On the Role of Style in Selecting Middleware and Underwear

Elisabetta Di Nitto
CEFRIEL – Politecnico di Milano
Via Fucini, 2
20133 Milano, Italy
dinitto@elet.polimi.it

David S. Rosenblum
University of California, Irvine
Dept. of Information & Computer Science
Irvine, CA 92697-3425 USA
dsr@ics.uci.edu

Abstract

Middleware infrastructures are becoming a pervasive part of many distributed software systems. Wileden and Kaplan argue that middleware, like underwear, should not be the center of attention but should instead be kept hidden from public view, and it should never constrain or dictate what is publicly visible. These are admirable goals, yet the architects of distributed software systems must nevertheless recognize and account for the intimate relationship between middleware and the systems that use them. In particular, it is useful to view middleware infrastructures as inducing *architectural styles*, in the sense that they embody structural and behavioral constraints imposed on the systems that use them. Defining these styles and identifying the important relationships between them will allow architects to exploit these styles in a way that helps them defer, as long as possible, those architectural decisions that limit middleware choices, and to develop architectures that can accommodate the widest range of middleware. Otherwise, unattractive middleware choices may creep up on an architect in an annoying way.

1 Introduction

Software developers are beginning to make extensive use of *middleware infrastructures* to facilitate component interoperability in large-scale distributed systems. There is an increasingly large number of middleware infrastructures from which to choose, including systems based on middleware standards such as CORBA [6] and Enterprise JavaBeans [7]; proprietary commercial middleware products such as TIBCO's TIB®/Rendezvous™ publish/subscribe software and Talarian's SmartSockets® middleware; and research systems such as JEDI [3] and SIENA [2]. The fact that middleware plays a key role in facilitating interoperability means that it ends up being a critical and pervasive element of any system it supports, fundamentally affecting the architecture, implementation and evolution of the system.

Wileden and Kaplan recently argued that middleware should be treated like underwear [8]. In brief, they said that middleware, like underwear, should not be the center of attention but should instead be kept hidden from public view, and it should never constrain or dictate what is publicly visible. We agree that this is an admirable ideal to strive for, yet as a practical matter one cannot deny the existence of an intimate relationship between middleware and the systems that use them. Middleware, like underwear, must be well matched to the things that clothe it. A particular middleware, like a particular item of underwear, has certain properties, imposes certain

Figure 1. A pipe and filter architecture.

constraints, and achieves certain effects that make it well-suited for some situations and ill-suited for others. Indeed, Wileden and Kaplan note that

> *[N]o one style can be expected to meet all needs. Just as football uniforms and tuxedos are best worn with different styles of underwear, different software applications have different middleware needs [8].*

We note further that while the football player must make his or her underwear choices before suiting up, middleware is often not the first item selected to outfit a software system. Other application elements and properties may be decided upon first, with the choice of middleware postponed to the final stages of implementation development. There are numerous sound engineering reasons for postponing middleware decisions, such as maintaining a high level of abstraction and a separation of concerns in early stages of design. But there is no guarantee that an arbitrarily chosen middleware will be tailor-made for the system and will be slipped on with ease. Even if the middleware is selected at the outset of a project (perhaps for economic reasons, or to maintain compatibility with an existing system), the architecture must develop in a way that allows it to tastefully accommodate the middleware.

There arises then a tension between the architectural properties and constraints of an application under development and the properties and constraints of the middleware that will be selected for the application. In other words, a middleware induces a particular *architectural style* in the systems that use it, imposing a variety of structural constraints on the configuration of system components and behavioral constraints on the interactions that take place through the middleware. As more and more decisions are *explicitly* made about the architecture of a system, more and more choices are *implicitly* made about the middleware that will be selected.

We have begun studying the notion of middleware-induced architectural styles and the ways in which they can be defined and exploited in distributed system development [4]. In this paper we discuss some of the issues we are addressing in our work.

2 An Example: A Pipe-and-Filter Application

The following example shows how the design of an architecture can conflict with the selection of a specific middleware. The architecture that we consider is shown in Figure 1 and is an instantiation of the pipe-and-filter style as it is defined in [1]. The first component of the architecture, the Source, generates some data and sends them to Filter 1. Pipe 1 is in charge of managing the communication between the two. Both Filter 1 and Filter 2 receive data from the component on their left, perform some computation and produce new data that is forwarded to the component on the right through the connecting pipe. Finally, the Sink consumes the data received from Filter 2. There are several variations of the pipe and filter style that concern the communication mode between components. In particular, components can produce output data incrementally or all at once. Moreover, the communication can proceed according to a *push* model, in which the producer always initiates communication, or a *pull* model, in which the consumer always initiates it.

To demonstrate that the choice of the middleware infrastructure is not independent of the architecture of the system to be implemented, let us compare the implementation we would obtain if we used CORBA as middleware to the implementation we would obtain if we used JEDI. CORBA provides the mechanisms to support point to point communication through remote method invocation. Its main component, the ORB, allows the elements of a system to abstract from the physical location of their counterparts. JEDI is an event-based middleware developed according to the publish/subscribe approach. Components can publish events, and they can subscribe for patterns of events and receive all the events that match their patterns, independently of the source that has published them. An event dispatcher is in charge of managing the dispatching of published events.

By using CORBA, the architecture in Figure 1 is implemented by defining as CORBA objects the Source, the two filters, and the Sink. The pipes are simply implemented as remote method invocations between objects. If we use a push communication model between components, the Source acts as the initiator of a computation by invoking a method PushData provided by Filter 1. In turn, Filter 1 invokes the method PushData provided by Filter 2, which, finally, invokes the method PushData provided by the Sink. In this case, therefore, Filter 1, Filter 2, and Sink act as CORBA servers, and they define an IDL interface containing the method PushData.

If components communicate according to a pull model, the structure of the system is fairly similar. The difference is that the CORBA servers in this case are the Source and the two Filters while the Sink acts as a client.[1] The IDL interface provided by the CORBA servers consists of the method PullData.

Notice that since in both cases all the components that act as CORBA servers share the same interface, the system can be easily reconfigured. The actual attachments between components can be defined when the system is started by passing as a parameter to each component the name of the next component.

Let us try to implement the same architecture using JEDI. As in the CORBA case, the two filters, the Source, and the Sink are implemented as JEDI components. The JEDI event dispatcher acts as a pipe between components. The data transmitted among components are encapsulated in events, and each component must subscribe to the events carrying the data in which it is interested. Figure 2 depicts a UML collaboration diagram representing an interaction scenario between Filter 1 and Filter 2. As shown in the figure, the choice of JEDI as a middleware infrastructure completely transforms the topology of the architecture of Figure 1. Indeed, if we look at the details of the way the communication is managed, the difference becomes even more evident. In this scenario we assume a pull model of communication, and we focus on the interaction that occurs between the two filters when Filter 2 decides to pull data from Filter 1. Filter 2 simulates a data request by generating the event ReadyToGetData and by subscribing to the event Data that will carry the data produced by Filter 1 (see Figure 2). However, Filter 1 must subscribe to the event ReadyToGetData before Filter 2 generates it. Since the event-based communication is one way, the relationship between events ReadyToGetData and Data is not managed explicitly by the infrastructure, but instead must be managed directly by the components. This places many additional requirements and constraints on the components that were not reflected in the initial choice of pipe-and-filter for the system's architectural style.

In summary, the implementation of the pipe-and-filter architecture in CORBA is straightforward, since CORBA provides direct support for the characteristics of point-to-point

---

[1] In both the push case and the pull case, Filter 1 and Filter 2 act as both clients and servers.

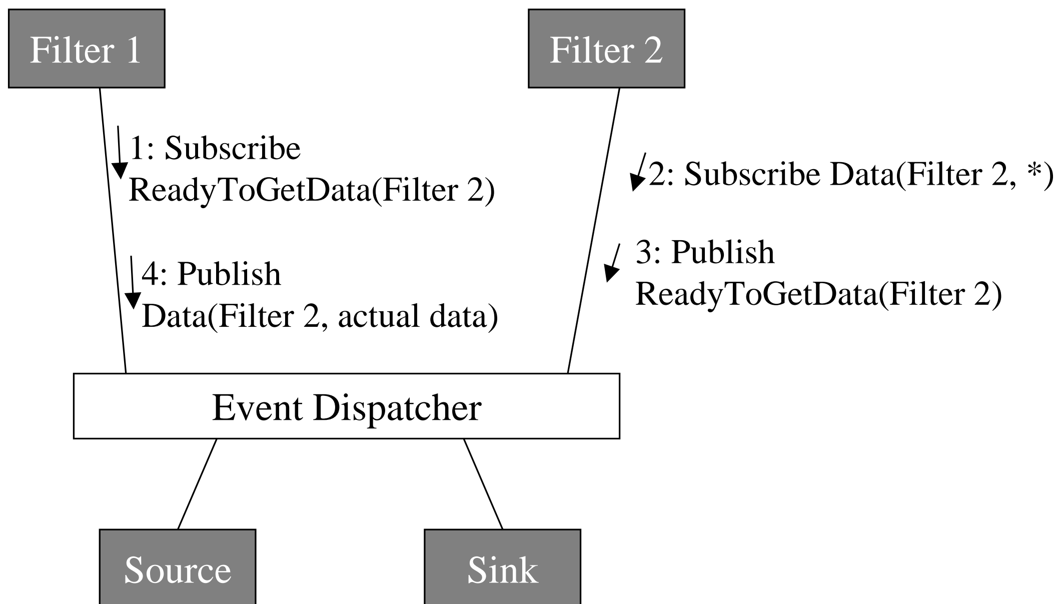

Figure 2. Implementation of the pipe and filter architecture using JEDI.

and synchronous communication underlying the style itself. If more complex pipes are needed (e.g., buffered pipes), they would also be defined as CORBA objects and inserted into the chain of inter-object method invocations.

Conversely, the implementation in JEDI requires the architecture to be modified to the point where it no longer resembles a pipe-and-filter architecture. Indeed, the components interact only with the Event Dispatcher, and they must carefully manage synchronization issues in order to simulate the effect of pipe-and-filter style interaction.

In other words, CORBA induces an architectural style that is compatible with the explicitly chosen pipe-and-filter style, while JEDI induces an architectural style that is incompatible (or difficult to reconcile with) the pipe-and-filter style.

3 Defining and Exploiting Middleware-Induced Styles

As shown in the previous section, because of the pervasiveness of middleware, every *explicit* decision made in developing the architecture of a system potentially introduces, at least in the case we considered, an *implicit* decision about the middleware that will be used to implement the architecture. As the structural and behavioral aspects of the architecture become defined, the range of middleware choices becomes increasingly limited. How then should the effects of such decisions be made known to the architect? We believe that decisions must be guided by knowledge about the different architectural styles induced by different middleware:

- Middleware-induced styles should be *explicitly defined* and made available to the community of software designers. These styles, in fact, would provide a valuable help in

the definition of the architecture of a system, so that the architecture can be easily implemented with a selected middleware infrastructure.

- Middleware-induced styles should also be related to the middleware-independent architectural styles that have been defined so far in the software architecture community [1]. We envisage the definition of a *style map*, where one or several style specialization hierarchies are defined. For instance, one specialization hierarchy could be rooted by a middleware-independent event-based style and could contain all the styles induced by specific event-based middlewares. Definition of an architecture would begin with the style at the root of one of these hierarchies. As architectural decisions are made, the architect would gradually transit the hierarchy toward the leaves. An important design guideline would be to defer making transitions in the hierarchy as long as possible, and to remain as close to the root of the hierarchy as possible, so as to offer the widest possible choice at the time the middleware is to be selected. With such hierarchies, one could determine how significant a design decision is in terms of the effect it has on limiting middleware choices. For instance, it might turn out that selecting between synchronous and asynchronous interaction places greater limitations on middleware choices than selecting between topologies that separate components structurally and those that separate components through data subtyping.

- The styles and style map should be formally described in an *architecture description language* (ADL) [5]. ADLs are being conceived as languages for specifying architectures and architectural styles. The formal definition of middleware-induced styles in a suitable ADL could provide substantial advantages to the architect. In particular, the architect could exploit the features of the ADL to define an architectural model as an instance of a particular style. The architect could then formally check the model for consistency with the properties of the style as the model is refined toward an implementation.

  In a recent paper we evaluated a number of architecture description languages (ADLs) on their ability to support the definition and exploitation of middleware-induced architectural styles [4]. We found that no one ADL provides all the necessary capabilities and features, and hence new ADLs are needed that can support middleware-based architectural development.

4 Conclusion

In this paper we have discussed the pervasive nature of middleware infrastructures and the architectural styles they induce in software systems. We have also touched upon a number of issues we are exploring in our study of middleware-induced styles.

Wileden and Kaplan argued that middleware should never constrain or dictate the architecture of a system, just as underwear should never constrain or dictate the rest of one's clothing. We feel that it is nevertheless useful to provide ways of informing architects about the consequences of their design decisions with respect to middleware choices, without restricting their freedom of movement. In this way it may be possible to avoid having unattractive choices of middleware creep up on an architect in an annoying way.

Acknowledgments

Authors would like to thank Professor Alfonso Fuggetta for his useful comments and suggestions on the issues discussed in this paper and Fabio Lomazzi and Laura Sfardini who have been working at the implementation of the pipe and filter architecture we have used as an example.

This effort was sponsored by the Defense Advanced Research Projects Agency, and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021; by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-98-1-0061; and by the National Science Foundation under grant number CCR-9701973. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Air Force Research Laboratory, Air Force Office of Scientific Research or the U.S. Government.

References

[1] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Reading, MA: Addison Wesley, 1998.

[2] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf, "Design of a Scalable Event Notification Service: Interface and Architecture", Department of Computer Science, University of Colorado at Boulder, Boulder, CO, Technical Report CU-CS-863-98, September 1998.

[3] G. Cugola, E. Di Nitto, and A. Fuggetta, "Exploiting an Event-Based Infrastructure to Develop Complex Distributed Systems", *Proc. 20th International Conference on Software Engineering*, Kyoto, Japan, pp. 261–270, April 1998.

[4] E. Di Nitto and D.S. Rosenblum, "Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures", *Proc. 21st International Conference on Software Engineering*, Los Angeles, CA, May 1999.

[5] N. Medvidovic and R.N. Taylor, "A Framework for Classifying and Comparing Architecture Description Languages", *Proc. 6th European Software Engineering Conference/5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Zurich, Switzerland, pp. 60–76, September 1997.

[6] J. Siegel, *CORBA Fundamentals and Programming*. New York, NY: Wiley, 1996.

[7] A. Thomas, "Enterprise JavaBeans™ Technology: Server Component Model for the Java™ Platform", Patricia Seybold Group, Boston, MA, white paper prepared for Sun Microsystems, Inc. December 1998.

[8] J.C. Wileden and A. Kaplan, "Middleware as Underwear: Toward a More Mature Approach to Compositional Software Development", Digest of the OMG-DARPA-MCC Workshop on Compositional Software Architectures, Monterey, CA, January 1998.