Are Architectural Aspects Style-Dependent?

Christina Chavez
Computer Science Dept.
UFBA, Brazil
flach@dcc.ufba.br

Alessandro Garcia
Computing Dept.
Lancaster University, UK
a.garcia@lancaster.ac.uk

Thais Batista
Computer Science Dept.
UFRN, Brazil
thais@ufrnet.br

Abstract

Over the past years, architectural aspects have been often defined as widely-scoped properties that naturally crosscut the boundaries of system components. This definition, however, neglects the concept of architectural style. Architectural styles prescribe different kinds of system decomposition and modularization by adopting distinct component and connector types. This position paper analyzes the influence exerted by architectural styles over the nature of architectural aspects in order to address the following research question: are architectural aspects style-dependent? Based on concrete examples, we claim that: *(i)* it is natural to make the assumption that some system concerns will be well modularized while others will not, depending on the selection of architectural style(s), and *(ii)* there are some concerns that are recurrently crosscutting independently of the architecture stylistic choices.

1 Introduction

Architectural aspects are expected to "modularize widely-scoped properties that naturally crosscut the boundaries of system components at the software architecture level" [1, 4, 7, 24]. In fact, most aspect-oriented approaches at the architecture level tend to focus on object or component modularity (e.g. [4, 7, 16, 24]), hence mimicking the object and component call graph based join point models in aspect-oriented programming languages. However, they tend to neglect other well established architecture-level modularity mechanisms such as architectural styles.

An *architectural style* prescribes a certain form of system decomposition and modularization, and adopts distinct component types and connector types, with style-specific semantics. Typical examples of architectural styles are [3, 9] client-server, layers, and pipe-and-filters. The selection of a particular architectural style [3, 18, 22,] or combination of styles [15, 21] for a software system has a significant impact on system decomposition and system-wide properties. Different architectural styles applied to the same problem can lead to designs with significantly different properties [20].

Hence, ignoring well-established architecture modularity mechanisms such as architectural styles makes it difficult to deploy an "aspect-oriented thinking" within an existing architecture design process. Architects need to have a broader understanding the effects of the crosscutting phenomena in the context of well-known architectural abstractions. Instead, discussions about aspect-oriented architectures have been reduced to a high-level model of the aspect-oriented program to be developed.

This position paper provides a first, significant stepping stone on addressing a key research question: how is aspect modularity related to style modularity? We claim that since a style prescribes some kind of system decomposition and modularization, some system concerns are well modularized while others are not, depending on the choice of architectural style(s). Also, based on our previous experience [6, 24, 25], we observed that there are some concerns that are recurrently crosscutting independently of the dominant architecture stylistic choice or specific style compositions. The following sections discuss some representative examples to support our claims. The examples are rooted at traditional widely-scoped crosscutting concerns, such as exception handling and distribution, and conventional architectural styles, such as pipe-and-filter and client-server.

2 Views, Styles and Aspects

Architectural aspects are often addressed at the module view, for which the chosen architectural style is the object-oriented style [12] and aspects are modular units that complement objects to modularize crosscutting concerns with respect to object-oriented decomposition. Therefore, in the module view, aspects are style-dependent.

However, software architecture description may comprise multiple views [2,12] and, for each view, architects can choose different architectural styles. In the runtime view, architecture descriptions rely on components to modularize computations and connectors to respectively modularize the interactions between components that have runtime presence. Architectural styles such as client-server, pipe-and-filters, implicit invocation, etc., define distinct component types and connector types that can be also regarded as different kinds of system decomposition and modularity. Several aspect-oriented approaches (e.g. [4, 7, 16, 24]) at the architecture level focus on component modularity in the runtime view but neglect architectural styles as an established architecture-level modularity mechanism.

This section presents two well-known architectural styles – client-server and pipe-filter (Section 2.1) – and discusses the ways they support the description of two different "widely-scoped properties" or concerns: error handling (Section 2.2), and distribution (Section 2.3). In particular, these concerns are evaluated with respect to their degree of scattering and/or tangling across components and connectors in client-server and piper-filter decompositions. The two architectural designs will be also contrasted with respect to their ability to support modular treatment of a those concerns.

2.1 Two Architectural Styles

This section presents architectural designs for the KWIC (*Key Word In Context*) index system expressed in two different architectural styles. The KWIC has been used in different contexts to illustrate the benefits and drawbacks of design choices driven by modular decomposition [17] and architectural styles [9]. The graphical notation used is supported by the ACME Studio tool.

2.1.1 Pipe and Filter

Pipe and filter systems are used for processing or transforming data streams. As such, the pipe-and-filter style prescribes functional decomposition. Components are functional entities ("filters") that usually apply local transformation to their input streams, incrementally, so that output begins before input is

consumed. Each filter has a set of inputs and a set of outputs. A filter reads streams of data on its inputs and produces streams of data on its outputs, delivering a complete instance of the result in a standard order. The connectors of this style ("pipes") serve as conduits for the streams, transmitting outputs of one filter to inputs of another [9].

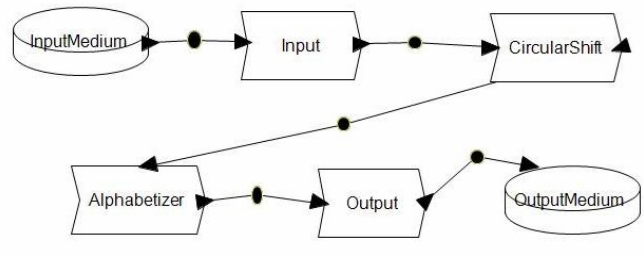


Figure 1: KWIC using a pipe and filter style [9].

Figure 1 presents the KWIC system using the pipe and filter style. The KWIC system reads an ordered set of lines (Input filter), and any line may be "circularly shifted" (CircularShift filter) by repeatedly removing the first word and appending it at the end of the line. The KWIC system sorts all circular shifts of all lines in alphabetical order (Sort filter) and outputs a listing of these ordered lines (Output filter) [17].

There are several reported advantages associated with the use of pipe and filter systems, resulting in enhanced reusability and maintainability of filters [9]. These advantages are aligned with the functional decomposition prescribed by the style. One of its disadvantages is related to data manipulation. The pipe and filter style is function-oriented – therefore, data representation is treated as a secondary concern. Changing the data representation is typically difficult because data type information may be spread across filter and pipe interfaces.

2.1.2 Client-server

The client-server style defines a service-oriented architecture where components can only be a client or a server. They are organized in terms of requesters and providers of services. Servers provide services to clients. A client can only be connected to a server and a server can only be connected to a client. A client requests a service provided by a server, which performs the service and returns the result to the client. Connectors specify the interaction between clients and servers.

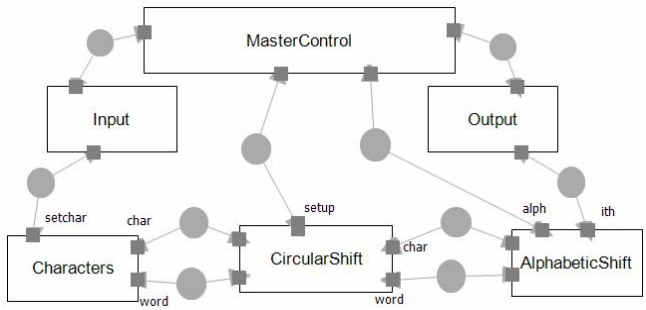


Figure 2: KWIC using the client and server style (adapted from the ADT style [9]).

Figure 2 presents a client-server version for the KWIC system adapted from the abstract data type (ADT) style [9]. LineStore, CircularShifter and Alphabetizer are servers that provide character-handling, shift and sort services, respectively. Input and Output are clients that read an input (a set of lines) and write output (ordered lines), respectively.

The main advantage of the client-server style is the facility of their use in distributed systems. Distribution of tasks among servers is straightforward. In addition, it is easy to add new servers and clients. Connectors separate and encapsulate the distribution protocols, thereby provide a modular handling of the distribution-related changes.

2.2 Error Handling

Error handling is widely recognized as a global design issue [8, 19] and has been extensively referred in the literature as a classical crosscutting concern in systems following different kinds of architecture decompositions [11, 13, 23]. Furthermore, it may affect almost all the system modules and their interfaces [5, 14], both in generic and context-specific fashions [5], representing an anti-modularity factor in several well-known architectural styles [3], and exhibiting intricate, heterogeneous relationships with the normal system behavior [6, 10].

Error handling is reported as the Achilles' heel of the pipe and filter style [3]. Since pipes and filters do not share any global state, error handling is hard to address in a modular way and, as a result, is often neglected in pipe-and-filter decompositions. Moreover, error handling on pipe and filter systems may require resynchronizing the elements when a filter or pipe crashes, and restarting the system.

Supporting error handling in the client-server style is also a daunting task. There are several points of failures: client invocation at the client side, client invocation at the server side, server reply at the server side, and server reply at the client side. For this reason, it is very difficult to apply a similar error handling strategy to all the servers in a modular fashion [3].

Error handling can also be rather expensive for others styles, such as layered architectures, with respect to processing time and programming effort [3]. An error can either be handled in the layer where it occurred or be passed to the next higher layer. In the latter case, the lower layer must transform the error into an error description meaningful to the higher layer [3].

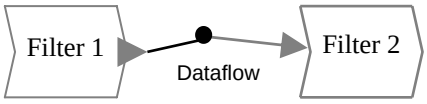
As a result, error handling can be considered as a classical example of architectural aspect that is not style-dependent. In another words, it does not matter what kind of dominant architectural decomposition is selected: global error handling strategies will be always crosscutting. In fact, default error handling policies, such as exception propagation or error-specific information logging [6], are consistently present in almost all types of architectural decomposition and, in those architectures, tend to "traverse" many interfaces and connectors.

2.3 Distribution

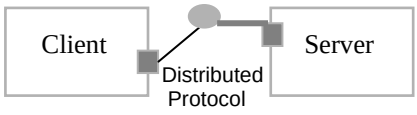
As mentioned in section 2.1.2, in the client-server style elements are servers, clients and client-server connectors. This style is widely used in the modeling of distributed applications because client-server connectors typically implement a distributed communication protocol such as a remote procedure call (RPC), a publisher-subscriber protocol, or a broker. In fact, the client-server connector is an element that explicitly addresses the remote communication between clients and servers. In this way, no additional support for distribution is needed in the client-server style as the style itself intrinsically offers an element for this purpose. In order to have a smooth composition of peers the style defines that clients and servers components are modeled in a remote communication-oriented way (clients require services that are provided by servers).

On the other hand, in the pipe-and-filter style, as discussed in section 2.1.1, the connectors of this style – the pipes – are

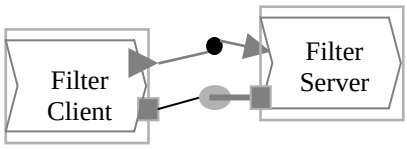
necessarily concerned with transmitting outputs of one filter to inputs of another [9]. Like the client-server connector, the pipe connector could also incorporate the implementation of a distributed communication protocol. However, this is not intrinsically defined by the pipe-and-filter style because this is not the primary function of the pipe connector. Its main function is to synchronously transport a stream of typed data from a source to a sink. If this transport function uses a remote protocol, the filter components must also be prepared to communicate via a distributed protocol. Again, this is not the role defined for filter-like components in a pipe-and-filter style. In addition, we could try to model filters and pipes in a client-server distributed fashion implies in the insertion of elements in both filters and pipes to represent the distributed communication protocol. The result is distribution-specific concerns being scattered and tangled all over the basic elements of the style.



(a) Pipe-Filter Style



(b) Client-Server Style



(c) Hybrid Style

Figure 3. Hybrid style combining client-server and pipe-and-filter styles.

Another possibility for including remote communication support in the pipe-and-filter style is to adopt a hybrid architectural solution; i.e. a new hybrid style could be defined based on the conjunction of both piper-filter and client-server styles. Filters become client or servers and pipes' functionalities are mixed with the distributed communication protocol implemented by the client-server connector. This solution still introduces a certain degree of tangling in the implementation of both filters and pipes (Figure 3).

Following the definition of architectural aspects provided in [24], an aspect could be defined to localize the distributed communication in a base piper-filter decomposition. Based on the notion of aspectual connectors [24], the pipe-and-filter elements maintain their original functionality and the distribution-specific aspectual connector implements the remote communication protocol. The later acts upon the communication between the filter elements (i.e. the join points in which the aspectual connectors are attached to). This modular solution would support distribution in instances of the pipe-and-filter style, while reducing tangling and scattering of concerns

In summary, certain architectural aspects depend on the dominant stylistic choices because: *(i)* in some styles a given concern can be intrinsically modularized by the style (this is the case of the distribution concept in the client-server style), and *(ii)* in another style(s) the same concern is not even supported by the style. In this last case, if the insertion of the target concern necessarily cuts across the basic style elements, it is necessary to define an architectural aspect to modularly represent such a concern.

3 Concluding Remarks

The tyranny of the dominant architectural style. The kind of decomposition supported by the two styles described may favor the clean modularization of some concerns while others may be not well-modularized. Therefore, styles may interfere with the nature of the crosscutting concerns at the architectural level. Architectural aspects can be style-dependent – e.g., distribution for the client-server style, data representation for the pipe and filter style. Like any abstraction, the chosen style emphasizes some concerns of the problem and suppresses others [20]. Hence, some concerns are expected to be well localized within specific kinds of modular units defined by the style, while others are expected to crosscut their boundaries.

Style-agnostic crosscutting concerns. As discussed in Section 2.1, error handling is a typical example of concern that is persistently crosscutting in all the conventional architectural styles. Hybrid architectures do not solve the problem either. In fact, error handling has been extensively referred in the literature as a classical crosscutting concern in systems following different kinds of architecture decompositions [11, 13, 23]. Error handling is widely recognized as: *(i)* a global design issue [8, 19], *(ii)* affecting almost all the system modules and their interfaces [5, 14], *(iii)* both in generic and context-specific fashions [5], *(iv)* an anti-modularity factor in several well-known architectural styles [3], and *(v)* exhibits intricate, heterogeneous relationships with the normal system behavior [10, 6].

Style-specific architectural join point models. Our previous discussions imply that we might need more expressive join point models (and pointcut languages) for aspect-oriented architecture design. As the selection of one or more architecture styles determine the design space for the particular problem, the style semantics should provide a basis for architectural aspect description. However, this is a topic that needs further exploration.

References

[1] E. Baniassad, P. C. Clements, J. Araujo, A. Moreira, A. Rashid, and B. Tekinerdogan. Discovering Early Aspects. *IEEE Software*, 23(1):61–70, 2006.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley, Reading, Massachusetts, USA, 2003. 2nd ed.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[4] R. Chitchyan, A. Rashid, P. Sawyer, A. Garcia, M. P. Alarcon, J. Bakker, B. Tekinerdogan, and A. J. Siobhán Clarke and. Survey of aspect-oriented analysis and design approaches. Technical Report AOSD-Europe-ULANC-9, AOSD-Europe, May 2005.

[5] F. Cristian. A recovery mechanism for modular software. In *ICSE '79: Proceedings of the 4th international conference on Software engineering*, pages 42–50.A, Piscataway, NJ, USA, 1979. IEEE Press.

[6] F. Filho, N. Cacho, R. Ferreira, E. Figueiredo, A. Garcia, and C. Rubira. Exceptions and Aspects: The Devil is in the Details. In *Proc. of FSE-14, 14th*

International Conference on Foundations on Software Engineering, 2006.

[7] A. Garcia and C. Lucena. Taming heterogeneous agent architectures with aspects. *Communications of the ACM*, October 2006.

[8] A. Garcia, C. M. F. Rubira, A. Romanovsky, and J. Xu. A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of Systems and Software*, 59(2):197–222, 2001.

[9] D. Garlan and M. Shaw. An introduction to software architecture. Technical Report CMU-CS-94-166, Carnegie Mellon University, January 1994.

[10] J. B. Goodenough. Exception handling: issues and a proposed notation. *Commun. ACM*, 18(12):683–696, 1975.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–355. Springer, 2001.

[12] P. Kruchten. The 4+1 view model of architecture. *IEEE Softw.*, 12(6):42–50, 1995.

[13] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning, 2003.

[14] M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 418–427, New York, NY, USA, 2000. ACM Press.

[15] N. R. Mehta and N. Medvidovic. Composing architectural styles from architectural primitives. In *Proc. 9th European software engineering conference (ESEC)*, pages 347–350, New York, NY, USA, 2003. ACM Press.

[16] P. Merson. Representing aspects in the software architecture practical considerations. In E. Baniassad, J. Araújo, P. Clements, and P. Merson, editors, *Early Aspects: OOPSLA 2005*, Oct. 2005.

[17] D. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[18] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[19] M. P. Robillard and G. C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Trans. Softw. Eng. Methodol.*, 12(2):191–221, 2003.

[20] M. Shaw. Comparing architectural design styles. *IEEE Softw.*, 12(6):27–41, 1995.

[21] M. Shaw and P. Clements. Toward boxology: preliminary classification of architectural styles. In *Joint Proc. of the 2nd software architecture workshop (ISAW-2) and workshop on multiple perspectives in software development (Viewpoints '96)*, pages 50–54, New York, NY, USA, 1996. ACM Press.

[22] M. Shaw and D. Garlan. *Software Architecture: Perspective on an Emerging Discipline*. Prentice Hall, 1996.

[23] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with aspectj. In *OOPSLA*, pages 174–190, 2002.

[24] Garcia et al. On the Modular Representation of Architectural Aspects. Proc. of the 3rd. European Workshop on Software Architecture, Nantes, France, September 2006.

[25] U. Kulesza, C. Sant'Anna, A. Garcia, R. Coelho, C. Lucena. Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study. Proc. 9th Intl. Conf. on Software Maintenance (ICSM'06), Philadelphia, USA, September 2006.