Architectural Mismatch: Why Reuse Is *Still* So Hard

David Garlan, *Carnegie Mellon University*

Robert Allen, *IBM*

John Ockerbloom, *University of Pennsylvania*

In 1995, when we published "Architectural Mismatch: Why Reuse Is So Hard"[1] (an earlier version of which had appeared elsewhere[2]), 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.

In January 2009, I asked for follow-up pieces from several sets of authors whose insightful and influential *Software* classics made the magazine's 25th-anniversary top-picks list (Jan./Feb. 2009, pp. 9–11). Here, David Garlan, Robert Allen, and John Ockerbloom provide fresh perspectives on their winning article, addressing how their thinking has evolved over the years, what has changed, and what has remained constant.

—Hakan Erdogmus, Editor in Chief

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,

0740-7459/09/$25.00 © 2009 IEEE

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.

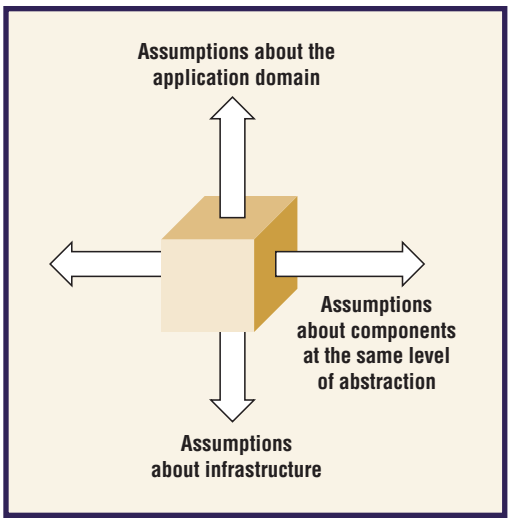


Figure 1. Three facets of component interaction.[1] Each facet identifies a set of assumptions that a component may make about its environment.

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.[3,4] 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.

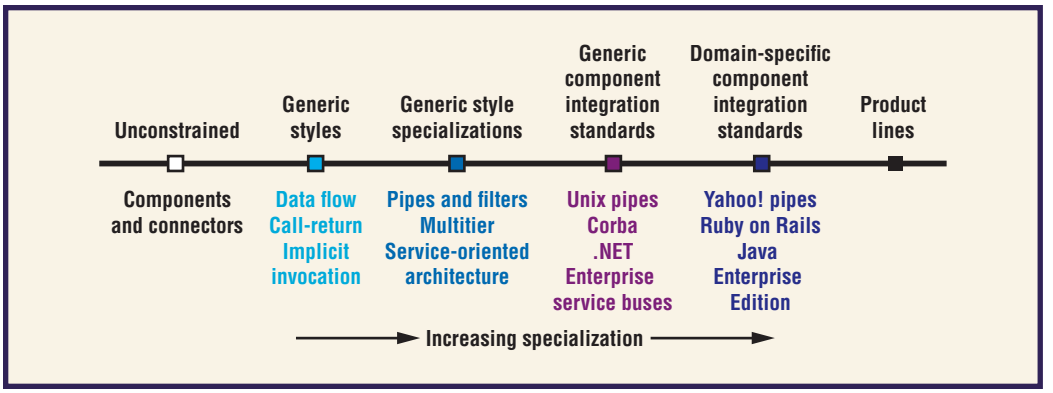


Figure 2. The spectrum of architectural specialization. The figure depicts representative points along a spectrum that characterizes the degrees of specialization, or domain specificity, of a class of architectures. Elements below the axis are examples of architectures in each class.

Open source. Many open source software communities are developing and distributing software collaboratively over the Internet. Widespread global collaboration, and the social and informational infrastructure that arises with it, can prevent many instances of architecture mismatch in two ways. The first is by standardizing on particular frameworks and architectural styles. The second is by producing a body of experience and examples that clarify which architectural assumptions and application domains go with a particular collection of software.

Some open source communities have also developed common build conventions such as the standard four-step build procedure that many CPAN (Comprehensive Perl Archive Network) modules use. The scale of open source software development means that there are now often multiple implementations of similar functionality. So, developers can more easily find appropriate software packages that are compatible with a given architecture.

Virtualization and common user interfaces. High-level communication protocols and data standards, as well as common language and browser environments such as JVM (Java Virtual Machine) and Ajax, make it easier to develop software that operates in a common virtual environment running on a variety of low-level infrastructures. These advances help eliminate some mismatch arising from platform incompatibilities and different user interface requirements, two of the three dimensions of Figure 1.

Continuing problems. Although the developments we just mentioned can help reduce architectural mismatch, they have not eliminated the problem. Specialization works well only if you can shoehorn your application into that domain. Moreover, combining systems and parts from multiple styles tends to be the norm, not the exception. Open source can provide a reusable baseline, but such systems' quality (for example, their documentation and extensibility) varies considerably. Virtual environments are often implemented inconsistently on different platforms. This turns the ideal of "write once, run anywhere" into the reality of applications that must include extensive code to ensure compatibility with different browsers, languages, and library implementations. Particular frameworks go in and out of fashion, raising the possibility that a software application can begin its life in a richly supported environment that eventually becomes obsolete and incompatible with newer software components.

Applications must include extensive code to ensure compatibility with different browsers, languages, and library implementations.

Detecting Architectural Mismatch

This technique has benefited from a number of developments in such areas as documentation standards and process guidance.

Documentation standards. One positive development is guidance on how to document architectures to make assumptions explicit. Such documentation is often based on multiple "views" because different stakeholders might care about different classes of assumptions.[5,6]

Taking this a step further, standardized architecture description languages now let you document certain assumptions. For example, you can use UML to document a component's provided and required resources, as well as various forms of component behavior. Some of these languages can also document extrafunctional attributes, such as timing behavior and resource consumption. Similarly, service-oriented architectures (SOAs) often use standard interface description languages, such as WSDL (Web Services Definition Language), and let you document additional assumptions through service-level agreements.

Unfortunately, standards do little to combat architectural mismatch if few people use them. In practice, today's architectural documentation remains impoverished at best. Furthermore, commonly used documentation languages generally do not support tool-assisted detection of architectural mismatch, limiting these standards' usefulness.

Process guidance. Software processes derived from the spiral model and based on iterative, risk-driven development are seeing increased acceptance. Many of these emphasize early "architectural" prototypes, with the explicit purpose of exposing architectural issues such as mismatch early in a system's life cycle. Although these processes do not provide specific techniques for detecting mismatch, it is relatively easy for developers to observe in a functioning system, as we mentioned in our article. However, although process guidance might lead to early detection, it unfortunately does not provide diagnostic or corrective power.

Repairing Architectural Mismatch

In many cases, avoiding mismatch is impossible. Consequently, the past decade has seen increased research on ways to repair it. Examples include mechanisms such as wrappers, adapters, mediators, and bridges, many of which have been cataloged in software design and architecture books. Additionally, some frameworks provide built-in mechanisms such as protocol and data adaptors to integrate legacy components and services that would not otherwise work together.

Although these techniques can help, they address only a small part of the problem, and only in narrowly constrained situations. For example, developers trying to integrate a legacy stand-alone application into an SOA often find that to "wrap" the

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.

Architecture Lock-In

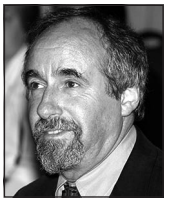
Even supposing that you have appropriately modeled evolution's cost, the potential for architectural mismatch might eventually make changing an existing system too expensive to allow effective innovation. Once you have successfully developed a system in an architectural style using a given infrastructure, moving it to a new setting without introducing crippling mismatch might involve nearly as much effort as its original development. This problem might significantly affect the economics of future computing and software platforms such as Web services or clouds, as inflexibilities and the high cost of changing providers hinder free competition.

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.

References

1. D. Garlan, R. Allen, and J. Ockerbloom, "Architectural Mismatch: Why Reuse Is So Hard," *IEEE Software*, Nov./Dec. 1995, pp. 17–26.
2. D. Garlan, R. Allen, and J. Ockerbloom, "Architectural Mismatch, or, Why It's Hard to Build Systems Out of Existing Parts," *Proc. 17th Int'l Conf. Software Eng.* (ICSE 95), IEEE CS Press, 1995, pp. 179–185.
3. M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.
4. F. Buschman et al., *Pattern-Oriented Software Architecture: A System of Patterns, Vol. 1*, John Wiley & Sons, 1996.
5. P. Clements et al., *Documenting Software Architectures: Views and Beyond*, Addison-Wesley, 2003.
6. *Recommended Practice for Architectural Description of Software-Intensive Systems: ANSI/IEEE Std 1471 :: ISO/IEC 42010*, IEEE, 2009; www.iso-architecture.org/ieee-1471.

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 roballen@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.