Architectural Issues in Software Reuse: It's Not Just the Functionality, It's the Packaging

Mary Shaw
School of Computer Science & Software Engineering Institute
Carnegie Mellon University
Pittsburgh PA 15213

Proc IEEE Symosium on Software Reusability, April 1995

Abstract

Effective reuse depends not only on finding and reusing components, but also on the ways those components are combined. The informal folklore of software engineering provides a number of diverse styles for organizing software systems. These styles, or architectures, show how to compose systems from components; different styles expect different kinds of component packaging and different kinds of interactions between the components. Unfortunately, these styles and packaging distinctions are often implicit; as a consequence, components with appropriate functionality may fail to work together. This talk surveys common architectural styles, including important packaging and interaction distinctions, and proposes an approach to the problem of reconciling architectural mismatches.

1. Software designers use diverse architectural styles

The software architecture group at Carnegie Mellon has studied descriptions of software system architectures and identified a number of patterns that occur regularly [Shaw & Garlan 95]. Some of these patterns govern the overall style for organizing the systems; others determine the character of component interfaces or abstractions for component interaction. A few of the patterns (e.g., objects) have been carefully refined [Booch 86], but others are still used quite informally, even unconsciously. Nevertheless, the idiomatic patterns are widely recognized. System designs often appeal to several of these patterns, combining them in various ways. Inspecting descriptions of actual systems shows that the motivations for using different patterns are often not carefully separated, and the interactions of the patterns are correspondingly obscure.

Systems are composed from identifiable *components* of various types. The components interact in identifiable, distinct ways. Components correspond roughly to compilation units of conventional programming languages and to other user-level objects such as files. *Connectors* mediate interactions among components: they govern component interaction and any auxiliary implementation mechanism required. Connectors do not in general correspond directly to compilation units; they manifest themselves as table entries, instructions to a linker, dynamic data structures, system calls, initialization parameters, servers that support multiple independent connections, and so on. An architectural *style* is based on selected types of components and connectors, together with a *control structure* that governs execution and rules about other properties of the system. An overall *system model* captures the intuition about how these are integrated. Here are the major elements of some popular architectural styles. They have many variations.

Pipeline

System model	mapping data streams to data streams
Components	filters (purely computational, local processing)
Connectors	data streams (ASCII data streams for unix pipelines)
Control structure	data flow

Data abstraction (object-oriented)

System model	localize state maintenance
Components	managers (e.g., servers, objects, abstract data types)
Connectors	procedure call (method invocation is essentially procedure call with dynamic binding)
Control structure	decentralized, usually single thread

Implicit invocation (event-based)

System model	independent reactive processes

Components processes that signal significant events without knowing recipients of signals

Connectors automatic invocation of processes that have registered interest in events

Control structure decentralized; individual components are not aware of recipients of signal

Repository (includes databases and black–board systems)

System model centralized data, usually richly structured

Components one memory, many purely computational processes

Connectors computational units interact with memory by direct data access or procedure call

Control structure varies with type of repository; may be external (depends on input data stream, as for databases), predetermined, or internal (depends on state of computation, as for blackboards)

Interpreter

System model virtual machine

Components one state machine (the execution engine) and three memories (current state of execution engine, program being interpreted, current state of program being interpreted)

Connectors data access and procedure call

Control structure usually state-transition for execution engine; input-driven for selection of what to interpret

Main program and subroutines

System model call and definition hierarchy

Components procedures

Connectors procedure calls

Control structure single thread

Layered

System model hierarchy of opaque layers

Components usually composites; composites are most often collections of procedures

Connectors depends on structure of components; often procedure calls, might also be client-server

Control structure single thread

Notice that each style relies on particular types of components. The characteristics that are significant to the style are not the computational functionality of the components, or the performance, or other properties usually included in specifications. Rather, they are the characteristics that affect the abstractions of the component's interface—how it interacts with other components.

Two components might well have the same apparent functionality but still be non-interchangeable because they are packaged differently. A familiar example is the sort operation in unix, which is available as either a filter or a system call.

Similarly, each architectural style relies on particular kinds of connectors. The connector types for each style are selected to match the component types.

The overall properties of the style—especially its design integrity—depend on this judicious match. Components packaged in other ways may not interoperate properly within the style, and other connectors (or "frameworks" with other interaction rules) may not provide appropriate glue to integrate the system.

2. Reconciling Architectural Mismatch

2.1. Heterogeneity is Inevitable

Advocates of some specific architectural styles argue that all systems should be designed within a single paradigm. This approach is intrinsically flawed.

- Most fundamentally, different architectural styles have different strengths and weaknesses, and a system architecture should be chosen to fit the problem at hand.

In addition,

- Multiple standards for packaging, frameworks, communication, and other architectural issues are bound to exist. Even if one standard dominates at some time, it will eventually change.
- We will always have legacy code that works, doesn't fit the new system, but nevertheless will not be rewritten for some combination of technical and economic reasons.
- Even in restricted communities that share a packaging or interaction standard, there will be differences in interpretations or representation conventions. This can be seen in unix, where a single standard (ASCII) guarantees communication among filters—but filters can still be incompatible because they make different assumptions about how information is represented in the ASCII streams.

Despite these problems, we regularly compose systems from pre-existing fragments. Sometimes we reuse whole components (even, sometimes, from libraries); other times we copy fragments of unpackaged free-standing code .

Most applications devote less than 10% of their code to the overt function of the system; the other 90% goes

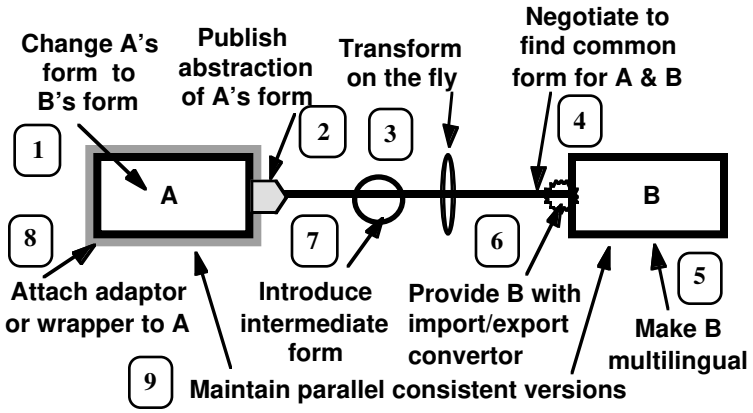
into system or administrative code: input and output; user interfaces, text editing, basic graphics, and standard dialogs; communication; data validation and audit trails; basic definitions for the domain such as mathematical or statistical libraries; and so on.

It would be very desirable to compose the 90% from standard parts. Unfortunately, even if you can find a collection of parts with the right *functionality*, you are likely to discover that they don't conveniently work together. Often the problem is that the parts make *different assumptions* about how data is represented, about the nature of system interactions, about specific details in interaction protocols, or about decisions that are usually explicit (for example, who "owns" the main control thread). Garlan describes in detail the problems encountered in assembling a collection of subsystems that were explicitly intended to be reusable [Garlan et al 95]. A simple example is the sort operation in unix: both a filter and a system call are provided as part of the standard configuration. Although both sort, they are far from interchangeable.

2.2. *Many Ad hoc Tricks Cope with Mismatched Parts*

Software engineers have a wide range of techniques for dealing with architectural mismatch. The simplest setting for examining these techniques is the composition of two components. They might be peer components, a pair of independent applications, a library and a caller, a client and a server, etc.

A and B might fail to work together because they make different assumptions about representations, communication, packaging, synchronization, semantics, control, or other properties. We'll refer to the offending property as the "form". Here are some of the ways the mismatch between A and B might be resolved, together with some common examples. Naturally the relations are symmetric; the roles A and B can be exchanged.



1. *Change A's form to B's form:* It is, in principle, possible (but expensive) to completely rewrite one of the components to work with the other.
2. *Publish an abstraction of A's form:* Application Program Interfaces (APIs) publish the procedure calls used to control a component. Open Interfaces usually provide some abstractions in addition. Projections or views may be used to provide abstractions of databases, especially for federated databases.
3. *Transform from A's form to B's form on the fly:* Some distributed systems do on-the-fly conversions from big-endian to little-endian representations.
4. *Negotiate to find a common form for A and B:* Modems commonly negotiate to find their fastest common protocol.
5. *Make B multilingual:* Macintosh "fat binaries" will execute on either 680x0 or PowerPC processors. Portable unix code will run on many processors.
6. *Provide B with import/export converters:* These come in two important forms. First, standalone applications provide representation conversion services. Several are available for graphics (one of these translates among over 50 formats on 10 platforms) and word processor formats. Second, some systems accommodate extensions or external add-ons that translate to and from foreign formats on demand. Many personal

computer applications come with several of these; some include a table of features that may not be handled properly.

7. *Introduce an intermediate form:* First, external interchange representations, sometimes supported by Interface Description Languages (IDLs), can provide a neutral base. This is especially useful when many components with different forms are involved. Second, standard distribution forms, such as RTF, MIF, Postscript, or Adobe Acrobat provide another alternative—a widespread safe representation. Third, active mediators can be inserted as intermediaries.
8. *Attach an adapter or wrapper to A:* The ultimate wrapper may be the code that causes one machine to emulate another. Software wrappers can mask differences in form, as Mosaic and other Web browsers hide representations of the documents they display.
9. *Maintain parallel consistent versions of A and B:* It is possible, though delicate, for A and B to maintain their own forms and be extremely careful to make all changes to both forms.

These techniques have different advantages and disadvantages. They vary in initial expense, in time and space performance during operation, in flexibility, and in absolute capability. Selecting the appropriate technique is an important design problem. By making both the problem and the alternatives more explicit, we make a first step toward providing good engineering guidance.

2.4. Things to Do to Make Progress

Clearly, this taxonomy of mismatch resolution techniques must be elaborated and refined. This is the objective of ongoing work, and I hope to get feedback and more examples from this workshop.

In order for the descriptive taxonomy to be useful, it must be elaborated with information on behavior with respect to properties of interest. Run-time performance and absolute capability (i.e., interchange representations don't provide any help at all with control mismatches) would be good properties to start with.

Working designers need not only organized information, but also operational guidance on how to apply it. A third stage of progress will be a designer's assistant that offers advice on deciding which technique best fits a given problem. This is especially needed when multiple components with multiple mismatches are involved. Lane [Lane 90] developed a prototype designer's assistant of this kind for the problem of selecting the structure of the user interface component of a system.

3. Research Support

This work reported here was sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency, under grant F33615-93-1-1330, by a grant from Siemens Corporation, and by Federal Government Contract Number F19628-90-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a Federally Funded Research and Development Center. Some of this material was previously covered in position papers for the First Workshop on Pattern Languages for Programming and the 1995 Dagstuhl Workshop on Software Architecture.

4. References

[Booch 86] Grady Booch. Object-Oriented Development. *IEEE Trans. on Software Engineering* SE-12, 2, Feb. 1986.

[Garlan et al 95] David Garlan, Robert Allen, and John Ockerbloom. Architectural Mismatch, or Why it's hard to build systems out existing parts. *Proc 17th International Conf on Software Engineering (ICSE-17),* April 1995.

[Lane 90] Thomas G. Lane. Studying Software Architecture Through Design Spaces and Rules. *Carnegie Mellon University Technical Report* , September 1990.

[Shaw & Garlan 95] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1995, to appear.