CSci 658: Software Language Engineering
Architectural Mismatch

H. Conrad Cunningham

17 February 2018

Contents

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-5358

Advisory: The HTML version of this document requires use of a browser that supports the display of MathML. A good choice as of February 2018 is a recent version of Firefox from Mozilla.

Architectural Mismatch

These notes accompany discussion of the paper [Garlan 1995]:

- David Garlan, Robert Allen, and John Ockerbloom. Architectural Mismatch: Why Reuse is So Hard, *IEEE Software*, Vol. 12, No. 6, November 1995.

Introduction

The ability to systematically construct new software systems from pre-existing components remains an elusive goal.

Why?

- In some cases the needed components are not available.
- In other cases seemingly appropriate components are available, but they do not fit together well because of:
  - incompatible low-level characteristics (e.g., in programming language, platform, data format, etc.)
  - incompatible assumptions about the overall structure and operation of the system of which the component is a part

The authors call the latter type of incompatibility an *architectural mismatch*.

Architectural mismatches are usually more subtle and pervasive than the low-level incompatibilities. Frequently the architectural assumptions of a component are implicit; they are seldom documented and often not even acknowledged by the designer of the component.

Aesop

The authors of the paper, who are software architecture researchers at Carnegie Mellon University, discovered the problem as they were developing Aesop, an implementation platform for experimenting with architectural development environments.

Aesop generates an environment as an open collection of tools built around a shared architectural-design database. The tools execute as separate processes and access the database via remote procedure calls.

Aesop also uses event-broadcast system as a tool-integration mechanism. A tool can register to receive notification of changes to certain database objects; it can also announce important events to other tools via the system.

Example tools include a graphical editor for creating, examining, and modifying architectural design descriptions, managers for the repositories of components, analysis and consistency checking tools, and code generators.

In the design of Aesop, the authors sought to reuse existing components:

- an object-oriented database system (the public domain OBST system)
- a graphical user interface toolkit (InterViews and Unidraw from Stanford University)
- an event-based tool-integration mechanism (SoftBench from Hewlett Packard)
- a remote-procedure-call mechanism (the Mach RPC Interface Generator from Carnegie Mellon)

The chosen tools seem quite stable; all have been used on several projects. They also seem compatible; all are implemented using either C or C++ with source code available. Thus the task of building an Aesop prototype did not seem to be a massive effort—estimated at approximately one staff-year of work during a six-month schedule.

Unexpected Problems

The authors' rosy development plan failed. The first prototype took five times as long and five times as much effort to get working—approximately five person-years of work during a two-and-a-half-year period. And then the resulting system performed very sluggishly and was very difficult to maintain.

The authors identify six primary difficulties they had in integrating the four components.

1. **Code bloat**

   Any program that interacted with the Aesop system grew excessively large in size. A simple 20-line program might end up as 600,000 ones of code when all the interfaces were included.

2. **Poor performance**

   The programs performed poorly because of the overhead of the tool-to-database communication and because of the excessive code size.

3. **Need to modify the pre-existing packages**

   Some of the tools have subtle incompatibilities or deficiencies that required considerable time to understand and remedy.

   For example, the event mechanisms in SoftBench and InterViews are not completely compatible. Since the event communication mechanism was a critical piece of functionality, the code had to be modified for Aesop.

4. **Need to reimplement some existing functions**

   Even though a capability was present in a pre-existing component, it was sometimes necessary to reimplement the capability to allow it to work as needed with the other components in Aesop.

   For example, the authors replaced the OBST transaction mechanism by a version that allows the sharing of transactions across multiple address spaces.

5. **Unnecessarily complex code**

   Simple sequential tools often had to become quite complex multithreaded tools because of the need to work through the standard interface.

6. **Error-prone construction process**

   Building a system from its sources was a very time-consuming process. Because of the large code size, compilation was slow. Because of the high degree of interdependence, a simple code change might cause the need to recompile the entire system. The automated build procedures proved quite fragile.

What went wrong?

In analyzing the problem, the authors discovered that most of the problems resulted from architectural mismatch. The architectural assumptions made by the various components are in conflict.

Components and Connectors

To understand the architectural mismatch, it is helpful to view a system as made up of components and connectors.

***components*:** the high-level computational and data storage entities in the system.
***connectors*:** the interactions among the components.

The authors identify four primary categories of assumptions that can lead to architectural mismatch.

1. **Nature of components**

   - infrastructure -- what kind of underlying structure the components are built upon
   - control model -- what components control the sequencing of activities
   - data model -- how the component manages the data

2. **Nature of connectors**

- protocols -- what patterns of interaction are associated with a connector
- data model -- what kinds of data do the connector transmit

3. **Global architectural structure**
    - topology of the system's communication structure
    - presence or absence of particular components or connectors in the system
4. **Construction process**
    - order in which the various components and connectors are instantiated and combined into the system

Conflicting Assumptions in Aesop

Aesop architecture:

- components -- the various tools and the architectural-design database
- connectors -- the communication links provided by remote procedure call and event broadcasts

Nature of components conflict

- **Infrastructure.** The pre-existing packages assume they have to provide a large infrastructure, much of which is not needed in Aesop.

  For example, OBST provides a large library of standard objects for general purpose programming. The inclusion of these unneeded elements contribute to the code bloat.

  Some of the packages make assumptions about the nature of other components in the system.

  For example, SoftBench assumes all components have their own GUI interfaces and, hence, have the X library's communication primitives loaded. Thus, to use SoftBench's event system, all other components must load the X library even though they do not need X otherwise—resulting in more code bloat.

- ***Control model.*** The different packages also make different assumptions about which portion of the system holds the main thread of control.

  In particular, SoftBench, InterViews, and the Mach Interface Generator all use event loops to handle communications with other components. Unfortunately, these three are implemented with mechanisms that are incompatible with one another when executed in the same process. For Aesop, the InterViews loop had to be modified to work with SoftBench's X-based events. There was not time to do the modification of the Mach events.

- ***Data model.*** The packages assume different things about the nature of the data and how it can be manipulated.

  For example, Unidraw maintains a hierarchical model for its visual objects and only allows the top-level objects to be directly manipulated by users of the package. Child objects can only be manipulated by their parent objects.

  However, Aesop requires that both parent and child objects can be directly manipulated by users. Thus the authors chose to reimplement the hierarchy to get the capability needed by Aesop.

Nature of connectors conflict

- ***Protocols.*** The original Aesop design needed two types of connectors: an event broadcast and a remote procedure call (i.e., a request/wait-for-reply interaction).

  SoftBench, the chosen tool integration package, supports both types of communication mechanisms. It handles the two more-or-less uniformly by mapping the remote procedure call within the event framework. The remote procedure call is broken into two events—a request and a reply.

  Unfortunately, SoftBench's handling of the request/reply pair adds considerably to the complexity of the caller's code. The caller must break its processing of the "procedure call" into two callback routines. The caller must also handle concurrency; it may need to do other processing between a request and the corresponding reply.

  To avoid the complications of SoftBench's handling of remote procedure calls, the authors switch to use of Mach's RPC mechanism.

- ***Data model.*** SoftBench's event mechanism and Mach's RPC assume quite different things about the nature of the data to be transmitted. Mach's RPC supports communication among arbitrary C programs and, hence, supports transmission of C data types. SoftBench assumes that the data transferred will be ASCII strings.

  Since the Aesop tools work with general C++ objects, translation between the formats is often necessary. Every call of the database involves a considerable amount of overhead. This developed as one of the most significant performance bottlenecks in Aesop.

Global architectural structure conflict

OBST assumes that all communications in the system occurs in a star configuration, with itself at the center of the star. It assumes that there is no direct interaction between other tools. It's transaction mechanism is designed to control access and maintain consistency in that type of environment.

However, the Aesop's communication structure is a more general graph and its tools must often cooperate without involving the OBST database. OBST's transaction mechanism can result in deadlock or database inconsistency in this environment.

As a result, the authors found it necessary to build their own transaction mechanism to run on top of OBST.

Construction process conflict

Several of the packages used assume that there are three categories of code in the system:

a. the unchanging, lower-level infrastructure it assumes (e.g., the package's own runtime library and the X Window library)

b. the user's application code (written in C or C++) that uses the infrastructure but is otherwise self-contained

c. code generated by the package that controls and integrates the rest of the application. (The user may have supplied information for generating this code in the package's "scripting language".)

However, several of the packages used in Aesop took such an approach. This added a fourth category of code to the above three.

d. code generated by other packages in the system.

The code in the fourth category makes the build process more complicated and fragile.

For example, to build systems involving SoftBench, OBST, and Mach Interface Generator (MIG), the authors had to "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." ... Doesn't sound like fun to your instructor!

Recommendations

The authors suggest that a long-term solution to the problem of architectural mismatch should have at least the following aspects:

1. **Make architectural assumptions explicit.**

   Creation of good documentation is a problem at all levels of design. In the code, the problem is probably more cultural and managerial than technical. For example, good mechanisms exist for documenting code, but they are often not used in practice.

   Documentation of the architectural level has the additional problem of having less well-developed concepts, notations, and tools for supporting the documentation. Architectural description languages is an area of current research.

   Instructor's note: Part of the authors' research deals with architecture description languages and their formal underpinnings.

2. **Construct large software pieces using orthogonal subcomponents.**

   In most current systems, the *architectural* design assumptions are spread throughout the constituent modules. In general, it is very difficult to separate a system into pieces or change the way the components work together.

   The principle of information hiding seems just as important at the architectural level as it is at the code level. Architectural assumptions that might change should be hidden (encapsulated) within modules, so that a change in the assumptions would only affect a few modules of the system.

3. **Provide techniques for bridging mismatches.**

   The authors suggest several techniques for alleviating the problems resulting from architectural mismatches.

   - ***Modify several components and connectors to relieve the mismatch problem.***

     This, of course, may not be possible for components where the source code is not available to the system developer.

   - ***Install more versatile components and connectors.***

     These can either assume the role of the original element or they can act as *mediators* between the original element and other elements in the system. For example, a mediator might translate data or communication protocols between what the original element and other elements in the system.

   - ***Introduce a negotiated interface between two elements.***

     In this type of design, the elements are constructed to support a range of possible interaction styles. The actual choice of which style to use in a situation is determined dynamically by "negotiation" among the components.

4. **Develop sources of architectural guidance.**

   Some progress is being made in this area. Various systems of *design patterns* are being recorded and disseminated in books and articles. Architecture- and application-specific software frameworks are being developed.

Acknowledgements

I wrote these notes to accompany the discussion of the Architectural Mismatch paper [Garlan 1995] in the first offering of my Software Architecture topics course in Spring 1998.

I subsequently revised it for use in later offerings of my Software Architecture and Software Families courses.

In Spring 2018 I reformatted these notes from HTML to Pandoc Markdown for possible use in my CSci 658 Software Language Engineering course.

I maintain these notes in the Pandoc dialect of Markdown using embedded LaTeX markup for and mathematical formulas, and then translate the notes to HTML, PDF, and other forms as needed.

References

[**Garlan 1995**] David Garlan, Robert Allen, and John Ockerbloom. Architectural Mismatch: Why Reuse is So Hard, *IEEE Software*, Vol. 12, No. 6, November 1995.

Concepts

TODO