makes it an ideal platform for demonstrating the crosscutting impact of styles; and (2) Prism-MW is open source, which allows us to demonstrate the weaving of the style specific code with the middleware's implementation. However, note that the overall approach presented in this paper is independent of any particular platform. As discussed in Section 9, we have also applied and evaluated the approach on another open-source middleware platform.

Prism-MW supports architectural abstractions by providing classes for representing each architectural element, with methods for creating, manipulating, and destroying the element. These abstractions enable direct mapping between an architecture and its implementation. Fig. 2 shows a partial class design view of Prism-MW. Essentially Fig. 2 corresponds to the *Generic Architectural Support* layer of the middleware stack (recall the left hand side of Fig. 1), and shows the interrelationships between its constructs.

Brick is an abstract class that represents an architectural building block. It encapsulates common features of its subclasses (*Architecture*, *Component*, *Connector*, and *Port*). *Architecture* records the configuration of its constituent components, connectors, and ports, and provides facilities for their addition, removal, and reconnection, possibly at system run-time.

*Event*s are used to capture communication in an architecture. An event consists of a name and payload. An event's payload includes a set of typed parameters for carrying data and meta-level information (e.g., sender, type, and so on). An event type is either a *request* for a recipient to perform an operation or a *reply* that a sender has performed an operation.

*Port*s are the loci of interaction in an architecture. A port can be connected to at most one other port. Each port has a type, which is either *request* or *reply*. Request events are always forwarded from request to reply ports; reply events are forwarded in the opposite direction.

*Component*s perform computations in an architecture and may maintain their own internal state. The developer provides the application-specific logic by extending the component class. Each component can have an arbitrary number of attached ports. Components interact via their ports.

*Connector*s are used to control the routing of events among the attached components. As components, each connector can have an arbitrary number of attached ports.

Components and connectors attach to one another by creating a *link* between a component port and a single connector port. A *link* between two ports is made by *welding* them together. In order to support the needs of dynamically changing applications, each Prism-MW component or connector is capable of adding or removing ports at run-time (Malek et al., 2005).

Finally, Prism-MW's core associates the *Scaffold* class with every *Brick*. *Scaffold* is used to schedule and queue events for delivery (via the *AbstractScheduler* class) and pool execution threads used for event dispatching (via the *AbstractDispatcher* class) in a decoupled manner. Prism-MW's core provides the default implementations of *AbstractScheduler* and *AbstractDispatcher*. For brevity, we do not discuss many other Prism-MW facilities (e.g., distribution, monitoring, reflection, service discovery) that are not directly relevant to this research. Interested reader should refer to (Malek et al., 2005).

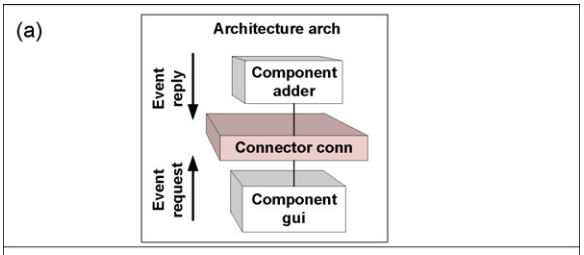
Prism-MW's core provides the necessary support for developing arbitrarily complex applications, as long as they rely on the provided default facilities (e.g., event scheduling, dispatching, and routing) and stay within a single address space. The first step a developer follows (performs) is to create a subclass (or to extend) from the *Component* class for all components in the architecture and to implement their application-specific methods. The next step is to instantiate the *Architecture* class and to define the needed instances of components, connectors, and ports. Finally, attaching component and connector instances into a configuration is achieved by using the *weld* method of the *Architecture* class.

For illustration, Fig. 3 shows a simple usage scenario of the Java version of Prism-MW. The application consists of two components communicating through a single connector. The *Calculator* class's *main* method instantiates components, connectors, and ports; adds them to the architecture; and composes (*weld*s) them into a configuration. Fig. 3 also demonstrates event-based communication between the two components. The *GUI* component creates and sends an event with two numbers in its payload, in response to which the *Adder* component adds the two numbers and sends the result back via an event. In core Prism-MW, an event need not identify its recipient components; they are uniquely defined by the topology of the architecture and routing policies of the employed connectors (Mehta et al., 2000).

5. Crosscutting impact of style

We have previously argued in (Malek, 2008a) that effective support for architectural styles in a middleware platform requires:

- The ability to distinguish among different architectural elements of a given style (e.g., distinguishing Clients from Servers in the Client–Server style).
- The ability to specify the architectural elements' stylistic behaviors (e.g., Clients block after sending a request in the Client–Server style, while C2Components send requests asynchronously in the C2 style (Taylor et al., 1996)).
- The ability to specify the rules and constraints that govern the architectural elements' valid configurations (e.g., disallowing Clients from connecting to each other in the Client–Server style, or allowing a Filter to connect only to a Pipe in the Pipe–Filter style).



(b)

Architecture initialization

```
class Calculator {
  static public void main(String argv[]) {
    FIFOScheduler sched = new FIFOScheduler(50);
    RRobinDispatcher disp = new RRobinDispatcher(5);
    Architecture arch = new Architecture(sched, disp);

    // create components and connectors here
    Component adder = new Addition();
    Component gui = new GUI();
    Connector conn = new Connector();

    // add components and connectors to architecture
    arch.add(adder);
    arch.add(gui);
    arch.add(conn);

    // establish the interconnections
    arch.weld(gui, conn);
    arch.weld(conn, adder);

    arch.start(); } }
```

GUI component sends an event

```
Event request = new Event ("add");
request.addParameter ("num_1", new Integer (2));
request.addParaemter ("num_2", new Integer (5));
send (request);
```

Adder component handles the event

```
public void handle (Event e){
  if (e.equals("add")) {
    ...
    Event reply = new Event ("response");
    reply.addParameter("result", new Integer (7));
    send (reply);
    }...
}
```

Fig. 3. Illustration of Prism-MW usage: (a) architecture of a small calculator and (b) application implementation fragments for the calculator.

This suggests that styles have a significant impact on the behavior and structure of all the architectural constructs. Below we further demonstrate the extent of this using Prism-MW. Prism-MW's extensive separation of concern and modularized implementation of architectural constructs allow us to demonstrate the crosscutting impact of styles most effectively.

Like the majority of commercial middlewares, Prism-MW's core is style agnostic, and to provide support for an architectural style, one would have to modify its facilities. There are two ways of doing this: (1) leverage Prism-MW's extensible classes to override the core behavior (shown in Fig. 2 and discussed in Malek et al., 2005), or (2) modify the implementation of the core classes directly. Neither results in a localized, modularized, and decoupled solution, as desired.

For the clarity of exposition, we describe the changes to the middleware using the second approach:

- As mentioned in our first requirement, before we can enforce the stylistic rules and constraints, we need to be able to distinguish the style of each architectural construct. One approach is to define a new interface for each stylistic type. For instance, to provide support for Client and Server component types, we change the Component class to implement a Client or Server interface, respectively.
- As mentioned in our second requirement earlier, we may need to:
  - Modify the behavior of core Prism-MW *Connector* to support style specific event routing policies. For example, Pipe forwards data unidirectionally, while a C2Connector uses bidirectional event broadcast (Taylor et al., 1996). For this we would need to modify the core connector's *handle* method, which is responsible for routing events.
  - Modify the behavior of core Prism-MW *Component* to provide synchronous component interaction. The default, asynchronous interaction is provided by the core component's *send* method. For example, we may need to modify the default behavior to enable a Client block after it sends a request to a Server and unblock when it receives a response.
  - Modify the behavior of core Prism-MW *Port* to support different types of inter-process communication (e.g., socket-based, infrared). Prism-MW's core ports only provide support for a single address space.
  - Modify core Prism-MW *Event* to support new event types. For example, a C2Component in the C2 style exchanges Notifications and Requests, while Publisher and Subscriber components in Publish–Subscribe style exchange Advertisements, Subscriptions, and Publications.
- As mentioned in our third requirement earlier, we may need to specify and enforce constraints on the allowable configurations. For this, we would need to modify the *Architecture*'s *weld* method to ensure that the topological constraints of a given style are satisfied. The *weld* method is used to connect components and connectors by associating their ports with one another. For example, in the Client–Server style, Clients can connect to Servers, but two Clients cannot be connected to one another.

From the above discussion it is evident that supporting a new architecture style in Prism-MW impacts most of the middleware's facilities. It also shows that changes are dispersed among the various parts of the middleware's implementation. In fact, supporting a style may require changing all of the grayed out classes shown in Fig. 2.

The situation is exacerbated with middlewares that do not provide the same level of support for architecture-based development as Prism-MW. Finding the classes that need to be modified to realize a particular feature of a style is fairly straightforward in Prism-MW. This is not the case with the majority of commercial middlewares that do not provide explicit support for some of the architectural concepts (e.g., connector, configuration).

6. Stylistic aspects

Below we detail our approach for providing implementation support for three well-known styles on top of Prism-MW and by using AspectJ (AspectJ). The three styles are Client–Server, Layered, and Cache.

Our approach consists of four steps: (1) define a new aspect for each architectural style; (2) define style specific roles for the different architectural constructs; (3) provide new style specific facilities using the aspect's inter-type declaration; and (4) override or refine the middleware's default behavior by interjecting the new logic.

6.1. Client–Server style

We first describe an aspect that realizes one of the most commonly used styles in the development of distributed software systems: Client–Server. Clients send requests to Servers, and Servers process the requests and return the results. Client–Server style disallows two clients from connecting and making requests to one another. Moreover, a Client's thread of execution should block

```
01. public aspect ClientServerStyle {

02.   // the component types (roles) in the style
03.    public interface Client {};
04.    public interface Server {};

05.   //the places in the middleware that need to change
06.    pointcut requestService(Client c):
07.         (call(* *.send(Event)))&& target(c) ;
08.    pointcut receiveResponse(Client c):
09.         (call(* *.handle(Event)))&& target(c);
10.    pointcut connect(Brick b1,Brickb2):
11.         (call(* *.weld(Brick,Brick)))&&
12.            target(b1)&&target(b2);

13.   // blocking the client's thread of execution
14.    after(Client c): requestService(c)
15.     { ((Object)c).wait(); }

16.   // unblocking the client's thread of execution
17.    after(Client c): receiveResponse(c)
18.     { ((Object) c).notifyAll(); }

19.   // checking the topological constraint
20.    before(Brick b1,Brick b2): connect(b1,b2) {
21.     if((b1 instanceof Client)&&(b2 instanceof Client)){
22.         System.out.println("Two clients cannot be
23.        connected to each other");
24.         System.exit(0); }
25.    }
26. }
```

Fig. 4. Client–Server style aspect.

until its request is processed by the server (Fielding, 2000). Finally, some interpretations of the style include the ability to buffer client requests on the service-side.

Fig. 4 depicts the Client–Server's aspect. In lines 2–4 the two main types in the style are defined: *Client* and *Server*. Note that the notion of type corresponds to the role of an architectural element in a style. As will be detailed in Section 7, we use aspect's inter-type declaration to make application components implement one of these interfaces, and hence play a specific role in the architecture. This allows us to type the system's components based on their role in the intended architecture.

Lines 5–12 show the declaration of the pointcuts that precisely specify the places in the middleware that need to be modified. For instance, *requestService* is a pointcut that picks out join points dealing with the invocation of the Client's *send* method. Similarly, *receiveResponse* is a pointcut that picks out join points dealing with the invocation of the Client's *handle* method. Finally, connect is a pointcut that picks out join points dealing with the invocation of the architecture's *weld* method.

Lines 13–25 show the advices that realize the required changes for supporting Client–Server behavior in the middleware as follows: *after* a Client makes a service request, the execution thread is blocked; *after* a response is received, the Client is unblocked; and *before* components are connected, the topological constraints are checked, where two clients are disallowed from connecting to one another.

Fig. 4 shows only one possible implementation of Client–Server. Other interpretations of the style are also possible. For instance, we have also created an alternative implementation, where the Server components queue incoming requests. For brevity, we do not show other implementations of the style. However, it is evident that in our approach fine-tuning a style's implementation is relatively easy. It requires changes to only a single aspect, and since the places in the middleware where pointcuts impose are already identified, the changes could be made by an informed application developer familiar with the middleware.

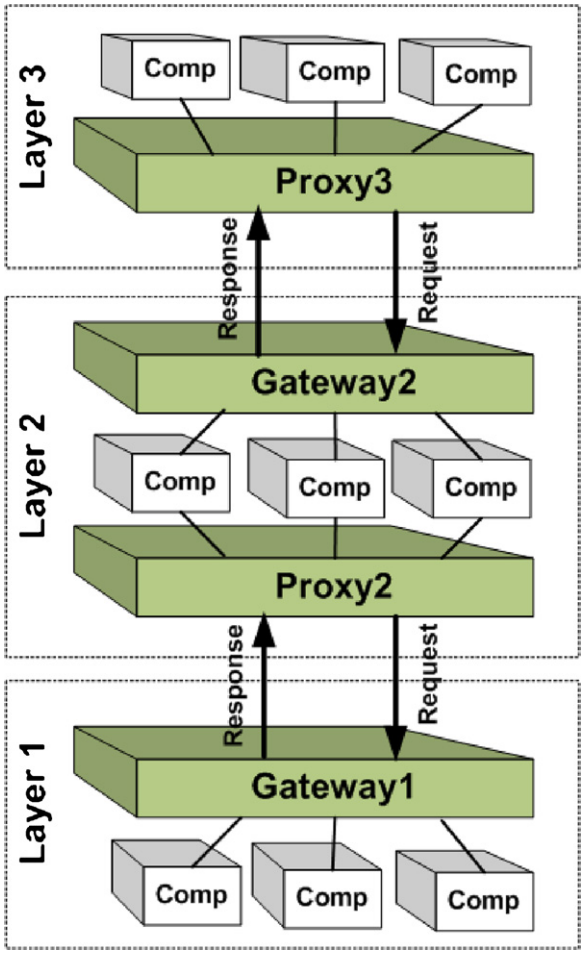


Fig. 5. Layered style.

6.2. Layered style

Another architectural style that we have implemented using our approach is the Layered style. A layered system is organized hierarchically such that each layer provides services to the layer above it and uses services of the layer below it (Fielding, 2000). As depicted in Fig. 5, the Layered architecture consists of two connector types: Proxy and Gateway.

A Proxy connector provides a shared interface for one or more requesting components. It accepts the requests from the "inner layer" components, and forwards them, with possible translation, to the servicing components below. Gateway connector provides a similar functionality for the servicing components. The "inner layer" components reply with the service results to a Gateway, which forwards the results, with possible translation, to the layers above. Additional facilities can be provided for features like load balancing, authentication, and so on.

Unlike the Client–Server style, the Layered style defines roles for connectors, instead of components. Moreover, it specifies relatively more complex topological constraints. The style disallows direct connection between two proxies and two gateways. In addition to this, *layer jumping* is not allowed. Layer jumping occurs when the proxy of one layer connects to a gateway other than the one directly below it. For example, in Fig. 5, a connection between Proxy3 and Gateway1 is not allowed.

Fig. 6 shows the aspect that realizes the Layered style. Lines 3 and 4 define the two roles in the style: Proxy and Gateway. This time we have modeled the roles as Java abstract classes, instead of interfaces, since we would like the role to define a new vari-

```
01. public aspect LayeredStyle {

02.  // the connector types (roles) in the style
03.  public abstract class Proxy {  int layerNum;  };
04.  public abstract class Gateway {  int layerNum;  };
...
05.   // the place in the middleware that is impacted
06.   pointcut connect(Brick b1,Brick b2):
07.      (call(* *.weld(Brick, Brick)))&& target(b1)&& target(b2);

08.  // check topological constraints
09.  before(Brick b1, Brick b2): connect(b1, b2) {
10.     if((b1 instanceof Proxy)&& (b2 instanceof Proxy)) {
11.        System.out.println("Cannot weld two proxies");
12.        System.exit(0); }
13.     else if((b1 instanceof Gateway)&& (b2 instanceof Gateway)) {
14.        System.out.println(" Cannot weld two Gateways");
15.        System.exit(0); }
16.      //check for layer jumping
17.     else {
18.        if ((b1 instanceof Proxy)&& (b2 instanceof Gateway)){
19.           int proxyLayNum = ((Proxy)b1).layerNum;
20.           int gatewayLayNum = ((Gateway)b2).layerNum; }
21.        else {
22.           int proxyLayNum = ((Proxy)b2).layerNum;
23.           int gatewayLayNum = ((Gateway)b1).layerNum; }
24.         if (! ((proxyLayNum-1) == gatewayLayNum)) {
25.            System.out.println("Proxy of Layer "+proxyLayNum+
26.             " Cannot be connected to Gateway of Layer "+
27.            gatewayLayNum);
28.            System.exit(0); }
29.     } } }
```

Fig. 6. Layered style aspect.

able: *layerNum*. *layerNum* specifies the layer number a Gateway or Proxy belongs to. For brevity, we have elided the code that sets the *layerNum* variable of Gateway and Proxy connectors.

Similar to the Client–Server style, lines 5–7 specify a pointcut to intercept the connection (*weld*) calls. Lines 8–29 use the pointcut and specify a *before* advice to check the topological constraints mentioned earlier.

6.3. Cache style

In the previous examples, the stylistic role (type) was associated with only a single type of architectural construct: Client–Server specifies two types of components, while the Layered style specifies two types of connectors. In this section, we present the implementation of Cache style. Unlike the previous styles, the Cache role may be associated with any of the architectural constructs.

A cache is a storage where the responses to prior requests can, if considered cacheable, be stored and reused to service similar requests in the future. The underlying assumption is that the response to a new request is similar to that in the cache if the request was to be forwarded to the processing component (Fielding, 2000). Fig. 7 depicts an instance of the Cache style, where the caching is performed by one of the components. Similarly, the caching activity may be performed by a connector or a port.

Fig. 8 shows the implementation of the Cache style. In line 3 we introduce the role of a *Cache*, which as will be discussed in Section 7 can be associated with any architectural construct. Line 5 allocates storage space for the cached data. Lines 6–11 define the *requestService* and *receiveResponse* pointcuts, which respectively pick out join points that call the *send* and *handle* method of any construct implementing *Cache*. The two pointcuts indicate the places in the middleware where cache activities are performed.

Lines 12–20 implement the cache lookup functionality. The *around* advice is invoked in place of the *send* method. As you may recall from Fig. 3, the *send* method is how any architectural construct in Prism-MW sends events. If an event with the same name is in the storage, it is immediately retrieved and sent back to the requester by calling its *handle* method. Otherwise, when there is a cache miss, the AspectJ's *proceed* construct is used to continue with the normal flow of execution (i.e., call the *send* method).

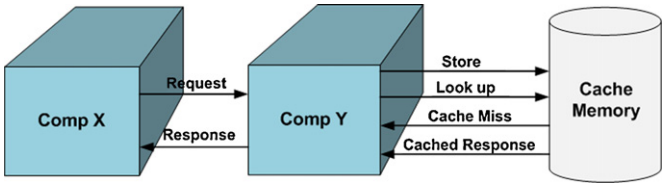


Fig. 7. Cache style.

```
01. public aspect CacheStyle {

02.  // the cache type (role)
03.  public interface Cache{};

04.  // storage used for caching data
05.  static Map storage = new HashMap();

06.  // place where the cache lookup is performed
07.  pointcut requestService(Cache c):
08.          (call(* *.send(Event)))&& target(c);
09.  // place where the cache store is performed
10.  pointcut receiveResponse(Cache c):
11.          (call(* *.handle(Event)))&& target(c);

12.  // lookup in the cache
13.  void around(Cache c, Event e):requestService(c)&& args(e) {
14.    Object key = e.name;
15.    if (storage.containsKey(key))  {
16.       Event cachedResponse = (Event) storage.get(key);
17.       ((Brick)c).handle (cachedResponse); }
18.    // if not in the cache proceed to send the request
19.    else
20.         proceed(c,e); }

21.  // cache the result in the storage
22.  after(Cache c, Event e):receiveResponse(c)&& args(e) {
23.    if (e != null){
24.       Object key = e.name;
25.       if (!storage.containsKey(key))
26.          storage.put(key, e);
27.    } } }
```

Fig. 8. Cache style aspect.

Finally, lines 21–27 show the advice that implements the functionality of storing the response for future cache lookups. This advice is invoked *after* the *handle* method is called.

As will be discussed in Section 8, Cache style may be combined with other styles (e.g., Client–Server) to eliminate some interactions, and hence improve efficiency, scalability, and performance. The trade-off, however, is that caching can decrease the reliability. This is a problem when the stale data in the cache differs significantly from the data that would have been obtained if the request had been processed (Fielding, 2000).

To that end, we have also developed a more practical implementation of this style, where each stored event is tagged with an expiration time stamp. When the lookup advice encounters an expired event, it deletes it from the storage. For brevity we have not shown the details of this in the code snippet.

The above examples demonstrate the efficacy of implementing styles in middleware using our approach. Most notably, the resulting style specific code is both localized and modularized, which in turn improves the system's ability to evolve, and aids with the system understandability.

7. Using the weaved middleware

So far we have described how aspects can be used to modify the core behavior of a middleware to provide support for styles. In this manner, we have increased the decoupling between the middleware core facilities and the stylistic functionality, since the middleware does not contain any style specific code. In other words, we have inverted the dependency relationship, such that the style aspects depend on the middleware code, and not the other way around.

In this section, we describe another advantage of our approach. We remove the style dependency between the application logic and the middleware as follows. To specify the stylistic role of an application-level component we use aspect's *inter-type declaration*. Inter-type declarations allow the programmer to modify a program's static structure, namely, the members of its classes and the relationship between classes.

Inter-type declarations are used to specify the stylistic role of the components in the calculator application (recall Fig. 3) as follows:

```
public aspect CalculatorStyleConfig {
   declare parents: gui implements Client;
   declare parents: adder implements Server;
}
```

The *declare parents* construct allows aspects to modify existing classes without changing their code. This open class mechanism can attach fields, methods, or – as in this case – interfaces to existing classes (AspectJ). Here *gui* and *adder* components of the calculator application are instructed to implement the *Client* and *Server* interfaces, respectively.

Using inter-type declaration, the application developers are able to configure the style of their software systems after its development. This is unlike the existing middleware technologies, where the style of the application is determined based on the stylistic properties assumed by the middleware. Moreover, by inverting the style dependency relationship, we are able to control the style of an application externally. In turn, with the new advancements in dynamic AOP (Greenwood and Blair, 2004), our approach enables run-time adaptation of a software system's architectural style by simply (un)weaving new aspects.

8. Composing hybrid styles

Most large-scale software systems cannot be built using a single style (Malek et al., 2007). As a result, the application developers are forced to use multiple middleware platforms, where each is geared to a particular style, for implementing the system. It is also often desirable to use *hybrid* or *domain-specific* styles in the construction of software systems (Malek et al., 2007). A hybrid style inherits the properties of two or more "pure" styles, while a domain-specific style may refine the rules of an existing style to account for the unique characteristics of a domain.

There are several advantages in using hybrid styles. For instance, the Layered–Client–Server is generally considered a more appropriate style than Client–Server for the construction of large-scale distributed systems (Andrews, 1991). This is because the Gateways and Proxies can be utilized for load balancing, and also for scaling the service-discovery by reducing the number of identities that need to be managed.

Our approach enables rapid composition of hybrid styles by simply weaving two or more style aspects with the middleware. However, an issue of concern is the compatibility of different styles. Each style imposes its own set of constraints, which may conflict with the rules of another style. In turn, the combined use of incompatible styles in a single system can lead to unpredictable and expensive mismatches (Garlan et al., 1995).

During style composition, there are multiple aspects that may impose at a particular join point. When more than one aspect superimpose at a particular join point, the join point is said to be shared (Babu and Ramnath Krishnan, 2009). Different orders of execution among aspects at a Shared Join Point (SJP) may exhibit different behavior.

Two categories of interference may arise:

- Changing the execution order of two aspects at a SJP results in the same observable behavior. For example, the advices that check the topological constraints in the Client–Server and Layered aspects do not refer to the effect of the other, and simply maintain their own state. In other words, the topological constraint checks for a hybrid style, such as Layered–Client–Server, may be performed in any order.
- Changing the execution order of two aspects at a SJP results in different observable behavior. More importantly, some execution orders may violate the intended behavior of the system. For example, consider a scenario where the Client–Server and Cache aspects are composed together. During the calls to the *handle* method, there are two advices that are superimposed at that particular join point: *after* of *receiveResponse* defined in both Cache (line 22 of Fig. 8) and Client–Server aspects (line 17 of Fig. 4). The desired order of execution is for the advice of Cache to be executed before the advice of Client–Server. This ordering enables the Cache to store the response before the Client is unblocked. If the advices execute in the opposite order, the Client is unblocked first, and may send more requests before the response is stored. As a result, some of the responses may not be cached.

The execution order of aspects can be specified using the *declare precedence* construct, as follows:

```
public aspect StylePrecedence{
   declare precedence:
         CacheStyle, ClientServerStyle;
}
```

Here we instruct the Cache aspect to be executed prior to the Client–Server aspect to form the Cache–Client–Server style.

When more than one advice within a single aspect applies at a particular join point, it is also important to specify the order of execution. In single style as well as in hybrid styles, the relative order in which such advices execute needs to be well defined. The rule for determining advice ordering (also known as *specificity*) that superimpose at a particular join point is that whichever piece of advice appears first in the aspect declaration's body is considered to be more specific (Kiczales et al., 2001). This commonly happens when there are matching before and after advice, but it can also happen with two pieces of advice that are of the same kind.

Finally, we should mention that if a domain-specific style does not refine the rules of an existing style, then it would be implemented from scratch, just like any other "pure" style. Clearly, in such a case, there would be no code reuse.

9. Evaluation

We provide a detailed evaluation of the approach, including our experience with applying the approach in real-world context. In support of our claims that the overall approach is independent of any platform, we have applied it to another open-source middleware, called Spread (Spread Toolkit). Spread provides a high performance messaging service that is resilient to faults across local and wide area networks. Both Prism-MW and Spread have a modular design, which makes them a natural choice for the illustration of our approach. We evaluate the approach using both middleware platforms.

9.1. Benchmark results

The performance and efficiency of a middleware platform is often of utmost concern to the middleware engineers. We evaluated the performance overhead of the approach by comparing the weaved versions (i.e., AOP) against manually revised versions

Table 1
Benchmark results: weaved version (WV) and manually modified version (MMV).

Middleware	Architectural style	Execution time (ms)		Memory usage (KB)	
		WV	MMV	WV	MMV
Prism-MW	Client–Server	2022	2015	117	114
	Caching	2002	1999	129	121
	Caching–Client–Server	2042	2031	145	132
	Layered–Client–Server	2068	2054	138	127
Spread	Client–Server	2889	2877	176	169
	Caching	2876	2862	184	172
	Caching–Client–Server	2901	2893	191	186
	Layered–Client–Server	2923	2904	189	178

(i.e., OO) of the middleware. That is for each weaved version, we created an equivalent instance of the middleware by adding the style specific code directly to the middleware's core classes. Four representative architectural styles (i.e., Client–Server, Cache, Cache–Client–Server, and Layered–Client–Server) were selected and implemented using both approaches.

Since we were interested in the overhead induced by the middleware, the complexity of the application-level logic was irrelevant. Therefore, we used the simple calculator application depicted in Fig. 3a for benchmarking the middleware's performance. For each data point we invoked the addition functionality 1,000,000 times and measured the running time and average memory overhead of executing the 8 versions of each middleware. The environment set-up consisted of a mid-range PC with Intel Pentium IV 1.86 GHz processor and 3 GB of RAM running JVM 1.5.0 on Windows XP.

Table 1 shows the execution time and memory usage of Prism-MW and Spread Middleware under different realization of styles. The results indicate that the manually modified versions are slightly faster than the ones automatically generated. However, this overhead is negligible, and does not outweigh the benefits of the approach when faster running times are not critical.

The results also indicate that our approach on average consumes 3–10% more memory than an implementation without aspects. While the memory overhead is an importance concern, we expect the recent advances in aspect compilers, in particular those (Avgustinov et al., 2005) directed at AspectJ, will soon make it possible to generate optimized code with significantly smaller memory footprint.

9.2. Implementation properties

Our experience with using the approach in implementing more than 20 different architectural styles described in Fielding (2000) has been very positive. In the process, we noticed that the amount of effort required to implement a new style decreased over time. That is as more styles were developed, the level of code reuse increased. Our approach achieves reuse in two ways: partial code reuse among similar styles, and complete code reuse through style composition. The former is attributed to the fact that architectural styles promoting similar quality attributes often have common traits with one another. For instance, both Layered and C2 style promote flexibility and separation of concerns, and as a result there is a significant implementation overlap between the two style aspects. While similarity of styles is a good indicator of potential for partial code reuse, it often has the opposite effect on the compositionality of those styles. Our experience shows that the styles that are similar (e.g., C2 and Layered) often cannot be composed together. We further elaborate on this below.

Table 2 shows the properties that were evaluated for the AOP implementation of six representative styles in both Prism-MW and Spread. As you may recall from Section 3, the key properties of interest are as follows: (1) Locality is an indicator of the ability to realize stylistic properties within an aspect, making the middleware's core functionality free from any style specific code, and thus improving the middleware's understandability and maintainability. (2) (Un)pluggability is the quality of the middleware facilities not being aware of their role in architectural styles. This in turn facilitates changing the architectural style of an application through weaving the stylistic aspect with the underlying middleware, potentially at run-time. (3) Compositionality is the ability to reuse styles by weaving them together and constructing hybrid and more advanced capabilities.

As depicted in Table 2, the implementation of styles in both middleware platforms satisfied the locality and (un)pluggability properties. While each style could be composed with at least one other style (the right most column of Table 2), as expected some styles were not compatible and could not be composed with one another. For instance, Client–Server and Publish–Subscribe rely on fundamentally different assumptions that make them incompatible with one another (i.e., Publish–Subscribe assumes time, space, and synchronization decoupling, while Client–Server does not, Fielding, 2000). In some cases, even though it was possible to compose two styles, the resulting style would not be meaningful and hence not shown in Table 2. For instance, C2 inherently has the notion of layering (Taylor et al., 1996), and hence a composition of C2 with Layered would be redundant. Note that some styles, such as Caching, were so flexible that could be composed with any other style. We believe the ability to rapidly compose styles in this manner opens up new avenues of research to study compatibility of styles with one another and to develop new hybrid and domain-specific styles.

Finally, Table 2 shows that our experiments showed no major difference in the implementation properties of styles in the two middlewares. While implementation of a style in one middleware could guide the development of support for that style in another middleware, complete reuse of style aspects across different middlewares without any modification to the code is not likely. When porting a style aspect from one middleware platform to another that has a similar implementation, the aspect's pointcuts would have to be modified to pick the appropriate join points.

9.3. Design quality

As mentioned before through the use of aspects we are able to separate the stylistic concerns from other concerns in the middleware's design. To evaluate the quality of a middleware designed and developed in this fashion, we use the modularity metric described in Cacho et al. (2006) and Sant'Anna et al. (2003). These metrics are useful to capture important design quality dimensions namely separation of concerns, coupling, and cohesion. These metrics capture the degree to which a single concern in the system maps to the design components (classes and aspects), operations (methods and advice), and lines of code. We found the following metrics adopted

Table 2
Implementation properties for the weaved version of the middleware.

Style name	Middleware	Locality	(Un)pluggability	Compositionality
Client–Server	Prism-MW/Spread	√	√	Layered Caching
Layered	Prism-MW/Spread	√	√	Client–Server Caching Pipe–Filter
Caching	Prism-MW/Spread	√	√	Client–Server Layered Publish–Subscribe Pipe–Filter C2
Publish–Subscribe	Prism-MW/Spread	√	√	Caching C2
Pipe–Filter	Prism-MW/Spread	√	√	Caching Layered Publish–Subscribe
C2	Prism-MW/Spread	√	√	Caching Publish–Subscribe

from Sant'Anna et al. (2003) to be most sensible for evaluating our research:

- Concern Diffusion over Components (CDC): counts the number of classes and aspects whose main purpose is the implementation of a concern and the number of other classes and aspects that access them.
- Concern Diffusion over Operations (CDO): counts the number of methods and advices whose main purpose is the implementation of a concern and the number of other methods and advices that access them.
- Concern Diffusions over LOC (CDLOC): counts the number of transition points for each concern through the lines of code. Transition points are points in the code where there is a "concern switch".
- Coupling Between Components (CBC): counts the number of other classes and aspects to which a class or an aspect is coupled.
- Lack of Cohesion in Operations (LCOO): counts the number of methods and advice pairs of each class or aspect that do not access the same instance variable.
- Similar to previous work (Cacho et al., 2006; Sant'Anna et al., 2003), the data collection for metrics was preceded by the shadowing of every class, interface, and aspect realizing an architectural style in both manually modified and weaved versions of the middleware. The shadowed areas are lines of code that implement a given concern, and the transition points are the points in the code where there is a transition from a non-shadowed area to a shadowed area and vice-versa. Afterwards, the metrics were manually collected. We ensured that both OO and AOP realization of an architectural style implement the same set of constraints and facilities.

Table 3 shows the metrics for seven representative styles supported in two middleware platforms. Side-by-side comparison of each metric for the two methods of realizing each style clearly demonstrates the superiority of weaved version. Note that in all of the metrics a smaller value is an indicator of improved modularity, and hence a better design. CDC, CDO, and CDLOC metrics indicate significant modularity improvements due to the separation of concerns achieved by employing AOP. CBC and LCOO metrics indicate reduced coupling and increased cohesion due to the fact that AOP enables localization of stylistic concerns within an aspect.The benefits of the AOP approach are best realized in the composition of hybrid styles. Table 3 shows that the metric differences between manually modified and weaved versions of the middleware are larger for hybrid styles. This is attributed to the fact that hybrid styles are typically more complex than basic styles; thus, the potential for code scattering and tangling increases even further.

9.4. Experience

We have validated and applied the approach on a real-world application family that was developed previously in collaboration with Bosch Research and Technology Center. The application family, called MIDAS (Malek et al., 2007), is representative of large-scale sensor network applications. MIDAS is composed of a large number of sensors, gateways, hubs, and PDAs that are connected wirelessly. MIDAS could be used for a variety of purposes, including smart hospitals to warn the doctors of extreme health condition of patients, smart factories to detect and prevent hazardous waste spills, and so on.

MIDAS application instances are developed according to a domain-specific architectural style that is frequently used by Bosch for the development of their embedded sensor network applications. Fig. 9 shows a partial view of MIDAS's reference architecture, which incorporates features similar to three well-known architectural styles. The Layered portion of this architecture is used for the deployment and run-time adaptation of software running on MIDAS platforms; hence, the components that provision this functionality are tagged as *meta-level components*. The Publish–Subscribe portion corresponds to the communication backbone of MIDAS that is responsible for routing and processing of sensor data among the various platforms. Unlike the services provided by Publish–Subscribe components that are platform-specific, MIDAS applications also require a number of more generic but less frequently used services. To minimize resource utilization, these services are distributed among the platforms and comprise the Service-Oriented portion of MIDAS.

Previously, in a collaborative effort with Bosch engineers we had manually extended Prism-MW to provide support for MIDAS. The details of this experience are reported in our previous work (Malek et al., 2007). To evaluate the AOP approach of realizing support for styles, we developed three basic style aspects, each of which corresponds to one of the three types of styles used in MIDAS, and composed them together to realize a middleware customized to MIDAS.

It took 4 days for two developers relatively familiar with both MIDAS and Prism-MW to develop a MIDAS specific style aspect and to generate a new customized version of Prism-MW. Clearly the actual amount of effort required depends on many factors, including the familiarity of the developers with the middleware, AOP, and the desired style. At the same time, our experience shows that after the initial development of the support for MIDAS style, even the developers that were not expert in the middleware were able to fine-tune the implementation when necessary. For instance, in several occasions, developers that were not familiar with the internals

Table 3
Design quality metrics: manually modified version (MMV) and weaved version (WV).

Middleware	Architectural Style	CDC		CDO		CDLOC		CBC		LCOO	
		MMV	WV	MMV	WV	MMV	WV	MMV	WV	MMV	WV
Prism-MW	Client–Server	9	1	15	3	42	9	10	2	4	0
	Caching	10	1	13	2	35	7	12	2	5	0
	Publish–Subscribe	12	1	16	5	55	14	14	3	4	1
	Pipe–Filter	14	1	18	4	62	13	11	2	3	0
	C2	8	1	10	2	28	12	13	3	4	2
	Caching–Client–Server	11	2	18	5	67	14	16	4	6	0
	Layered–Client–Server	17	2	22	5	78	15	16	4	5	0
Spread	Client Server	6	1	7	2	20	4	6	1	1	0
	Caching	5	1	7	1	22	5	8	2	2	0
	Publish–Subscribe	4	1	9	2	33	7	9	2	1	0
	Pipe–Filter	7	1	12	2	18	4	7	2	2	1
	C2	6	1	9	2	28	6	8	3	3	1
	Caching–Client–Server	8	2	14	3	38	9	9	3	2	0
	Layered–Client–Server	10	2	16	3	40	6	10	3	3	0

of Prism-MW faced situations where it was beneficial to slightly modify or fine-tune the style. In one case the SOA portion of MIDAS was modified to provide support for fault-tolerance by creating replicas of services, while in another case the Layered portion of MIDAS was changed to Peer-to-Peer for efficiency purposes. Since the style implementation is localized in an aspect external to the middleware, it was relatively easy for the developers to conceptualize the support for a style and its impact on the middleware's behavior. As we had hypothesized, the localization made it significantly easier to modify and maintain the middleware.

In the context of this project, we studied the effectiveness of the MIDAS specific middleware in ensuring architectural compliance (recall Section 3). To that end, a team of developers was asked to develop a portion of MIDAS application depicted in Fig. 9 using the MIDAS specific version of the middleware. The total size of the resulting code was approximately 12.3 KSLOC and consisted of 15 software components. To streamline the development, the teams were allowed to reuse code written for a previous version of the software. We compared the code developed using the MIDAS specific version of the middleware against the original code developed in our previous work using a style-agnostic version of the middleware (Malek et al., 2007). We observed that the developers using the style-agnostic middleware had made 17 implementation choices in the original implementation that conflicted with the MIDAS's stylistic requirements. Moreover, some of the implementation choices violated the system's principal architectural decisions. On the other hand, MIDAS specific version of the middleware successfully prevented deviations from the intended style by enforcing the stylistic constraints and providing early feedback to the developers. Only 1 architectural decision, which was not stylistic in nature, did not get implemented correctly. This result corroborated that support for advanced domain-specific styles in middleware could significantly reduce the possibility of architectural erosion.

10. Related work

We classify the literature related to our research into six categories: (1) middleware solutions for architecture-based development, (2) use of aspects in realizing middleware facilities, (3) use of model-driven engineering in the design and construction of mid-

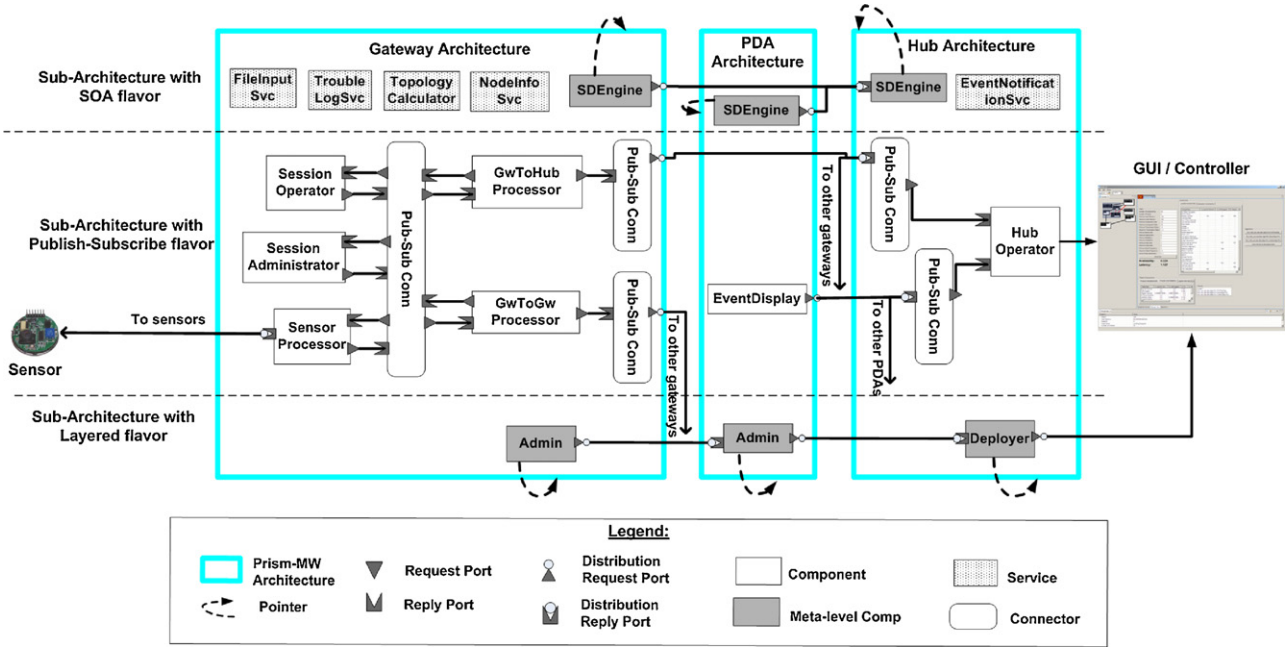


Fig. 9. Partial view of MIDAS's architecture, including its three dominant architectural styles.

dleware platforms, (4) AOP methods for realizing design concepts, (5) aspect-oriented architecture modeling, and finally (6) AOP languages and technologies. Below we first provide an overview of the most prominent works from each category and afterward compare them against our work.

Several middleware technologies for architecture-based development have been proposed:

- C2 Framework (Medvidovic et al., 1996) is an object-oriented library for implementing software systems that comply with the rules and constraints of C2 architectural style (Taylor et al., 1996).
- ArchJava (Aldrich et al., 2002) is an extension to Java that unifies software architecture with implementation. ArchJava does not provide support for enforcing topological constraints, and therefore lacks the support for implementing and enforcing a software system's architectural style.
- Aura (Sousa and Garlan, 2002) is an architectural style and supporting middleware for ubiquitous computing applications with a special focus on user mobility, context awareness, and context switching.
- Aspect-Oriented Development Framework (AODF) is an approach presented by Lee and Bae (2004) for the development of systems according to the collaboration-based architectural style. AODF achieves separation of concerns in both non-functional requirements as well as intra-component and inter-component functional aspects. Most notably, the collaboration-based architectural style proposed by the authors provides the mechanisms for achieving the separation of concern through Aspectual Composition Rules and Aspectual Collaborative Composition Rules.
- Middlewares such as TAO, Orbix/E (IONA Orbix/E Datasheet), .Net, and MobiPADS (Chan and Chuang, 2003) provide partial support for architectural abstractions in the form of explicit components.

With the exception of AODF (Lee and Bae, 2004) and C2 Framework (Medvidovic et al., 1996), none of the above technologies provides explicit support for architectural styles. Instead, the notion of architectural style in these technologies is limited to the style of communication. Unlike our approach, AODF and C2 Framework provide support for only a single predetermined architectural style, and hence do not address the challenges tackled in this work.

Several researchers have investigated the benefits and challenges of using aspects in realizing middleware facilities:

- Zhang and Jacobsen (2003a,b) have undertaken aspect-oriented refactoring in a selection of ORBs. They have shown that through aspect-oriented programming one could obtain a modularity level that is unattainable via traditional programming techniques.
- Cacho et al. (2006) have presented an AOP approach for supporting crosscutting concerns in reflective middleware. They also quantitatively demonstrated the benefits of AspectJ implementation of the middleware over pure Java.
- Hunleth et al. (2001) have used AOP to build customizable middleware by the selection of features that could be weaved into the middleware as needed using specification from a configuration file.
- Colyer and Clement (2004) have used a case study to demonstrate the complexities that surface during middleware construction, and demonstrated the benefits of aspects for mitigating them.

Similar to these works, we employ aspects to realize crosscutting behaviors (capabilities) in middleware platforms. Unlike them, the focus of our research is on providing support for architectural styles, while their focus has been on supporting traditional system-level facilities (e.g., ORB, thread scheduling, and logging) often provided by middlewares.

Several researchers have employed Model-Driven Engineering (MDE) (Schmidt, 2006) and Model-Driven Architecture (MDA)techniques in the context of middleware platforms. Some of the representative examples in this area are as follows:

- CoSMIC (Schmidt et al., 2002) tool-suite provides support for developing domain-specific tools for composing and deploying Distributed Real-time and Embedded (DRE) middleware-based applications. The tool-suite is designed to (1) use a domain-specific modeling language for modeling and analyzing DRE application functionality and QoS requirements, and (2) generate CORBA Component Model deployment meta-data for CIAO (Nanbor et al., 2002) and QuO (Vanegas et al., 1998) middleware platforms.
- Wadsack and Jahnke (2002) argue that since middleware platforms are often relied upon as the means for bridging the heterogeneity among various software and hardware components, their evolution and maintenance are relatively more challenging than traditional software systems. To that end, they propose an MDA approach that aims to reduce the manual effort and increase the quality of resulting middleware.
- Parallax (Silaghi and Strohmeier, 2005) is an MDA framework that enables the developer to configure the design models with middleware-specific concerns at different levels of abstraction, and then generate the implementation of these concerns for different middleware infrastructures. The approach employs a combination of AspectJ aspects with Eclipse plug-ins, which enables aspects to encapsulate concerns that crosscut plug-in boundaries.

Unlike our work, none of the above approaches aims to provide support for the implementation of architectural styles in middleware platforms. However, we believe some of the MDE techniques described above could be used in conjunction with the AOP techniques proposed in our work. As further detailed below, we consider this to be an interesting avenue of future work.

Another area of related work has been the use of aspects in realizing the design decisions:

- Hannemann and Kiczales (2002) have shown the benefits of employing AOP in the implementation of design patterns.
- Cunha et al. (2006) have presented a collection of high-level concurrency patterns and mechanisms in AspectJ. They demonstrate that in comparison to basic Java implementation, the AOP approach achieves higher modularity, reuse, understandability, and (un)pluggability.

Our research is different from these works in two ways. Firstly, our work is geared to the implementation of architectural styles, as opposed to design/concurrency patterns. Secondly, our approach deals with enhancing middleware support for styles.

Related to our research is the notion of *early aspects* (Chitchyan et al., 2005; Cuesta et al., 2005; Rashid et al., 2003), which is the idea of applying aspects during initial stages of software development (e.g., requirements elicitation and architecture modeling phases). Chitchyan et al. (2005) present a comprehensive survey of the state-of-the-art in modeling early aspects. Cuesta et al. (2005) provide a detailed description of the concerns and techniques for modeling architectural aspects. Batista et al. (2006a) and Navasa et al. (2002) describe some of the key issues regarding the integration of AOSD and ADLs. They argue that a systematic integration of architectural

abstractions and AOSD would enhance the existing support for separation and modular representation of crosscutting concerns at the architectural level. The following research is representative of the related work in this area:

- PRISMA (Pérez et al., 2008) is an approach that integrates the software architecture modeling with aspect-oriented software development techniques to take advantage of both. The PRISMA model (Pérez et al., 2005) is a symmetrical model that does not consider functionality as a kernel entity different to aspects and it does not constrain aspects to specify non-functional requirements. PRISMA provides a formal Aspect-Oriented Architecture Description Language (AOADL) (Pérez et al., 2006). Aspects have been introduced in the PRISMA AOADL as a new concept rather than using other architectural constructs (e.g., component, connectors, views, etc.). A recent extension of this approach with concepts from Ambient Calculus has resulted in Ambient-PRISMA (Ali et al., 2010). Ambient-PRISMA supports design and development of mobile applications using Ambient-PRISMANET, a middleware realized on top of .NET technology.
- AspectualACME (Batista et al., 2006b), an extension to ACME (Garlan et al., 1997), incorporates Aspectual Connectors and other facilities to modularize crosscutting concerns in the architectural models. The objective of AspectualACME has been to reduce the number of extensions and additional constructs necessary for modeling architectural aspects.
- AspectLEDA (Navasa et al., 2009) is an aspect-oriented ADL that builds on an aspect-oriented architecture modeling methodology, called AOSA Model (Navasa et al., 2005), and LEDA (Canal et al., 2001). AOSA Model (Navasa et al., 2005) is an architectural description methodology allowing the behavior of a system to be changed by adding or removing logical restrictions without changing the components that constitute them. This is achieved via a base level containing the initial system (core components) and a meta-level containing the new elements (aspect and coordinators in the model). The models constructed in this manner can be translated to LEDA (Canal et al., 2001), which is an ADL with formal underpinnings (pi-calculus) allowing the system's correctness to be verified and the corresponding prototype of the system in Java to be generated.

All of the above approaches are concerned with modeling crosscutting concerns in the architectural models. Unlike them, our research is concerned with the effective implementation of architectural styles in middleware platforms, which is a fundamentally different objective. At the same time, as detailed in the next section, we believe these approaches could complement our work. They provide a high-level language (e.g., AO-ADL) that could potentially be used to model stylistic concerns, which if employed together with MDE techniques could generate middleware-specific style aspects.

We have used AspectJ's pointcut language for our implementation. Some of the other AOP technologies include Aspectwerkz, Alice (Eichberg, 2005; Eichberg and Mezini, 2004), Prose (2010)Prose, and JBoss AOP. We feel our approach could be realized using any of the aforementioned technologies. We chose AspectJ for two reasons: (1) AspectJ can be used to modularize individual middleware services, regardless of the component model supported by the middleware and without any particular assumption on the underlying facilities. (2) AspectJ is widely used in many projects, especially for the implementation of infrastructural services (Eichberg, 2005). While AspectJ is limited to middleware platforms implemented in Java, advances in aspect-oriented programming are making it possible to employ AOP techniques in other languages (e.g., AspectC++).

11. Conclusions and future work

Middlewares have been shown to aid the architecture-based development of software systems. However, due to the crosscutting structure of architectural styles, there is a lack of adequate support for styles in the existing middleware platforms. In this paper, we demonstrated the crosscutting impact of styles on architectural middleware platforms. We presented a new approach to implementing styles that is based on the aspect-oriented programming paradigm. Finally, we showed how style aspects can be reused in composing more advanced hybrid styles.

Aspects allow for modularized and localized implementation of stylistic support in middlewares. They allow an informed engineer to modify the default behavior of a middleware by implementing support for an arbitrary, possibly domain-specific, architectural style. By enforcing the stylistic rules in middleware, we are able to alleviate the developers from resorting to manual techniques for verifying the fidelity of the implemented system with respect to the architectural models.

As part of our future work, we plan to extend our work to existing commercial middleware platforms. Another interesting avenue of future work is to investigate the dynamic aspect weaving of style concerns. This would enable dynamic adaptation of a software system's style by simply (un)deploying the appropriate style aspect at run-time. Finally, we believe the approach presented in this paper could be complemented with the Aspect-Oriented Architectural Description Languages, such as AspectualACME (Batista et al., 2006b), PRISMA AOADL (Pérez et al., 2006), and AspectLEDA (Navasa et al., 2009), to model the crosscutting behavior of architectural styles in terms of architectural constructs. We hypothesize that such models in conjunction with model- and code-transformation techniques from Model-Driven Engineering (Schmidt, 2006) could be used to automatically generate the required middleware-specific code for supporting styles. Such an approach would be useful, as it would free the developer from having to learn AOP as well as the details of the middleware. Instead, the developer would specify the rules and properties of the style in terms of high-level architectural constructs (e.g., using an AO-ADL), which would then be transformed into the corresponding middleware-specific aspect code.

References

Aldrich, J., Chambers, C., Notkin, D., 2002. ArchJava: connecting software architecture to implementation. In: Proceedings of the International Conference on Software Engineering, Orlando, Florida, May 2002, pp. 187–197.

Ali, N., Ramos, I., Solis, C., 2010. Ambient-PRISMA: ambients in mobile aspect-oriented software architecture. Journal of Systems and Software 83 (6 (June)), 937–958.

Andrews, G., 1991. Paradigms for process interaction in distributed programs. ACM Computing Surveys 23 (1 (March)), 49–90.

AspectJ web site. http://www.eclipse.org/aspectj/.

AspectC++ web site. http://www.aspectc.org/.

Aspectwerkz web site. http://aspectwerkz.codehaus.org/.

Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhotak, J., Lhotak, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J., 2005. Optimizing AspectJ. In: Proceedings of the International Conference on Programming Language Design and Implementation (PLDI 2005), Chicago, Illinois, June 2005, pp. 117–128.

Babu, C., Ramnath Krishnan, H., 2009. Fault model and test-case generation for the composition of aspects. ACM SIGSOFT Software Engineering Notes 34 (1 (January)), 1–6.

Batista, T., Chavez, C., Garcia, A., Sant'Anna, C., Kulesza, U., Rashid, A., Castor Filho, F., 2006a. Reflections on architectural connection: seven issues on aspects and ADLs. In: Proceedings of the International Workshop on Early Aspects, Shanghai, China, May 2006, pp. 3–10.

Batista, T., Chavez, C., Garcia, A., Kulesza, U., Sant'Anna, C., Lucena, C., 2006b. Aspectual connectors: supporting the seamless integration of aspects and ADLs. In: Proceedings of the ACM SIGSOFT Brazilian Symposium on Soft-

ware Engineering (SBES'06), Florianópolis, Brazil, October 2006, pp. 17–32.
Cacho, N., Batista, T., Garcia, A., Sant'Anna, C., Blair, G., 2006. Improving modularity of reflective middleware with aspect-oriented programming. In: Proceedings of the International Workshop on Software Engineering and Middleware, Portland, Oregon, November 2006, pp. 31–38.
Jean, G.C., Sourrouille, J.L., Pascal, B.B., Cedex, F.V., 2002. Model mapping in MDA. In: Proceedings of the Workshop in Software Model Engineering, Dresden, Germany, October 2002.
Canal, C., Pimentel, E., Troya, J.M., 2001. Compatibility and inheritance in software architectures. Science of Computer Programming 41 (2), 105–138.
Chan, A.T.S., Chuang, S., 2003. MobiPADS: a reflective middleware for context-aware mobile computing. IEEE Transactions on Software Engineering 29 (12 (December)), 1072–1085.
Chitchyan, R., Rashid, A., Sawyer, P., Garcia, A., Pinto, M., Bakker, J., Tekinerdogan, B., Clarke, S., Jackson, A., 2005. Report synthesizing state-of-the-art in aspect-oriented requirements engineering, architectures and design. In: AOSD-Europe Deliverable D11, AOSD-Europe-ULANC-9, Lancaster University, Lancaster, pp. 1–259.
Colyer, A., Clement, A., 2004. Large-scale AOSD for middleware. In: Proceedings of the International Conference on Aspect-Oriented Software Development, Lancaster, UK, March 2004, pp. 56–65.
Cuesta, C.E., Romay, M.P., de La Fuente, P., Barrio-Solórzano, M., 2005. Architectural aspects of architectural aspects. In: Proceedings of the European Workshop on Software Architecture (EWSA 2005), Pisa, Italy, June 2005, pp. 247–262.
Cunha, C.A., Sobral, J.L., Monteiro, M.P., 2006. Reusable aspect-oriented implementation of concurrency patterns and mechanisms. In: Proceedings of the International Conference on Aspect Oriented Software Development, Bonn, Germany, March 2006, pp. 134–145.
Eichberg, M., 2005. Component-based software development with aspect-oriented programming. Journal of Object Technology 4 (3 (April)), 21–26.
Eichberg, M., Mezini, M., 2004. Alice: modularization of middleware using aspect-oriented programming. In: Proceedings of the International Workshop on Software Engineering and Middleware (SEM 2004), Linz, Austria, September 2004, pp. 47–63.
Fielding, R., 2000. Architectural Styles and the Design of Network-Based Software Architectures. PhD thesis, University of California Irvine, June 2000.
Garlan, D., Allen, R., Ockerbloom, J., 1995. Architectural mismatch, or, why it's hard to build systems out of existing parts. In: Proceedings of the International Conference on Software Engineering (ICSE 1995), Seattle, USA, April 1995, pp. 179–185.
Garlan, D., Monroe, R., Wile, D., 1997. ACME: an architecture description interchange language. In: Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON 1997), Toronto, Canada, November 1997, pp. 169–183.
Gorton, I., 2006. Essential Software Architecture. Springer-Verlag, Berlin, Germany.
Greenwood, P., Blair, L., 2004. Using dynamic aspect-oriented programming to implement an autonomic system. In: Proceedings of the Dynamic Aspects Workshop, Lancaster, England, March 2004, pp. 76–88.
Hannemann, J., Kiczales, G., 2002. Design pattern implementation in Java and AspectJ. In: Proceedings of the Object-Oriented Programming Systems Language and Applications (OOPSLA 2002), Seattle, Washington, November 2002, pp. 161–173.
Hunleth, F., Cytron, R., Gill, C., 2001. Building customizable middleware using aspect oriented programming. In: Proceedings of the Workshop on Advanced Separation of Concerns in Object-Oriented Systems, Tampa, Florida, October 2001.
IONA Orbix/E Datasheet. http://www.iona.com/whitepapers/orbixe-DS.pdf.
Jboss AOP web site. http://labs.jboss.com/jbossaop/.
Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J., 1997. Aspect-oriented programming. In: Proceedings of the European Conference on Object-Oriented Programming, Jyvaskyla, Finland, July 1997, pp. 220–242.
Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G., 2001. An overview of AspectJ. In: Proceedings of the European Conference on Object-Oriented Programming, Budapest, Hungary, June 2001, pp. 327–353.
Lee, J., Bae, D., 2004. An aspect-oriented framework for developing component-based software with the collaboration-based architectural style. Journal of Information and Software Technology 46 (2 (February)), 81–97.
Malek, S., Mikic-Rakic, M., Medvidovic, N., 2005. A style-aware architectural middleware for resource-constrained, distributed systems. IEEE Transactions on Software Engineering 32 (3 (March)), 256–272.
Malek, S., Seo, C., Ravula, S., Petrus, B., Medvidovic, N., 2007. Reconceptualizing a family of heterogeneous embedded systems via explicit architectural support. In: Proceedings of the International Conference on Software Engineering (ICSE 2007), Minneapolis, Minnesota, May 2007, pp. 591–601.
Malek, S., 2008a. Dealing with the crosscutting structure of software architectural styles. In: Proceedings of the International Computer Software and Applications Conference, Turku, Finland, July 2008, pp. 385–392.
Malek, S., 2008b. Effective realization of software architectural styles with aspects. In: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA 2008), working session track, Vancouver, BC, February 2008, pp. 313–316.
Medvidovic, N., Oreizy, P., Robbins, J.E., Taylor, R.N., 1996. Using object-oriented typing to support architectural design in the C2 style. In: Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 1996), San Francisco, CA, October 1996, pp. 24–32.
Medvidovic, N., Dashofy, E.M., Taylor, R.N., 2003. The role of middleware in architecture-based software development. International Journal of Software Engineering and Knowledge Engineering 13 (4 (August)), 367–393.
Mehta, N., Medvidovic, N., Phadke, S., 2000. Towards a taxonomy of software connectors. In: Proceedings of the International Conference on Software Engineering, Limerick, Ireland, June 2000, pp. 178–187.
Nanbor, W., Balasubramanian, K., Gill, C., 2002. Towards a real-time Corba component model. In: Proceedings of the OMG Workshop on Embedded and Real-Time Distributed Object Systems, Washington, DC, July 2002.
Navasa, A., Perez, M.A., Murillo, J.M., Hernandez, J., 2002. Aspect-oriented software architecture: a structural perspective. In: Proceedings of the Workshop on Early Aspect, Enschede, The Netherlands, April 2002.
Navasa, A., Perez, M.A., Murillo, J.M., 2005. Aspect modeling at architecture design. In: Proceedings of the 2nd European Workshop on Software Architecture (EWSA 2005), Pisa, Italy, June 2005, pp. 41–58.
Navasa, A., Pérez-Toledano, M.A., Murillo, J.M., 2009. An ADL dealing with aspects at software architecture stage. Journal of Information and Software Technology 51 (2 (February)), 306–324.
Object Management Group's Model-Driven Architecture. http://www.omg.org/mda/.
Pérez, J., Ali, N., Carsí, J.A., Ramos, I., 2005. Dynamic evolution in aspect-oriented architectural models. In: Proceedings of the European Workshop on Software Architecture, Pisa, Italy, June 2005, pp. 59–76.
Pérez, J., Ali, N., Carsí, J.A., Ramos, I., Álvarez, B., Sánchez, P., Pastor, J.A., 2008. Integrating aspects in software architectures: PRISMA applied to robotic tele-operated systems. Journal of Information and Software Technology 50 (9–10 (August)), 969–990.
Pérez, J., Ali, N., Carsí, J.A., Ramos, I., 2006. Designing software architectures with an aspect-oriented architecture description language. In: Proceedings of the International Symposium on Component-Based Software Engineering (CBSE 2006), Vasteras, Sweden, June 2006, pp. 123–138.
Perry, D.E., Wolf, A.L., 1992. Foundations for the study of software architectures. ACM SIGSOFT Software Engineering Notes 17 (4 (October)), 40–52.
Prose web site. http://prose.ethz.ch/.
Rashid, A., Moreira, A., Araújo, J., 2003. Modularisation and composition of aspectual requirements. In: Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD 2003), Boston, USA, March 2003, pp. 11–20.
Real-time Corba with TAO (The ACE ORB). http://www.cs.wustl.edu/~schmidt/TAO.html.
Sant'Anna, C., Garcia, A., Chavez, C., Lucena, C., von Staa, A.V., 2003. On the reuse and maintenance of aspect-oriented software: an assessment framework. In: Proceedings of the Brazilian Symposium on Software Engineering, Manaus, Brazil, October 2003.
Schmidt, D.C., 2006. Model-driven engineering. IEEE Computer 39 (2 (February)), 25–31.
Schmidt, D.C., Gokhale, A., Natarajan, B., Neema, S., Bapty, T., Parsons, J., Gray, J., Nechypurenko, A., Wan, N., 2002. CoSMIC: an MDA generative tool for distributed real-time and embedded component middleware and applications. In: Proceedings of the OOPSLA 2002 Workshop on Generative Techniques in the Context of Model-Driven Architecture, Seattle, WA, November 2002.
Shaw, M., DeLine, R., Klein, D.V., Ross, T.L., Young, D.M., Zelesnik, G., 1995. Abstractions for software architecture and tools to support them. IEEE Transactions on Software Engineering 21 (4 (April)), 314–335.
Shaw, M., Garlan, D., 1996. Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, Upper Saddle River, NJ.
Silaghi, R., Strohmeier, A., 2005. Parallax—an aspect-enabled framework for plugin-based MDA refinements towards middleware. In: Beydeda, S., Book, M., Gruhn, V. (Eds.), Model-Driven Software Development. Research and Practice in Software Engineering, vol. II. Springer-Verlag, pp. 237–267.
Sousa, J.P., Garlan, D., 2002. Aura: an architectural framework for user mobility in ubiquitous computing environments. In: Proceedings of the IEEE/IFIP Working Conference on Software Architectures, Montreal, Canada, August 2002, pp. 29–43.
Spread Toolkit web site. http://www.spread.org/.
Taylor, R.N., Medvidovic, N., Anderson, K.M., Whitehead, E.J., Robbins, J.E., Nies, K.A., Oreizy, P., Dubrow, D.L., 1996. A component- and message-based architectural style for GUI software. IEEE Transactions on Software Engineering 22 (6 (June)), 390–406.
Vanegas, R., Zinky, J.A., Loyall, J.P., Karr, D., Schantz, R.E., Bakken, D.E., 1998. QuO's runtime support for quality of service in distributed objects. In: Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open-Distributed Processing, The Lake District, United Kingdom, September 1998, pp. 207–222.
Wadsack, J.P., Jahnke, J.H., 2002. Towards model-driven middleware maintenance. In: Proceedings of the OOPSLA 2002 Workshop on Generative Techniques in the Context of Model-Driven Architecture, Seattle, WA, November 2002.
Zhang, C., Jacobsen, H.A., 2003a. Quantifying aspects in middleware platforms. In: Proceedings of the International Conference on Aspect-Oriented Software Development, Boston, MA, March 2003, pp. 130–139.

Zhang, C., Jacobsen, H.A., 2003b. Re-factoring middleware systems: a case study. In: Proceedings of the International Symposium on Distributed Objects and Applications (DOA 2003), Catania, Sicily, November 2003, pp. 1243–1262.

Sam Malek is an Assistant Professor in the Department of Computer Science at George Mason University (GMU). He is also a faculty member of the C4I Center at GMU. Malek's general research interests are in the field of software engineering, and to date his focus has spanned the areas of software architecture, distributed and embedded software systems, middleware, autonomic computing, service-oriented architectures, and quality of service analysis. The underlying theme of his research has been to devise techniques and tools that aid with the construction, analysis, and maintenance of large-scale distributed, embedded, and pervasive software systems. His research has been funded by NSF, US Army, and SAIC. Malek received his Ph.D. in 2007 from the Computer Science Department at the University of Southern California (USC). His dissertation research was nominated by USC for the final round of the ACM Doctoral Dissertation Competition in 2007. He also received an M.S. degree in Computer Science in 2004 from USC, and a B.S. degree in Information and Computer Science cum laude in 2000 from the University of California, Irvine. Malek is the recipient of numerous awards, including USC Viterbi School of Engineering Fellow Award in 2004, and the USC Computer Science Outstanding Student Research Award in 2005. He is a member of the ACM, the ACM SIGSOFT, and the IEEE.

Harshini Ramnath Krishnan received an M.S Degree in Computer Science from George Mason University in 2010 and a B.S in Computer Science from Anna University, India, in 2008. Initially, an Intern in VeriSign Inc with the Platform Product Development group, she is now an Engineer in the Rapid Prototyping team of VeriSign Inc. Her research interests span the areas of aspect-oriented software development, architectural styles, and software architecture for distributed systems.

Jaya Srinivasan received an M.S. degree in Software Engineering from George Mason University in 2010 and a B.E degree in Electronics and Communications Engineering from University of Madras, India, in 2003. Previously, she has worked as a Programmer Analyst for Cognizant Technology Solutions in Chennai, India, where she developed several software products. Jaya was awarded Outstanding Academic Achievement award in her M.S program. Her research interests are in software architecture for distributed systems, middleware facilities for architectural implementation, and architectural styles.