Style-Based Reuse for Software Architectures

Robert T. Monroe and David Garlan

School of Computer Science, Carnegie Mellon University, Pittsburgh PA 15213
{bmonroe,garlan}@cs.cmu.edu

Abstract

Although numerous mechanisms for promoting software reuse have been proposed and implemented over the years, most have focused on the reuse of implementation code. There is much conjecture and some empirical evidence, however, that the most effective forms of reuse are generally found at more abstract levels of software design. In this paper we discuss software reuse at the architectural level of design. Specifically, we argue that the concept of "architectural style" is useful for supporting the classification, storage, and retrieval of reusable architectural design elements. We briefly describe the Aesop system's Software Shelf, a tool that assists designers in selecting appropriate design elements and patterns based on stylistic information and design constraints.

1. Introduction

Traditionally, the primary focus of reuse research has been on the reuse of code-level entities, such as classes, subroutines, and data structures. While there have been significant improvements in code reuse technology and methods, code-level artifacts are not the only ones that can be profitably reused. In this paper we describe an approach and supporting tool for a class of *design* reuse – namely, architectural reuse.

Broadly speaking, design reuse appears promising for at least three reasons. First, since designs address early phases of system development, many of the up-front (and hence most costly) errors can be avoided. Second, reuse of familiar designs can improve the understandability of a system, making it easier to evolve and maintain. Third, design reuse promotes code reuse: often much of the infrastructure to support a design can be shared among applications that share that design.

It is perhaps not surprising then, that some of the more impressive examples of reuse today involve a strong component of design reuse. Prominent examples include specialized frameworks such as user interface toolkits, application generators (such as Visual Basic™), domain-specific software architectures [MG92], and object-oriented patterns [GHJV94].

But what exactly is design reuse and how can it be exploited? Most examples, such as those just mentioned, capitalize on a specialized domain, providing specific facilities for a fairly narrow class of system. By trading generality for power, they leverage the domain to provide common infrastructure, and specific instantiation mechanisms for reusing that infrastructure.

We have been exploring a different, but complementary approach. Rather than focus on a specific class of system we consider the more general problem of the reuse of *architectural designs*. An architectural design is concerned with the gross decomposition of a system into a set of interacting components [GS93, PW92]. At this level of abstraction, key issues include the assignment of functionality to design elements, protocols of interaction, system extensibility, and broad system properties such as throughput, schedulability, and overall performance. The reuse problem for architectural designs then becomes how to exploit the basic elements of architectural design (large-scale components and their interactions), as well as common structures and idioms for their composition.

The approach we describe in this paper uses the concept of *architectural style*, for representing and reusing architectural designs and design fragments. As we will illustrate, the use of architectural style can provide assurance that elements built following the stylistic guidelines are interoperable. Further, the use of style supplements traditional mechanisms for classifying design elements, storing those elements in a repository, and narrowing the search space to more accurately locate potential element matches in a given context.

We begin by contrasting our approach to existing work in the areas of reuse and software architecture. Next we motivate the challenges for design reuse by considering existing problems with code-level reuse. We then show how an architectural view of the problem, supported by the concept of architectural style, addresses those issues. As a concrete demonstration of how these ideas can

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. Robert Monroe is partially supported by a National Science Foundation Graduate Research Fellowship. 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.

be used, we describe a prototype tool called the "Software Shelf" which provides a repository for reusable architectural design elements.

2. Related Work

The two areas most closely related to the research presented here are software reuse and software architecture.

2.1. Software Reuse

Over the past decade there has been a vast amount of work in the area of software reuse (e.g., see [BP89]). Within this broad arena a number of researchers have looked at the problem of design reuse. In particular, efforts to raise the level of abstraction in code reuse date back at least to the Draco system [Nei84], which was aimed at reusing domain specific knowledge, tools, and frameworks, and Baxter's work on Design Maintenance Systems [Bax92].

Currently the most successful forms of design reuse are in three areas. The first is the area of application frameworks, including user interface toolkits (such as X-Windows and Motif, Microsoft Windows™ API, and the Macintosh Toolbox™), and application generators (such as Lotus Notes™, Microsoft Visual Basic™, Lex and Yacc). In this area reuse is achieved by exploiting a shared implementation base and a common design framework for a specific class of software application [Kru92].

The second area is domain-specific software architecture (DSSA). A DSSA system focuses on software support for an application domain, such as avionics, mobile robotics, or command and control [MG92]. Typically it provides one or more domain-specific notations for characterizing a specific use of the architecture, together with tools for using these notations to generate a specific system implementation.

The third area is the recent development of object-oriented design patterns. These patterns attempt to capture common idioms of object-oriented software organization, such as the model-view-controller paradigm of Smalltalk-80 [KP88]. Recently a number of patterns have been collected into published handbooks [GHJV94, Pree95].

Broadly speaking, our work is motivated by the same concerns that these three areas address – providing high-level abstractions for constructing new systems based on previous designs and implementations. It does, however, also differ in several ways.

With respect to the first two areas, our work attempts to address the broader concern of architectural modeling: unlike application frameworks and DSSA, our mechanisms are not specifically tied to a particular application domain. While we also exploit architectural commonalities (in our use of "architectural style"), we start from a much more generic base of components and interactions. The advantage of taking this more general approach is that we provide a framework for architectural reuse that cuts across many application domains.

Our work is much more closely related to the third area – object-oriented patterns. As with our approach, object-oriented patterns are not tied to specific applications. Consequently, they can provide general design guidelines for structuring a wide variety of systems. On the other hand, object-oriented patterns are (not surprisingly) object-oriented. That is, they rely on method invocation as their primary form of compositional glue. In contrast, our work allows designers to define and use higher-level abstractions (such as pipes, event broadcast, and other complex protocols) for component composition. This permits the description of more abstract designs, but at the same time places our descriptions at a farther distance from real implementations.

2.2. Software Architecture

The second broad area of related work is software architecture, a topic that is receiving increasing attention from researchers and practitioners in areas such as module interface languages, domain-specific architectures, software reuse, codification of organizational patterns for software, architectural description languages, formal underpinnings for architectural design, and architectural design environments. Collectively these efforts are attempting to establish an engineering basis for architectural design, and make principles and techniques of architectural design more widely accessible.

In this paper we leverage the results of this community of research in several ways. First, as we detail later, we adopt the architectural vocabulary of components, connectors and configurations that is becoming increasingly well-accepted as the basis for architectural description [GS93, PW92]. Second, we make essential use of the notion of architectural idioms – or styles – as a way of characterizing a family of architectural designs that share a set of common assumptions [AAG93]. Third, we take advantage of the emerging class of architecture support tools for developing and analyzing architectures [Shaw+95, Luc+95]. Indeed, their very existence motivated us to provide tools for reusing architectural designs.

One of these support tools is our own Aesop system [GAO94]. As we outline later in this paper, Aesop is a development environment for software architectures. In previous papers we have described the basic concepts behind Aesop and illustrated its use. In this paper we

extend those results by showing how a new tool, the Software Shelf, can further exploit architectural designs and architectural styles to support design reuse.

Outside of our own research, there are two closely related tools for architectural design: Weaves [GR91] and MacSTILE [SW88]. Weaves supports the development of systems based on asynchronous typed datastreams. It provides a repository mechanism, called a "tray," that, like our Software Shelf, uses the context of an architectural design to determine likely candidates for reuse. On the other hand, Weaves focuses on a specific style. It does not specifically address the more general issues of architectural design reuse. Similarly, MacSTILE provides an environment for graphically describing logical relations between components. These relations, however, work only within a fixed number of styles.

3. Limitations of code-level reuse

Although reusing implementation-level code can significantly improve the economics of software development, it faces at least three fundamental problems.

1. **The assumptions about the context in which an implementation will work are unstated:** With concrete code reuse it is often difficult to discover the assumptions about the context in which the component is intended to work [GAO95]. Context includes expected protocols of interaction, locus of control, scheduling constraints, etc.

2. **Programming languages do not adequately support the description of complex software assets:** For example, a software component may be designed to interact in different ways with different parts of its environment, such as communicating over data streams to some entities, providing a procedural API to a user interface, and relying on operating systems libraries for other interactions. Ideally one would like to characterize each of these "interfaces" separately. But it is difficult to describe more complex packages, such as those involving multiple cooperating objects with current programming languages.

3. **Storage and retrieval mechanisms are generally based on ad-hoc classification systems:** If the elements in a repository are arbitrary implementation code then the (re)user will have to rely on whatever meta-data and documentation has been associated with the repository entities, or else examine the code itself. Although numerous repository schemas have been developed for classifying software assets, such as component facets [PDF87], most of these mechanisms are based on informal classification schemes. Reliance on the accuracy of the informal descriptions of the code is suspect, especially if the annotations were added independently by different people over a significant period of time.

A plausible antidote for these ills is a stronger focus on *design reuse*. By making available higher-level design entities it should be possible to address the problems just enumerated. First, by relying on a richer design vocabulary – and not primarily code – the context of use can be explicitly defined. Moreover, design-oriented notations can permit the description of more complex entities than one could describe using an implementation-level language. Finally, design-level taxonomies can be used to classify and retrieve reusable parts.

To realize these potential benefits, however, the notion of design reuse must be made much more concrete. What kinds of design-oriented parts can be reused? How are they described, classified, stored, and retrieved? In the next three sections we show how an architectural approach leads to one set of answers.

4. Software Architecture and Architectural Style

Software architecture is the level of design at which a system is defined as a composition of interacting, module-scale components [GS93,PW92]. Adopting the emerging vocabulary of this field, a software architecture can be defined in terms of *components* representing the application-level computational entities, *connectors* representing interactions between components, and *configurations* representing assemblages of components and connectors. As a simple example, the architecture of a small client-server system might be a configuration of two components (a client and a server), joined by a single connector (representing the client-server protocol).

Architectural designs are ubiquitous as system documentation for complex industrial software systems. Commonly they are represented as informal box-and-line diagrams, although recently architectural definition languages are beginning to emerge [Luc+95, Shaw+95], as well as formalisms for architectural specification and analysis [AG92, Mor94].

Architectural description has a number of important design-level benefits over code-level descriptions. First, it permits richer descriptions of components. In particular, it is common to partition a component's interface into multiple "ports," each port determining an interaction with some part of its environment. This is in contrast to module languages that provide a single flat interface. Second,

architectural descriptions typically provide the ability to define new kinds of system "glue," or connectors. This allows expressiveness beyond what is provided by programming languages (and their module facilities), since it permits first class abstractions for interactions such as piped data streams, event broadcast, and other complex protocols of interaction.

Third, the use of architectural description naturally leads to the capability for exploiting "architectural styles." An architectural style provides a specialized architectural design vocabulary for a family of systems, and typically incorporates a number of idiomatic uses of that vocabulary and design rules for system composition [GS93, PW92, Mor94, GAO94]. As a simple example, Unix pipe-and-filter systems provide a specialized component vocabulary of filters (as data stream transformations) and a connector vocabulary of pipes (as data channels). Among the idiomatic patterns are the notions of a strict "pipeline" topology.

From the point of view of a designer, architectural style is important for several reasons. It limits the design space, thereby simplifying design choices. It allows a designer to exploit recurring patterns of organization, such as topological configurations, or even specific organizations of components (such as the MVC pattern in object-oriented systems). It provides a context within which certain kinds of design integrity can be enforced, such as the fact that no cycles are allowed. It permits specialized analyses such as detection of deadlock. And finally, as we detail in the next section, it provides a basis for supporting reuse of architectural building blocks and patterns.

While the field of software architecture is only now emerging as an engineering discipline, there are already many indications that architectural design will become an increasingly important component of large scale software development. The DSSA program (mentioned earlier) has demonstrated that considerable cost savings can be achieved through support of architectural design within specific domains. More recently, other projects such as the Advanced Distributed Simulation Program has developed a reusable "high-level" software architecture for integration of diverse sets of simulations. Further, there are the increasingly prominent results of the object-oriented patterns community [GHJV94].

5. Style-Based Architectural Reuse

By focusing on the reuse of architectural entities and exploiting the notion of architectural style, it is possible to realize many of the benefits of design-level reuse. To demonstrate how this is accomplished at a concrete level, we describe the *Software Shelf*,[1] a tool for storing and retrieving architectural designs. As we elaborate below, the main features of the Shelf are:

- **Support for storage of rich design elements:** The Shelf can store not only components with associated implementations, but also connectors and design patterns. Moreover, by associating stylistic information with these design entities, we are able to explicitly define the intended context of use. This addresses problems 1 and 2 enumerated in Section 3.
- **A common vocabulary:** Every design element stored on the shelf has a number of architectural attributes associated with it, including the styles in which it works, its architectural category and class (see table 1), and the interface(s) it provides to the outside world. Because all items that share a stylistic attribute conform to the constraints of that style, a potential reuser can quickly determine the high-level attributes of an item without having to review ad hoc annotations about that item, or worse, review its low-level code. This addresses problems 2 and 3 of Section 3.
- **Mechanisms to support design checking and guidance:** By attributing design entities with information such as the styles within which they can be used, the specific stylistic type within the vocabulary of the style (e.g., a filter, pipe, client, or server), as well as various extra-functional properties, our tools can check whether a component is being used in an appropriate context. Moreover, it also permits one to do context-sensitive browsing. That is, the context of use within a design can be exploited to limit the search of the Shelf to those objects that could legally fit in that context. This addresses problems 2 and 3 of Section 3.

6. A Style-Based Repository for Architectural Designs

As part of the process of exploring the concepts of architectural style and design reuse, we have built a toolkit called *Aesop* that is designed to support the rapid development of "style-aware" software architecture design environments. More recently, we have added to Aesop a repository for design elements and patterns called the Software Shelf. The following sections provides a brief overview of how Aesop and its Software Shelf support design reuse, focusing on how they use stylistic annotations to supplement more traditional mechanisms for classifying, storing, and retrieving software assets.

1. The term "Software Shelf" was originally suggested to us by Jon Ward at Honeywell.

6.1. Aesop

Aesop [GAO94] is an experimental platform for exploring software architecture and architectural style that, among other things, allows us to test different mechanisms and strategies for supporting architectural reuse. The basic thesis underlying Aesop is that software designers can benefit from using specialized design environments and tools that exploit stylistic knowledge about a family of systems. Historically, these specialized environments have been expensive to build and configure for individual styles and families of systems. Aesop attempts to minimize the cost of building these systems by providing a generic infrastructure of common tools (design database, GUI, Software Shelf, editors, protocol consistency checkers, etc.) and providing mechanisms for easily building a new environment to support a specific style of design.

A core capability of Aesop is the ability to define architectural styles. A style definition consists of a vocabulary of design elements that are available for use in that style, rules for determining when a configuration of components and connectors is well formed, and tools to manipulate and analyze designs created in the style. For example, a client-server style would likely define component classes for clients and servers, as well as connector classes for the interactions between clients and servers. Configuration rules might prohibit two clients from direct communication. Analyses might include checking for potential sources of deadlock (via cycles), compiling a design using existing RPC facilities of a host operating system, or calculating expected throughputs. Such analyses make use of style-specific properties of the composable design elements.

The nature of the checking that can be performed by Aesop depends considerably on the style in question. Some styles specify few constraints that can be automatically analyzed. Others have detailed semantics and criteria for completeness and consistency. (We return to this issue in the Evaluation section.)

6.2. The Software Shelf

The Software Shelf (or simply "Shelf") is a tool that communicates with Aesop to support the classification, storage, and retrieval of architectural elements: components, connectors, and configurations (or, design patterns). We now describe the strategies taken to implement this functionality, and provide examples to illustrate the utility of embedding stylistic information in the reusable assets stored on the Shelf. Many of the underlying mechanisms that we use to implement Shelf functionality are not new. Our intention has been to leverage well-known repository techniques while updating them to support and exploit the concept of architectural style.

6.2.1. Classification:

Entities stored on the Shelf are classified along three basic dimensions. The first dimension specifies whether the design element represents a computation (component), an interaction (connector), or a configuration (pattern). The second dimension is the specific style, or set of styles, for which that design element was created. This determines the context into which the element can be inserted (reused). The third dimension is the specific class to which the design element belongs within the styles. Table 1 illustrates this first-level classification scheme. In addition to this basic classification scheme, elements can also be annotated and classified by style-specific and class-specific attributes, as we will illustrate later.

The primary benefit that the use of style brings to this classification scheme is that users are able to determine a lot about a reusable item based solely on its category and style. A style definition provides a set of constraints and guidelines to which all elements claiming to support that style must conform. As a result, a (re)user familiar with the style being considered quickly knows a lot about that design element, including the interface(s) it provides to other design elements and some guarantees about how it will behave when put into a design.

For example, when reusing a "Tool-Invoker" connector (from Table 1), the fact that its class is "Event-Dispatch" indicates that it will support a particular protocol of event interaction, and can be used to connect other components in this style. Similarly, the fact that the "Anomaly-Detector" component (again from Table 1) is a *filter* assures the user that it has an interface that can be connected via pipes to any other filter in a design.

6.2.2. Storage:

Architectural entities provided by the Shelf are stored as objects in a persistent objectbase. These objects are either basic architectural entities (individual components and connectors) or else more complex configurations representing design patterns.

Representation of Basic Elements:

Components. Component storage is best illustrated by example. Consider two basic components built in the *pipe-and-filter* style [AG92] and stored on a Software Shelf. Both are represented as objects derived from the `Filter` class of architectural entities. The first component is a "generic" filter (visible as the dashed box labeled "filter" in figure 1). It defines the basic architectural structure of all `Filter` objects. In particular, any instantiation of it in a design will have a set of input and output ports, its behavior will be a transformation function, etc. It will,

Design Element Name	Category	Style(s)	Class
Kalman-Filter	Component	Pipe-and-Filter	Filter
Anomaly-Detector	Component	Pipe-and-Filter	Filter
Tool-Invoker	Connector	Implicit-Invocation	Event-Dispatch
Proxy	Design Pattern	Object-Oriented	Proxy

Table 1: Sample classifications for four design elements.

however, have no explicit behavioral description that determines what it will actually compute. This information must be provided in the context of a specific design when the filter is instantiated. A "generic" filter, such as this one, can be thought of as a template with a well defined interface and an implicit, albeit simple, behavioral description. This is a useful starting point for creating more specialized filters without having to recreate the structure common to all filters. This simplification can reduce both development time and the potential for subtle interface problems.

The second example component possesses a more detailed behavioral description. This component implements a "Kalman Filter" for reducing the noise in a stream of floating point values passed to it through its single floating-point input port (see the corresponding box in figure 2). It then writes a stream of filtered floating-point values to its single output port. Like the first example, this component possesses a large amount of implicit architectural information simply by being a `Filter` object. In addition, however, it includes explicit behavioral and structural data, in the form of attributes and attached representations. These representations could include: source code, a formal specification, and/or attributes describing details such as performance characteristics or algorithmic choices.

Connectors. In addition to components, a shelf can also store connectors, which define protocols of interaction between components. For example, within the pipe-and-filter style we store several kinds of pipes: a "raw" pipe that is minimally specified, a buffered pipe that specifies a buffer size, and pipes specialized by the form of data that they transmit. To take an example from a different style, a Shelf that supports a distributed message passing style might contain three types of connectors, each more fully instantiated than the previous: `message-pass-channel`, `async-message-pass-channel`, and `buffered-async-message-pass-channel`.

These examples illustrate that the design elements stored on the Shelf are classified and described primarily based on their architectural attributes, instead of just their implementation. Further, it shows how design elements stored on the Shelf can vary greatly in their degree of specificity. Finally, it shows how architectural entities can be stored using straightforward reuse mechanisms.

Design Patterns: Storing basic components and connectors in the repository is a necessary step towards achieving design reuse; it provides the basic building blocks for creating software architectures from reusable entities. In order to more fully exploit the concept of design reuse, however, the Shelf also provides mechanisms for reusing configurations of components and connectors in the form of design patterns. Like basic components and connectors, patterns may be stored on the Shelf in various degrees of instantiation. To expand on the earlier examples, the pattern counterpart to the "raw" filter might be a "raw" pipeline. Figure 1 depicts a two-stage version of the raw pipeline pattern, which is a sequence of two arbitrary filters, each with one or more input ports and one or more output ports, that are composed in a linear sequence with a series of pipe connectors. As a further constraint, none of the pipe connectors can create a cycle. Figure 2 shows a fully instantiated example of the "raw" two-stage pipeline pattern. It includes the `Kalman Filter` component of the previous example and an anomaly detection filter. This pattern is a small configuration that would generally be used as a fragment of a larger architecture. It provides a fully instantiated, albeit simple, noise reduction and anomaly reporting sub-architecture.

A further example: To expand on the previous examples, we now consider a reusable design pattern for a different architectural style. Figure 3 shows a forked memory pattern created in the Real-Time Producer/Consumer (RTP/C) style described in [Jef93]. This pattern shows two shared-memory data stores that are synchronously accessed by a pair of independent RTP/C Components. The architectural style defines the basic semantics for the shared-memory components, the synchronous message passing connectors, and other components and connectors available in the style. It also defines the composition rules specifying how these elements can be combined.

This particular pattern shows a standard composition of elements from this style's design vocabulary; it also adds an additional constraint (above and beyond those imposed by the style itself) on what components can be selected for instantiation in the placeholder positions. Only those components that satisfy the property of being "RTP/C Components" can be selected to "fill-in" the

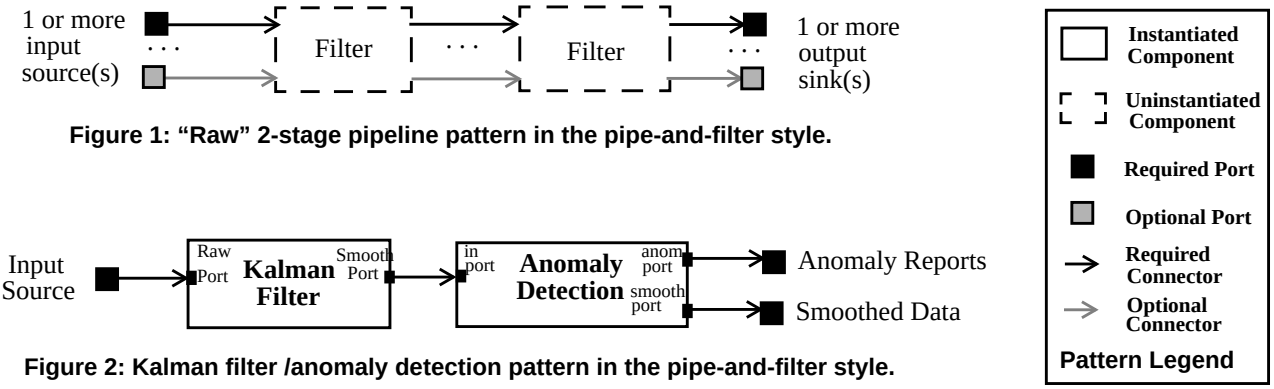

Figure 1: "Raw" 2-stage pipeline pattern in the pipe-and-filter style.

Figure 2: Kalman filter /anomaly detection pattern in the pipe-and-filter style.

dashed-box placeholders. By limiting the design space to include only a specific class of components we are then able to perform some checks on the system design at the architectural level. As an example, if there are formal specifications associated with the components that are chosen to instantiate this pattern, Aesop's tools will be able to check for deadlock-freedom as a property of the design as a whole. Likewise, various performance modelling analyses can be performed on the design itself to determine whether it will be possible to schedule the implemented system on a single processor running at a user-specified speed.

6.2.3. Retrieval:

Classifying and storing design elements are not the only tasks necessary for supporting effective reuse of software designs; users also need mechanisms for browsing, searching, and querying the repository, as well as customizing and/or instantiating the design element or fragment that they have selected.

To support search, the Shelf provides a query language that allows users to perform queries based on stylistic attributes, as well as traditional string and keyword matching. While the query language and search mechanisms are fairly standard, there are three novel features of the Shelf's retrieval mechanism. First, the items being queried and retrieved are design elements (as opposed to concrete code modules). Second, the class and style hierarchies provide a natural taxonomy of elements for searching the repository. Third, and arguably the most important, the use of architectural structures and architectural styles permits automated queries based on the context within which the reused design fragment will be integrated.

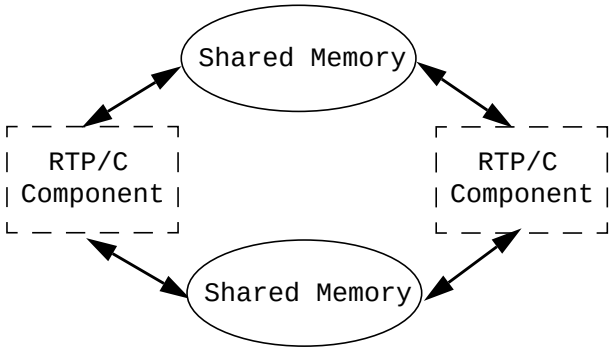

Figure 3: Forked-memory pattern in the Real-time Producer/Consumer (RTP/C) style. All arrows represent synchronous message passing connectors.

Table 2 provides a flavor of the types of style-based queries that a Shelf user can perform. A query is simply a series of predicates joined by "AND" and "OR" junctions. Predicates consist of a keyword attribute -- *Style*, *Type*, *Name*, or *Attribute* -- followed by a relational expression (==, !=, contains, etc.) and a value for that field. The value may be any valid regular expression. An *Attribute* keyword is followed by the name of a style-specific attribute that can be attached to design elements built in that style. The *Attribute* keyword allows styles to define their own attributes to be used as valid query keys.

Although it is readable by humans, we designed the query language with the expectation that many queries to the Software Shelf will be automatically generated. In particular, the standard structure of design elements and their stylistic attributes facilitates context-sensitive queries. For example, the design environment that is interacting with the Shelf can provide the Shelf with a list of the styles of design elements that it is willing to accept. The Shelf can then limit the search space by prepending a disjunctive series of style constraints at the beginning of the query (e.g. {{Style == Foo} OR {Style == Bar}} AND {rest of the query...}). More powerful context-sensitive searches are also possible with this scheme. For example, a user can select a pair of ports on two different components in the design environment (a port is an interface to a component), and have the design environment automatically generate a query for all connectors with an appropriate interface for linking the two components together.

Instantiation and customization: Retrieval may

	Informal Query Description	Shelf Query Language Description
1.	All design patterns in the pipe-and-filter style	{Style == pipe-and-filter} AND {Type == Pattern}
2.	All asynchronous RPC connectors.	{Style == RPC} AND {Type == Connector} AND {Attribute “Asynchronous” == True}
3.	The client-server style pattern called “anonymous-server-group”	{Style == client-server} AND {Type == Pattern} AND {Name == “anonymous-server-group”}
4.	All real-time-message-passing style components with a throughput of 10 packets/unit time	{Style == real-time-message-passing} AND {Type == Component} AND {Attribute “Throughput” == 10}

Table 2: Sample style-based queries on the Software Shelf repository

involve more than just finding an appropriate element. One of the important characteristics of architectural entities is that they can represent abstract or incomplete design fragments. Thus, when retrieving an item from the Shelf, an architect will often need to specialize it to fit within the target design. The Software Shelf provides an instantiation model for retrieval. Once an item has been selected for insertion into a design, an instantiation tool appropriate for the selected item is invoked to guide the architect in specializing the selected design element to meet his or her needs. Instantiation tools are often style-specific and capable of exploiting the knowledge they have about both the design element being instantiated and that element’s style. As a default, the Shelf includes both a generic instantiation tool that provides limited support for specialization of any design element and an identity instantiation tool for retrieving fully instantiated items stored on the Shelf that have no instantiation options.

To illustrate instantiation, consider the process of pulling the patterns in Figures 1 and 2 off of the Shelf and placing them into a design environment. When a user instantiates the raw pattern (Figure 1), the pipe-and-filter instantiation tool recognizes the filter placeholders (represented as dashed boxes in the diagram) and asks the user to select filter components that meet the placeholder criteria. The user can select either zero, one, or two filters from the Shelf to be instantiated in the pattern. Any selected filters will then be placed in the pattern, hooked up appropriately, and instantiated as a configuration in the design environment. Depending on the filters selected it may also be necessary to provide instantiation information for the individual filters. Any dashed boxes that are left empty are instantiated in the design environment as generic filters that can be replaced or specialized at a later point.

The instantiation process for the Kalman Filter pattern (Figure 2) is similar to that for the raw pipeline pattern except the user has no placeholders to fill in. Because the pattern is almost fully specified at an architectural level, the instantiation tool only specializes the individual components and connectors in the pattern. For example, the instantiation tool might request that the user define the type of data expected from the input source (float, integer, etc.) and a threshold value for anomaly detection. This example assumes that the person who put this pattern on the Shelf provided these instantiation options. It is also possible that the pattern is fully specified and supports no further specialization at instantiation time.

7. Evaluation

The Software Shelf has been available as a research prototype for the past year, and is currently being distributed with the Aesop System to interested users. The system supports four styles: a generic architectural style, two kinds of pipe-and-filter styles, and a real-time style. Two of these were illustrated above. We have populated a number of different Shelf repositories with architectural design elements, building both highly specialized repositories, such as a Shelf full of Unix filters, as well as generic repositories that are capable of storing design elements and patterns created in any style. Currently a half dozen sites have installed the Aesop system.

We have built a graphical browser and query tool that implements these ideas on top of the Software Shelf repository. It currently supports simple point and click exploration of the repository and explicit queries. The hypertext-like browsing mechanisms are useful for supporting “accidental discovery,” or the process of learning about the structure and contents of the repository while traversing the repository in search of other design elements. Although we have not yet created large repositories, our experience thus far indicates that the browser should be able to deal with properly structured repositories containing as much as 500 to 1000 architectural design elements.

While feedback from external users is only now becoming available, we have had substantial in-house experience with the Shelf. Currently the Shelf provides a rich repository for each of Aesop’s styles, and it has functioned as a centerpoint for our own system development and demonstration. Ultimately, of course, the long-term viability of the approach will need to be determined by its usefulness to practitioners.

Despite the current lack of an established user base, the research described in this paper takes an important first

step towards establishing a better understanding of both the potential and the pitfalls of architecture-based reuse. In this regard, we have learned four important general lessons that should be of practical benefit to others working in the area of design reuse.

1. **Architectural abstractions can provide a concrete form of design reuse.** One of the problems with "design reuse" is that it is not well defined. What kinds of design knowledge can be captured? How can that knowledge be exploited? This research shows how to make one class of design artifact reusable. By providing mechanisms to represent architectural designs in terms of attributed components, connectors, and configurations, we provide a specific vocabulary with which to represent architectures.

   There are three innovative aspects of these representations. First is the use of connectors as first class reusable entities. In traditional approaches to reuse the interconnection mechanisms are dictated by the programming language (usually procedure call and data sharing). But the approach that we have adopted allows new kinds of interconnection abstractions to be characterized and reused. Second is the use of patterns as partially instantiated configurations. This allows us to support reuse of larger, more complex artifacts. Third, computational components are characterized by multiple interfaces, providing better discrimination of their capabilities for interacting with other componentry.

2. **Standard repository technology is sufficient.** We were somewhat surprised to discover that we needed no new mechanisms for object storage or retrieval. Reuse of architectural designs fits well within an object paradigm, together with its well-understood techniques for object browsing, selection, and documentation.

3. **The benefits of style can vary considerably.** In addition to focusing on architectural artifacts, the primary new thrust of this work is the exploitation of architectural styles. We have found, however, that the ability to take advantage of styles is not uniform. Some styles contain relatively few semantic constraints and a relatively sparse vocabulary. These styles do not permit detailed checking of context constraints, nor do they provide good discriminators for selection of design elements from the Shelf. Other styles, such as the Real-time Producer/Consumer style illustrated earlier, are associated with more detailed classes of information (such as types of data communicated over connectors, protocols of interaction, rates and timing of processing, resource consumption, etc.) This information allows the tools to do a better job of determining when a selected element will fit within a design.

4. **The long-term viability of the approach will require standardized interchange formats.** Currently the Shelf functions as a tool that is tightly bound to the Aesop System. We would like to make it more generally available as a stand-alone tool that could be used independent of Aesop. But this is problematic since there is no general agreement about how architectures should be represented or transmitted between different kinds of architecture-based environments. While it would be possible to write specialized translators from the Shelf to each of these tools, the long term solution will be to use more widely accepted standards for communicating architectural designs. Such standards are beginning to emerge [GMW95], although it will likely be some time before they are widely used.

In addition to the lessons learned, this research raises a number of questions about ways to further exploit architecture-based reuse and architectural styles. These can be viewed as a partial research agenda for others interested in pursuing these ideas.

- **Constraint retention.** As we have illustrated, when an architectural artifact is reused it frequently must be instantiated by supplying missing pieces. The question then arises: how is one allowed to later change aspects of the reused artifact? For example, if you instantiate a pipeline pattern, but later add a pipe that causes it to branch, should this be treated as a design constraint violation or are the constraints strictly intended to guide the initial instantiation process?

- **Architectural property formalism.** In order to exploit styles there must be some representation of the constraints imposed by those styles. For example, in the pipe-filter world, the types of data read and written to the pipe must be the same. In order for tools to check this property it must be (a) stated precisely, and (b) checkable by tools. Research is needed to determine the properties that can be formally stated, and the best ways to state them so that tools can take advantage of them.

- **Combining generators with architecture development environments**. To get practical benefits from architectural design reuse it is important to be able to reuse implementations of architectural artifacts. However, there is usually not a one-to-one association between code fragments and architectural entities. This is obvious in the case of reused patterns. But reusable connectors also present challenges. For example, a typical implementation of pipes requires the use of operating systems communications available through I/O libraries. In this case many different architectural elements – i.e. all of the pipes in the design – use the same implementation. This suggests a strong link between

application generation capabilitie, such as [Bat+94] and the forms of reuse that our tools support.

8. Conclusion

Design reuse would appear to be one of the more promising avenues for improving the prospects for software reuse. But "design reuse" can mean many things. In this paper we have described one approach, based on the reuse of architectural designs. The essential ingredients are (a) the use of components, connectors, and configurations as the basic vocabulary of reusable assets, and (b) the exploitation of architectural style to aid in the classification, retrieval, and instantiation of those assets.

The primary contribution of this work is the integration of the concepts of software architecture and architectural style with traditional mechanisms for performing standard software reuse tasks. We are thus able to leverage prior work to create software design environments that support the reuse of architectural designs, assist designers in selecting appropriate design elements and patterns, and flag potential architectural mismatch problems early in the design and development cycle.

9. Bibliography

[AG92] Allen, R. and Garlan, D. A Formal Approach to Software Architectures. *Proceedings of IFIP '92*. Elsevier Science Publishers B.V., Sept. 1992.

[AAG93] Abowd, G., Allen, R. and Garlan, D. Using Style to Give Meaning to Software Architecture. In *Proc. of SIGSOFT '93: Foundations of Software Engineering*, Software Engineering Notes 118(3), pp 9-20, ACM Press, Dec. 1993.

[Bat+94] Batory, D. et al. The GenVoca Model of Software System Generators. *IEEE Software*, Sept. 1994.

[Bax92] Ira D. Baxter, Design Maintenance Systems. *Comm. of the ACM*, April 1992, pp. 73-87.

[BP89] Biggerstaff, T. and Perlis, A. *Software Reusability*, Vols. 1 and 2, ACM Press/Addison Wesley, 1989.

[GHJV94] Gamma, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[GAO95] Garlan, D., Allen, R, and Ockerbloom, J. Architectural Mismatch, or, Why it's hard to build systems out of existing parts -- Experience Report. *Proc. of the 17th International Conference on Software Engineering*. April 1995.

[GAO94] Garlan, D., Allen, R and Ockerbloom, J. Exploiting Style in Architectural Design Environments. In *Proc. of SIGSOFT '94: Foundations of Software Engineering*, December 1994, ACM Press.

[GMW95] Garlan, D., Monroe, R., and Wile, D. ACME: An Architectural Interchange Language. Carnegie Mellon University Technical Report CMU-CS-95-219, December 1995.

[GR91] Michael M. Gorlick, M. and Razouk, R. Using Weaves for Software Construction and Analysis. *Proc. of the 13th International Conference on Software Engineering.* pp. 23-34, May 1991.

[GS93] Garlan D. and Shaw, M. An Introduction to Software Architecture. *Advances in Software Engineering and Knowledge Engineering, Vol. 1*, New Jersey, 1993. World Scientific Publishing Co.

[Jef93] Jeffay, K. The Real-Time Producer/Consumer Paradigm: A paradigm for the construction of efficient, predictable real-time systems. *Proc.of the 1993 ACM/SIGAPP Symposium on Applied Computing*, ACM Press, February 1993, pp. 796-804.

[KP88] Krasner, G. and Pope, S. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, Aug/Sept. 1988, pp. 26-49.

[Kru92] Krueger, C. Software Reuse, *Computing Surveys*, February 1992.

[Luc+95] Luckham, D. et al. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, April 1995.

[MG92] Mettala, E. and Graham, M.. The Domain-Specific Software Architecture Program. Special Report, Carnegie Mellon University Software Engineering Institute CMU/SEI-92-SR-9. 1992.

[Mor94] Moriconi, M. and Qian, X. Correctness and Composition of Software Architectures, In *Proc. of the ACM SIGSOFT '94: Symposium on Foundations of Software Engineering*, December, 1994.

[Nei84] Neighbors, J. The Draco Approach to Constructing Software from Reusable Components. *IEEE Transactions on Software Engineering*, Vol. SE-10, September 1984 pp. 564-574.

[PDF87] Prieto-Diaz, R. and Freeman, P. Classifying Software for Reusability. *IEEE Software* Volume 4(1), January 1987, pp. 6-16.

[Pree95] Pree, W. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995.

[PW92] Perry, D. and Wolf, A. Foundations for the Study of Software Architecture. *ACM Software Engineering Notes*, 17(4), October 1992, pp. 40-52.

[Shaw+95] Shaw, M. et al. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4):314-335, April 1995.

[SW88] Stovsky, M. and Weide, B. Building Interprocess Communication Models Using STILE. *Proc. of the 21st annual Hawaii Intl. Conference on System Sciences.* Vol. II