Ian Gorton

Essential Software Architecture

Second Edition

Springer

Essential Software Architecture

Ian Gorton

Essential Software Architecture

Second Edition

Springer

Ian Gorton
Laboratory Fellow
Pacific Northwest National Laboratory
PO Box 999
MSIN: K7-90
Richland, WA 99352
USA
ian.gorton@pnl.gov

ACM Computing Classification (1998): D.2

ISBN 978-3-642-19175-6 e-ISBN 978-3-642-19176-3
DOI 10.1007/978-3-642-19176-3
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011926871

Cover design: KuenkelLopka GmbH

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Welcome to the second edition of Essential Software Architecture. It is 5 years since the first edition was published, and in the software architecture world, 5 years is a long time. Hence this updated version, with refreshed chapters to capture new developments in methods and technologies, and to relate relevant experiences from practise. There's new material covering enterprise architecture, agile development, enterprise service bus technologies and RESTful Web services. All chapters have an updated and more extensive list of recommended reading, capturing many of the best new books, papers, web sites and blogs that I know of.

Most notably, the completely new Chap. 10 provides a case study on the design of the MeDICi technology, which extends an open source enterprise service bus with a component-based programming model. The MeDICi technology is open source and freely downloadable (http://www.medici.pnl.gov), making it a highly suitable tool for teaching the advanced concepts of middleware and architecture described in this text.

At its heart however, this remains a book that aims to succinctly impart a broad sweep of software architecture knowledge relating to systems built from mainstream middleware technologies. This includes a large, diverse spectrum of systems, ranging from Web-based ecommerce sites to scientific data management and high performance financial data analysis systems.

Motivation

What hasn't changed in the last 5 years is that many projects I work with or review lack an explicit notion of an architectural design. Functional requirements are usually captured using traditional or agile techniques, agreed with stakeholders, and addressed through highly iterative or traditional waterfall methods. But the architectural issues, the "how" the application achieves its purpose, the "what" happens when things change and evolve or fail, are frequently implicit (this means they are in somebody's head, maybe) at best. At worst, they are simply not addressed in any way that can be described in terms other than accidental. Frequently, when I ask for an overview of the application architecture and the driving nonfunctional requirements at the first technical meeting, people start drawing on a whiteboard. Or they show me code and dive into the iternals of the implementation based around their favorite, trendy technology. Either of these is rarely a good sign.

The problems and risks of poor architectural practices are well known and documented within the software engineering profession. A large body of excellent architectural knowledge is captured in broadly accessible books, journals and reports from members of the Software Engineering Institute (SEI), Siemens and various other renowned industrial and academic institutions.

While the focus of much of this literature is highly technical systems such as avionics, flight simulation, and telecommunications switching, this book leans more to the mainstream world of software applications. In a sense, it bridges the gap between the needs of the vast majority of software professionals and the current body of knowledge in software architecture. Specifically:

- It provides clear and concise discussions about the issues, techniques and methods that are at the heart of sound architectural practices.
- It describes and analyzes the general purpose component and middleware technologies that support many of the fundamental architectural patterns used in applications.
- It looks forward to how changes in technologies and practices may affect the next generation of business information systems.
- It uses familiar information systems as examples, taken from the author's experiences in banking, e-commerce and government information systems.
- It provides many pointers and references to existing work on software architecture.

If you work as an architect or senior designer, or you want to 1 day, this book should be of value to you. And if you're a student who is studying software engineering and need an overview of the field of software architecture, this book should be an approachable and useful first source of information. It certainly won't tell you everything you need to know – that will take a lot more than can be included in a book of such modest length. But it aims to convey the essence of architectural thinking, practices and supporting technologies, and to position the reader to delve more deeply into areas that are pertinent to their professional life and interests.

Outline

The book is structured into three basic sections. The first is introductory in nature, and approachable by a relatively nontechnical reader wanting an overview of software architecture.

The second section is the most technical in nature. It describes the essential skills and technical knowledge that an IT architect needs.

The third is forward looking. Six chapters each introduce an emerging area of software practice or technology. These are suitable for existing architects and designers, as well as people who've read the first two sections, and who wish to gain insights into the future influences on their profession.

More specifically:

- *Chapters 1–3*: These chapters provide the introductory material for the rest of the book, and the area of software architecture itself. Chapter 1 discusses the key elements of software architecture, and describes the roles of a software architect. Chapter 2 introduces the requirements for a case study problem, a design for which is presented in Chap. 9. This demonstrates the type of problem and associated description that a software architect typically works on. Chapter 3 analyzes the elements of some key quality attributes like scalability, performance and availability. Architects spend a lot of time addressing the quality attribute requirements for applications. It's therefore essential that these quality attributes are well understood, as they are fundamental elements of the knowledge of an architect.
- *Chapters 4–10*: These chapters are the technical backbone of the book. Chapter 4 introduces a range of fundamental middleware technologies that architects commonly leverage in application solutions. Chapter 5 is devoted to describing Web services, including both SOAP and REST-based approaches. Chapter 6 builds on the previous chapters to explain advanced middleware platforms such as enterprise service bus technologies. Chapter 7 presents a three stage iterative software architecture process that can be tailored to be as agile as a project requires. It describes the essential tasks and documents that involve an architect. Chapter 8 discusses architecture documentation, and focuses on the new notations available in the UML version 2.0. Chapter 9 brings together the information in the first 6 chapters, showing how middleware technologies can be used to address the quality attribute requirements for the case study. It also demonstrates the use of the documentation template described in Chap. 8 for describing an application architecture. Chapter 10 provides another practical case study describing the design of the open source MeDICi Integration Framework, which is a specialized API for building applications structured as pipelines of components.
- *Chapters 11–15*: These chapters each focus on an emerging technique or technology that will likely influence the futures of software architects. These include software product lines, model-driven architecture, aspect-oriented architecture and the Semantic Web. Each chapter introduces the essential elements of the method or technology, describes the state-of-the-art and speculates about how increasing adoption is likely to affect the required skills and practices of a software architect. Each chapter also relates its approach to an extension of the ICDE case study in Chap. 9.

Richland, WA, USA Ian Gorton
December 2010

Acknowledgments

First, thanks to the chapter contributors who have helped provide the content on software product lines (Mark Staples), aspect-oriented programming (Jenny Liu), model-driven development (Liming Zhu), Web services (Paul Greenfield) and the Semantic Web (Judi Thomson). Adam Wynne also coauthored the chapter on MeDICi. Your collective efforts and patience are greatly appreciated.

Contact details for the contributing authors are as follows:

Dr Mark Staples, National ICT Australia, email: mark.staples@nicta.com.au

Dr Liming Zhu, National ICT Australia, email: liming.zhu@nicta.com.au

Dr Yan Liu, Pacific Northwest National Lab, USA, email: jenny.liu@nicta.com.au

Adam Wynne, Pacific Northwest National Lab, USA, email: adam.wynne@pnl.gov

Paul Greenfield, School of IT, CSIRO, Australia, email: paul.greenfield@csiro.au

Dr Judi McCuaig, University of Guelph, Canada, email: judi@cis.uguelph.ca

I'd also like to thank everyone at Springer who has helped make this book a reality, especially the editor, Ralf Gerstner.

I'd also like to acknowledge the many talented software architects, engineers and researchers who I've worked closely with recently and/or who have helped shape my thinking and experience through long and entertaining geeky discussions. In no particular order these are Anna Liu, Paul Greenfield, Shiping Chen, Paul Brebner, Jenny Liu, John Colton, Karen Schchardt, Gary Black, Dave Thurman, Jereme Haack, Sven Overhage, John Grundy, Muhammad Ali Babar, Justin Almquist, Rik Littlefield, Kevin Dorow, Steffen Becker, Ranata Johnson, Len Bass, Lei Hu, Jim Thomas, Deb Gracio, Nihar Trivedi, Paula Cowley, Jim Webber, Adrienne Andrew, Dan Adams, Dean Kuo, John Hoskins, Shuping Ran, Doug Palmer, Nick Cramer, Liming Zhu, Ralf Reussner, Mark Hoza, Shijian Lu, Andrew Cowell, Tariq Al Naeem, Wendy Cowley and Alan Fekete.

Contents

Chapter 1
Understanding Software Architecture

1.1 What is Software Architecture?

The last 15 years have seen a tremendous rise in the prominence of a software engineering subdiscipline known as software architecture. *Technical Architect* and *Chief Architect* are job titles that now abound in the software industry. There's an International Association of Software Architects,[1] and even a certain well-known wealthiest geek on earth used to have "architect" in his job title in his prime. It can't be a bad gig, then?

I have a sneaking suspicion that "architecture" is one of the most overused and least understood terms in professional software development circles. I hear it regularly misused in such diverse forums as project reviews and discussions, academic paper presentations at conferences and product pitches. You know a term is gradually becoming vacuous when it becomes part of the vernacular of the software industry sales force.

This book is about software architecture. In particular it's about the key design and technology issues to consider when building server-side systems that process multiple, simultaneous requests from users and/or other software systems. Its aim is to concisely describe the essential elements of knowledge and key skills that are required to be a software architect in the software and information technology (IT) industry. Conciseness is a key objective. For this reason, by no means everything an architect needs to know will be covered. If you want or need to know more, each chapter will point you to additional worthy and useful resources that can lead to far greater illumination.

So, without further ado, let's try and figure out what, at least I think, software architecture really is, and importantly, isn't. The remainder of this chapter will address this question, as well as briefly introducing the major tasks of an architect, and the relationship between architecture and technology in IT applications.

[1] http://www.iasahome.org/web/home/home

I. Gorton, *Essential Software Architecture*,
DOI 10.1007/978-3-642-19176-3_1, © Springer-Verlag Berlin Heidelberg 2011

1.2 Definitions of Software Architecture

Trying to define a term such as software architecture is always a potentially dangerous activity. There really is no widely accepted definition by the industry. To understand the diversity in views, have a browse through the list maintained by the Software Engineering Institute.[2] There's a lot. Reading these reminds me of an anonymous quote I heard on a satirical radio program recently, which went something along the lines of "the reason academic debate is so vigorous is that there is so little at stake".

I've no intention of adding to this debate. Instead, let's examine three definitions. As an IEEE member, I of course naturally start with the definition adopted by my professional body:

> Architecture is defined by the recommended practice as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.
>
> [ANSI/IEEE Std 1471-2000, *Recommended Practice for Architectural Description of Software-Intensive Systems*]

This lays the foundations for an understanding of the discipline. Architecture captures system structure in terms of components and how they interact. It also defines system-wide design rules and considers how a system may change.

Next, it's always worth getting the latest perspective from some of the leading thinkers in the field.

> 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.
>
> [L.Bass, P.Clements, R.Kazman, *Software Architecture in Practice (2nd edition),* Addison-Wesley 2003]

This builds somewhat on the above ANSI/IEEE definition, especially as it makes the role of abstraction (i.e., externally visible properties) in an architecture and multiple architecture views (structures of the system) explicit. Compare this with another, from Garlan and Shaw's early influential work:

> [Software architecture goes] beyond the algorithms and data structures of the computation; designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives.
>
> [D. Garlan, M. Shaw, *An Introduction to Software Architecture,* Advances in Software Engineering and Knowledge Engineering, Volume I, World Scientific, 1993]

It's interesting to look at these, as there is much commonality. I include the third mainly as it's again explicit about certain issues, such as scalability and

---

[2] http://www.sei.cmu.edu/architecture/definitions.html

distribution, which are implicit in the first two. Regardless, analyzing these a little makes it possible to draw out some of the fundamental characteristics of software architectures. These, along with some key approaches, are described below.

1.2.1 Architecture Defines Structure

Much of an architect's time is concerned with how to sensibly partition an application into a set of interrelated components, modules, objects or whatever unit of software partitioning works for you.[3] Different application requirements and constraints will define the precise meaning of "sensibly" in the previous sentence – an architecture must be designed to meet the specific requirements and constraints of the application it is intended for.

For example, a requirement for an information management system may be that the application is distributed across multiple sites, and a constraint is that certain functionality and data must reside at each site. Or, an application's functionality must be accessible from a web browser. All these impose some structural constraints (site-specific, web server hosted), and simultaneously open up avenues for considerable design creativity in partitioning functionality across a collection of related components.

In partitioning an application, the architect assigns responsibilities to each constituent component. These responsibilities define the tasks a component can be relied upon to perform within the application. In this manner, each component plays a specific role in the application, and the overall component ensemble that comprises the architecture collaborates to provide the required functionality.

Responsibility-driven design (see *Wirfs-Brock* in Further Reading) is a technique from object-orientation that can be used effectively to help define the key components in an architecture. It provides a method based on informal tools and techniques that emphasize behavioral modeling using objects, responsibilities and collaborations. I've found this extremely helpful in past projects for structuring components at an architectural level.

A key structural issue for nearly all applications is minimizing dependencies between components, creating a loosely coupled architecture from a set of highly cohesive components. A dependency exists between components when a change in one potentially forces a change in others. By eliminating unnecessary dependencies, changes are localized and do not propagate throughout an architecture (see Fig. 1.1).

[3] *Component* here and in the remainder of this book is used very loosely to mean a recognizable "chunk" of software, and not in the sense of the more strict definition in *Szyperski C. (1998) Component Software: Beyond Object-Oriented Programming, Addison-Wesley*

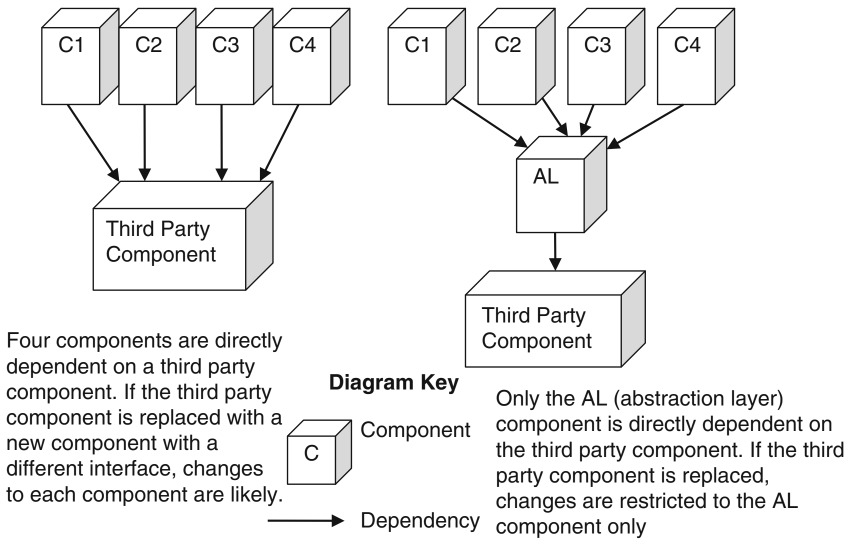

Fig. 1.1 Two examples of component dependencies

Excessive dependencies are simply a bad thing. They make it difficult to make changes to systems, more expensive to test changes, they increase build times, and they make concurrent, team-based development harder.

1.2.2 Architecture Specifies Component Communication

When an application is divided into a set of components, it becomes necessary to think about how these components communicate data and control information. The components in an application may exist in the same address space, and communicate via straightforward method calls. They may execute in different threads or processes, and communicate through synchronization mechanisms. Or multiple components may need to be simultaneously informed when an event occurs in the application's environment. There are many possibilities.

A body of work known collectively as architectural patterns or styles[4] has catalogued a number of successfully used structures that facilitate certain kinds of component communication [see *Patterns* in Further Reading]. These patterns are essentially reusable architectural blueprints that describe the structure and interaction between collections of participating components.

Each pattern has well-known characteristics that make it appropriate to use in satisfying particular types of requirements. For example, the client–server pattern

[4]Patterns and styles are essentially the same thing, but as a leading software architecture author told me recently, "the patterns people won". This book will therefore use patterns instead of styles!

has several useful characteristics, such as synchronous request–reply communications from client to server, and servers supporting one or more clients through a published interface. Optionally, clients may establish sessions with servers, which may maintain state about their connected clients. Client–server architectures must also provide a mechanism for clients to locate servers, handle errors, and optionally provide security on server access. All these issues are addressed in the client–server architecture pattern.

The power of architecture patterns stems from their utility, and ability to convey design information. Patterns are proven to work. If used appropriately in an architecture, you leverage existing design knowledge by using patterns.

Large systems tend to use multiple patterns, combined in ways that satisfy the architecture requirements. When an architecture is based around patterns, it also becomes easy for team members to understand a design, as the pattern infers component structure, communications and abstract mechanisms that must be provided. When someone tells me their system is based on a three-tier client–server architecture, I know immediately a considerable amount about their design. This is a very powerful communication mechanism indeed.

1.3 Architecture Addresses Nonfunctional Requirements

Nonfunctional requirements are the ones that don't appear in use cases. Rather than define *what* the application does, they are concerned with *how* the application provides the required functionality.

There are three distinct areas of nonfunctional requirements:

- *Technical constraints*: These will be familiar to everyone. They constrain design options by specifying certain technologies that the application must use. "We only have Java developers, so we must develop in Java". "The existing database runs on Windows XP only". These are usually nonnegotiable.
- *Business constraints*: These too constraint design options, but for business, not technical reasons. For example, "In order to widen our potential customer base, we must interface with XYZ tools". Another example is "The supplier of our middleware has raised prices prohibitively, so we're moving to an open source version". Most of the time, these too are nonnegotiable.
- *Quality attributes*: These define an application's requirements in terms of scalability, availability, ease of change, portability, usability, performance, and so on. Quality attributes address issues of concern to application users, as well as other stakeholders like the project team itself or the project sponsor. Chapter 3 discusses quality attributes in some detail.

An application architecture must therefore explicitly address these aspects of the design. Architects need to understand the functional requirements, and create a platform that supports these and simultaneously satisfies the nonfunctional requirements.

1.3.1 Architecture Is an Abstraction

One of the most useful, but often nonexistent, descriptions from an architectural perspective is something that is colloquially known as a *marketecture*. This is one page, typically informal depiction of the system's structure and interactions. It shows the major components and their relationships and has a few well-chosen labels and text boxes that portray the design philosophies embodied in the architecture. A *marketecture* is an excellent vehicle for facilitating discussion by stakeholders during design, build, review, and of course the sales process. It's easy to understand and explain and serves as a starting point for deeper analysis.

A thoughtfully crafted *marketecture* is particularly useful because it is an abstract description of the system. In reality, any architectural description must employ abstraction in order to be understandable by the team members and project stakeholders. This means that unnecessary details are suppressed or ignored in order to focus attention and analysis on the salient architectural issues. This is typically done by describing the components in the architecture as black boxes, specifying only their *externally visible properties*. Of course, describing system structure and behavior as collections of communicating black box abstractions is normal for practitioners who use object-oriented design techniques.

One of the most powerful mechanisms for describing an architecture is hierarchical decomposition. Components that appear in one level of description are decomposed in more detail in accompanying design documentation. As an example, Fig. 1.2 depicts a very simple two-level hierarchy using an informal notation, with two of the components in the top-level diagram decomposed further.

Different levels of description in the hierarchy tend to be of interest to different developers in a project. In Fig. 1.2, it's likely that the three components in the top-level description will be designed and built by different teams working on the

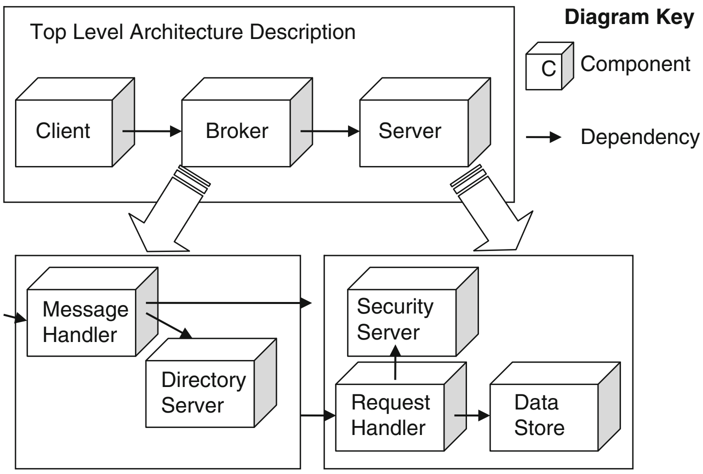

Fig. 1.2 Describing an architecture hierarchically

application. The architecture clearly partitions the responsibilities of each team, defining the dependencies between them.

In this hypothetical example, the architect has refined the design of two of the components, presumably because some nonfunctional requirements dictate that further definition is necessary. Perhaps an existing security service must be used, or the *Broker* must provide a specific message routing function requiring a directory service that has a known level of request throughput. Regardless, this further refinement creates a structure that defines and constrains the detailed design of these components.

The simple architecture in Fig. 1.2 doesn't decompose the *Client* component. This is, again presumably, because the internal structure and behavior of the client is not significant in achieving the application's overall nonfunctional requirements. How the *Client* gets the information that is sent to the *Broker* is not an issue that concerns the architect, and consequently the detailed design is left open to the component's development team. Of course, the *Client* component could possibly be the most complex in the application. It might have an internal architecture defined by its design team, which meets specific quality goals for the *Client* component. These are, however, localized concerns. It's not necessary for the architect to complicate the application architecture with such issues, as they can be safely left to the *Client* design team to resolve. This is an example of suppressing unnecessary details in the architecture.

1.3.2 Architecture Views

A software architecture represents a complex design artifact. Not surprisingly then, like most complex artifacts, there are a number of ways of looking at and understanding an architecture. The term "architecture views" rose to prominence in Philippe Krutchen's 1995[5] paper on the *4+1 View Model*. This presented a way of describing and understanding an architecture based on the following four views:

- *Logical view*: This describes the architecturally significant elements of the architecture and the relationships between them. The logical view essentially captures the structure of the application using class diagrams or equivalents.
- *Process view*: This focuses on describing the concurrency and communications elements of an architecture. In IT applications, the main concerns are describing multithreaded or replicated components, and the synchronous or asynchronous communication mechanisms used.
- *Physical view*: This depicts how the major processes and components are mapped on to the applications hardware. It might show, for example, how the database and web servers for an application are distributed across a number of server machines.

[5]P.Krutchen, *Architectural Blueprints–The "4+1" View Model of Software Architecture*, IEEE Software, 12(6) Nov. 1995.