Dealing with the Crosscutting Structure of Software Architectural Styles

Sam Malek
Department of Computer Science
George Mason University
Fairfax, VA 22030-4444 U.S.A.
smalek@gmu.edu

Abstract

Architecture-based software development is the implementation of a software system in terms of its architectural constructs (e.g., components, connectors, ports). It has been shown as an effective approach to realizing and managing the architecture of large-scale software systems. Several techniques and tools have been developed that are intended to aid with the architecture-based development of software systems. While these approaches provide adequate implementation support for some aspects of software architectures, they often lack sufficient support for implementing and enforcing the system's software architectural style. In this paper, we argue that the lack of sufficient support for architectural styles is a by-product of its crosscutting structure. In turn, this makes it impossible to realize styles using the object-oriented programming methodology. We present a new approach to implementing architectural styles that is based on the aspect-oriented programming paradigm.

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) [15]. Software *architectural styles* (e.g., publish-subscribe, client-server, pipe-and-filter) further codify structural, behavioral, interaction, and composition guidelines that are likely to result in software systems with desired properties [15].

For the software architectural models and guidelines to be truly useful in a development setting, they must be accompanied by support for their implementation [14]. 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 *architectural middleware* solutions [2][8][14], which provide native implementation-level support for the architectural concepts.

The state-of-the-art architectural middlewares provide implementation support for some architectural concepts (e.g., components, ports, events), but they do not provide adequate support for the others, most importantly architectural styles. In fact, most commercial and architectural middleware solutions ignore, mimic, or at best assume a particular architectural style. Therefore, forcing the software architect 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.

We argue that the lack of sufficient support for implementing architectural styles is due to the crosscutting structure of styles. Architectural styles often prescribe rules and guidelines that impact the behavior and structure of all the other architectural concepts and constructs. Therefore, unlike any other architectural concept, architectural styles cannot be effectively abstracted and implemented using the traditional object-oriented programming language 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 [8].

In this paper, we propose a new approach for implementing architectural styles that is based on the *aspect-oriented programming (AOP)* paradigm [6]. We provide an overview of our approach on top of an architectural middleware platform, called Prism-MW [8]. Our approach allows for modularized

representation of stylistic concerns, which are weaved into the middleware's implementation at compile time. In turn, this aids system understanding, 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.

The remainder of the paper is organized as follows. Section 2 motivates our research in response to the shortcomings of the current technologies. Section 3 describes our overall approach. Section 4 provides an overview of Prism-MW, an architectural middleware platform that we have leveraged to implement our approach. Section 5 demonstrates the crosscutting structure of styles using Prism-MW. Section 6 presents an example style that we have implemented using our approach. Section 7 describes our experience with the validation of this research. Section 8 highlights the related work. Finally, the paper concludes with a summary of our contributions, and future work.

2. Motivation

Software engineers have often relied on the facilities and abstractions provided by a middleware platform to aid with the implementation of complex large-scale software system. Ideally, to deal with the complexity of building large-scale software systems, the engineers should be able to construct a software architecture that best satisfies the system's requirements, irrespective of the candidate middlewares that could be used for the implementation. This is consistent with the Object Management Group's two stage modeling approach: Platform Independent Model (PIM) and Platform Specific Model (PSM) [12]. However, in practice, the engineers are constrained significantly by a subset of architectural choices, including architectural styles, that can effectively be realized in a given middleware platform. In fact, previous researchers have realized that deriving PSM from PIM is often very challenging and sometimes not even feasible [3]. This is largely due to the rigidity of the target implementation platforms that are often developed with certain intrinsic assumptions about the structural and behavioral characteristics of the applications that will be deployed on top of them. One objective of our research has been to make the mapping from PIM to PSM a more straightforward process. To this end we have previously developed an extensible architectural middleware that provides one-to-one mapping between constructs in the architectural diagrams and their implementation counterparts. In this paper, we focus on the next logical problem: mapping the stylistic decisions to their implementation counterparts.

An important shortcoming of the most commonly used state-of-the-art middlewares (e.g., Java RMI, CORBA, DCOM, and Microsoft .net) is the lack of sufficient support for architectural styles. They often provide support for some aspects of architectural styles, such as the communication style (e.g., synchronous vs. asynchronous), but do not support others. For example, the necessary implementation-level facilities for some key elements of software architectures are often missing─explicit architecture-level connectors are usually distributed (and thus "lost") across different implementation-level modules as combinations of method calls, shared memory, network sockets, and other facilities in the middleware [11]. Moreover, stylistic rules and constraints on the valid configurations of a software system are rarely ensured and enforced by the middleware. Thereby, it is extremely difficult to verify the fidelity of the constructed software system with respect to the system's intended architecture. An important goal in our research has been to provide complete support for arbitrary complex styles in a given middleware. This is in contrast to previous well-known architectural middlewares that either do not provide stylistic support at all (e.g., ArchJava [2]) or are targeted specifically to a predetermined style (e.g., C2 Framework [9]).

Finally, one approach to providing support for several styles in a middleware is to parameterize it, such that it can be configured to behave according to the rules of a style at run-time. This is in fact the approach that was taken in an earlier work [8]. However, we argue that this is not a viable solution for any general purpose middleware for the following reasons. Firstly, there are many well-known architectural styles and infinitely more domain-specific patterns and hybrid styles that an architect may opt to use. Predicting those a priori (i.e., during the construction of the middleware) is infeasible. Secondly, providing implementation support for any substantive set of styles inevitably makes the middleware bulky and has a significant impact on its performance. One of our key objectives in this research has been to develop a technique for efficiently configuring a middleware to support arbitrary, potentially domain specific, architectural styles.

3. Approach

Figure 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 on heterogeneous platforms; the abstraction facilities provided by the virtual machine are leveraged by the middleware's *architectural constructs*, and application logic that lay on top of it. In this research, we are more interested in the top layer, i.e., *architectural support* layer. As mentioned 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 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 Section 2 most middlewares do not provide sophisticated support for architectural styles by default. We implement the stylistic concerns of the middleware in one or more aspects, which when weaved with the generic middleware constructs provide style-specific support. For example, as depicted in Figure 1, to provide support for the Client-Server style, the *Client-Server Aspect* is weaved with the middleware to generate a more 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). In the remainder of the paper, we describe a middleware platform and a style that we have implemented support for in the middleware using the above approach.

4. Overview of Prism-MW

In this section, we provide an overview of Prism-MW, an architectural middleware platform that we have leveraged 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 [8]. Prism-MW is a suitable platform for describing and applying our approach: 1) Prism-MW provides support for straightforward one-to-one mapping of architectural constructs to their implementations, which is ideal for demonstrating the crosscutting impact of styles; and 2) Prism-MW is open source, which allows us to weave the style-specific code with the middleware's implementation.

4.1 Core design

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. Figure 2 shows a partial class design view of Prism-MW. The shaded classes constitute the middleware core, which represents a minimal subset of Prism-MW that enables implementation and execution of architectures in a single address space. Essentially Figure 2 corresponds to the "Generic Architectural Support" layer of the middleware stack (depicted on

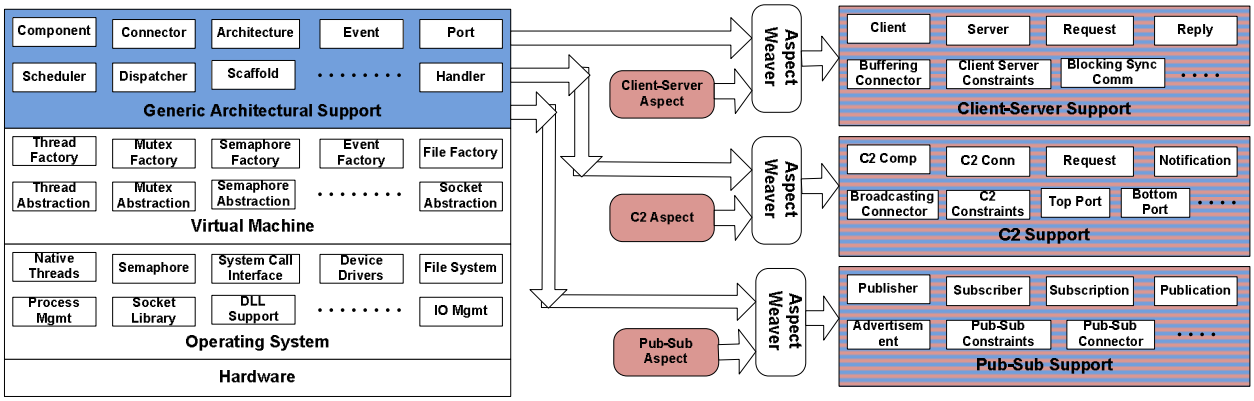

Figure 1. Overall approach.

the left hand side of Figure 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 runtime.

*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 per-formed an operation.

*Port*s are the loci of interaction in an architecture. A *link* between two ports is made by *weld*ing them together. A port can be welded 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. Like components, each connector can have an arbitrary number of attached ports. Components attach to connectors by creating a link between a component port and a single connector port. In order to support the needs of dynamically changing applications, each Prism-MW component or connector is capable of adding or removing ports at runtime [8].

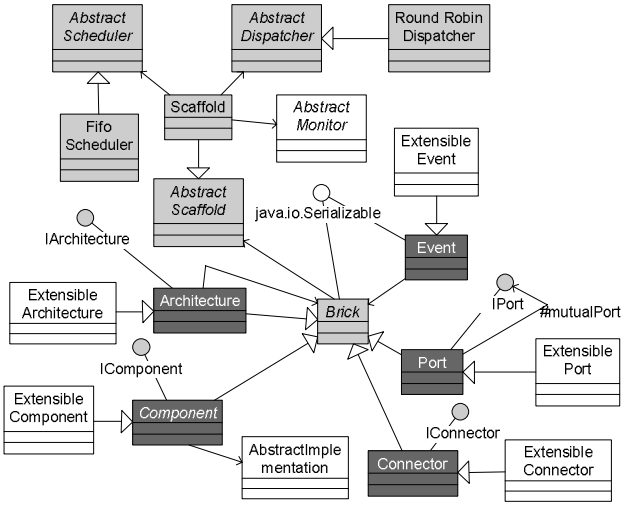

Figure 2. Abridged UML class design view of Prism-MW. Middleware core classes are highlighted.

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 default implementations of *AbstractScheduler* and *AbstractDispatcher*: *FIFOScheduler* and *RoundRobinDispatcher*, respectively. 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 [8].

4.2 Using the middleware

Prism-MW's core provides the necessary support for developing arbitrarily complex applications, so 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 takes is to subclass 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, Figure 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. Figure 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 [11].

4.3 Under the hood

Figure 4 depicts the most commonly used event routing mechanism in Prism-MW, as implemented in *FIFOScheduler*, and *RoundRobingDipatcher*. A pool of shepherd threads (implemented in *RoundRobinDispatcher* class) is associated with an event queue (implemented in *FIFOScheduler*). They are used to handle events sent by any component in a given address space. The size of the thread pool and queue is parameterized and, hence, adjustable. To process an event, a shepherd thread removes the event from the head of the queue. For local communication, the shepherd thread is run through the connector attached to the sending component; the connector dispatches the event to relevant components using the same thread (see Figure 4). If a recipient component generates further events, they are added to the tail of the event queue; different threads are used for dispatching those events to their intended recipients. Routing the events in a distributed system is achieved very similarly, via a special type of Prism-MW port that is capable of communicating remotely. For brevity, the details of Prism-MW's support for distributed communication and other routing mechanisms that are not directly relevant to this research have been omitted. Interested reader should refer to [8].

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);

        // attach the communication ports
        Port guiPort = new Port(gui);
        Port adderPort = new Port(adder);
        Port connGuiPort = new Port(conn);
        Port connAdderPort = new Port(conn);

        // establish the interconnections
        arch.weld(guiPort, connGuiPort);
        arch.weld(connAdderPort, adderPort);

        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);
    }...
}
```

Figure 3. Illustration of application implementation fragments.

5. Crosscutting impact of style

We believe effective support for architectural style in a middleware platform requires at least:

- 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 [18]);
- 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-and-filter style).

The above discussion suggests that architectural styles could 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. We could have selected any other middleware solution for this purpose. However, as mentioned earlier, Prism-MW's extensive separation of concern and modularized implementation of architectural constructs, allow us to demonstrate the crosscutting impact of styles most effectively. We believe the

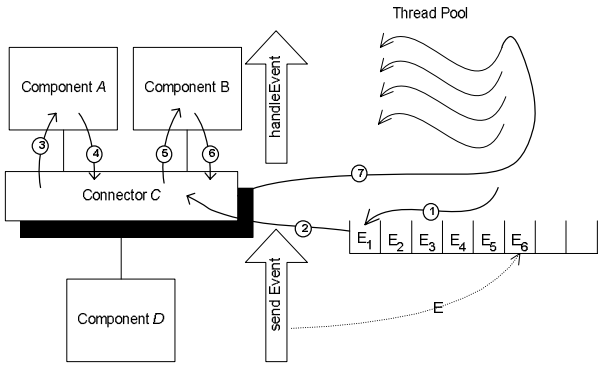

Figure 4 Event dispatching in Prism-MW for a single address space. Steps (1)-(7) are performed by a single shepherd thread.

lessons learned here are more generally applicable.

By default Prism-MW's core is style agnostic, and to provide support for an architectural style, one would have to modify Prism-MW. There are two ways of doing this: 1) leverage Prism-MW's extensible classes to override the core behavior (shown in Figure 2 and discussed in [8]), or 2) modify the implementation of the core classes directly. Note that neither approach allows us to represent and implement a style in a modularized and decoupled manner, nor do they allow for the implementation of a domain-specific style by the middleware users. For the clarity of exposition, we describe the changes to the middleware using the second approach:

1. As mentioned in our first requirement earlier, before we can enforce the stylistic rules and constraints, we need to be able to distinguish the style of each architectural construct. One (and probably the most trivial) approach is to define a new variable in *Brick* that identifies the style of an architectural object. The value of this variable corresponds to a given architectural style element, e.g., Client, Server, Pipe, Filter, and so on.
2. As mentioned in our second requirement earlier, we may need to modify the behavior of architectural constructs. 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 [18]. 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, a Client blocks after it sends a request to a Server and unblocks 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.
3. 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 core facilities (dark gray classes in Figure 2). It also shows that changes are dispersed among the various parts of the middleware's implementation. The situation is exacerbated with middlewares that do not provide the same level of support for implementing software architectures as Prism-MW. In fact, finding the classes that need to be modified for a particular characteristic of a style is fairly straightforward in Prism-MW. This is not necessarily the case with the more traditional middlewares that do not provide explicit support for some of the architectural concepts (e.g., connector, port, and configuration).

6. Stylistic aspect

In this section, we provide a concrete example of an architectural style that we have implemented on top of Prism-MW using AOP paradigm. The steps for providing implementation support for a new architectural style are as follows:

1. define a new aspect for each architectural style;
2. define the new style-specific facilities and properties using the aspect's inter-type declaration; and
3. override or refine the middleware's default behavior using the aspect's pointcut and advice constructs.

Below we detail the approach for providing implementation support for the C2 [18] architectural style in Prism-MW using AspectJ [1].

Figure 5 shows portion of an aspect that we have developed for supporting the C2 style in Prism-MW. Line 5 of the code snippet shows the ability to augment the architectural elements with new properties. In this case, we are using aspect's inter-type declaration capability to add a new member variable to each *Brick* object of Prism-MW, which allows us to determine its architectural style. Recall from Section 4.1 that all architectural elements (e.g., components, connectors, ports) in Prism-MW extend *Brick*. Thus, with the

newly added variable we are able to determine the style of each architectural construct, which is required for enforcing the stylistic rules and constraints.

C2 style does not allow two components to be connected directly (i.e., without being mediated by explicit connectors) [18]. Therefore, the default behavior of the *Architecture*'s *weld* method, which does not enforce any constraints, needs to be modified. Recall from Section 4.2 that the *Architecture*'s *weld* method is used to connect components and connectors by associating their corresponding ports with one another (shown in Figure 3). Lines 7-21 in Figure 5 show a pointcut for picking out join points that are calls to the *weld* method of the *Architecture* object, and the corresponding advice that gets executed. Basically, the *before* advice is executed before the *architecture*'s *weld* method is executed. In this advice, we have implemented the necessary checks to enforce that two C2 components are not connected directly to one another (lines 13-15 show the condition statement).

C2 connectors broadcast *Request* events on their *Request* (top) ports, and *Notification* events on their *Notification* (bottom) ports [18]. Recall from Section 5 that Prism-MW connector's *handle* method implements the default routing, which in our version of Prism-MW just broadcasts events on its ports (shown in Figure 4). We override the connector's routing via the pointcut and advice shown in Lines 23-33 of Figure 5. The pointcut (lines 24-25) picks out join points that are calls to the connector's *handle* method. The corresponding *around* advice gets executed in place of the connector's *handle* method, and provides support for routing events according to the C2 guidelines.

```
1. import Prism.core.*;
2.
3. public aspect C2Style {
4.
5.  public String Brick.archStyle;
6.
7.  void before (Architecture arch, Port p1, Port p2):
8.    call (void Architecture.weld(Port,Port))
9.    && target(arch) && args(p1, p2)
10.   {
11.      Brick b1 = p1.getParentBrick();
12.      Brick b2 = p2.getParentBrick();
13.      if (b1 instanceof Component && b2 instanceof Component
14.         && b1.archStyle.equals("C2Comp")
15.         && b2.archStyle.equals("C2Comp"))
16.       {
17.          System.out.println
18.            ("C2 does not allow conneting two components");
19.          System.exit(0);
20        }
21.    }
22.
23. void around (Connector conn, Event e):
24.   call (void Connector.handle(Event))
25.   && target (conn)&& args (e)
26.   {
27.      for (int i=0; i < conn.ports.size(); i++)
28.      {
29.         Port thisPort = (Port)conn.ports.elementAt(i);
30.         if (thisPort.getPortType() == e.eventType)
31.             thisPort.handle(e);
32.       }
33.    }
     ...
     }
```

Figure 5. Code snippet of an aspect that implements rules and constraints of the C2 architectural style.

Due to space constraint, we do not provide the full detail of implementation support for the C2 style. However, the above examples demonstrate that aspects can be effectively leveraged in providing support for the stylistic concerns in middlewares. Furthermore, the resulting style-specific code is both localized and modularized, which in turn improves the system's ability to evolve, and aids with system understanding.

7. Validation

To validate the research, we have implemented more than twenty architectural styles from [5] using our approach. In the process, we have noticed that the amount of the effort required to implement a new style decreased over time. That is as more styles were developed, the level of code reuse from previously developed styles increased. This is attributed to the fact that sometimes seemingly different architectural styles have common traits with one another. For example, to implement the aspect that realizes a layered-client-server style, we were able to leverage a significant portion of the two aspects that realized the layered, and the client-server style.

8. Related work

Several previous works have developed technologies for architecture-based software development. One of these is Prism-MW that was discussed earlier. Below we provide an overview of some of the other prominent technologies.

ArchJava [2] is an extension to Java that unifies software architecture with implementation, ensuring that the implementation conforms to architectural constraints. 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 [17] is an architectural style and supporting middleware for ubiquitous computing applications with a special focus on user mobility, context awareness, and context switching. Similar to Prism-MW, Aura has explicit, first-class connectors. However, Aura does not provide support for specifying a new architectural style that could be supported by the middleware.

Middlewares such as TAO [16], Orbix/E [13], .Net, and MobiPADS [4] provide partial support for architectural abstractions in the form of explicit components. However, none of these middleware solutions support multiple, explicit, and/or tailorable software connectors. Furthermore, none of them support explicit architectural styles, thus clearly distinguishing our work from them. The styles in all of the surveyed technologies are implicit and mostly fall within the distributed objects category.

Another area of related work has been the usage of aspects in realizing the design decisions. Most prominently, in [7] a technique is presented for aspect-oriented development of design patterns. Our work is different from this work in several ways. Firstly, our work is geared towards the implementation of software architectural styles, as opposed to design patterns. Secondly, our approach deals with realizing support for styles in middleware solutions, as opposed to traditional programming languages.

9. Conclusion

Architectural middlewares have been shown as an effective approach to implementing a system's software architecture. However, due to the crosscutting structure of styles, there has been a lack of adequate support for implementing architectural styles in most middleware solutions, including architectural middlewares. In this paper, we demonstrated the crosscutting impact of styles on an architectural middleware platform. We also presented a new approach to implementing architectural styles that is based on the aspect-oriented programming paradigm. Aspects allow for modularized and localized implementation of stylistic support in middlewares. Furthermore, they allow an informed engineer to modify the default behavior of a middleware by implementing support for an arbitrary, possibly domain specific, architectural style. As part of our future work, we plan to extend our work to other architectural styles, and middleware solutions. An interesting avenue for future study is to determine the feasibility of providing support for a given hybrid style (e.g., layered-client-server) via composition (e.g., abstract-aspect, sub-aspect) of several basic style aspects (e.g., layered aspect, client-server aspect).

10. References

[1] AspectJ web site. http://www.eclipse.org/aspectj/

[2] J. Aldrich, et al. ArchJava: Connecting Software Architecture to Implementation. *Int'l Conference on Software Engineering*, Orlando, Florida, May 2002.

[3] G. Caplat, et al. Model Mapping in MDA. *Workshop on Software Model Engineering*, Dresden, Germany, Oct. 2002.

[4] A. Chan, et al. MobiPADS: A Reflective Middleware for Context-Aware Mobile Computing. *IEEE Transactions on Software Engineering*, Dec. 2003.

[5] R. Fielding. Architectural Styles and the Design of Network-Based Software Architectures. *PhD thesis, Univ. of California Irvine*, June 200.

[6] G. Kiczales, et al. Aspect-Oriented Programming. *European Conference on Object-Oriented Programming*, Jyvaskyla, Finland, July 1997.

[7] J. Hannemann, and G. Kiczales. Design pattern implementation in Java and aspectJ. *Object-Oriented Programming Systems Language and Applications*, Seattle, Washington, 2002.

[8] S. Malek, et al. A Style-Aware Architectural Middleware for Resource-Constrained, Distributed Systems. *IEEE Transactions on Software Engineering*, March 2005.

[9] N. Medvidovic, et al. Reuse of Off-the-Shelf Components in C2-Style Architectures. *International Conference on Software Engineering*, Boston, MA, May 1997.

[10] Shelf Components in C2-Style Architectures.

[11] N. Mehta, et al. Towards a Taxonomy of Software Connectors. *International Conference on Software Engineering*, Limerick, Ireland, June 2000.

[12] Object Management Group's Model Driven Architecture. http://www.omg.org/mda/

[13] IONA Orbix/E Datasheet, http://www.iona.com/whitepapers/orbixe-DS.pdf

[14] M. Shaw, et al. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, April 1995.

[15] M. Shaw, et al. Software Architecture: Perspectives on an Emerging Discipline. *Prentice Hall*, 1996.

[16] Real-time Corba with TAO (The ACE ORB). http://www.cs.wustl.edu/~schmidt/TAO.html

[17] J. P. Sousa, and D. Garlan. Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments. *Working International Conference on Software Architectures*, Montreal, Canada, 2002.

[18] R. N. Taylor, et al. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, June 1996.