What is Style?

David Garlan
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

1. The Value of Style

A central aspect of architectural design is the use of recurring organizational patterns and idioms—or *architectural styles* [GS93, PW92, MG92, GHJV94]. Examples include generic system organizations such as those based on dataflow or layers, as well as specific organizational structures such as the classical decomposition of a compiler, the OSI communication stack, and the MVC user interface paradigm.

The principled use of architectural styles has a number of practical benefits. First, it promotes design reuse: routine solutions with well-understood properties can be reapplied to new problems with confidence. Second, it can lead to significant code reuse: often the invariant aspects of an architectural style lend themselves to shared implementations. Third, it is easier for others to understand a system's organization if conventionalized structures are used. For example, even without giving details, characterization of a system as a "client-server" organization immediately conveys a strong image of the kinds of pieces and how they fit together. Fourth, use of standardized styles supports interoperability. Examples include CORBA object-oriented architecture [Cor91], and event-based tool integration [Ger89]. Fifth, by constraining the design space, an architectural style often permits specialized, style-specific analyses. For example, it is possible to analyze pipe-filter systems for schedulability, throughput, latency, and deadlock-freedom. Such analyses might not be meaningful for an arbitrary, ad hoc architecture – or even one constructed in a different style. Sixth, it is usually possible to provide style-specific visualizations: this makes it possible to provide graphical and textual renderings that match engineers' domain-specific intuitions about how their designs should be depicted.

Unfortunately, the use of architectural styles is almost completely ad hoc. It is virtually impossible to answer with any precision what aspects of system design can/should be specified by a style, to compare different styles based on their properties, to relate systems developed in different styles, to develop general-purpose tools for exploiting style, to select appropriate styles for a given problem, or to combine several styles to produce a new one.

Evidentally what is needed (among other things) is a more rigorous basis for understanding architectural style and ways to exploit it. In this paper I briefly outline and compare three approaches to providing such a basis.

2. Basic Properties

Before outlining the approaches, it is worth noting four salient aspects of architectural styles that any model should account for:

- They provide a *vocabulary* of design elements – component and connector types such as pipes, filters, clients, servers, parsers, databases etc.
- They define a set of *configuration rules* – or topological constraints – that determine the permitted compositions of those elements. For example, the rules might prohibit cycles in

a particular pipe-filter style, specify that a client-server organization must be an n-to-one relationship, or define a specific compositional pattern such as a pipelined decomposition of a compiler.

- They define a *semantic interpretation*, whereby compositions of design elements, suitably constrained by the configuration rules, have well-defined meanings.
- They define *analyses* that can be performed on systems built in that style. Examples include schedulability analysis for a style oriented toward real-time processing [Ves94] and deadlock detection for client-server message passing [JC94]. A specific, but important, special case of analysis is code generation: many styles support application generation (e.g., parser generators), or enable the reuse of code for certain shared facilities (e.g., user interface frameworks and support for communication between distributed processes).

3. Three Views of Architectural Style

Now let us consider three ways of understanding style.

1. **Style as Language** In this view a stylistic design vocabulary is modelled as a set of grammatical productions. Configuration rules are defined as context-free and context-sensitive rules of the grammar. A semantic interpretation can be given using any of the standard techniques for assigning meaning to languages. Analyses are essentially those that one can perform on architectural "programs" – namely checking for satisfaction of grammatical rules, flow analysis, compilation, etc. Characteristic of this view is [AAG93], where a style is viewed as a denotational semantics for architectural diagrams.

2. **Style as a System of Types** In this view the architectural vocabulary is defined as a set of types. For example a pipe-filter style might define filter and pipe types. If specified in an object-oriented context, hierarchical definitions are possible: "filter" would be a subclass of a more generic "component," and "pipeline-stage" a subclass of "filter". Similarly a "pipe" would be a subclass of a generic "connector." Constraints on these types can be maintained as datatype invariants, operationally realized in the code of the procedures that can modify instances of the types. Analysis can exploit the type system, to perform type checking, and other architectural manipulations that depend on the specific types involved (such as code generation). Representative of this view is Aesop, which provides an object-oriented approach to defining new styles using subclassing [GAO94]. (Aesop actually goes farther by defining a subclassing relationship between full styles, not just the individual types in the style.)

3. **Style as a Theory** In this view a style is defined as a set of axioms and inference rules. Vocabulary is not represented directly, but in terms of the logical properties of elements. For example, the fact that an architectural component is a filter would allow you to deduce that its ports are either input ports or output ports. Similarly the fact that something is a pipe would allow you to deduce that there are two ends, one for reading, and the other for writing. Configuration constraints and semantics are defined as further axioms. Analysis takes place by proving a new theorem, thereby extending the theory of the style. Representative of this approach is the work by Moriconi and his colleagues [MQ94, MQR95].

4. Comparisons

At first glance, all three views of style might appear interchangeable. For example, it is possible to view a grammar production as an object-oriented class, where the components of the production are the instance variables of the class. Similarly, both of these views can be modelled formally (e.g., in Z [GD90]), hence providing a theory for the style.

But on closer examination the models have very different properties along several key dimensions:

- **Representation of structure:** In both the language and types views the architectural structure of a specific system is explicit: in the former as an expression in the language (or abstract syntax tree), in the latter an interconnected collection of objects. In the theory view structure must be encoded as a set of assertions, distinguished from other assertions primarily by convention. However, even in the first two, there are differences: in an object representation the graphical structures of a system's architecture are directly represented, while in the grammar-based approach they must be encoded in terms of the hierarchical representation determined by the language's abstract syntax.

- **Substyles:** Styles can be related in various ways, but one of the most important is the "substyle" relation. The basic idea is that one style is a substyle of another if instances of the former can be "treated as" instances of the latter. This is important both theoretically and practically. Theoretically it allows us to know that the results we prove about one style will carry over to its substyles. Practically it allows us to reuse tools: if a tool can manipulate systems of one style, then it can manipulate those of its substyles. (For example, a throughput analyzer for a pipe-filter system should also work on a pipeline system.) Moreover, it may lead to techniques for defining new styles in terms of old ones: instead of creating each new style from scratch we reuse old style definitions.

  In the theoretical view a notion of substyle falls out naturally, since one can simply define substyle in terms of theory inclusion. In the case of an object-oriented model, it can also be made to work. Each type of a new style is a subtype of a superstyle. (This is the approach taken by Aesop [GAO94].) In the language-based view, however, it is not at all clear how substyles would be defined. Generally languages are treated as stand-alone entities, with little formal relation to any other language.

- **Refinement:** In many cases a system is best described as layered abstraction, each layer defined by a different style. Layers are related by refinement—that is, there exists a formal abstraction mapping between layers, and a lower layer preserves the properties of its upper layer (via the mapping).

  The theoretical view has a natural explanation for refinement. Moreover, as Moriconi has shown [MQR95], is possible to use *styles* to factor out patterns of refinement. To date there has been little work in showing how either the language or types views handle refinement.

- **Automated support:** Architectural design is motivated by practical concerns, and one would hope that appropriate models of style would lead to practical tools for description and analysis of architectural descriptions. In this respect all three views have merits, but in different ways. The language approach can exploit automated support developed for programming languages: type checkers, code generators, attribute grammar evaluators, etc. The types approach can exploit the use of object-oriented databases and tools for storing, visualizing, and manipulating architectural designs. The theoretical view can exploit the power of formal manipulation systems, such as theorem provers and model checkers.

So which is better? Obviously there is not a definitive answer to this. As this brief description has indicated, each view has its respective merits, and all of the views complement each other. Moreover, there remains a considerable amount of research and experimentation that is needed to fully understand the strengths and weakness of the different approaches, and, indeed, other approaches not mentioned here.

Acknowledgement

The research reported here was sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant F33615-93-1-1330; by National Science Foundation Grant CCR-9109469; and by a grant from Siemens Corporate Research. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Wright Laboratory, the US Department of Defense, the United States Government, the National Science Foundation, or Siemens Corporation. The US Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon.

References

[AAG93] Gregory Abowd, Robert Allen, and David Garlan. Using style to understand descriptions of software architecture. In *Proceedings of SIGSOFT'93: Foundations of Software Engineering*, Software Engineering Notes 18(5), pages 9–20. ACM Press, December 1993.

[Cor91] The Common Object Request Broker: Architecture and specification. OMG Document Number 91.12.1, December 1991. Revision 1.1 (Draft 10).

[GAO94] David Garlan, Robert Allen, and John Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of SIGSOFT'94: Foundations of Software Engineering*. ACM Press, December 1994.

[GD90] David Garlan and Norman Delisle. Formal specifications as reusable frameworks. In *VDM'90: VDM and Z – Formal Methods in Software Development*, pages 150–163, Kiel, Germany, April 1990. Springer-Verlag, LNCS 428.

[Ger89] Colin Gerety. HP Softbench: A new generation of software development tools. Technical Report SESD-89-25, Hewlett-Packard Software Engineering Systems Division, Fort Collins, Colorado, November 1989.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Design.* Addison-Wesley, 1994.

[GS93] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39, Singapore, 1993. World Scientific Publishing Company. Also appears as SCS and SEI technical reports: CMU-CS-94-166, CMU/SEI-94-TR-21, ESC-TR-94-021.

[JC94] G.R. Ribeiro Justo and P.R. Freire Cunha. Deadlock-free configuration programming. In *Proceedings of the Second International Workshop on Configurable Distributed Systems*, March 1994.

[MG92] Erik Mettala and Marc H. Graham. The domain-specific software architecture program. Technical Report CMU/SEI-92-SR-9, Carnegie Mellon Software Engineering Institute, June 1992.

[MQ94] M. Moriconi and X. Qian. Correctness and composition of software architectures. In *Proceedings of ACM SIGSOFT'94: Symposium on Foundations of Software Engineering*, New Orleans, Louisiana, December 1994.

[MQR95] M. Moriconi, X. Qian, and R. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 1995. To appear.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[Ves94] Steve Vestal. Mode changes in real-time architecture description language. In *Proceedings of the Second International Workshop on Configurable Distributed Systems*, March 1994.