Architectural Mismatch: Why Reuse is Still So Hard

David Garlan  
*Carnegie Mellon University,* garlan@cs.cmu.edu

Robert Allen  
*IBM,* roballen@us.ibm.com

John Mark Ockerbloom  
*University of Pennsylvania,* ockerblo@pobox.upenn.edu

Follow this and additional works at: https://repository.upenn.edu/library_papers

Part of the Software Engineering Commons, and the Systems Architecture Commons

**Recommended Citation**


Appeared in IEEE Software 26:4, July/August 2009, pp. 66-69. Copyright 2009 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

This paper is posted at ScholarlyCommons. https://repository.upenn.edu/library_papers/68

For more information, please contact repository@pobox.upenn.edu.
Architectural Mismatch: Why Reuse is Still So Hard

Abstract
In this article, David Garlan, Robert Allen, and John Ockerbloom reflect on the state of architectural mismatch, a term they coined in their 1995 IEEE Software article, "Architectural Mismatch: Why Reuse Is So Hard." Although the nature of software systems has changed dramatically since the earlier article was published, the challenge of architectural mismatch remains an important concern for the software engineering field.

Keywords
architectural mismatch, software architecture, software reuse

Disciplines
Software Engineering | Systems Architecture

Comments
Appeared in IEEE Software 26:4, July/August 2009, pp. 66-69. Copyright 2009 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

This journal article is available at ScholarlyCommons: https://repository.upenn.edu/library_papers/68
Architectural Mismatch: Why Reuse Is Still So Hard

David Garlan, Robert Allen, and John Ockerbloom

Vol. 26, No. 4
July/August 2009

This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder.
update

25th-anniversary top picks

In 1995, when we published "Architectural Mismatch: Why Reuse Is So Hard" (an earlier version of which had appeared elsewhere), we had just lived through the sobering experience of trying to build a system from reusable parts but failing miserably. Although the system had the required functionality, developing it took far longer than we had anticipated. More important, the resulting system was sluggish, huge, brittle, and difficult to maintain.

Why had things gone so awry? The usual explanations for reuse failure did not seem to apply. The parts had been engineered for reuse. We were reasonably skilled implementers. We had the source code and were familiar with all the parts’ implementation languages. We knew what we wanted, and we used the parts in accordance with their advertised purposes.

In searching for answers, we realized that virtually all our problems had resulted from incompatible assumptions that each part had made about its operating environment. We termed this phenomenon “architectural mismatch,” and our article tried to explore in more depth how and why it occurs.

The Problem
Specifically, we examined four general categories for assumptions that can lead to architectural mismatch:

- the nature of the components (including the control model),
- the nature of the connectors (protocols and data),
- the global architectural structure, and
- the construction process (development environment and build).

We also noted three facets of component interaction in which assumptions can lead to mismatch:

- the infrastructure on which the component relies,
- application software that uses the component (including user interfaces), and
- interactions between peer components.

Figure 1 illustrates these facets.

Finally, we argued that to make progress,
two things would be necessary. First, designers must change how they build components intended to be part of a larger system. Second, the software community must provide new notations, mechanisms, and tools that let designers accomplish this.

**The World Has Changed**

In the decade and a half since that publication, the state of the practice in component-based reuse has changed dramatically. The problems we identified might seem behind us. Today’s software systems routinely build on many layers of reusable infrastructure (for example, for distributed communication and remote data access), interact with users through standard interfaces (for example, Web browsers), and use large corpuses of open source software (for example, Apache Tomcat). They also have sophisticated development environments that provide direct access to reuse libraries (for example, Eclipse and NetBeans), and they exploit services created in a global virtual operating environment. Indeed, for every line of code that developers write, they reuse thousands of lines written by someone else.

But has the problem gone away, or has it simply found a new home in a more modern setting?

**The State of Architectural Mismatch Today**

Three basic techniques exist for dealing with architectural mismatch. One is to prevent it. Another is to detect it when it does occur, hopefully early in the development life cycle, when you can easily consider alternatives. The third is to repair it when it is unavoidable. Modern software development methods have made advancements in each of these techniques.

**Preventing Architectural Mismatch**

This technique has benefited from developments in a number of areas, including architectural specialization, open source practices, and virtualization and common user interfaces.

**Architectural specialization.** One way to help prevent architectural mismatch is to work in an architecturally specialized design domain. Specialization restricts the range of permissible components and the interactions between them, thereby eliminating some of the variability that contributes to mismatch.

Figure 2 illustrates common points in the specialization space. At the far left are completely unconstrained architectures. (This would arguably include the system we described in our original article.) Moving to the right, architectures must fit in a narrower design context—for example, generic styles, such as data flow and call-return. More specific still are specializations of those styles, such as pipes and filters. Further to the right are component integration standards, which typically dictate the kinds of connectors you can use, the kinds of interfaces that components must have, and the global control structures. Next are domain-specific integration standards, and to the far right are product lines.

**Figure 2. The spectrum of architectural specialization.** The figure depicts representative points along a spectrum that characterizes the degrees of specialization, or domain specificity, of a class of architectures. Elements below the axis are examples of architectures in each class.
Applications must include extensive code to ensure compatibility with different browsers, languages, and library implementations.
component to have a service interface, they must almost completely rewrite the application—for example, to decouple application code from its user interface.

**New Challenges**

Not only do the mismatch problems we noted in our article persist, but today’s computing landscape also introduces new challenges.

**Trust**

One crucial issue that Internet-scale software raises is trust between components. Numerous security breaches have resulted from software that was not sufficiently hardened for the variety of imperfect or malicious software that could interact with it in unanticipated ways. At the same time, software components that are fully hardened to deal with untrustworthy software can have significantly higher performance overhead and development costs than components running in a more trusted environment. So, finding appropriate matches in trust between components can be essential.

**Dynamism**

Our 1995 article portrayed mismatch as a development-time problem, occurring before system deployment. Today, however, systems must increasingly support dynamic reconfiguration to cope with component failure, variable resources, and changing user needs. This requirement leads to a new concern for ways to avoid, detect, and repair mismatch dynamically. This problem is substantially more difficult because composition must be achieved in the presence of ongoing computation.

**Architecture Evolution**

The scenario in our article involved creating a new system from existing parts. Today, a much more common situation is an existing system that evolves over its lifetime. From an architectural perspective, new components or connectors might need to be introduced, old systems might need to interoperate with others, and standards and frameworks might change. So, we must consider how to evolve an architecture, factoring in the costs and risks of architectural mismatch that might occur.

Although the set of advancements we have briefly touched on here is hardly exhaustive, we believe that architecture mismatch will be an issue for some time to come. Indeed, as the level of reuse and the complexity of assumptions made by reusable parts increase, architecture mismatch becomes even more of an issue requiring the software engineering community’s attention. We hope that other people will continue to report not only on successes and new techniques but also on failures in this area. As we saw with our original article and its reception, we often learn more from frank discussion of what goes wrong than from promotion of what we hope will be right.

**About the Authors**

**David Garlan** is a professor of computer science and the Director of Professional Programs in Software Engineering at Carnegie Mellon University. His research interests include software architecture, cyberphysical systems, autonomic systems, and software engineering education. Garlan has a PhD in computer science from Carnegie Mellon University. He is a member of the ACM and IEEE. Contact him at garlan@cs.cmu.edu.

**Robert Allen** is a software engineer in IBM’s Systems and Technology Group. He received his PhD in computer science from Carnegie Mellon University. Contact him at raballen@us.ibm.com.

**John Ockerbloom** is a digital-library planner and architect at the University of Pennsylvania. His research interests include distributed software architectures supporting interoperability, information discovery and ontologies, and digital preservation. Ockerbloom has a PhD in computer science from Carnegie Mellon University. Contact him at ockerblo@pobox.upenn.edu.

**References**


For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdil.