Essential Software Architecture

Ian Gorton

Essential Software Architecture

With 93 Figures and 11 Tables

Springer

Author

Ian Gorton

National ICT Australia
Bay 15, Locomotive Workshop
Australian Technology Park, Garden St
Eveleigh NSW 1430, Australia
Ian.gorton@nicta.com.au

Library of Congress Control Number: 2006921741

ACM Computing Classification (1998): D.2

ISBN-10 3-540-28713-2 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-28713-1 Springer Berlin Heidelberg New York

Typeset by the author
Production: LE-TEX Jelonek, Schmidt & Vöckler GbR, Leipzig
Cover design: KünkelLopka Werbeagentur, Heidelberg

Printed on acid-free paper 45/3100/YL - 5 4 3 2 1 0

Foreword

Architecture is something of a black art in the IT world. Architects learn on the job, bringing years of experience in design and technology to the business problems they tackle. It's not an easy task to impart architecture knowledge.

So when Ian first spoke to me about the idea of writing this book, I thought "Great! Finally there will be a book that I can recommend to the many developers and students who approach me asking what they have to do to become an IT architect". I knew that from reading the book they would discover many of the essential ingredients of being a good practicing IT architect.

In the years that I have known Ian, he has been an inspirational educator, a pragmatic and decisive software architect, and an idealistic software architecture researcher. On top of all that, he is an excellent communicator, who articulates advanced computing concepts clearly and succinctly irrespective of his audience – the novice or the experienced. Ian is also full of great life stories to tell – all told with a great sense of humor (especially after a few glasses of good wine!).

It is not until Ian handed me drafts of the various chapters that I realized that this is a *must have* book for the experienced IT architects too. As consulting enterprise architects, we are usually working inside an enterprise's boundary, trying to influence the IT directions of the various departments within the enterprise, and designing the next evolution of IT architecture that breaks down the silos within the organization. We are often lone spirits, making important technology acquisition and design decisions without having a reference to look upon for validation of what we are doing. Now, for practicing architects, Ian's book serves this precise purpose – it brings a sense of relief knowing that we are not alone, and that there are many others who also face similar architecture challenges. Ian's book, although by no means a silver bullet to all of our IT architecture challenges, certainly helps us head in the right direction through the various techniques and approaches presented.

So here it is, an essential guide to computer science students as well as practicing developers and IT professionals who aspire to become an IT architect. For the experienced architects, it serves as a reference, a good validation of our thinking, and provides a summary of emerging technologies and practices that will be important in the not-too-distant future.

I hope you will enjoy the read as much as I have.

Dr. Anna Liu
Architect Advisor, Microsoft
Sydney, Australia

Preface

Motivation

In the not too distant past, on a decent sized project I was working on, I convinced a senior developer, a highly skilled and experienced software engineer, to purchase a copy of "Documenting Software Architectures: Views and Beyond". The project was a little sparse in terms of documentation and process, the team was well aware of this, and I was trying to help improve the situation. Soon after the book arrived, a brief corridor conversation saw strong expressions of enthusiasm. So strong, an additional project team meeting was called for the next week.

In the meeting, the senior engineer held up this "wonderful" book, and espoused many of its key messages to the team. I was, of course, pleased that the book I'd recommended had made such an impact. Then the follow line caught my attention:

"I read the first 30 or so pages, which were great, but only skimmed the rest."

I was a little surprised at this statement. Why was the content of this very informative and incredibly useful book mostly "skimmed"? Surely anyone could learn much from investing a little time in reading the more technical chapters? I certainly did. This set me pondering the root cause of this issue, as I really wanted to instill more architectural knowledge in the development teams I was working with.

In my career as a *roving* software architect, I've spent a lot of time consulting on projects, providing architectural design skills and knowledge. These projects have spanned many organizations and different application domains over the last decade.

A common theme though, is that I work mostly in what would be considered general information technology (IT) application domains. The sort of applications that financial institutions, utilities and Government agencies build to manage and deliver information to their customers and trading partners. These are, broadly, *business information systems* that leverage

Commercial-off-the-shelf (COTS) technologies like databases, middleware, packaged applications and web technologies.

I occasionally work on projects in what are considered *more technical* domains, such as military, embedded control and telecommunications applications. I can do this because many of the underlying architectural issues are the same across domains. However, the way these issues manifest themselves, the particular technology solutions that are commonly adopted and the technical vocabularies used are radically different. Hence I specialize in IT systems – this is where I can hopefully add more value.

Sometimes my role involves designing new application architectures, or actually more frequently evaluating existing ones and helping evolve them. In the process, I work closely with the members of projects teams. This is enjoyable. I always learn from them, and I hope they sometimes learn from me.

A strikingly common characteristic of most of these projects is a lack of explicit architectural design. Functional requirements are usually captured, agreed with stakeholders and managed, and designs that address the functional specifications are fleshed out in detail. 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) 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 non-functional requirements at the first technical meeting, people start drawing on a whiteboard. Or they show me code. 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.

So, I pondered further, why is this information on best practices and tools not permeating through the IT industry? In response, I can only posit the following.

In general, the many sources of software architecture information are extremely thorough, learned and lengthy, requiring a serious investment of time to fully digest. The SEI books, for example, are based upon many years of experience working in mostly in military applications. These typically comprise large embedded, real-time software systems, with a set of architectural approaches and issues that have a particular emphasis to this application domain. For example, many of the case studies are about avi-

onics, flight simulation, and engine control applications, and present solutions, such as fixed priority scheduling and process distribution, to the problems that are encountered in such systems.

I suspect this emphasis on military and embedded domains makes these materials a difficult read for IT software professionals who are unfamiliar with the problems and solutions described. The vocabularies used tend towards those that are prevalent in academic circles – I still have not heard many IT architects discuss architectural styles, connectors or the merits of formal architecture description languages. They do though discuss architecture and design patterns, middleware and use UML and informal techniques to model aspects of their architectures.

Further, in the software architecture literature, there is little discussion of the types of off-the-shelf technologies that are commonly used to address architectural problems in business information systems. Fixed priority schedulers and embedded operating systems are mostly irrelevant in information systems. Application servers, component technologies and messaging infrastructures are the basic building blocks that are important to an IT architect. These are the foundations of the architectures of modern information systems. It is therefore essential that architects understand how these technologies can be leveraged to effectively provide the architectural mechanisms required by a given application.

This book, then, is an attempt to bridge the gap between the needs of IT professionals and the current body of knowledge in software architecture.

- It attempts to provide 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 also provides pointers and references to existing work on software architecture.

If you work as an architect or senior designer, or you want to one 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 ar-

chitecture, 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 non-technical 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 Chapter 7.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 4–7:** These chapters are the technical backbone of the book. Chapter 4 introduces a range of middleware technologies that architects commonly leverage in application solutions. Chapter 5 presents a three stage iterative software architecture process. It describes the essential tasks and documents that involve an architect. Chapter 6 discusses architecture documentation, and focuses on the new notations available in the UML version 2.0. Chapter 7 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 Chapter 6 for documenting an application architecture.

- **Chapters 8–14:** 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, service-oriented architectures and Web services, the Semantic Web and agent technologies. Each chapter introduces the essential elements of the method or technology, describes the state-or-the-art and speculates about how increasing adoption is likely to affect the required skills and practices of a software architect.

Acknowledgements

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). Your efforts and patience are greatly appreciated. Contact details for the contributing authors are as follows:

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

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

Dr Yan Liu, Empirical Software Engineering, National ICT Australia, email: jenny.liu@nicta.com.au

Paul Greenfield, School of IT, University of Sydney,
email: p.greenfield@computer.org

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, 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.

Ian Gorton
March 2006

Table of Contents

1 Understanding Software Architecture 1
- 1.1 What is Software Architecture? 1
- 1.2 Definitions of Software Architecture 2
  - 1.2.1 Architecture Defines Structure 3
  - 1.2.2 Architecture Specifies Component Communication 5
  - 1.2.3 Architecture Addresses Non-functional Requirements 6
  - 1.2.4 Architecture is an Abstraction 6
  - 1.2.5 Architecture Views 8
- 1.3 What Does a Software Architect Do? 10
- 1.4 Architectures and Technologies 11
- 1.5 Summary 13
- 1.6 Further Reading 13
  - 1.6.1 General Architecture 13
  - 1.6.2 Architecture Requirements 14
  - 1.6.3 Architecture Patterns 14
  - 1.6.4 Technology Comparisons 15

2 Introducing the Case Study 17
- 2.1 Requirements Overview 17
- 2.2 Project Context 18
- 2.3 Business Goals 20
- 2.4 Constraints 21
- 2.5 Summary 21

3 Software Quality Attributes 23
- 3.1 Quality Attributes 23
- 3.2 Performance 24
  - 3.2.1 Throughput 25
  - 3.2.2 Response Time 25
  - 3.2.3 Deadlines 26
  - 3.2.4 Performance for the ICDE System 27
- 3.3 Scalability 27
  - 3.3.1 Request Load 28
  - 3.3.2 Simultaneous Connections 29

3.3.3 Data Size....30
3.3.4 Deployment....31
3.3.5 Some Thoughts on Scalability....31
3.3.6 Scalability for the ICDE Application....31
3.4 Modifiability....31
3.4.1 Modifiability for the ICDE Application....33
3.5 Security....33
3.5.1 Security for the ICDE Application....34
3.6 Availability....34
3.6.1 Availability for the ICDE Application....35
3.7 Integration....35
3.7.1 Integration for the ICDE Application....36
3.8 Other Quality Attributes....37
3.9 Design Trade-Offs....38
3.10 Summary....38
3.11 Further Reading....39

4 A Guide to Middleware Architectures and Technologies....41
4.1 Introduction....41
4.2 Technology Classification....42
4.3 Distributed Objects....43
4.4 Message-Oriented Middleware....46
4.4.1 Message-Oriented Middleware Basics....46
4.4.2 Exploiting Message Oriented Middleware Advanced Features....49
4.4.3 Publish-Subscribe....54
4.5 Application Servers....59
4.5.1 Enterprise JavaBeans....60
4.5.2 EJB Component Model....61
4.5.3 EJB Programming....63
4.5.4 Deployment Descriptors....67
4.5.5 Responsibilities of the EJB Container....69
4.5.6 Some Thoughts....70
4.6 Message Brokers....71
4.7 Business Process Orchestration....78
4.8 Integration Architecture Issues....82
4.9 Summary....87
4.10 Further Reading....88
4.10.1 CORBA....88
4.10.2 Message-Oriented Middleware....88
4.10.3 Application Servers....89
4.10.4 Integration Middleware....89

5 A Software Architecture Process 91
5.1 Process Outline 91
5.1.1 Determine Architectural Requirements 92
5.1.2 Identifying Architecture Requirements 93
5.1.3 Prioritizing Architecture Requirements 94
5.2 Architecture Design 95
5.2.1 Choosing the Architecture Framework 97
5.2.2 Allocate Components 106
5.3 Validation 108
5.3.1 Using Scenarios 109
5.3.2 Prototyping 112
5.4 Summary and Further Reading 113

6 Documenting a Software Architecture 115
6.1 Introduction 115
6.2 What to Document 116
6.3 UML 2.0 117
6.4 Architecture Views 119
6.5 More on Component Diagrams 122
6.6 Architecture Documentation Template 125
6.7 Summary and Further Reading 126

7 Case Study Design 129
7.1 Overview 129
7.2 ICDE Technical Issues 129
7.2.1 Large Data 129
7.2.2 Notification 131
7.2.3 Data Abstraction 132
7.2.4 Platform and Distribution Issues 132
7.2.5 API Issues 132
7.2.6 Discussion 133
7.3 ICDE Architecture Requirements 134
7.3.1 Overview of Key Objectives 134
7.3.2 Architecture Use Cases 134
7.3.3 Stakeholder Architectural Requirements 135
7.3.4 Constraints 137
7.3.5 Non-functional Requirements 137
7.3.6 Risks 138
7.4 ICDE Solution 138
7.4.1 Relevant Architectural Patterns 138
7.4.2 Architecture Overview 138
7.4.3 Structural Views 139
7.4.4 Behavioral Views 143

7.4.5 Implementation Issues 146
7.5 Architecture Analysis 147
7.5.1 Scenario Analysis 147
7.5.2 Risks 148
7.6 Summary 148

8 Looking Forward 149
8.1 The Challenges of Complexity 149
8.1.1 Business Process Complexity 150
8.1.2 Agility 151
8.1.3 Reduced Costs 152
8.2 What Next? 154

9 Software Product Lines 155
9.1 Product Lines for ICDE 155
9.2 Software Product Lines 156
9.3 Benefiting from SPL Development 158
9.3.1 Product Lines for ICDE 160
9.4 Product Line Architecture 160
9.4.1 Reuse Mechanisms 161
9.4.2 SCM for Reuse 163
9.4.3 Variation Mechanisms 164
9.4.4 Product Line Architecture for ICDE 166
9.5 Adopting Software Product Line Development 166
9.5.1 Starting Points for Adopting SPL Development 167
9.6 Product Line Adoption Practice Areas 169
9.6.1 Product Line Adoption for ICDE 170
9.7 Ongoing Software Product Line Development 170
9.7.1 Change Control 171
9.7.2 Architectural Evolution for SPL Development 173
9.7.3 Product Line Development Practice Areas 174
9.7.4 Product Lines with ICDE 174
9.8 Conclusions 176
9.9 Further Reading 177

10 Aspect Oriented Architectures 179
10.1 Aspects for ICDE Development 179
10.1.1 Introduction to Aspect-Oriented Programming 180
10.1.2 Crosscutting Concerns 181
10.1.3 Managing Concerns with Aspects 181
10.1.4 AOP Syntax and Programming Model 182
10.1.5 Weaving 184
10.1.6 Example of a Cache Aspect 185

10.2 Aspect-Oriented Architectures 186
10.2.1 Architectural Aspects and Middleware 187
10.3 State-of-the-Art 188
10.3.1 Aspect Oriented Modeling in UML 188
10.3.2 AOP Tools 189
10.3.3 Annotations and AOP 189
10.4 Performance Monitoring of ICDE with AspectWerkz 190
10.5 Conclusions 194
10.6 Futhur Reading 195

11 Model-Driven Architecture 197
11.1 Model-Driven Development for ICDE 197
11.2 What is MDA 199
11.3 Why MDA? 202
11.3.1 Portability 203
11.3.2 Interoperability 203
11.3.3 Reusability 204
11.4 State-of-Art Practices and Tools 205
11.4.1 AndroMDA 205
11.4.2 ArcStyler 206
11.4.3 Eclipse Modelling Framework (EMF) 206
11.5 MDA and Software Architecture 207
11.5.1 MDA and Non-Functional Requirements 208
11.5.2 Model Transformation and Software Architecture 208
11.5.3 SOA and MDA 209
11.5.4 Analytical Models are Models too 210
11.6 MDA for ICDE Capacity Planning 211
11.7 Summary and Further Reading 214

12 Service-Oriented Architectures and Technologies 217
12.1 Service-Oriented Architecture for ICDE 217
12.2 Background 218
12.3 Service-Oriented Systems 219
12.3.1 Boundaries are Explicit 222
12.3.2 Services are Autonomous 223
12.3.3 Share Schemas and Contracts, not Implementations 224
12.3.4 Service Compatibility is Based on Policy 224
12.4 Web Services 225
12.5 SOAP and Messaging 227
12.6 UDDI, WSDL and Metadata 230
12.7 Security, Transactions and Reliability 232
12.8 Web Services and the Future of Middleware 233
12.9 ICDE with Web Services 234

12.10 Conclusion and Further Reading 236

13 The Semantic Web 239
13.1 ICDE and the Semantic Web 239
13.2 Adaptive, Automated, and Distributed 240
13.3 The Semantic Web 241
13.3.1 Metadata 241
13.3.2 Semantics 244
13.4 Ontologies in ICDE 246
13.5 Semantic Web Services 248
13.6 Cautious Optimism 249
13.7 Further Reading 251

14 Software Agents: An Architectural Perspective 253
14.1 Agents in the ICDE Environment 253
14.2 What is an Agent? 253
14.3 Abstraction Revisited 257
14.4 An Example Agent Technology 258
14.5 Architectural Implications 264
14.5.1 Concurrency 264
14.5.2 Scalability 264
14.5.3 Mobility 266
14.6 Agent Technologies 267
14.7 Conclusions 267
14.8 Further Reading 268

15 Concluding Thoughts 271
15.1 Challenges 271
15.1.1 Architecture Knowledge Management 271
15.1.2 Adaptive Architectures 273

1 Understanding Software Architecture

1.1 What is Software Architecture?

The last decade has seen a tremendous rise in the prominence of a software engineering sub-discipline known as software architecture. *Technical Architects* and *Chief Architects* are job titles that now abound in the software industry. There's a Worldwide Institute of Software Architects[1], and even a certain well-known wealthiest person on earth has architect in his job title. 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. 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. 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.wwisa.org/

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:

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

"[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 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 inter-related 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. Both 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 man-

[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*

ner, 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.

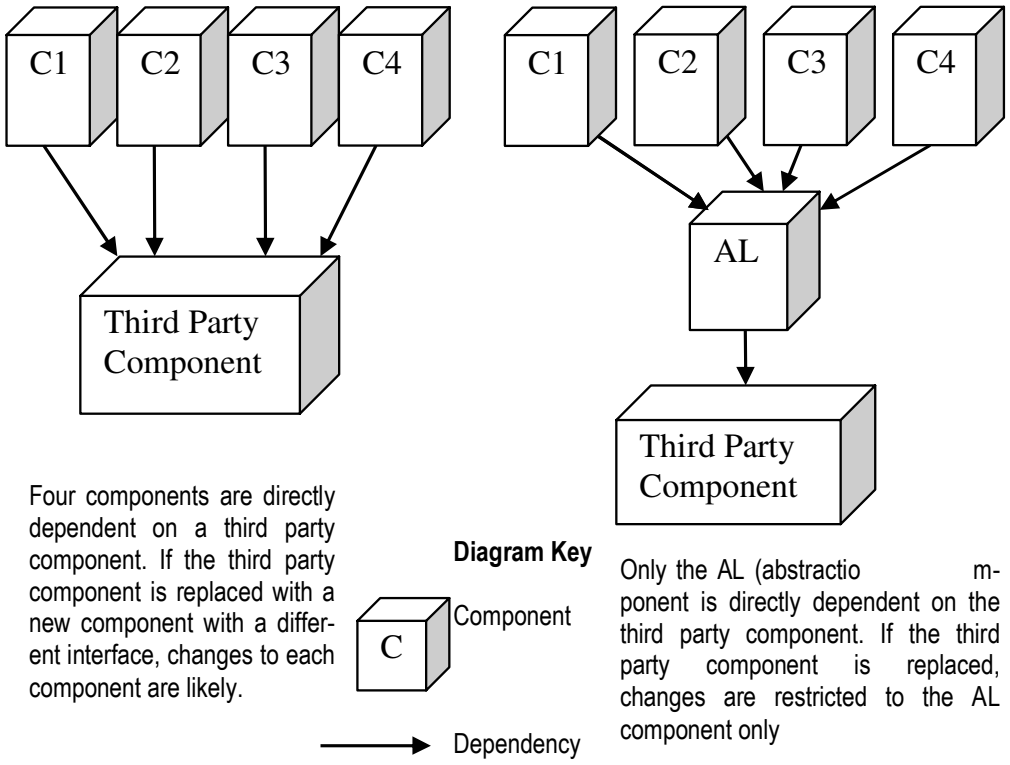

Fig. 1. Two examples of component dependencies

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).

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 to satisfy particular types of requirements. For example, the client-server pattern 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.

---

[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!

1.2.3 Architecture Addresses Non-functional Requirements

Non-functional 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 non-functional 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 non-negotiable.
- **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 non-negotiable.
- **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 non-functional requirements.

1.2.4 Architecture is an Abstraction

One of the most useful, but often non-existent, 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, 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 application. 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. 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. 2 it's likely that the three components in the top level description will be designed and built by different teams working on the 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 non-functional 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 throughput. Regardless, this further refinement creates a structure that defines and constrains the detailed design of these components.

The simple architecture in Fig. 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 non-functional 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.

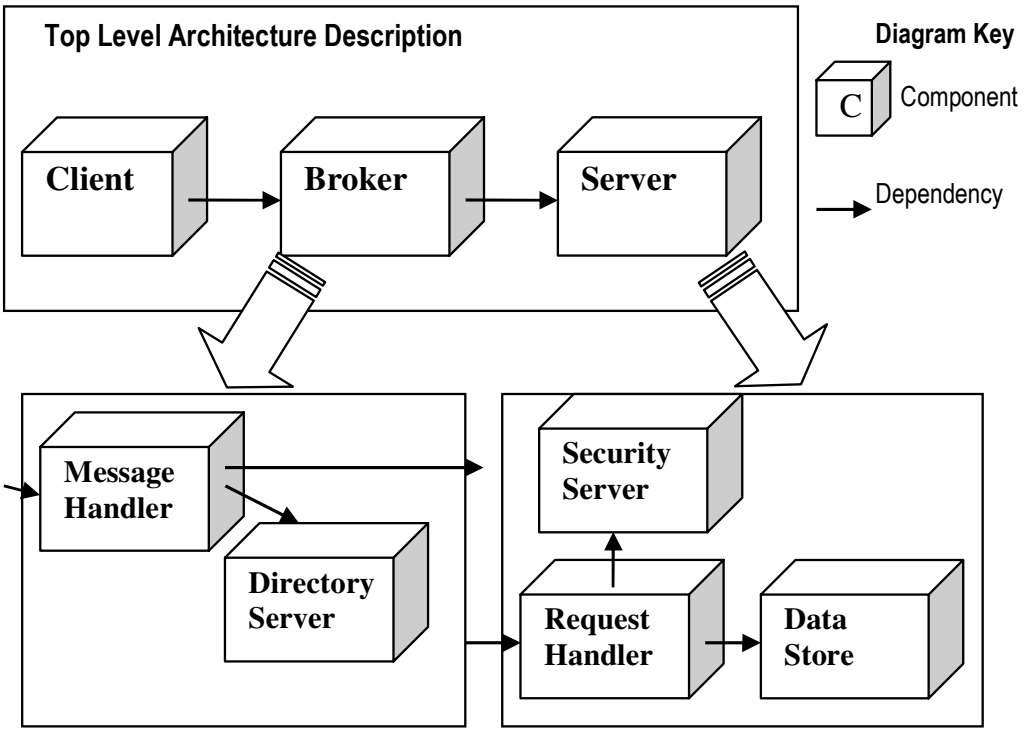

Fig. 2. Describing an architecture hierarchically

1.2.5 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 multi-threaded or replicated components, and the synchronous or asynchronous communication mechanisms used.

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

- **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.
- **Development view:** This captures the internal organization of the software components, typically as they are held in a development environment or configuration management tool. For example, the depiction of a nested package and class hierarchy for a Java application would represent the development view of an architecture.

These views are tied together by the architecturally significant use cases (often called scenarios). These basically capture the requirements for the architecture, and hence are related to more than one particular view. By working through the steps in a particular use case, the architecture can be "tested", by explaining how the design elements in the architecture respond to the behavior required in the use case. We'll explore how to do this 'architecture testing' in Chapter 5.

Since Krutchen's paper, there's been much thinking, experience and development in the area of architecture views. Mostly notably is the work from the SEI, colloquially known as the "Views and Beyond" approach (see Further Reading). This recommends capturing an architecture model using three different views:

- **Module:** This is a structural view of the architecture, comprising the code modules such as classes, packages and subsystems in the design. It also captures module decomposition, inheritance, associations and aggregations.
- **Component and Connector:** This view describes the behavioral aspects of the architecture. Components are typically objects, threads or processes, and the connectors describe how the components interact. Common connectors are sockets, middleware like CORBA or shared memory.
- **Allocation:** This view shows how the processes in the architecture are mapped to hardware, and how they communicate using networks and/or databases. It also captures a view of the source code in the configuration management systems, and who in the development group has responsibility for each modules.

The terminology used in "Views and Beyond" is strongly influenced by the architecture description language (ADL) research community. This community has been influential in the world of software architecture, but

has had limited impact on mainstream information technology. So while this book will concentrate on two of these views, we'll refer to them as the structural view and the behavioral view. Discerning readers should be able to work out the mapping between terminologies.

1.3 What Does a Software Architect Do?

The environment that a software architect works in tends to define their exact roles and responsibilities. A good general description of the architect's role is maintained by the SEI on their web site[6]. Instead of summarizing this, I'll briefly describe, in no particular order, four essential skills for a software architect, regardless of their professional environment.

- **Liaison:** Architects play many liaison roles. They liaise between the customers or clients of the application and the technical team, often in conjunction with the business and requirements analysts. They liaise between the various engineering teams on a project, as the architecture is central to each of these. They liaise with management, justifying designs, decisions and costs. They liaise with the sales force, to help promote a system to potential purchasers or investors. Much of the time, this liaison takes the form of simply translating and explaining different terminology between different stakeholders.
- **Software Engineering:** Excellent design skills are what get a software engineer to the position of architect. They are an essential pre-requisite for the role. More broadly though, architects must promote good software engineering practices. Their designs must be adequately documented and communicated and their plans must be explicit and justified. They must understand the downstream impact of their decisions, working appropriately with the application testing, documentation and release teams.
- **Technology Knowledge:** Architects have a deep understanding of the technology domains that are relevant to the types of applications they work on. They are influential in evaluating and choosing third party components and technologies. They track technology developments, and understand how new standards, features and products might be usefully exploited in their projects. Just as importantly, good architects know what they don't know.

[6] http://www.sei.cmu.edu/ata/arch_duties.html

- **Risk Management** Good architects tend to be cautious. They are constantly enumerating and evaluating the risks associated with the design and technology choices they make. They document and manage these risks in conjunction with project sponsors and management. They develop and instigate risk mitigation strategies, and communicate these to the relevant engineering teams. They try to make sure no unexpected disasters occur.

Look for these skills in the architects you work with or hire. Architects play a central role in software development, and must be multi-skilled in software engineering, technology, management and communications.

1.4 Architectures and Technologies

Architects must make design decisions early in a project lifecycle. Many of these are difficult, if not impossible, to validate and test until parts of the system are actually built. Judicious prototyping of key architectural components can help increase confidence in a design approach, but sometimes it's still hard to be certain of the success of a particular design choice in a given application context.

Due to the difficulty of validating early design decisions, architects sensibly rely on tried and tested approaches for solving certain classes of problems. This is one of the great values of architectural patterns. They enable architects to reduce risk by leveraging successful designs with known engineering attributes.

Patterns are an abstract representation of an architecture, in the sense that they can be realized in multiple concrete forms. For example, the publish-subscribe architecture pattern describes an abstract mechanism for loosely coupled, many-to-many communications between publishers of messages and subscribers who wish to receive messages. It doesn't however specify how publications and subscriptions are managed, what communication protocols are used, what types of messages can be sent, and so on. These are all considered implementation details.

Unfortunately, abstract descriptions of architectures don't yet execute on computers, either directly or through rigorous transformation. Until they do, abstract architectures must be reified by software engineers as concrete software implementations.

Fortunately, software products vendors have come to the rescue. Widely utilized architectural patterns are supported in a variety of commercial off-the-shelf (COTS) technologies. If a design calls for publish-subscribe mes-

saging, or a message broker, or a three-tier architecture, then the choices of available technology are many and varied indeed. This is an example of software technologies providing reusable, application-independent software infrastructures that implement proven architectural approaches.

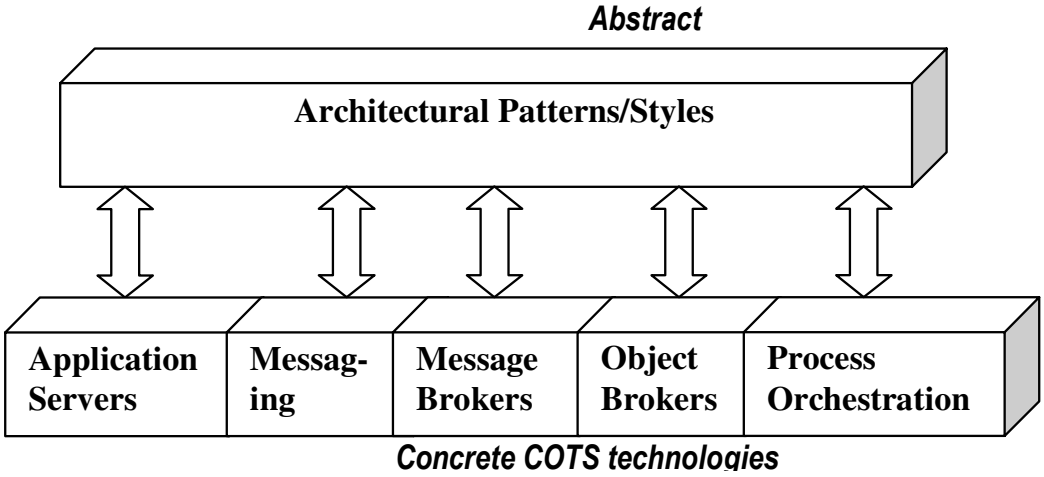

Fig. 3. Mapping between logical architectural patterns and concrete technologies

As Fig. 3 depicts, several classes of COTS technologies are used in practice to provide packaged implementations of architectural patterns for use in IT systems. Within each class, competing commercial and open source products exist. Although these products are superficially similar, they will have differing feature sets, be implemented differently and have varying constraints on their use.

Architects are somewhat simultaneously blessed and cursed with this diversity of product choice. Competition between product vendors drives innovation, better feature sets and implementations, and lower prices, but it also places a burden on the architect to select a product that has quality attributes that satisfy the application requirements. All applications are different in some ways, and there is rarely, if ever, a *one-size-fits-all* product match. Different COTS technology implementations have different sets of strengths and weaknesses and costs, and consequently will be better suited to some types of applications than others.

The difficulty for architects is in understanding these strengths and weaknesses early in the development cycle for a project, and choosing an appropriate reification of the architectural patterns they need. Unfortunately, this is not an easy task, and the risks and costs associated with selecting an inappropriate technology are high. The history of the software industry is littered with poor choices and subsequent failed projects.

Chapter 4 provides a detailed description and analysis of these infrastructural technologies.

1.5 Summary

Software architecture is a fairly well defined and understood design discipline. However, just because we know what it is and more or less what needs doing, this doesn't mean it's mechanical or easy. Designing and validating an architecture for a complex system is a creative exercise, requiring considerable knowledge, experience and discipline. The difficulties are exacerbated by the early lifecycle nature of much of the work of an architect. To my mind, the following quote from Philippe Krutchen sums up an architect's role perfectly:

"*The life of a software architect is a long (and sometimes painful) succession of sub-optimal decisions made partly in the dark"*

The remainder of this book will describe methods and techniques that can help you to shed at least some light on architectural design decisions. Much of this light comes from understanding and leveraging design principles and supporting technologies that have proven to work in the past. Armed with this knowledge, you'll be able to tackle complex architecture problems with more confidence, and after a while, perhaps even a little panache.

1.6 Further Reading

There are lots of good books, reports and papers available in the software architecture world. Below are some I'd especially recommend. These expand on the information and messages covered in this chapter.

1.6.1 General Architecture

In terms of defining the landscape of software architecture, and describing their project experiences, mostly with defense projects, it's difficult to go past the following books from members of the Software Engineering Institute.

L. Bass, P. Clements, R Kazman. Software Architecture in Practice, Second Edition. Addison-Wesley, 2003.

P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford. *Documenting Software Architectures: Views and Beyond.* Addison-Wesley, 2002.

P. Clements, R. Kazman, M. Klein. *Evaluating Software Architectures: Methods and Case Studies.* Addison-Wesley, 2002.

For a description of the 'Decomposition Style', see *Documenting Software Architecture*, page 53. And for an excellent discussion of the *uses* relationship and its implications, see the same book, page 68.

1.6.2 Architecture Requirements

The original book describing use-cases is:

I. Jacobson, M. Christerson, P. Jonsson, G. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1992.

Responsibility-driven design is an incredibly useful technique for allocating functionality to components and sub-systems in an architecture. The following should be compulsory reading for architects.

R. Wirfs-Brock, A. McKean. Object Design: Roles, Responsibilities, and Collaborations. Addison-Wesley, 2002.

1.6.3 Architecture Patterns

There's a number of fine books on architecture patterns. Buschmann's work is an excellent introduction.

F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal,. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns.* John Wiley & Sons, 1996.

D. Schmidt, M. Stal, H. Rohnert, F. Buschmann. *Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects.* John Wiley & Sons, 2000.

Two recent books that focus more on patterns for enterprise systems, especially enterprise application integrations, are well worth a read.

M. Fowler. Patterns of Enterprise Application Architecture. Addison-Wesley, 2002.

G. Hohpe, B. Woolf. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley, 2003

1.6.4 Technology Comparisons

A number of papers that emerged from the Middleware Technology Evaluation (MTE) project give a good introduction into the issues and complexities of technology comparisons.

P.Tran, J.Gosper, I.Gorton. *Evaluating the Sustained Performance of COTS-based Messaging Systems*. in Software Testing, Verification and Reliability, vol 13, pp 229-240, Wiley and Sons, 2003.

I.Gorton, A Liu. *Performance Evaluation of Alternative Component Architectures for Enterprise JavaBean Applications,* in IEEE Internet Computing, vol.7, no. 3, pages 18-23, 2003.

A.Liu, I. Gorton. *Accelerating COTS Middleware Technology Acquisition: the i-MATE Process.* in IEEE Software, pages 72-79,volume 20, no. 2, March/April 2003.