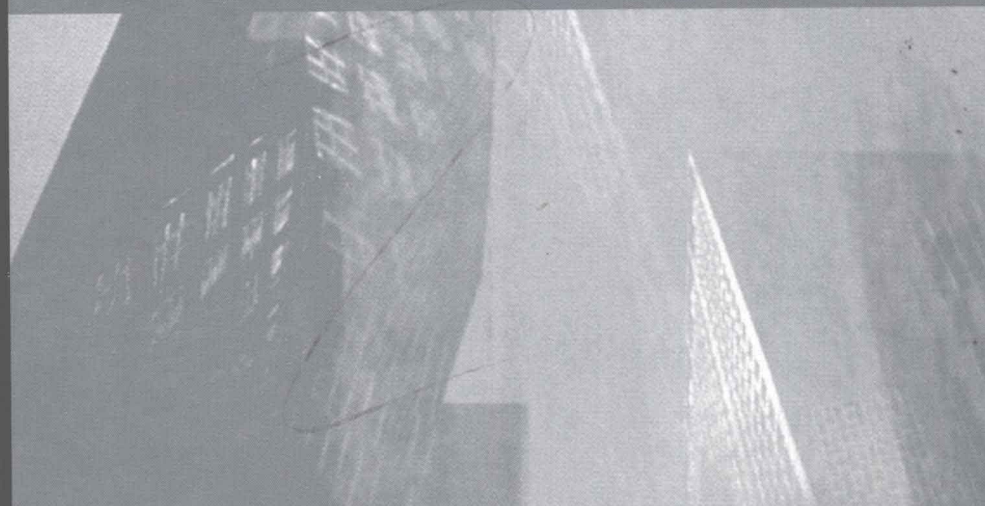
ADVANCED TOPICS IN SCIENCE AND TECHNOLOGY IN CHINA

Zheng Qin
Jiankuan Xing
Xiang Zheng

Software Architecture



ZHEJIANG UNIVERSITY PRESS
浙江大学出版社

Springer

Zheng Qin
Jiankuan Xing
Xiang Zheng

Software Architecture

With 161 figures

ZHEJIANG UNIVERSITY PRESS
浙江大学出版社

Springer

图书在版编目 (CIP) 数据

软件体系结构= Software Architecture / 覃征，邢剑宽，郑翔著. —杭州：浙江大学出版社，2007.10
ISBN 978-7-308-05453-9

Ⅰ.软… Ⅱ.①覃…②邢…③郑… Ⅲ.软件-系统结构 Ⅳ.TP311.5

中国版本图书馆 CIP 数据核字 (2007) 第 169644 号

软件体系结构

覃征 邢剑宽 郑翔 著

封面设计	Joe Piliero
责任编辑	傅 强 尤建忠
出版发行	浙江大学出版社
	网址：http://www.zjupress.com
	Springer-Verlag GmbH
	网址：http://www.springer.com
排　　版	浙江大学出版社电脑排版中心
印　　刷	富阳市育才印刷有限公司
开　　本	787mm×960mm 1/16
印　　张	22
字　　数	719千
版 印 次	2007年10月第1版 2007年10月第1次印刷
书　　号	ISBN 978-7-308-05453-9 (浙江大学出版社)
	ISBN 978-3-540-74342-2 (Springer-Verlag GmbH)
定　　价	120.00元

Preface

Building software nowadays is far more difficult than it can be done several decades ago. At that time, software engineers focused on how to manipulate the computer to work and then solve problems correctly. The organization of data and implementation of algorithm were the crucial process of software designing then. However, more and more tasks in low level, such as memory management and network communication, have been automatized or at least can be reused with little effort and cost. Programmers and designers, with the help of high level programming languages and wieldy development tools, can pay more attention to problems, rather than bury themselves into the machine code manuals. However, the side effect of these utilities is that more complicated problems are given according to the requirements from military, enterprise and so on, in which the complexity grows rapidly day by day. We believe that software architecture is a key to deal with it.

Many people become aware of the existence of software architecture just recently. Nevertheless, it in fact has a long history, which may surprise you. Before the invention of C++or even C, some computer scientists had begun to notice the concept of software structure and its influence to software development. In the 1990s, software architecture started its journey of bloom, when several communities, workshops and conferences were held with a great amount of published articles, books and tools. Today, software architect, the job of taking software designing, analysis and dealing with different concerns and requirements from different stakeholders, is considered as the center of development team.

But there is an ironical problem that most existing architects in fact do not take any study or training in this field, some of whom even do not realize that software architecture is a kind of realm requiring academic effort, just as artificial intelligence or data mining. The reason is that software architecture has no widely-accepted definitions and standards of basic theories and practical methods, which leads to that there is almost no universal course about this subject. Meanwhile, the rapid growth and division of software architecture result in too many branches and sub-fields, most of which still keep non-dominant and unified. These changes aggregate the trouble in learning even a subset of software architecture area. In this book, we will provide an overview among the classic theories and some latest progresses of software architecture and try to touch the software architecture's essence.

This book is a collaboration of three authors: Zheng Qin, Jiankuan Xing and Xiang Zheng. More particularly, Professor Qin is the primary author who decides the contents and issues what you can see in this book. And Jiankuan Xing organizes the work of writing, and facilitates the cooperation with authors and other contributers.

Targets

This book aims to give an introduction to the theory foundations, various sub-fields, current research status and practical methods of software architecture. In this book, readers can acquire the basic knowledge of software architecture, including why software architecture is necessary, how we can describe a system's architecture by formal language, what architecture styles are popular for practice use and how we can apply software architecture into the development of systems. Study cases, data, illustrations and other materials which are released in the recent years will be used to show the latest development of software architecture. This book can be used as the learning material for touching software architecture.

How to Read This Book

We target to give readers an inside-out understanding of software architecture, therefore this book is divided into two parts (not shown explicitly in content):

- Basic Theories: Chapter 1—Chapter 5
- Advance Topics: Chapter 6—Chapter 9

In detail, we give the overview descriptions for each chapter as follows:

Chapter 1: Introduction. The theme of this chapter is the basic introduction to software architecture, where readers will see why we need it, how it emerged and what its definitions look like. We hope to give readers a clear vision on it, considering a great many misunderstanding and arguments' presence. In addition, with the development of research, concerns and usage of software architecture have become different, which we will mention at the last section of this chapter.

Chapter 2: Architectural Styles and Patterns. Initially, the research on software architecture emphasized the categorization of software in architectural level. Some systems share the common structure and properties are classified into one set in which the same vocabulary and similar models for representing these systems can be used. Each vocabulary and models specified for a category are called "architectural style". What's more, we abstract and represent some representative structure and reuse them with style. Each structure is called an "architectural pattern". Architecture styles and patterns are very precise utilities for constructing complex systems. In Chapter 2, we provide descriptions, study cases and comparison of them.

Chapter 3: Application and Analysis of Architectural Styles. After characterizing several popular styles, we continue to offer a few study cases, each of which combines more than one architectural style. Academically, this is called "heterogeneous style constructing". As a matter of fact, applied software always uses multiple styles simultaneously, no matter how simple they are. The goal of this chapter is to tie the abstract styles to practice use.

Chapter 4: Software Architecture Description. How to describe software architecture is the centric subject of architecture realm, because it is the foundation to represent software design, perform effective communications among stakeholders and measure systems' behaviors according to requirements. In this chapter, we pay attention to architectural formal description, which stands on the mathematic basis. However, for UML, the language widely used as architecture representation in practice, you can find excessive materials about it.

Chapter 5: Design Strategies in Architecture Level. This chapter gives you a chance to touch the concept of architectural design with formal foundation. In contrast to practical software development processes, such as RUP (Rational Unified Process), formal architectural design strategies stress the relationship and calculus of function space and structure space, both of which abstract the development process performed in the real world. To get through with this chapter, a fair capability of set theory and automata theory is required.

Chapter 6: Software Architecture IDE. Although software architecture is useful for software development, using it with pure handwork incurs too much overhead, and then time and cost, to the development process, which may obliterate its benefits. That's the key why software architecture was not popularly accepted in the 1990s. Now, we have the handy assist, software architecture IDE. The purpose of IDE is to enable an organization to manage its software architecture and other related actions and processes in a way that meets business needs by providing a foundational utility upon which design, communication, framework code generation and validation can be carried out automatically.

Chapter 7: Evaluating Software Architecture. After the initial architectural design is finished, any stakeholder would finger out whether this design is good or not, whether it will contribute to a successful development and then output the satisfying production or doom to crush resulting from the design defects. That's the evaluation's task. In this chapter, currently widely-used evaluation methods are discussed and compared. However, evaluation methods still lack the formal foundation, and more focus on the experience and capability of participators. Therefore, the description here will bring you the practical architectural methods and technologies, based on which evaluation is performed.

Chapter 8: Flexible Software Architecture. Flexible software architecture means the structure of a system which can metamorphose during runtime according to users' instructions, executing environment's changes or other requirements and the related actions and processes. That's crucial for systems' needs of self-healing and self-adaptation abilities. The systems with these needs before normally mix the structure metamorphosis code and application code, which insults more trouble in maintaining and improving procedures. What's more, failing to divide this confusion causes the system as conceived and the system as built to diverge over time. In this chapter, we give an introduction to what flexible software in architecture level looks like and what the principles and organization patterns of constructing it are.

Chapter 9: A Vision on Software Architecture. This is a chapter far away from theories, methods and technologies, in which the applications of software architecture in current software industry and in other fields, such as medicine, electronic engineering and military are presented in general. After that, we will provide several future research directions of software architecture at the end of this book.

Considering the relative independence of each chapter, readers can choose several chapters they are interested in. But we recommend Chapter 1 should be read carefully since it can help you understand other chapters easier and better. In addition, you can find more detail and deeper description about some topics through the reference materials we give.

Who Should Read This Book

The graduates and undergraduates whose majors are elated to software design and development will benefit much from this book. Also, other people who are interested in software architecture would be guided to this field by reading this book. Then, experienced software designers and project leaders who want to adopt architecture as the centric concerns and utility of their software development process are our target readers, too. But they may suffer pain for a moment when converting their original mind to the new world, from which they will at last benefit. We assume our readers should have simple experience as follows. (Each capability may only be involved in several chapters rather than the whole book)

- Programming using C++, Java or C#
- Software design (even a simple project would be fine)
- Software project management

Acknowledgements

It is a great pleasure to acknowledge the profound and original work of Software Architecture Group of Tsinghua Univ., especially Jiankuan Xing (Chapters 1, 5, 7, 8) and Xiang Zheng (Chapters 3, 4). Their insights, collaboration and diligence have been a constant source which gestates the publication of this book.

For the current years I have been considering the problems of software

architecture. During the book's writing, we have profited greatly by collaboration with many people, including Kaimo Hu, who prepares lots of materials for Chapters 2 and 9. Meanwhile, he often inspired us with wide knowledge and ideas; and Juan Wang who buried herself into various software architecture IDEs and taught us how to use them in a great detail, which contributed much for Chapter 6. She is also participating the XArch project focusing on ADL parsing and model generating. And many thanks to Hui Cao, a nice reader who has inspected most manuscript and offered valuable criticisms and comments.

Beijing
June 2007

Zheng Qin

ADVANCED TOPICS
IN SCIENCE AND TECHNOLOGY IN CHINA

ADVANCED TOPICS
IN SCIENCE AND TECHNOLOGY IN CHINA

Zhejiang University is one of the leading universities in China. In Advanced Topics in Science and Technology in China, Zhejiang University Press and Springer jointly publish monographs by Chinese scholars and professors, as well as invited authors and editors from abroad who are outstanding experts and scholars in their fields. This series will be of interest to researchers, lecturers, and graduate students alike.

Advanced Topics in Science and Technology in China aims to present the latest and most cutting-edge theories, techniques, and methodologies in various research areas in China. It covers all disciplines in the fields of natural science and technology, including but not limited to, computer science, materials science, the life sciences, engineering, environmental sciences, mathematics, and physics.

Contents

1

Introduction to Software Architecture

Compared to the traditional software several decades ago which were simple machine instructions or the combination of data structures and algorithms, current software are more complicated and harder to control and maintain. Normally, software systems are constructed through the assembly of components, whatever those which are developed according to new specifications or those which are stored in the libraries. In this circumstance, a team is needed to face different facets of the system. Some of them deal with the necessary functions to be implemented or reused in components, while others have to focus on how the work from different divisions can be coordinated and communicated correctly. Meanwhile, in this process some qualities of software must be guaranteed in order to approach the success.

Software architecture is a rising subject of software engineering to help people solve problems mentioned above. With it, designers or project managers have the chance to oversee the status of software in a high level. In addition, software architecture can be reused, resulting in the saving of huge cost and the reduction of risks within the development processes and the activities after them, including designing, modeling, implementation, test, evaluation, maintaining and evolution.

However, tracking software architecture is difficult, because it always hides itself behind what you can touch. Visualizing it requires a deep grasp of global information of systems as well as excellent skills and methods. People from different organizations or enterprises use different strategies to handle it, but most of them have something in common. Abstract and summary of these experiences have become the foundation of software architecture science today.

In this chapter, we start from the history of software development, trying to uncover the origin of software architecture. Then we discuss the definitions and meanings of architecture and other related activities. At last, we focus on what benefits we will gain from it.

1.1 A Brief History of Software Development

Revolutions in software development paradigm are not singular since the word "software" was approximately born in the 1940s when the initial stored-program computers emerged. Each shift, along with development methodologies, patterns and tools, occurred to meet new environment and requirements. We believe that software architecture is the next revolution. Many people have begun to follow this trend, while, however, many others do not care about it, just as several years ago the people who were reluctant to change their habits and use new development technologies. Upon history level, we can get more clear sight of how software architecture gradually becomes crucial for current software industry and why we should change our manner of work to follow it.

1.1.1 The Evolution of Programming Language—Abstract Level

Abstract is the process that simplifies the real systems, activities or other entities by ignoring or factoring out those trivial details without missing their essential running mechanisms. To construct a solution with a computer, we abstract it and implement it with programming language, in which the target model of abstract greatly affects what programmers see that problem. The progress of programming languages so far regularly increases their abstract level, transforming the emphases from machine manipulating to problem solving.

In the 1950s, stored-program computers became popular and thereby monopolized programmers' work manner at that time. Programmers used machine instructions which can be executed directly by their computers and data with naïve categories such as byte, word, double word to express their logic. The layout of instructions and data in memory had to be controlled by hand, that is, programmers must keep in mind where the beginning and end positions of each constant and variable exactly are. When the program needed update, programmers spent a lot of time to check and modify every reference for data or code position that needs a movement to keep program's consistency.

Soon, some people were aware of that these functions could be automated and reused. Therefore, symbolic substitution and subroutine technology were created. The great thing about these was that they liberated you from those trivial but important works for the machine. However, commonly useful patterns, such as conditional control structure, loop structure, evaluation of numeric computation expressions, still had to be decomposed to simple control and computation instructions that machine was able to carry out, which drew programmers' much attention to the computation's realization rather than the problem itself. This improved the high-level programming. In the middle of the 1960s, FORTRAN from IBM became the dominant programming language in scientific computation for its convenience and high-efficiency.

In the latter part of 1960s, Ole-Johan Dahl and Kristen Nygaard created Simula, a superset programming language of Algol, introducing the object-oriented paradigm. The data type in FORTRAN serves to construct a map between FORTRAN types to machine primitive data types. On the contrary, object-oriented paradigm considers data type as the abstraction of entities from real problems. Although FORTRAN and C also have the utility such as "structure" and "union", they are just the accumulation of data in that data type and operations specific to this type are separated, and object-oriented rules, including encapsulation, implementation hiddenness, access control and polymorphism are not touched. With the growth of C++, a widely accepted object-oriented language, the programming world was thoroughly changed.

The prime goal of C++ or other contemporary object-oriented languages was to put class as the basic reuse unit. However, the design and realization themselves of these languages doomed to fail. On the one hand, absence of class meta data ruins the promise of the update capability of a class's implementation; on the other hand, disregard of the separation between the communication contracts among classes and classes' implementation limits their capability of reuse. We can see that majority of reuse performed in C++ stand on source code level, while reuse in binary level may introduce more problems than its benefits. You can find more details about this subject in (Joyner, 1996). When people find that software can be assembled by several independent parts and thus can reduce the cost and time in building larger system, it is clear that finding a proper reuse unit or establishing principles for this kind of unit is crucial. (Ning, 1996) gave the first complete picture of component-based software development model.

Component further raises the design level by increasing the concept size of building block in software. The great thing about this is that it permits designers to construct a system by using interindependent components, under the premise of that strict communication contracts are defined and followed. Object-oriented paradigm is a good basis for component development model, but not each component must be implemented by objects. After the middle of the 1990s, COM and CORBA became popular because they extended C++ or other languages to meet component model's requirements and principles. Java and Net platform support development and deployment in component level since their birth, with the help of explicit utility of interface and meta information. What's more, the design model created by UML can be easily converted to the source code in these two platforms. UML combines concepts, advice and experience of countless designers, software engineers, methodologists and domain experts to provide a suit of fundamental notations, with which people care only components and the relationships, constraints among them. In other words, UML achieves the peak of abstract level so far.

We believe software architecture will bring next shift in software development paradigm. But just as the relation between high-level programming languages and UML, software architecture will not exterminate old methods and tools, but to complement them to deal with large-scale, rapid-changing software intensive