Effective Realization of Software Architectural Styles with Aspects

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, making it impossible to realize styles using the object-oriented programming methodology. We propose 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) [7]. 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 [7].

For the software architectural models and guidelines to be truly useful in a development setting, they must be accompanied by support for their implementation[6]. However, there is a gap between the high-level architectural concepts and the low-level programming language constructs that are used for implementing the architectural models. 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 [2,5,6], which provide native implementation-level support for the architectural concepts.

While the state-of-the-art architectural middlewares provide support for some of the architectural concepts (e.g., components, ports, events), they do not provide adequate support for others, most importantly architectural styles. In fact, most commercial and architectural middleware solutions either 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 can not 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.

In this paper, we propose a new approach for implementing architecture styles that is based on the *aspect-oriented programming (AOP)* paradigm [3]. We provide an overview of our approach on top of an architectural middleware platform, called Prism-MW [5]. Our approach allows for modularized representation of stylistic concerns, which are weaved into the middleware's implementation at compile time. In turn, aiding 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 provides an overview of Prism-MW, an architectural middleware platform that we have leveraged in describing our approach. Section 3 demonstrates the crosscutting structure of styles using Prism-MW. Section 4 discusses our approach to realizing styles with aspects. Section 5 highlights the related work. Finally, the paper concludes with a summary of our contributions, and future work.

2. Prism-MW

In this section, we provide an overview of Prism-MW, a middleware platform that we have leveraged in describing our approach. Prism-MW [5] is an architectural middleware platform that provides implementation-level support for architectural constructs in an extensible, efficient, and scalable manner. 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.

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 1 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.

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 component to perform an operation or a *reply* that a sender component has performed an operation.

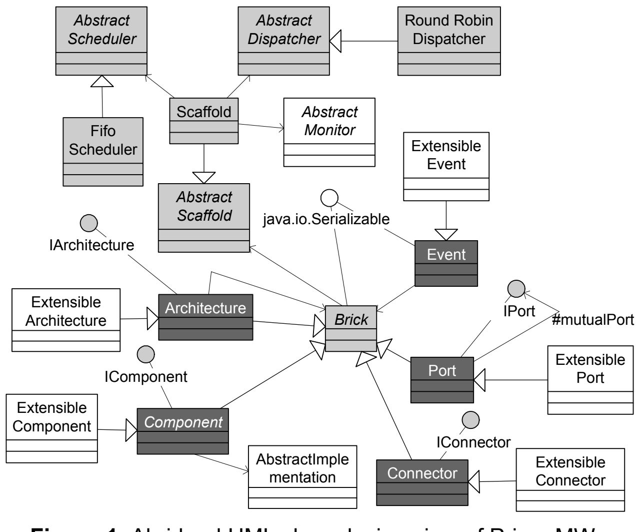

Figure 1. Abridged UML class design view of Prism-MW. Middleware core classes are highlighted.

*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.* An event placed on one port is forwarded to the port linked to it in the following manner: request events are forwarded from request ports to reply ports, while reply events are forwarded in the opposite direction.

*Component*s perform computations in an architecture and may maintain their own internal state. A component is dynamically associated with its application-specific functionality via a reference to the *AbstractImplementation* 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 [5].

Finally, Prism-MW provides support for event dispatching and queuing, monitoring, and reflection facilities that the developer can associate with the system's architecture. The developer provides the application-specific logic by implementing the *AbstractImplementation* class.

3. Crosscutting structure of architectural style

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

1. the ability to distinguish among different architectural elements of a given style (e.g, distinguishing `Clients` from `Servers` in the client-server style);
2. 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 [9]);
3. 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 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 1 and discussed in [5]), or 2) modify the implementation of the core classes directly. Note that neither approaches allow us to represent and implement a style in a modularized and decoupled manner. For the clarity of exposition, we describe the changes to the middleware using the second approach:

1. As mentioned in our first requirement above, 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 above, 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 [9]. 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 the publish-subscribe style exchange `Advertisements`, `Subscriptions`, and `Notifications`.
3. As mentioned in our third requirement above, 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 1). 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, configuration).

4. Architectural style aspect

In this section, we present a new approach for implementing architectural styles that is based on 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 [9] architectural style in Prism-MW, and using AspectJ [1].

Figure 2 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 2 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) [9]. Therefore, the default behavior of the *Architecture*'s *weld* method, which does not enforce any constraints, needs to be modified. Recall from Section 3 that the *Architecture*'s *weld* method is used to connect components and connectors by associating their corresponding ports with one another. Lines 7-21 in Figure 2 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 [9]. Recall from Section 3 that Prism-MW connector's *handle* method implements the default routing, which in our version of Prism-MW just broadcasts events on

```
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 2. Code snippet of an aspect that implements rules and constraints of the C2 architectural style.

its ports. We override the connector's routing via the pointcut and advice shown in Lines 23-33 of Figure 2. 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 provide support for routing events according to the C2 guidelines.

Due to space constraint, we cannot provide the full detail of implementation support for the C2 style. However, the above example demonstrates 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.

5. 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 two 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 [8] 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.

Another area of related work has been the usage of aspects in realizing the design decisions. Most prominently, in [4] 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.

6. 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. In this paper, we demonstrated the crosscutting impact of styles on an architectural middleware platform. We also provided an overview of 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 hybrid styles (e.g., Layered-Client-Server) via composition (e.g., abstract-aspect, sub-aspect) of several basic style (e.g., Layered, Client-Server) aspects.

7. References

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

[2] J.Aldrich, et. al. ArchJava: Connecting Software Architecture to Implementation. *ICSE*, Orlando, Florida, May 2002.

[3] G. Kiczales, et. al. Aspect-Oriented Programming. *ECOOP*, Jyvaskyla, Finland, July 1997.

[4] J. Hannemann, and G. Kiczales. Design pattern implementation in Java and aspectJ. *OOPSLA*, Seattle, Washington, 2002.

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

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

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

[8] J. P. Sousa, and D. Garlan. Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments. *WICSA*, Montreal, Canada, 2002.

[9] Richard N. Taylor, et. al. A Component- and Message-Based Architectural Style for GUI Software. *IEEE TSE*, June 1996