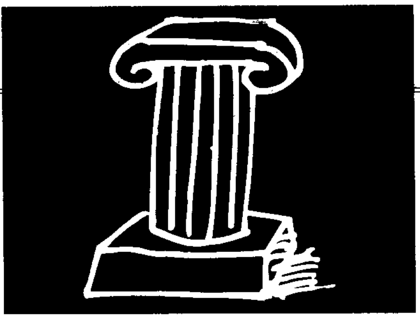
IEEE Software
www.computer.org/software

Architectural Mismatch: Why Reuse Is So Hard

David Garlan, Robert Allen, and John Ockerbloom

Vol. 12, No. 6
November 1995



Architectural Mismatch: Why Reuse Is So Hard

DAVID GARLAN, ROBERT ALLEN, and JOHN OCKERBLOOM
Carnegie Mellon University

◆ *Why isn't there more progress toward building systems from existing parts? One answer is that the assumptions of the parts about their intended environment are implicit and either don't match the actual environment or conflict with those of other parts. The authors explore these problems in the context of their own experience with a compositional approach.*

Future breakthroughs in software productivity may well depend on the software community's ability to combine existing pieces of software to produce new applications. The current build-from-scratch techniques that dominate most software production must eventually give way to techniques that emphasize construction from reusable building blocks. If not, software designers may hit a production ceiling in generating large, high-quality software applications.

The last decade has seen increased support for compositional approaches to software. There is considerable research and development in reuse; industry standards like CORBA have been created for component interaction; and many domain-specific architectures, toolkits, application generators, and other related products that support reuse and open systems have been developed.

Yet the systematic construction of large-scale software applications from existing parts remains an elusive goal. Why? Some of the blame can rightfully be placed on the lack of pieces to build on or the inability to locate the desired pieces when they do exist.

But even when the components are in hand, significant problems often remain because the chosen parts do not fit well together. In many cases these mismatches may be caused by low-level problems of interoperability, such as incompatibilities in programming languages, operating platforms, or database schemas. These are hard

0740-7459/94/$04.00 © 1994 IEEE

problems to overcome, but recent research has been making good progress in addressing many of them.

In this article, we describe a different, and in many ways more pervasive, class of problem, which we term *architectural mismatch*. Architectural mismatch stems from mismatched assumptions a reusable part makes about the structure of the system it is to be part of. These assumptions often conflict with the assumptions of other parts and are almost always implicit, making them extremely difficult to analyze before building the system.

To illustrate how the perspective of architectural mismatch can clarify our understanding of component integration problems, we describe our experience of building a family of software design environments from existing parts. On the basis of our experience, we show how an analysis of architectural mismatch exposes some fundamental, thorny problems for software composition and suggests some possible research avenues needed to solve them.

AESOP SYSTEM

For the last five years, we have been carrying out research in the ABLE (Architecture-Based Languages and Environments) Project at Carnegie Mellon University, which is aimed at developing foundations for an engineering discipline for software architecture. Part of this research is dedicated to finding ways to build tools and environments that will support architectural design and analysis. The box on pp. 20-21 describes the motivation for our work.

The Aesop system was envisioned as the project's implementation platform for experimenting with architectural development environments.[1] It was to be a kind of environment generator that, when given a description of a set of architectural styles, would produce an environment tailored to the development of systems in those styles. The project team completed the first Aesop prototype in August 1993 and has recently built a second prototype.

Aesop provides a toolkit for constructing open, architectural design environments that support architectural styles. The basic idea is that Aesop makes it easy to define new styles and then use them to create architectural designs. Thus, each Aesop environment is configured around a set of styles that guide the designer in producing architectural designs.

Figure 1 shows the Aesop system and the structure of the environments it generates. To produce an environment, Aesop combines a set of style definitions with some shared infrastructure. The shared infrastructure is incorporated into each environment as a set of basic support services for architectural design. The elements of a style definition are a description of

- ♦ an architectural design vocabulary (as a set of object types),
- ♦ visualizations of design elements suitable for manipulation by a graphical editor, and
- ♦ a set of architectural analysis tools to be integrated into the environment.

For each design environment, the set of basic support functions provided by the shared infrastructure includes a design database for storing and retrieving designs; a graphical user interface for modifying and creating new designs; a tool-integration framework that makes it easy to add new tools (such as compilers, architectural analy-

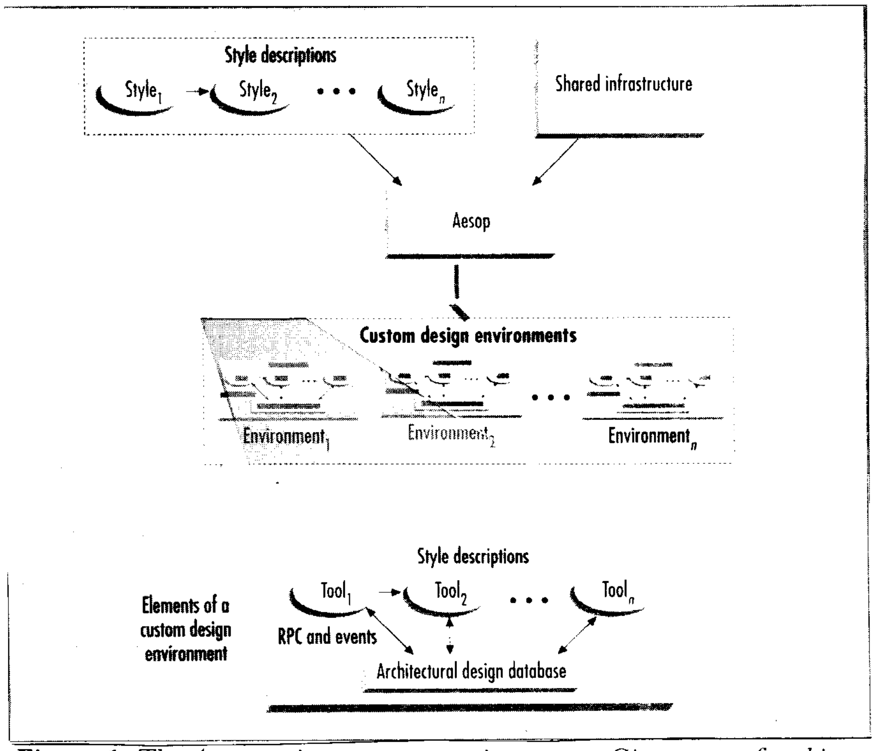


Figure 1. *The Aesop environment-generating system. Given a set of architectural-style descriptions, Aesop produces a custom design environment. All environments have the shared infrastructure provided by Aesop, and they have the same organization: a collection of tools, one of which is a graphical user interface; a database that contains architectural designs; and remote-procedure-call and event-broadcast mechanisms for communication between the tools and the design database.*

sis tools, and so on) to the environment; and a repository mechanism for reusing fragments and patterns from previous designs.

Every Aesop environment has the same structure: As Figure 1 shows, each is an open collection of tools that access an architectural-design database. The database stores architectural designs and provides tools with a high-level, object-oriented interface to architectural designs. The database also manages concurrent access to shared data and enforces the architectural design constraints specified by the architectural styles.

The tools run as separate processes and access the database through a remote-procedure-call mechanism that lets them invoke methods on objects in the database. (The tools may also access databases and file systems outside the Aesop environment, but such access is not relevant here and so is not shown.) Additionally, the environment includes a tool-integration mechanism based on event broadcast.[2] With this mechanism, tools can register to be notified about changes to database objects and announce significant events to other tools. Typical tools include a graphical editor for creating and browsing architectural designs as well as style-specific tools for carrying out architectural analyses, such as checking for architectural consistency, generating code, and interacting with component repositories.

WISHFUL THINKING

We faced two important challenges in building Aesop:

♦ Designing the notations and mechanisms to support style definition.

♦ Creating the infrastructure for the environment-support functions, such as the design manager and tool-integration framework.

In this article we focus on the second challenge.

Viewed abstractly, the infrastructure required by Aesop environments is hardly novel. Indeed, it is now commonplace to construct environments in this fashion, as open, loosely integrated collections of tools that access shared data. Moreover, graphical editors are common components of drawing packages, computer-aided software-engineering tools, and other user interfaces. We were therefore optimistic that we could obtain most of the infrastructure needed for Aesop by building on existing software. Specifically we wanted to reuse four standard pieces:

♦ an object-oriented database,

♦ a toolkit for constructing graphical user interfaces,

♦ an event-based tool-integration mechanism, and

♦ an RPC mechanism.

We had numerous candidates for each piece. In making our selections we picked systems that seemed to have promise for working well together and within our development environment. In particular, we wanted to be sure that we could process the systems using the same compilers, that each piece had been used successfully in several development projects, and that each piece was compatible with the operating system (in this case, Mach) and machine platforms (in this case, Sun machines) on which we were running.

Our choices for the four subsystems were

♦ OBST, a public-domain object-oriented database.

♦ InterViews, a toolkit for constructing graphical user interfaces,[3] developed at Stanford University, which we used with Unidraw, a reusable framework for creating drawing editors that was also produced by the InterViews developers.[4]

♦ SoftBench, an event-broadcast mechanism from Hewlett-Packard, which provides event-based tool integration.[5]

♦ Mach RPC Interface Generator (or MIG), an RPC stub generator, developed at the Carnegie Mellon University, that was well-suited to our host operating system.[6]

WE ENCOUNTERED SIX MAIN PROBLEMS IN INTEGRATING THE FOUR SUBSYSTEMS, VIRTUALLY ALL OF THEM CAUSED BY ARCHITECTURE.

All we had to do was put the subsystems together. It did not appear to be a difficult task, especially since all the subsystems were written in either C++ or C, all had been used in many projects, and all had available source code. We estimated the work would take six months and one person-year.

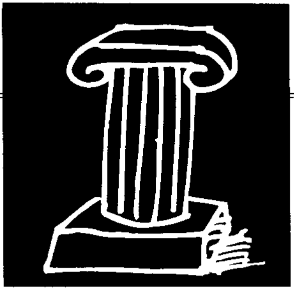
HARSH REALITY

Two years and nearly five person-years later, we managed to get the pieces working together in our first Aesop prototype. But even then, the system was huge (even though we had contributed only a small portion of our own code to the system), the performance was sluggish, and many parts of the system were difficult to maintain without detailed, low-level understanding of the implementations.

Integration problems. We encountered six main difficulties in integrating the four software subsystems:

♦ *Excessive code.* The binary code of our user interface alone was more than 3 Mbytes after stripping. The binary code of our database server was 2.3 Mbytes after stripping. Even small tools (of, say, 20 lines of code) interacting with our system were more than 600,000 lines after stripping! In an operating system without shared libraries, running the central components plus the supporting tools (such as external structure editors, specification checkers, and compilers) overwhelmed the resources of a midsize workstation.

♦ *Poor performance.* The system



operated much more slowly than we wished. Some of the problems occurred because of overhead from tool-to-database communication. For example, saving the state of a simple architectural diagram (containing, say, 20 design objects) took several minutes when we first tried it out. Even with performance tuning, it still took many seconds to perform such an operation.

The excessive code also contributed to the performance problem. Under the Andrew File System, which we were using, files are cached at the local workstation in total when they are opened. When tools are large, the start-up overhead is also large. For example, the start-up time of an Aesop environment with even a minimal tool configuration took approximately three minutes.

♦ *Need to modify external packages.* Even though the reusable packages seemed to run "out of the box" in our initial tests, we discovered that once we combined them in a complete system they needed major modifications to work together at all. For example, we had to significantly modify the SoftBench client-event loop (a critical piece of the functionality) for it to work with the InterViews event mechanism. We also had to reverse-engineer the memory-allocation routines for OBST to communicate object handles to external tools.

♦ *Need to reinvent existing functions.* In some cases, modifying the packages was not enough. We also had to augment the packages with different versions of the functions they already supplied. For example, we were forced to bypass InterViews' support for hierarchical data structures because it did not allow direct, external access to hierarchically nested subvisualizations. Similarly, we ended up building our own separate transaction mechanism that acted as a server on top of a version of the OBST database software, even though the original version supported transactions. We did this so that we could share transactions across multiple address spaces, a capability

SOFTWARE ARCHITECTURE

A critical aspect of any complex software system is its architecture. There is currently no single, universally accepted definition of software architecture, but typically a system's architectural design is concerned with describing its decomposition into computational elements and their interactions. Frequently these descriptions are presented as informal box and line diagrams depicting the gross organizational structure of a system, and they are often described using idiomatic characterizations such as client-server organization, layered system, or blackboard architecture.

Design tasks at this level include organizing the system as a composition of components; developing global control structures; selecting protocols for communication, synchronization, and data access; assigning functionality to design elements; physically distributing the components; scaling the system and estimating performance; defining the expected evolutionary paths; and selecting among design alternatives.

Motivation. Architectural design is important for at least two reasons. First, an architectural description makes a complex system intellectually tractable by characterizing it at a high level of abstraction. In particular, architectural design exposes top-level design decisions and lets the designer reason about how to satisfy system requirements in assigning functionality to design elements. For example, if data throughput is a key issue, an appropriate architectural design would let the designer make systemwide estimates that are based on the values of the throughputs for the individual components.

Second, architectural design lets designers exploit recurring organizational patterns. Such patterns — or architectural styles — ease the design process by providing routine solutions for certain classes of problems, by supporting the reuse of underlying implementations, and by permitting specialized analyses. Consider, for example, an architectural design that uses a pipe-and-filter style. When mapped to a Unix implementation, the system can take advantage of the rich collection of existing filters and the operating system support for pipe communication. Another example is the traditional decomposition of a compiler together with supporting development tools, which has made it possible for undergraduates to build a nontrivial system in a semester course.

Hot research areas. While at present, architectural-design practice is largely ad hoc, the topic is receiving increasing attention from researchers and practitioners. Particularly active areas are

♦ *Architectural description.* Researchers have proposed several new languages for architectural description.[1,2] Among their novel features are the ability to characterize architectural glue (or connectors) as first-class abstractions, the ability to describe patterns of structure and interaction, and the introduction of new forms of system analysis.

♦ *Formal underpinnings.* Several researchers are attempting to provide a sound semantic basis for architectural description and analysis. Most efforts have adapted existing formalisms to the problems of software architecture. Representative formal models include process algebras,[1] partially ordered sets,[2] the Chemical Abstract Machine,[2] and Z.[3]

♦ *Design guidance.* A key issue for architectural design is the ability to codify and disseminate expertise. Ideally, there would be a handbook of the original version did not permit.

♦ *Unnecessarily complicated tools.* Many of the architectural tools we wanted to develop on top of the infrastructure were logically simple sequential programs. However, in many cases it was difficult to build them as such because the standard interface to their environment required them to handle multiple, independent threads of computation simultaneously.

♦ *Error-prone construction process.* As we built the system, modifications became increasingly costly. The time to recompile a new version of the system became quite long and seemingly simple modifications (such as the introduction of a new procedure call) would break the automated build routines. The recompilation time was due in part to the code size. But more significantly, it was also because of interlocking code dependencies that required minor changes to propagate (in the form of required recompilations) throughout most of the system.

Underlying cause. The creators of the reusable subsystems we imported were neither lazy, stupid, nor malicious. Nor were we using the pieces in ways inappropriate to their advertised scope of application. So what went wrong? One possibility is simply that we were poor systems builders, but we suspect our problems are not unfamiliar to anyone who has tried to compose similar kinds of software, and not everyone can be a poor system builder. Therefore, the root causes must lie at a deeper systemic level.

UNDERSTANDING ARCHITECTURAL MISMATCH

Indeed, when we began to analyze our problems through an architectural lens, we realized that we could attribute virtually all our problems to what we now call architectural mismatch, specifically to conflicting assumptions among the parts.

To understand the nature of archi-

software architecture. Among the more recent developments in this area are initial steps toward a description of architectural styles[4] and catalogs of object-oriented design patterns.[5] Other steps are embodied in recent prototype tools for architectural guidance. One of these is our Aesop system, which helps designers conform to stylistic rules. Another is Tom Lane's design assistant for user interfaces.[6]

♦ *Domain-specific architecture.* Several development projects have realized significant improvements by tailoring architectures to an application domain or a product family.[7] Typically, these projects have developed notations and tools that allow specialists in the application domain (as opposed to the software domain) to develop components and systems from high-level descriptions of the desired system behavior.

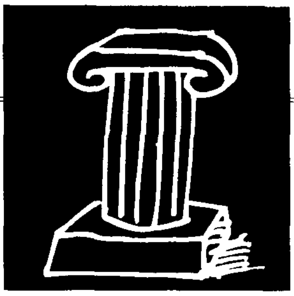
♦ *Architecture in context.* Some researchers have begun to examine the role of software architecture in the broader engineering context of software development processes for architecture, the relationship between architecture and requirements specification, and the use of architectures in software acquisition.[8-9]

♦ *Role of tools and environments.* As architectural design emerges as a discipline within software engineering, it will become increasingly important to support architectural description and analysis with tools and environments. Indeed, we are already seeing a proliferation of environments oriented around specific architectural styles. These environments typically provide tools to support particular architectural design paradigms and their associated development methods. Examples include architectures based on dataflow, object-oriented design, blackboard shells, and control systems.

Unfortunately, each such environment is built as an independent, handcrafted effort. Although development efforts may exploit emerging software environment infrastructures (including persistent object bases, tool-integration frameworks, and user-interface toolkits), the architectural aspects are typically redesigned and reimplemented from scratch for each new style. The cost of such efforts can be quite high. Moreover, once built, each environment typically stands in isolation, supporting a single architectural style tailored to a particular product domain. The Aesop system illustrates one approach to solving this problem.

REFERENCES

1. R. Allen and D. Garlan, "Formalizing Architectural Connection," *Proc. Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., 1994, pp. 71-80.
2. *IEEE Trans. Software Eng.*, special issue on software architecture, Apr. 1994.
3. G. Abowd, R. Allen, and D. Garlan, "Using Style to Understand Descriptions of Software Architecture," *Software Eng. Notes*, Dec. 1993, pp. 9-20.
4. D. Garlan and M. Shaw, "An Introduction to Software Architecture," in *Advances in Software Eng. and Knowledge Eng.*, V. Ambriola and G. Tortora, eds., World Scientific Publishing Co., Singapore, 1993.
5. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Design*, Addison-Wesley, Reading, Mass., 1994.
6. T. Lane, "A Design Space and Design Rules for User Interface Software Architecture," Tech. Report CMU/SEI-90-TR-22 ESD-90-TR-223, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, Nov. 1990.
7. W. Tracz, "DSSA Frequently Asked Questions," *Software Eng. Notes*, Apr. 1994, pp. 52-56.
8. *Proc. First Int'l Workshop Architectures for Software Systems*, D. Garlan, ed., Tech. Report CMU-CS-95-151, Carnegie Mellon University, Pittsburgh, 1995.
9. M. Jackson, "Problems, Methods and Specialization," *IEEE Software*, Nov. 1994, pp. 57-62.



tectural mismatch, it is helpful to view the system as a configuration of components and connectors.[7,8] The *components* are the primary computational and storage entities of the system: tools, databases, filters, servers, and so on. The *connectors* determine the interactions between the components: client-server protocols, pipes, RPC links, and so on. These abstractions are typically expressed informally as box and line drawings, although formal notations for architectural description have begun to emerge, as the box on pp. 20-21 describes.

In terms of components and connectors, we identified four main categories of assumptions that can contribute to architectural mismatch. These categories form a taxonomy for understanding how conflicting assumptions arise.

♦ *Nature of components.* This category includes assumptions about the substrate on which the component is built (infrastructure), about which components will control the computation sequencing (control model) and about the way the environment will manipulate data managed by a component (data model).

♦ *Nature of the connectors.* This category contains assumptions about the patterns of interaction characterized by a connector (protocols) and about the kind of data communicated (data model).

♦ *Global architectural structure.* This category includes assumptions about the topology of the system communications and about the presence or absence of particular components and connectors.

♦ *Construction process.* In many cases the components and connectors are produced by instantiating a generic building block. For example, a database is instantiated, in part, by providing a schema; an event-broadcast mechanism is instantiated, in part, by providing a set of events and registrations. In such cases the building blocks frequently make assumptions about the order in which pieces are instantiated and combined in a system.

ONE OF OUR MOST SERIOUS PROBLEMS WAS DUE TO THE ASSUMPTIONS MADE ABOUT WHAT SOFTWARE PART HELD THE MAIN THREAD OF CONTROL.

CONFLICTING ASSUMPTIONS

Using the definitions just given for components and connectors, the main components of an Aesop environment are the collection of tools and the architectural-design database (which consists primarily of a persistent object base). The main connectors are the communication links of the RPC and event-broadcast mechanism. The parts that we attempted to import provide an implementation basis for two components — the database (via OBST) and the graphical user interface (via InterViews) — and two connectors — RPC (via MIG) and event broadcast (via SoftBench).

Nature of components. Within this assumption category are three main subcategories: infrastructure, control model, and data model.

Infrastructure. One kind of assumption packages make about components is the nature of the underlying support they need to perform their operations. This support takes the form of additional infrastructure that the package either provides or expects to use. In our case, each package assumed it had to provide considerable infrastructure, much of which we did not need. The unwanted infrastructure was part of why we had excessive code.

For example, OBST provided an extensive library of standard object classes to make general-purpose programming easier. However, we needed only a few of these classes because we have a constrained, special-purpose data model.[1]

Additionally, some packages made assumptions about the kind of components that would exist in the final system, and therefore used infrastructure that did not match our needs. For example, the SoftBench Broadcast Message Server expected all the components to have a GUI and therefore used the X library to provide communication primitives. This meant that even tools that did not have their own user interface (such as compilers or design-manipulation utilities) had to include the X library in their executable code.

Control model. One of our most serious problems was due to the assumptions made about what part of the software held the main thread of control. Three of the packages, SoftBench, InterViews, and MIG, use an event loop to deal with communication events. The event loop encapsulates the details of the communication substrate, which lets the developer structure a component's interactions with its environment around a set of callback modules.

Unfortunately, each package uses a different event loop. SoftBench bases its main thread of control on the X Intrinsics package. InterViews provides its own, object-based abstraction of an event loop, implemented directly in terms of Xlib routines. MIG has a handcrafted loop for the server to wait for Mach messages. Each of these control loops is incompatible with the others.

Because the event loops were operating in the same process, we could not use simple event gateways to bridge different event-control regimes. This meant that we had to reverse-engineer the InterViews event loop and modify it to poll for SoftBench events before we could have the user interface respond to events. In the time we had available for the project, we were unable to modify the MIG control loop so that the server could receive events, although we had originally seen this as an important way to provide modular control over the design data.

Data model. The packages also assume certain things about the nature of the data they will be manipulating. For example, Unidraw provides a hierarchical model for its visualization objects. One object can be part of another object, and any manipulation of the parent object (such as changing its position on the screen) results in a corresponding change in the child object. The critical assumption of Unidraw, however, is that *all* manipulations will be of top-level objects. In other words, the user could not change a child object except to have the parent object manipulate it. This was not acceptable. Although the data we wanted to present and manipulate was strongly hierarchical, we wanted the user to have direct control over both parent and child objects. Thus, we had two alternatives: modify Unidraw to support the direct manipulation of children, or create a flat Unidraw data structure and build our own, parallel hierarchy to support the correspondences we wanted. It turned out to be less costly to reimplement the hierarchy from scratch.

Nature of connectors. Within this assumption category are two subcategories: protocols and data model.

Protocols. When we began the project, we expected to have two kinds of tool interactions. The first, a pure event broadcast, involves one tool informing others about the state of the world. For example, the database broadcasts that a particular data object has changed. The second interaction, a request/reply pair, provides a simple means for multiple tools to cooperate in perfoming a complex manipulation. This connector follows the model of a procedure call in that the requesting tool cannot generally continue until the receiving tool completes its task.

The SoftBench Broadcast Message Server provides both these kinds of interaction through different kinds of messages. The *notify* message handles the first kind of interaction. The change is announced and then forgotten by the announcing tool. The *request* and *reply* messages work together to handle the second kind of interaction.

Unfortunately, SoftBench attempts to handle both kinds of interaction uniformly. To receive any message, a tool registers a callback procedure for that message. When the message arrives, the SoftBench client library invokes the callback procedure. This callback technique is used for all three message types (notify, request, and reply). This means that when a tool makes a request, it does not simply make the request, wait for the reply, and then continue — as you would expect. Instead, it must divide its manipulation into two callback routines, one to be done before the call and one to be done after receiving the reply. This breaks up the natural structure of the tool and makes it difficult to understand.

A TRANSLATION OVERHEAD ON EVERY CALL TO THE DATABASE CAUSED THE BIGGEST PERFORMANCE BOTTLENECK.

Moreover, if the server receives any other messages (such as a notify) before the reply message, it delivers those to the tool and invokes its callback procedure before the tool can process the reply. This means, in effect, that if a tool wishes to delegate any part of its processing, then it must be able to deal with multiple threads of control simultaneously, one for each message that might be delivered before a reply is received. Thus, SoftBench's handling of the request/reply protocol forces tools to handle concurrency even if it would be simpler to construct and understand them as sequential programs. For this reason, we ended up using Mach RPC instead of SoftBench for the database interaction because the tool-database path is the most critical and heavily used path in our system.

Data model. Just as the packages make assumptions about the kind of data the components will manipulate, so also do they make assumptions about the data that will be communicated over the connectors. The two communication mechanisms we used, Mach RPC and SoftBench, make different assumptions about the data. Mach RPC supports integration between arbitrary C programs, and so provides a C-based model: data passed through procedure calls is based on C constructs and arrays. SoftBench, on the other hand, assumes that most communication will be about files and the data contained in them, and so all data to be communicated by SoftBench is represented as ASCII strings. Because our tools manipulate primarily database and C++ object pointers, we had to develop translation routines and intermediate interfaces between the different models. The result was that we had translation overhead on every call to the database, which caused the most significant performance bottlenecks in the system. The problem occurred even though we were working in C and C++ exclusively and had compatible data models in all the components we developed.

Global architectural structure. As it turned out, OBST assumes that the communications in the system form a star with the database at the center. Specifically, OBST assumes that all the tools are completely independent of each other. In other words, it assumes that there is no direct interaction between tools, and it views any concurrency among tools as conflict, not cooperation. To protect against this "conflict," OBST selects on a tool-by-tool basis, a mechanism that blocks

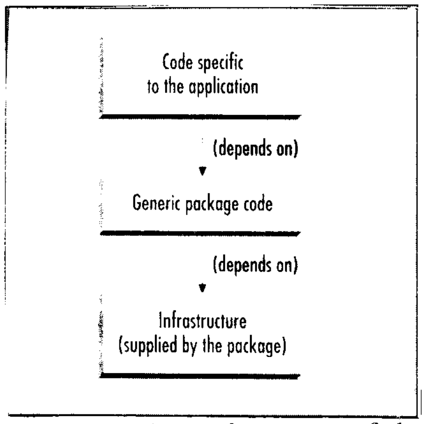


Figure 2. *Assumed structure of the dependencies in the system-construction process. Each layer in the figure represents part of the software ensemble that must come together to instantiate a generic package: infrastructure that the package depends on; generic code of the package itself; and code that specializes the package to the particular application.*

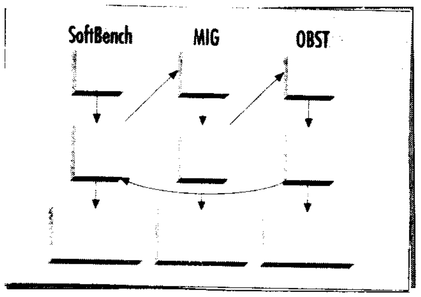


Figure 3. *Actual structure of the dependencies in the system-construction process. The three subsystems are instances of the structure in Figure 2, but rather than existing as independent stacks, the three stacks have interstack dependencies that affect the order in which the pieces must be compiled and combined.*

transactions. Because the tools in an Aesop environment can coordinate their efforts by delegating part of their computation to other tools, this model was totally unacceptable for our purposes. Either cooperating tools would deadlock by holding conflicting locks, or conflicting tools would create inconsistencies when a tool attempted to release the database to a cooperating tool. To solve this problem, we built our own transaction manager as a server on top of OBST.

Construction process. Several of our packages assume that there are three categories of code being combined in the system:

♦ the existing infrastructure (such as the X libraries and the package's own runtime libraries), which would not change;

♦ the application code developed in a generic programming language, which would use the infrastructure but otherwise be self-contained; and

♦ the code developed using the notations specific to the reuse package, which would control and integrate the rest of the application.

Figure 2 shows the assumed dependency structure for building an application from a package. This structure dictated that we should first build the generic parts of the application, then possibly specify them for the package's build tool, and finally preprocess, compile, and link the package-specific sections. Generally, a change to the interface of the generic section meant that we had to both respecify and rebuild the package-specific section. This made sense for each package in isolation, because we think of the packages as providing glue code to integrate the parts of the generic application. For example, MIG assumes that the rest of the code is a flat collection of C procedures, and that its specification describes the signature (name, parameters, and return type) of all these procedures.

In our case, however, more than one package was making these kinds of assumptions. This meant that there were in fact four categories of code: the three previous categories plus the code generated from the other packages. The integration of the code generated by different packages presented the most difficulty in the process of building the system. We had to take the generated code and make it look like whatever generic structure the other packages were expecting.

Figure 3 shows what the build-process dependencies actually were. There are three instances of the structure in Figure 2 — one each for SoftBench, MIG, and OBST. However, as the figure shows, there were dependencies between the instances, which dictated the order in which each must be compiled and combined. To follow these dependencies, we had to, for example, take the output of the OBST preprocessor and specify the resulting procedure calls in MIG's notation, run MIG to generate a server version of the database, and then rebuild all the tools (including rebuilding and linking the SoftBench wrapper code) to recognize the new client interface.

The two sets of conflicting assumptions about the build process resulted in time-consuming and complicated construction.

THE WAY FORWARD?

We believe our experience is typical of any construction that involves assembling large-scale components into a new system. And although some problems encountered will be the result of issues such as language interoperability, platform independence, and heterogeneous data manipulation, the really hard problems — the ones that result from architectural mismatch — do not go away once you solve these low-level problems.

What can be done? We believe that two broad-based approaches are needed to improve the prospects for successful software composition. First, designers must change the way they build components that are intended to be part of a larger system. Second, the software community must provide new notations, mechanisms, and tools that will let designers accomplish this. There are at least four aspects of a long-term solution:

♦ Make architectural assumptions explicit.

◆ Construct large software pieces using orthogonal subcomponents.

◆ Provide techniques for bridging mismatches.

◆ Develop sources of architectural design guidance.

We believe that each of these is ripe for further research and offer a brief outline of possible directions.

Make assumptions explicit. One of the most significant problems is that the architectural assumptions of a reusable component are never documented. True, the current software-design culture is one in which documentation is generally lacking, but the problem goes deeper. Software engineers have neither the proper vocabulary nor the structure to help them express these assumptions. For example, although good documentation of an abstract data type may list preconditions for calling its interface routines, there is no comparable convention or theory for documenting many of the architectural assumptions we described earlier.

Moreover, an architectural view goes beyond the notion of a single component interface. One of the important features of reusable infrastructure is that it must live in a three-dimensional world. As illustrated in Figure 4, the interface at the bottom documents assumptions about lower level infrastructure that the component must interact with. An interface at the top concerns interactions with components that use the reused component as *their* infrastructure. Side interfaces describe interactions with other components at the same level of abstraction. Each of these interfaces can be mismatched in its assumptions about the control model, data model, protocol, and so on.

Of course, documenting assumptions will not make mismatches disappear, but at least it will let designers detect problems early on. Some initial steps toward this goal are emerging in recent work on architecture description languages and formal underpinnings for software architecture, as the box on pp. 20-21 describes.

Use orthogonal subcomponents. Although most large reusable subsystems are themselves constructed out of smaller subcomponents, it is extremely difficult to separate the pieces or change the way in which these subcomponents work together. The software community has known for some time that modules should hide certain design assumptions to increase their chance of reuse. Unfortunately, the *architectural* design assumptions of most systems are spread throughout the constituent modules.

Ideally, designers would be able to tinker with the architectural assumptions of a reused system by substituting different modules for the ones already there. In reality, however, recombining such building blocks will require much more sophisticated processing than link editing, for example, as illustrated in work by Don Batory.[9]

Provide bridging techniques. Even with good documentation and appropriate modularization, mismatches will inevitably occur. Software engineers now use a number of techniques for dealing with such mismatches. In implementing Aesop, we used several of these techniques.

We tried *modifying several components and connnectors* to alleviate mismatches. This technique, however, may require a large investment in reverse-engineering and may be impractical or even impossible for legacy systems or programs, for which source code is often not available.

Another technique is to *install more versatile components and connectors*, either to take over some of the tasks of the original architectural elements, or *to act as mediators* between original elements. Mediation can take place either via smart connectors (connectors that can translate data and communication in multiple protocols) or via mediator components that take over some of the computation.

A special but somewhat common case of mediation involves putting a *wrapper* around a component or connector. The wrapper provides a convenient interface to the rest of the system, and implements its interface by calls to the interface of the original architectural element.

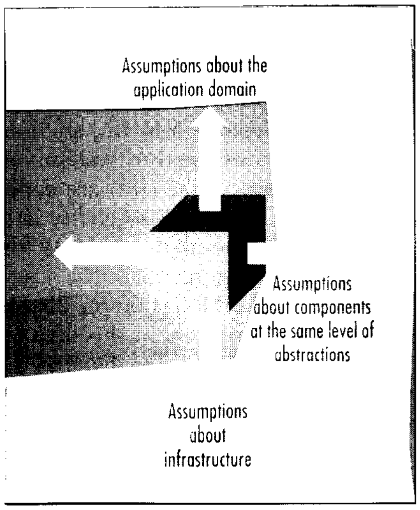

Assumptions about the application domain
Assumptions about components at the same level of abstractions
Assumptions about infrastructure

Figure 4. Three-dimensional view of component interaction.

In rapidly changing systems and environments with many software components *negotiated interfaces* may be appropriate. Components and connectors are built to handle a range of interaction styles, and different elements of the system decide dynamically what sort of communication is most appropriate. Negotiated interfaces are already common in low-level interactions like modem protocols; we would like to see them at the architectural level as well.

These mismatch-bridging techniques are not exhaustive, nor has the software community successfully standardized them. Over time, however, software engineers can expect to see more detailed and comprehensive catalogs of standard techniques, eventually leading to tools that help implement them.

Develop sources of design guidance. Developing good intuitions about what kinds of architectural components work well together is not easy. Designers now rely on trial and error, and it is many years before even skilled designers acquire expertise at putting systems together from parts — and such expertise is typically confined to a specific application domain, such as management information systems or signal processing. The software community must find ways to codify and disseminate principles and rules for software composition.

As we note in the box on pp. 20-21, there is some progress in this area in the form of handbooks for the reuse of design patterns, the creation of architectural design environments, and the development of design tools for certain application domains.

The root causes of the software community's inability to achieve widespread reuse are not going to be solved by low-level improvements in compiling and linking software modules. Rather, the problems are of a deeper, systemic nature. As we have tried to illustrate, viewing assumptions in architectural terms reveals possible ways to explicitly document architectural assumptions and incorporate principled techniques for detecting and bridging architectural mismatches.

This approach opens a rich set of research dimensions, four of which we have outlined. Within the ABLE project, we are investigating many of these. In particular, our most recent implementation of Aesop supports the ability to document certain classes of architectural assumptions. We are currently developing tools that use these annotations to automatically check and repair several of the kinds of architectural mismatch that we have described here. ◆

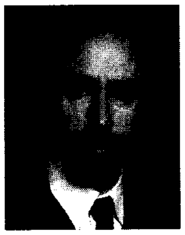
ACKNOWLEDGMENTS

Aesop embodies many ideas from collaborative work with fellow researchers. In particular, our work was strongly influenced by Daniel Jackson, Mary Shaw, and Jeannette Wing, whom we gratefully acknowledge. We also thank the students and staff who have contributed to the Aesop implementation described here: Mike Baumann, Steven Fink, Doron Gan, Huifen Jiang, Curtis Scott, Brian Solganick, and Peter Su. Finally, we thank Ralph Johnson and John Vlissides for their comments on a draft of this article.

This article is based on a paper in *Proceedings of the Seventeenth International Conference on Software Engineering* (IEEE CS Press, 1995). This research was sponsored by the National Science Foundation under grant CCR-9357792, by the Wright Laboratory, Aeronautical Systems Center, US Air Force Materiel Command and the Advanced Research Projects Agency under grant F33615-93-1-1330, and by Siemens Corporate Research. The views and conclusions in this document are ours alone and should not be interpreted as representing the official policies, either expressed or implied, of Wright Laboratory, the US government, or Siemens Corp.

REFERENCES

1. D. Garlan, R. Allen, and J. Ockerbloom, "Exploiting Style in Architectural Design Environments," *Proc. Sigsoft '94*, ACM Press, New York, 1994, pp. 179-185.
2. S. Reiss, "Connecting Tools Using Message Passing in the Field Program Development Environment," *IEEE Software*, July 1990, pp. 57-66.
3. M. Linton, J. Vlissides, and P. Calder, "Composing User Interfaces with Interviews," *Computer*, Feb. 1989, pp. 8-24.
4. J. Vlissides and M. Linton, "Unidraw: A Framework for Building Domain-Specific Graphical Editors," *ACM Trans. Information Systems*, July 1980, pp. 237-268.
5. C. Gerety, "HP SoftBench: A New Generation of Software Development Tools," Tech. Report SESD-89-25, Hewlett-Packard, Software Eng. Systems Div., Fort Collins, Co., 1989.
6. L.R. Walmer and M.R. Thompson, "A Programmers' Guide to the Mach User Environment," tech report, School of Computer Science, Carnegie Mellon University, Pittsburgh, 1988.
7. D. Garlan and M. Shaw, "An Introduction to Software Architecture," in *Advances in Software Eng. and Knowledge Eng.*, V. Ambriola and G. Tortora, eds., World Scientific Publishing Co., Singapore, 1993.
8. D. Perry and A. Wolf, "Foundations for the Study of Software Architecture," *Software Eng. Notes*, Oct. 1992, pp. 40-52.
9. D. Batory et al., "Scalable Software Libraries," *Proc. Sigsoft*, ACM Press, New York, 1993, pp. 191-199.

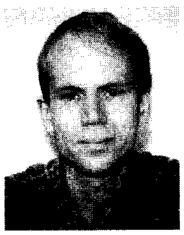


David Garlan is assistant professor of computer science at Carnegie Mellon University. His research interests include software architecture, formal methods as applied to the construction of reusable designs, and software-development environments. He is coauthor with Mary Shaw of *Software Architecture: Perspectives on an Emerging Discipline* (Prentice-Hall, to appear 1996). Garlan is head of the ABLE Project, which focuses on the development of languages and environments to support the construction of software system architectures. Before joining the CMU faculty, he worked in the Computer Research Laboratory of Tektronix, Inc., where he developed formal, architectural models of instrumentation software.

Garlan received a PhD in computer science from Carnegie Mellon University. He is a member of the IEEE and ACM.

Robert Allen is a graduate student of computer science at Carnegie Mellon University. His primary research focus is software architecture, particularly the formalization of architectural descriptions and exploitation of style in the development of families of systems.

Allen received a BS in computer science from Williams College and an MS in computer science from Carnegie Mellon.



John Ockerbloom is a graduate student in the School of Computer Science at Carnegie Mellon University. His primary research focus is composable systems and software architecture. His current work is in support for the distributed growth of structured data types in wide area information systems. He is a member of the ABLE project and was active in the early implementation of Aesop.

Ockerbloom received a BS in computer science from Yale University and an MS in computer science from Carnegie Mellon University.

Address questions about this article to Garlan at CS Dept., Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213-3891; garlan@cs.cmu.edu.