What is a software architecture?

Level: Introductory

Peter Eeles, Senior IT Architect, IBM

15 Feb 2006

from The Rational Edge: This introduction to the relatively new discipline of software architecture is the first of a four-part series on "architecting" in general. The author begins by defining the discipline's key terms and goes on to explore what a well-designed architecture contributes to the environment in which it is deployed.

More dW content related to: **software architecture**

There is no doubt that the world is becoming increasingly dependent on software. Software is an essential element of the ubiquitous cell phone, as well as complex air traffic control systems. In fact, many of the innovations that we now take for granted -- including organizations such as eBay or Amazon -- simply wouldn't exist if it weren't for software. Even traditional organizations, such as those found in the finance, retail, and public sectors, depend heavily on software. In this day and age, it's difficult to find an organization that isn't, in some way, in the software business.

In order for such innovations and organizations to survive, the software they depend on must provide the required capability, be of sufficient quality, be available when promised, and be delivered at an acceptable price.

All these characteristics are influenced by the architecture of the software, the subject of this article. My focus here is on "software-intensive systems," which the IEEE defines as follows:

A software-intensive system is any system where software contributes essential influences to the design, construction, deployment, and evolution of the system as a whole. [from IEEE 1471. See the "Architecture defined" section below.]

In this article, the term "architecture," when unqualified, is synonymous with the term "software architecture." Although this article focuses on software-intensive systems, it is important to remember that a software-intensive system still needs hardware in order to execute and that certain qualities, such as reliability or performance, are achieved through a combination of software and hardware. The hardware aspect of the total solution cannot therefore be ignored. This is discussed in more detail later in this article.

Architecture defined

There is no shortage of definitions when it comes to "architecture." There are even Websites that maintain collections of definitions.[1] The definition used in this article is that taken from IEEE Std 1472000, the IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, referred to as IEEE 1471.[2] This definition follows, with key characteristics bolded.

*Architecture is the fundamental **organization** of a **system** embodied in its **components**, their **relationships** to each other, and to the **environment**, and the principles guiding its design and evolution. [IEEE 1471]*

This standard also defines the following terms related to this definition:

*A **system** is a collection of components organized to accomplish a specific function or set of functions. The term system encompasses individual applications, systems in the traditional sense, subsystems, systems of systems, product lines, product families, whole enterprises, and other aggregations of interest. A system exists to fulfill one or more **missions** in its environment. [IEEE 1471]*

*The **environment**, or context, determines the setting and circumstances of developmental, operational, political, and other influences upon that system. [IEEE 1471]*

*A **mission** is a use or operation for which a system is intended by one or more **stakeholders** to meet some set of objectives. [IEEE 1471]*

*A **stakeholder** is an individual, team, or organization (or classes thereof) with interests in, or concerns relative to, a system. [IEEE 1471]*

As we can see, the term "component" is used throughout these definitions. However, most definitions of architecture do not define the term "component," and IEEE 1471 is no exception, as it leaves it deliberately vague to cover the many interpretations in the industry. A component may be logical or physical, technology-independent or technology-specific, large-grained or small-grained. For the purposes of this article, I use the definition of component from the UML 2.0 specification; and I use the term fairly loosely in order to encompass the variety of architectural elements that we may encounter, including objects, technology components (such as an Enterprise JavaBean), services, program modules, legacy systems, packaged applications, and so on. Here is the UML 2.0 definition for "component":

> [A component is] a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. A component defines its behavior in terms of provided and required interfaces. As such, a component serves as a type, whose conformance is defined by these provided and required interfaces (encompassing both their static as well as dynamic semantics).[3]

The definitions provided here cover a number of different concepts, which are discussed in more detail later in this article. Although there is no generally agreed definition of "architecture" in the industry, it is worth considering some other definitions so that similarities between them can be observed. Consider the following definitions where, again, I've bolded some of the key characteristics.

*An architecture is the **set of significant decisions** about the organization of a software system, the selection of **structural elements** and their interfaces by which the system is composed, together with their **behavior** as specified in the collaborations among those elements, the **composition** of these elements into progressively larger subsystems, and the **architectural style** that guides this organization -- these elements and their interfaces, their collaborations, and their composition. [Kruchten][4]*

*The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those **elements**, and the **relationships** among them. [Bass et al.][5]*

[Architecture is] the organizational structure and associated behavior of a system. An architecture can be recursively decomposed into parts that interact through interfaces, relationships that connect parts, and constraints for assembling parts. Parts that interact through interfaces include classes, components and subsystems. [UML 1.5][6]

The software architecture of a system or a collection of systems consists of all the important design decisions about the software structures and the interactions between those structures that comprise the systems. The design decisions support a desired set of qualities that the system should support to be

successful. The design decisions provide a conceptual basis for system development, support, and maintenance. [McGovern][7]

Although the definitions are somewhat different, we can see a large degree of commonality. For example, most definitions indicate that an architecture is concerned with both structure and behavior, is concerned with significant decisions only, may conform to an architectural style, is influenced by its stakeholders and its environment, and embodies decisions based on rationale. All of these themes, and others, are discussed below.

An architecture defines structure

If you were to ask anyone to describe "architecture" to you, nine times out of ten, they'll make some reference to structure. This is quite often in relation to a building or some other civil engineering structure, such as a bridge. Although other characteristics of these items exist, such as behavior, fitness-for-purpose, and even aesthetics, it is the structural characteristic that is the most familiar and the most-often mentioned.

It should not surprise you then that if you ask someone to describe the architecture of a software system he's working on, you'll probably be shown a diagram that shows the structural aspects of the system -- whether these aspects are architectural layers, components, or distribution nodes. Structure is indeed an essential characteristic of an architecture. The structural aspects of an architecture manifest themselves in many ways, and most definitions of architecture are deliberately vague as a result. A structural element can be a subsystem, a process, a library, a database, a computational node, a legacy system, an off-the-shelf product, and so on.

Many definitions of architecture also acknowledge not only the structural elements themselves, but also the composition of structural elements, their relationships (and any connectors needed to support these relationships), their interfaces, and their partitioning. Again, each of these elements can be provided in a variety of ways. For example, a connector could be a socket, be synchronous or asynchronous, be associated with a particular protocol, and so on.

An example of some structural elements is shown in Figure 1. This figure shows a UML class diagram containing some structural elements that represent an order processing system. Here we see three classes -- OrderEntry, CustomerManagement, and AccountManagement. The OrderEntry class is shown as depending on the CustomerManagement class and also the AccountManagement class.

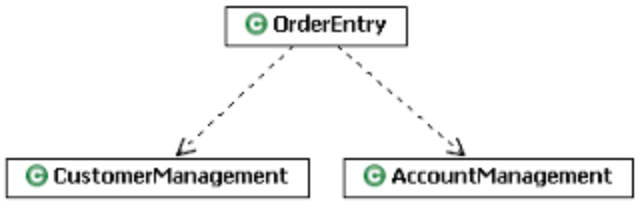

Figure 1: UML class diagram showing structural elements

An architecture defines behavior

As well as defining structural elements, an architecture defines the interactions between these structural elements. It is these interactions that provide the desired system behavior. Figure 2 shows a UML sequence diagram showing a number of interactions that, together, allow the system to support the creation of an order in an order processing system. Here we see five interactions. First, a Sales Clerk actor creates an order using an instance of the OrderEntry class. The OrderEntry instance gets customer details using an instance of the CustomerManagement class. The OrderEntry instance then uses an instance of the AccountManagement class to create the order, populate the order with order items, and then place the

order.

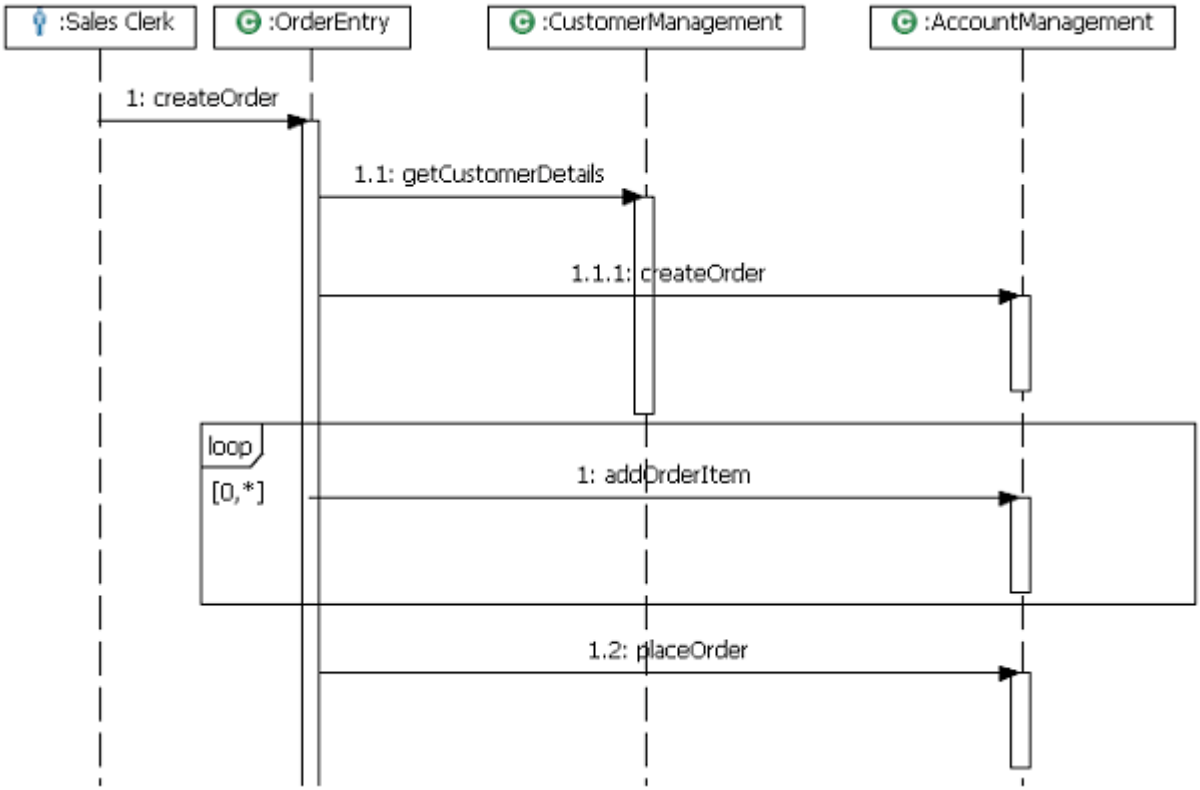

Figure 2: UML sequence diagram showing behavioral elements

It should be noted that Figure 2 is consistent with Figure 1 in that we can derive the dependencies shown in Figure 1 from the interactions defined in Figure 2. For example, an instance of OrderEntry depends on an instance of CustomerManagement during its execution, as shown by the interactions in Figure 2. This dependency is reflected in a dependency relationship between the corresponding OrderEntry and CustomerManagement classes, as shown in Figure 1.

An architecture focuses on significant elements

While an architecture defines structure and behavior, it is not concerned with defining *all* of the structure and *all* of the behavior. It is only concerned with those elements that are deemed to be significant. Significant elements are those that have a long and lasting effect, such as the major structural elements, those elements associated with essential behavior, and those elements that address significant qualities such as reliability and scalability. In general, the architecture is not concerned with the fine-grained details of these elements. Architectural significance can also be phrased as economical significance, since the primary driver for considering certain elements over others is the cost of creation and cost of change.

Since an architecture focuses on significant elements only, it provides us with a particular perspective of the system under consideration -- the perspective that is most relevant to the architect.[8] In this sense, an architecture is an abstraction of the system that helps an architect manage complexity.

It is also worth noting that the set of significant elements is not static and may change over time. As a consequence of requirements being refined, risks identified, executable software built, and lessons learned, the set of significant elements may change. However, the relative stability of the architecture in the face of change is, to some extent, the sign of a good architecture, the sign of a well-executed architecting process, and the sign of a good architect. If the architecture needs to be continually revised due to relatively minor changes, then this is not a good sign. However, if the architecture is relatively

stable, then the converse is true.

An architecture balances stakeholder needs

An architecture is created to ultimately address a set of stakeholder needs. However, it is often not possible to meet all of the needs expressed. For example, a stakeholder may ask for some functionality within a specified timeframe, but these two needs (functionality and timeframe) are mutually exclusive. Either the scope can be reduced in order to meet the schedule or all of the functionality can be provided within an extended timeframe. Similarly, different stakeholders may express conflicting needs and, again, an appropriate balance must be achieved. Making tradeoffs is therefore an essential aspect of the architecting process, and negotiation, an essential characteristic of the architect.

Just to give you an idea of the task at hand, consider the following needs of a set of stakeholders:

- The end user is concerned with intuitive and correct behavior, performance, reliability, usability, availability, and security.
- The system administrator is concerned with intuitive behavior, administration, and tools to aid monitoring.
- The marketer is concerned with competitive features, time to market, positioning with other products, and cost.
- The customer is concerned with cost, stability, and schedule.
- The developer is concerned with clear requirements, and a simple and consistent design approach.
- The project manager is concerned with predictability in the tracking of the project, schedule, productive use of resources, and budget.
- The maintainer is concerned with a comprehensible, consistent, and documented design approach, and the ease with which modifications can be made.

As we can see from this list, another challenge for the architect is that the stakeholders are not only concerned that the system provides the required functionality. Many of the concerns listed are nonfunctional in nature in that they do not contribute to the functionality of the system (e.g., the concerns regarding costs and scheduling). Such concerns nevertheless represent system qualities or constraints. Nonfunctional requirements are quite often the most significant requirements as far as an architect is concerned.

An architecture embodies decisions based on rationale

An important aspect of an architecture is not just the end result, the architecture itself, but the rationale for why it is the way it is. Thus, an important consideration is to ensure that you document the decisions that have led to this architecture and the rationale for those decisions.

This information is relevant to many stakeholders, especially those who must maintain the system. This information is often valuable to the architect when he or she needs to revisit the rationale behind the decisions that were made, so that they don't end up having to unnecessarily retrace steps. For example, this information is used when the architecture is reviewed and the architect needs to justify the decisions that have been made.

An architecture may conform to an architectural style

Most architectures are derived from systems that share a similar set of concerns. This similarity can be described as an architectural style, which can be thought of as a particular kind of pattern, albeit an often complex and composite pattern (a number of patterns applied together). Like a pattern, an architectural style represents a codification of experience, and it is good practice for architects to look for opportunities to reuse such experience. Examples of architectural styles include a distributed style, a pipe-and-filter style, a data-centered style, a rule-based style, and so on. A given system may exhibit more than one architectural style. As Shaw and Garlan describe it:

> *[An architectural style] defines a family of systems in terms of a pattern of structural organization. More specifically, an architectural style defines a vocabulary of components and connector types, and a set of constraints on how they can be combined.*[9]

And in terms of the UML:

> *[A pattern is] a common solution to a common problem in a given context.*[10]

In addition to reusing experience, the application of an architectural style (or a pattern) makes our lives as architects somewhat easier, since a style is normally documented in terms of the rationale for using it (and so there is less thinking to be done) and in terms of its structure and behavior (and so there is less architecture documentation to be produced since we can simply refer to the style instead).

An architecture is influenced by its environment

A system resides in an environment, and this environment influences the architecture. This is sometimes referred to as "architecture in context." In essence, the environment determines the boundaries within which the system must operate, which then influence the architecture. The environmental factors that influence the architecture include the business mission that the architecture will support, the system stakeholders, internal technical constraints (such as the requirement to conform to organizational standards), and external technical constraints (such as the need to interface to an external system or to conform to external regulatory standards).

Conversely, as eloquently described in Bass, Clements, and Kazman,[11] the architecture may also influence its environment. Not only does the creation of an architecture change the environment from a technology perspective -- it may, for example, contribute reusable assets to the owning organization -- the creation of the architecture may also change the environment in terms of the skills available within the organization.

When it comes to software-intensive systems, there is a particular aspect of the environment that must always be considered, as discussed earlier in this chapter. In order for software to be useful, it must execute. In order to execute, the software runs on some kind of hardware. The resulting system is therefore a combination of both software and hardware, and it is this combination that allows properties such as reliability and performance to be achieved. Software cannot achieve these properties in isolation of the hardware on which it executes.

IEEE Std 12207-1995, the IEEE Standard for Information Technology -- Software Life Cycle Processes, defines a system differently from the IEEE 1471 system definition noted earlier (which focuses on software-intensive systems), but is in agreement with the definitions found in the systems engineering field:

> *[A system is] an integrated composite that consists of one or more of the processes, hardware, software, facilities and people, that provides a capability to satisfy a stated need or objective. [IEEE 12207]*[12]

A configuration of the Rational Unified Process for Systems Engineering (RUP SE) contains a similar definition.

> *[A system is] a set of resources that provide services that are used by an enterprise to carry out a business purpose or mission. System components typically consist of hardware, software, data, and workers*[13].

In the systems engineering field, tradeoffs are made regarding the use of software, hardware, and people. For example, if performance is key, then a decision may be made to implement certain system elements in hardware, rather than software or people. Another example is that in order to provide a usable system to customers, a decision is made to provide a customer interface that is a human being, rather than an interface implemented in software or hardware. More complex scenarios require certain system qualities to be achieved through a combination of software, hardware, and people. (Accordingly, this series of articles makes reference to elements other than software where appropriate.)

It is interesting to note that systems engineering is specifically concerned with treating software and hardware (as well as people) as peers, thus avoiding the pitfall where hardware is treated as a second-class citizen to the software and is simply a vehicle for executing the software, or where software is treated as a second-class citizen with respect to the hardware and is simply a vehicle for making the hardware function as desired.

An architecture influences team structure

An architecture defines coherent groupings of related elements that address a given set of concerns. For example, an architecture for an order processing system may have defined groupings of elements for order entry, account management, customer management, fulfillment, integrations with external systems, persistency, and security.

Each of these groupings may require different skill sets. It therefore makes perfect sense to align software development team structures with the architecture once it has been defined. However, it is often the case that the architecture is influenced by the initial team structure and not vice versa. This is a pitfall that is best avoided, since the result is typically a less-than-ideal architecture. "Conway's Law" states that "If you have four groups working on a compiler, you'll get a 4-pass compiler." In practice, we often unintentionally create architectures that reflect the organization creating the architecture.

However, it is also true to say that this somewhat idealized view is not always practical since, for purely pragmatic reasons, the current team structure and the skills available represent a very real constraint on what is possible and the architect must take this into account.

An architecture is present in every system

It is also worth noting that every system has an architecture, even if this architecture is not formally documented or if the system is extremely simple and, say, consists of a single element. There is usually considerable value in documenting the architecture. Documented architectures tend to be more carefully considered -- and therefore, more effective -- than those that are not, since the process of recording the architecture naturally leads to thoughtful consideration.

Conversely, if an architecture is not documented, then it is difficult (if not impossible) to prove that it meets the stated requirements in terms of addressing qualities such as maintainability, accommodation of best practices, and so on. Architectures that are not documented, which appear to be the majority in existence today, tend to be accidental rather than intentional.

An architecture has a particular scope

There are many kinds of architecture, the best known being the architecture associated with buildings and other civil engineering structures. Even in the field of software engineering, we often come across different forms of architecture. For example, in addition to the concept of *software architecture*, we may encounter concepts such as *enterprise architecture, system architecture, organizational architecture, information architecture, hardware architecture, application architecture, infrastructure architecture*, and so on. You will also hear other terms, each of which defines a specific scope of the architecting activities.

Unfortunately, there is no agreement in the industry on the meaning of each of these terms or their

relationship to one another, which results in different meanings for the same term (homonyms) and two or more terms meaning the same thing (synonyms). However, the scope of some of these terms can be inferred from Figure 3. As you consider this figure and the discussion that follows, there are almost certainly elements of it that you disagree with or that you use differently within your organization. But that is exactly the point -- to show that these terms do exist in the industry, but that there is no consensus on their meaning.

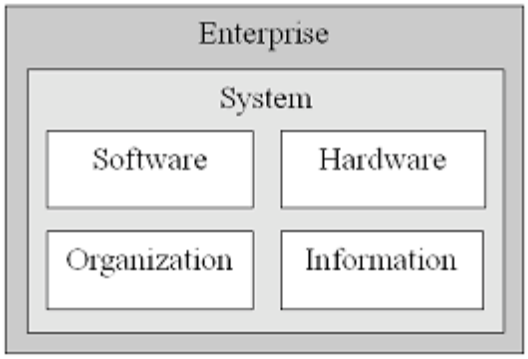

Figure 3: The scope of different terms

The elements shown in Figure 3 are:

- Software architecture, which is the main focus of this article as defined earlier.
- A hardware architecture, which considers elements such as CPUs, memory, hard disks, peripheral devices such as printers, and the elements used to connect these elements.
- An organizational architecture, which considers elements that are concerned with business processes, organizational structures, roles and responsibilities, and core competencies of the organization.
- An information architecture, which considers the structure by which information is organized.
- Software architecture, hardware architecture, organizational architecture, and information architecture, which are all subsets of the overall system architecture, as discussed earlier in this chapter.
- An enterprise architecture, which is similar to a system architecture in that it, too, considers elements such as hardware, software, and people. However, an enterprise architecture has a stronger link to the business in that it focuses on the attainment of the business objectives and is concerned with items such as business agility and organizational efficiency. An enterprise architecture may cross company boundaries.

As one would expect, there are corresponding forms of architect (for example, software architect, hardware architect, and so on) and architecting (for example, software architecting, hardware architecting, and so on).

Now that we've gotten through these definitions, there are many unanswered questions. What is the difference between an enterprise architecture and a system architecture? Is an enterprise a system? Is an information architecture the same as the data architecture found in some data-intensive software applications? Unfortunately, there is no set of agreed-upon answers to these questions.

For now, you should simply be aware that these different terms exist, but that there is no consistent definition of these terms in the industry and how they relate. The recommendation, therefore, is for you to select the terms relevant to your organization and define them appropriately. You will then achieve some consistency at least and reduce the potential for miscommunication.

Summary

This article has focused on defining the core characteristics of a software architecture. However, there are

many questions left unanswered. What is the role of the software architect? What are the key activities that the architect is involved in? What are the benefits of "architecting"? These questions, and others, will be answered in subsequent articles in this series.

Acknowledgements

The contents of this article have been derived from a forthcoming book, provisionally entitled "The Process of Software Architecting." As a result, the content has been commented upon by many individuals that I would like to thank, who are Grady Booch, Dave Braines, Alan Brown, Mark Dickson, Holger Heuss, Kelli Houston, Luan Doan-Minh, Philippe Kruchten, Nick Rozanski, Dave Williams, and Eoin Woods.

Notes

[1] The Software Engineering Institute (SEI) Architecture Website -- architecture definitions, offers a good example. See http://www.sei.cmu.edu/architecture/definitions.html

[2] IEEE Computer Society, IEEE Recommended Practice for Architectural Description of Software-Intensive Systems: IEEE Std 1472000. 2000.

[3] Object Management Group Inc., UML 2.0 Infrastructure Specification: Document number 03-09-15. September 2003.

[4] Philippe Kruchten, *The Rational Unified Process: An Introduction*, Third Edition. Addison-Wesley Professional 2003.

[5] Len Bass, Paul Clements, and Rick Kazman, *Software Architecture in Practice*, Second Edition. Addison Wesley 2003.

[6] Object Management Group Inc., *OMG Unified Modeling Language Specification Version 1.5*, Document number 03-03-01. March 2003.

[7] James McGovern, et al., *A Practical Guide to Enterprise Architecture*. Prentice Hall 2004

[8] A role that will be covered in a subsequent article in this series.

[9] Mary Shaw and David Garlan, *Software Architecture -- Perspectives on an Emerging Discipline*. Prentice Hall 1996.

[10] Grady Booch, James Rumbaugh, and Ivar Jacobson, *The Unified Modeling Language User Guide*. Addison Wesley 1999.

[11] Bass et al., Op. cit.

[12] IEEE Computer Society, IEEE Standard for Information Technology -- Software Life Cycle Processes. IEEE Std 12207-1995.

[13] Murray Cantor, "Rational Unified Process for Systems Engineering." *The Rational Edge*, August 2003. http://download.boulder.ibm.com/ibmdl/pub/software/dw/rationaledge/aug03/f_rupse_mc.pdf

About the author

Peter Eeles works for IBM Rational, IBM Software Group, and has spent much of his career architecting and implementing large-scale, distributed systems. Based in the UK, he assists organizations in their adoption of the Rational Unified Process and the IBM Software Development Platform. He is coauthor of "Building J2EE Applications with the Rational Unified Process" (Addison-Wesley, 2002), coauthor of "Building Business Objects" (John Wiley and Sons, 1998), and a contributing author to "Software Architectures" (Springer-Verlag, 1999).