IEEE Software
www.computer.org/software

The Artistry of Software Architecture

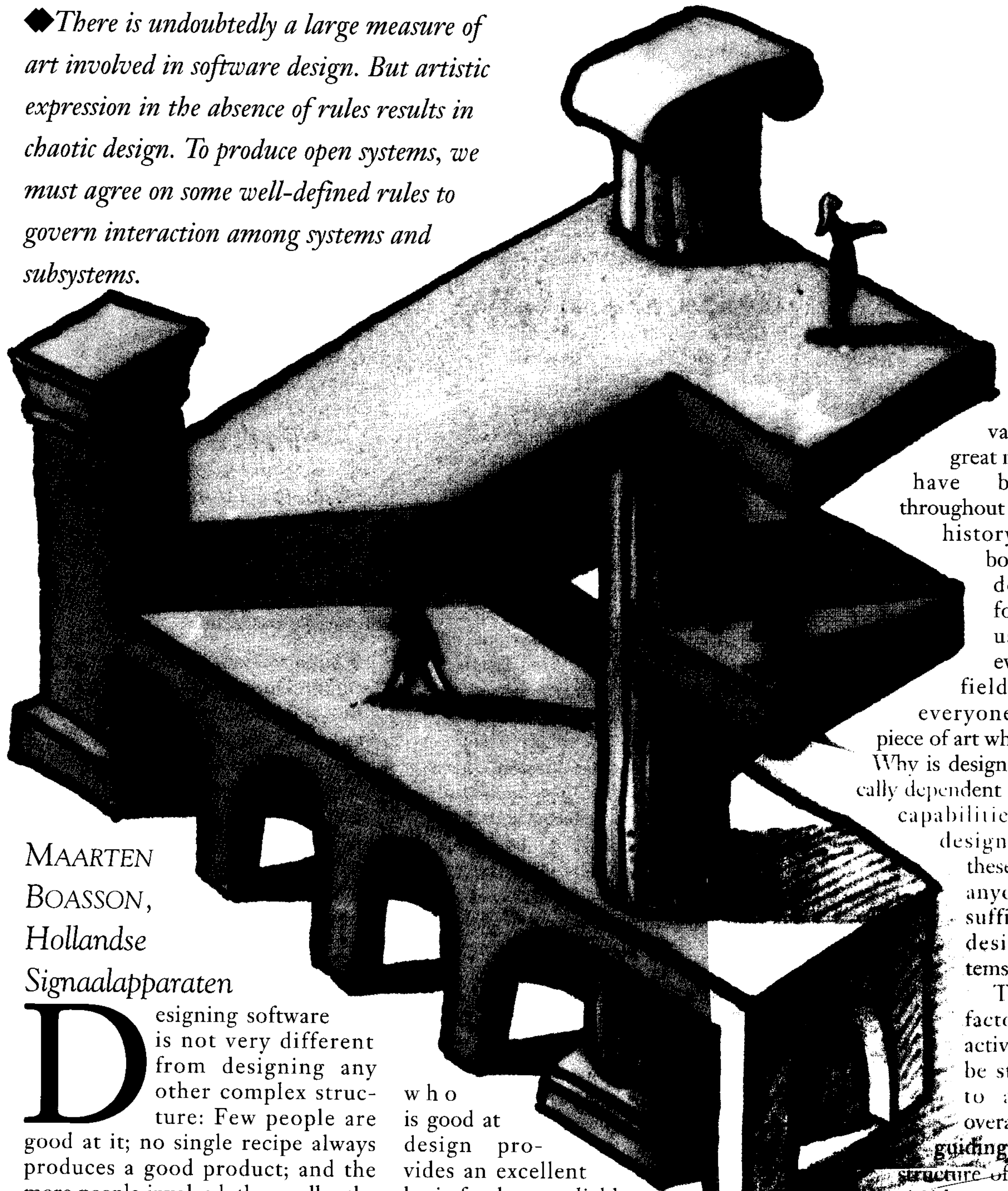
Maarten Boassan

Vol. 12, No. 6
November 1995

GUEST EDITOR'S INTRODUCTION

The Artistry of SOFTWARE ARCHITECTURE

◆*There is undoubtedly a large measure of art involved in software design. But artistic expression in the absence of rules results in chaotic design. To produce open systems, we must agree on some well-defined rules to govern interaction among systems and subsystems.*



MAARTEN BOASSON, *Hollandse Signaalapparaten*

Designing software is not very different from designing any other complex structure: Few people are good at it; no single recipe always produces a good product; and the more people involved, the smaller the probability of success. On the other hand, a design produced by someone who is good at design provides an excellent basis for long, reliable service. In exceptional cases, a good software design is no less valuable than the great masterpieces that have been created throughout our rich cultural history. Examples of both bad and good designs can be found all around us, in almost every engineering field; practically everyone recognizes a piece of art when they see it.

Why is design quality so critically dependent on the skills and capabilities of a single designer? What are these skills? And can anyone learn them sufficiently well to design good systems?

The common factor in all design activities appears to be strict adherence to a well-chosen overall structure and guiding principle. The structure of physical things like bridges and buildings is largely dictated by the laws of nature, which limit a designer's choices.

ARTICLE SUMMARIES: SOFTWARE ARCHITECTURE

♦ **Architectural Mismatch: Why Reuse Is So Hard, pp. 17-26**

David Garlan, Robert Allen, and John Ockerbloom

Architectural mismatch stems from mismatched assumptions a reusable part makes about the system structure 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.

♦ **Comparing Architectural Design Styles, pp. 27-41**

Mary Shaw

One of the more difficult decisions designers face in this area is selecting an appropriate architectural style.

In this article, I examine 11 designs for an automobile cruise-control system. Most of the designs appeal to multiple styles, but they generally fall into four main groups: object-oriented architectures, including information hiding; state-based architectures; feedback-control architectures; and architectures that emphasize the system's real-time properties.

It is my hope that this evaluation will not only make it easier to understand the relative merits of different architectural design idioms, but also serve as a springboard for analyzing the remaining obstacles to practical architectural design at the system level.

♦ **The "4+1" View Model of Software Architecture, pp. 42-50**

Philippe Kruchten

The 4 + 1 View Model describes software architecture using five concurrent views, each of which addresses a specific set of concerns: The logical view describes the design's object model, the process view describes the design's concurrency and synchronization aspects; the physical view describes the mapping of the software onto the hardware and shows the system's distributed aspects, and the development view describes the software's static organization in the development environment. Software designers can organize the description of their architectural decisions around these four views and then illustrate them with a few selected use cases, or scenarios, which constitute a fifth view. The architecture is partially evolved from these scenarios.

The 4+1 View Model allows various stakeholders to find what they need in the software architecture. System engineers can approach it first from the physical view, then the process view; end users, customers, and data specialists can approach it from the logical view; and project managers and software-configuration staff members can approach it from the development view.

♦ **Creating Architectures with Building Blocks, pp. 51-60**

Frank J. van der Linden and Jürgen K. Müller

At Philips Communications Industry (PKI), we develop embedded telecommunication-infrastructure systems. Because we must deliver each product in site-specific configurations — of which there are many — and because the development of such systems is a major investment, we must create a *product family* rather than a single product. We organize system construction according to three design dimensions covered by the system architecture: structure, aspects, and behavior.

Of the three dimensions, we consider structure to be the most important. In this dimension, reducing complexity is our main concern. We thus organize system functionality into four layers, or subsystems. These subsystems are composed of software modules — 'building blocks' — which are the basic software entities in the system architecture.

The Building-Block Method is an architectural method. It does not prescribe the precise method you should use to develop the building blocks. You can use different methods within one system according to the specific requirements for each building block. You can also use formal or informal specifications for building blocks, depending upon your application domain.

♦ **Implementing Dialogue Independence, pp. 61-70**

Drasko M. Sotirovski and Philippe B. Kruchten

Dialogue independence — the decoupling of the Computer-Human Interface from the core application software — can be achieved simply through an appropriate architectural framework, with no loss of efficiency. We show that the objective of dialogue independence can be decomposed into three separate subgoals that a software architecture must resolve: existence, property, and transition. We identify architectural patterns that satisfy all three subgoals, and give a rough sketch of their design and implementation.

We chose an air-traffic-control system to illustrate our proposed decomposition because of our experience with it and because it exposes many of the difficulties inherent in a typical, large CHI software architecture. We use the terminology of object-oriented software architecture, but we propose a decomposition that is independent of the methodology used.

Related Articles to Appear

We were unable to accommodate these two articles in this issue. They will appear in a future issue. In the meantime, the unedited articles are accessible through our Web site: http://www.computer. org/pubs/software/software.htm

♦ **Architectural Design of a Common Operating Environment**

Shawn Butler, David Diskin, Norman Howes, and Kathleen Jordan

We have developed a Common Operating Environment for a Global Command and Control System. The underlying design principle of our effort has been to reduce complexity and use architectural components that are supported by existing commercial products and standards.

♦ **A Generic Model for Software Architectures**

Wilhelm Rossak, Vassilka Kirova, Leon Jololian, Harold Lawson, and Tamar Zemel

We present an architecture-based approach for developing domain-specific, large and complex software systems within the context of the Engineering of Computer Based Systems.

Best Copy Available

Pure art has no such restrictions, and artists have used this freedom more or less successfully throughout history. Classical artists created their own sets of rules for each period because the mind cannot deal effectively with the unlimited freedom that comes when all rules are suspended. Modern artists consciously abandoned structural rules, which accounts for the chaotic nature of many contemporary pieces of art.

DESIGN AS AN ART

Software design is both similar to and different from artistic design. Software designers are not limited to physical restrictions, but neither are they guided by rules that have proven their usefulness over a very long period of refinement. As with art, this easily results in chaotic designs that fail to meet even the most elementary requirements, equivalent to structural integrity and safety in physical designs. In fact, the potential for chaos is ever greater, because software is subject to much more modification during its lifetime than other artifacts (no one would think of modifying Rembrandt's *Nightwatch*, the Eiffel Tower, or the Golden Gate Bridge!).

As we have tried to come to grips with the difficulties of large-system software development, we have paid a lot of attention to improving the structure of the design process. The research areas labeled "structured programming," "software engineering," and, more recently, "software process modeling" have significantly contributed to a better understanding of the problems.

Unfortunately, the benefits of the software-engineering tools that rapidly came into existence — and are themselves now large, complex systems — have been very heavily oversold. As a result, much attention has been focused on improving and extending tools to better support the design process — as if better drawing boards would result in better bridges and buildings.[1] The better role for such tools is to speed development once a good system structure has been conceived. Modern bridges are possible because we now have tools that support the verification of structural ideas, not the other way around.

This leads to the all-important concept of a system's architecture. Here I use the term "architecture" to mean a system structure that consists of active modules, a mechanism to allow interaction among these modules, and a set of rules that govern

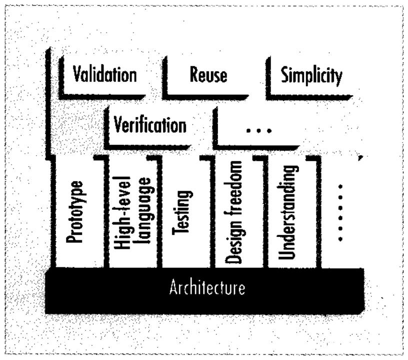


Figure 1. Relationship between solutions, techniques, and architecture.

the interaction. In this issue, "The 4+1 View Model of Architecture," by Phillipe Kruchten expands this definition to include many more aspects of a system (and see a related article, "A Generic Model for the Use and Specification of Software Architecture" by Wilhelm Rossak, Vassilka Kirova, Leon Jololian, and Harold Lawson in an upcoming issue).

In contrast to this ideal approach, a typical development approach results in an architecture with a separate set of rules for almost every interface. If the size of the rule set is used to grade an architecture, these architectures would score extremely low. Limiting this size is a useful goal, and yet a rule set that is too small reduces interaction possibilities, potentially to the point where it is impossible to implement the required functionality. Some large-system development processes do adopt an architecture-centered approach (as described by Kruchten in this issue or by Grady Booch in an address to the Software Technology Conference last spring[2]).

Figure 1 illustrates the central role played by architecture. As the figure shows, developers today use a plethora of techniques to try to address the cost-explosion problem observed in nearly all large-system developments. Without a suitable architecture, it is difficult to apply these techniques individually, let alone in combination.

DATA-CENTRIC ARCHITECTURE

Clearly, architecture plays a key role in the development process, and without a suitable architecture developers can achieve little. Today, the major question is: What rules should we follow to successfully develop systems that meet all functional and nonfunctional (performance, extensibility, and fault-tolerance) requirements. There is no one answer to this question; in fact there are several.

At a very high level, there are two distinct approaches: The first orients the architecture around the system's functions; the other around a global data model. The majority of efforts, past and present, have focused on the functional paradigm, which defines how individual functions interact, synchronize, and communicate.

Many researchers have proposed a solution to the architecture problem (Mary Shaw describes a number of them in the article "Comparing Architectural Design Styles"). The majority of these solutions, however, result from a particular system-design approach and suffer (somewhat) from their need for special instances of generic architectural rules to govern each individual interaction.

This is not the inevitable result of a function-oriented approach, but instead largely reflects the strong tendency to adopt information-hiding. Information hiding, in this context, has been universally interpreted as the need for data abstraction, but this is not the only possible approach. A better alternative is a dual approach, in which you develop a global data model first and then develop the functions necessary to bring these postulated data elements into existence. This approach leads to a radically different architecture. Relatively little has been published in this area, but the few experiments and products that have used this approach have had extremely promising results (see, for example, "The Architectural Design of the Common Operating Environment for the Global Command and Control System" by Shawn Butler, David Diskin, Norman Howes, and Kathleen Jordan in an upcoming issue and related articles.[3-6]).

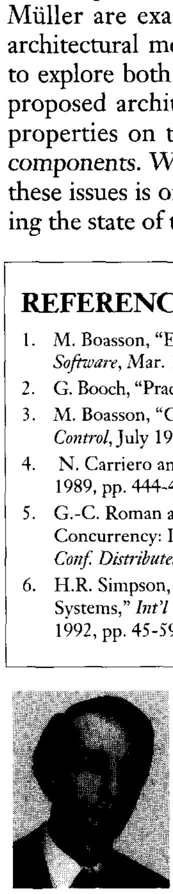
Different types of systems — or even different subsystems within a single system — may require different architectures, as described by Drasko Sotirovsky and Philippe Kruchten in "Implementing Dialogue Independence." Architecture choice is crucial — it may be the difference between a successful project and a failure (as David Garlan describes in "Architectural Mismatch: Why Reuse Is So Hard"). As open systems are increasingly emphasized, we clearly need standards that will allow us to couple systems with different internal architectures. This requires not only well-defined connectors (plugs that fit outlets), but also well-understood data semantics (outlets and appliances made for the same voltage).

In this area there are still many open questions, and the problem is only beginning to be recognized as crucial. This will change in the next few years, and the industry may recognize that a data-centered approach has important advantages after all. Even as implementations of concepts like the Common Object Request Broker Architecture becomes readily available, software developers still doubt their universal applicability and continue to seek other architectural models.

The Rossak article and "Creating Architectures with Building Blocks" by Frank van der Linden and Jürgen Müller are examples of numerous attempts to formalize architectural models. We need a sound mathematical basis to explore both the potential benefits and the limitations of proposed architectures and to ultimately prove a system's properties on the basis of the properties of its individual components. We still have a long way to go — research into these issues is of the greatest possible importance in advancing the state of the art of large-system development. ◆

REFERENCES

1. M. Boasson, "Exploding Complexity May Be Our Own Fault," *IEEE Software*, Mar. 1993, p. 12.
2. G. Booch, "Practical Software Engineering," *CrossTalk*, July 1995, pp. 4-13.
3. M. Boasson, "Control Systems Software," *IEEE Trans. Automatic Control*, July 1993, pp. 1094-1107.
4. N. Carriero and D. Gelernter, "Linda in Context," *Comm. ACM*, Apr. 1989, pp. 444-458.
5. G.-C. Roman and H.C. Cunningham, "A Shared Dataspace Model of Concurrency: Language and Programming Implications," *Proc. Int'l Conf. Distributed Computing Systems*, 1989, pp. 270-279.
6. H.R. Simpson, "Real Time Networks in Configurable Distributed Systems," *Int'l Workshop Configurable Distributed Systems*, IEE, London, 1992, pp. 45-59.



Maarten Boasson is head of Signaal's Applied Systems Research Department and a member of Thomson's Scientific and Technical Council, where he is responsible for research and long-term development strategy in distributed, real-time systems. His research focuses on preventing unnecessary complexity in system design. Boasson has a master's degree in mathematics from Gorningen University. He is a member of the IEEE Computer Society, ACM, and the American Association of Artificial Intelligence, and cofounder of the IEEE task force on computer-based systems engineering.

Address questions about this issue to Boasson at Hollandse Signaalapparaten BV, PO Box 42, 7550 GD Hengelo, The Netherlands; boasson@hgl.signaal.nl.