PEARSON Education

Software Architecture

A Case Based Approach

Vasudeva Varma

FOREWORD BY LEN BASS

Software Architecture

Supported by

Business Transformation. Together.

Satyam Learning World,
Satyam Computer Services Limited

Software Architecture

A CASE BASED APPROACH

Vasudeva Varma

PEARSON
Education

Chennai • Delhi • Chandigarh

ISBN 978-81-317-0749-4

First Impression

Published by Dorling Kindersley (India) Pvt. Ltd., licensees of Pearson Education in South Asia.

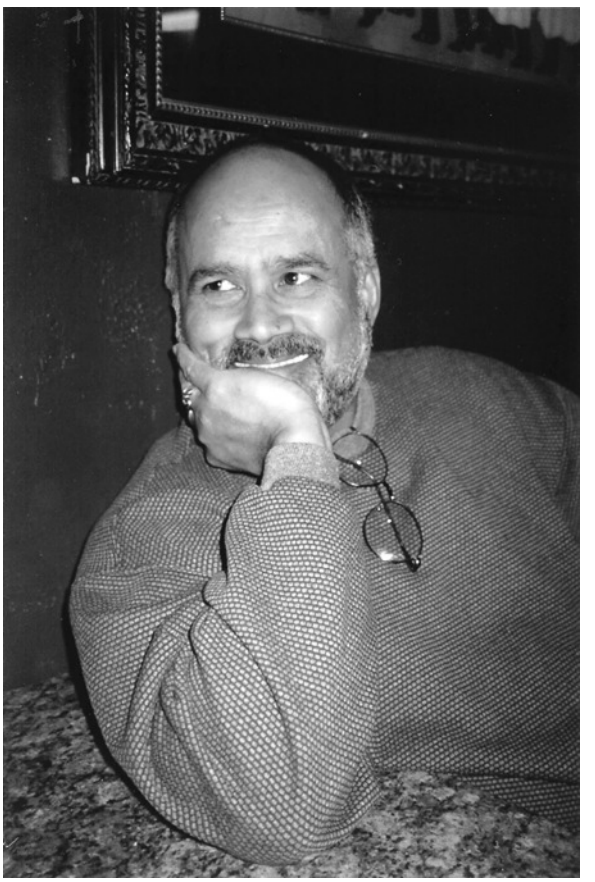
Head Office: 482, F.I.E., Patparganj, Delhi 110 092, India.
Registered Office: 14 Local Shopping Centre, Panchsheel Park, New Delhi 110 017, India.

Laser typeset by QuADS.

Printed in India

Dedicated to the fond memory of

Dr Sri Sridharan



A friend, philosopher, guide and an architect of more important things in life

Contents

Foreword

Every field uses case studies. Artists study the masters and the lesser amateurs. Civil engineers study bridges that stand up as well as bridges that fall down. The virtue of case studies is their concreteness. Students can see realizations of the concepts that might have been mainly abstractions before. A basic pattern such as model view controller becomes much more meaningful when the context in which it is used is observable.

Case studies can span eras as well. Forty years ago, the concept of "re-entrant" code was well known. Today, it is far less well known. I frequently run into programmers and designers who do not know the concept. The degradation of knowledge is inevitable because there are always new things to learn and only a limited amount of time to learn them. Consequently, some things must be left out of any curriculum. A case study of a 40-year-old system might still be illuminating because of this degradation of knowledge.

More case studies are always useful. New techniques and application domains are continually emerging and the new case studies can illuminate those techniques and domains. Organizational and cultural contexts change and case studies can define these contexts as well.

Software engineering provides challenges in constructing case studies. Software systems of any interest are large and ungainly. Not only are the systems themselves complicated but, usually, the organizational context is also complicated. A recent exercise I was involved in had 14 different stakeholder types. Add to this the fact that many of these stakeholders represent different organizations and the organizational context is inherently complicated.

A case study must present the system and its organizational context in sufficient detail so that students can draw lessons from the case study and, yet, not in so much detail that students must be totally immersed in the case study for extended periods of time in order to get anything of interest out.

Luckily, there are two moderating factors when considering case studies. The first is the abstraction ability of the human brain. I became interested in software architecture initially because I was fascinated by the amount of time a group of people could spend arguing about two circles and an arrow on a whiteboard. Add a third circle and several more arrows and literally months could be spent. The second moderating factor is an explanation of the case study. An instructor or a textbook can present guidance as to what to look for in the case study and what lessons to take away from the case study. If the case study has sufficient detail, the student can come up with alternative interpretations, and the exchange between the student and the instructor adds richness to the case study and to the lessons that the student is able to derive from the case study.

The case studies in the book provide as much detail as is possible in a chapter and provide an interpretation of the lessons. They span a variety of different application domains and circumstances. They also span a variety of different phases of the life cycle. This spanning of the life cycle allows the lessons from one case study to be applied to another. For example, one case study is about architectural evaluation. Another is about architecture design. The design fragment presented in the design case study can be

evaluated using the techniques discussed in the architectural evaluation case study. Another case study discusses re-factoring the architecture for performance. Can the architectures in both the architecture evaluation case study and the design case study be re-factored in case the performance requirements became more stringent?

Because the case studies are elaborated in some detail, every decision made by the development team can be second guessed by a student and discussed with the instructor. This will provide ample opportunity for independent work. For example, in the component-based case study, the development team decides to construct a content management system (the e-Publish component). Why construct this component rather than use one of the open source content management systems? This is a decision that was not explained in the case study, and a possible assignment is to analyse the factors that might have led to that choice.

Another virtue of the range of case studies is that different approaches to the same problem can be studied. Design, for example, is a matter of generate and test. That is, generate a hypothesis, test it and gain insight that enables the intelligent generation of a new hypothesis. In the case studies in this book, several different approaches to the generation of the initial hypothesis can be observed. In the component-based case study, the initial hypothesis is the concatenation of the components to be used in the solution. The next hypothesis in this case would add connectors to the components. The case study on the mobile trading system generated the initial hypothesis through examining particular use cases. Subsequent iterations might cause techniques to achieve various quality attributes to be added to the design.

In summary, examining case studies is an important portion of the education process for both students and professionals who have entered this field. An analysis of the case studies in this book and the explanations associated with them will help shorten the process by which software architects are created.

Len Bass
Pittsburgh, PA
USA

Preface

Software architecture is an unconventional discipline. It is still in its early years, and it holds a lot of promise. However, it is understood little by students, teachers and practitioners of software engineering, including those in the role of architects. On the one hand, it addresses the most important challenge of producing quality software, while on the other, it lacks proven methods, tools, techniques and theories. Theory in this discipline has so far followed practice and not the other way round.

Most of the existing literature on software architecture describes methodologies that can be applied during the "process" of designing software architecture, such as attribute-driven design (ADD) methodology, various architecture evaluation methods and performance-oriented design techniques. Related literature describes patterns in software architecture: pattern-oriented software architecture (POSA), domain-specific software architecture (DSA), architecture frameworks such as Zackman and reference model for open distributed processing (RM-ODP), which defines five essential viewpoints for modelling systems architecture. The famous "4 + 1 view" model has a similar profile.

For a practicing architect, the studies in the literature are perhaps not of direct help as none defines the environment in which the architect works. The answers to the questions given below are important to all of those who are interested in this discipline:

- What problems do we, as architects, solve?
- What are our tools?
- What are our constraints?
- Where is the consistent theory to work in this environment?

We are yet to find answers to these questions as we are waiting for the discipline to mature. However, these are important questions that need to have at least approximate answers, which would help us in performing our tasks better. We need the knowledge that will help us in solving contemporary software-related issues, and proven examples that we can learn from. However, in software architecture, it is hard to find such good examples.

While learning to program in a specific programming language, we first learn various constructs and other syntactic elements of the language. We also solve several problems and practice by solving exercises before we attempt serious programming. Understanding these examples and solving exercises are crucial steps in learning any programming language.

Unfortunately, unlike programmers, most software architects make their first attempt in designing architecture in a live and mission-critical software system development project, whose success is vital to many stakeholders. This book provides good examples for budding architects so that they have an opportunity to master their craft before venturing into the actual project. This book also provides the required theory and best practices in the context of a given problem to ensure that the information stays

with the reader longer. We use case-study-based learning as a tool to master the art of designing software architecture.

ABOUT CASE STUDY METHODOLOGY

Learning among humans happens faster in an environment that facilitates learning by doing, learning from stories and learning from mistakes. Software engineering education is experiencing a shift towards non-conventional methods, backed by these cognitive science theories. The case-study-based approach is one such non-conventional method that creates an effective learning environment for all participants.

Case studies are known to be instruments that facilitate "thinking forward from first principles". Hence, case studies emphasize the use of tools, techniques and, most importantly, the concepts learnt. Case studies are more relevant to the kind of challenges professionals, both young and experienced, face. They bring in the concepts that help find solutions to problems and resolve complex situations in a practical way. The case studies presented in this book depict the true nature of the software industry in its present-day context, which is complex, evolving and challenging. No theoretical textbook can serve this purpose or be sensitive to an organization's culture and its specific problems.

When evaluated as a teaching instrument, case studies to a great extent simulate the real world. They serve the purpose of documenting the learning from real projects that other professionals can benefit from, without having to go through the actual experience. They provide an opportunity for analysis and application of skills to understand the context and to take appropriate decisions. This not only reinforces the theory behind the concepts but also leads to its better understanding. Further, the efficacy of a case study lies in its power to inculcate the ability to make sound decisions.

HOW DID EACH CASE STUDY COME UP?

Each case study presented in this book was written based on actual projects at Satyam Computer Services Limited during years 2005-2007. However, we believe that these case studies and lessons learnt are not specific to Satyam. They are applicable to any software services company. This is the reason why we refer to a fictitious company named "Fact-Tree" throughout this book. We will discuss more about Fact-Tree later.

Each case study in this book went through several phases. Initially, we worked with various teams within Satyam to identify the potential projects that can be converted into case studies. After getting a clear understanding of the teaching goals and the key lessons to be taught through the case studies, we evaluated and finalized the projects that could be used. We continued to work with the teams of these projects to acquire more details, study various project artefacts and to understand their complexity before coming up with various case studies to choose from.

These case studies were first published in *Satyam Technology Review* (*STR*), a quarterly publication for internal circulation within Satyam. Solutions to these case studies were invited from architects and designers. In addition, we conducted workshops at various locations of Satyam each month where these case studies were discussed and an effective environment for learning various aspects of software architecture was created. Participant feedback and solutions were studied to further improve the material, which culminated in the inception of this book. We have tried to package the story, problem, concepts and the background necessary to solve the problem. We hope this material will help all serious readers gain a perspective on each of the topics discussed.

ORGANIZATION OF THE BOOK

This book contains seven chapters. Except the first and the last chapters, each chapter is built on a case study. The first chapter introduces the reader to the discipline of software architecture. Each chapter, excluding the first and seventh, primarily focuses on one aspect of software architecture, namely, the need and role of software architecture, the process of refining and re-factoring software architecture, the process and techniques of evaluating it, the process of moving from software architecture to detailed design, and finally, the component-based development.

This is not to say that these are the only aspects or important topics of software architecture. These are the five aspects that we had an opportunity to work with from 2005 to 2007. A detailed description of each of the chapters is given below.

The first chapter makes an attempt to provide a primer that discusses all important concepts in software architecture that are needed in later chapters. Besides making the reader familiar with the basic terminology and some of the important concepts in software architecture, one of the major objectives of this chapter is to introduce software architecture as a discipline of problem solving. We apply Polya's approach to solve problems in the field of software architecture.

The second chapter tells the story of a health insurance application that failed in spite of having a very motivated and skilled team. This project recorded absolutely no issues when the process quality was assessed during the execution of the project. In this chapter, we try to bring out the importance of software architecture and the role it plays during the entire development life cycle. We distinguish between marketing architectures and technical architectures, which actually help the development team. Most projects do not have technical architectures and they stop with the creation of a marketing architecture.

The third chapter describes a call centre management software application whose architecture was provided by a highly reputed firm. It details a scenario where the proposed architecture, when implemented, failed to meet the performance, scalability and other non-functional requirements. The development team had to identify the bottlenecks in the architecture and refine and re-factor them. This case study deals with a sophisticated, large-scale and distributed system with stringent performance requirements. Under this development scenario, the system does not clear all performance tests. There are high chances that the system will not pass the client's acceptance test, and heavy penalty may be imposed. The only hope is to fine-tune or re-factor the architecture. However, the bigger challenge is to identify the right bottlenecks and improve the overall user experience. In this chapter, we learn how to achieve performance improvement by identifying and removing the problems in a given architecture.

The fourth chapter is about creating a search engine for the travel industry. In this case study, we come across an opportunity that can change the way people plan their travel. The development team is very cautious and does not want anything to go wrong. They follow some of the best practices like extensive domain study, employing the best minds on this project, and validating each step to ensure correctness of the proposed system. They come up with an architecture that seems to be perfect, and decide to evaluate it to ensure that they are in the right track. In this context, we address the following questions about software architecture evaluation: How to proceed? Which method to use? When to evaluate the architecture? This case study provides an opportunity to understand the need for and the process of architecture evaluation in a real-world context.

The fifth chapter is about moving from the description of the architecture to the detailed design. Here, we look at the design of "Mobile Trading System", a software application for mobile devices that helps its users manage their finances, including their stock market transactions, from anywhere at any time. The case discusses various best practices used by the software development team to come up with the design. It also reviews the usage of design methodologies and tools for developing design artefacts. This case study discusses in detail how different UML diagrams such as class, sequence and use case

help designers map user requirements to software design components and come up with a blueprint for the system to be developed.

The sixth chapter describes the development effort to bring more than 600 Web sites of a bank under one framework. This is the story of the Universal Bank, which has a large number of portals, both internal and external, each with its own idiosyncrasies. The case study deals with the huge exercise of integrating these portals and Web sites that were previously developed on multiple platforms and technologies with their own style and structure. They need to be re-engineered using a common set of components. The case study brings out the importance, approach and the process of component-based development and addresses many related meta-issues.

Chapters 2–6 are organized in a similar fashion. Each chapter begins with the description of larger issues and briefly gives the context of the case study. A high-level background, the necessary concepts and theory to address the relevant issues are given before the case study is presented. The readers are expected to read the case study (several times, if needed) until they are completely familiar with the environment, the issues stated and the context. Later, a postmortem of the case study is performed, wherein its salient features are discussed. We then move on to provide a detailed background, which is written based on the available literature on software architecture, the existing know-how and the best practices in the industry, to help solve the case study. Possible solutions are given to some of the important issues described, before we conclude the analysis with its best practices and key takeaways.

In the seventh (and the last) chapter, we discuss emerging trends in software architecture. We talk about the past, present and future of software architecture as a discipline. Reusability runs as a common theme in the content, and we consolidate discussion on this topic. We also describe service-oriented architecture (SOA) and software as a service (SaaS) as major trends in software architecture. We end the chapter with a discussion on the dimensions of future software architecture and important software architecture elements one needs to pay attention to.

ABOUT FACT-TREE

In this book, we attribute all case studies to a fictitious company called Fact-Tree Global Solutions. Fact-Tree is a 25,000-strong multinational company listed in various stock exchanges including the New York Stock Exchange. It has a global presence in 45 countries across six continents and has 18 development centres across the world.

Fact-Tree, like many global software services companies, has several vertical and horizontal business units. Each vertical business unit (VBU) takes care of specific markets such as banking, insurance, finance, healthcare, manufacturing and government. The horizontal business units (HBUs) provide competency in various packages, platforms and technologies. For example, enterprise application integration, business intelligence, Microsoft Technologies, Web services and middleware are some of the HBUs. The business and administration functions that are needed by the organization or by all the vertical and horizontal units were defined as strategic support units (SSUs). Thus, HR, accounts, network, security, auditing, among others, were support units.

Case studies are drawn from different VBUs or HBUs of Fact-Tree. We typically give necessary domain background within the context of these VBUs and HBUs in the case studies, and more pointers are provided to the reader to explore further.

Vasudeva Varma

Acknowledgements

Writing a book based on several complex real-world case studies was certainly not an easy task for me. This exciting journey started in 2005, and there are several people who helped me along the way.

Rajul Asthana, Senior Vice President and Global Head of Satyam Learning Centre, deserves all the credit for making this book a reality. He played the role of a mentor, cheer leader, critique and a friend. I believe that without his persistence and encouragement this book would not have fructified. I also appreciate his patience, as the final version of this book took much longer than expected.

The Satyam Learning World (SLW) team was very helpful during the first phase of this project. Aravind Venkat supported this activity and helped me in developing a framework for the chapters. He also edited earlier versions of the case studies and their solutions before publishing them in the Satyam Technology Review (STR). I thank Kavitha Thonangi for her help, enthusiasm and efficiency in execution; Aneetha Kanukolanu for meticulously reviewing initial drafts of the chapters and providing useful feedback with regard to language and presentation; Pragnya Seth for her help with the earlier versions of the case studies and all the learning officers at Hyderabad, Chennai, Pune and Bangalore for their support in the conduct of the workshops.

I thank Nagaraju Pappu, one the best architects I have known and a dear friend, for his interesting inputs on various topics relating to software architecture. Some ideas presented in the first and last chapters owe their origin to his mentorship. A major part of the sub-section "Criticality of Serviceability" in the last chapter is his contribution.

I am obliged to Bhudeb Chakravarti, a partner in this venture, for working with me closely and co-creating several case studies. Kirti Garg, my Ph.D. student, deserves special mention. I thank her for shouldering much responsibility, being ready to take up new challenges and always give her best. STP Vamsi played a good supporting role in the initial stages of this project, and Amit Sangroya helped in the final proof reading and creating the index. Venkata Bangaru inspired and provided all the required support.

Working with various project leaders, architects and software engineers has been a very enriching experience for me and my team. I have included all those who made significant contributions as co-authors of the respective chapters. I am indebted to Jayanti Vemulapati, Bidhu Mahapatra, Ram Gopal, Murali Mohan Nandipati, Prashanti Reddy, Shubhangi Bhagawat, Dilip Bhonde and Srikanth for sparing their time and effort in helping us to understand the complexity of their projects, providing required details and reviewing earlier versions of the case studies. I thank Madhav Negi, Jeet Chawre, Prakash Rao, Keshav Tripathi and Rajiv Ranjan for their help during the initial stages of this project. All the participants in the workshops helped us improve the material. Many solutions given by the participants were innovative and a few of them were actually better than the original solutions. I thank Satish Chandra, Prasad Kadari, Rajasekar Nanduri, Mahesh Khule, Narasimha Rao, Anand Kiran, Balaji Raghupati, Pankaj Gurumukhi, Krishna Koneru and Srinivas for submitting their solutions.

I thank Prof. Rajeev Sangal, Director of International Institute of Information Technology, Hyderabad, for his encouragement and giving me the needed flexibility to complete this work. I also thank my other colleagues at International Institute of Information Technology, Hyderabad, for their support.

I am grateful to Dr Len Bass of Software Engineering Institute (SEI), CMU, for writing the foreword for this book. I acknowledge all the help I received from the editorial team at Pearson Education. I specially thank Rajesh Shetty and M. R. Ramesh for providing me the necessary support.

Finally, I thank my wife, Sunanda, and my children, Mira and Rumi, for providing solid support while I was working on this book.

Vasudeva Varma

About the Author

Vasudeva Varma is a faculty member at International Institute of Information Technology, Hyderabad, since 2002. He is currently heading the Search and Information Extraction Lab (SIEL) at the Language Technologies Research Center and Software Engineering Research Lab (SERL).

Prior to his present affiliation at IIIT Hyderabad, he was President, MediaCognition India Pvt Ltd and Chief Architect at MediaCognition Inc. (Cupertino, CA). A former director of engineering and research at InfoDream Corporation, Santa Clara, CA, Dr. Varma has also worked with Citicorp and Muze Inc. at New York as senior consultant.

He received his Ph.D. from the Department of Computer and Information Sciences, University of Hyderabad, in 1996. With several publications in journals and conferences to his credit, he received the young scientist award from the Department of Science and Technology, Government of India, in 2004. He received a research faculty award from AOL Labs in 2007.

His areas of interests include search and information extraction, knowledge management and software engineering. He is also interested in experimenting with non-conventional methods for teaching software engineering in general, and case-study-based approach in particular.

Chapter 1

Software Architecture Primer

Software architecture is a relatively young discipline. There is as much confusion in it as there is excitement. In the literature one finds far too many perspectives, approaches, methodologies, frameworks, techniques and tricks. However, there is not much proven or established theorization of the subject. Software architecture is all about figuring out abstractions. Abstraction of abstractions takes time to mature.

In this chapter we introduce you to the discipline of software architecture. The aim is not to give a detailed description of all the concepts, methods and techniques of software architecture. There are very good books doing a good job of that. Our aim is to make you familiar with some important aspects of software architecture. In the subsequent chapters, each of which revolves around a case study, you will be able to relate the various challenges dealt with in them to the concepts described in this chapter.

Our treatment of software architecture concepts is not very formal. The software architecture discipline is introduced using a famous story—the browser war between Microsoft and Netscape. In the context of growing demands on software development, we have tried to discuss the role of software architecture. An informal introduction to software architecture follows. We then make you familiar with important terminology of software architecture before talking about more advanced concepts such as types, drivers, frameworks and styles of software architecture. The most important part of this chapter is connecting software architecture with problem-solving activities and establishing the fact that software architecture is indeed a problem-solving activity. We explore the link between two problem-solving disciplines using software architecture. To further explore the concepts discussed in the chapter, we provide links in the *Further Reading* section.

THE DISCIPLINE OF SOFTWARE ARCHITECTURE

Software architecture is among the computer science and software engineering terms that have the greatest number of definitions. This fact can be quickly verified if you visit the software architecture page on the Software Engineering Institute (SEI) Web site.[1] It is also one of the most used and abused terms of computer science. It means several things to several people. We would like to introduce you to the discipline of software architecture without getting into formal definitions.

In the past 40 years or so, since the term *software engineering* was coined during a NATO conference, the practice of software development has come a long way. We are able to build very complex, huge and smart systems using a large number of people. We have mastered many techniques, and developed tools and methods to aid in large-scale software development. For example, imagine the complexity of a system that makes your travel bookings based on your needs. It consults various air, hotel and car rental databases and finds out the best possible deal for you, and accepts a single credit card payment for various services from different service providers it is able to put together for your need. Please note that in today's software development environment, it takes less than 90 days to put together such complex systems.

However, it is a well-known fact that a significant number of software systems are not successful. In fact, a majority of the software systems are not built within the time and budget. What separates successful software systems from not-so-successful systems? An interesting story comes to mind.

The Rise and Fall of Netscape (or the Browser War 1994–1999)

Netscape Communications Corporation is one of the most talked about companies for its innovation and agility. Their flagship product, Netscape Navigator or simply 'Netscape', was very popular in the 1990s. Netscape was a proprietary Web browser with a huge user base and was dominant in the market. It has very good features and very attractive commercial licensing schemes combined with good attention and promotion from Internet Service Providers (ISPs) as well as the general media. Netscape was well positioned to take advantage of the consumer Internet revolution during the late 1990s.

Between April and December 1994, Netscape released its Web browser Navigator 1.0 in three platforms, and it was an instant success especially on the Windows platform. It came up with several innovations through the late 1990s including frames, cookies and, in its version 2.0, JavaScript. Netscape remained the market leader and was widely available on almost every operating system.

Microsoft realized the importance of having its own Web browser. It began development of Internet Explorer (IE) in October 1994 and shipped it in August 1995. The competition was clearly one sided. Microsoft could not make any significant dent in the market share of Netscape.

Meanwhile, Netscape was experimenting with a Web-based mini operating system (codenamed 'Constellation') that would enable users to access, edit and share their files independent of the hardware or operating system they used. Obviously, this was a big threat for Microsoft.

Netscape released its version 4.0 bundled with other groupware tools such as an e-mail client, messenger, collaborative threaded discussions, audio and video conferencing, group calendaring and scheduling. This package was called Communicator. Communicator 4.0 had around 3 million lines of code, and around 120 developers were involved in its release. This was a period of rapid growth for Netscape.

Microsoft saw an obvious threat to its operating system, and it began a full-fledged campaign to increase its share in the browser market. This period is famous as the 'Browser war' era. Microsoft could not catch up with Netscape until its version 3.0. With IE 4.0, it appeared like serious competition for Netscape, and with version 5.0 Microsoft surpassed Netscape. Finally, the war was won by Microsoft.

What is interesting for us is to re-look at the fact that Netscape bundled a lot of functionality with its version 4.0. This became spaghetti code because all the developers were just focused on adding a lot of functionality on a war footing. What was earlier fast and robust now became slow, buggy and crash-prone. Netscape abandoned its effort to release version 5.0 and instead threw open the code base of 5.0, for which there were not any takers. Netscape decided to start all over again with Communicator 6.0 with a new code base and later released version 7.0 before its disappearance. Many industry experts feel that Microsoft is not solely responsible for the fall of Netscape. In fact, from the technical standpoint, it scripted its own end. Microsoft only helped it to get there faster.

The Netscape story is a classical example to showcase the result of attempting to scale up without planning for such growth. It tells us that ad hoc software development cannot win in the long run. Most importantly, software systems that are developed without a solid and open architecture and design cannot scale up to meet the growing demands of the environment.

The Growing Demands on Software Development

If you observe the trends in the software industry, especially in the services sector, it iwill be very clear that demands on software development teams are growing very fast. Customers expect better, faster, larger and cheaper systems all the time. We cannot achieve all these qualities by accident. There has to be something more than adding more people to the project, and that has to do with having a good blueprint of the proposed system and its possible growth before actually building it. The following are some observations on building large-scale software systems:

- A major challenge of software system development is to figure out appropriate interfaces between the domain components and the software components. While developing a software system in a specific domain, most of the effort goes into modelling the domain and developing components that are specific to that domain. These domain-specific components need to be integrated into the software framework, which usually includes components such as Web servers, application servers and databases. These interfaces between domain engineering and software engineering are becoming very complex because of the distributed nature of the platforms as well as development methodologies.

- Non-functional requirements such as performance and security cannot be planned for *after* the functional requirements are achieved. This observation is in contrast with general perceptions and common practice in the industry. One needs to plan for all these so called non-functional requirements right from the beginning along with functional requirements.
- Complexity and modularity go hand in hand while developing software systems and are often misunderstood. The complexity of software systems is growing continuously, and one of the ways of reducing or managing complexity is to make use of the modularity principle. If this understanding is missing, modularity for the sake of modularity increases the complexity.
- The ability to re-use effectively and to the maximum extent possible holds the key to faster, cheaper, better and bigger systems. If the components are built in such a way that they remain useful even after the current need, then they have the potential to contribute to the bottom-line of the company in a more definite manner. It needs to be emphasized that re-usability needs a special organizational frame of mind and support from all the stakeholders.
- Requirements engineering has become less of mystery, and clients have become more knowledgeable. The gap between software requirements engineering and the system that gets finally delivered is increasing and is becoming obvious. Hence, meeting the functional and non-functional requirements has become a challenge.
- Similarly, the relation between the proposed software system and the rest of the phases in the software development life cycle (such as design, coding, testing and maintenance, if you are considering waterfall model) needs to be brought into the focus of the review at regular intervals. Otherwise, there is a potential loss of control as the software development progresses.
- The biggest and most important challenge of large-scale software development is establishing proper communication channels between all the stakeholders—customers, sponsors, developers, management, users and any other parties. This can be considered as the single most important success factor.

Most of the above issues can be addressed very effectively by having a proper blueprint of the proposed software system before it is actually built. The need for such a blue print can never be over-emphasized. This blueprint gives us guidance, sets us right goals, tells us if we are going wrong somewhere, reminds us of priorities, manages the complexity of the system for us and most importantly works as a communication tool. We refer to this blueprint as *software architecture*.

WHAT IS SOFTWARE ARCHITECTURE?

There are several definitions and descriptions for *software architecture*. You will find them in most of the pointers we have provided in the *Further Reading* section as well as in the *References* section. We do not wish to repeat those definitions. Instead of offering yet another definition for *software architecture*, we like to introduce you to the concept in a non-formal way. In most simple terms, *software architecture* has to do with two things:

- Doing the right things
- Doing the things right

Doing the right things involves understanding the market dynamics and figuring out what works well and what doesn't. This means the ability to understand component and platform suppliers, system integrators and all third-party stakeholders including retailers, providers and finally consumers.

In the value chain, the equations change very fast in a dynamic market. We cannot hang on to old knowledge or assumptions. Figuring out what the right things to do are is the most important function of an architect.

Similarly, doing the things right involves understanding the solution space and the technology to create such a solution. Most often, this know-how comes from the lessons learnt from the past and the experience gained. Typically, the architect only guides the implementation, and most of actual work is done by the designers and engineers. The architect provides them with guidelines in the form of high-level designs that are presented as many viewpoints.

Creating effective software architecture involves a proper understanding of the problem and creation of an appropriate solution, that is, working with the problem space as well as the solution space. The problem space includes understanding customer needs as well as the domain, and the solution space includes know-how related to the solution and technologies to make it possible to build the solution. The role of an architect is to do justice to all the stakeholders while working independently and to come up with an effective solution.

Muller (2003) puts this very nicely by breaking software architecture activities into three parts:

- Understanding *Why*
- Describing *What*
- Guiding *How*.

The market dynamics, convergence, integration and diversity are the focus of study in the Understanding *Why* phase. Creating configurable platforms and frameworks, and family architectures with various viewpoints is the function of Describing *What* and working with the designers and engineers to realize the architecture by providing guidelines and various viewpoints of the architecture forms goes into the phase of Guiding *How*.

THE IMPORTANCE OF SOFTWARE ARCHITECTURE

For naive software developers, it is not easy to understand the importance of software architecture. Even most software development managers do not really appreciate the importance of software architecture. Most of the time, they only pay lip service to it. In this section, we attempt to answer the question that often lingers in people's minds but for some reason never gets asked: Why focus on architecture especially when there are time and budget pressures? We will re-visit this question in the next chapter when we discuss the need for software architecture with respect to a case study called *Assure-Health*. Here we attempt to give only a gist of some of the important benefits of software architecture.

Clements and Northrop (1996) suggested that there are three fundamental reasons why software architecture is important:

1. Tool for communication. A well documented software architecture helps in forming various communication channels among all the stakeholders of the system. This is a high-level representation or abstraction of the system under consideration. The software architecture is a common talking point among developers, sponsors, managers, customers and users as it helps in capturing various viewpoints of the problems space as well as the solution space.
2. Early design decisions. It is well known that the earlier we make right design decisions in the software development life cycle, the higher the returns are. A software architecture helps in making the right decisions early before we get into the detailed design and actual implementation phases.

3. Transferable abstraction of a system. A software architecture can be considered as a model or an abstraction of the actual system. This abstraction is typically at a platform and technology level, which implies that it can be re-targeted towards a new platform or some other system with almost similar requirements.

Besides these reasons mentioned by Clements and Northrop, there are some other important benefits of software architecture:

- *Abstraction*. Software architecture provides a model of components and connections along with their behaviour, which in turn results in improved communication between stakeholders. The model is the abstraction of the actual proposed software system. With the help of this abstraction, we can achieve a better understanding of the proposed software system. This understanding may be achieved by reducing the system to its essence.
- *Architectural level of analysis.* Once we have an architecture in place, it is possible to determine the degree to which a system satisfies its requirements. That means we actually perform a lot of analysis at the level of architecture without really spending our energies in creating a very detailed design or coding it on a particular platform. This kind of analysis is performed at the platform-independent level and always keeping the big picture in mind, no matter how complex the proposed system is. This activity helps us to analyse the completeness, safety, component interaction, performance and many other important aspects of the software system. We can also perform tasks such as localizing faults, regression testing, detecting drift, re-use and reverse engineering at this level.

In addition to these important aspects, software architecture provides many other side benefits, such as the following:

- *Longevity.* A better designed system with a solid architecture will be able to adapt to the changing environment and hence it has a long life. This is evident from our Netscape story. Many developers leave the teams, but architectures that are created by these teams have a longer tenure at the organizations.
- *Competition advantage.* In order to withstand the market dynamics, software systems need to be designed in such a way that any set of features can be added while the system is being used by the customer community. Only those enterprises with this capability will be able to maintain their advantage over the competition. If the turn-around time for adding a new functionality is too high, there is a significant risk of losing the customer base; on the other hand, if you are able to design a system that can offer a new functionality within a very small time period, the chances for success will be increased multifold.
- *Stability.* A well designed architecture provides stability by ensuring minimum fundamental reworking as the system is extended to provide additional functionality. This additional functionality may be achieved over multiple releases, and at the same time the development team can hold on to the foundation instead of working on something that is constantly changing. Obviously, this results in minimizing the costs.
- *Ensuring architecture attributes.* Architectural attributes (such as adaptability, security, usability, testability, maintainability, availability, reliability, recoverability, performance and safety) impact the design and development of various parts of software. The development team needs to constantly keep evaluating the system with respect to these architectural attributes to determine if the system meets the desired goals.

- *Patterns.* Patterns capture known solutions to recurring problems in ways that enable you to apply this knowledge in new situations. While designing the software architecture, it is pragmatic to look for documented patterns and make use of the ones that are applicable. If you find some new problem pattern and the corresponding solution, document it so that this knowledge can be re-used later.
- *Design plan.* Software architecture provides a design plan or a blueprint of the system so that the complexity of the system can be managed by abstracting out the details. Software architecture captures the early design decisions and provides an organizational structure to the development team. It also specifies the constraints on the lower level design and implementation.
- *Verification of requirements.* The architectural viewpoints can be used in verification of the system requirements, be they functional or non-functional. The architecture exposes missing, incomplete, invalid and inconsistent requirements.
- *Making modifications.* Making changes to software systems is inevitable—in fact this is the most expensive item in the software life cycle. The need to make modifications can be because the environment is changing or the technology is changing or both of them are changing. Reasoning at an architectural level can provide the insight necessary to make decisions and plans related to change. More fundamentally, however, the architecture partitions possible changes into three categories: local, non-local and architectural.
- *Aids in testing.* Testers need to understand the overall system as well as all its sub-systems or components along with their interfaces to perform white box testing. The software architecture views also capture all key process interactions and the corresponding performance information that can be verified during the testing procedures.
- *Project management.* Project managers make use of software architecture throughout the project life cycle. They use it to structure the development teams and identify the work to be performed by each team. Project managers also use the architecture information to identify the interface elements to be negotiated among the development teams. The software architecture can aid the project managers to make some of the most critical decisions such as reducing the functionality or moving the functionality to a later release.
- *Training.* Various architectural views and viewpoints along with their high-level description of how the various sub-systems interact with each other to carry out the required behaviour often provide a high-level introduction to the system for new project members. Any new member who has been assigned to an existing software project can benefit from well documented software architecture to bring her or him up to speed. Even customers, managers, testers and operations people, not just the developers, can benefit from the architecture documentation.

ROLE OF A SOFTWARE ARCHITECT

The role of a software architect is quite varied. An architect typically performs several roles and is involved in several tasks which include the following:

- Envisioning the software system
- Being chief strategist, advisor and mentor to the development team
- Mimicking the role of a customer or an actual user

His or her primary task is that of a problem solver. That is the reason why we give a lot of emphasis on problem-solving activities in this book. Problem solving is required in the problem space, where the architect focuses on requirements understanding and specification, and in the solution space, where the activities include the following:

- Coming up with various architecture viewpoints and design
- Making sure that the design is realized correctly during the implementation stage
- Making adjustments to the design to ease implementation challenges
- Communicating it well among all the stakeholders of the system

The deliverables of an architect are typically various kinds of documents and communication material. These may include various reports and studies, specification documents, design documents, guidelines and best practices to help the development team in realizing the design through correct implementation.

An architect needs to be involved in a lot of communication activities—most importantly, listening to various stakeholders. You can find a good architect always engaged in activities such as talking, discussing, brainstorming, presenting, meeting, reading, writing, consolidating, reviewing, browsing, thinking, analysing, teaching and mentoring. An architect is expected to help the project manager with work breakdown analysis, scheduling the tasks, risk analysis and mitigation activities. An architect invests a lot of time in talking to customers, partners and various third party vendors.

One of the major challenges of an architect is to stay ahead of all others in the learning curve and have the ability to predict common pitfalls. In other words, an architect is a person who is supposed to have all the answers. That means he or she needs to invest a lot of time in order to stay ahead of the rest by reading about the state of the art technology and business developments, attending and contributing in conferences, and continuing to have hands-on experience so that pilot tasks, testing, integration and verification can be done before involving other members in the development team.

There are several professional certification courses and formal training programmes conducted by high-end training centres and institutions such as SEI (Software Engineering Institute, Carnegie Mellon University) and Bredemeyer Consulting. In addition most of the technology platforms such as Net, J2EE, SAP, Symbian have their own training and certification programmes targeting designers and architects.

An architect should be able to see the big picture of the proposed system and the environment in which it will be operating at all times. If he or she loses sight of this big picture, several problems can creep in. In fact, this ability distinguishes an architect from others in the development, where the focus typically is on one aspect of system development like hardware, database design, functionality or project management.

This is the reason why it is highly recommended that the software architect be involved throughout the software development life cycle. Initially, at least till the implementation phase begins, the involvement and number of activities of the architect will be great—right through the requirements engineering phase, architecture and design phase. After the design phase, an architect is still required to make sure that her or his design is realized in the correct manner. That is when he or she can make minor modifications to the design if it helps in making implementation easier. Even during the integration and testing phase, the role of an architect becomes significant. During the maintenance phase, the architect can help in extending existing systems with newer functionality and in fixing errors.

Currently, in many software product companies, the role of a software architect is very well defined, but unfortunately the same cannot be said in most software services companies. There is also confusion about this role as there are several titles and designation labels used for the role of a software architect: Solution Architect, Systems Architect, Technical Architect, Senior Consultant, J2EE Architect, etc., which does not really capture the semantics. You can find many solution architects actually performing other design tasks such as such as systems architecture or domain modelling functions. This causes further confusion among engineers who aspire to become software architects.

The role of an architect in the services industry should be defined very clearly in the organizational hierarchy, and there should be a technical career path clearly defined for software engineers to become software designers and then software architects. This technical track should be recognized and compensated on par with the managerial track. Once this is in place, there will be a motivation for software engineers to choose this option, which is otherwise largely missing because the only career path software engineers see is to become team leaders and project managers after gaining a few years of experience, which may not involve a technical role at all.

SOME IMPORTANT TERMS USED IN SOFTWARE ARCHITECTURE

In this section we introduce some terminology used in the software architecture literature. We will not be making an attempt to go in depth into the subject, but this should help you to connect with the rest of the material in a better way in case you are not already familiar with the terms.

Component

A component is a basic building block of the software architecture and is an encapsulated building block of the entire software system which provides interfaces to make its services available. A component that does not implement all the elements of its interfaces is called an abstract component. A concrete component has all its interfaces implemented and can be instantiated. This distinction is similar to the one between abstract and concrete classes.

Another perspective is that software components conform to a component model and can be independently deployed and composed without making modifications to the rest of the system. However, one needs to follow a composition standard.

Some of the related terms are described in the following.

Component-Based Development (CBD)

Component-based development refers to a situation where all of or a large part of the software development is done using already available and/or a new set of components.

Component-Based Software Engineering (CBSE)

CBSE is a sub-discipline of software engineering that is primarily concerned with developing software with already available software components. The focus of CBSE is on the ability to re-use the available components in other applications and the ability to maintain and customize these components to produce new functionality or features in these new applications.

In the literature you may find that the terms *CBD* and *CBSE* are used in almost in the same sense.

Commercial Off-the-Shelf (COTS) Systems

COTS systems include software components at the binary level or at the source code level, which can be purchased from software vendors or obtained from an open source.

Relation

A relationship is a static or dynamic connection between components. Static connections are pre-defined and hard coded, whereas dynamic connections are established during the run time, dealing with the interaction between components.

Model

A model is a representation of the entire system that helps in understanding and documenting one or more aspects of it. A model can be built for either a problem or a solution.

Framework

A framework defines the architecture for a family of systems and their sub-systems and provides basic building blocks to instantiate them. In an object-oriented paradigm, it typically means that there are a set of classes, both concrete and abstract, which can be sub-classed or composed. These classes define the framework.

Architectural View

An architectural view is a representation of a system or its sub-systems from a specific perspective of that system. For example, a proposed system, from the hardware infrastructure perspective, can be represented as a *deployment view*, and the same system from the perspective of arrangement of all sub-systems and corresponding modules can be represented as a *logical view*. We will discuss these views further in several occasions subsequently in this book.

Viewpoint

For a given architectural view, a template used for describing that view is called a viewpoint. Typically a viewpoint tells us how to create and use an architectural view.

SOFTWARE ARCHITECTURE CONCEPTS

In this section, we introduce you to some of the important concepts of software architecture. This section will provide you sufficient background to appreciate rest of the book in a better manner. However, each of the topics discussed in this section deserves in-depth study, and if you are interested we advise you to go through the pointers in the *Further Reading* section of this chapter.

We discuss the following facets of software architecture in the subsequent sub-sections:

- Types of architectures
- Architectural drivers
- Architecture frameworks
- Architectural styles or architecture patterns

Types of Architectures

The term *architecture* can be interpreted differently based on what we are focusing on. In the literature of software architecture, the following types of architectures are discussed:

- Enterprise architecture
- Systems architecture
- Software architecture
- Technical architecture.

Each of these terms is applicable at different layers of the model. Figure 1.1 illustrates this concept.

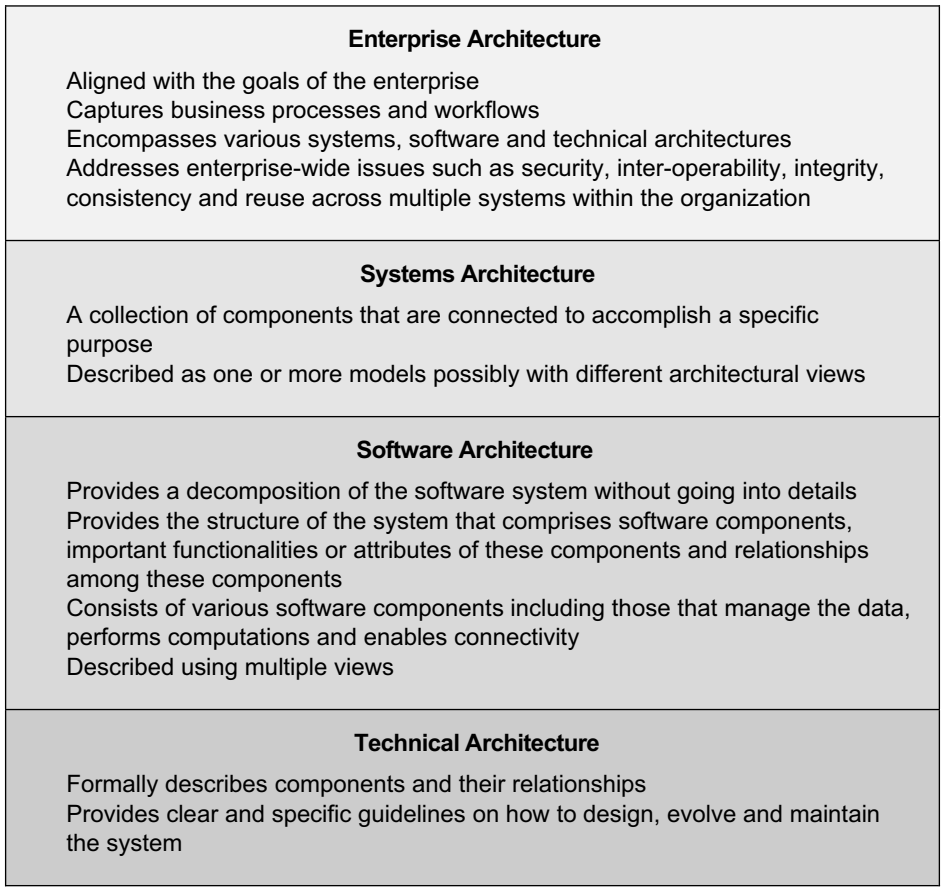


Figure 1.1 Types of architectures

Enterprise Architecture

The main goal of enterprise architecture is to capture the business processes and workflows so that the big picture is clear for all stakeholders and the scope of the software and technical aspects are very clear. It is not necessary to build IT or software solutions to achieve the organizational goals—the overall solution may consist of people, processes and software (along with hardware).

Enterprise architecture encompasses all other types of architectures and makes sure that all of them are aligned with the overall goals and objectives of the enterprise. A typical example of an enterprise-level objective is to have the flexibility to build, buy or outsource IT solution components. Re-use across product families is another enterprise-level objective.

When we talk about software (or application) architecture at the enterprise level, we may only focus on the product line or product family specification rather than giving details of individual components or specific products. In that sense, enterprise architecture provides meta-architecture—that is, guidelines to aid in structuring the systems rather than the structure itself. Similarly, when we talk about technical architecture at the enterprise level, we need to emphasize the common platform on which various systems are being built. This includes frameworks, components, shared infrastructure and tools. If there is a planned

re-use across a product family and from the common platform point of view, that needs to be specified at this level.

Systems architecture

The term *systems architecture* is somewhat confusing because if we look at any type of architecture, it is a 'system' by itself. The entire enterprise is a system, and so are the software and technical systems. However, we would like to distinguish systems architecture from the rest mainly because of the following reasons.

- The systems architecture defines the scope of the system we are trying to build.
- It unambiguously defines the purpose for which the system is being built.
- It helps us understand how this system interacts with the rest of the environment.
- It helps us understand how the individual sub-systems interact with each other and contribute to the purpose of the system. In other words, it helps us understand the overall system before we attempt to develop or build a sub-component of this larger system.

The whole is greater than the sum of the parts—that is, the system has properties beyond those of its parts. In order to appreciate the benefits of systems architecture, we shall look at a few definitions of *system*:

> The IEEE Std. 610.12-1990 specification defines it as '… a collection of components organized to accomplish a specific function or set of functions'.
>
> The UML 1.3 specification defines the system as 'a collection of connected units that are organized to accomplish a specific purpose. A system can be described by one or more models, possibly from different viewpoints'.

These definitions make a case for systems architecture and its role in comparison with other types of architectures. The earlier statement about the whole being more than the sum of its parts emphasizes that not only does the system perform a unique function, but it has unique characteristics or qualities that are inherent in the system and not just the parts (Rechtin, 1991).

Most of these definitions and discussions are influenced by a discipline called *Systems Thinking*, which has its roots in work by Buckminster Fuller and Russell Ackoff, among others. Many intellectual leaders such as Peter Checkland and Peter Senge (author of *The Fifth Discipline*) developed the ideas of systems thinking. Earlier we have mentioned the work by Eberhardt Rechtin (1991). His book *Systems Architecting: Creating and Building Complex Systems* is one of the most influential works in this domain. We discuss systems thinking in greater detail subsequently in this chapter.

Software Architecture

The software architecture is the set of software components or sub-systems, the properties of these components (or sub-systems), and the relationships and interactions among them that define the structure of the software system. Bass *et al.* (2003) define *software architecture* as

> Structure or structures of the system which comprise of software components and *extremely visible properties* of these components and relationships among them.

By 'externally visible' properties, they are referring to assumptions other components can make of a component, such as the services it provides, performance characteristics, fault handling and shared resource usage.

Software architecture typically contains three kinds of views:

Conceptual architecture. The purpose of conceptual architecture is to provide direct attention at an appropriate decomposition of the system without delving into details. The main purpose of this view is to communicate the architecture to non-technical stakeholders such as management, investors, marketing or sales personnel, and sometimes to the customers or users. The conceptual architecture consists of major components or sub-systems and an informal component specification. It does not show or describe any interfaces among the components.

Logical architecture. Logical architecture provides more details about each of the components including their interface specifications and collaborative diagrams that show interactions between various components. Logical architecture adds enough detail and precision to such an extent where component developers and component users can work independent of each other. This view provides detailed architecture including the interfaces between components and documents the discussions, decisions and rationale behind the decisions.

Execution architecture. Execution architecture typically provides the process view and the deployment view. The process view shows the mapping of components onto the processes of the physical or actual system. The deployment view shows the mapping of the physical components in the executing system onto the nodes of the physical system.

Figure 1.2 shows the most salient features of all these views.

Each of the software architecture views has a structural dimension as well as a behavioural dimension. These dimensions enhance our understanding of the architecture and help us in addressing these two aspects (that is structural and behavioural aspects) separately.

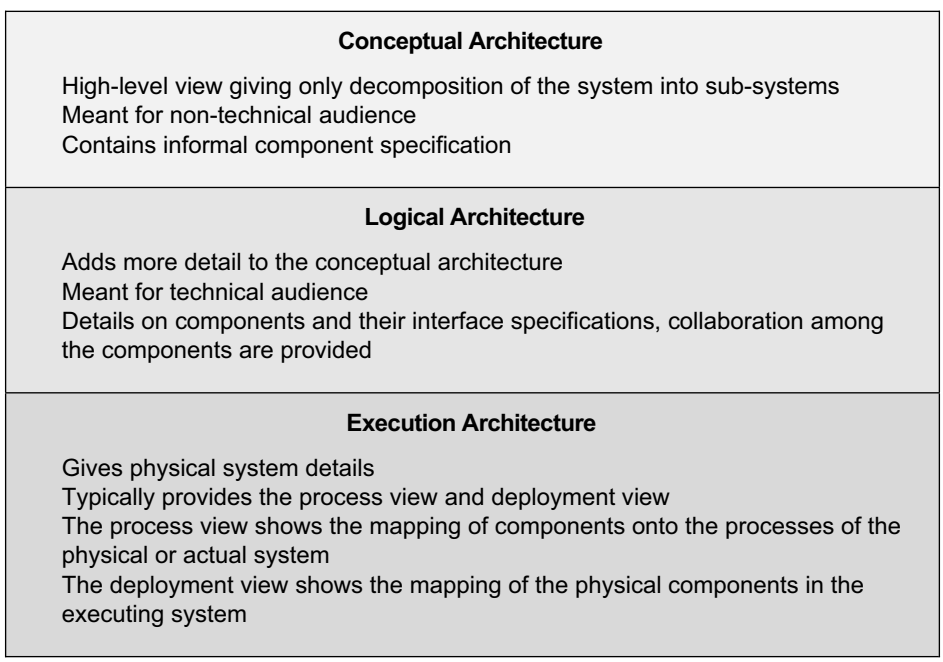


Figure 1.2 Software architecture views

The structural view is central to decomposing the system into various components, their (visible) properties and their relationships. From the engineering point of view, this decomposition is very critical, and we are interested in making sure that we reduce the complexity of the system and understand the system using the structural view.

The behavioural view provides the answer to the most important question that comes after we decompose the system into a structure. Given the components and their interfaces, how does the system work? This is the understanding provided by the behaviour view with the help of collaboration diagrams and sequence diagrams, which we discuss in more detail subsequently.

Technical Architecture

Technical architecture, also known as IT architecture, can be described as a formal description of an information technology (IT) system, organized in a way that supports reasoning about the structural properties of the system. Technical architecture formally describes components and their relationships. It also provides clear and specific guidelines on how to design, evolve and maintain the system.

The US army's joint technical architecture (JTA) defines IT architecture as follows:

> It is the minimal set of rules governing the arrangement, interaction, and interdependence of the parts or elements that together may be used to form an information system. Its purpose is to ensure that a conformant system satisfies a specific set of requirements

Technical architecture defines the components or building blocks that make up an overall information system, and provides a plan from which products can be procured, and systems developed, that will work together to implement the overall system. The Open Group Architectural Framework (TOGAF) for the Open Group's IT Architecture Development Method is greatly focused on technical architecture. We talk about TOGAF as one of the software architecture frameworks subsequently in this chapter.

In addition to enterprise, system, software and technical architecture, we hear about other types of architectures. We outline some of those which occur very commonly: data architecture, reference architecture and product line architecture.

Data Architecture

Data architecture refers to specification of various databases (how many databases and what kind of databases) along with their logical and physical database design, allocation of data to these servers, backup/archival and data replication strategy, and the strategy and design of a data warehouse.

Data architecture is supposed to answer all questions related to structuring, storing and handling of data for a given system.

Reference Architecture

Reference architecture is an architecture defined for a particular application domain such as healthcare, insurance, banking or telecommunications. Reference architectures describe a set of high-level components along with their interactions in a given application domain.

These components need to be purposely at a high level or general so that they can be customized for a large number of applications in a given domain. Reference architectures provide an excellent framework for the development of several applications, saving a lot of time and effort for the software architects. Most of the savings come from not having to re-discover the common elements.

Product Line Architecture

Product line architecture is used to define a set of products or a family of products in such a way that these products share many common components as well as design and implementation information among various teams developing these products.

Product line architecture helps in making all the members of a product family have not just a uniform look and feel; it also provides the consistency in the way they are designed, developed, tested and supported. There is a very high level of re-use if product line architectures are employed, which saves a lot of money for the organization.

Architectural Drivers

Architectural drivers are the most important requirements, be they functional, non-functional or quality related. Architectural drivers are this combination of most influential requirements that shape the software architecture for a base set of applications.

For example, the security aspects of the electronic fund transfer functionality of an online baking system development may be an architectural driver whereas the ability to view past one year's transaction data may not be an architectural driver for the same system.

In fact, during the process of designing the software architecture, identifying architectural drivers is the most critical step. After exploring the problem space very thoroughly, the architect identifies the functional, non-functional and quality attributes of the system based on her or his past experience, making use of known guidelines and principles as well as all the support she or he gets from the methodology that she or he subscribes to. This exercise is done both at a high level as well as very concretely at a fine-grain and detailed level. Once these attributes are discovered and documented, the most important of them are identified as architectural drivers. All the key stakeholders need to have an agreement on the architectural drivers. These architectural drivers determine the kind of architectural style we choose (we discuss architectural styles later in this chapter). The architectural style chosen needs to satisfy all the architectural drivers.

Software Architecture Frameworks

Wikipedia defines an architecture framework thus: 'An architecture framework is a specification of how to organize and present an [...] architecture'.

Architectural framework is a thinking tool to capture the information about an organization and to understand how things work so that one can build the information systems efficiently.

Any software architecture framework is expected to provide the following to aid in the process of designing software architecture:

- An ability to find out all the important dependences by critically looking at the models.
- Support for decision making in a dynamic business environment as the architecture brings together the business aspects as well as the technical aspects of the system.
- An ability to trace the impact of any changes at the organizational level or at the business level on the systems.

Use of an architecture framework increases the chance of success for your architecture initiative. From the many frameworks available, choose the best one for you, and augment it with components from other frameworks.

There are several software architecture frameworks that are popular. We shall discuss the following frameworks:

- The Zachman framework
- The Open Group Architecture Framework (TOGAF)
- Reference Model for Open Distributed Processing (RM-ODP)
  - 4+1 Views
  - Rational Unified Process (RUP)

There are some other frameworks such as POSIX 1003.23, Federal Enterprise Architecture Framework (FEAF) and Garner Enterprise Architecture Framework (GEAF). Though we will not be discussing all these frameworks in this book, we need to be aware of the fact that no framework is complete. We need to choose the one that is the best for the organization. At the same time we should be ready to augment this framework with the components from other frameworks. In reality, organizations end up defining their own framework with ideas, methodologies, philosophies, tools, components and processes from a variety of sources. However, in an enterprise architecture initiative, it is recommended to describe the framework and process that are going to be used so that all stakeholders in the project understand the approach. It is better to keep this description simple, short and easy to read if you actually want people to read it.

The Zachman Framework

John Zachman conceived a framework for enterprise architecture in 1980s at IBM which was later made public and became well known as the Zachman framework. It allows a highly structured and formal way of defining an enterprise's systems architecture.

This framework, presented in the form of a table (Table 1.1), uses a grid model based around six basic questions (What, How, Where, Who, When and Why), each representing a column. The rows represent the stakeholder groups to whom these questions are asked. The five nominated stakeholder groups are Planner, Owner, Designer, Builder and Sub-contractor. In other words, the vertical axis represents multiple dimensions of the overall architecture; the horizontal axis classifies various artefacts produced based on the interests of particular stakeholder groups representing their perspectives. If we can fill in each cell in the grid, we will have a holistic view of the enterprise. There is alignment of cells with each other. Each cell must be aligned with the cells that are next to it both horizontally and vertically, that is, each cell is aligned to the cells left, right, above and below. There is no alignment between diagonal cells.

It is very difficult to imagine any aspect missing from this framework. Any artefact is included in some way in the framework. That is the reason why the Zachman framework is known to be the most comprehensive framework. At the same time, it is so comprehensive that it is takes an enormous amount of effort for organizations to populate all the cells. The Zachman framework generates a lot of documentation due to its completeness. Most of these details are hard to consume and digest, and sometimes the utility of these details is also questionable. Hence it is recommended that only the relevant cells be populated.

The initial three dimensions of what, how and where (that is, data, function and network aspects) were part of the original framework. Later in 1993 the other dimensions of who, when and why were added (people, time and motivation). These dimensions are not ordered and are completely arbitrary.

There is one major missing link in the Zachman framework. It does not cover the actual process of designing architecture. But it is claimed that any architecture process can be used *with* the Zachman framework. Hence it is method-neutral and can be used as a complimentary framework along with other frameworks. This framework per se does not include governance and communication aspects either, although John Zachman has written about these and many other architectural aspects.

The Zachman framework is considered as the mother of all frameworks or a reference framework against which other frameworks can be mapped or positioned. The Zachman framework has become part

Layer	View	Data (What)	Function (How)	Network (Where)	People (Who)	Time (When)	Motivation (Why)
1	Scope/ contextual Planner	List of things important to the business	List of processes the business performs	List of locations in which the business operates	List of organizations important to the business	List of events significant to the business	List of business goals/ strategies
2	Business model/ conceptual Owner	Semantic or entity–relationship model	Business process model	Business logistics system	Workflow model	Master schedule	Business plan
3	System model/ logical Designer	Logical data model	Application architecture	Distributed system architecture	Human interface Architecture	Processing structure	Business rule model
4	Technology model/ physical Builder	Physical data model	System design	Technology architecture	Presentation architecture	Control structure	Rule design
5	Detailed representations (out-of-context) Sub-contractor	Data definition	Program	Network rchitecture	Security architecture	Timing definition	Rule specification
6	Functioning enterprise	Data	Function	Network	Organization	Schedule	Strategy

Table 1.1 Zachman table— Zachman framework for information system architecture

of any enterprise or system architecture review exercise within IT departments. However, its popularity has not reached the developer and user communities.

The Open Group Architecture Framework (TOGAF[2])

The Open Group, consisting of Cap Gemini, HP, NEC, EDS and IBM, developed a framework to provide an approach for designing, planning, implementing and governance of an enterprise information architecture called the Open Group Architecture Framework (TOGAF). TOGAF is very actively updated and matured compared with other architecture frameworks. It has about 55–60 members who are responsible for the development of TOGAF. It is currently in version 8.1.1. The versions up to 7.0

are known as technical versions, and from version 8 onwards as enterprise versions because TOGAF is focusing on enterprise architecture.

TOGAF provides a step-by-step method of how to understand requirements and of how to build, maintain and implement enterprise architecture. The graphic of TOGAF (see Figure 1.3) is dynamic unlike the rectangular grid of cells of the Zachman. It consists of a set of circles showing the progression cycle through various phases.

According to TOGAF, architecture is typically modelled at four levels or domains: Business, Application, Data and Technology. Unlike the Zachman framework, TOGAF is a process-driven framework. The central process is called the Architecture Development Method (ADM). Using this process, architects can develop the different aspects of enterprise architecture to meet the business and information technology needs of an organization. The ADM should be adapted to each organization's needs and is then used in architecture planning activities.

The ADM process depicted in Figure 1.3 (from the Open Group Web site).

A brief description of each of the main phases follows:

Phase A: Architecture Vision

First, one needs to decide what to do in this round of development, which includes determining the scope of the project and the stakeholders involved and obtaining the necessary approvals and the necessary support. The baseline (current architecture) and the target architecture have to be determined at a high level and documented in this phase.

Phase B: Business Architecture

Here one focuses on determining the business aspects in depth, which requires extensive modelling of the present and target architectures. Gap analysis is performed to determine what needs to be done to bridge the gap between the current system and the target system.

Phase C: Information System Architecture

Here the data and application architectures are analysed in depth. The system is broken down into building blocks that may or may not yet exist.

Phase D: Technology Architecture

The business and information architectures that were created in phases B and C are implemented in phase D. It involves breaking down the main functionality into re-usable building blocks and describing these blocks with respect to the foundation architecture. Technology architecture has sub-phases that are pretty much similar to the main phases as can be seen in the figure.

Phase E: Opportunities and Solutions

In this phase, one needs to determine which building blocks can be re-used, which ones must be replaced and which ones must be provided (a build or buy decision is also involved). TOGAF documentation says, 'The most successful strategy for the Opportunities and Solutions phase is to focus on projects that will deliver short-term payoffs and so create an impetus for proceeding with longer-term projects'.

Phase F: Migration Planning

In this phase decisions will be made about the order in which the implementation of the new system will be done.

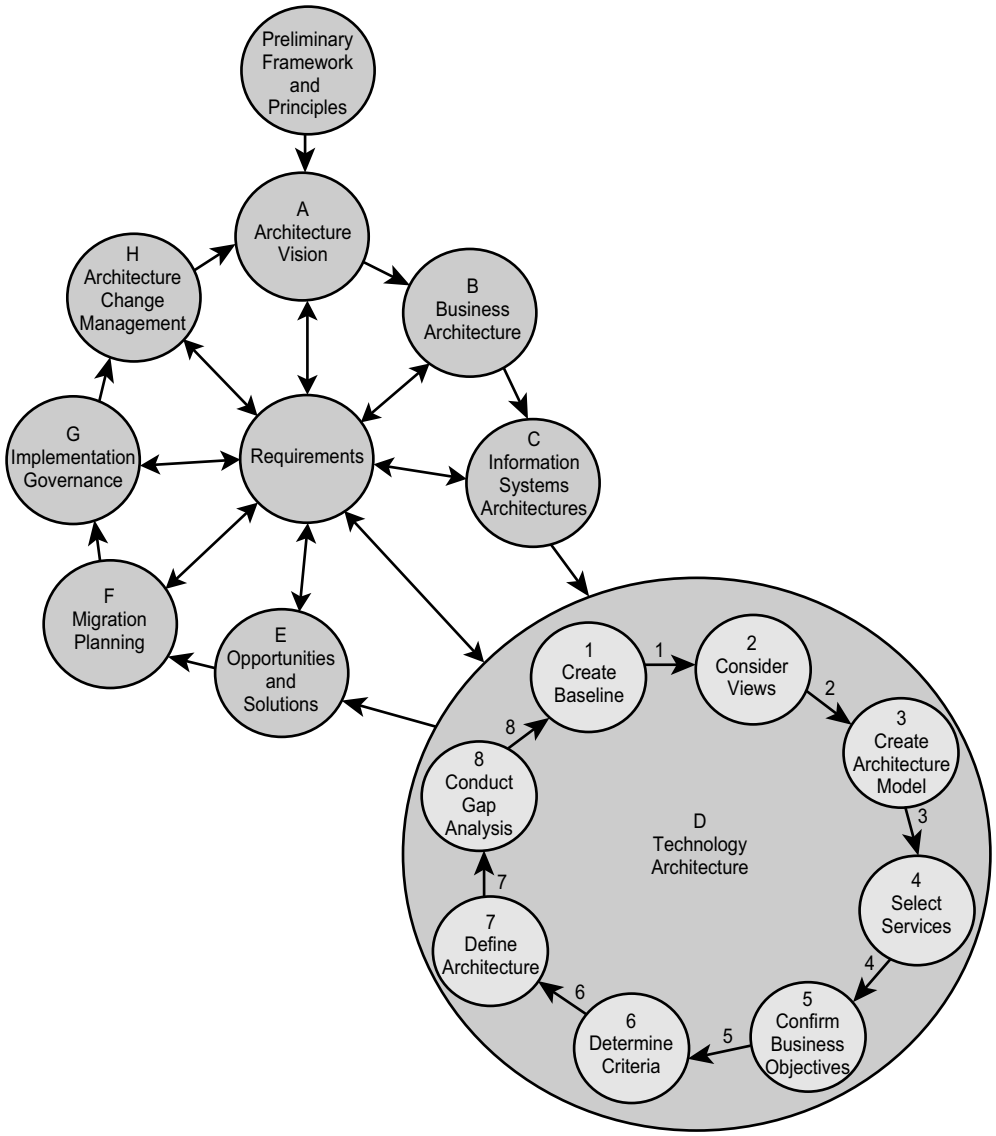


Figure 1.3 TOGAF's ADM process (Source: Open Group)

Phase G: Implementation Governance

Though the actual development process is outside TOGAF, in this phase the processes to ensure that the actual development is in compliance with the target architecture are put together.

Phase H: Architecture Change Management

Once the system's development is complete, we enter this phase to monitor and act upon the change requests. If there is a sufficient need or accumulation of change requests, we enter one more cycle of the ADM process.

Here are some observations on the ADM process.

- The ADM is a cyclical and iterative process where each phase checks the requirements (central element). In addition to the requirements each phase takes input from the previous phase and creates the input to the next phase.
- Information systems architecture (phase C) involves both data architecture and application architecture to some extent.
- TOGAF focused only on phase D (technology architecture) until version 7. Other phases are added in version 8.
- Although this process is meant to be a generic one and is very well articulated, it may not be suitable equally for all organizations. For some organizations it may be overly prescriptive.
- One main criticism levelled at TOGAF is that it is 'deliverable-agnostic' as it is more concerned with the process of developing artefacts. There is no emphasis on the quality or the format of the artefacts.
- A major advantage with TOGAF is that the group has developed several extensions to this framework. For example, there is a lot of documentation on governance, competency models for architects, and certification of products, individuals and services for TOGAF compliance.

In addition to the ADM, TOGAF also provides what is known as *enterprise continuum* as a resource for developing enterprise architecture through re-usable building blocks. It defines two kinds of building blocks, architecture building blocks (ABBs) and solution building blocks (SBBs), and specifies how to develop architectures and solutions using ABBs and SBBs in a continuous and iterative fashion. The enterprise continuum is rather a philosophy composed of the Architecture and Solution continuums. The Architecture Continuum is a set of set of guidelines and tools that provide direction and support to use the Solutions Continuum so that you can ultimately build the technology architecture. The Solutions Continuum consists of a set of solutions to realize the architecture including commercial off-the-shelf solutions and the solutions that are built within the organization.

In summary, TOGAF provides a set of foundation architectures to aid architects understand the present and future needs of the architecture. In the enterprise version, TOGAF was expanded to cover the business, application and information aspects of enterprise architecture; the process for these is not as fully developed as the process for the technical architecture. Similarly, the relationships between the different aspects of architecture are also not completely captured and documented.

Reference Model for Open Distributed Processing (RM-ODP)

RM-ODP is a reference model that provides a framework for specification of distributed computing systems. It is based on the current practices in the distributed processing community as well as on the use of formal description techniques for specification of architectures. Its foundations are in the information management and systems thinking disciplines. It combines the concepts of open systems in specifying distributed systems. The RM-ODP model is found to be very useful by the entire architecture community and is being widely adopted.

The RM-ODP framework is the origin of the famous '4+1 view' model. The RM-ODP framework guides the modelling process of systems architecture by giving five viewpoints that are considered essential. It provides the following explanation for each of the viewpoints (RM-ODP, 2007).

- *The enterprise viewpoint* is concerned with the purpose, scope and policies governing the activities of the specified system within the organization of which it is a part. This viewpoint captures enterprise policies such as permissions given, prohibitions and obligations.

- *The information viewpoint* is concerned with the kinds of information handled by the system and constraints on the use and interpretation of that information. It captures static, invariant and dynamic schemas of the information flowing through the system.
- *The computational viewpoint* is concerned about decomposing the system functionally into a set of objects that interact at interfaces—enabling system distribution. The computational viewpoint captures interfaces among various sub-systems and their behaviour using object encapsulation methods.
- *The engineering viewpoint* is concerned with the infrastructure required to support system distribution.
- *The technology viewpoint* is concerned with the choice of technology to support system distribution.

The 4+1 View Model

There are several profiles of RM-ODP in use today. The representational aspects of RM-ODP are made use of in coming up with several profiles including the famous '4+1 View Model'. Shown in Figure 1.4, it is one of the best known RM-ODP profiles. This framework provides a viewpoint-based architecture approach (Kruchten, 1995).

The viewpoints given by the 4+1 View Model are the following.

- *Logical.* Logical representation of key packages and subsystems. This supports mostly the functional requirements and represents the problem domain. Here the system is decomposed into several key abstractions in the form of objects and classes by exploiting principles of abstraction, encapsulation and inheritance.
- *Process:* The process view is concerned about non-functional requirements such as availability, performance, reliability and security. The focus is on concurrency and distribution aspects and fault-tolerance. The process view addresses questions such as how various OS threads, tasks or

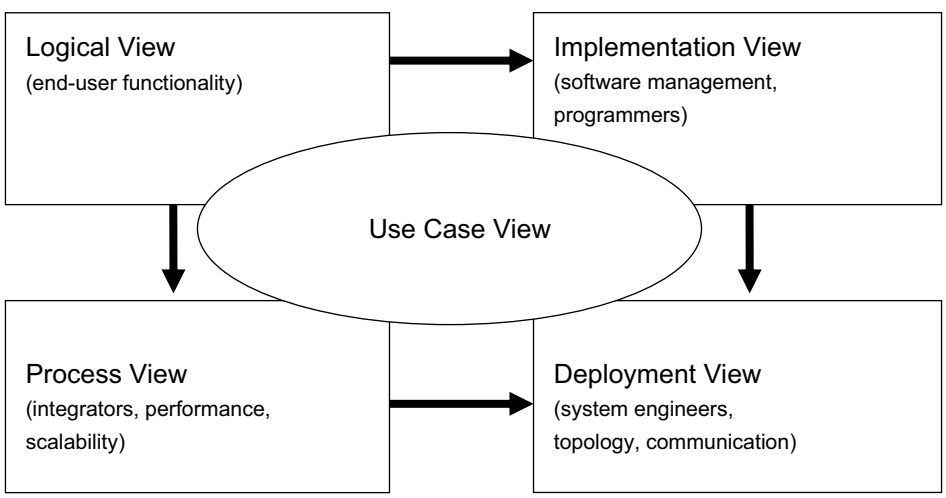


Figure 1.4 The 4+1 view model

processes communicate with each other. The process view can be described at several levels of abstraction, and at each level, the issues of concern and the detail are different.

- *Implementation:* The implementation or development view describes how actual software is implemented (for example, code, directory structure, library structure). The software system is packaged as a set of sub–systems, each of which is developed by a small team of engineers. Each sub-system provides interfaces to other sub-systems, and these are typically arranged in layers, where each layer talks to only its neighbours.
- *Deployment:* The deployment or physical view describes how actual processes are instantiated and deployed on physical hardware. This viewpoint is also primarily focused on non-functional requirements. This view shows the mapping of the software to different process nodes (such as a computer) and therefore it is expected that the deployment viewpoint should be very flexible in terms of its impact on the source code—that is, one should be able to decouple software packages that can be deployed on a given process node.
- *Use case:* This view captures all important scenarios and in general the use cases of the system, which is essentially the behaviour of the system. This is the additional view (the '+1') of the system, and strictly speaking this is a redundant view (and hence the name) if all other views are given. But it is useful in discovering all the architectural elements and also in validating the systems at various stages as it is being built. That is how most test cases are derived from the use cases of the system.

In summary, use the case view models enterprise objects through a set of scenarios. The logical view captures objects, classes, packages and relationships among them. The process view models the control flow and communication among objects. The structure of the software and various modules is captured by the implementation view. Thus, 4+1 views cover all aspects of architecture.

Viewpoint Languages: Notations

Within each viewpoint, the RM-ODP approach uses formal notations (or specification languages) that support architecture description.

One of the most useful notations for specifying computational viewpoints is the ODP interface definition language (ODP IDL). ODP IDL is a related international standard that is identical to CORBA IDL. ODP IDL enables the specification of object encapsulations that can be implemented on multiple infrastructures, such as CORBA, Microsoft COM and the Adaptive Communication Environment (ACE).

Since ODP IDL is programming language independent, a single interface specification suffices to define inter-operable interfaces for C, C++, Ada95, COBOL, Smalltalk, Java and Microsoft IDL. These mappings are defined by open system standards and supported by commercial products. Another useful notation for describing architecture viewpoints is the Unified Modeling Language (UML). Through out this book we make extensive use of UML in designing architectures.

Rational Unified Process (RUP)

RUP is also a variation of RM-ODP, and it makes use of the 4+1 View Model as described above as a foundation. RUP benefited from contributions of various high-profile researchers including Grady Booch, James Rumbaugh and Ivar Jacobson. The RUP framework in turn contributed to creating the Unified Modeling Language (UML).

The use case view is most important in RUP. In addition to all important scenarios and use cases, RUP recommends documenting what it calls 'use case realization'. The purpose is to give the solution outline along with the problem description, that is, for a few important use cases or scenarios, to illustrate how

the software actually works by giving the corresponding realizations. It explains how the functionality is achieved by various design model elements.

In addition to the other four views (logical, process, implementation and deployment), the RUP documentation consists of an additional and optional *data view*. The data view gives a description of any persistent data storage perspective of the system. For systems where the persistent data are not important, deriving the data model from the logical model is trivial. In these cases, the data model can be ignored.

The RUP documentation also recommends describing important characteristics of the software that impact the performance targets and the architecture. This description is presented along with the quality attributes and the non-functional attributes of the system.

Architectural Styles or Architectural Patterns

Mary Shaw and David Garlan, in their book on software architectures (Shaw and Garlan, 1996), described several architectural styles with the following definition: 'Architectural styles are recurring organizational patterns and idioms'. They are the established and shared understanding of common design forms. The Pattern-oriented Software Architectures (POSA) community popularized them as architecture patterns similar to design patterns. For POSA practitioners, the architectural style is an abstracted and re-usable characteristic of recurring composition and interaction of several architectures. The difficulty with architectural patterns is that although they occur regularly in systems design, they occur in ways that makes it difficult for us to detect them. The main reason for this difficulty is that each domain uses different names to refer to the same elements. To address this problem, architectural patterns are being catalogued, and more such patterns are getting added to this catalogue.

Each architectural style will have some basic properties. These properties define the characteristic of the style. An architectural pattern is determined by the following.

- *Vocabulary of the design elements.* What are the components and the connector types? Which one of the components are storage types and which compute a function, etc.
- *A set of configuration rules.* How can these components be composed? What are the topological constraints in connecting these components? Which are valid compositions and which ones are not?
- *A semantic representation.* Every composition of design elements will have well-defined meanings. Each component and connector will have semantic constraints on what it can do and what it cannot.
- *Built-in analysis within the pattern.* Some patterns have possible analysis of the system built in. For example, code generation is a kind of such built-in analysis.

There are several benefits of architectural patterns. Besides design level and code level re-use, they greatly contribute to the understanding of the system-level organization. For example, the moment one describes their system as 'three tier architecture', a lot of information is understood. Architectural patterns contribute to the inter-operability of the components. As the styles are standardized, it is possible to make the components across the platforms as can be seen in CORBA or EJB. Another large benefit of architectural patterns is that of pattern-specific analyses and visualization of the system. These pattern-specific depictions match the mental models of the engineers so that they are able to related to and work with and enhance the new architectures quickly.

Architecture patterns can be classified into two major categories: domain *independent* and domain *dependent*. Domain-independent styles deal with architectural characteristics that are globally applicable organizational characteristics and are popularly known as idioms or simply patterns.

The domain application-dependent architectural styles are also known as reference models as they capture specific configurations for specific application areas. Reference architectures typically constitute the architectural styles that are applicable to specific domains. It does not mean that they are not applicable outside their initial domains. It is possible that they might be useful in completely different domains. RM-ODP was initially designed for distributed processing, but we saw that many of the ideas from there are useful in non-distributed computing environments as well. Similarly, if we develop a reference model for a specific domain such as insurance, some of these styles can be useful for a different domain such as telecommunications.

The POSA community categorized *domain-independent* architectural patterns into the following types.

- *Structural patterns*. This category provides ways to sub-divide the system into sub-systems. This category includes patterns such as pipes and filters, and blackboard and layered architectures
- *Distributed patterns.* Patterns in this category help address the needs of distributed systems as opposed to centralized systems. Popular distributed patterns include peer-to-peer architectures and broker architectures. Please note that patterns such as pipes and filters and micro-kernel that are listed under other categories can also help in creating distributed systems (this is true in other cases also). The broker pattern helps in structuring the software systems that are decoupled from each other by coordinating communication using remote procedure calls.
- *Interactive patterns.* Model–view–controller and presentation–abstraction–control patterns are popular interactive patterns which help in building systems that have very important user interface elements that should be largely independent of the functional core and the underlying data model.
- *Adaptable patterns.* Systems evolve over time with new functionality and modifications to existing functionality. Adaptable patters such as micro-kernel and reflection help design systems that accommodate the new changes. A micro-kernel pattern separates a minimal functional core from extended functionality and customer-specific parts. The reflection pattern helps split the software into the meta-level and base level, where the meta-level helps software to be self aware and the base level includes the core functionality. Changes made to the meta-level information results in changing the base-level behaviour.

We will have an opportunity to discuss interactive patterns and adaptable patterns (chapters 3, 4 and 6) later in this book. In this section we discuss the following structural and distributed patterns:

- pipers and filters
- blackboard
- layered abstract machines
- N-tier
- distributed peer-to-peer systems (including special cases such as client–server and master–slave).

Pipes and Filters

In the pipes and filters pattern, components are filters and connectors are pipes. Each filter takes input in some form and produces output in some other form, which may or may not be similar to the input form. Hopefully, each filter will add value to the output stream because of processing done inside the filter. Each filter is independent and is unaware of the up and down stream filters. Pipes are conduits of the data streams. Famous examples of the pipes and filters architectural pattern are UNIX shells, signal processing systems and distributed systems.

There are several variations within this pattern. Linear sequences of filters are known as *pipelines*. *Bounded pipes* limits the amount of data on pipes. *Typed pipes* impose constraints on the data they conduit with strong typing. In *batch sequential* pipes data streams are not incremental.

Figure 1.5 provides an example of pipes and filters. In compiler design, typically the source code (input text or a set of strings) passes through a series of filters before the object code is generated. These filters include the preprocessor, lexical analyser (lexor), parser, semantic processor, optimizer and finally code generator. The text preprocessor removes noise such as comments and does the processing of macros or include files. The lexor takes each character to compose valid tokens and the parser builds a parse tree to check if the input text is syntactically correct. The semantic parser checks if the statements are meaningful. The optimizer does code optimization and finally the code generates the object code. In this diagram, each of these processors is shown as filters and the pipes shows the intermediate data that flow between the two filters.

The main advantage of the pipes and filters pattern is that the behaviour of the system is the sum of the behaviour of the individual components, and that makes the system easy to manage. It allows a high degree of reuse, replacement and addition of filters. It allows concurrent processing very naturally as filters are independent of each other. However, it also has very serious disadvantages because it is structured in the form of batch processing. It allows several bottlenecks in the process because the slowest and lowest common denominator is data transmission. The pipes and filters pattern is not suitable for interactive applications.

Blackboard

The backboard pattern is used for problems for which there is no known deterministic solution. In the blackboard solution strategy, there are several sub-systems, each working with its own strategy to create a solution.

There are two kinds of components in the blackboard architectural pattern. One is the central data structure called the blackboard and other is a set of processors operating on this blackboard. Many artificial intelligence (AI) systems, integrated software environments and compiler architectures employ blackboard architectural patterns. The systems control is entirely driven by the blackboard state.

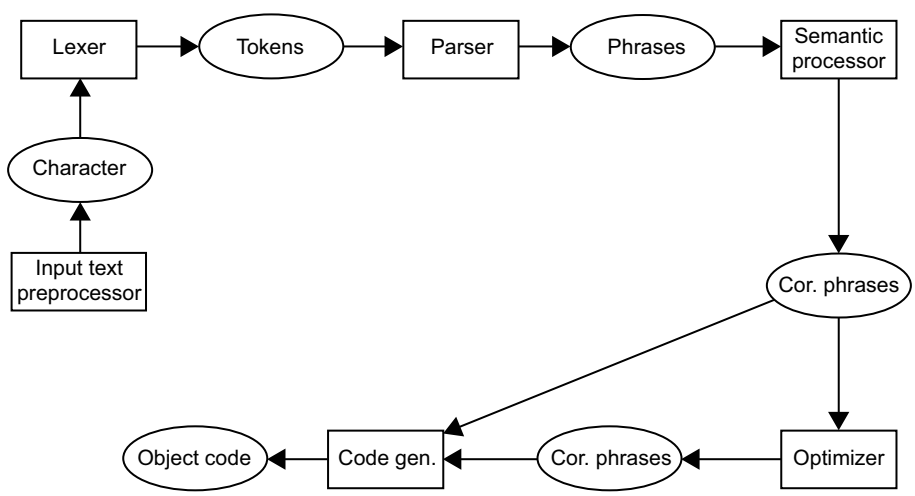


Figure 1.5 Pipes and filters example—compiler design

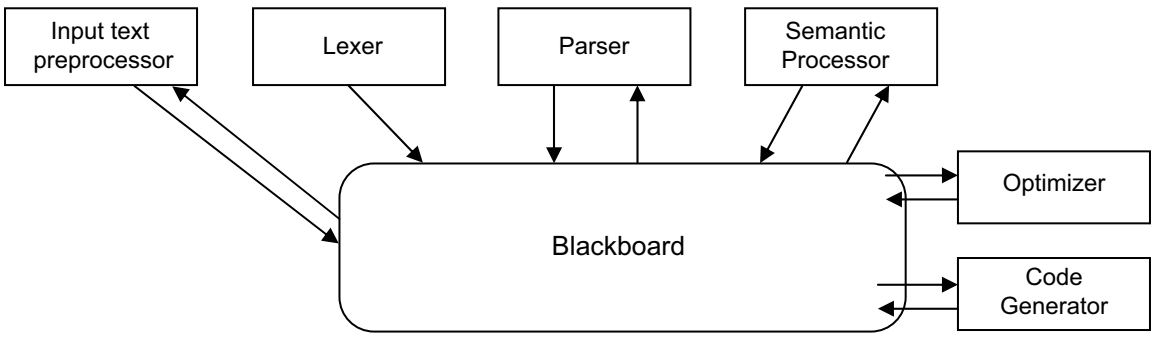


Figure 1.6 Blackboard architecture style

The compiler design example that we have shown for the pipes and filters architectural pattern can be re-designed using the blackboard architectural style shown in Figure 1.6.

The advantages of blackboard architectures are that they allow experimentation, easy modifications and maintainability and provide support for fault tolerance. However there are some disadvantages. Blackboard architectures are not efficient, they are difficult to test, there is no guarantee of good solution, and the cost of development is high.

Layered Abstract Machines

While capturing a complex system's functionality, we typically divide the functionality into several layers of abstraction. Layering means grouping of functionality in an ordered fashion. In other words, layering architecture partitions the functionality into separate layers stacked vertically, each layer interacting with layers underneath. Such layering is called *strict layering.* There are non-strict layering architectures where this restriction of ordered interfacing is not followed, often resulting in sacrificing the benefits of layering. They however impose less overheads.

As shown in Figures 1.7a and 1.7b,[3] layering is typically done by packaging application-specific functionality in the upper layers, deployment specific functionality into lower layers and the functionality that spans across the application domains in the middle layers. The number of layers and how these layers are composed is determined by the complexity of the problem and the solution.

In most layered architectures you will find the following layers:

- *The application layer.* This layer contains the application-specific services.
- *The business layer.* This layer captures several business-specific services or components that are common across several applications.
- *The middleware layer.* This layer packages several functions such as GUI builders, interfaces to databases, platform-independent operating system services, reports, spreadsheets and diagram editing libraries and so on.
- *The database or system software layer.* This bottom layer contains operating systems, databases and interfaces to special hardware components.

Depending on the complexity of the system, the business layer or the middleware layer can be split further into several layers. This helps in achieving better abstraction and clarity. However, there is generally only a single application-specific layer. If the problem space is well represented in business later, the solution

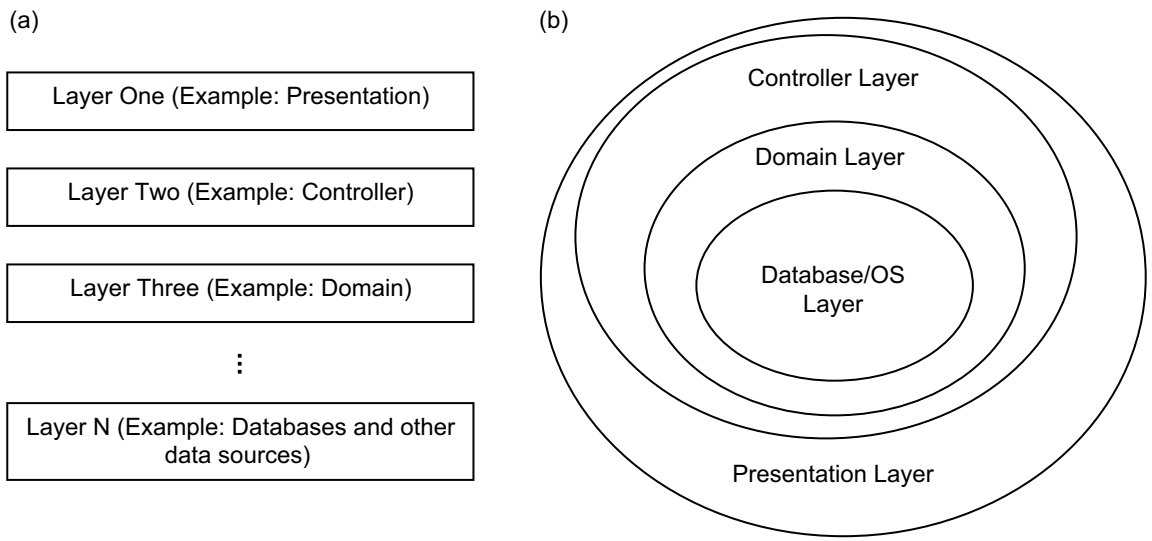


Figure 1.7a,b Layering architecture

spaces are well supported by middleware layer libraries. If the software systems needs to interface with several and diverse hardware devices, then there is a need to organize the system software layer, which means that it will have well-developed lower layers, with perhaps several layers of middleware and system software.

The layered architecture style is very popular among development teams in the current software development environment. There are several advantages with layered architectures which are behind its popularity. They include the following:

- There are different semantics associated with the functionality of the system. With the layered approach, it is possible to define different semantics at each level separately.
- Layered approach aids in making the system more modular. Modularity helps separate the local problems and addresses them more efficiently. This helps in achieving a high degree of coupling and also in keeping the coupling to a lower level. Low coupling with high cohesion is always a desired design principle.
- Because of the modularity most layers end up embodying well-defined abstractions.
- If you take a layered approach, you can scale up the application very naturally. You can always restructure the layers by splitting a bulky layer into several smaller layers if that helps in achieving a better abstraction and efficiency.
- Another major advantage with a layered structure is that you do not have to bother about interfaces of non-neighbouring layers. While designing any layer, the architect or the designer needs to look at the interfaces provided by only its neighbours.

The layered architecture style also has disadvantages. The two main disadvantages are that each layer will add additional overheads and it is often difficult to define clear and independent layers. First, each layer needs to be constructed in such a way that it exposes proper interfaces to its neighbouring layers. As the data or the process moves from one layer to another, there are many overheads including marshalling, un-marshalling, encryption and decryption. Second, it is not always possible to define layers independently.

For example, when you make a change like adding one edit box in the user interface, it may be necessary to make changes in all layers—not just in the database layer. That means, for any change, small or big, it may be necessary to make changes in almost all the layers, which is not a very desirable situation.

N-Tier Architectures

Layered architectures are often realized as N-tier architectures. However, there are some differences between these two architectures. The first major difference is that in an N-tier architecture each tier is independent of the other tiers, whereas a layer is dependent on its upstream or downstream layers. Layering is ordered, and N-tier architecture does not impose ordering of the tiers. A non-strict layering can be loosely termed as an N-tier architecture.

A two-tier architecture represents the client–server model. A typical three-tier architecture consists of a presentation layer, application/business layer and data source layer. A four-tier system consists of a presentation layer, user session layer, application/domain layer and data source layer. This kind of architecture is needed to support the advanced behaviour required, which is achieved by abstracting out partial functionality from the business layer and packaging it as a user session layer. A four-tier model is like a three-tier model where the application layer is split into two. Similarly, a typical five-tier architecture may add either a workflow layer on the client or a business rule layer on the server (or both) to the four-tier architecture (Figure 1.8).

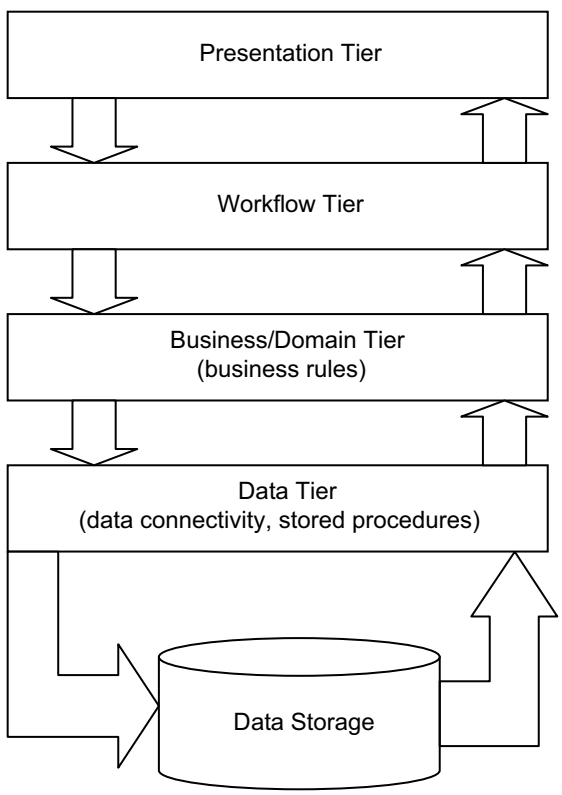


Figure 1.8 Five-tier architecture

Given all these architectural patterns, it is important to note that different architectural patterns result in different architectures with highly different characteristics. It does not mean that all architectures based on the same pattern will be the same. A given architectural pattern may result in different architectures, and the pattern alone does not fully influence the resulting architecture. There will be sufficient room even after adopting a particular architectural style or a pattern for individual judgement and variation that is brought in by the individual architect. Customer requirements and interaction also brings in different emphases to the resulting architecture.

There are still many open issues in architectural patterns. In practice, the use of patterns is generally ad hoc. The boundary between the system and the pattern is not very clear. That is, it is difficult to delimit the system aspects that can be or should be specified by the pattern. It is often very difficult to compare the systems that are based on the different patterns and similarly compare two different patterns based on their properties. It is also not very clear how existing architectural patterns can be combined to form a new pattern though we know that most actual systems are based on combining multiple architectural patterns.

Distributed Peer-to-Peer Systems

This is a generic style of which popular styles are the client–server and master–slave styles. In the distributed peer-to-peer pattern the components are independent objects or programs that offer public services or consume these services. Connectors in this style are the remote procedure calls (RPCs) over the computer networks.

The client–server pattern is a specialized case of the peer-to-peer style where there are only two peers: a server, which offers the service, and the client, which uses the service. In this pattern, the server interacts with several clients but it does not know identities or the number of its clients. Each client knows the identity of the server. In the client–server pattern, the components are clients and servers, and the connectors are the RPC-based interaction protocols.

The master–slave architecture style, another variation of the peer-to-peer system, the master component organizes work into distinct sub-tasks, and the sub-tasks are allocated to isolated slaves. Slaves report their results to the master, and the master integrates results. Examples of the master–slave architecture style include process control systems, embedded systems and fault tolerant systems.

The main advantages of the distributed peer-to-peer architectural style are that it enables inter-operability, re-usability, scalability and distributability. It is a natural choice for high-level heterogeneous distributed systems. This style is typically used in making the legacy systems re-usable by providing new interfaces to them. They also help in scaling by creating powerful servers that can support several clients. The main disadvantage with peer-to-peer systems is that they are very difficult to analyse and debug. This style is also criticized for being too simple in providing inter-operability mechanisms (such as legacy wrappers, data marshalling and un-marshalling, proxies and stubs for RPCs) unlike layered architectures.

Anti-patterns

Anti-patterns are counterparts to the patterns we have discussed. These are the patterns to be avoided. If we study them and are able to recognize them, then we should be able to avoid them. If we are not aware of anti-patterns, we risk repeating the mistakes others have made several times. By formally capturing the repeated mistakes, one can recognize the symptoms and then work towards getting into a problem situation. Anti-patterns tell us how we can go from the problem to a bad solution. In fact, they look like a good solution but when applied they backfire. Knowing bad practices is perhaps as valuable as knowing good practices. With this knowledge, we can re-factor the solution in case we are heading towards an anti-pattern. As with patterns, anti-pattern catalogues are also available. Books on anti-patterns (including *Anti-Patterns: Re-factoring Software, Architectures, and Projects in Crisis*, *Anti-Patterns and Patterns*

in Software Configuration Management and *Anti-Patterns in Project Management*) and Web sites such as Portland Pattern Repository's Wiki[4] give several anti-patterns related to design, development, architecture, project management, etc. We will have the opportunity to discuss some popular anti-patterns such as *Architecture by Implication, Cover the Assets, Design by Committee, Reinvent the Wheel, Spaghetti Code, Stovepipe Enterprise, Stovepipe System, Swiss Army Knife* and *Vendor Lock-In* later in this book.

It is interesting to know about some of the popular anti-patterns relating to the way architects design architectures (not about specific architectural anti-patterns). Portland Pattern Repository's Wiki lists some of them.

- *Over-generalized interfaces.* This anti-pattern is found when architects become very ambitious and try to design a system with too much flexibility. This kind of attempt often results in very flexible systems but is almost impossible to maintain. A good example for this anti-pattern is designing all interfaces using text strings or xml messages. Since there are no explicit contracts between the interfaces, these kinds of systems end up being very hard to maintain.
- *Too many buzzwords.* It is a very common to find architects using the latest buzzwords (such as SOA, SaaS, MDA, .NET, XML) in their architectures even when they do not fit into their scheme of things. It can only result in the creation of *marchitectures* (marketing architectures, more on them in next chapter) but not architectures. It does not help in solving problems. In fact, marchitectures work against creating effective solutions.
- *Product architecture.* It is also common to find architects who are very familiar with one platform or a product to tend to create all their architectures very close to the architectures or design of these platforms or products. This again results in just a marketing architecture and will be of no use in solving problems.
- *FUD architecture.* Some architects are afraid of being wrong and worried that their architecture will be changed later. With their fear they cannot really do any thing concrete and end up with a design that actually solves nothing.

We will discuss anti-patterns further in Chapter 3.

SOFTWARE ARCHITECTURE AS A PROBLEM-SOLVING ACTIVITY

Software architecture is actually a discipline of problem solving. As a software architect progresses from being an expert in platform-specific technology know-how to an expert in platform-independent technology know-how, she or he masters several problem-solving techniques. An architect needs to be aware of the environment and recurring problems and solutions that are actually working. Once this awareness sets in, documenting, internalizing and applying problem-solving techniques readily distinguishes a good architect from others. The design and architecture patterns community encourages this kind of an approach.

In this section, we describe a problem-solving approach in general that is inspired by Polya's famous work *How to Solve It* and discuss how systems thinking discipline can help you master problem-solving techniques.

Polya's How to Solve It—A Problem-Solving Guide

George Polya was a Hungarian mathematician who spent his later days characterizing the general methods that people use to solve problems. He spent a lot of time in describing how to teach and learn problem solving. His book *How to Solve It* became a classic and is recommended highly not just for students of

mathematics but for almost all engineering and science disciplines. His other books include *Mathematics and Plausible Reasoning Volume I: Induction and Analogy in Mathematics* and *Mathematics and Plausible Reasoning Volume II: Patterns of Plausible Reasoning*. Both these books address the subject of problem solving as well.

How to Solve It is a small volume describing problem solving. It provides general heuristics for solving problems of all kinds. The book starts with advice for teaching mathematics to the students in a classroom. It also describes a four-phase technique to solve any problem (Polya, 1956), which is the core of this book:

1. Understanding the problem (What is the unknown? What are the data? What are the conditions?)
2. Devising a plan (Have you seen the problem before? Have you seen the same problem in a slightly different form?)
3. Carrying out the plan (Can you see clearly that each step is correct?)
4. Examine the solution obtained (Looking back, how could it be better?)

If this technique fails, Polya advises, 'If you can't solve a problem, then there is an easier problem you can solve: find it'. A major part of this book is a dictionary (a catalogue) of heuristics. This book is a must-read for every budding architect.

If you look closely at the technique described by Polya and the process of designing architecture, they are very similar. Architects can learn a lot of techniques from Polya's approach to problem solving.

The key in both domains is to acknowledge that there are different kinds of problems and different kinds of solutions. As you move from the problem space to the solution space, at each step you must make informed choices to match problems with solutions. You need to be careful while making decisions that impact the overall system organization.

All design decisions require a proper understanding of the problem as well as the solution. This means not just the understanding of various dimensions of the requirements but also the limitations of various design or solution alternatives. There are no silver bullets or magic recipes for good design.

One needs to understand various parts of the problem and distinguish between what is known, what is unknown, what the data are, what the process or functionality is and what the conditions are. It is recommended that suitable notations be used and figures drawn. All these techniques help architects understand and represent the problem correctly.

Problem analysis or the problem frames approach (Jackson, 2001), designed based on the ideas discussed, is used while gathering requirements and creating the specifications document. The problem frames technique enables an understanding of the problem context. It recommends drawing context diagrams, problem diagrams and various kinds of problem frames. A problem frame is similar to a problem pattern. It captures an abstraction of a class of problems which in turn guide you in finding an appropriate class of solutions.

After understanding various aspects of the problem space, we will have a better clarity regarding what kind of solutions we need to look for. Problems require a context, which is the subject matter of the solution that needs to be developed. The problem needs to capture all relevant parts of the real world, relationships between these parts and the properties of these parts. Most often, the knowledge about the problem domain exists in a very non-formal and ad hoc fashion. As the architect is learning about the domain, he or she needs to represent this ad hoc knowledge of the domain in a more formal description and ideally in a form that is similar to the problem frames because such a description brings the problem closer to possible solutions.

The solution characterizes the proposed software system or the machine that solves the problem. The solution space includes the software architecture, design patterns and programming idioms.

The most important contribution made by the problem-solving guide to the software architecture process is mapping the known problems to known solutions. That is *devising a plan* in Polya's words. It encourages the architect to think about the problem in terms of a set of known problems and to check if the new problem is actually one of the old problems but in a slightly different form. It guides the architect while looking at the *unknown aspect* of the problem to look at all the known problems that have the same or similar unknown. If there are solutions for problems that are similar to the ones you are solving, it asks you to think if an old solution can be used. In case the entire solution is not useful as it is, it encourages you to consider if its result, or its method or even a part of this solution, is applicable in the new context. It also asks you to consider augmenting the old solution with some auxiliary element so that it can be useful. In addition it says, 'If you cannot solve the proposed problem try to solve first some related problem' (Polya, 1951, p. xvii).

Polya recommends transforming the proposed problem into one of the known and more accessible related problems. This can be done by:

- generalizing the given problem
- making the given problem a more special one
- searching for a more analogous problem.

Other techniques recommended are:

- solving a part of the problem instead of trying to solve it in its entirety
- varying the given conditions of the problem by keeping only a part of the condition
- determining the unknown by studying how it varies
- deriving something useful from the data
- looking for other appropriate data which may be available elsewhere to understand what is unknown in the current scenario
- changing the unknown so that the data and the new unknown are closer to each other
- changing the data so that the new data and the unknown are closer to each other
- changing both the data and the unknown so that new data and the new unknown are closer to each other.

There are several heuristics related to the craft of software architecture that are inspired by Polya's dictionary of problem-solving heuristics. For example, many architecture and design patterns are based on the principles of generalization, decomposing and recombining. Many successful architects document and apply their own personal library of heuristics.

Systems Thinking Approach to Problem Solving

Most real-world systems are very complex, and they are increasingly becoming more chaotic. There seems to be randomness in the way these systems function. In spite of making use of very sophisticated methodologies and techniques to understand the characteristics of these real-world systems, we are unable to explain systems' behaviour in a convincing manner. Some systems are successfully implemented and some are not, both unfortunately because of wrong reasons. This is a constant feeling an architect or a system builder gets while building complex systems. There seem to be something missing.

We are trained to think in an analytical manner throughout our education. We call it rational thinking or adaptive thinking. Adaptive thinking is reactive to an event and looks for linear causal relations and thus misses the big picture or the environment in which an event occurs many times. This kind of thinking deals with independent variables of the system.

Systems thinking is a more scientific problem-solving approach than the rational thinking approach as it deals with inter-dependent variables and looks for non-linear or circular causal relations. Hence, systems thinking is a realistic and useful method for solving problems. Systems thinking is a way of looking at real-world systems for understanding them and possibly for improving them. However, it may be important to note that in an absolute sense systems do not exist—we use this term in a more metaphorical sense to give various perspectives on real-world problems. Systems thinking is increasingly becoming a profound problem-solving tool in software architecture.

To understand systems thinking, let us look at some of its basic concepts.

A *system* is a group of parts that work together to achieve a common goal. Parts are generally systems themselves and are composed of other parts, and this definition can be applied recursively. There is a concept of the *connectedness* between the parts of the system. Sherwood (2002) defines this connectivity in his famous book *Seeing the Forest for the Trees: a Manager's Guide to Applying Systems Thinking* as follows:

> If you wish to understand a system, and so be in a position to predict its behavior, it is necessary to study the system as a whole. Cutting it up into bits for study is likely to destroy the system's connectedness, and hence the system itself.

Systems thinking is based on a popular fundamental principle which says 'the whole is bigger than the sum of all its parts'. In general, the parts are affected by being in the system, and their behaviour may change when they leave the system. If the parts are changed or removed or added, the system changes its behaviour. The way these parts are assembled adds value that is more than the value brought in by individual parts.

Any system is defined and can be characterized in terms of the following:

- *Purpose*. Every system exists because it has a purpose or objectives. It is impossible to characterize or understand a system whose purpose is not clear.
- *Input* and *output*. Every system takes input from the environment and provides output back to the environment. The output does not necessarily and directly achieve the purpose of the system. In other words, the output of the system and the purpose of the system can be different.
- *Function*. The function of the system transforms the input to the output.

Every system has two kinds of boundaries: an *inner boundary* and *outer boundary*. The inner boundary separates the system from rest of the systems or components with which the system interacts. The outside boundary in which a system exists is called the *system's environment*. This outside boundary influences or affects the way systems function.

The difference between the purpose and the output is created by the inside cause and the outside cause. The inside cause occurs within the system, and we will have the ability to control this whereas the outside cause occurs from the environment on, which we may not have any control. The result or the output of the system can be influenced by either the inside cause or the outside cause.

For example, consider a retail store. The purpose of the store is to conduct business and make profits. If we cannot conduct business, this situation is an *output*. If we cannot open the store because of bad weather, the bad weather is an *outside cause* because we cannot change the weather. If we cannot open the stores because we forgot the keys to the store and leave them at our home, it becomes an *inside cause*, which is solvable.

Systems Principles

Jamshid Gharajedaghi (1999) in his book *Systems Thinking: Managing Chaos and Complexity: a Platform for Designing Business Architecture* talks about five systems principles. These principles are important to

understand because these principles govern the behaviour of systems we attempt to understand or build. These principles are the means using which we can understand the complexity of systems in a non-linear fashion. Understanding these principles help us overcoming limitations of our analytical or rational approach to understanding a system's behaviour. In this section, we only introduce these principles. We advise the reader to refer to the original material for more details.

Principle of openness. This principle states that the behaviour of any system can be understood only in the context of its environment. Therefore, we can neither understand the problem nor design a solution free of context. Most often, while designing a system we tend to define the problem in terms of solution. This solution is typically independent of the context or the environment that must have worked in other situations. Since it worked for a different (may be similar) problem, we try to force this to the current problem without understanding the environment. This results in a non-solution, and there are ample experiences to prove this point. Hence the openness of the systems becomes very important in designing the right solution.

Principle of purposefulness. This principle brings out the difference between the rational and systems thinking approaches very clearly. It says 'choice' is in the heart of any system. A true development of a system is to enhance its capacity to choose. Designing is a means for enhancing choice. In the systems thinking literature, it is said that designers seek to choose rather than predict the future.

Systems do what they do as a matter of this choice. This is the purpose of the system. This choice has several dimensions including the rational, emotional and cultural ones. Rational choices are made based on self-interest (that of the decision maker), not the ones that are affected by this decision. This rational decision is not necessarily a wise choice. In the long run, it may turn out to be a lose–lose decision and the holistic thinking strives to make a win–win decision. The emotional decision is based on the excitement or beauty of that decision. The characteristics of an emotional decision include excitement and challenge, which most often determine the final choice being made. A cultural decision is based on the collective behaviour as opposed to the criteria of an individual. Culture provides certain default values (or decisions) when we do not explicitly know what to choose.

Principle of multi-dimensionality. This is arguably the most valuable principle of systems thinking. This principle tells us to look at two opposing tendencies of a given system as complimentary, as opposed to competitive. There is a mutual dependence between the opposite tendencies of systems. This is typically characterized by an 'and' relationship instead of an 'or' relationship. In traditional or analytical thinking these opposites are treated in such a way that a win for one is a loss another and vice versa. This is called a win–lose equation. The principle of multi-dimensionality emphasizes that win–lose, lose–lose and win–win equations are also possible. It also states that if x is good, then more of x is not necessarily better.

Principle of counter-intuitiveness. Earlier we have discussed how the complexity of open systems is beyond the reach of the analytical approach. Counter-intuitiveness states that the expected or desired results will not match actual results for intended actions. Most often, the actual results are the opposite of the desired results. Various examples illustrate this principle where things getting better before they get worse (or vice versa). As discussed before, either the system can be successfully implemented or it may be a failure, but both for wrong reasons. The principle of counter-intuitiveness tries to explain this criterion.

Principle of emergent properties. Emergent properties are the property of the system as a whole, not the parts. They exist due to the interactions among the parts, and hence they cannot be analysed. By nature, these properties are dynamic because they are products of the interactions between its parts. They are produced online and in real time. They cannot be saved for future use if the parts are no longer interacting in the same way.

Systems Methodology of Problem Solving

Systems thinking recommends a methodology for solving complex real-world problems. This methodology can be summarized as follows.

- The higher levels of understanding can be achieved by iteratively approaching the problem by first starting the enquiry or search process, then modifying and verifying initial assumptions and finally evolving closer to the solution that is satisfactory.
- Design as a holistic process deals with three dimensions: structure, function and process, together with the environment in which the system operates.
- To reduce unnecessary complexity and produce manageable simplicity, we need to do the following: start from a basic frame of reference, focus on relevant issues and avoid an endless search for useless information.

A crucial step in systems methodology is to clearly separate the problem definition from the solution design. This idea may sound familiar, but the treatment is different. The following are the steps (often iterative in nature) in systems methodology as described by Peter Checkland (1999) in his book *Systems Thinking, Systems Practice*.

1. Understanding the problem (often referred to as a *mess* in the systems literature) always starts in an unstructured state. Start your investigation and find out about something about the problem. The basic research into the problem area includes investigating who the key players are, how the process works, etc.
2. 'A picture is worth a thousand words'. Express the problem situation through rich pictures. Rich pictures capture a lot of information relating to the problem situation. Pictures show system boundaries, structure, information flows, communication channels and human activity.
3. Look at the problem situation from different perspectives. 'Root definitions' are written to elaborate a transformation. Every well-formed root definition has six elements: Customer (the one who benefits from a system), Actor (the one who transforms inputs into outputs and performs the activities defined in the system), Transformation process (conversion of inputs to outputs), Worldview (making the transformation process meaningful in context), Owner (proprietor, who has the power to start up and shut down the system) and Environmental constraints (external elements including organizational policies and legal and ethical matters).
4. For each root definition, build conceptual models of what the system must do. In other words, root definitions provide information on *what* to do and this step begins to define *how* to do.
5. In this step, compare the conceptual models with the real world. That is, compare the results from steps 4 and 2 to find out their differences and similarities.
6. In light of the results observed in step 5, identify the changes that are possible and that are needed to improve the situation.
7. Come up with recommendations to improve the problem situation based on changes that are identified in step 6.

In this book, systems thinking as a problem-solving activity plays a significant but invisible role. In every case study, there is an opportunity to apply systems thinking in understanding the problem and designing a solution. Sometimes systems thinking principles are applied in an obvious way in the foreground, but most often they remain in the background. In other words, we make use of systems thinking as a source of light to illuminate the problem space as well as the solution space. Since every case study is

an exercise of problem solving at different levels, we urge the reader to carefully identify the boundaries of the systems they are going to create or improve, the environment in which systems are operating and then think about the problem and solution spaces.

Notes

1. http://www.sei.cmu.edu/architecture/definitions.html
2. http://www.opengroup.org/architecture/togaf/
3. Please note that Figures 1.7a and 1.7b are exactly the same—they are only shown differently to illustrate the difference between a layered architecture and an N-tier architecture, which we will discuss very soon.
4. http://c2.com/cgi/wiki?WelcomeVisitors

Further Reading

There are several good books to start learning about software architecture. One of the seminal books is the one written by Mary Shaw and David Garlan (Shaw and Garlan, 1996). Later many popular books including *Software Architecture In Practice* written by Len Bass, Paul Clements and Rick Kazman (Bass *et al.*, 2003) and the entire series of books on software architecture published by Software Engineering Institute (SEI) of Carnegie Mellon University (CMU) and Addison Wesley give various perspectives on software architecture.

One very effective way of learning about the basic concepts of software architecture is to explore the pages of software architecture community Web sites. The following Web sites are very useful.

- www.softwarearchitectures.com contains useful information about various aspects of software architecture.
- Similarly, www.iasahome.org, which is the Web site of the International Association of Software Architects, and www.wwisa.org (home page of Worldwide Institute for Software Architects) contain resources useful for gaining expertise in software architecture.
- www.softwarearchitectureportal.org is the Web site of IFIP Working Group 2.10 on Software Architecture. This Web site provides information about Working Group 2.10, the WICSA (Working IEEE/IFIP Conference on Software Architecture) conference series, the Software Architecture Village Wiki and other resources on software architecture.
- Grady Booch maintains a good resource on his Web site, a sort of handbook on software architecture. You can find it at http://www.booch.com/architecture/index.jsp.
- The Gaudi Project Web site, http://www.gaudisite.nl, contains very useful information about systems architecture. This includes e-books, courses, case studies and research papers. Philips Research funded and supported the Gaudi Project for many years, and it has gradually moved to the Embedded Systems Institute.
- http://www.bredemeyer.com, is a very resourceful Web site from Bredemeyer Consulting focusing on the software architecture discipline and on enterprise architecture resources, along with their workshops and other training links.
- Of course, for every thing else, we have Wikipedia. More specifically, start with the page on software architecture (http://en.wikipedia.org/wiki/Software_architecture) and explore all its siblings.

The UML 1.3 and 2.0 documentation gives some important definitions and concepts about software architectures. We have made use some of them in this chapter, and it is a good idea to refer to them if you want to investigate deeper into this area. Similarly, IEEE standards on software architecture and software design (such as 610.12-1990,

referred in this chapter) are typically considered as starting points for any exploration in this area. You can find them at the IEEE Xplore site, http://ieeexplore.ieee.org/.

A very useful approach to understanding the problem space in general and software requirements in particular is known as the problem frames approach. It was developed by British software consultant Michael A. Jackson. In his book *Software Requirements & Specifications* (Jackson, 1995) he gives an outline of this approach and describes it more fully in his later book *Problem Frames: Analysing and Structuring Software Development Problems* (Jackson, 2001). We have already discussed Polya's book on problem solving.

On systems thinking, there are again several good books. Gerald Weinberg's *An Introduction to General Systems Thinking*, Peter Senge's *The Fifth Disciple*, Peter Checkland's *Systems Thinking, Systems Practice*, Sherwood's *Seeing the Forest for the Trees: a Manager's Guide to Applying Systems Thinking and* Jamshid Gharajedaghi's *Systems Thinking: Managing Chaos and Complexity: a Platform for Designing Business Architecture* are some of the most influential books. Russell Ackoff's various works including *Recreating Corporations: a Design of Organizations for 21st Century* (1999)*, The Democratic Corporation* (1994) *and Creating the Corporation Future* (1981) form a very useful reading list for understanding systems thinking.

Chapter 2

Where Is the Architecture? Story of a Sick Health Insurance Application

Contributors
Bhudeb Chakravarti
Kirti Garg
Jayanti Vemulapati
Bidhu Mahapatra

In many real-world applications, it is very common to pay little attention to or even ignore the architecture of a system. The role of an architect is completely missing in most projects. Even if there are architectural artefacts, most of them are used as sales material rather than as technical blueprints. Typically, these *marketing architectures*, also known as *marchitectures,* give a high-level view and are used to convince the customer of the team's ability to achieve the required functionality (and the non-functional requirements) of the system. These materials are externally directed towards the customer than towards the members of the internal development team. These artefacts contribute very little to the software development process. They lack standards and hence cannot be used for verification, validation and evaluation of the actual system. In such situations, individual developers are left with the challenge of filling in the details of design before building the system. Obviously, this limits the vision while building the system and many issues come up during integration and scaling of the system.

The role of an architect continues to be important throughout the software development life cycle. If this important aspect is missed, there is a great danger of making a disaster out of a perfectly fine project. An architect has multiple roles to play during the software development life cycle, including that of a client, inspector, trouble-shooter, problem solver, mentor, trainer, designer as well as a visionary. The architect should always keep the 'big picture' in mind. The big picture here is the complete system with its context as a set of interacting components. The ability to have the overall perspective is probably what distinguishes the architect on a project from those whose mental model focuses on individual elements such as the functionality, hardware, project management or sponsorship. At the same time, an architect should be able to understand and communicate various aspects of the system using other mental models with all the stakeholders, most importantly with the members of the development team, and should have the ability to act as an interpreter between them. Any large-scale system development not supported by good architects and good architecture is bound to fail.

In this case study, we shall look at a health insurance application that was executed with great zeal, enthusiasm and effort but failed at the end. One of the reasons for the failure, as found in the postmortem analysis by the project management office (PMO), was that the system did not have any proper architecture. The PMO also identified that many communication gaps could have been avoided had there been a technical architecture in place. This case study also focuses on requirements engineering, estimation, prototyping, domain engineering, business context and project management issues, helping the learner investigate the relationship between the architecture and all the other important aspects of software development.

BACKGROUND

In many projects, software architecture is confused with the technology stack, typically depicted as a three-tier layout: the presentation tier, the business logic tier and the data or storage tier. Many software engineers draw some variation of this technology stack and call it an architecture diagram to express the so-called enterprise architecture of the system they are building. This technology stack diagram tells us nothing about how the functional or non-functional requirements are handled by the system. It can at best help understand what technologies are being used and how they are integrated. Typically, the sole purpose of these architectures is as sales tools rather than as technical blueprints. We can call them 'marketing architectures' because they are directed at external customers and not at system developers.

Sometimes marketing architectures are very useful for advertising the features of software products and systems. That is, the intended audience of potential or existing customers, investors and higher management if you need their buy-in. However, they are of very little use for the development team as there is very little, if any, technical information included.

Besides lack of technical information, marketing architectures also are indifferent or completely alienated from the development process. The software architect is expected to provide the required technical

detail to the development team. It means that the architecture should capture the entire model of the proposed system, including major components or modules and sub-modules and the interfaces between them. Marketing architectures cannot provide these details and hence they are of no real use to the developer. The technical architecture (or the one intended for developers) helps in serving new business requirements as well as extending the old functionality using new technologies. Marketing architectures cannot serve these purposes.

Marketing architectures also spread the misconception that there is a direct mapping between each tier in such an architecture diagram to the individual technologies. For example, the data or storage tier is an RDBMS, the business logic tier is Enterprise Java Beans (EJBs) and the presentation layer is Java Server Pages.

In real life, the situation is more complex. For example, the presentation layer of any large system is typically composed of complex objects that can be mapped to servlets, JSPs, EJBs and data stored in a relational database (user preferences, for instance). Similarly, the business logic and data storage layers make use of multiple technologies.

The common modules such as reporting and security are not limited to any one of the layers. They are spread across almost all the layers. These modules are typically various aspects of the system. Most of the architectural diagrams fail to capture these aspects of the system. On the other hand, these architectures treat layers as major modules. For example, in many projects, it is not uncommon to find the entire development being structured around the presentation layer, treating the user interface as if it were a true module. As a result, the system becomes difficult to develop and maintain.

One of the major goals of software architecture is to provide various conventional viewpoints of the proposed system before it is built so that one can review, evaluate and optimize before developing the actual system. These viewpoints include the technology stack viewpoint that was discussed before, the data or the object viewpoint and the behavioural viewpoint (or the use case model). Each of these viewpoints capture different kinds of design decisions, system functionality and some of the system's quality attributes. However, they do not represent other quality attributes such as modifiability, buildability, security, reliability and performance. They also do not represent business or operational qualities, such as the ability to reduce development and maintenance costs.

Case Study: Assure-Health—Story of a Sick Health Insurance Application

MidAlliance Ltd is a North American software company specializing in developing and marketing software solutions for the health insurance sector. Their main focus is on the third-party administrator (TPA) market segment of the health insurance industry.

Health, auto and life insurance (HALI) is a vertical business unit (VBU) of Fact-Tree Global Solutions (VBU-HALI), focusing on end-to-end solutions in the insurance domain for customers across the globe. VBU-HALI was approached by MidAlliance to develop a Web-based application that would ultimately take the shape of a product that can be customized according to the needs of the customer. This Web application was meant for TPAs of the health insurance sector (see Exhibit A). The agreement was that MidAlliance would market and hold the intellectual property rights (IPR) of this software and VBU-HALI will be responsible for developing the application.

During this period, while the project was still in the early stages of development, MidAlliance identified a TPA, PayMinders Inc, as a potential customer for this application. MidAlliance convinced PayMinders and entered into an agreement to provide them with a complete Web-based solution to cater to their

administration and customer service needs. It brought Fact-Tree into the agreement as a software development service provider. The earlier project got transformed into a new project as a tripartite agreement between Fact-Tree, PayMinders and MidAlliance.

PayMinders Inc is a federal institute. Its bureaucratic structure and the strict laws of the land did not allow them to give the project to a foreign company directly. Hence, there was a three-party agreement between Fact-Tree, PayMinders and MidAlliance, wherein VBU-HALI will customize the Assure-Health product for MidAlliance. MidAlliance, in turn, would deliver it to PayMinders. It was also decided that MidAlliance will receive a fixed payment from PayMinders, which in turn will pay a major part of that money to VBU-HALI. In the three-party agreement, PayMinders was the customer, VBU-HALI was taken in as the technical partner and MidAlliance was the domain partner who was supposed to provide all the domain knowledge.

When the agreement was finalized, MidAlliance gave a prototype demo to PayMinders (based on the work done from the previous project between MidAlliance and VBU-HALI) with a sales pitch sounding as though the product already existed and that the job to be done for PayMinders was just customization of the product.

For VBU-HALI, this project was the entry point to all the TPAs operating in different states of the United States. They were just waiting for a reference project to showcase. Each state has its own set of laws that govern health insurance issues and opportunity exists to customize and sell the base product for different TPAs. It was thus very important for VBU-HALI to accept this agreement and start the project. Once the agreement and scope was finalized, the development process began at VBU-HALI.

The new project involved a complete analysis, design and development of a Web-based system that would be capable of providing the functionalities related to the administration, eligibility, billing as well as interfacing outside agencies. This usually covers functions such as eligibility determination and enrolment, billing, premium collection, carrier payment, rate quote inquiry, reporting and maintenance and administration.

Note that all these functions were strictly role based, that is, the system behaviour and responses were based on the role of the user. The intended users were system administrators, clients, those insured, dependents, providers, customer service representatives, employers and carriers. For each of the users, everything from the user interface to available functionalities to the result displayed should be different. Based on the user type, relevant menus are to be generated dynamically. Hence, the system requires a different look and feel according to user type. The administrator should have ultimate control and be able to define the rules as well as the roles specific to the organization.

PayMinders had its own legacy system performing the same job of administration and billing, but to upgrade and make use of the latest technology, they were planning to shift to a modern Web-based system that would serve not only the TPAs but also the end customers.

This application was christened Assure-Health.

Exhibit A Background on Health Insurance and Third-Party Administrators

Thanks to the pressures of modern life, insurance business (especially health insurance) in the recent times is growing rapidly, invading all aspects of life. As a result, there are many complex scenarios coming into play in the way business is conducted. Usually, health insurance companies or carriers sell the policies directly to the consumers. They also sell insurance through third-party insurance administrators (commonly known as TPA). These Administrators typically offer various policies from different insurance companies customizing the needs of the customers. TPAs typically add business value by performing various tasks such as eligibility check, enrollment and billing. Checking for the eligibility of the applicant involves collecting information on current health, payment capability, family and employment information, and so

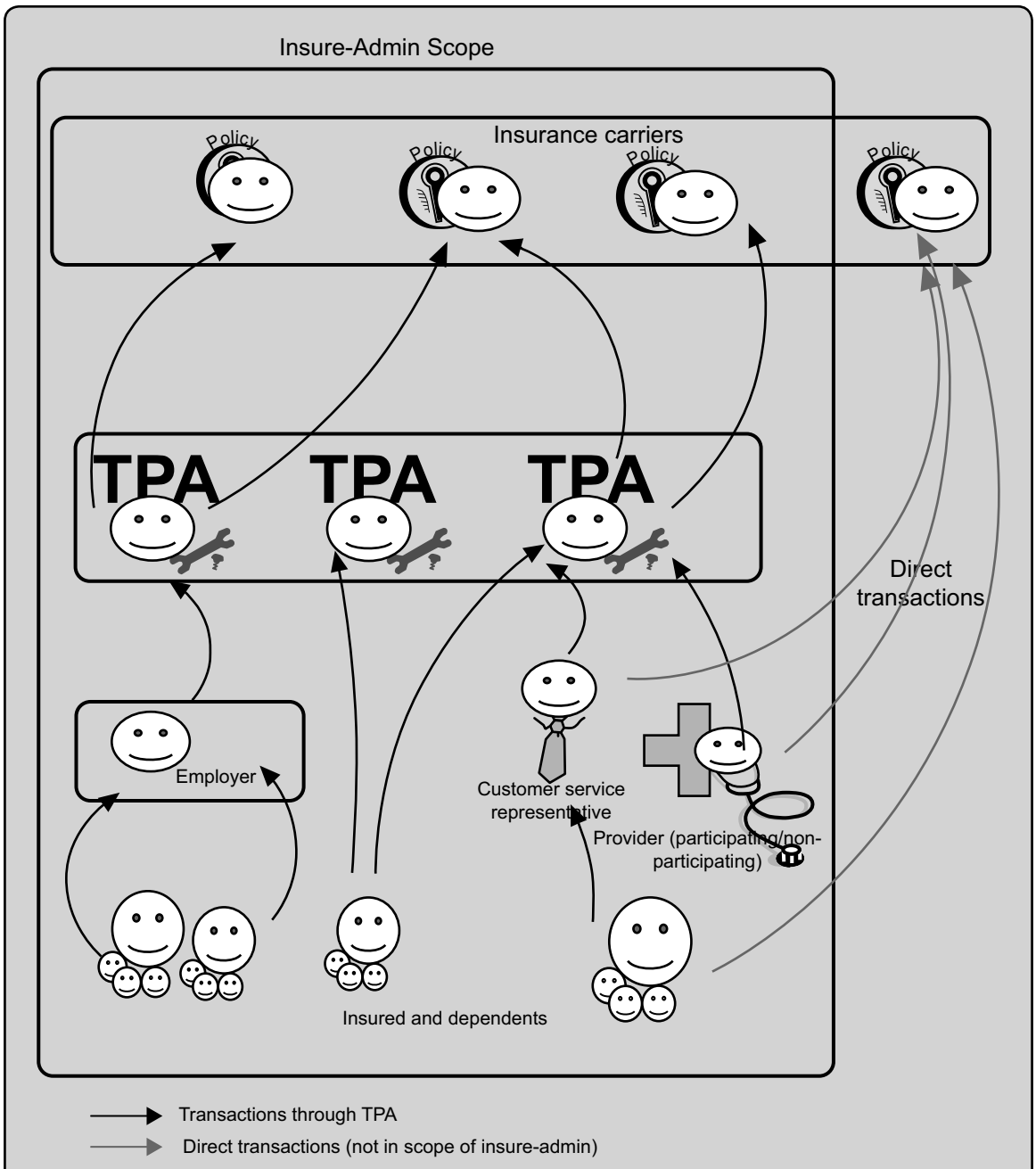


Figure 2.1 Work flow and business logic in the health insurance sector involving a TPA

on Once found eligible, the applicant is enrolled for insurance and his/her billing starts. It is the job of the TPA to generate billing at regular intervals in accordance with the policy agreement with the customer and collect these bills on behalf of the insurance company. Claims are handled either directly by the insurance

company or by the TPA depending on the service contract and/or the business policy of the TPA (in the case of PayMinders, claims are handled by the insurance companies directly).

In order to get the best deal, customers spread insurance coverage over a number of insurance carriers, where, for example, basic health coverage may come from policy X from company A, dental coverage can come from policy Y from company B, and so on. A TPA may handle all the accounts of a customer and/or may be handling accounts for a number of insurance companies. TPAs thus act as the cushion between the consumers (individuals or employer organizations) and the carriers.

The typical business logic is depicted in Figure 2.1.

For more information on organization and functioning of the health insurance sector in the United States see http://www.bls.gov/oco/cg/cgs028.htm.

For a glossary of terms used in the health insurance sector, see http://www.pinnacletpa.com/terms.html and http://www.siho.org/en/common/glossary.asp?section=Adtors.

Development Journey

The VBU-HALI team followed the standard Fact-Tree development methodology. A team of 20 associates was initially divided into an offshore team and an onsite team. A six-member analyst team went to the client's site to understand the requirements. This was the same team that was associated with the development of the previous project and had moderate understanding of the application. This onsite team discussed the required functionality with the client representatives and successfully generated a user requirement document (URD). The onsite team also helped the offshore team understand the requirements. The knowledge transfer process from the onsite team to the offshore team was not easy, as most of the members of the offshore team did not have any prior knowledge of the application. This resulted in the onsite team capturing and internalizing most of the information among themselves and passing on very little to the offshore team.

Glossary

Standard Fact-Tree Development Methodology

Every mature organization has its own development methodology—typically adopted from standard practices. This methodology consists of processes and guidelines supported by technology tools such as workflow and project management software.

Major Modules

The requirement-gathering phase took nearly a month at the client's site and the analyst team divided the entire application into six modules. Each member in the analyst team later became a module leader for each of these modules as they are more knowledgeable about the client's requirements of their respective modules. All the module leaders were at the same level in managerial hierarchy. Hence, the team composition resulted in a flat structure with no one in charge of all the modules, resulting in every module being handled independently. This resulted in a situation where all the modules were being developed in parallel.

The six modules are as follows:

- *Eligibility*. This module tracks and maintains information about those insured and information about the eligibility of dependents. All the interactive eligibility changes made by the user at client and group levels are considered in this module. Other modules such as the call tracking, letter issuance, group billing and individual billing make use of this module for their functioning.
- *Billing and accounting*. This module performs the task of calculating the premiums and arrears that are to be collected from the client. This was one of the most complex modules, as it uses information from the eligibility module and there are very complex rules defined by the client that would guide the calculations. This billing was done at two levels, individual as well as group.
- *Customer services*. This module provides Web-based self-help services to all the end customers of the TPA, that is, those insured, the employers and the medical facility providers.

- *Security*. This is a role-based security module. Apart from defining various security levels, the interfaces are also defined based on the administrative settings. For example, the user functionality may be set at group or individual level. The security module logged all activities of the users. Thus, the module consisted of authentication, authorization and audit trailing.
- *Letter issuance and call tracking*. This module generated various kinds of letters for various parties, such as collection agencies, insurance companies, and so on, who are involved in the business.
- *Reporting*. This module generates various types of reports according to the authorization.

Eligibility was one of the most important modules and almost all modules used information from it. In order to make the product highly customizable, each module consisted of a number of user-defined information fields that could be defined by the TPA.

Since an initial product was already under development, by the time all the requirements were gathered from the client some parts of the project were already implemented. The URD developed consisted of the requirements given by MidAlliance in particular, but the requirements given by PayMinders were also incorporated in a different format. This was done because the primary aim was to develop a generic product first and then to customize it as per PayMinders' requirements.

'We already developed a part, but there was no verification that no mismatch occurred between the PayMinders requirements and the product already developed', commented one of the developers.

It was decided earlier that the team would follow the waterfall approach for this project. One of the reasons for following the waterfall approach was that the requirements were completely known. Thus, once the requirements were written and signed by the client, the team decided to draw the architecture diagram. The architecture diagram as drawn by the team is shown in Figure 2.2.

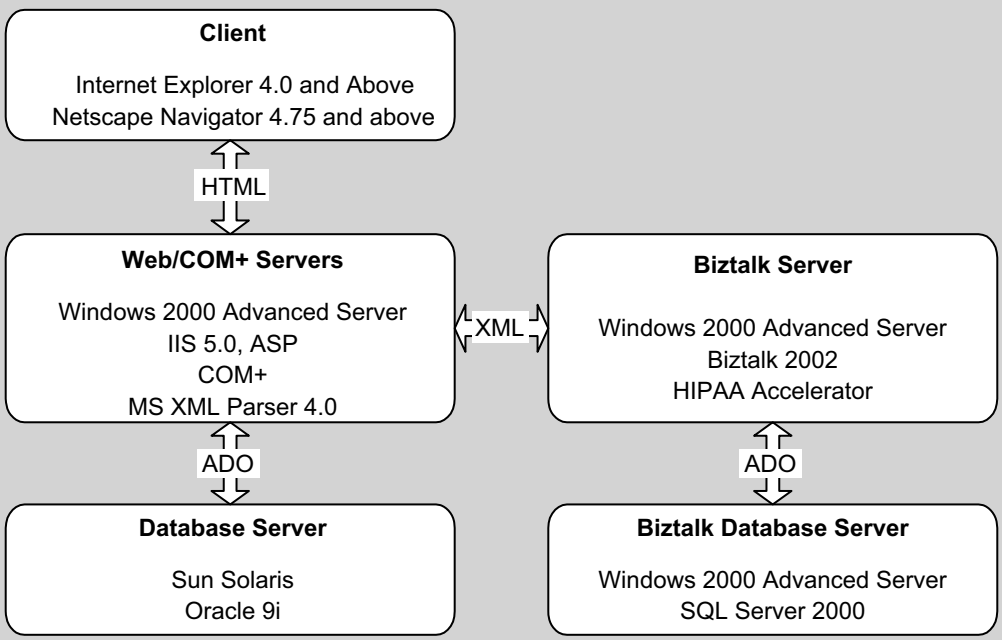


Figure 2.2 Systems architecture for Assure-Health

It was also decided that they would follow a three-tier architecture. That is, the system was divided into three tiers: presentation tier, middle tier and database tier. The presentation tier was a thin client, presented on the client's machine through HTML embedded on ASP. The business logic and middleware were bunched into the middle tier. It was decided that the database will be accessed only through data access objects, which would be created in the middle tier. The application server, Web server and component repository were integrated into the same machine for ease of maintenance.

A separate BizTalk server was used mainly for compliance requirements of the Health Insurance Portability and Accountability Act (HIPAA). The HIPAA was enacted by the US Congress in 1996, and it is mandatory for all medical systems to follow the policies recommended by HIPAA. The data access method for the BizTalk server followed the same concepts.

Prototyping

As MidAlliance knew the domain well and they worked very closely with the client, it was decided that they would develop a static prototype of all the screens and get it signed-off from the client. The prototype was developed as a set of screens that incorporates all navigations but without any business logic. The navigation flow between the modules was defined and the flow was exhibited through Web links between the modules. But the data structures needed for input and output of different modules were not frozen. The data model was not ready because the characteristics of data in the generalized version were different from the requirements given by PayMinders. Another problem of a static prototype was that no integration plan was made for different modules at that point in time, that is, if one module was supposed to trigger a set of activities in another module, this triggering action was missing and hence the modules could not be integrated. Obviously, if the prototype contained everything, there would be no difference between the prototype and the actual system.

A prototype is a handy tool for estimation as well. All the estimations were entirely based on the number of screens. The client (PayMinders) assumed that a COTS (commercial off-the-shelf) product was available and this product would be customized for them. So the estimation team was convinced that effort estimation as well as effort distribution could be based only on the number of screens. They considered screen complexity but failed to understand the complexity of the business logic behind those screens. This estimation was also influenced by the requirement of the client that the system should be delivered within a specified time limit (Exhibit B).

Exhibit B List of Key Requirements from the Eligibility Module

Eligibility module is one of the core modules in the Assure-Health application. This module assumes that the eligibility of the insured and his dependents is already decided and provides the Web interface to add or edit this eligibility in the system.

- To add, delete, modify insured information
- To add, delete, modify dependent information by the insured
- Make sure that the eligibility check is performed before adding insured or dependents
- Search and report facility

Communication Protocol

To ensure proper and complete communication between the various parties, various communication channels were established between the end user and the development team. Conference calls served as the

means of communication. MidAlliance was the middle customer for VBU-HALI and all the communication was usually through them. Typically, VBU-HALI would send documents to MidAlliance for verification and review and then MidAlliance would forward them to PayMinders. But this was not a very rigid rule. Sometimes PayMinders talked with VBU-HALI directly, when the need arose, to convey requirement changes or additions.

Further in the Development Cycle—Into Requirements and Design

All developers in the group have a thorough knowledge of QMS and all quality processes were followed. The URD and the Software Requirement Specification (SRS) document were created as per procedure. Each module was described in a different URD and SRS. The offshore team performed the Herculean task of preparing the documents. This was more like an acceptance document, that is, it simply captured the user needs, and whose fulfilment will make the system acceptable to the end user. Hence, the URD was supposed to map acceptance testing. The business development manager from MidAlliance verified the URD and this URD was used to prepare the developer's version of the requirements, which is the SRS.

Glossary

Quality Management System (QMS)

Every organization defines a quality management system based on its own context. QMS is a set of policies, processes and procedures that aid in planning and executing core businesses.

The SRS captured all the functionalities expected from the system being developed. It contained the software requirements and the functionalities, as well as the non-functional requirements. It even incorporated the screen shots that described the interface as well as the functionalities. These screen shots were a part of the prototype.

Some of the additional requirements that were not specified by the client were also identified, as it seemed impossible to provide the user-specified functionalities without including them. The VBU-HALI team found them necessary for the proper functioning of the whole system. The SRS was supposed to be used as a map to system testing.

The design of the system was purely based on the SRS documents and they were used to create the high-level design documents. The high-level design documents were verified by MidAlliance and were then forwarded to PayMinders for final approval.

Data Modelling

The data model for Assure-Health was not properly defined as the data structure of the client was different from the base product. The modules were designed to interact with each other by means of the database only, that is, each module writes to the database and the other modules read from the database, and there is no direct communications between modules.

This involved extensive data transactions between the Web pages and the database. XML was chosen as the language for this arduous task. Though XML had a severe limitation, that is, any change in the data model would directly affect the functioning of the application, it was the most efficient solution due to the presence of a number of user-defined fields.

A copy of the database, known as the 'pure copy', was kept untouched, and there were working copies of the database. Only the MidAlliance DBA, as per the requirement of the client, was authorized to change the 'pure copy'. Whenever new records were added (during development or testing) to the current version of the database, the data were validated and synchronized with the other working databases. As the data model evolved with time, a number of change requests were coming in regularly from the client about the data fields, and the MidAlliance DBA was incorporating them in the 'pure copy' database schema. Every week the offshore database schemas were synchronized with the 'pure copy'.

Coding and Testing

As Assure-Health was a Web application and effort distribution was done screen-wise, each developer tried to code the functionality of the given screen keeping in mind the data that were needed to be displayed.

The prototype (all the user screens without the functionalities) was present and the developers had a fair understanding of the data fields. Developers went into the coding phase directly from the user interface (given as prototypes) and each developer took ownership of a set of screens and worked independent of each other.

Testing was also taken very seriously as the team realized that the scope and extent of the project was huge. For each module, a separate test team was formed comprising people other than the code owners, that is, the developers. This team not only developed the system and integration and acceptance test cases but also wrote the unit test cases independently. Most of the testing was done manually and test automation tools were not being used initially. This idea of code owners not performing unit testing was not accepted by many.

Everything seemed to be going smoothly as the development started on time and the teams were reporting constant progress. Though frequent requirement changes were coming in, no danger signal was raised because this is a common scenario in all projects.

The team was very busy with development. Whenever PayMinders found some divergence from the needed functionality, or a new idea crossed their minds, a change request was issued by them. PayMinders or MidAlliance would simply change the prototype and send it to VBU-HALI for implementation. Hence, the prototype was continuously changing and evolving.

Whenever developers needed a modification in the underlying data model due to changes in prototype, they would simply request the DBA and get the changes done. Hence, the data model was also evolving continuously.

Now It Is Time to Deliver

The deadline was approaching fast. The work was in full swing. But the developers realized that no matter how hard they worked they would not be able to deliver the application on time. The difference between the initial understanding of the requirements along with the effort and time estimation to meet these requirements and the current expectations of the client was brought to the attention of every one. When the initial estimation was done, the basic assumption was that the project was to customize an existing product for a client. But as new issues and new requirements and requirement changes (so many of them) emerged, the project scenario changed completely. Now it turned out that VBU-HALI was actually developing a totally new application for PayMinders.

PayMinders wanted a major change in the *Reporting* module. They asked for a change in the structure of the report and also wanted that users be able to change the report structure dynamically. This required a change of tools to be used for reporting. They also asked for generation of online reports in place of a batch process generating all the reports during off-hours. The analyst team failed to understand the effect of this request and accepted it initially. But within a short span of time it was evident that with the current architecture and development model it would be a tough job for the developers.

> 'Changing the reporting module entirely with the same architecture is like asking too much. Had we followed OOAD instead of SSAD, the job would have been easier', exclaimed Rita. She was one of the developers and was closely watching the project.

At this point in time, the realization dawned that the delivery that VBU-HALI was working on was not same as what PayMinders was expecting.

After taking stock of the situation, the team management figured that the product could not be delivered on time by any means. Immediately the delay in delivery was conveyed to MidAlliance. This information was not conveyed by MidAlliance to PayMinders as they expected it might trigger PayMinders to stop the payment of the invoice already submitted. Unfortunately, VBU-HALI was not aware of this communication gap.

It was just a week before the scheduled delivery date and during a conference call was the delay exposed. PayMinders was shocked to know that the release was not happening and the project manager was astonished to know that PayMinders did not have any information about the delay. PayMinders was under the impression that the first delivery was being made on time and hence they were busy making preparations for testing the system and training its staff on the new application. Knowing that the delivery was getting late, PayMinders got upset.

During the same conference call, one of the senior managers of PayMinders said, 'OK, we understand that the delivery is delayed, but this information should have come to us early, not just a week before the scheduled delivery date'.

PayMinders was upset because they had scheduled training and other activities accordingly, and now it was all a sheer waste of their efforts and money. This became a point of contention.

Finally, VBU-HALI gave its first delivery to PayMinders, late by several days. After the first delivery, it was very clear to everyone that the requirements still needed to be worked on. The gap between the expectation of the client and the actual system delivered came into light and the criticality of the situation made everyone very unhappy.

Fact-Tree, including VBU-HALI, follows a system where the higher management can be contacted in case of a serious problem or issue between the development team and the customer. This process is known as 'escalation'. The PayMinders authorities reported the prevailing conditions to the VBU-HALI higher officials.

Immediately, the higher management sprang into action and as part of the crisis management process a PMO was formed. The PMO consisted of three members. They investigated the cause for this escalation and found that PayMinders was actually reporting a number of problems. The major categories of problems that were reported by PayMinders included the following:

- *Communication gap***.** The communication channel was extremely bad and there was lack of proper communication between the development team and the client. The most prominent example of this was the lacuna on behalf of MidAlliance when they did not convey the delay in delivery.
- *Late on schedule***.** It was very clear that the first delivery was delayed and in all possibility this was going to affect all future deliverables.
- *Poor quality*. The current implementation delivered was found to be of poor quality in terms of number of bugs. PayMinders found that the number of bugs was high and attributed it to a lack of system testing. There was much discussion that system testing had not been done and PayMinders informed that they had found some defects that would not have been there had system testing been done.

Glossary

Project Management Office (PMO)

In a typical organization, PMO is the department that defines and maintains project management related standards of process. The PMO ensures that all the projects that are undertaken by the organization are standardized so that the repeated execution of the projects will be economically beneficial to the organization. PMO provides documentation, guidance and metrics on the practice of project management and execution. In case of emergency or crisis, the PMO steps in to analyse the situation and comes up with a recovery plan.

User Requirements Document (URD)

The URD is a mandatory outcome of the user requirements definition phase, which needs to be produced before the software project begins. The URD captures the general description of what the user wants the software system to perform, all the operations users want to perform with the software system, all known user requirements, all constraints and the external interfaces to the software system.

Software Requirements Specification (SRS) Document

SRS captures the complete description of the behaviour of the system to be developed. It includes a set of use cases that describes all the interactions that the users will have with the software.

Earlier during the signing of the project, MidAlliance told PayMinders that the product already existed and that only customization was to be done. Hence, it was not possible for VBU-HALI to inform PayMinders that system testing was not carried out as it is done for a product.

System testing was done on the requirements mentioned in the URD and SRS. The large number of changes requested by the client and MidAlliance were not included in the URD and no one thought about synchronizing the SRS with the changed prototypes. These factors created the gap between the observations of the system testers and the expectation of the client.

In addition, PayMinders was finding it difficult to install the delivered system. When the team analysed it, they uncovered many problems relating to requirement changes, data model changes, system integration and performance. Some of the changes are shown in Exhibits C and D.

Exhibit C Technical Analysis of the Assure-Health Development Problems

When the product was perceived to be nearing completion, there was a sudden downpour of change requests from PayMinders. A total of 82 change requests came in, covering almost the entire application. But most of them belonged to the accounting and billing module, on which almost all the other modules were dependent.

When bugs were analysed, it was found that the major share of defects was related to the requirements, either as 'incomplete requirements' or 'clarification in requirement'.

The development team found a serious problem with the integration of the various modules. For example, a serious defect became apparent when an insured was terminated. Termination meant that the premium that was already paid by this insured was to be adjusted or refunded. This had a wide affect on almost all the modules including letter issuance and call tracking and billing and accounting. Thus, whenever any feature in eligibility module was changed, a change in the billing module became necessary, but this was not happening and hence lots of rework was needed to remove these creeping defects. Any such little defect involved rework in terms of modification of database, modification of business logic and change in interface as well.

Another major integration problem was that though, by design, all the modules were supposed to talk to each other only through the database, it was found that some modules were communicating directly. Sometimes the screens were talking directly using the XML string as this was figured as a simpler and efficient communication technique than involving the database at each step. The ad hoc communication between the modules was leading to major integration problems.

It seemed that there was no end to the problems related to the interface and integration. Some problems were pretty interesting. For example, a bug came up where any change in the family size of the insured was not affecting the premium to be paid by the insured, when it should have. This bug was assigned to Steve. He was the owner of the billing module. After long analysis he reported that due to a change request, a new field of family size was added to the eligibility module. Other module owners were not aware of this. Naturally, in the billing module Steve did not make use of this field for calculating premium. Then Steve added the new field in his module, modified his logic and modified the queries to get the proper premium.

Some performance issues also came up. The pages were slow to load and process. Developers attributed it to the use of XML for the underlying data transaction between the screens and the database. XML was being used for maintaining page status between the navigation as well. Every time the page was to load, an intermediate step of parsing of the XML was to be performed, hence slowing the system considerably.

Exhibit D Examples to Showcase Some Problems Faced by the Development Team

Mismatched Interface Between Different Modules

The interface between different modules was not documented anywhere. The data model was always changing. In this scenario, developers took the liberty to define the interfaces for their respective module as per their convenience and easiness in implementation. As the modules were independent in nature, it was working fine. There was no complaint in the initial stages of development.

But at later stages of development, the deficiency in this approach surfaced when two modules named 'Enrollment' and 'Termination' were being integrated. In all the modules XML was used to store intermediate data across html pages and the same XML at the end to update the database. This XML schema was not documented anywhere and the schema had been changed frequently as and when required. Earlier, the 'Termination' program, which was developed by a different developer, used its own XML structure to store the intermediate data between pages and update the database, which is given below:

```
<TerminateInsured>
    <InsuredDemographicsInfo/>
    <InsuredEmploymentInfo/>
    <DependentDemographics/>
    <SelectedCoverageAndPlans/>
</TerminateInsured>
```

Later, when the termination module wanted to use some functionality (e.g., calculation of premium/adjustments) of the enrollment module, it could not communicate to the enrollment module and an error message appeared, 'Enrollment Node not found in the Input XML'. The problem occurred due to different XML structures being used by the Enrollment module for the same record, as given below:

```
<Enrollment>
    <InsuredDemographics_page1/>
    <InsuredDemographics_page2/>
    <DependentDemographics_page1/>
    <DependentDemographics_page2/>
    <IncomeInfo/>
    <CoverageAndPlans/>
</Enrollment>
```

Once we found out this problem, now the point was which XML structure to be changed so that the impact on the whole project will be minimal. The owner of Enrollment module defended her XML structure saying that this is the structure used in the data model throughout the system. The same thing was repeated by the owner of the Termination module. Finally, it took nearly 3 days after a long discussion with all module leaders to come up with a solution that an XML converter be used to solve this problem as there was no time to change the XML structure in any of the module, which will have effects on the complete system.

A number of similar problems ware observed later while fixing bugs reported by the system tester and in most of the cases a quick fix was used to avoid the problem.

Prototype Changes Resulted in Design Changes

Background. The security module was used to create new users with different roles and privileges (Add/View/Update/Delete). These permissions were given at the resource level, that is, to each hyperlink that is displayed to the user. It is also used to decide the newly created users' access permission to different resources/functionality.

Previous design. While using the Create New User function, the user who has only Add and View permission could create a new user having all the permissions (Add/Update/Delete/View). The permissions were given as per the user policy of the system.

When the first phase is delivered, the customer came back with the following objection: how can a user who does not have Modify/Delete permissions create a new user with the same permissions. This is against the security policy.

Prototype change. The customer sent a change request mentioning that 'when the new user is created by the user who has only (Add, View, Update) permissions, then only those (Add, View and Update) permissions will be carried over to the newly created user'. The customer also sent a changed prototype to display only those access permissions of the new user with check mark in their access permission fields, which is owned by the user creating the new user.

New design. This became a major problem because during the design there was no facility of tracking the permissions of the user who is creating the new user. Another problem was that if a new user is to be created with all permissions, then it can be done only by a user who has all permissions. We needed to add some more screens to distinguish between different log-ins with the user permissions.

Changes in Database Were Not Reported in Data Model

During the development process, the developers altered some of the database tables, that is, added or deleted some new column, to add some missing functionality or to accommodate the change requests from the customer. The module leaders who were having the permission to do so did all these changes to the database without changing the data model or without informing other module leaders in the development team and the DBA. As the modules were being developed independently, there was not much of a problem till the integration of the modules began. While integrating these modules, it became a nightmare for the DBA to accommodate all these changes in the database structure, which was done by the module leaders. Some places the modules that were working before started giving problems due to these changes.

For example 'FamilySize' was one field that was added to the 'Insured' table later to consider it as a billing factor. The Enrollment module leader changed the database table and changed the corresponding data access component (DAC is a COM component DLL). After testing by the Enrollment module team the component was sent to the client.

The other components, which were dependent on this DAC, failed because they were referring to the older version of the DAC. Besides, even the billing module did not consider 'FamilySize' as a billing factor for calculating the coverage premium. Impacts of this small change effected the whole application as 'Insured' DAC was used widely throughout the application.

Change of a Simple Database Rule

The project requirement indicated that only one ID (unique ID) would be used for each insured and its dependent members. Accordingly, the design was made and the code was implemented. Just a few days before delivery, the client informed that the same insured can register in different groups and so there is

a possibility that different IDs will be allocated to the user. This seemed like a simple change and was accepted by the Change Request group. This was also not clashing with any of the business rules because the rules for different groups were independent in nature. The module leaders accepted and implemented the same and it passed the unit testing.

While doing the system testing, the following was found: if there is a search 'Insured', multiple records are displayed having exactly the same information (such as name, addresses, social security numbers). The user does not know which one is to be selected for the purpose of the current activity.

Finally, it was decided that the design of the prototype and the functional requirement be changed to solve this problem. A new master table was designed which will store multiple IDs created for a particular insured along with a primary ID. The insured need to know only the primary ID, whereas the system uses the secondary IDs for different operations used in the system. The prototype was also changed because a new page was required where the secondary IDs will be displayed so that the user can select the correct record. A new function was also added to manage the primary and secondary IDs.

Solve this Case and Save the Project

Assuming that you are part of the PMO that wants to save the project, what are the actions you would suggest? Focus on the technical aspects of the case rather than on managerial elements.

Start with analysing the root causes that led to the escalations. The following are some guidelines to help you analyse the problem better and come up with a solution—take them only as starting points.

1. What phases of a typical software development life cycle were executed strongly and which phases were relatively neglected?
2. What is the role of architecture and architect in this project?
3. What are the requirements elicitation, analysis, specification and engineering methods that were followed and what methods should have been followed? What is the relationship between requirements engineering and architecture?
4. Come up with a complete architecture of the system with all possible viewpoints and explain how these architecture viewpoints help avoid many of the communication gaps that are identified during the postmortem analysis by the PMO.

POSTMORTEM

This case study can be analysed from different viewpoints. For example, it can be analysed from the requirements engineering point of view, from the project management point of view or from the software development methodologies point of view. From the requirements engineering point of view, this case study brings out issues such as the following:

- Managing the requirements of custom application (project) versus that of a customizable product.
- Verification and proper sign-off of the requirements by all the parties concerned, not just because the process demands it but because of the need for complete understanding of the implications of accepting the SRS and the URD.
- The need for one person to have a bigger picture of the application from the beginning, even if individual teams work at specific sub-system levels. Unless there is coordination between the

teams and unless there is understanding of how various parts of the system combine to function as one well-integrated system, it will be impossible to build the desired application.

- Traceability matrix and change control board become crucial in any environment where the initial perceived requirements are changing. Scope creep can controlled be using the above-mentioned methods.
- Prototypes are important to understand the proposed system. It is important not to stretch the prototype to the actual application.

Similarly, from the project management point of view, the following issues arise:

- Establishment and management of effective communication channels between all stakeholders.
- Understanding and dealing with dangerous 'informal' communication channels such as the client directly requesting the programmer to make a correction over the telephone.
- Making sure that all the groups and individual contributors understand their roles and responsibilities. It is also important to make sure that everyone in the team understands what is expected of the other members and group.
- Importance of involving all the stakeholders while making crucial decisions.
- The role of domain experts in the development cannot be overstated. Their contributions will be the success factor in making the software ultimately useful to the end users.
- Using the right estimation techniques so that project planning can be done more accurately.

Since this book is focused on software architecture, let us look at it from a purely software architecture perspective and analyse it. We will learn about the importance of creating a technical architecture by analysing this case study and we should be in a position to propose an architecture at the end. There are typically several solutions for any given case study. In fact, this is what makes a case study a very potential tool for learning. The goal of this analysis is not to provide 'the' solution or 'a' solution but to help the reader arrive at his or her own solution. We will provide one good solution or a model solution in the Web site of this book. Hence, let us not worry too much about the specific architecture we come up with, instead let us focus on the process of designing the right architecture.

This postmortem by the PMO team (see Exhibits C and D) cries for a well-designed architecture. Almost every problem cited is a result of lack of architecture. These are problems of improper integration, improper interfaces, poor performance and bad design. Some of these problems include the following:

1. Mismatched interface between different modules
2. Prototype changes resulting in design changes
3. Changes in database not being reported in the data model
4. Change of a simple database rule

Each of these problems has resulted because to the lack of an architectural roadmap that guides the development of the application. A proper architecture defines how various modules communicate with each other. Architecture is concerned about minimizing the coupling between modules and maximizing the cohesion of each module. It makes sure that all interfaces between modules are properly designed.

It is hence clear that had the project started with a proper architecture many of the development problems could have been avoided. Why is software architecture perceived as providing a solution to the inherent difficulty in designing and developing large, complex systems?

WHY IS SOFTWARE ARCHITECTURE IMPORTANT?

As suggested by Clements and Northrop (1996), fundamentally there are three reasons why software architecture is important:

1. Software architecture represents a common high-level abstraction of the system that most, if not all, of the system's stakeholders can use as a basis for creating mutual understanding, forming consensus and communication.
2. *Early design decisions*. Software architecture represents the embodiment of the earliest set of design decisions about a system. These early bindings carry weight far out of proportion to their individual gravity with respect to the system's subsequent development, its service in deployment and its maintenance.
3. *Transferable abstraction of a system*. Software architecture embodies a relatively small, intellectually graspable model about how the system is structured and how its components work together. This model is transferable across systems; in particular, it can be applied to other systems exhibiting similar requirements and can promote large-scale reuse.

Communication

The case study reflects all the above-mentioned reasons clearly. Let us focus on communication first. Whether you blame the crisis situation on poor project management or on lack of appropriate requirements engineering practices, it is clear that there were no proper communication channels between all the stakeholders of this project. Architecture is the vehicle for stakeholder communication.

Typically, there are several stakeholders for the software system, such as customers, actual end users, project managers, higher management, developers and testers. Each of these stakeholders looks at the software architecture from their perspective or point of view. They will be looking for answers to their own questions. For example, end users want to make sure that the system is available and reliable in addition to providing the required functionality; the sponsor or customer wants to make sure that the system is delivered on time and within budget and the project manager wants to make sure that the given architecture can be realized in a manner that reduces interaction among the teams allowing them to work more or less independently in addition to worrying about the budget and time factors.

The developer is worried about strategies to achieve all of these goals. Architecture provides a common language in which competing concerns can be expressed, negotiated and resolved at a level that is intellectually manageable even for large, complex systems. Without such a language, it is difficult to understand large systems sufficiently to make well-informed early decisions that greatly influence their quality and usefulness.

Architecture also embodies the earliest set of design decisions about a system. These early decisions are the most difficult to get right, are the hardest to change and have the most far-reaching downstream effects:

- *Architecture imposes constraints on implementation*. Architecture constrains the implementation either directly or indirectly. Systems that are good implementations of well-designed architectures are said to exhibit architectural decisions relating structural decomposition of the system even when the user is just experiencing the system without seeing any of the architectural documentations. The implementer will have to build components according to the structural decomposition given by the architecture. As the architecture decomposes the system into several sub-systems and each sub-system is captured by a set of related components, each component must fulfil its responsibility by appropriately interacting with other components. It is the architecture that determines this interaction. Some of these architectural decisions may further constrain the platforms, choice

of components and even programming languages. Typically, the system-wide or project-wide decisions that are being made will not be visible to individual component implementers. This gives rise to several benefits, such as allowing management decisions that make best use of personnel. Component builders must be fluent in the specification of their individual components but not in system trade-off issues and, conversely, the architects need not be experts in algorithm design or need not understand the intricacies of the programming language.

- *Architecture has major influence on development team composition*. Not only does architecture prescribe the structure of the system being developed but that structure becomes reflected in the work breakdown structure and hence the inherent development project structure. Teams communicate with each other in terms of the interface specifications with major components. This influence remains throughout the software development life cycle and beyond. The maintenance activity when launched will also reflect the software structure, with maintenance teams formed to address specific structural components.
- *Architecture is the bottleneck in achieving the system's targeted quality attributes*. Whether or not a system will be able to exhibit its required quality attributes is largely determined by the time the architecture is chosen. The key architectural decisions are all about prioritizing the quality attributes and designing ways to achieve them. Quality attributes may be divided into two categories. The first includes those that can be measured by running the software and observing its effects: performance, security, reliability and functionality. The second includes those that cannot be measured by observing the system but rather by observing the development or maintenance activities. This includes maintainability in all of its various flavours: supportability, adaptability, portability, reusability, and so on. Modifiability, for example, depends extensively on the system's modularization, which reflects the encapsulation strategies.
  - There is always a relation between the *reusability* and *coupling* of components. If a component is highly coupled (i.e., there are too many dependencies between various components and every component has to call other components many times to deliver its functionality), it is not reusable. The realistic scenario in today's software development is distributed processes that are physically spread across machines and regions. If there is a huge volume of inter-process or inter-component communication, obviously, the performance of such systems gets affected severely.
- It is important to understand, however, that architecture alone cannot guarantee the functionality or quality required of a system. Poor downstream design or implementation decisions always undermine an architectural framework. Decisions at all stages of the life cycle, from high-level design to coding and implementation, affect system quality. Therefore, quality is not completely a function of an architectural design. A good architecture is necessary, but not sufficient, to ensure quality.
- *Architecture helps in predicting certain qualities about a system*. if the architecture allows or precludes a system's quality attributes (but cannot ensure them), is it possible to confirm that the appropriate architectural decisions have been made without waiting until the system is developed and deployed? If the answer is 'no', then choosing an architecture would be a fairly hopeless task. Random architecture selection would suffice. Fortunately, it is possible to make quality predictions about a system based solely on the evaluation of its architecture. Architecture evaluation techniques such as the software architecture analysis method (SAAM) proposed at the Software Engineering Institute obtain top–down insight into the attributes of software product quality that are enabled (and constrained) by specific software architectures. SAAM proceeds from the construction of a set of domain-derived scenarios that reflect qualities of interest in the end-product software. This set includes direct scenarios (which exercise required software functionality) and

indirect scenarios (which reflect non-functional qualities). Mappings are made between these domain scenarios and candidate architectures, and a score is assigned depending on the degree to which the candidate architecture satisfies the expectations of each scenario. Candidate architectures can then be contrasted in terms of their fulfilment of their scenario-based expectations (Abowd *et al.*, 1994; Clements *et al.*, 1995; Kazman *et al*., 1995).

- *Architecture can be the basis for training*. The structure in addition to a high-level description of how the components interact with each other to carry out the required behaviour often serves as the high-level introduction to the system for new project members.
- *Architecture helps to reason about and manage change*. The software development community is finally coming to grips with the fact that roughly 80 per cent of a software system's cost may occur after initial deployment, in what is usually called the maintenance phase. Software systems change over their lifetime; they do so often and often with difficulty. Change may come from various quarters, including the following:
  - *The need to enhance the system's capabilities*. Software-intensive systems tend to use software as the means to achieve additional or modified functionalities for the system as a whole. Systems such as the joint stars battlefield surveillance radar and the MILSTAR network of telecommunication satellites are examples of systems that have achieved enhanced capability through software upgrades. However, with each successive change, the complexity of the system software has increased dramatically.
  - The need to incorporate new technology, whose adoption can provide increased efficiency, operational robustness and maintainability.

Deciding when changes are essential, determining which change paths have the least risk, assessing the consequences of the changes proposed and arbitrating sequences and priorities for requested changes all require broad insight into relationships, dependencies, performance and behavioural aspects of system software components. Reasoning at an architecture level can provide the insight necessary to make decisions and plans related to change. More fundamentally, however, architecture partitions possible changes into three categories: local, non-local and architectural. A local change can be accomplished by modifying a single component. A non-local change requires multiple component modifications but leaves the underlying architecture intact. An architectural change affects the way in which the components interact with each other and will probably require changes throughout the system. Obviously, local changes are the most desirable, and so the architecture carries the burden of making sure that the most likely changes are also the easiest to make.

Architecture as a Transferable Model

Greater benefit can be achieved from reuse if applied earlier in the life cycle. While code reuse provides a benefit, reuse at the architectural level provides a tremendous leverage for systems with similar requirements. When architectural decisions can be reused across multiple systems, all the early decision impacts described above are also transferred.

Product line architecture. Product lines are derived from what Parnas referred to in 1976 as program families (Parnas, 1976). It pays to carefully order the design decisions one makes, so that those most likely to be changed occur latest in the process. In an architecture-based development of a product line, the architecture is in fact the sum of those early design decisions. One chooses an architecture (or a family of closely related architectures) that will serve all envisioned members of the product line by making design decisions that apply across the family early and by making others that apply only to individual members later. The architecture defines what is fixed for all members of the product line and what is variable.

The term domain-specific software architecture (DSSA) applies to architectures designed to address the known architectural abstractions specific to given problem domains. Examples of published domain-specific software architectures come from the ARPA DSSA program.

Restricting the vocabulary of design alternatives. When it comes to program cooperation and interaction, we gain by voluntarily restricting ourselves to a relatively small set of choices. Advantages include enhanced reuse, more capable analysis, shorter selection time and greater interoperability.

Architecture permits template-based component development. Architecture embodies design decisions of how components interact, which while reflected in each component at the code level can be localized and written just once. It is possible to build complex systems by importing externally developed systems that are compatible with architectures that have already been defined. Architecture permits the functionality of a component to be separated from its component interconnection mechanisms.

ROLE OF ARCHITECTURE IN SOFTWARE DEVELOPMENT

What activities are involved in taking an architecture-based approach to software development? At the high level, they include the following, which are in addition to the normal design program and test activities in conventional small-scale development projects:

- Understanding the domain requirements
- Developing or selecting the architecture
- Representing and communicating the architecture
- Analysing or evaluating the architecture
- Implementing the system based on the architecture
- Ensuring that the implementation conforms to the architecture

Each of these activities is briefly discussed below.

Understanding the Domain Requirements

Since the primary advantage of architectures is the insight they lend across whole families of systems, it is prudent to invest time early in the life cycle to conduct an in-depth study of requirements, not just for the specific system but for the whole family of systems of which it is or will be a part. These families can be the following:

- A set of related systems, all fielded simultaneously, that differ incrementally by small changes
- A single system that exists in many versions over time, the versions differing from each other incrementally by small changes

In either case, domain analysis is recommended. Domain analysis is an independent investigation of requirements during which changes, variations and similarities for a given domain are anticipated, enumerated and recorded for review by domain experts. An example of a domain is command and control. The result of domain analysis is a domain model that identifies the commonalities and variations among different instantiations of the system, whether fielded together or sequentially. Architectures based on domain models yield the full advantage of architecture-based development because domain analysis can identify the potential migration paths that the architecture has to support.

Here, we have emphasized on having a domain expert working with the development team members throughout the software development phase. However, identifying and recruiting domain experts who can work with software engineers is always a challenge. This especially becomes more difficult if the organization does not have any understanding of the domain. Most of the large organizations have realized this problem and are coming up with various innovations to address this issue. We are aware of cases where medical doctors or insurance agents with huge expertise in their field are part of the development team. Software development organizations recruit them and train them on software development issues. Another way to resolve this issue is to involve the client in a more structured way with a very well defined role, assuming that client has the domain expertise (like MidAlliance in our case study).

Developing (Selecting) the Architecture

While some advocate a phased approach to architectural design, experience has shown that the development of an architecture is highly iterative and requires prototyping, testing, measurement and analysis.

Clements (1996) in his work 'A Survey of Architectural Description Languages', posits that architects are influenced by factors in three areas:

- Requirements (including required quality attributes) of the system or systems under development.
- Requirements imposed (perhaps implicitly) by the organization performing the development. For example, there may be requirements or encouragement to utilize a component repository, object-oriented environment or previous designs in which the organization has significantly invested.
- Experience of the architect. The results of previous decisions, whether successful, not successful or somewhere in between, affect the architect's decision to reuse those strategies.

Brooks argues forcefully and eloquently that conceptual integrity is the key to sound system design and that conceptual integrity can only be had by a very small number of minds coming together to design the system's architecture (Brooks, 1975).

Representing and Communicating the Architecture

In order for the architecture to be effective as the backbone of the project's design, it must be communicated clearly and unambiguously to all the stakeholders. Developers must understand the work assigned to them, testers must understand the task structure it imposes on them and management must understand the scheduling implications it suggests. The representation medium should therefore be informative and unambiguous and should be readable by people with varied backgrounds. The architects themselves must make sure that the architecture will meet the behavioural, performance and quality requirements of the system(s) to be built from the architecture. Therefore, there is an advantage if the representation medium can serve as input to formal analysis techniques such as model building, simulation, verification or even rapid prototyping. Towards this end, the representation medium should be formal and complete. The field of architecture description languages (ADLs) is young but growing prolifically (Clements, 1995).

Analysing or Evaluating the Architecture

There are several methods for evaluating architectures at various levels. We will in fact spend an entire chapter later in this book on evaluation of architectures. In addition to employing architectural description languages, several methods such as SAAM and architecture trade-off analysis method (ATAM) are popular. We will study a generalized scenario-based method, similar to SAAM and ATAM, in detail in Chapter 3. There is an emerging consensus on the value of scenario-based evaluation to judge the architecture with respect to non-runtime quality attributes.

The analysis capabilities that many ADLs bring to the table are valuable, but tend to concentrate on the runtime properties of the system, its performance, its behaviour, its communication patterns, and so on. Less represented are analysis techniques to evaluate architecture from the point of view of non-runtime quality attributes that it imparts to a system. Chief among these is maintainability, the ability to support change. Maintainability has many variations: portability, reusability, adaptability and extensibility. All these are special perspectives of a system's ability to support change.

THE USE CASE ANALYSIS

Use case analysis aids in understanding the problem space better. It starts in the problem definition or requirements gathering phase and captures the requirements in the behavioural methodology. Use cases have now become a standard way to gather requirements. In the book *Unified Modeling Language Users Guide* (Booch *et al*., 1999), the importance of use case analysis was mentioned as

> You apply use cases to capture the intended behavior of the system you are developing, without having to specify how that behavior is implemented. Use cases provide a way for your developers to come to a common understanding with your system's end users and domain experts.

Test cases are derived from use cases and hence they also help in validating and verifying the architecture and the system as it evolves during development. The use case analysis is followed by use case design and use case implementation. This is an iterative process that will be continued until all use cases are exhausted (Figure 2.3).

Objectives of Use Case Analysis

The main objective of use case analysis is to document the business processes that need to be ultimately supported by the system under development. Use cases are developed keeping this objective in mind and with the following goals in order to be effective.

- *Use cases must enable communication among all stakeholders*. Use cases very effectively communicate the business requirements to the developers and help customer verify if their requirements are understood properly. Through use cases, software developers communicate how the business requirements will be met by the system. It means that the software developers should be able to map specific functionalities or the features of the system to use cases. In case any specific functionality cannot be mapped to the use case, then the solution does not meet the business requirements. Use cases also become very handy if there is a disagreement between the customer and software developer after the completion of the software solution.
- *Use cases must identify business processes, decisions and actions*. This will help in scoping the business objectives that are expected to be achieved by the proposed system. Use case development for a business application is best achieved by identifying business decisions and business actions in the use case steps. We need to ensure that all the steps are documented, but it is important to understand how each one of these steps in developing a use case supports a business process, a business decision or a business action, which in turn helps accomplish the use case goal. Using this approach, we can identify and eliminate all unnecessary steps and make sure that each of the steps has a clear purpose.

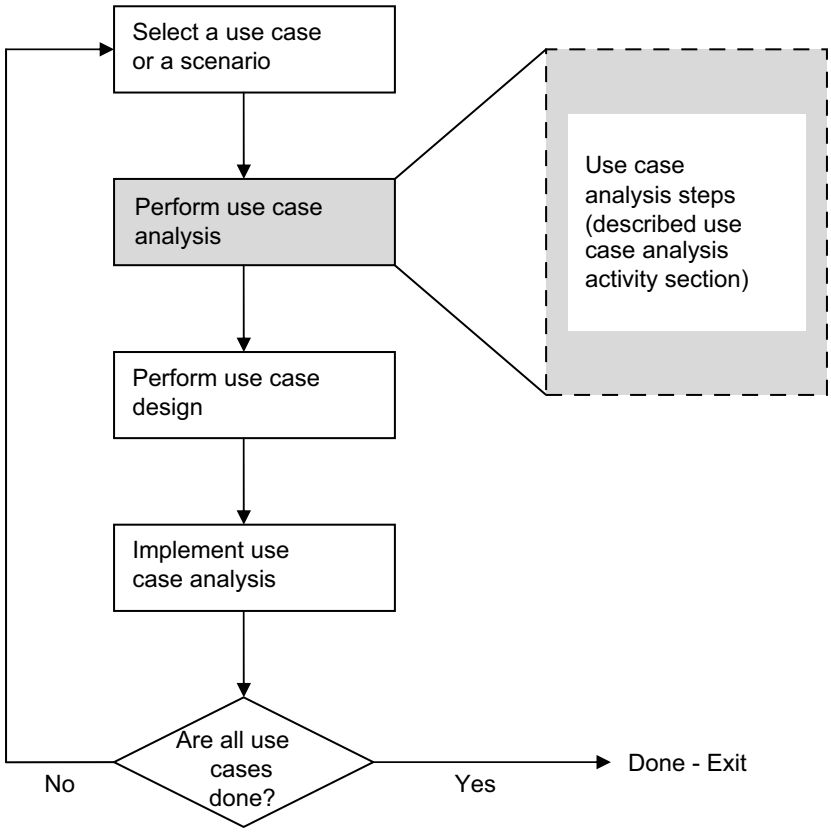


Figure 2.3 Use case analysis steps in a unified process

- *Each use case step should be clearly known (whether automated or manual).* Each use case describes interactions between the actor and the system, but at this stage in use case development it is hard to know whether a given step will be automated or will be manual. Even if it is known, the manual steps should still be fully documented in order to make the use case a more effective communication tool. To identify whether a use case step is automated or manual, one needs to observe the focus of that particular step. Typically, if the system is assigning responsibility for each business decision and business action then the step is automated; if it is done by the actor (non-system) then it is manual.

In other words, the goal of use case analysis is to iteratively refine and transform the initial understanding of the requirements that are represented in the form of system's use cases so that the transformed use case representations support the business concepts. These refined use cases are expected to meet the business goals that are within the scope of the proposed system. The main advantage of use case analysis is that it brings in the customer and the business focus in place of functionality focus.

Once the use case analysis is done, the next step would be use case design, wherein the business concepts are transformed into classes, objects and relationships, components and interfaces. These artefacts of the use case design can be implemented in the later stages of the software development life cycle.

Use Case Analysis Activity

According to rational unified process (Kruchten, 2003), the use case analysis activity occurs within the context of the other analysis and design activities. Before doing the use case analysis, one needs to ensure that the architecture analysis is performed. The architectural analysis provides the refined software architecture document (with refined 4+1 views), a design model and a deployment model. Once we have these models we can perform the use case analysis.

Use case analysis is composed of the following phases according to Kruchten (2003):

1. *For each use case obtain a corresponding analysis class*. The analysis classes are obtained first by creating a use case realization and then by supplementing the use case descriptions. The analysis classes are derived from the use case behaviour. Sometimes this behaviour needs to be distributed across a set of analysis classes.
2. *Enrich analysis classes with more details*. Once the analysis class is obtained, the class responsibilities, attributes and associations need to be described. This activity results in clearly defining the attributes of the class, establishing associations between various analysis classes and finally describing event dependencies between analysis classes.
3. *Put together refined use case realizations*. With the additional information obtained from the previous step, the use case realizations are reconciled and refined.
4. *Establish traceability*. As the requirements are represented in the form of use cases, which in turn are being transformed to analysis classes and then as design classes, it is important to establish the traceability between these transformed objects.
5. *Qualify analysis mechanisms*. Analysis mechanisms are specific to problem domain and not technical or solution specific. These are high-level architectural services that are later translated into design and implementation mechanisms. In Assure-Health, the business requirement was that the data about the patent's health history, claims and so on needed to be maintained throughout the life span of the record. This business requirement is translated into the analysis mechanism of 'persistence' of these data. The persistence analysis mechanism might be translated into design mechanisms like an RDBMS or an implementation mechanism like ODBC-SQL server, which means that the transformation is from platform independent to platform or vendor specific solution space.
6. *Evaluate the results of the use case analysis*. As in case of every phase, and as a part of the process, one needs to evaluate the results of the use case analysis phase and check if all the objectives are achieved.

However, note that these steps are not ordered, which means that the actual order in which these steps are carried out will be dependent on your understanding of the domain you are analysing, your experience with RUP or UML, your personal preferences for the models you use or the metaphor you follow for characterizing the properties of your analysis classes (Evans, 2004).

THE TECHNICAL PROCESS OF DESIGNING ARCHITECTURES

The most important deliverable of the process of designing the architecture is the architecture document(s), describing the structure of the system through various views. An architect develops a design by making a collection of design decisions and then reasoning about these decisions by considering different architectural structures and views.

Architectural requirements need to focus on the structuring or designing activities. Architectural requirements are used to motivate and justify design decisions. Different views are used to express information pertinent to achieve quality goals, and quality scenarios are used to reason about and validate the design decisions. Design decisions are frequently initiated with knowledge of architectural styles, design patterns or the use of particular tools. Lastly, a validation phase provides early indicators of and hence an opportunity to resolve problems with the architecture.

Architecture design is an iterative process (see Figure 2.4), with a collection of decisions being made and reasoned about, then reconsidered and remade until closure is reached.

Architectural Requirements

All of the system requirements are equally important from the software architecture design viewpoint. Some of these systems requirements are more relevant for making decisions during the architectural phase and such requirements are called architectural requirements.

It is important to make sure that the business objectives are always aligned with the proposed architecture. Hence, you need to determine which of the business requirements should be considered as architectural requirements. The scoping of the system can be determined by studying the environment in which the system will be (or is) operational, that is, the systems context. The context of the system also determines various system interfaces as well as all the factors that can potentially influence or intrude on the proposed architecture.

The high level or the most important functionality of the proposed software system is typically captured in terms of use cases, which is a popular way of capturing functional requirements of the system. It is most important that the software architecture provides support for all functional requirements, failing

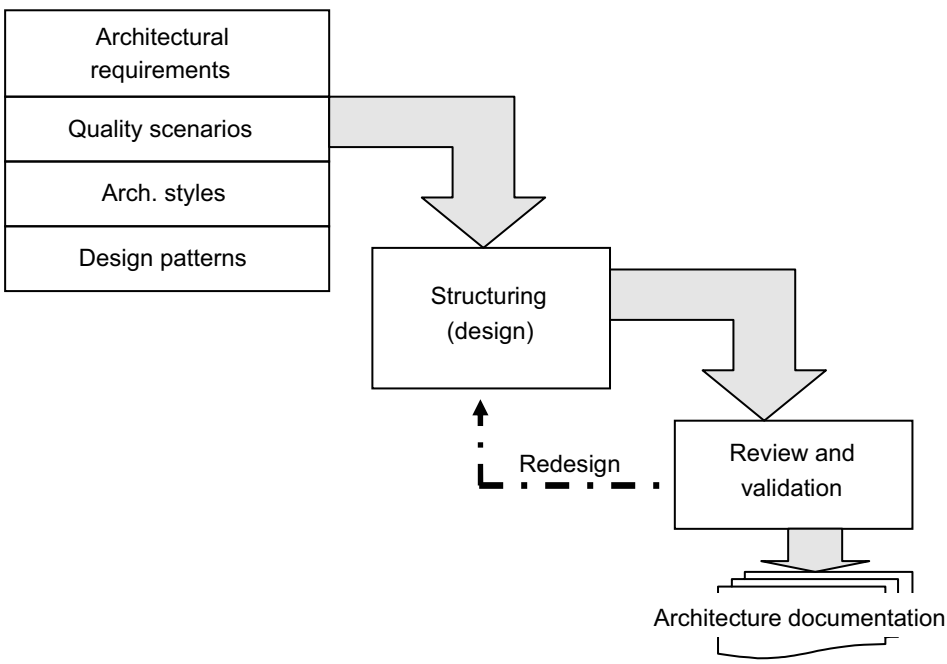


Figure 2.4 The technical process of designing an architecture

which the structure of the system cannot be considered as useful. In addition to these, all the non-functional requirements, such as performance, reliability, availability and security, which have architectural significance, are considered as architectural requirements as they influence the architectural decisions during the structuring phase.

The systems requirements are collected by the product teams or analysts during the requirements engineering phase—typically before the architectural team is involved in a full fledged manner. The architectural team needs to review all the requirements to determine whether they are complete or not and to determine how relevant they are from the architectural point of view. The check for completeness is most important for all non-functional requirements, as there may not be as much focus on collecting them when compared with functional requirements.

If the architectural team is designing a product line, then it becomes very important to distinguish between the architectural requirements that belong to individual products and those that are common to all the products within the product family. This distinction helps in coming up with common services or components and a specific component for each product and hence has a huge influence on structuring of the system.

System Structuring or Designing the Architecture

The architecture is created and documented in the system structuring phase. Architectural structures are foundational representations of architectural information, representing the base artefacts that we design and code: classes, functions, objects, files, libraries, and so on. Views are *derived* representations arrived at by selecting a subset of a structure or by fusing information from multiple structures. These canonical structures may be elaborated in a variety of orders depending on the context of development. Some of the most important ones are functional, code, concurrency, physical and developmental structures. We will describe these five structures below.

Functional structure. The functionality the proposed system needs to support is decomposed in the form of a functional structure. This decomposition results in a set of domain entities and their connectors. The resultant structure helps in understanding how various entities interact with each other, giving better insight into the problem space. That is, we have a better understanding of the how domain changes and hence can plan for functionalities in a better way, which ultimately results in a better product line.

Code structure. Software systems can also be viewed as a collection of code components. These components may be functions/methods, procedures, objects, classes or packages, which are means of packaging the system's functionalities. Each of these code components are at various levels of abstraction. There is communication defined between these code components in the form of relations. The code structure captures the distribution aspects of the system and helps us in understanding how good the engineered system is in terms of various quality attributes such as reliability, maintainability, reusability and portability.

Concurrency structure. When a system is viewed in terms of units of concurrency, such as processes and threads, we get the logical concurrency view. This structure helps us in understanding which processes are of higher priority than others, which ones need to be synchronized, which of these processes need to exchange data between them, which processes are potential performance bottlenecks, and so on. This view captures various properties of the processes (like execution time, priority, whether the process can be pre-emptied).

Physical structure and development structures. These are concerned with hardware and files/structures, respectively. The hardware aspects include configurations of the machines (servers and clients), bandwidth, and so on, which will help in better estimating the availability and performance aspects of the

proposed software system. A good file and directory organization will help in managing and maintaining the system in a better manner, for example, configuring a source code control system and release management system.

It is very important to separate various aspects of the system—that is, when one is thinking about the functionality, distribution and performance need not be brought into focus. Similarly, when one is thinking about distribution, performance becomes an important criterion and the functionality need not be taken into account. This is the primary reason why multiple views, each capturing and exposing some information and ignoring or hiding other information, will immensely help the architect.

Designing the architectural process begins with an assumption that we have a list of architectural requirements and a list of classes of functionalities derived from the functional requirements. The goal of the first step is to develop a list of candidate sub-systems. All of the classes of functionalities that are derived from the functional requirements are automatically candidate sub-systems. We derive the other candidate sub-systems from the architectural requirements.

For each architectural requirement, we enumerate possible architectural choices that would satisfy that requirement. For example, if the requirement is to allow for the change of operating systems, then an architectural choice to satisfy that requirement is to have a virtual operating system adaptor. Some architectural requirements may have multiple choices for satisfaction, while others may have a single choice. This enumeration comes from a consideration of design patterns, architectural styles and the architect's experience. From this list of possible choices, a selection is made so that all of the architectural requirements are potentially satisfied, the list of choices is consistent and the list is minimized. If one choice will satisfy two requirements (even if less than optimally), then that choice should be preferred over two different choices. Fewer choices mean fewer components, less work in implementation and maintenance and, in general, fewer possibilities for error. Each of the choices is added to the list of candidate sub-systems.

The next step in the process is to choose the sub-systems. Each candidate sub-system is categorized as an actual sub-system, a component in a larger sub-system or expressible as a pattern to be used by the actual sub-system. The actual sub-systems is then recorded in the functional structure.

Once a list of actual sub-systems is generated, the next step is to populate the concurrency structure. This structure is populated by reasoning about units of distribution and units of parallelism with respect to the sub-systems. Each sub-system may be distributed over several physical nodes, in which case the units of distribution are identified as components belonging to that sub-system. Units of parallelism are identified by reasoning about threads and synchronization of threads within the sub-system. The threads are 'virtual threads' in that they identify elements of parallelism if all of the sub-systems were to reside on a single processor. When the physical structure is considered, it is possible to identify the places where the virtual threads turn into physical threads and the necessary network messages derived from this transformation.

At the end of this design step, the sub-systems and their concurrency behaviours have been identified. This step is then validated and the sub-systems are refined, the concurrency behaviour again identified, other structures populated, and so on. Of course, the validation step may cause decisions to be revisited and so the actual process is highly iterative between decisions and validation.

Architecture Review and Validation

While coming up with the initial version of the architecture, the architects obviously make their best effort to meet the requirements. However, it requires external validation. It is always good practice to involve people from outside the team in the architecture review and validation process. They will help in looking at the architecture in an unbiased manner and provide objective assessment. This external validation will enhance the confidence in the system for all the stakeholders as well as help create additional buy-in for

the architecture. There are several methods to evaluate software architectures. We will discuss some of them in detail in a later chapter. Most of these methods are based on thought experiments. They model the scenarios by walking through a proposed user experience from the functional as well as non-functional aspects. In addition to these thought experiments or scenario-based methods, experienced architects will be able to help in finding major flaws or potential gaps and weaknesses using their expertise.

As mentioned earlier, the quality scenarios are used as the primary validation mechanism. The proposed structures are examined via the quality scenarios to see if the scenarios are achievable at the current level of design. If they are, then the next refinement of the design can proceed; if not, then the design at the current level of refinement must be reconsidered.

Each of the iterations is actually performing a mini-architecture analysis. The scenario-based reasoning can be used for design or in an evaluative setting. The scenarios structure the analysis, describe the performance threads, operational failures, security threats and anticipate changes to the system.

The mapping of these scenarios on the appropriate architecture structures and views tells us *what* to analyse. For example, one typically does not want to analyse system performance as a whole—it is too vague and there are far too many things that could be measured. We typically want to predict average throughput or worst-case response time or some similar measure, as derived from and motivated by particular usage scenarios. Once the architecture has been designed, it is useful to have an architectural evaluation carried out by a group that consists of people who have not developed the architecture. An external analysis makes the results of the design process visible to management and forces the documentation of the architecture.

Developing a proof-of-concept or a prototype is another common (and complimentary) way of validating architecture. This starts with taking only the core part of the architecture (stripping off all the auxiliary systems and sub-systems and their interfaces). The architecture team then comes up with a detailed design, implements the design and tests and validates the architecture. This gives them confidence as well as the understanding of the issues that might come during later phases of the software development life cycle.

Iterations

Though described sequentially above, the architecting process is best conducted iteratively, with multiple cycles through requirements, structuring and validation.

In the iteration model, it is important to spend a couple of initial cycles (call them trail runs) in understanding all meta or conceptual issues and develop guidelines and educational material such as tutorials that will help in executing later cycles in a better manner.

In each of the later cycles, it is good practice to take up one or two next high priority architecture requirements and proceed with the design or structuring process and end with validated and completed architecture for the current scope with the current level of maturity.

The iterative process of architecture development can be viewed also as a hierarchical model, wherein different architectural teams are working at different levels and each of them may stop at different points from which other teams start functioning to provide more detail. At the end of the first level, the core team of architects will create meta-level architecture and develop a set of guidelines and constraints to take this meta-architecture into further detail. Various product or component teams start with this architecture and create a more detailed architecture following the guidelines and constraints. They may also enhance or modify the guidelines and constraints. This continues further until, at the other end of this hierarchical spectrum, we have the detailed architecture complete with logical architecture specification. This is an effective model but not always practical. A more practical model is where the same team refines the architectural detail in every cycle.

CASE ANALYSIS

From the software architecture perspective, this case study clearly shows that there is no clearly defined architecture at all. What was shown as an architectural diagram is only a marketing architecture, which is of no help to the developers. One can also clearly see that the architecture shown is talking about various platforms and specific software packages. Clearly, this is a wrong way to represent architecture as it cannot provide any information about the system.

In this case, the project organization structure has been decided even before the requirements were clear. Typically, this would occur in situations where the customer commits a fixed number of people to work in a time and material model. However, this being a fixed-price project there is no compulsion for Fact-Tree to commit their numbers in advance.

Among the onsite team of six analysts, no one has been designated as the leader who can be the architect, master designer or the project leader (if he/she can carry two hats). This role is crucial for orchestrating the development, documentation, prioritization and managing changes and act as technical spokesperson for Fact-Tree. Instead, each of the six analysts focused on his or her own module, ignoring the bigger picture. This situation has resulted because there is no common platform in the form of architecture and there is no architect who can coordinate different groups. The role of an architect was missing in the project. None of the analysts bothered about designing the architecture for the complete system, specifying all interactions, dependencies and component structure.

As mentioned earlier in this chapter, the first step in the process of designing the architecture is to understand the architectural requirements properly. This includes understanding and documenting the domain requirements, quality requirements and other software architecture knowledge including that of design patterns. We follow the following steps in designing an architecture for any given system:

- *Step 1*. Document our understanding of the domain requirements and other architectural requirements
- *Step 2*. Structure various sub-systems to come up with an initial architecture
- *Step 3*. Review and evaluate the architecture in various iterations until the solution meets all requirements

Initially, we try to document various domain requirements by understanding the case study and list various possible scenarios. From these scenarios, we come up with generic use cases and document these use cases. These use cases serve two purposes: they work as a basis for coming with a testing plan to make sure that all functional and non-functional requirements are being met and they are also used as the main input to come up with a solution in terms of possible architecture and detailed design.

Once the use cases are identified, a detailed architecture of the system should be worked out using a set of formal architectural diagrams with a standard notation (such as UML's 4+1 views), which would have helped avoid all design and implementation problems later on. The development approach (SSAD or OOAD) can be decided after the architecture was completed, as the architecture will suggest what could be a better approach. The information that is available within the case study is not sufficient to come up with all views. However, one should try to gather as much information as possible by carefully studying the details given and come up with as many views as possible.

In this section, we give two views of the standard UML representation, namely, the use case view and the component view. The use cases are derived after understanding the functionality of the TPA domain. Flow of events in each of the use cases is given after the use case diagram is prepared. Note that the use

case diagrams given are only the samples and by no means can they be claimed as perfect. The focus here is to deal with lack of architecture and come up with necessary architectural artefacts with the limited information available and to learn about the benefits of having proper architecture. In the next chapter, we will see how to come up with various views.

Understanding Domain Requirements and Architectural Requirements

Understanding the domain is the key for successfully building any application. One of the major problems with this case is the lack of proper domain expertise. Lack of domain knowledge in the development team and relying heavily on an unreliable third party (MidAlliance in this case) to provide this domain knowledge resulted in chaos. This not only led to a faulty estimation but also created an environment where complexities involved were not clear to the team members for a long time. Here, involving a domain expert will help the architect to understand the domain better.

An example use case diagram for the system is given in Figure 2.5.

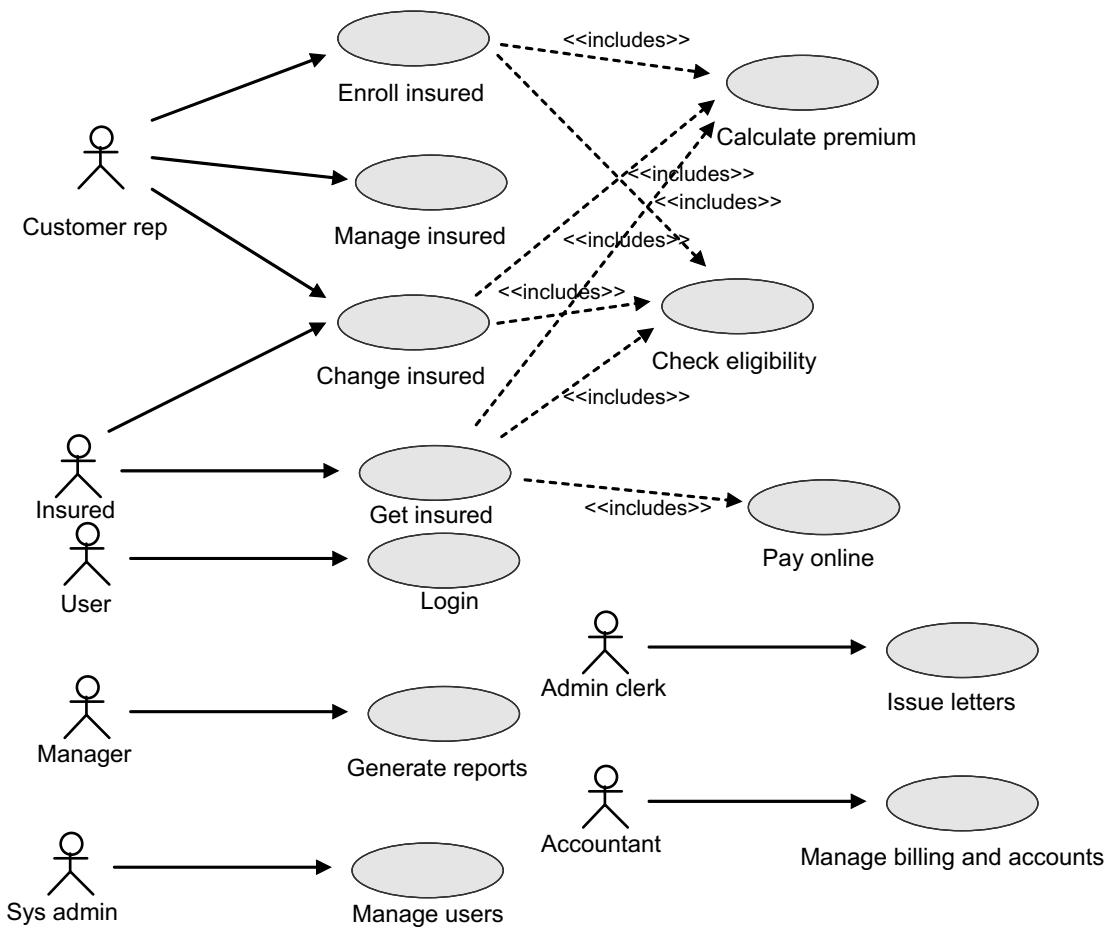


Figure 2.5 Use case diagram of Assure-Health

The details of the use cases are given below. As the main purpose of this case is to make the reader understand the need for the use cases and the flow of events, only the basic flows are presented here. The alternative flows of events are not presented.

Use Case: Enroll Insured

This use case starts when a customer comes to the customer representative and requests her to get an insurance plan. The customer representative is the actor for this use case.

The flow of events of this use case is as follows:

- The customer representative collects information about the client who wants insurance (hereafter termed *insured*) manually either using a form or verbally
- The customer representative enters the information of insured into the system
  - Demographic information
  - Family details
  - Dependent details
  - Employer's information
  - Financial details
- The customer representative enters the primary care physician details
- The system invokes the use case 'Check Eligibility' to check the eligibility of the insured
- The customer representative adds the insurance plans and other information
- The system invokes the use case 'Calculate Premium' and displays the same to the customer representative
- The customer representative manually informs the insured about the premium and after confirmation confirms the insurance plan into the system
- The system generates an insured ID
- The system stores the information into the database
- The system displays the insurance details with the insurance ID, which can be printed and given to the customer

Use Case: Get Insured

This use case starts when the customer wants to get insured through the online facility. The insured is the actor for this use case.

The flow of events of this use case is as follows:

- The insured connects to the Assure-Health system through the Internet
- The insured enters the information into the system
  - Demographic information
  - Family details
  - Dependent details
  - Employer's information
  - Financial details
- The system displays the primary care physician details as available for the location of the insured.

- The system invokes the use case 'Check Eligibility' to check the eligibility of the insured
- The system displays the insurance plans available for the insured
- The insured selects the insurance plan and enters other information
- The system invokes the use case 'Calculate Premium' and displays the same to the insured
- The insured confirms the insurance plan in the system
- The system invokes the use case 'Pay Online' and completes the payment of the premium for the selected insurance plan
- The system generates an Insured ID
- The system stores the information into the database
- The system displays the insurance details and requests the insured to take a print out of the same

Use Case: Check Eligibility

This use case is invoked by the system when the customer representative enrols an insured or when an insured wants to get insured through the Internet.

The flow of events of this use case is as follows:

- The system collects the eligibility conditions from the database
- The system identifies different insurance plans possible for the insured
- The system verifies the insurance plans with the eligibility conditions
- The system passes the insurance plans to the use case that invoked this use case

Use Case: Change Insured

This use case is used when an insured wants to change the demographic or other information. This use case is also used to change insurance plans. The actors of this use case are the customer representative and the insured.

The flow of events of this use case is as follows:

- The user (customer representative or insured) selects the option 'Change Insured' in the home page of the system
- The system displays a form to enter the insured ID or details of the insured
- The system searches the database to collect the information of the insured and displays the same on the screen
- The user changes the relevant information of except for name, age, sex, employment details and financial Information
- The user also can change the insurance plan
- The system invokes the use case 'Check Eligibility' and displays the alternative insurance plans
- The user selects the insurance plan
- The system invokes the use case 'Calculate Premium' to calculate the premium for the new insurance plan
- The system invokes the use case 'Pay Online' if the user is insured and if the premium is higher than earlier

- The system stores and modifies the information about the insured
- The system displays the modified information to the user and requests the user to take a print out of the insurance

Use Case: Calculate Premium

This use case is invoked by the system when the customer representative enrols an insured or an insured wants to get insured or the user uses the use case 'Change Insured'.

The flow of events of this use case is as follows:

- The system collects the financial information of the selected insurance plan
- The system calculates the premium for the selected insurance plan
- The system passes the premium information to the use case that invoked this use case

Use Case: Pay Online

This use case is invoked by the system when the insured uses the use case 'Get Insured' and confirms the insurance plan.

The flow of events of this use case is as follows:

- The system displays a payment form
- The insured enters the credit card information and submits
- The system connects with the credit card service interface
- The system asks the credit card interface to verify the credit card information and approve the amount
- The system confirms to the insured that online payment has been made

Use Case: Manage Insured

This use case is used when the customer representative wants to manage the information of the insured. The customer representative is the actor for this use case.

The flow of events of this use case is as follows:

- The customer representative selects the option to manage insured information
- The system displays a form to enter the insured ID or details of the insured
- The system searches the database to collect the information of the insured and displays the same on the screen
- The customer Representative changes the following information
  - Name
  - Age
  - Sex
  - Employment information
  - Financial information
- The system stores the information in the database
- The system displays the changes that were made

Use Case: Issue Letters

This use case is used when the administration clerk wants to generate the letters for the insurances registered in the system. The administration clerk is the actor for this use case.

The flow of events for this use case is as follows:

- The administration clerk selects the option to Issue Letters on the home page
- The system displays the form to enter the insured ID or insurance plan
- The administration clerk either enters the insured ID or the insured plan
- The system displays all the insured IDs with the selected insured plans
- The administration clerk selects the insured ID from the ones displayed
- The system displays the insurance information
- The administration clerk selects the types of letters to be issued and requests the system to generate the letters
- The system generates the letters and displays
- The administration clerk selects to print the letters
- The system prints the letters through the printer
- The letters are issued manually to the insured

Use Case: Generate Reports

This use case is used when the manager wants to generate reports for the insurances registered in the system. The manager is the actor for this use case.

The flow of events for this use case is as follows:

- The manager selects the option to Generate Reports on the home page
- The system displays the form to enter the insured ID or insurance plan
- The manager either enters the insured ID or the insured plan
- The system displays all the insured IDs with the selected insured plans
- The manager selects the insured ID from the ones displayed
- The system displays the insurance information
- The manager selects the report types and enters the required information
- The system generates the reports and displays
- The manager selects to print the reports
- The system prints the reports through the printer

Use Case: Manage Billing and Accounts

This use case is used when the accountant wants to manage billing and accounting for the insurances registered in the system. The accountant is the actor for this use case.

The flow of events for this use case is as follows:

- The accountant selects the option to manage Billing and Accounts option on the home page
- The system displays the form to enter the insured ID or insurance plan
- The accountant either enters the insured ID or the insured plan

- The system displays all the insured IDs with the selected insured plan
- The accountant selects the insured ID from the ones displayed
- The system displays the insurance information
- The accountant selects the option to generate the bill
- The system generates the bill and displays the same
- The accountant accepts the payment manually from the insured
- The accountant enters the payment details into the system
- The accountant requests the system to print the receipt
- The system prints the receipt through the printer

Use Case: Manage Users

This use case is used when the systems administrator wants to manage the users who can logon into the system. The systems administrator is the actor for this use case.

The flow of events for this use case is as follows:

- The systems administrator selects the option to manage users on the home page
- The system displays the system administration page
- The systems administrator selects the option to create a user
- The system displays the user details form
- The systems administrator enters the user details and submits
- The system generates a new user ID and displays the same
- The systems administrator selects the option to change the user information
- The system displays the form to enter the user ID or user name
- The systems administrator either enters the user ID or the user name
- The system displays all the user IDs for the entered user name.
- The systems administrator selects the user ID from the ones displayed
- The system displays the user information
- The systems administrator changes the information of the user
- The systems administrator resets the password of the user
- The system stores the user information

Use Case: Login

This use case is used when any user of the system wants to login into the system. This use case is the pre-condition for all the use cases in the system. The actor 'User' is the abstraction of all the actors used in this system. All other actors are derived from this actor.

The flow of events for this use case is as follows:

- Once the user selects to use the system, the system displays the login form
- The user enters the user ID and password in the login form and submits
- The system gets the user information from the database
- The system verifies the user ID and password
- The system displays the home page of the system

Structure Various Sub-systems to Come Up with Initial Architecture

The requirements analysis helps us identify various sub-systems and the functionalities of each of them. Use case diagrams also help us observe how each of the sub-systems talks to each other and with various actors. This helps us design the interfaces for components that will form these sub-systems. Component view expresses these components and how they are related to each other in terms of the interfaces they provide.

Here we identified the six major sub-systems based on our analysis in the previous section. They are eligibility, customer services, reporting, letter issuance and call tracking, security and billing and accounting. We also studied how each of the sub-systems interacts with each other. Based on these inputs, we can come up with a component view of the proposed system. An example component diagram is given in Figure 2.6.

There are several techniques and heuristics that can be used in designing the architecture. These methods apply to many design tasks. These include the following:

- *The method of persistent questions*. This method is a tool an architect may use when analysing the problem to discover useful abstractions, as well as for evaluating solutions. The architectural team keeps a database of standard sets of questions for various purposes. Such a database helps extend the memory of architects and transfer the useful design methods and knowledge to others.

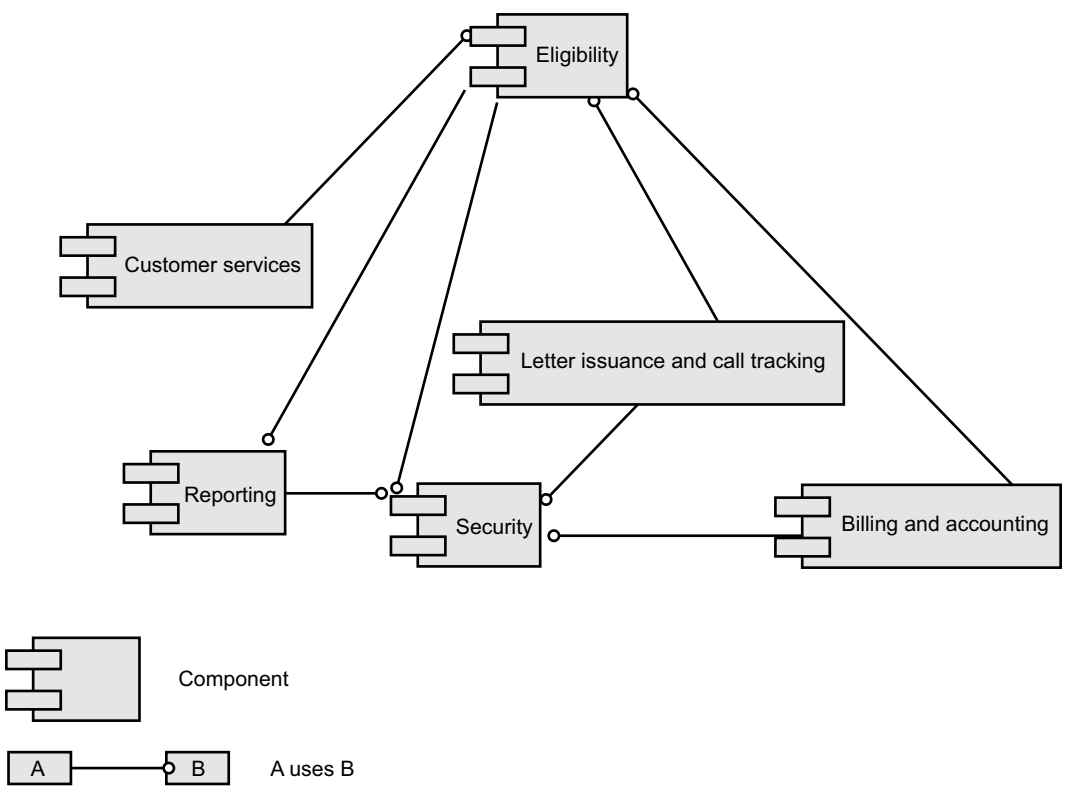


Figure 2.6 The component diagram

- *The method of navigation*. This method is also known as *deliberate negation* or *systematic doubting*. The known solution is divided into various statements or truthful predictable relations. These statements are then negated one after another in a systematic fashion. For example, there is a statement about two-phase commit of transactions, by negating this statement and assuming that the system will not use a two-phase commit approach, we are forced to consider the consequences or some alternatives.
- *The method of forward steps*. It starts with a first solution attempt and proceeds to follow as many solution paths as possible, yielding other solutions. This method is usually not systematic and starts with unsystematic divergence of ideas.
- *The method of backward steps*. In this method, the architect starts with a goal in mind rather than an initial problem. Starting with the final objective of the development effort, all possible paths that could have led up to the objective are retraced.
- *The method of factorization*. The method involves breaking a complex system into less complex elements or factors. Factorization is a type of analysis method. In designing the architecture, this method is used to find essential problems being solved.

Using techniques such as the ones listed above, we can produce a solution field consisting of several candidate structural architectural designs. These candidates are not only purely logical models but also specify working principles, which may include identification of techniques or design patterns that can be used to solve the problems.

The architecture design or design candidates are reviewed, evaluated, disposed of or further refined during embodiment design. The same rational operations and design methods are applied but now the focus is more on the elaboration of the solution and less on formulation of the problem. Detailed design involves the specification of the layout, implementation and testing. The architectural artefacts developed in this process are refined until they are fit for the purpose.

There are some useful things that we can learn from this case study, which we can call as an architecture process survival guide.

Identifying Domain Expert

In the case of product development, the domain expert is the product designer/owner. He is the interface between the system developer and the customer/user. He is the key person who decides the functionality of the product in a particular release.

In the case of Assure-Health, which is a project, the domain expert is a person who tries to understand the requirement of the user and transfers the same to the development team. He is the sole gatekeeper of the project requirements, who ensures that the correct system is delivered to the users. He does requirements management (requirement engineering) and also system testing before delivering the system to the client. Nothing goes past the development organization unless validated and certified to be correct by the domain expert.

The primary difference between a domain expert and a business analyst is that a domain expert knows the domain inside out, being a person professionally trained in it. A business analyst is a person who analyses the business requirements from a requirements management point of view and is able to 'translate' the requirements from the user's perspective to something that a software engineer may be able to understand easily and can design the system that the user can successfully use. No matter how hard we may train a business analyst with non-domain background, his understanding of the domain can in no way be compared to that of a domain expert.

So to identify a domain expert, the development company should select a person who is a professional in this field and has in-depth knowledge of the functionalities/user scenarios in the domain. This is the primary requirement to identify a domain expert. In the case of Assure-Health, a domain expert would be a person who either worked in an insurance company for some time or worked as an insurance agent and is an insurance license holder. The additional requirement would be that the person should have basic knowledge of how software systems are developed. That is, he should have knowledge of requirement engineering and use case analysis. I am calling it a secondary requirement because even it he does not have this knowledge, he can be trained on this very fast (may be within 2 months).

Role of a Client

If the development team does not have any domain expert, it is suggested that the client can support the team with its own domain experts. Usually, the client will have a number of domain experts in its organization, but there are two challenges to be overcome to use them in the project effectively:

1. They are usually very busy in their scheduled work and there is no motivation for them to perform this extra work. (They consider them as extra work. Sometimes, they even think that they should not help the development team as the development team is paid to design the system.)
2. The domain experts have no knowledge on how the requirements are to be elicited and communicated to the development team. This is a serious problem and most of time the development team misunderstands the requirements explained by the domain experts.

It is thus important to train domain experts of the client in requirement engineering and basic software development methodologies before using them as domain experts for the project.

Role of MidAlliance

Actually MidAlliance was supposed to play the role of a domain expert. They should have hired a professional from the insurance domain and used the expert as an interface between HALI and PayMinders. Interestingly, MidAlliance wanted to develop a product and the product should have an owner. No product can be developed efficiently and effectively if the owner of the product is not a domain expert. This is the reason they failed in both product development and the project.

The primary role of MidAlliance would have been to support HALI in providing domain knowledge and performing system testing. All the domain-related problems should have been identified by MidAlliance before they were delivered to PayMinders. It was more so as the primary communication was between PayMinders and MidAlliance.

If for any reason it was difficult for MidAlliance to provide a domain expert, they should have ensured that HALI had allocated a domain expert in the development team and they should have been in constant touch with the domain expert to verify that the requirements were identified and being developed correctly.

Conclusions

Using the Assure-Health case study we tried to emphasize on the need for architecture in the software development life cycle. Unfortunately in many real-world projects, this emphasis is missing. We have also tried to provide directions in which one can start designing architecture. Case studies in the subsequent chapters will build on the concepts and pointers provided in this chapter. For instance, complete 4+1 views are given in Chapter 3 while discussing about a call centre software system. Use case analysis is dealt in detail in Chapter 5.

As an active learner one can look at developing various architectural elements given in the first cut version of the case solution. For example, there are different roles played by different users as discussed in this case study (see Figure 2.1), such as TPA agent, end customer and corporate customer. As far as Assure-Health as a software system is concerned, the eligibility criteria may be different for each of these users. For example, if a TPA is adding a customer, it is expected that the TPA must have already done some background checks and hence the Health-Assure system need not insist on all checks that are done as in the case of an end user directly approaching the insurance agency. Similarly, types of security-related issues and volume of transactions that are expected from each type of user can be very different. We recommend that you should refine the proposed architecture by considering these kinds of requirements, whether they are functional or non-functional. In fact, the initial solution that is given is the starting point. Note that we have not given all details, such as all the 4+1 views. It may not possible to create all views given the information in the case study. The aim of this case study is to convey the importance of the case study and that it has to be described in sufficient detail.

To appreciate the concepts presented in this chapter, we recommend that the reader should think through the role of architecture in software development life cycle from this case study point of view. To enhance the analysis skills, refer to the section 'Use case Analysis'; start from the initial use cases and perform the use case analysis to come up with analysis classes. Fill in the missing artefacts of the architecture with the help of material presented in the section 'The Technical Process of Designing Architectures'.

Best Practices and Key Lessons from the Case Study

Key points of the case study exercise relating to architecture are given below:

- Every project team should have defined the role of the architect. The architect will be responsible for the overall architecture of the system, specifying all interactions, dependencies and component structure. The architecture should be created, reviewed and agreed upon before proceeding with other development activities.
- A detailed architecture using a set of formal architectural diagrams with a standard notation (such as 4+1 views) should be prepared. The development approach (SSAD or OOAD) should be decided based on the architecture.
- The team should have a reliable domain expert and this expert should act as a gatekeeper of any software solution being developed.
- Low-level design must be created to enable translation of complex architecture to more manageable requirements for programmers.
- The design should be mapped properly with regard to the prototype and its navigation.
- The testing team must understand the business context for the application.
- Proper communication channels should be set up between all stakeholders.
- Have an architect as a key member of the team to create, review and oversee technical architecture. This role is necessary to exercise the authority given to the technical partner. It is necessary for someone to hold the 'big picture' of the entire application and guide the development team and coordinate with the customer. He needs to be made accountable for designing the software architecture for the system.
- The architect needs to get assistance from a domain specialist.
- Create a standard description of the architecture covering all the relevant views for stakeholders.
- Perform architectural trade-off analysis and architectural evaluation of the system.
- Use an architecture-driven development methodology.
- Have a domain expert in the project to review solution offering.

- ❖ Create a low-level design and define detailed interfaces between all interacting modules.
- ❖ Make sure that the testing team understands the domain and business context for the solution.
- ❖ Perform impact analysis of requested changes within and outside the native module.

Further Reading

In this chapter, we have tried to motivate readers about taking architecture seriously. Most of the pointers provided at the end of the first chapter can be used to gain more insights into this aspect.

We have also provided various domain-related pointers while discussing about TPAs in the case study (Exhibit A). It is necessary to gain some understanding of this domain to solve the case study fully.

To understand object-oriented development, we recommend *The Object Primer,* 2nd ed., SIGS, 2001, written by Scott Ambler. Interestingly, this book takes a single case study and covers end-to-end object-oriented development. Another recommended book to understand object-oriented thinking is *Object Technology: A Manager's Guide*, Addison-Wesley, 1998, written by Martin Flower and David Taylor.

Martin Flower has also written an excellent book called *UML Distilled*, 3rd ed., Addison-Wesley, 2004. This provides a good introduction to UML (version 2.0). It will be very useful for those who are learning UML for the first time.

From the architecture point of view, RUP provides a lot of literature on how to come up with use cases, analysis classes, design classes and then writing code. The following books are very useful. IBM's Rational Edge Web site has very interesting and detailed articles on various issues related to RUP, including the following:

- Bell, Donald. UML basics—An introduction to the Unified Modeling Language, *The Rational Edge*, June 2003. http://www-106.ibm.com/developerworks/rational/library/769.html
- Bell, Donald. UML basics: The activity diagram, *The Rational Edge*, September 2003. http://www-106.ibm.com/developerworks/rational/librarycontent/RationalEdge/sep03/f_umlbasics_db.pdf
- Bell, Donald. UML basics: The class diagram, *The Rational Edge*, November 2003. http://www-106.ibm.com/developerworks/rational/librarycontent/RationalEdge/nov03/t_modelinguml_db.pdf
- Bell, Donald. UML's sequence diagram, *The Rational Edge*, January 2004. http://www-106.ibm.com/developerworks/rational/library/3101.html
- Gary Evans. Getting from use cases to code, *The Rational Edge*, July 2004. http://www-128.ibm.com/developerworks/rational/library/5383.html

Chapter 3

Refining and Re-factoring Architecture—Story of McCombbs Call Centre

Contributors

Kirti Garg, Ram Gopal, Bhudeb Chakravarti, Sugi Murmu, Muralikrishna Nandipati and Prashanti Reddy

In the software services industry, it is not uncommon to start a project with a given architecture or a high-level design. Often, in the outsourcing model, initial or so-called high-level activities such as requirements analysis and design of the architecture are done by one team (typically by the client) and the rest of the development is performed by some other team. However, there is no guarantee that the given initial architecture will satisfy all the functional and non-functional needs of the proposed system. If the development team religiously sticks to the original architecture, it might end up building a system that does not meet the goals set, in terms of performance, scalability, reliability, availability and security. Even if the original architecture is evaluated, verified and approved, the development team should have the required skill to re-factor the given architecture to suit the demands of the project. This, however, is not an easy task, especially when you have an architecture coming from a highly credible team.

In this case study, we will look at how a well-designed architecture by a leading global player in the software consultancy market space has failed to meet the performance, scalability and other non-functional requirements. The Fact-Tree team had to study the bottlenecks of the architecture and come up with a refined and re-factored architecture. This case study deals with a sophisticated, large-scale distributed system being developed for a Fortune 100 company. The challenge before the developing team is to develop a next-generation call centre management software. There are stringent performance requirements, and under the current development scenario, the system is not clearing all performance tests. There are very high chances that the system will not pass the client's test and heavy penalty may be imposed. The only hope is to fine-tune or re-factor the architecture, but the bigger challenge is to identify the bottlenecks and improve the overall user experience.

BACKGROUND

Good architecture is a factor behind the success of modern day complex applications. Various techniques, tactics, methods and strategies are being employed in designing good architectures. Understanding and meeting the critical non-functional requirements become the key, as software systems are becoming critical for business. The software should be up and running 24×7 (well, almost), the system should be made easy to modify, the organization should have complete hold on its information in terms of privacy and security aspects and, finally, the performance of the systems measured in terms of response time and throughput will ultimately decide the usability of the system.

There is a misconception that performance and other non-functional requirements can be 'tuned' once we have all the functionalities in place. Software performance cannot be integrated or injected into the system just before the delivery. A typically strategy observed in many projects is to first build a system that fulfils all functional requirements and then tune (or 'fix') the system so that all performance and other non-functional issues are resolved. However, if the software architecture has been designed and the code already written, then the modifications required would be too expensive.

The system should be designed keeping all the non-functional requirements in mind from the beginning and all components of the system should contribute towards it. Taking performance-oriented measures while designing the architecture and detailing the design of the system is the most effective way of building systems that fulfil performance objectives.

Planning, designing and monitoring for performance is becoming an integral part of software development and cannot be an ad hoc process. These concerns are addressed in a rapidly growing discipline called performance-oriented engineering or software performance engineering (SPE). *Fine-tuning* and *re-factoring* the system are the most common performance-oriented measures taken by architects. Applying these concepts and techniques to move towards a performance-oriented system is what we aim to learn through this case study.

Performance, like every other requirement that is either functional or non-functional, is to be planned and designed for. Typically, when the service-level agreement (SLA) is drafted, the performance issues are given attention for the first time and later the job is handed over to the architect for designing a high-performance system that will be able to meet the requirements stated in the SLA. However, a more preferable strategy for performance-oriented design would be to consider the performance starting from the requirements phase and subsequently transform the requirements in specific software design choices. Architectural and design patterns can aid in the architecture and design, respectively, since they generalize and conceptualize the knowledge that experienced software designers have already experimented with. Another recommended practice is that the architect should decide and prioritize the architectural goals of the system. These goals are derived from functional and non-functional requirements and play an important role in how the carved system will turn out to be. We shall revisit concepts such as architectural drivers, architectural patterns and architecture re-factoring after we go through the case study and understand the context in which we need to apply these concepts.

Case Study: Technical Architecture of McCombbs Call Centre Software

McCombbs is a global player in the consumer finance sector, providing financial services to the clientele including retailers and merchants in North America and Europe.

Formed in 1974 as a provider of consumer financing for electronic appliances and computers, McCombbs soon shifted its focus to provide private-label credit cards, commercial programs and card-related financial services for hundreds of retailers and manufacturers across North America. Clients of McCombbs are in a variety of industries, including appliances and electronics, department stores, discount industries, home furnishings, home improvement, oil and gasoline and specialty retailing (such as home shopping and apparel). They also provide services in the form of corporate cards for its commercial clients and include purchase, travel and fleet vehicle cards.

In 1986, McCombbs collaborated with various renowned retailers, such as Win-Mart and Q's Merchandise, to offer credit facilities to the end customers who make purchases from these stores. The business advantage that was forwarded to the retailers was that they could focus on their core business while transferring the risks and rewards of credit financing to McCombbs. The sales got boosted as the end customers had an additional way of making payments.

McCombbs has the onus of serving the customers and also managing its portfolio by ensuring timely and effective collection of dues. Hence, they basically provide two types of services: 'customer services' and 'collections'.

McCombbs has always been a technology savvy company. It started using mainframes very early, one of the first companies to adapt technology to address business issues. They still work with a legacy application that provides centralized information to all the call centres spread in different parts of the country.

Glossary

Customer Services

Customer services include providing information to the customers regarding available services, adding more services, changing services, promotions, explaining bills or any other service to interface with the customers.

Collections

These are the services related to payment processing and collecting the bills from the customers including monthly bill payments, overdue payments, interfacing with various agencies.

McCombbs' Business

At McCombbs, a major business component is call centres interacting with customers who purchase products using cards provided by retail stores. Apart from the cardholders, sometimes the retailers may also

connect to the call centres for enquiries such as payments through demand draft, authenticity of a transaction, completion of transaction.

After going through interactive voice response systems (IVRSs) or a voice response unit, if needed, customers interact with call centre executives (CCEs), also known as live agents.

IVRSs receive inbound calls from the customers and diallers make outbound calls such as automatically dialling a group of customers to prompt them for payment reminders, play the auto prompts and facilitate callers in providing balance enquiry, services offered and speak to live agents. Auto-Dialler software (such as Davox) makes predictive calls based on the call list allocated from a given promotion campaign, defaulters list, and so on.

When an inbound call is received, the IVRS prompts the caller for an account number, which acts as a unique identifier for the caller. Using this account number, details of the caller are retrieved from the host database. Once the details are retrieved, the call is routed to an available workstation CCE. For the initial support and training of the application, there are support personnel known as the floor walkers, who are also logged into the application in parallel to help the CCEs with the system. Floor walkers sometimes take the call in parallel and help the CCE to navigate through the system and thus provide satisfactory service to the callers.

There is a complete logging of information pertaining to the call made by the customer; the call history will be stored for future reference. The information logged includes the time and nature of service provided and the information about the CCE who handled the client call. Transaction history and details of each customer are stored and can be retrieved in whole or in parts by the agents as per their authorization level.

In addition to logging of information, live and recorded calls may be monitored for quality improvement and training purposes. The floor walkers are also logged into the system (they do not physically walk around) and they can monitor calls or they can receive calls that are transferred to them by CCEs. These transfers can be warm transfers or cold transfers.

Different clients of McCombbs, such as Win-Mart and Q's Merchandise, may have different business rules applicable to their customers. Hence, McCombbs should be aware of the rules applicable to each client and should process accordingly.

Glossary

Cold Transfer

When a customer calls and the call goes to an agent/CCE and the agent is unable to help the customer and transfers the call back to the rerouting system, the transfer is called cold transfer.

Warm Transfer

In a similar situation, if the agent who receives the call first identifies another agent who can answer the customer and connects with the second agent, informs him about the call and then transfers, the transfer is called warm transfer.

For McCombbs' call centre business, it is important to have a dialler strategy. Dialler strategy includes the ability to configure the rules in the auto-dialler. These rules can be configured based on the following:

- *Special campaigns and promotions*. If there is a special promotion or event, the auto-dialler can be given a list of customers based on criteria such as minimum amount of money spent, past history of purchasing a type of item, age, income group and location.
- *Defaulters*. Depending on the collections strategy, a list of defaulters based on criteria such as amount due, time since last payment and credit history is generated and given to the auto-dialler.
- *Call back at a specific time*. When a customer is not available or chooses to receive a call at a later point in time, the system should be able to schedule the call at the specified time.
- *Regulatory issues*. All outbound calls, especially those related to collections, should honour all the regulations of the federal and state governments. These regulatory acts include Fair Debt Collection Practices Act, Data Protection Act, debtor's right and privacy options on telemarketing.

The dialler strategy should include the ability to switch from automated to manual dialling mode or from predictive dialling to pre-view dialling based on the context and specific requirements during the call.

It is also possible that the outbound call does not reach the right party, in which case there should be a provision to handle such situations by correcting contact numbers, collecting right contact numbers, preferred calling times, and so on.

Description of the Existing System

CCEs can view and manage customer profiles and accounts through the front-end application for mainframes. System administration is also done through this application. But this is a character user interface (CUI) based application and hence the user of this system needs lots of training and practice before using it. Additional personnel (floor walkers) are needed to train the CCEs and help them in navigating through the application.

McCombbs uses a third-party data source, known as FDR, and is provided by First Data Corporation. This company provides transaction-processing services to a large number (approximately 1,400) of card-issuing clients. It provides the current credit profiles of the customers to McCombbs. McCombbs pays for each hit to this data source; hence, its use is costly for McCombbs, but there is no alternative.

Another data source is an MIDS database. This is a home-grown system of McCombbs. It stores the data related to security profiles, access control areas, hierarchical user roles, and so on.

Glossary

Character User Interface (CUI)

CUI applications are very simple (typically on monochrome monitors), old-fashioned applications without any graphical user interface. They have commands in the form of characters and control keys (e.g., CTRL + ALT + O means press Ctrl key and Alt key and the letter 'O' simultaneously to perform an operation such as opening a record).

Now It Is Time to Move on to Latest Technology and a Build Better System

McCombbs decided to move on to new technologies and address all the issues present in the existing system (shown in Exhibit A). The IT department conceived a project to achieve this goal. The new project is in line with McCombbs' major initiative on digitization, globalization and 'complete e-services'.

Exhibit A Major problems with the current system

- The users at call centres deal with more than 150 CUI screens and have to remember about 200 codes to perform their actions.
- There is no context-sensitive help, which implies that users will have to go through extensive and exhaustive training on the usage of the application.
- Not all processes are supported by the system. Also, the business policies applicable to a particular retailer cannot be referred or applied throughout the system; the user will have to remember the policies and business rules or refer to the appropriate authority, hence making the process burdensome.
- The statistics related to the efficiency of the team have to be collected manually, as managers need these data.

The objectives of the project are defined as follows:

- This project should implement a user-friendly, self-help enabled, web-based front-end system to be accessible from 'collections' and 'customer service' call centres and eventually via the Internet.

Being a web-based front-end system, it should be accessible through an HTML interface from any machine connected to the Internet and, hence, accessible to all authorized persons irrespective of the geographic location.

- The proposed system should streamline as well as simplify the business processes in customer service and collection businesses. The system should provide a seamless integration between collections, customer service and consumer finance businesses.

The proposed system should be integrated with the existing components (such as the automated dialling software), existing databases (such as FDR, MIDS) and with all the other call management systems (such as Avaya).

The entire functionality of the application was divided into a collection of use cases. Each use case typically represents a piece of requirement to be fulfilled and may include complex computations and functionalities and/or may depend on other use cases. Some of the use cases, such as searching a customer's financial profile, were commonly used in many other use cases. There were 23 use cases from the collections services and 36 use cases from the customer services. Exhibit B and C give some of the important features (functional and non-functional) of the proposed system.

Exhibit B Important features of the proposed system

- The system should provide ample scope for future enhancements, such as integration with other consumer finance services like banking, as well as more effective and efficient roll out of future enhancements of existing services.
- The system should aim at providing accurate, consistent and up-to-date information to customers, retailers and other authorized knowledge seekers.
- Policies that change depending on the location of the store, especially those related to the government (e.g., local tax), should be handled appropriately. It should also provide support on issues such as consumer rights and hence should improve statutory compliance.
- It should also collect and manage the information related to the performance of the call centre team. This accurate and up-to-date management information may be used for effective process enhancements. The system should ensure a standardized method of collecting and reporting the following data:
  - Agent activity (time stamps for login/logoff/Break)
  - Account interaction data (date, time, inquiry channel, agent-ID, etc., for screen pop, mail, fax, etc.)
  - Details of the interaction (tasks performed, changed values, time taken for each task, etc.)
- The system should reduce the support requirements. Thus, it should be a system that can be used with automatic application of business rules and incorporated with an online context-sensitive help. There should be facilities for front-end customization in order to provide flexibility.
- The proposed system should eliminate the requirements for multiple skills and simplify the job process.
- It should improve standardization amongst the call centres that are present in different parts of the world. Unlike the current application, which behaves as a standalone application at different locations, this should be manageable and controlled from a single location.

Exhibit C Non-functional requirements of the proposed system as given in the SLA

A part of the service-level agreement (SLA), where the performance requirements are explicitly mentioned

Inbound calls per hour	50,000
Transactions per call	5
Outbound (dialer) calls per day	150,000
Users	4,000

Table 1 Estimated workload for software product

Maximum allowable downtime per month, exclusive of scheduled downtime	30 minutes
Maximum allowable downtime per incident, exclusive of scheduled downtime	8 seconds

Table 2 Software product availability

Data integrity accuracy	100%
Calculation accuracy	100%
Table-driven, business-rules-based accuracy	100%

Table 3 Software product workflow execution

- The system should deliver a minimum rate of 24 transactions per second and 95 per cent of these transactions should not take more than two seconds for complete processing.

Glossary

Service-Level Agreement (SLA)

An SLA is a formal contract between customers and the service provider, which has been negotiated earlier. The purpose is to document the common understanding between both parties about the level and quality of services to be provided along with priorities, responsibilities, guarantees and penalties.

The project was a very ambitious one and needed high levels of skill at both the technical and the managerial fronts. For any software services organization, this was a big opportunity because of the size of the project and the opportunity to showcase McCombbs, a world leader in its domain, in its client list.

McCombbs decided to get this important application developed by the biggest and the best software services provider—naturally, the project was awarded to LQNH, a leading business and software consulting firm. LQNH started this project very well and even implemented the system to a large extent. But they could not deliver on both the expected functionalities and the non-functional requirements. The system developed by LQNH failed during system testing.

McCombbs now decided to award the project to AnderAccess, another big player in business and technology consultancy. AnderAccess and McCombbs decided to start from the architecture given by LQNH. The architecture proposed by LQNH was acknowledged as being very good.

AnderAccess made some minor modifications in the architecture and started working on the project. However, McCombbs had to pull back the project as they found AnderAccess' services to be overpriced. McCombbs called for open-bid proposals from software service providers throughout the world. Fact-Tree won this fixed-bid project.

Knowledge Transfer

The business domain knowledge was transferred to Fact-Tree by the AnderAccess team over a period of two months primarily through conference calls. In these calls, the AnderAccess team of business analysts would facilitate transfer of knowledge about the business to their counterparts in Fact-Tree. Sometimes McCombbs representatives also would join the conference call to resolve any issues or concerns. Domain knowledge was transferred in two parts: the business requirement aspect and functional design aspect.

Some of the major challenges faced during the transition of business requirements were the following: the Fact-Tree team did not have access to the current system and the transition was done in an ad hoc fashion rather than proceeding according to a logical business flow. For example, even though use case 'UC2' depends on use case 'UC1', knowledge of UC2 was transferred before UC1.

As mentioned earlier, most of the modules of the system were categorized into use cases. There were 23 such use cases in scope that were transferred to Fact-Tree. Each use case captured a simple functionality such as *Lost Card, Card cancellation, Balance enquiry, current status* and *Auto-dial*. After the business knowledge transition, the Fact-Tree team was responsible for the preparation of the technical design of the use cases. It was agreed that only after sharing the use case design with AnderAccess and getting approval would the McCombbs team sign off the design.

Initially, McCombbs asked Fact-Tree to seek sign off from AnderAccess on the transfer of knowledge. However, as work progressed and looking at the efficiency of the teams involved, McCombbs asked Fact-Tree to sign off the transfer. The application is named 'Rover Workstation'.

Architecture

Fact-Tree also stared the development with the architecture that was initially designed by the LQNH team. Figure 3.1 shows the high-level architecture exactly as given by LQNH. This architecture includes all the hardware and software components that are not directly a part of Rover Workstation, the actual application.

The following are the different components along with platform-specific details in the system described by the architecture in Figure 3.1[1]:

- Computer telephony interface (CTI) components (IVR, ACD monitor and dialler)
- CTI servers and services (KANA API)
- Web server, DB server and app server services
- Middleware (MERCATOR)
- Gateways (TANDEM and ODS Gateway)
- Backend databases (MIDS, NOTES DB, FDR, KANA DB)
- LDAP services for authentication

Computer Telephony Interface Components

Interactive voice response is used to receive inbound calls and play the auto-prompts. It can direct the user's enquiry for balances, listen to the list of services offered and connect with live agents to speak with, if needed.

When an inbound call is received, the IVR will prompt for caller account details; based on the account number provided by caller, it retrieves minimal data of the caller from the database. Once the details are retrieved from the host, the call is routed to an available CCE.

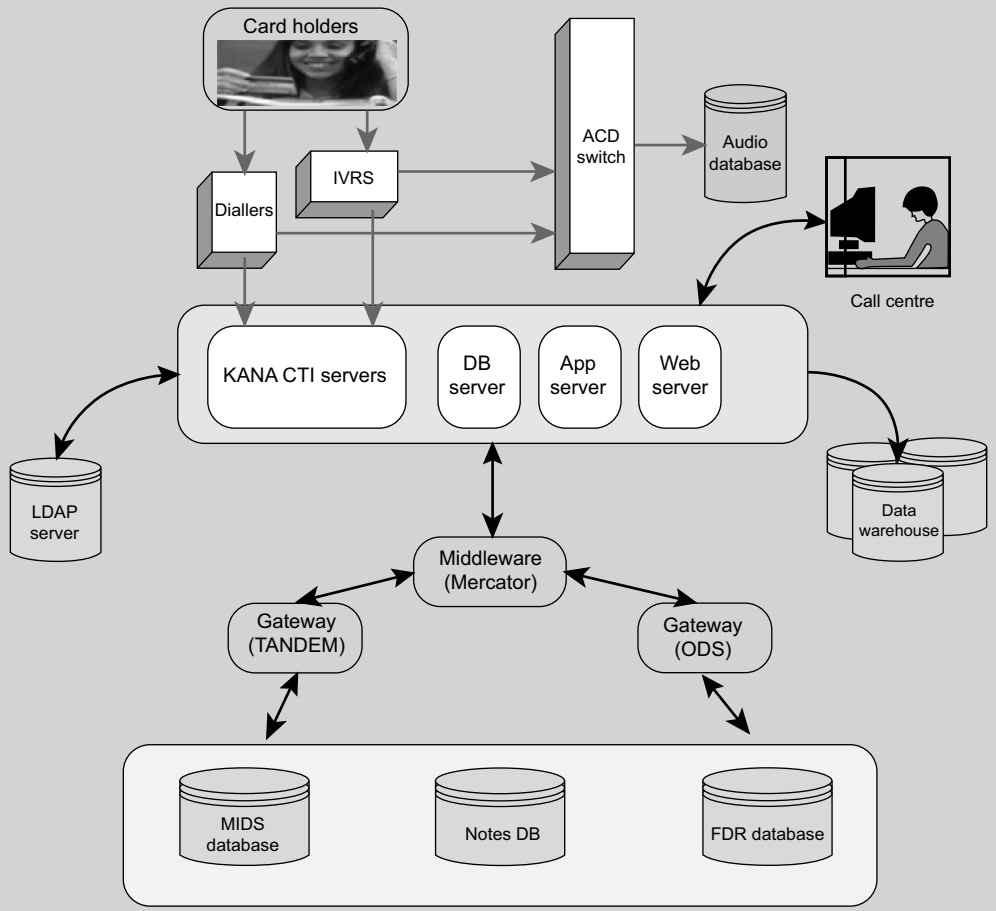


Figure 3.1 High-level architecture of McCombbs' call centre software system

Automatic call distribution (ACD) monitor: The GEOTEL ACD monitor is used to monitor the ACD for inbound call routing along with minimal screen pop up data. This has also been used for making manual outbound calls through workstation clients.

Dialler: A UNISON Davox dialler is used to make predictive calls based on the call list allocated from a given campaign. This is also known as the auto-dialler.

Computer Telephony Integration Server and Services

CTI components deal with data that are in the form of voice. The voice data need to be converted to a form that can be comprehended and processed by the App Tier.

The CTI services convert the data from the CTI components in a form compatible with the Rover Workstation application and send it to the App Server. For this purpose, the voice data are communicated using the XML-based simple object access protocol. This voice XML is parsed at each end of communication and the resultant data are used in further processing.

LQNH selected KANA server and services to perform this arduous task after evaluating various options that were commercially available in the market.

Web Server, App Server and DB Server and Services

Users from the call centres interact with the system by sending requests to the web server. The web server contains the presentation logic and hence forms the web tier part of the architecture and interacts with the other parts of the system on behalf of the CCE.

The DB server stores transactional data and the data to be used for reporting purposes as well as some static data, which are non-critical business data.

The application tier (app tier) consists of all the business rules and logic and performs the actual processing of data guided by the applicable business policies. The app tier is contained in the app servers.

The app tier components communicate with middleware for data from backend hosts, such as the MIDS or FDR, which contain critical business data, such as financial profiles of customers and their accounts. The app tier also communicates with the LDAP services to fetch the authentication and authorization data. A third-party component called *SiteMinder* searches the LDAP for the authorization data for the CCEs and administrators.

These three servers (the app server, the web server and the data server) are located in a geographically distributed environment, each hosted on a number of machines, facilitating communication with each other. Periodically, the data warehouse of McCombbs is also updated using the same application. The three servers, the data packets for middleware and the CTI app form the Rover Workstation system boundary.

Middleware

Middleware in computing terms is a software agent that acts as a mediator, or as a member of a group of mediators, between different components in a transactional process. In this architecture, it acts as interface between the backend and the workstation. Middleware will exchange packets and also route these packets to a specific backend database (MIDS, FDR or NOTES DB) depending upon the nature of the request from the workstation. MERCATOR is used as the middleware in the Rover Workstation.

Gateways

Gateways are avenues through which middleware will communicate with backend hosts such as MIDS or FDR. TANDEM and ODS are the two gateways used in the application.

TANDEM is used for communication between the middleware and the MIDS backend. Connection to TANDEM is through a simple socket from the middleware. ODS is used for communication between the middleware and the FDR backend. MQ Series web services from IBM are used to facilitate communication between the source and the destination on this path.

Backend Databases

The Rover Workstation application depends on a number of external data sources for its functioning. Each data source serves a specific purpose and has its own data format and hence needs an adapter that will convert the incoming and the outgoing data into the appropriate format.

- *MIDS*: MIDS is an integrated data management system. This is maintained by McCombbs internally and it stores the security profiles.
- *FDR*: FDR (First Data Corporation) manages portfolios for cardholder clients from retailers such as JC Penny, Wal-Mart and MCS. FDR maintains all account information for these clients' cardholders.
- *DB Notes*: Each CCE maintains some notes about the customer handled by him or her. These notes are stored in the DB Notes database.

The CTI components, middleware, databases and the various gateways are being used by the McCombbs system for a long time and the new system has to maintain the usage.

Logical Application Architecture

The logical view of the application architecture is shown in Figure 3.2.

The major components described are the following:

- Browser interface (HTML and web scripting language)
- Session manager component
- Model view controller (MVC) architecture (model view and component)
- Rover Workstation adapters

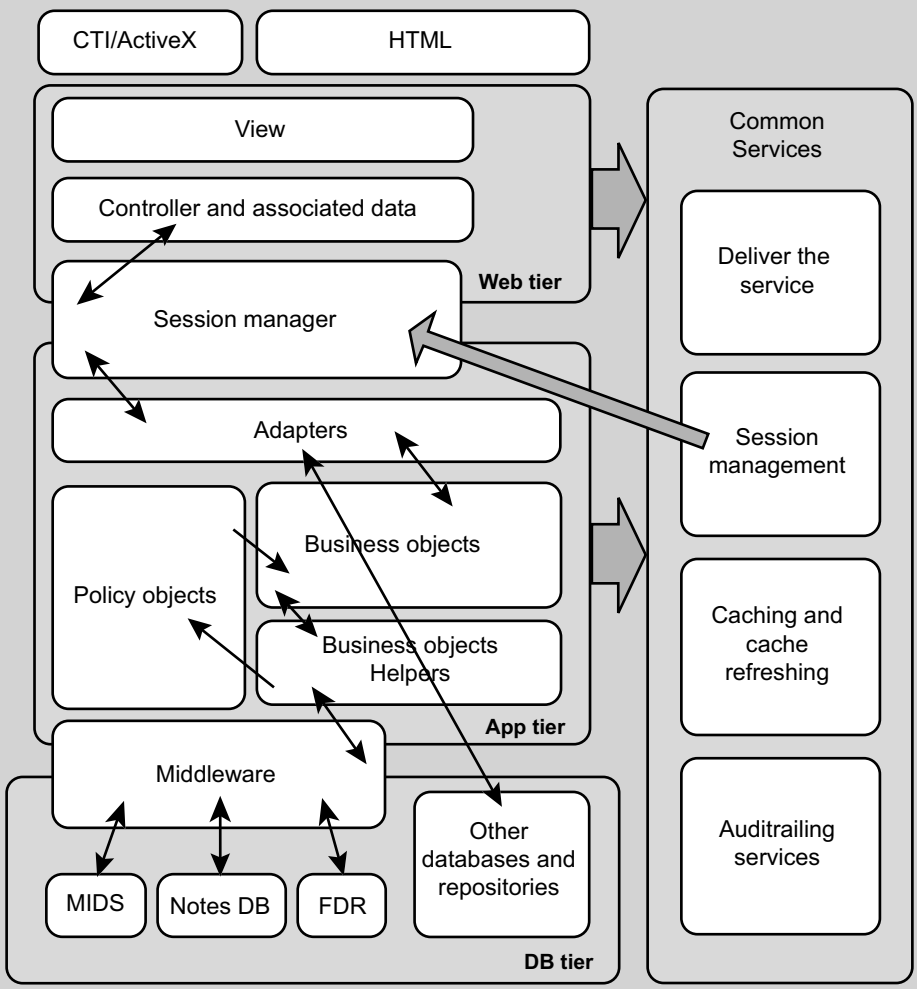


Figure 3.2 Logical diagram for the Rover Workstation

- Business object and business helper objects
- Policy objects
- Common services

The Web Tier

The web tier of the architecture is based on the MVC pattern. Here, the view manages the display of data. The CCE generates events using the HTML browser. Each event is associated with very specific processing, which is decided by the controller, which is also present in this layer.

The controllers package the processing request and associated data as a business object and send these business objects to the app tier for processing. When the processing is completed, it marshals the processed results from the app tier to display them. The web tier will apply the navigation and data validation types of rules.

The Application Tier

The app tier forms the actual model for this MVC architecture. It consists of the processing logic, business rules and some other functional components needed to support business logic processing.

Usually, minimal data are requested from the user, approximately 2-3 per cent of actual data needed for complete processing. A session is created and the rest of the data are fetched from the database (either MIDS or FDR). The session manager component performs session management in this tier as well as in the web tier. These data are then sent to the appropriate business logic component in the form of a business object after combining other data that are needed for the completion of the requested service. After the policy objects do the processing, the app tier adapters format the results according to the presentation needs and pass it on to the web tier.

Session Manager Components

Session management is done at two levels in the Rover Workstation. When a CCE logs in, a session is created. This session remains active till the CCE logs off. A unique session ID is assigned to this session.

When a customer calls the call centre and request for services, a call-level session object is created. This object contains the various business objects and other associated data along with the client data. This object is alive for the duration of the call only.

Each session may be having a number of transactions within it. For each of these transactions, a transaction-level session object is created. Such a transaction-level session object persists only till the particular transaction lasts.

The session objects are passed on to the adapters in app tier for further processing. The server keeps on broadcasting this object on the network, and whichever application needs the session object, it picks it up for processing. This model is highly suitable for applications where the volume of users is small.

KANA API and KANA server were selected as the session management components. The KANA server is placed between the app tier and the web tier and communication takes place using HTTP.

Adapters

The adapters act as the single entry point for the app tier. A facade pattern is used for this purpose. Adapters carry the business object from the web tier to the app tier (and vice versa) using messaging services. An acknowledgement is sought after the message has reached its destination.

Business Object and Business Helper Objects

Business objects contain the business data and hence are treated as parameters. These are processed in accordance with the business policies (from policy object). For example, credit balance refund amount authorizations may vary according to client. There are some support objects for the business objects. These support objects are called *business object helpers,* which fetch appropriate data from the database by communicating with the database.

To look up the parameter values the adapter needs to supply the business rule ID, the client ID and the name of the parameter. The parameter for that specific combination and client rule is then retrieved. A given rule ID may have multiple parameters that are to be retrieved. These parameters are all keyed to the same rule ID in the database. Parameter-driven business rules are not implemented at the HTML/scripting and model level.

Policy Objects

Policy objects contain the policies or business rules adopted by clients. Examples of such policies include '*the time period before which the sold goods can be returned*', '*kind of goods that are not returnable'* and policies regarding '*earning store points on promotion goods'*. By their very nature, these policies vary for each client and sometimes for each location.

Policy objects take the business objects as input and convert them to value objects after appropriate processing. Then this value object is passed back to adapters for passing them on to the web tier for presentation.

Common Services

Common services capture the functionality that is being used by the entire application and is not specific to a particular architectural layer such as the web tier or the app tier. The important common services that are being developed and used in the Rover Workstation are the following:

- Caching services for business rule objects and some other static data at server start up and other points in time. The system configuration is also cached at server start up.
- Audit trailing services, which log the agent communication data, event processing data, and so on.
- Deliver the service (DTS) for the web tier, where the navigation and the data entered by the CCE are logged in as session cookies and later stored in a repository. *ClickStream* is used to keep track of all input/output requests, data flowing in and out of each screen as well as the time spent on each screen. Its major purpose is to help strategic decision making as well as streamlining the functionality of the system. For example, the functionality used more often can be kept at front when compared with the other less used functionalities, such as issuance of cheque books.

Security—Authorization

As the application is handling financial data, security is of utmost importance at all the levels in the application; hence, a stringent security policy is adopted. Access to the Workstation logon page is secured through a *SiteMinder* agent authenticating the user against their single sign on ID (SSO ID). Whenever a CCE logs into the system, he is assigned an SSO ID. After access to the logon page is granted, the SSO ID is used to look up the user's profile, which contains IDs and passwords for other systems they need access to (e.g., Diallers, Geotel, MIDS, FDR, etc.) The SSO agents run on each workstation web server and connect to the corporate SSO LDAP servers.

Communications between the desktop and the workstation servers are encrypted.

Glossary

Single Sign On

SSO, also known as enterprise single sign on (E-SSO), is a software authentication module or application that enables a user to authenticate once and gain access to the multiple software systems that are related.

Current Scenario

The Fact-Tree team started the project with the architecture provided by LQNH. As their familiarity with the architecture provided increased, they felt the need to make several changes to the architecture, both critical and non-critical, in order to meet specified requirements. Some of the major problems that the Fact-Tree team is facing are as the following:

- *Scalability* is one of the major problems with the architecture. It works well for a limited number of users, but with time and with an increase in the number of simultaneous users, the system breaks down.
- *Overloaded network* is observed often. When an analysis test was performed for network traffic, it was found that 20-25 per cent of the total network traffic was from the broadcast done by the session manager component. For example, lots of details about the client flow through the network whenever the client's details are to be retrieved or processed.
- *Session management* is another area that needs much attention and tuning, as discovered in a post-mortem analysis. After the session objects are created, they are on the network all the time, leading to an overloaded network.
- *Response time* is slow. XML parsing is becoming a big overhead. Messages are being parsed twice, both at the sender's end and at the receiver's end. This makes data display as well as further processing slow. It was also observed that each packet waits for an acknowledgement for successful delivery from the receiver. This not only adds to the network traffic but also reduces the response time considerably.
- *Reliability* is a big challenge right now. In case of some hardware or software failure, there is no mechanism to ensure that the transaction that is under progress during the failure is completed. The CCE who initiated this transaction is not aware of the status of the transaction—thus, he is not sure whether to reinitiate a transaction or not.
- *Low transactions per second (TPS)* is a major hurdle that needs to be taken care of. Low TPS will mean that the system totally fails in the performance test.

Apart from these shortcomings, it is felt that there is ample scope for performance enhancement and that a number of common services are still left out.

Solve these Problems and Make the Project Successful

The Fact-Tree team is on the look out for suitable changes in the architecture that not only eliminate these existing problems but also bring in other changes that may improve the performance manifold. This improvement is essential to make the project successful and win the confidence of McCombbs.

Start with the initial architecture and look at it from all angles (try to come up with all 4 + 1 views). See whether the given architecture is represented in correct notation.

The following are some guidelines to help you analyse the problem better and come up with a solution:

1. Observe the strengths and weaknesses of this architecture.
2. Evaluate this architecture from various non-functional requirements perspectives, including reliability, availability, scalability, serviceability, security and performance. (You may want to do a detailed analysis from each of these angles based on the details given in the case study description.)
3. Analyse the interdependencies between these elements.
4. Suggest strategies and architectural changes to improve each of these.

POSTMORTEM

In this case study, we have seen several good practices. Readers can observe and learn a great deal about architectural principles and designing by looking at the initial architecture given by LQNH. These include the various design patterns used (e.g., facade, MVC); designing of multi-layer, multi-tier architecture; creating common services that can serve various components residing in various layers; establishing communication and designing interfaces between neighbouring layers; designing the workflow within the application; and dividing the functionality between the main components and the helper components.

Before analysing the case further (and to appreciate the given architecture better), let us look at some of the fundamental concepts in software architecture that are relevant to this case study. We shall discuss the architecture goals and drivers, architecture patterns and anti-patterns (with a detailed discussion on MVC patterns), architecture re-factoring and application of performance engineering techniques to identify and remove bottlenecks in software application. Later, we shall analyse the case study further and come up with some answers to the problems posed. We will also list the best practices in building high-performance systems at the end of this chapter.

SOFTWARE ARCHITECTURE GOALS AND DRIVERS

Any proposed architecture should fulfil certain objectives. Fulfilment of these objectives/goals determines the quality of the architecture; hence, these goals are also known as *quality attribute goals*. These inherited properties of the architecture guide the creative process of designing the architecture, coming up with a detailed design and implementing and even evaluating or validating the architecture. They even help elicit clearer requirements.

Quality attribute goals are the means by which a system is intended to meet its business goals. These requirements must be recognized and considered early in the life cycle, and the system and software architectures must be designed in such a way that the quality attributes are met. To reach the quality attribute goals, one must understand the architectural drivers. Architectural drivers are the keys to realizing these goals in a system. For example, the need to completely control one's own information can be one architectural goal, thus making *security* the architectural driver for this system's architecture. Working on a system that is easy to change later can be another architectural goal, making *modifiability* the architectural driver. Thus, architectural goals and drivers can be viewed as a set of *ends* and *means* for a software system, respectively.

Identifying architectural goals should be the first step towards any set of actions and designing architecture is no exception. A clear definition of the architectural goals and drivers is a critical ingredient for the success of a system. Architectural drivers impact the design decisions, which further impact the implementation, testing and even evolution of the software system.

SOFTWARE ARCHITECTURE PATTERNS AND ANTI-PATTERNS

Successful engineering is based on the application of concepts as well as best practices and not necessarily on the contemporary and latest practices. For the process of designing architecture, best practices come in the form of *architecture patterns*.

An excellent definition for architectural patterns is given by Frank Buschmann and colleagues in their book *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns* as

> An architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them (Buschmann *et al.*, 1996)

Every pattern is a reusable solution for a recurring problem. Hence, all good patterns have the following properties:

- Patterns are recurring in specific contexts
- Patterns help in reducing the complexity of the software system as they act as building blocks, which can be used to build more complex systems
- Patterns help in coming up with a more reliable software system
- Patterns provide skeleton solutions that can be adopted in a given context
- Using patterns is a good way to modify the system in the future as they document the architectural vision in a crisp and clear manner
- Patterns help in communicating the architecture much more easily
- Patterns are higher level abstractions when compared with classes and objects
- Patterns consist of a set of components and how they interact and collaborate with each other
- Patterns document responsibilities of each of the components
- Patterns are derived from real-world contexts; they are not synthesized from artificial problem spaces

In the pattern-oriented software architecture literature, as few as 79 architectural patterns have been identified. Applicable under a set of conditions, each represents a best practice.

An architectural pattern is a high-level abstraction. The choice of the architectural pattern to be used is a fundamental design decision in the development of a software system. It determines the system-wide structure and constraints the design choices available for the various sub-systems. It is, in general, independent of the implementation language to be used.

An example of an architectural pattern is the pipes and filters pattern. In UNIX, for instance, a filter is a program that reads a stream of bytes from its standard input and writes a transformed stream to its standard output. These programs can be chained together with the output of one filter becoming the input for the next filter in the sequence via the pipe mechanism. Larger systems can thus be constructed from simple components that otherwise operate independent of one another. As discussed in Chapter 1, architectural styles are considered architectural patterns. For instance, model view controller (MVC) architecture is said to be following the MVC pattern.

MVC Architecture

The MVC architectural pattern decouples data presentation, user interaction, data access and business logic. This facilitates the division of the application into logical components that can be designed and developed independently. This division increases the reusability of components by reducing the couplings between them.

This MVC pattern divides an interactive application into three components—model, view and controller. The model contains the core functionality (business logic) and data. The model is independent of specific output representations or input behaviour.

Views display information to the user. They obtain the data from the model. There can be multiple views for a single model. Each view has an associated controller component. Controllers handle user

input and manage the coordination between the model and view components. They receive input, usually as events that encode mouse movement, activation of mouse buttons or keyboard input. Events are translated to service requests for the model or the view. The user interacts with the system solely through controllers.

Views and controllers together comprise the user interface. The separation of the model from the view and controller components allows multiple views of the same model. If the user changes the model via the controller of one view, all the other views depending on these data should reflect the changes. The model, therefore, notifies all views whenever its data change. The views in turn retrieve new data from the model and update the display.

The major advantages of the MVC architecture are the following:

- Enhanced maintainability and extensibility of the system
- Exchangeability of look and feel framework
- Potential multiple views of the same model
- Pluggable views and controllers
- Reusability
- Synchronized views

But the MVC architecture also has some disadvantages:

- Complexity of design and implementation
- Data transformation overhead
- Observers are unaware of each other, causing problems while changing the state of the subject
- Potential for excessive number of updates
- Thus a trade-off is involved when the MVC pattern is applied in designing the architecture for enterprise application.

There are several resources for understanding MVC architectures better. Among many useful resources, we recommend Sun[2] and Apple's Developer[3] Web sites.

Anti-patterns

Though software are being developed for many decades now, software projects still suffer from basic fundamental mistakes that are repeated. Recently, these fundamental problems and their potential solutions have been crystallized into the concept of *anti-patterns*. According to the book *Anti Patterns—Refactoring Software, Architectures, and Projects in Crisis*,

> An Anti-pattern is a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences (Brown *et al*., 1998).

The starting point of the anti-pattern definition process is an existing problematic solution. This anti-pattern solution generates negative consequences to management and software development. There is a second defined solution: a *re-factored solution* is the goal of the anti-pattern application process and provides means for the project to direct the project towards success.

Thus, anti-patterns define a common and formally defined toolkit for categorizing, identifying and mitigating the typical problems in software development (Kärpijoki, 2001). We can call them excellent tools for 'learning from mistakes'.

Anti-patterns have been created to serve many purposes. They can be applied to totally prevent some of the chronic problems complex software development projects face. More often anti-patterns are, however, applied as counter-measures to provide a proper re-factored solution to existing problems the organization is facing. Anti-patterns also provide managers and developers the means to *identify* typical problems, to classify them with a common vocabulary and to discuss them together in a constructive way (Kärpijoki, 2001).

Re-factoring

Re-factoring is generally the process of re-creating material already available to improve its usability, efficiency or structure, with the explicit purpose of retaining its meaning or behaviour. This term in software engineering has been often used to describe the modifications in the structure of source code without changing its external behaviour and is sometimes informally referred to as 'cleaning it up'. Re-factoring also implies that sometimes one may have to replace a module with another existing or new module with same behaviour.

While patterns illustrate best practices, anti-patterns illustrate what not to do and how to fix problems. Essentially, the solution to anti-patterns is called re-factoring. Re-factoring the architecture is a transformation that improves the quality and preserves the semantics. Thus, the essence is to improve the internal structure while retaining the behaviour. The changes essentially take place in the design of the code. A simple example would be of converting a class that uses case statements into the one that uses *StatePattern*. Or suppose one wants to convert a class that hard codes the classes of its products into one that uses factory methods, and then one decides to pull those factory methods into an abstract factory.

Architectural anti-patterns have been successfully deployed to re-factor architectures for performance. Anti-patterns document common mistakes made during software design, which experienced software architects have found useful to document. We will see in detail how architectural anti-patterns can be used to move towards performance-oriented designs.

But anti-patterns should not be applied unless there are clear reasons to do so. In other words, 'if it's not broken, don't fix it'.

PERFORMANCE-ORIENTED DESIGN

We devote this section to an important architectural attribute: performance. The importance is realized when we analyse the case study in the next section, wherein will realize how various design considerations are actually related to performance. We shall rate each design consideration and see how well it fits into the current architecture. This exercise is very important to make changes in the existing architecture.

Performance Objectives

Before analysing the performance of any computing system, one must establish a measurable criterion for performance. The purpose of performance objectives is twofold:

- To state what is expected of the system in specific terms for each workload category (e.g., trivial versus nontrivial transactions) at distinct time periods (e.g., prime shift, off-shift and peak periods within each shift)
- To understand and document the resources required to meet the objectives

From the nature of these two goals, establishing performance objectives is an iterative process. Expect to update the performance objectives as the workload, resource requirements, turnaround and response

time requirements change. Performance objectives should be a measurable criteria, such as response time, throughput (how much work in how much time) and resource utilization (CPU, memory, disk I/O and network I/O).

In this case study, the performance objectives are well specified. The SLA gives the performance objectives. All these are measurable and quantified, which includes throughput, response time and data accuracy.

Performance Improvement

How well does an application perform? It is probably one of the toughest questions to answer accurately. It is not only a question of how many requests the application serves per second or per minute but also how it scales with respect to other performance metrics. It remains challenging to quantify application health quickly, because there are a number of variables to consider.

For large and complex systems, architecture and design plays an important role in defining the performance. Tuning of performance, so that it complies with the SLA requirements, is performed in many steps (see Figure 3.3). These include the following:

1. Get a proper set of architectural documents
2. Identify key performance scenarios
3. Identify problem areas/bottlenecks
4. Refine the system by addressing design and architecture performance issues, including hardware and operating system configuration; test the tuned system
5. Repeat the steps until the desired performance levels are achieved

Performance improvement is not necessarily done at the end of the development phase. If it was done at the end, it would be like a build-and-fix cycle. Note that it is not always possible to improve performance when the entire development exercise is over. A check is to be exercised on the performance from the very beginning.

We now look at some of the important steps in the process of performance improvement. This approach is motivated by the SPE process proposed by Connie Smith and Lloyd Williams in their book on performance solutions (Smith and Williams, 2002).

Understanding Architecture

To initiate the performance-improvement process, complete understanding of the architecture under study is required. The first step in this process is to collect all the architectural documents. Different facets of the same architecture that are portrayed by the 4 + 1 UML views prove to be useful for understanding the architecture. While looking at these details, it is important to keep the scope of evaluation clear and focused, otherwise one can get overwhelmed by the amount of available information. We also need to define what is being meant by bottlenecks. The performance requirements in the SLA can help decide performance variables. The expected outcome at the end of this exercise is get a set of performance variables that are to be investigated or studied. We need to ensure that these variables are measurable as well as testable for given targets.

Identifying Key Scenarios

After agreeing on the variables to be analysed, a set of use cases can be created. The objective is to identify key performance use cases and scenarios. The need to focus on representative use cases means that only a

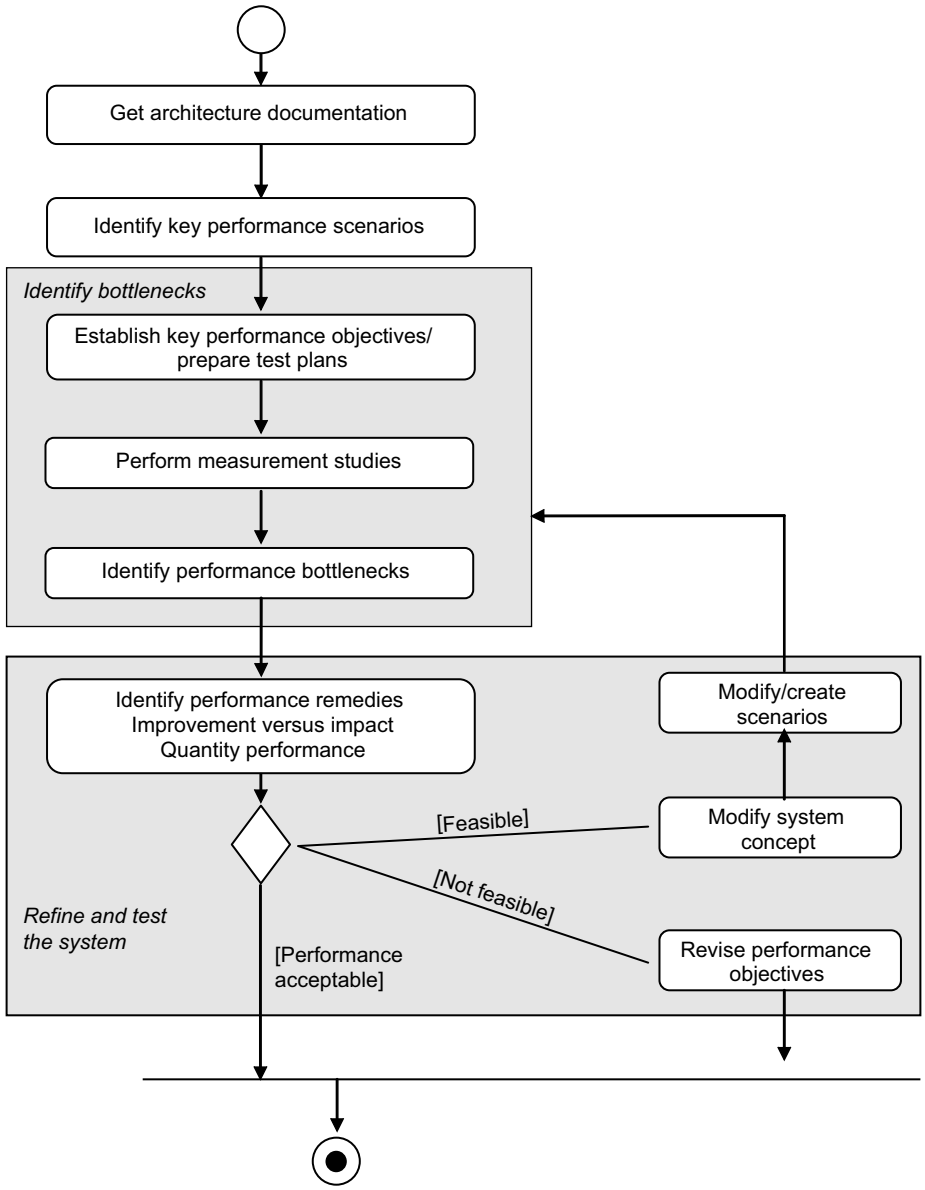


Figure 3.3 Performance-improvement steps

limited number of use cases can be considered; also, the priority of their selection must be clearly stated. Consider the following types of scenarios:

- Those executed more frequently
- Those executed less frequently but that have a great impact on the system
- Those whose performances are critical even though they are not executed frequently

Identifying Problem Areas or Bottlenecks

It is typical to observe that one part of the system is always slowest, which we call as bottleneck. Until it is remedied, no other tuning will actually improve the overall performance of the application along that path. Before the bottleneck can be removed, it must first be conclusively identified. Once a bottleneck is identified, resolution can be reached more quickly if the existing tests are modified to eliminate distraction from ancillary issues. Pinpointing exactly where the bottleneck is not an easy process, but the skill is acquired with practice and patience.

Bottlenecks can be identified in various ways. There are a few excellent observations that can help us identify bottlenecks quickly. The following rules are discussed for identifying bottlenecks (Smith, 1990):

- Bottlenecks do exists even when the system is not loaded. It is a mistake to assume that bottlenecks are created only when the system is loaded. The symptoms of the bottleneck are virtually never observed at the actual location of the bottleneck.
- A critical bottleneck is one that is along a particular user path. The removal of such a bottleneck will improve both performance and the ability to find other bottlenecks.
- If one has multiple paths through a system and assume there is a bottleneck, one should isolate each path and evaluate it separately.

Though finding out the right set of bottlenecks is not an easy process, it is not rocket science either. There are various ways to pinpoint bottlenecks:

- Questioning experienced users and system administrators
- Creating and analysing system performance data and reports (measurement studies)
- Using anti-patterns
- Reviewing the system
- Performance modelling

We will highlight each of these methods and discuss how to apply them in our case later.

The basic approach is to theorize first and then test the theory. In other words, once we have some basic idea, we can theorize the reason for a performance bottleneck. But that is not enough. This has to be verified or disputed and data should be collected for this purpose. These can be logged data (audit data collected by the application), some simulation model, and so on. Even stubs or instruments can be inserted into the code to generate a trace for performance variables such as the response time and throughput. The log files, which are events and their stamps, provide a snapshot of the system at run-time in the determined scenario. This is a dynamic reconstruction of the architecture. The reconstruction will be more or less complete depending on the variables traced. A note of caution is that enabling the tracing of many variables influences the normal activity of the software, which can produce misleading outcomes.

Instrumenting or inserting stubs in the software is an intrusive technique. Enabling the tracing of many variables affects the measurements when a relevant part of the CPU time is spent processing the instrumentation source code.

We then need to analyse these data to refine the theory. This step of theorizing, collecting data and refining can be done in an iterative way, at least for a couple of times, so that we can locate just the right bottleneck and spend our time and effort in the right direction.

Once the bottlenecks are identified, the next step is to fine-tune the existing system and remove the bottlenecks. This can be done in either of the following two ways:

- The architecture is optimized at a higher level of abstraction, that is, searching for the design anti-patterns and re-factoring.
- Inspect specific parts of the source code and adjust them, that is, fine tuning.

Note that refining is not the end. The system is to be tested if the performance objectives have been met; if not, the cycle has to be iterated. This may also include modifications in performance objectives in light of the trade-offs and improvements observed so far.

We will see the major steps in detail now.

Depending upon the availability of information and resources, different methods can be employed to pinpoint bottlenecks. Perform some analysis and measurement studies. This can include review and analysis, building and evaluating performance models, analysing logs and performance reports, and so on. One may also want to build some performance tests such as load or stress tests. For this, automated tools are of great help.

As discussed earlier, there are several techniques such as asking questions, measurement studies, using anti-patterns, review of the system and performance modelling that can be used in identifying bottlenecks. We shall discuss some of these techniques in detail.

Asking questions

Asking right questions will always help in narrowing down the focus and identifying problem areas. In fact, this technique of asking questions can be a good starting point. It is not easy to come up with a set of relevant questions that will help our cause. However, expert architects have a common set of questions that make sense and are typically are important. These questions fall under several categories. Here we list a few such questions, though not exhaustive.

Questions about the nature of the problem:

- Has the software component (or a specific server) always been slow? Or has it become slow when some thing has changed?
- Is the problem situation repeatable or random?
- Does the server become slow when more users are added?
- Does the server become slow when more data are added? (Assuming number of users has not increased drastically.)
- Does the server become slow when the number of transactions has increased? (Assuming that data and number of users have not increased drastically.)
- Is there any change in the software version? Is it upgraded?
- Is there any change in the hardware configuration? Is it upgraded?
- Is there any new functionality added to the software component?

Questions about the chronology of the problem:

- Does the problem occur during a particular time of the day?
- Does the problem repeat when a specific set of events repeat?
- Does it occur during peak time or during idle time? Does it occur during maintenance cycles?
- How long since the server has been up?

Questions about the topography or location of the problem:

- Is the problem specific to an application or a server or the network?
- Does the problem occur on all applications running on a given server? What about other servers? Are there any dependencies on other servers (database servers, application servers, file servers, etc.)?

- What are the kinds of servers on which these problems occur? What is the operating system and what is the configuration of these machines? Do all the problem servers have the same OS and/or configuration?
- If the problems are specific to a given network, is it meant for a group of users or a physical location? What type of users are they?

Questions about the relation between the problem and other actions:

- What is the problematic area of the application?
- Is there a performance slowdown while connecting with the database?
- Is there a performance slowdown when a user is opening a specific screen or a view? If so, what are those views?
- Does the system get into a problem mode when a user is adding new data?
- Does the system get into problem mode when a user is deleting data?
- Does the system get into problem mode when a user is modifying data?
- Does the system get into problem mode when a user is opening a file?
- Does the system get into a problem mode when a user is saving a file?
- Does the system get into a problem mode when a user is editing a file?
- Does the problem recur when a user clicks a particular button?

As you can see these are very useful questions to ask to gain insight into the kinds of possible bottlenecks. Often, performance problems were solved just by asking right questions and addressing the issues that came up. This is, however, a very unstructured approach to find out performance bottlenecks.

Performance modelling

Performance modelling is a more structured and repeatable approach to modelling the performance of the software. It begins during the early phases of the application design and continues throughout the application life cycle. This is a proactive approach where the performance models are created and the application scenarios and performance objectives are identified.

Modelling allows one to evaluate the design before investing time and resources to implement a flawed design. Having the processing steps for the performance scenarios laid out enables one to understand the nature of the application's work. By knowing the nature of this work and the constraints affecting that work, one can make more informed decisions. The performance model can reveal the following about the application:

- What are the relevant code paths and how do they affect performance?
- Where does the use of resources or computations affect performance?
- Which are the most frequently executed code paths? This helps one identify where to spend time tuning.
- What are the key steps that access resources and lead to contention?
- Where is the code in relation to resources (local, remote)?
- What trade-offs does one have to made for performance?
- Which components have relationships to other components or resources?
- Where are the synchronous and asynchronous calls?
- What is the I/O-bound work and what is the CPU-bound work?

The model can reveal the following about the goals:

- What is the priority and achievability of different performance goals?
- Where did the performance goals affect the design?

It is very important to understand that the upfront performance modelling is not a replacement for scenario-based load testing or prototyping to validate the design. In fact, one has to prototype and test to determine costs and to see if the plan makes sense. Data from the prototypes can help one evaluate early design decisions before implementing a design that will not allow one to meet the performance goals.

A small note here is that in this case study performance modelling may not be very useful because the system is in the advanced stages of development. Modelling is very effective when the system is being evaluated before the actual implementation. However, understanding performance modelling will help us in identifying bottlenecks and address them.

Using architecture anti-patterns

Architectural anti-patterns focus on the system-level and enterprise-level structures of applications and components and help us in identifying the bottlenecks.

The following anti-patterns focus on some common problems and mistakes in the creation, implementation and management of architecture. Lack of commonality between systems in terms of design and technology is the cause of frustrating inability to provide interoperability and reuse between related systems. Improved enterprise architecture planning can be used to align system developments. When anti-patterns are recognized in architecture, re-factoring the system is not difficult, as the exact problem has been pinpointed.

When the bottlenecks and problem areas are known, performance tuning can become focused. Re-factoring will keep the behaviour of the system as the same level, but the overall performance will improve.

It is very important to know that performance is also a user perspective. Performance improvement, though done in small steps, can add up to a favourable user experience. Very minor changes to the systems will be perceived by users as important improvements. For example, when certain objects needed at server start up are cached, the server will load up quickly. At the same time, the caching will reduce the traffic on the network.

Anti-patterns, both design and architectural, come very handy in re-factoring. As they exactly pinpoint the problem, as well as give the solutions to the anti-pattern, the re-factoring job reduces to finding out the Anti-patterns and replacing the related components without the anti-pattern. One can find architectural anti-patterns from the literature. A select list of architectural anti-patterns and re-factored solutions are taken from anti-patterns Web site[4] and are discussed below:

Architecture by implication This is a very common anti-pattern (one example is the case of Assure-Health we discussed in the previous chapter). This anti-pattern is found in systems that are developed without a documented architecture. This happens for various reasons. One common reason is that if a team has developed one or more successful systems in the (recent) past, overconfidence results in assuming that there is no need for complete documentation of the architecture. Another common reason is simply ignorance or insensitivity of the team management, as in the case of our previous case study.

If this anti-pattern is found, the immediate remedy is to stop the rest of the development and work on the architecture and present multiple viewpoints considering all the stakeholders. The absence of documented architecture is often the reason for failure of software projects in spite of having all other ingredients right.

Cover the assets This anti-pattern is found because of risk-averse architects or people who are architects without understanding the purpose of architecture. They prefer to list alternatives instead of

making decisions mainly because they are more comfortable with writing processes that favour fat documentation rather than creating solutions that actually work. They want to avoid the blame for failure and hence prefer to document only the possible options instead of documenting the actual decisions.

To overcome this situation, the purposes and guidelines for the task of documentation need to be clearly established. Once the actual documentation is complete, it can be checked for the value of the documented decisions.

Design by committee Sometimes instead of a single architect, a committee of experts designs the architecture. Most often in these committee-driven designs, every expert member brings in his or her own perspective, emphasizing on a specific aspect or viewpoint of the architecture. This results in designs that are very complex because the members are without a common architectural vision.

A good way to tackle this would be to clearly spell out the architectural vision and then facilitate the committee members with proper software development roles. Once this clarity is brought in and the role of each committee member is clearly defined, much more effective committee-based processes can be expected.

Reinvent the wheel When we need to deal with a large number of legacy systems that are built over the years with overlapping functionalities, we can expect to find this anti-pattern. Many of these systems do not interface with other systems and each system is built in isolation.

When this anti-pattern is found, it is recommended that we select the best system among the available ones and extend its functionality and generalize this system so that a common interface can be defined. This common interface can be used to integrate with the rest of the systems so that the system works without duplication of the functionality.

Spaghetti code As the system evolves over a period of time, it results in an ad hoc software structure that makes it difficult to maintain, modify, extend and run efficiently. The major part of the code base is not understood by the current developers and no one knows why a specific module exists at all. Everyone is afraid of removing or even touching this part of the code because they never know what will be affected.

To handle this anti-pattern, the best advice is to prevent this situation from coming into existence rather than trying to find a cure. Some suggestions are to re-factor the code frequently to improve the software structure and design the software structure in such a way that it can be a good serviceable system. The system's structure should make it easy to perform software maintenance and iterative development.

Stovepipe enterprise Stovepipe enterprise anti-pattern is the result of uncoordinated software systems that are developed in parallel. This results in waste of funds, resources and efforts while designing the systems. After the systems are put to use they are not adaptable, without any possibility of reuse or interoperability.

The stovepipe systems lack abstraction and lead to very rigid and most difficult to maintain architectures.

This can be avoided by planning for orchestrated enterprise architecture to coordinate system conventions, reuse and interoperability. Right levels of abstraction, appropriate interfaces between the subsystems and enterprise-wide coordination among all the systems with proper maintenance of metadata will help overcome this situation.

Swiss Army Knife Sometimes designers tend to provide several interfaces for a given component anticipating all possible ways of using this component. This often leads to unusable components that are difficult to understand, debug and implement because there are too many dependencies. Among all available interfaces, developers get confused as to which one is more natural and more suitable.

It is important to clearly define a component's purpose and provide only the appropriate interfaces in order to manage the complexity.

Vendor lock-in There is a tendency among architects to stick to a platform-specific or even product-dependent architecture without any proper reason. More often than not this kind of dependency on a vendor or a proprietary software results in loss of control and increase in maintenance costs.

It is always good practice to provide a platform- or vendor-independent layer in the architecture that provides product-dependent interfaces. Any change in a specific product or platform can be handled easily, thus making the complexity an easier task to manage.

Performance review of the system

A careful review of the system can give significant information about the possible performance bottlenecks. Reviews for performance usually accompany one or many methods to identify the bottlenecks. Usually, this step is interleaved with re-factoring, as the solution to the problem is also identified and analysed for side-effects as the system is being reviewed.

A scenario-based walkthrough, where a number of scenarios, each being related to the quality attributes/architectural drivers of the system, is mocked to verify if the current system performs the scenario as anticipated. If not, the potential source of problem can be pinpointed.

Another way can be to pick a potential bottleneck or symptom, identify the scenarios, components and processes that contribute to it and then take a complete look of the package to pinpoint the bottleneck.

For example, an overloaded network is one of the most prominent symptoms. Next we can list all components that generate packets for the network. We can consider the size and priority of these packets and try to reduce the number by clubbing similar kinds of data, batching some data, batching acknowledgements, and batching session components.

In the given architecture for the Rover Workstation, we see a heavy dependency on KANA components. Can this be an anti-pattern? Yes, this is an example of a vendor lock-in anti-pattern. The solution would be to reduce this dependency. The suggested solution is to provide an isolation layer between the product-dependent interfaces. In our case, replacing the KANA component for session management by in-memory session management seems a better choice. In fact, this further encourages us to re-factor the session management process so that the process is streamlined and offers lesser load on the network and the processing system.

Refining the System

Performance improvements made at the level of architecture design have a great impact on responsiveness and scalability. With a poor architecture or design, it is unlikely that any amount of clever coding will allow performance objectives to be achieved.

Architectural re-factoring is just one of the solutions.

Having a good architecture and design is essential, but it does not guarantee performance. It is not possible to ignore implementation choices and coding details.

Re-factoring the code also brings in significant advancements. A simple example will be optimizing the HTML generated for data display. Though it may seem like a very insignificant improvement, for the end-user this will be the most relieving performance improvement. Data can be flushed in sections so that the user can see partial pages more quickly, hence reducing response time. Similarly, customized tags act as performance overheads, thus minimizing their use. Extra characters (comments, spaces, returns and redundant attributes) can be removed safely without any change in appearance. Similarly, tables and nested tables in particular can be altogether avoided.

Technology-specific fine-tuning is also important. All major technology vendors provide tips on optimized use of their technology. For example, Oracle database performance optimization and fine tuning, Java Server Pages (JSP) fine tuning or server-side fine tuning can be some of the techniques the developers may find handy.

It is important to consider the trade-offs and the costs involved in performance optimization. The application should be tested and analysed to check if the modifications have brought in significant improvements. If desired levels are not achieved, are achieved partially or costs involved are very high, then the performance objectives should be reset.

CASE ANALYSIS

Comprehending the architecture and the overall system objectives should be the first step towards optimizing a system. We shall take an approach where we evaluate the existing architecture and then suggest strategies and architectural changes for improvement. In order to refine an existing architecture, we asses the quality of architecture using an evaluation criterion based on architectural attributes. To realize the architecture, each attribute is assessed based on the design considerations made. If an attribute has been implemented partially or not at all, then refinements to the architecture are carried out using the suggested changes. The quality of the refined architecture is assessed and the process repeated. When the desired quality is achieved, the refinement process is stopped. In this process, we first prepare for the evaluation of the current architecture and then execute the evaluation. Finally, we suggest refinements.

Architecture described in a standard notation eases the task and reduces the chances for misinterpretation. It is very important to collect various views of the architecture. In reality, not all architectural views are available for the analysis. If there are any missing views, they should be created beforehand. For example, in this case study analysis, we shall first come up with 4 + 1 architectural views and related diagrams for the architecture based on existing information.

Once we have the documentation of the architecture, we observe various views of this architectural representation to identify its strengths and weaknesses. In other words, we prepare for evaluating the existing architecture and identifying all performance objectives and bottlenecks. We take into consideration all the non-functional requirements such as performance, reliability, availability, security, serviceability and scalability during this step. We perform the independency analysis between these elements.

We finally execute the evaluation and suggest strategies and architectural changes for refining all the non-functional architectural drivers. Once we identify what the refinements are, we reflect these changes in the description of the architecture that is already available. This process is repeated until we have a system that is able to correct any non-functional requirement we want to improve.

In further analysis, for the purpose of focusing on one aspect of non-functional requirements, we take the performance aspects into consideration. The motivation also comes from stringent performance requirements that were not being met. The following analysis is motivated by the SPE process proposed by Connie Smith and Lloyd Williams in their book on Performance Solutions (Smith and Williams, 2002). In our discussion earlier on performance-oriented design, we have provided a four-step process (see Figure 3.3). We adopt a very similar approach in analysing this case study as well.

To state the above discussion in a more systematic way, we follow the following steps for achieving the required performance through architecture evaluation:

1. Describe the architecture in 4 + 1 views (or any other standard description)
2. Prepare for the architecture evaluation
    i. Define the template for requirements: summarize the functional and non-functional requirements in isolation and then try to find the relationships between the functional and non-functional requirements
    ii. Set the scope of the evaluation or determine the goals of refinement
    iii. Determine the architectural attributes and determine the weights for each of these attributes

3. Execute the evaluation and suggest the refinements
   i. Evaluate how the current architecture meets the architectural attributes using the design considerations implemented by it
   ii. Suggest strategies and architectural changes for improvement
4. Reflect these changes in the architecture description (like 4 + 1 views) and repeat these steps until the all the architectural goals are met

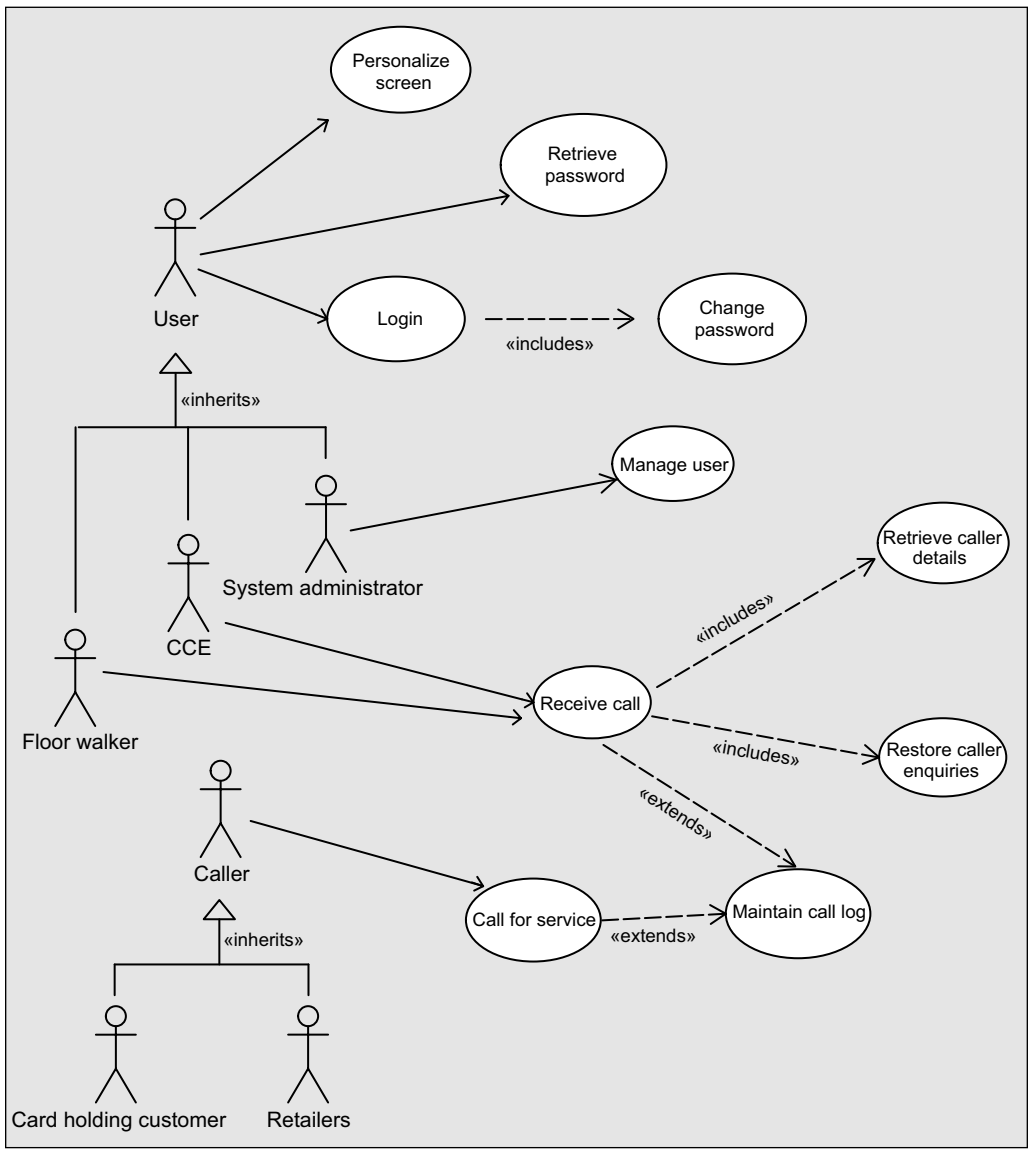


Figure 3.4 Use case diagram

Step 1: 4 + 1 View Model of the Proposed Architecture

Using the 4 + 1 view model, we can represent the given architecture in such a way that it can be understood better and hence can be evaluated and then refined. Here, we will be presenting various views of the given architecture as available. Note that these descriptions are very subjective and each architect can come up with their own versions.

Use case view

The 'use case' view is an important input in the selection of the set of scenarios and/or use cases constituting the iteration. For evaluating the architecture, we will concentrate on only those use cases (Figure 3.4) that have any or all of the following:

- A substantial architectural coverage and many architectural elements
- Stress or illustrate a specific, delicate point of the architecture

Note that the use cases 'receive call' and 'manage users' have a pre-condition that the use case 'login' be completed successfully. Table 3.1 explains the use case diagram given in Figure 3.4 in a very systematic way.

Use case	Actors	Flow of Events
Call for service	Caller	1. Caller (retailer, card-holding customer, guest) calls a particular number and gets connected to the system
		2. (a) The system (IVRS) passes the caller greetings and asks for the information he/she wants through a list of selections
		(b) The system initiates the use case 'maintain call'
		3. Caller selects the option
		4. System retrieves the information and passes it to the caller
		5. (a) Events 3 and 4 continue till the caller selects the option to exit or hangs on
		(b) System completes the use case 'maintain call log'
		6. The caller selects the option to talk to a service agent
		7. The caller is routed to an available workstation CCE
Receive call	1. CCE	1. (a) The CCE or floor walker receives the call by either lifting the phone or through a selection on the CCE monitor
	2. Floor walker	(b) The system initiates the use case 'maintain call log'
		2. System displays the caller's phone number
		3. System retrieves the details of the caller if possible and displays it

(Continued)

Use case	Actors	Flow of Events
		4. CCE or floor walker asks the caller about his/her identity type and identification number and enters the same into the system *Note*: Type of caller can be retailer, card holder or guest (for enquiries only) 5. CCE or floor walker verifies the details in the case of retailer or card holder 6. CCE or floor walker asks the caller about the query or service request and enters the same into the system 7. In the case of information query only, the system displays the information and the CCE or floor walker informs the same to the caller 8. In the case of service request, the CCE or floor walker initiates the use case 'retrieve caller details' 9. CCE or floor walker asks the caller about the service request and initiates the use case 'resolve caller enquiries' 10. The system completes the use case 'maintain call log'
Retrieve caller details	1. CCE 2. Floor walker	1. CCE asks caller and enters caller type and identification number 2. CCE requests the system to retrieve caller details 3. System verifies the authorization of the CCE or floor walker 4. As per the access permission, system displays the information 5. CCE or floor walker asks caller the verification question and verifies the genuineness of the caller
Resolve caller enquires	1. CCE 2. Floor walkers	1. If the call is for a customer problem, the CCE will answer the enquiry by referring to customer details, business policies and business rules. The screen pop response time should be less than or equal to one second. These can be policies related to government regulations like local tax, which change depending on the location of a store. These could be also policies on consumer rights 2. If the call is to modify customer profile or account profile, the CCE can verify the identity of the caller by asking for certain inputs. Once the confirmation is done, the CCE is allowed to modify the details and save them. All business rules and policies are built-in; online help is available to the CCE. The system confirms the status of the operation. Users will be notified of exceptions, if any *Note*: The floor walker should be able to do whatever operations the CCE can perform

(Continued)

Use case	Actors	Flow of Events
Maintain call log	System daemon	A log of information pertaining to the call made by the customer is created. The call history is stored for future reference. Usual information includes the following: ■ Agent activity (time stamps for login/logoff/break) ■ Account interaction data (date, time, inquiry channel, agent id, etc.) ■ Details of the interaction (tasks performed, changed values, time taken for each task, etc.)
Login	User	1. The user enters the user name and password 2. The system retrieves the user information 3. The system verifies the user login 4. The system displays the home page
Personalize screens	User	1. The user selects the personalize screen option 2. The system displays a form to enter the personalized data 3. The user answers all the queries and changes the screen as per his/her personal preferences 4. System stores the information in user info
Change password	User	1. The user selects an option to change his or her password 2. System prompts user to enter the old password and new password 3. The system changes the password in the user database 4. If the user is an administrator, then user can reset password of other users (Precondition: the 'login' use case is to be successfully completed)
Retrieve password	User	1. User selects the option 'forgot password' 2. System displays a clue questions and prompts the user to answer 3. The user writes the answer of the clue questions 4. The system verifies the answer 5. The system sends the user name and password to the registered e-mail ID of the user
Manage users	System administrator	1. System administrator selects the option 'manage users'

(Continued)

Use case	Actors	Flow of Events
		2. System displays a screen to do the following functions: ■ Manage users ■ Manage roles ■ Manage role assignment ■ Manage service association 3. The system administrator selects the options 4. System displays the form with data 5. System administrator changes the information 6. System stores the changed information (Precondition: the 'login' use case is to be successfully completed)

Table 3.1 Use case details

Logical view

The 'logical view' addresses the functional requirements of the system. This view usually represents the abstraction of the design model and identifies major design packages, subsystems and classes. A part of the 'logical view' for the case is shown in Figure 3.5. The system is designed following the n-tier architecture and this part of the logical view represents only the Web-tier, the application tier and the data tier.

Process view

The representation of XML parsing is shown in Figure 3.6 as an example of the process view artefacts. This figure describes an XML message received by the receiver and acknowledgement sent if the message could be parsed successfully.

Another important aspect of the process view is to show how data are replicated in the backup data server. The process of data replication is shown in Figure 3.7, which shows all the steps that need to be completed for the successful replication of data from the primary server to the back-up server.

Implementation view

The 'implementation view' presents the organization of static software artefacts such as components, executables, data files. This view also describes the packages and layers to represent the development environment. It addresses the issues of ease of development, management of software assets and reuse of components.

A part of the implementation view of the case is shown in Figure 3.8. This view captures some of the important components that are used in the case. The figure shows clearly that a package of server-side components is the heart of this call centre application; other important components are security, business rule engine, reporting, session management and transaction management.

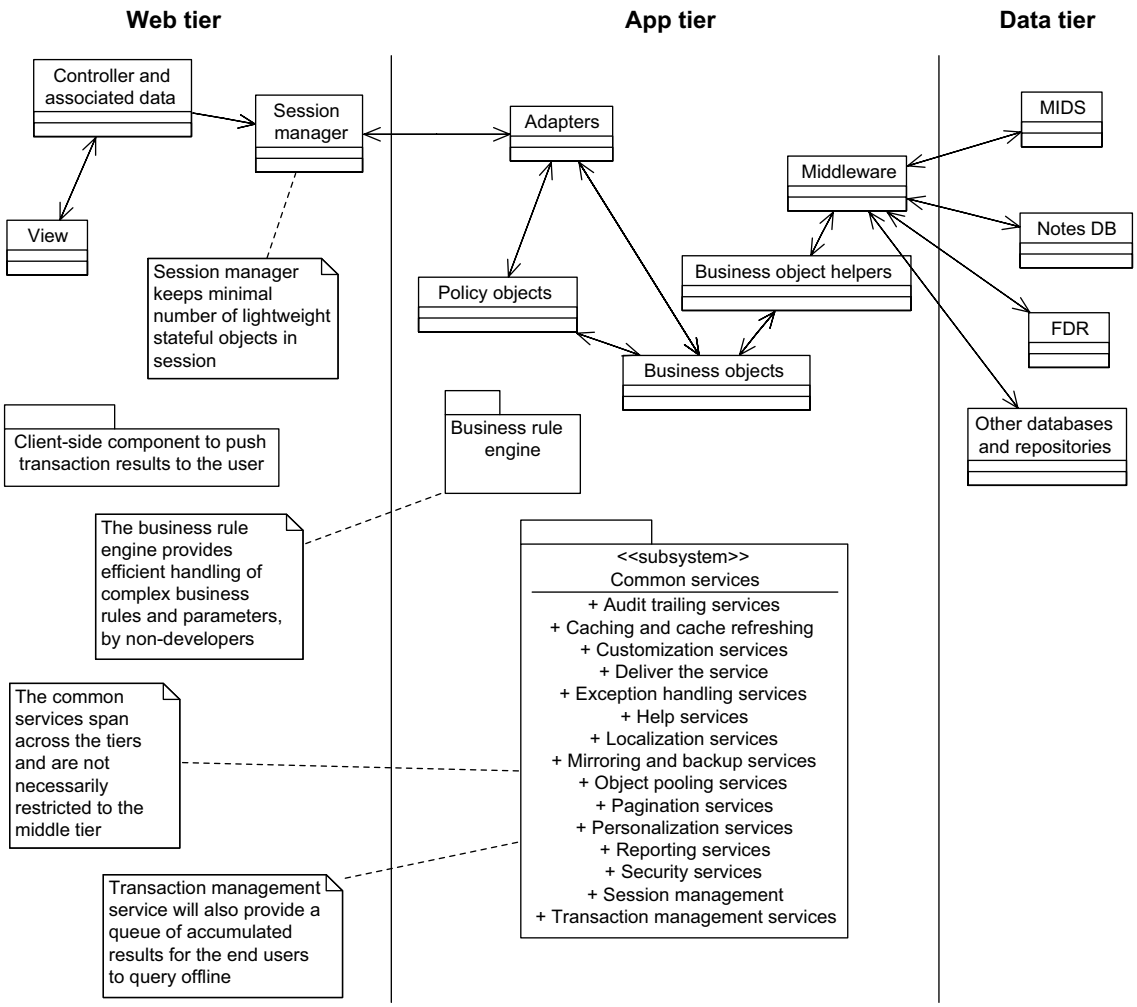


Figure 3.5 Logical view

Deployment view

The 'deployment view' of the case is shown in Figure 3.9. This view describes different servers used to place the components as mentioned in the implementation view. This diagram also shows that the Web server, the application server and the database server are replicated to keep 24 × 7 availability of the system.

It is important to revaluate the proposed architecture and assess the change in the quality of the architecture. If the architecture quality degrades then the design decisions should be revisited. For example, increasing the level of security may have trade-offs in the other attributes of performance.

Further iterations on the refined architecture can be carried out using the approach mentioned in the whitepaper 'Architectural blueprints—The "4+1" view model of software architecture' (Kruchten, 1995).

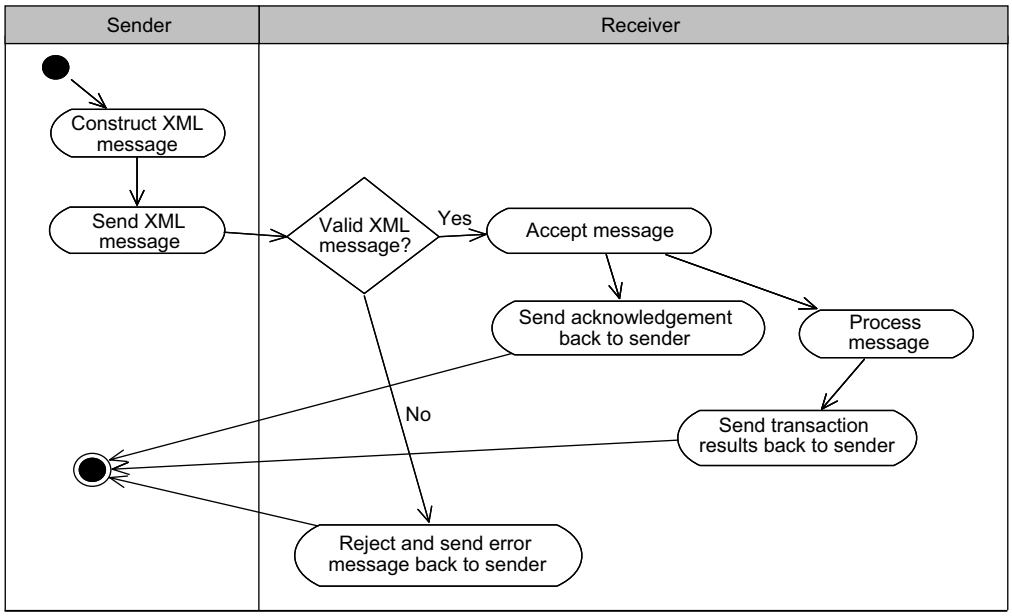


Figure 3.6 XML parsing

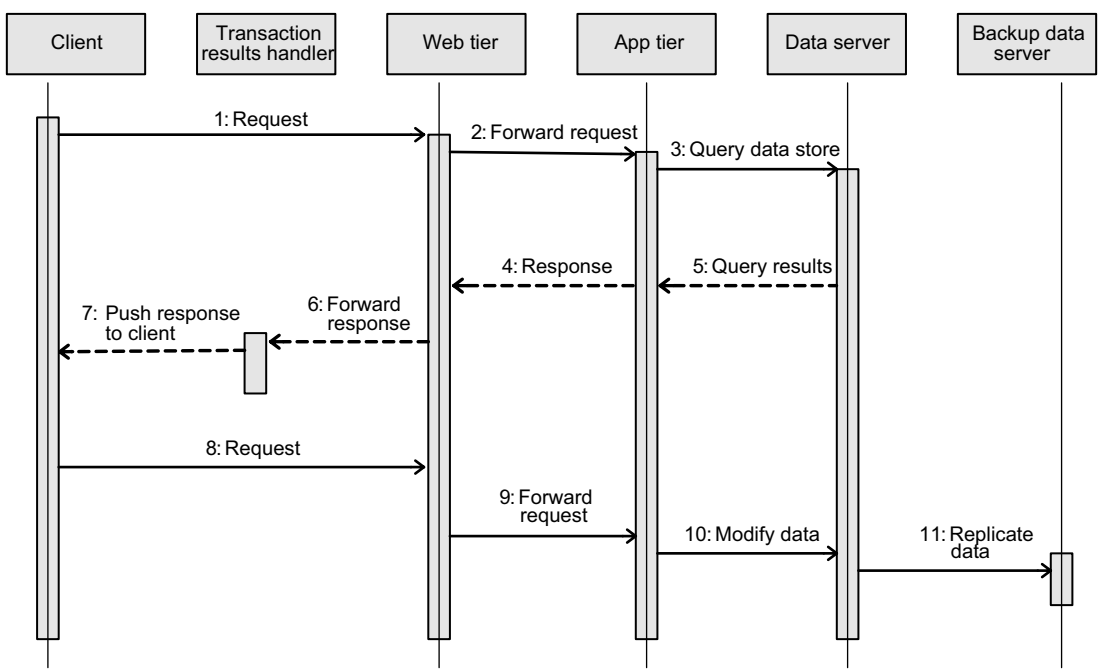


Figure 3.7 Communication of transaction results and data replication

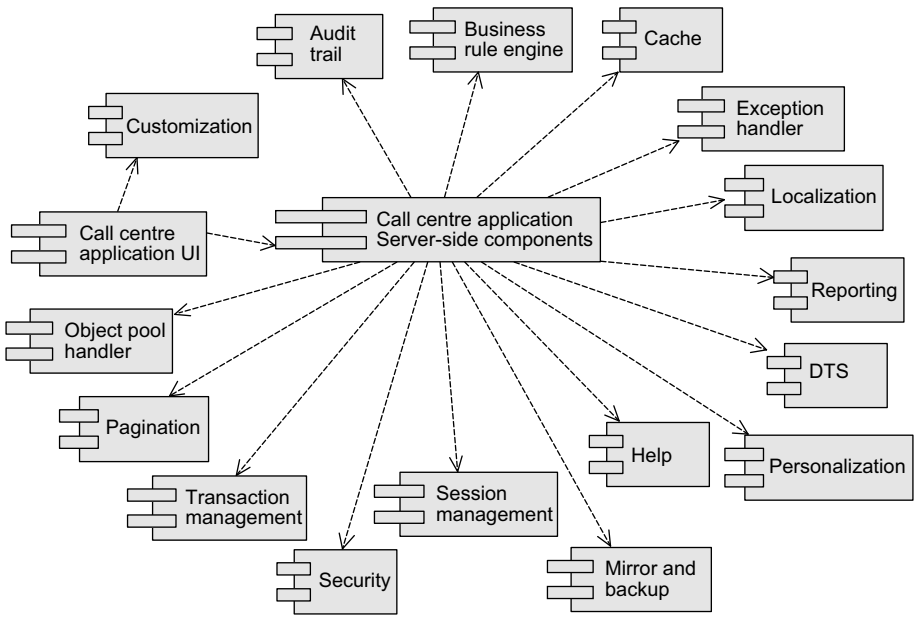


Figure 3.8 Implementation view

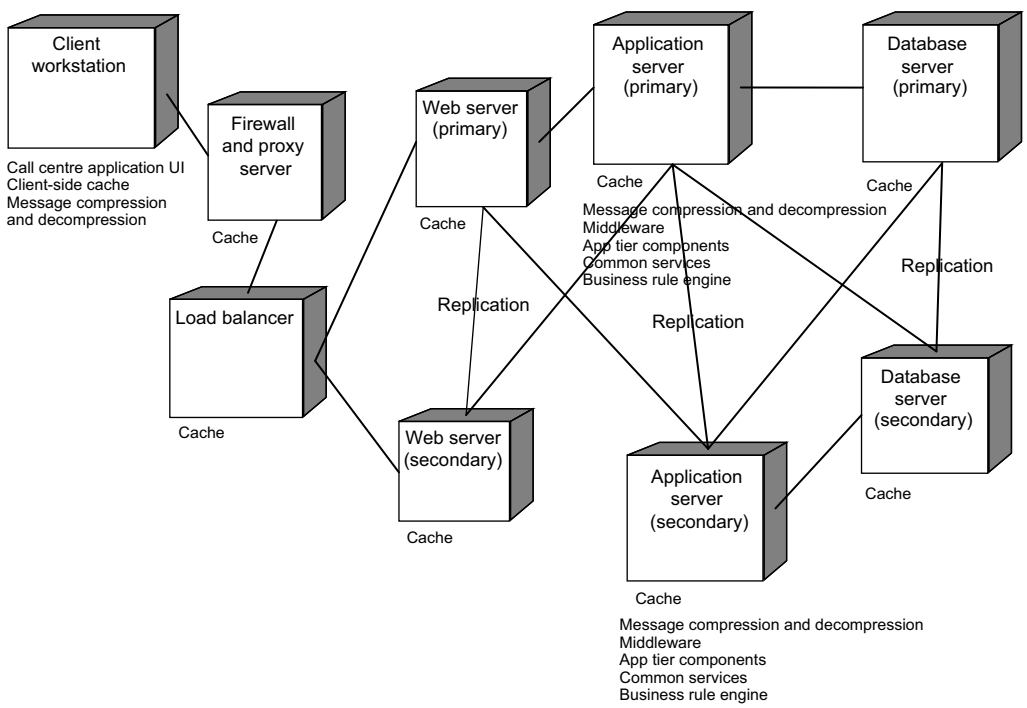


Figure 3.9 Deployment view

Step 2: Prepare for the Evaluation

Once we have the required documentation of the architecture, we are ready to evaluate the architecture. There are three activities we perform in this step.

- The first is to identify the major functional and non-functional requirements independently and then identify relationships between them.
- The second activity is to generate an evaluation contract, that is, we need to determine the scope of the evaluation.
- The third activity is to define the priorities for each of the architectural drivers.

The first step involves listing major functional and non-functional requirements and identifying the dependencies among them. Figure 3.10 shows the template for these dependencies among the requirements. Here, we have identified nine functional requirements (FR) and six non-functional requirements (NFR). Each non-functional requirement is stated in a quantitative manner grouped by the major non-functional heads: security, performance, reliability, availability, scalability and serviceability. If we take performance into consideration, we can notice that two functional requirements, FR2 and FR6, are directly dependent on this. FR2 is providing accurate, consistent and up-to-date information to customers, retailers and other authorized knowledge seekers; FR6 is about reducing the support requirements so that the system should be usable with automated business rules and incorporated with an online help.

The second step in which the scope of evaluation is determined is going to be easy in our case. Our scope of evaluation is limited to the set of non-functional requirements from Figure 3.10. We will consider each of them in a later step.

The third step is giving appropriate weights for each of the architectural elements. For example, we have considered security and performance as high-priority elements (assigning 20 per cent of the weight for each) and reliability, scalability and serviceability as next important elements (giving 15 per cent weight for each of these). Note that the sum of all the weights is 100 per cent.

Assignment of these weights may appear to be very subjective and they largely depend on the judgement of experts. There can be more logical and structured approaches in assigning these weights, such as quality function deployment (QFD) or Kano. A QFD tool known as House of Quality will help in identifying and then transforming the *voice of the customer* into engineering characteristics that are then prioritized and help in setting appropriate development targets. The Kano model classified the voice of the customer into five categories (attractive, one-dimensional, must-be, indifferent and reverse), which is used in structuring the comprehensive QFD models (Cohen, 1995).

In addition to assigning the weights, we also describe these elements to provide the details of what these architectural elements mean as well a s to provide justification for the assignment of weights. Figure 3.11 shows such a table giving weights for the architectural elements we have considered.

Step 3: Execute the Evaluation

We will evaluate how the current architecture meets the architectural attributes using the design considerations implemented. To be more objective in our review efforts, we will use the following rating model:

Design consideration	Rating
Absent	–1
Present and conforms to a high degree (exceeds)	+1
Present but conforms to a low degree (meets)	0

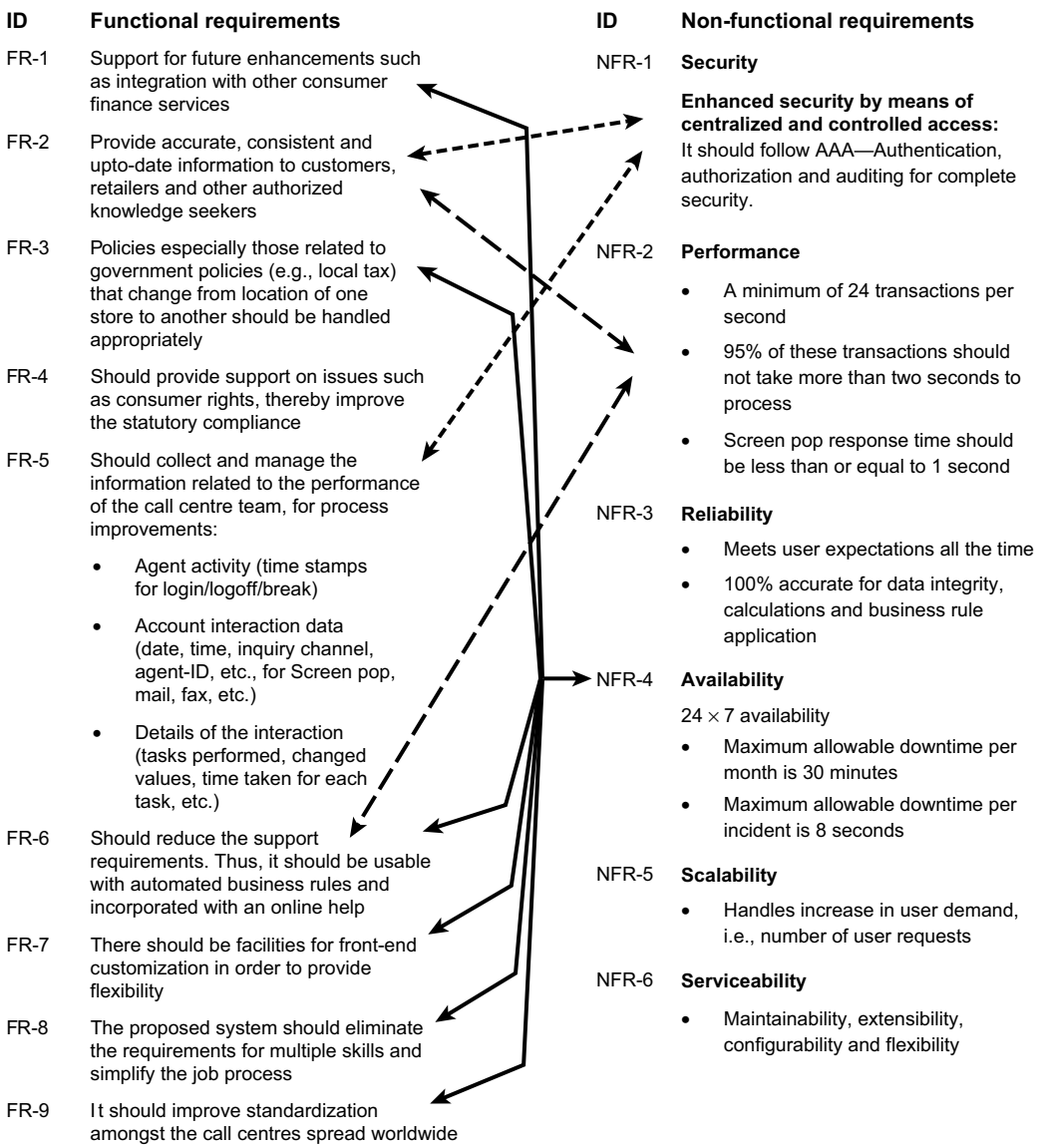


Figure 3.10 Template for requirements

An advanced model can use the weighted rating to indicate the relative importance of each consideration.

We need to consider each architectural attribute (performance, security, scalability, reliability, availability and serviceability) and rate the architectural features considering the above model. In this section, we shall consider one attribute, which is performance. The reader is expected to do the same exercise for rest of the attributes.

As we proceed we suggest strategies and architectural changes for improvement.

Architectural attributes	**Attribute weight** (sum of all should be 100%)	**Design considerations**
Security	20%	• Authentication and access to the system for the registered users • Encrypted communication between client and server • Audit trailing • Protection against virus
Performance	20%	• Caching • Object pooling • Minimize network traffic • Minimize XML parsing • Handling of high volume of data
Reliability	15%	• Reliable session management • Acknowledgement of every message • Exception handling • Accurate transaction management—commit and rollback, concurrency control including deadlock and live lock handling • Usage of latest information • Online and immediate updating of information • Communication to the end user of the results of the transaction • Immediate replication of changed information on servers and data stores
Availability	15%	• Redundancy using multiple app, web and data servers • Replication and synchronization • Business continuity in case of city, country failure • Database recovery in case of database failure
Scalability	15%	• Ability to add multiple end customers/retailers in the ASP model • Vertical scaling for addition of users or transaction volume
Serviceability	15%	• Use of MVC and layered architecture o Web tier o App tier o Business tier—Business object, business helper objects, business rules and policy objects • Use of patterns like controllers, adapters, facades and value objects • Modularization • Use of middleware and gateways • Integration with services: use of XML for interfacing • Internationalization and localization • Parameterization • Front-end customization and personalization • Externalization of business rules • Built-in and automated business rules • Online help • Standardized, configured and controlled from a single location • Reusable common services and their extendibility • Reporting

Figure 3.11 Weights table for architectural elements

Evaluating performance

We shall consider various design considerations and rate it using the above model. We will give the rationale for giving a particular rating and suggest the possible remedy if we rate the design consideration as absent or weak. We shall also give the views that are affected because of the specific design consideration. The design considerations are the following:

Design consideration 1: Caching Caching and cache refreshing services are only for business rule objects and some other static data at the startup and other points in time.

Rating: 0

Remedy: Caching strategies at the various servers, such as proxy server, Web server, application server, data server and middleware, can also be considered. Application-level caching can be extended to user transaction, common services and some backend data as well. For example, in the case of the FDR database, the design can look at possibilities of caching data appropriately. Client-side caching of static data and user transaction data can also be considered.

Views effected: Logical and deployment.

Design consideration 2: Minimize network traffic We have noticed that the system sends an acknowledgement for every message and there is broadcasting of session object to all the Web servers.

Rating: –1

Remedy:

- To optimize on message acknowledgements, a suggested solution is to consolidate acknowledgements, compress them for reduction in size of network packets and transmit them at a pre-determined frequency rather than for each message.
- This solution will have a trade-off in terms of low network traffic but increased time for the client and the server owing to the consolidations and the compression/decompression algorithm. The challenge is to maintain the performance with increasing numbers of requests or users without sacrificing reliability.
- A suggested solution for handling object broadcasting is to have some designated servers rather than all for holding the objects that need to be replicated; any application that needs these objects can pick them up from such a server.
- Deploy the services on the servers in such a way that interdependent components are deployed in close proximity, for example, the data tier should be physically deployed, preferably, on the data servers.

Views effected: Process and deployment.

Design consideration 3: Object pooling Object pooling has not been mentioned in the given architecture.

Rating: –1

Remedy: Consider object pooling.

Views effected: Logical.

Design consideration 4: Minimizing XML parsing Currently, XML parsing is at both the ends, that is, client and server, which is very inefficient and hampers performance.

Rating: –1

Remedy:

- At the interface layer, XML parsing can be done only once at the server, where the data anyway need to be validated.
- At non-interfacing layers, XML may be avoided by using other alternatives such as messaging.

Views effected: Logical.

Design consideration 5: Handling of high volume of data There is no mention of techniques for handling high volume of data, for example, as a result of querying.

Rating: –1

Remedy: Asynchronous mechanism for posting high volume of data; pagination services for retrieving high volume of data can be considered.

Views effected: Logical.

Earlier in this chapter, we have provided information about performance modelling, evaluation and identification of the remedies. Refer to the sections on performance goals, finding bottlenecks and reviewing the performance once more, if needed.

Step 4: Reflect the Evaluation

The architecture quality can be measured using the weighted average of the rating for the design considerations.

The quality of current architecture can be measured quantitatively as follows:

Weight for security × Weighted average rating for implementing security design considerations
+ Weight for performance × Weighted average rating for implementing performance design considerations
+ Weight for reliability × Weighted average rating for implementing reliability design considerations
+ Weight for availability × Weighted average rating for implementing availability design considerations
+ Weight for scalability × Weighted average rating for implementing scalability design considerations
+ Weight for serviceability × Weighted average rating for implementing serviceability design considerations

Architecture quality will fall in the range of –1 (worst fit) to +1 (best fit).

We need to incorporate the suggestions that come up during step 3 and refine the architectural descriptions (all 4 + 1 views in this case) and repeat all the steps again until we get a good score for the quality of the architecture.

Conclusions

In this chapter, we have looked at a very common challenge faced by the software services industry: a team is provided an architecture that is designed by someone else and faces the challenge of implementing the architecture and meeting all the functional and non-functional requirements. We have discussed how some of the best practices like identifying anti-patterns help in re-factoring the architecture so that the system is built according to the SLA.

It is very important to follow a proven process in addressing the non-compliant requirements. We have taken some of the non-functional requirements into consideration in re-factoring the given architecture. We have looked at performance requirements in detail.

SPE is a major challenge as well as the need of the hour. Performance-oriented design prevents many of the problems that might surface after the system is built. Analysing the architecture to find out bottlenecks and flaws will result in huge savings.

It is also important to avoid reinventing the wheel and make use of best practices and concepts like anti-patterns. Often, these approaches are critical under tight schedules and deadline pressures. But augmentation of this knowledge base is as essential as using it. In a large software system it is usually impossible to know every detail and it is near to impossible to be an expert in everything. Architects have a high level of knowledge of the system design, while developers have a high level of knowledge of the specific source code module. Hence, both need to work hand in hand to achieve the desired levels of performance. Though we focused more on performance in the case analysis, the same approach works well for all other architectural attributes.

We have not focused much on the usability aspect in this case study, which is an important feature in evaluating architecture. Once we make sure that all the required functionalities are achieved and the candidate architecture is refined using performance engineering, it should also be evaluated using the usability point of view. Various stakeholders including CCEs, customers and application programmers and system administrators should be considered in evaluating how the architecture is meeting all important usability aspects.

Best Practices and Key Lessons from the Case Study

- The performance objectives should be well defined and quantified. a well-defined performance objective would be something as follows: 'The end-to-end time for completion of a 'typical' correct ATM withdrawal performance scenario must be less than one minute, and a screen result must be presented to the user within one second of the user's input'. Vague statements such as 'The system must be efficient' or 'The system shall be fast' are *not* useful as performance objectives.
- The performance objectives should be measurable as well as testable. If not, then checking whether the system is compliant is not possible.
- Perform an early estimate of performance risk.
- Track the SPE costs and benefits.
- Integrate SPE into your software development process and project schedule. To be effective, SPE should not be an 'add-on', it should be an integral part of the way in which software is being developed.
- Establish precise, quantitative performance objectives and hold developers and managers accountable for meeting them.
- Identify critical use cases and focus on the scenarios that are important to performance. Use cases describe categories of behaviour of a system or one of its sub-systems. They capture the user's view of what the system is supposed to do. Critical use cases are those that are important to responsiveness as seen by users, or those for which there is a performance risk. That is, critical use cases are those for which the system will fail, or be less than successful, if performance goals are not met.
- Not every use case will be critical to performance. The 80–20 rule applies here. A small subset of the use cases (≤20 per cent) accounts for most of the uses (≥80 per cent) of the system. The performance of the system is dominated by these heavily used functions. Thus, these should be the first concern when assessing performance. Do not overlook important functions that are used infrequently but must perform adequately when they are needed. An example of an infrequently used function whose performance is important is recovery after some failure or outage. While this may not occur often, it may be critical that it be done quickly.

- Perform an architectural assessment to ensure that the software architecture supports performance objectives. Recent interest in software architectures has underscored the importance of architecture in determining software quality. While decisions made at every phase of the development process are important, architectural decisions have the greatest impact on quality attributes such as modifiability, reusability, reliability and performance.
- Use performance models to evaluate architecture and design alternatives before committing to code. While it is possible to re-factor code after it has been written to improve performance, re-factoring is not free. It takes time and consumes resources. The more complex the re-factoring, the more time and resources it requires.
- Be knowledgeable about the commonly occurring trade-offs in performance, for example, security and performance. These will make the process quick and foolproof.
- Start with the simplest model that identifies problems with the system architecture, design or implementation plans and then add details as your knowledge of the software increases. The early SPE models are easily constructed and solved to provide feedback on whether the proposed software is likely to meet performance goals. These simple models are sufficient to identify problems in the architecture or early design phases of the project. Refine the models as the information about the system gets generated.
- Use best and worst case estimates of resource requirements to establish bounds on expected performance and manage uncertainty in estimates. best and worst case analysis identifies when performance is sensitive to the resource requirements of a few components, identifies those components and permits assessment of the severity of problems as well as the likelihood that they will occur. When performance goals can never be met, best and worst case results also focus attention on potential design problems and solutions rather than on model assumptions. If you make all the best case assumptions and the predicted performance is still not acceptable, it is hard to fault the assumptions.
- Establish a configuration management plan for creating baseline performance models and keep them synchronized with the changes to the software.
- Instrument software to facilitate SPE data collection. You must instrument software by inserting code (probes) at key points to measure pertinent execution characteristics. For example, insert code that records the time at the start and end of a business task to measure the end-to-end time for that task.
- Measure critical components early and often to validate models and verify their predictions.

Notes

1. KANA, GEOTEL, UNISON Davox, MERCATOR, TANDEM and ODS Gateway, NOTES DB, SiteMinder and ClickStream are the registered trade marks of the respective companies.
2. See http://java.sun.com/blueprints/patterns/MVC-detailed.html
3. See http://developer.apple.com/documentation/Cocoa/Conceptual/AppArchitecture/Concepts/MVC.html
4. See www.antipatterns.com/

Further Reading

Evaluating architectures is an important topic and we will look into this in greater detail in the next chapter. However, an article by Heeseok Choi and Keunhyuk Yeom 'An Approach to Software Architecture Evaluation with the 4 + 1 View Model of Architecture' (Choi and Yeom, 2002) is a good starting point to explore this area, especially when you have all the 4 + 1 views documented. We have already recommended Kruchten's paper 'Architectural Blueprints—The "4+1" View Model of Software Architecture' (Kruchten, 1995) to understand how the 4 + 1 views got popularized.

We have discussed MVC architecture extensively in this chapter. We recommend the following Web sites to know more about MVC architecture:

- http://java.sun.com/blueprints/patterns/MVC-detailed.html
- http://developer.apple.com/documentation/Cocoa/Conceptual/AppArchitecture/Concepts/MVC.html

There are a few good resources to learn about software performance. A book by Connie Smith and Lloyd Williams is a masterpiece. It is available from Addison Wesley and is titled *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software* (Smith and Williams, 2002).

The Web site www.perfeng.com/ is also a good source of information on performance engineering.

We have also discussed anti-patterns in this chapter. The following resources provide very good introduction to this topic. There are a few good books in this topic. *Anti Patterns—Refactoring Software, Architectures, and Projects in Crisis* (William *et al.*, 1998) is perhaps the most popular book and a good starting point.

The other books from same group of authors are *Anti-Patterns and Patterns in Software Configuration Management* (Brown *et al.*, 1999), *Anti-Patterns in Project Management* (Brown *et al.*, 2000) and *J2EE Anti-Patterns* (Dudney *et al.*, 2003).

Another very useful material on Anti-Patterns is *Anti-patterns, a Seminar on Design Patterns* by Vesa Kärpijoki, Helsinki University of Technology (Kärpijoki, 2001).

The anti-patterns Web site www.antipatterns.com also has a tutorial and other useful material. You can start with a link on 'What is Anti-patterns' and move on to the Anti-pattern catalogue. This Web site also hosts an excellent term paper on Anti-patterns by Edward Jimenez: www.antipatterns.com/EdJs_Paper/Antipatterns.html.

Chapter 4

Architecture Evaluation—Developing a Futuristic Travel Search Engine

Contributors
Kirti Garg
Dilip Bhonde
Shubhangi Bhagwat

It is very important for the architect and all the other stake holders in the project to know how good the proposed architecture is, before actually building the system. Will the proposed architecture help and guide in building the right system? This is not an easy question to answer. In fact, some people take an extreme stand that architecture is a gamble. It is granted that the architect arrives at his choice in this gamble using all his mastery and skill, but still, will it not be nice to know if he is 'betting' on a winning architecture that meets all system requirements?

Until recently, there was no way to validate software architectures. Now, there are well-documented methods such as architecture trade-off analysis method (ATAM), cost–benefit analysis method (CBAM) and software architecture analysis method (SAAM) that help us review, evaluate and refine software architectures. As in the case of all software engineering disciplines, early detection of problems help build quality products with lower costs; early architectural review is providing the same benefits and hence is becoming very popular among software architects, organizations, clients and researchers. Software architecture evaluation is evolving as one of the recommended steps in developing large and complex software systems. The benefits being observed are immense as compared to the efforts and costs involved. The basic principle of software economics works very well here, that is, the cost of fixing errors later in the life cycle is much higher than fixing it early.

This case study deals with a very ambitious solution to the problem of search in the travel domain. All travel companies in the world need this solution. The development team in the Travel Vertical Business Unit of Fact-Tree discovered this great opportunity. The team, however, is very cautious and does not want anything to go wrong. They followed some best practices such as extensive domain study, employing the best minds on the project and validating each step to ensure correctness of the proposed system. They came up with an architecture that *seemed* to be perfect. The team is proud of it. But there are questions such as whether it is enough to have an architecture that seems good? Or should there be a formal verification or validation? They have decided to evaluate the architecture as it will be of immense help to know if the team is moving in the right direction. But the questions 'How to proceed?' 'Which method to use?' and 'When to do it?' need to be answered. The five 'W' (what, why, when, where and who) and one 'H' (how) question also needs to be answered. This case study provides an opportunity to understand the need and process of architecture evaluation and gives us a picture in the real-world context of the same.

BACKGROUND

What Is Architectural Evaluation?

Architecture evaluation can be defined as a task of quality assurance that makes sure that all the artefacts and work products relating to the architecture are evaluated. These work products include the architecture and the architectural descriptions. If the architecture of a system is an organizational investment, then a structured review may be considered as an insurance policy. Developing architecture without feedback from a non-partisan resource is bad practice. Paul Clements, Rick Kazman and Mark Klein in their book *Evaluating Software Architectures: Methods and Case Studies* (Clements *et al.*, 2002) discuss various aspects of architectural evaluation. Here, we present some of them in a structured form of answering the following questions:

- What is architectural evaluation?
- Why should architecture be evaluated and reviewed?
- When to evaluate and review?
- Who should evaluate and review?
- What should be reviewed?
- How to review architectures?

Why Should Architecture Be Evaluated and Reviewed?

Reviewing architecture offers many direct and indirect advantages. These advantages can be enjoyed by the entire range of stakeholders of the project (developer organization, client users, end users, developers, project manager, architect(s), installers, system administrators, maintainers, etc.). The reasons for evaluating and reviewing architecture include the following:

- A *huge saving in cost and efforts* has been observed through the early detection of errors and problems. Errors at the architectural level are usually logical errors that may prove to be very costly if left undetected. An early review and determination of such errors is a must to keep the cost and efforts within initial estimates. Architectural review tends to find the defects that can be fixed as well as analyse the defect trends so that the process of architecting can be improved.
- Architectural evaluation leads to a group conversation about the functional and non-functional (quality-related) requirements of the proposed system. Stakeholders get an opportunity to realize that some of the requirements severely impact factors such as design, costs and risks. This can lead to a *clarification and prioritization of requirements*, hence easing the developmental job. Gathering all stakeholders may seem a potential source of changed requirements and plans, but it is actually a boon in disguise. As it is early in the development cycle with no major decisions taken, this is the perfect time to change plans and requirements if needed. If the evaluation or design of the architecture changes, the cost and money saved at this stage will be much less than when implementing these changes later.
- Reviews ensure reduced risks of *disaster projects or acquisitions* through a thorough evaluation of projects, especially before acquisitions.
- One of the most crucial aspects of development is to have a *common understanding of the system* among all stakeholders. A complete representation of the architecture is a prerequisite for architectural evaluation and review. Usually, standards such as 4+1 views of UML or IEEE 1471 standard representations can be used. This is like converting project data to a form that a stakeholder at any level can understand. Being in a form comprehensive to all and explained to all by the architect(s) leads to a common understanding of the system along with increased and proper documentation as a by-product.
- A *comparison of architectural options* is possible when the review for system qualities (usually non-functional requirements and key functional requirements) is done early in the life cycle.
- Reviews are excellent tools to *promote organization-wide knowledge* about a specific process or discuss best practices and are hence good instruments for organizational learning.

When to Evaluate and Review?

Architectural review is not necessarily a one-time job. It can be performed at different points in time, as per the need of review, available resources and, of course, fulfilment of the review prerequisites:

- *Regularly*, as part of an iterative architecting process
- *Early*, to validate the overall course of the project, just after finalizing the requirements specification documents and coming up with ideas for architecture or creation of a logical architecture
- *Toll-gate*, as a check before major implementation starts, or in case of an acquisition, when the supplier is selected
- *(Too) late*, to determine how to resolve architectural problems

The review can be a formal one, following the process thoroughly, or may be a quick informal review, done by the architecting team itself.

Reviews that are thoroughly planned are ideally considered an asset to the project, at worst a good chance for mid-course correction. It means that the reviews are scheduled well in advance, are built into the project's work plans and budget and are expected to be followed-up. The review can be perceived not as a challenge to the technical authority of the project's members but as a validation of the project's initial direction. Planned evaluations are proactive.

Unplanned reviews are reactive steps, as the evaluation is unexpected and is usually an extreme measure to salvage the previous efforts. Usually, they come into picture when the management perceives that a project has a substantial possibility of failure necessitating a mid-course correction. This unplanned evaluation is more of an ordeal for project members, consuming scarce project resources and tightening schedules. It tends to be more adversarial than constructive.

Who Should Evaluate and Review?

Can anybody and everybody review architecture? The answer is *no*. Depending on the purpose of the review, different parties are involved. These can be the *development team* itself or *another group in the organization* or can be some expert e*xternal reviewer*, preferably partisan, with no interest in the outcome of the evaluation, and who is an expert in evaluation.

To evaluate architecture without any bias, it is very important to bring in an external reviewer as it will help in getting an outsider perspective of a project. However, this can sometimes cause problems because the project's architect and other people who are in key positions in the project may feel uncomfortable or even threatened. Hence, it is important to make them feel comfortable and make them realize that what is being evaluated is only the architecture and not the people who are responsible for it. It is important to communicate that the evaluation will be objective and is done in the interest of completing the project successfully. It is important for architects and managers to understand that having an eye from the outside provides an excellent situation to get feedback and find out if the design has weak spots. After all, the project team might have been working on the project for a long time without being able to clearly see certain problems (the same problem as code blindness, where a programmer cannot find his own defects).

The stakeholders are also involved in most of the reviews. It gives them an opportunity to voice their individual concerns. (For instance, if the head of customer service wants to have an account search facility that is cross-referenced by certain esoteric customer vitals, he can continue to make the case for its inclusion.) The stakeholders can also either agree or disagree if the showcased architecture meets the project goals.

The evaluation team must be assembled in alignment with the goals and purpose of review and in a way that addresses the following:

- The team must be perceived (by members of the development project) as impartial, objective and respected. The team must be seen as being composed of people appropriate to carry out the evaluation, so that the project personnel do not regard the evaluation as a waste of time and so that the team's conclusions carry weight.
- The team should include people highly skilled in architecture and architectural issues and be led by someone with experience in designing and evaluating projects at the architectural level.
- The team should include at least one system domain expert, someone who has built systems in the area being evaluated.
- The team should be located as close as possible to the source of the artefacts it will examine. Locality will simplify logistics and enhance communication with project personnel.

Domain knowledge and *unbiased views* are two essential characteristics to be shown by the personnel involved in the review.

What Should Be Reviewed?

What can we review exactly when we talk about 'architectural review'? A review process must have a *well-defined goal*. The deliverables and the inputs to the review process should be well defined and this is as essential as the need that the qualities being looked for should be well defined. Any of the following can be evaluated as per the need:

- *Functional requirements*: Evaluating the architecture from the point of view of functional requirements is straightforward. The evaluation teams check if each of the functional requirements is fulfilled by the architecture and how it is being fulfilled. If the architect does his or her work carefully, this should not cause any problem. Checking for functional requirements makes sure that none of the critical functionalities has been missed by the architect. The evaluation team goes through all the use cases and each item in the SRS (software requirements specification) document and checks if the team are satisfied with the architecture. Most architecture tools will help in this process though you really do not need anything more than a simple scoring system.
- *Non-functional (quality) requirements*: The non-functional requirements or the quality attributes are more challenging to evaluate as they create many indirect effects. For example, a layered architecture is more expensive and slower to build than a fat client, but for future changes the layered architecture is more flexible and does not limit development. The evaluation team takes into account all the typically used quality attributes even though the project members have not made them a requirement (or even seen them as important). It is the responsibility of the evaluators to come to the same conclusion (that individual quality attributes are not important enough to focus on) or to point out possible obstacles on the way (that individual quality attributes pose a problem for the project). For example, the project group and the stakeholders may not have portability in the requirements specification because they might think it is not important (or do not realize its importance) for the project. The evaluation team might have had experience with similar projects (and their aftermaths), so the team may suggest that the issue of portability is a necessity and project a time frame for its need in the project.
- *Architecture description*: The architectural description documents can be reviewed for completeness, consistency and correctness. They can be reviewed against the requirements and the standard notations being used to check for discrepancies.
- *Overall architecture issues*: Completeness, correctness, consistency, feasibility and testability of the architecture.
- *Future changes and impact*: This can be a 'what-if' analysis to judge the impact of possible future changes in the architecture and changes in the environment that can lead to architectural changes. This kind of evaluation is usually done to check for 'modifiable architectures' that are assumed to serve for long durations.
- *Architecture process*: The process of architecture itself can be reviewed to check if it has been followed. Believers of the thought that 'in process lies the quality' recommend and practise this kind of evaluation.

How to Review Architectures?

Let us ask the most important question. How to review? Architectural review, like any other review process, is to be carried out in three major steps—fulfilling the pre-conditions, conducting the review and generating or documenting the outputs/findings.

The *pre-conditions* are the set of activities that must be performed before a review can be performed successfully. These include a proper understanding of the review context, the plan of the review, obtaining proper organizational and logistic support, building the right team for review and, most importantly, getting the right and complete set of documents. These documents include the architectural representations, evaluation criteria, non-disclosure agreements from the involved parties, and so on.

Next is the actual review process itself, where the architecture is evaluated using one of the decided methods (discussed next). But simply executing the methods is not enough; a thorough record is to be prepared. All issues/concerns/comments/risks that come into the picture as a result of the review process should be elaborately recorded along with the comments from the reviewers.

These issues/concerns/risks should also be *ranked* after discussion among members of the review team.

As the concluding step, various outputs should be generated. These include the reports of review results, ranked issues, enhanced system documentations (as clarifications and prioritization occur) and the cost and benefit estimations for implementation of changes if any.

Case Study: Evaluating the Architecture of a Futuristic Travel Search Engine

Trav'mart is one of the major world players in the travel, hospitality and leisure industry. It is one of the most geographically diverse and vertically integrated travel distribution companies. It possesses holiday homes and resorts all over the world. Trav'mart provides hospitality and leisure facilities to its members, but the major share of its revenue comes from the business and leisure travel service. Its customers include leisure travellers, corporate travellers as well as small businesses, such as travel agencies, travel management companies and travel suppliers, including the different airlines and hospitality and leisure market segments.

Trav'mart manages a travel distribution system (TDS) that has more than 5,000 customers located in nearly 120 countries throughout the world. Earlier, the TDS dealt with 10 distinct brands, each representing some service from the segment, but Trav'mart slowly abandoned the practice of managing these complementary businesses by brand and aligned the assets under business units designed to serve the unique needs of its customer segments.

Recently, Trav'mart felt that the industry is taking a new turn. It found that no single company could offer end-to-end solutions and meet all the travel needs of customers. Collaboration of travel players and all support organizations alone can make complete service offerings. Trav'mart also realized that the Internet is a big facilitator for this unity. Trav'mart already has its presence on the Web and uses e-commerce as a major business medium to establish and develop its business. However, Trav'mart sees Web-based travel search engines as the future of the business.

Now the big question before Trav'mart was to find a search engine product that could perfectly match its requirements and one that could be easily adapted to its current as well as predicted business processes.

The Changing Business Scenario

The world is now perceived more as a 'global village' and travel has become an essential part of business for every industry. Traditionally, travellers were restricted to business travel and leisure was a secondary concern. But now retail leisure travellers are the kings of this segment. Whatever the customer type, corporate (business) travellers, leisure travellers or resort travellers, travellers now look forward to more services that are personalized for their comfort and pleasure. This personalization is inclusive of personal preferences for car, AC and radio stations; biometric check-in in the plane and hotel; perfectly equipped rooms with right entertainment, newspapers, menu choices and even toilet items; arrangements for meetings and even suggestions and bookings for sight seeing or shopping.

Every customer today would go for air, hotel and car rental booking together, which will involve minimum effort and money but maximum possible comfort and pleasure. Customers also look for more service offerings such as sports, leisure tours, cruise, amenities, and so on.

Usually, a customer's information may be present in fragments in various databases of travel segment businesses. At one place the customer's information is present as a frequent airline traveller and at another place it may be present as a VIP corporate customer, though in reality it belongs to the same person. This information is maintained by individual businesses to customize and personalize the services and products offered. In the current scenario, there is hardly any information sharing between these various parties. Hence, there is no significant personalization of services or quality control over services offered by the travel industry. This record is known as a passenger name record (PNR).

Glossary

Passenger Name Record (PNR)

PNR is a single record containing passenger information and is used by the travel agent/service provider to uniquely identify the passenger.

The same person may have different PNRs for different services.

The present-day customer has grown wise enough to base his decisions on available information such as direct information, market analysis, trends and projection and experience. This has become a major challenge for most travel agencies, hotels and resorts. They are receiving ever-increasing requests for information-rich content. They are expected to serve the information through various distribution channels such as Web sites, call centres, travel partners and agents with extended multilingual support, graphical contents and guest-focused product offerings. Usually, the relevant information is presented to the customer as contents on the Web or as responses to the search queries they place.

Until now, information was presented to travellers in parts, and hence often keeping the travellers away from getting the best possible value for their money. Also, the traveller could not plan his itinerary in one step. He would book the airline separately, book the hotel separately and would even book the cab to his hotel from a different service provider. This may happen because the segment is so broad and distributed that it is tough to find a single provider who can provide satisfactory services. Though some travel agencies tried this through what is known as 'the Direct Connect', where they offer multiple products or offerings through a single portal, the benefits of 'collaborative business' were missing.

Business players found that if they could devise a way to exchange information about their products and services among themselves and with travellers, then they could maximize business as well as enhance customer satisfaction. Global distribution systems (GDS), like Trav'mart's TDS, were the first steps in this direction. However, the inability of GDS and Direct Connect to cope with the evolving business needs separately and the need to leverage the strengths of both led to the concept of Integrated Direct Connect (IDC).

Glossary

Global Distribution System (GDS)

GDS gives the most recent/real-time availability/status of products. For example, booking for a flight is done by several agents. In order to know the latest booking status, that is, availability of seats, some connection with the centralized repository of the airline is needed. GDS provides this link. Airlines connect their inventory database to the GDS. All availability queries are forwarded through the GDS.

IDC (Figure 4.1) is a network of high-speed, reliable broadband Internet connections between travel service suppliers, central reservation systems, selected travel agencies and corporate clients that allows in-depth access to inventory and customer data. It also allows the suppliers to price and display offers effectively. It aims to meet all aspects of a traveller's itinerary as well. It can be developed as an interactive system, providing counselling to both customers and suppliers, and hence supporting an open data exchange between suppliers for obtaining maximum business.

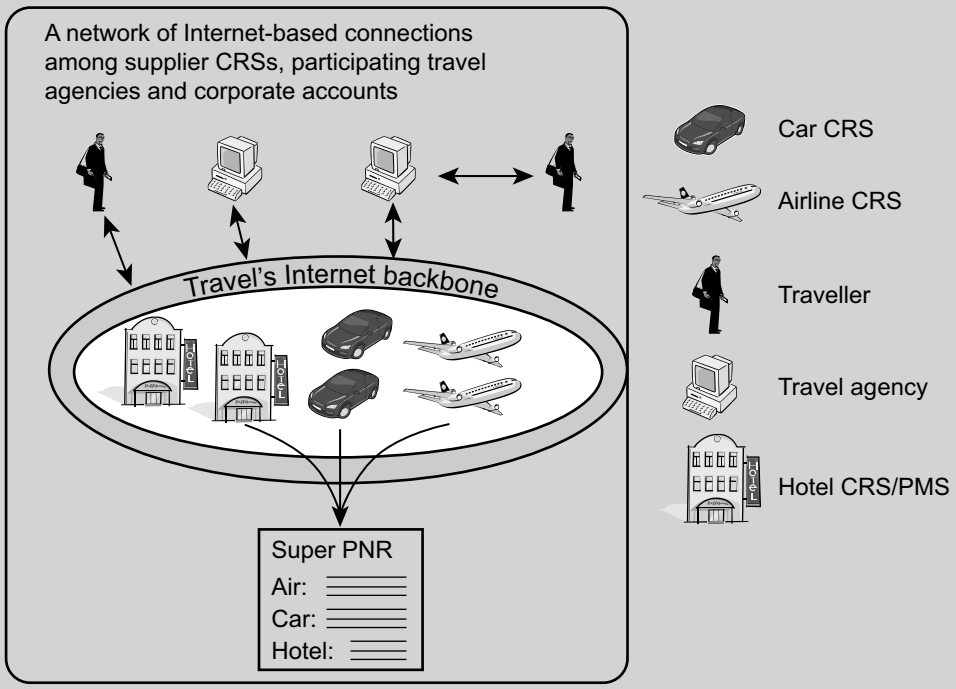


Figure 4.1 Integrated Direct Connect—a perspective

The concept of Super PNR has also emerged along with IDC. This global pattern contains all the distributed PNR information for a passenger as a single record. It can be used to provide personalized services while taking care of privacy aspects. Hence, IDC is bound to change the very look and feel of the travel industry dramatically. But adapting this concept depends upon the capacity and interest of the companies involved in the travel business.

New technologies also open the door for multiple distribution channels. For most of the businesses, geographical distances have been reduced with Web-based solutions. Not only can the information be disseminated through a variety of media but also the searching and booking can be done through the Internet using various media such as wireless, cable, interactive television (iTV), PDAs and even mobile telephones.

Today, business is based on collaboration and consolidation. Customers like to have complete solutions at one stop. For example, a single online request should provide for travel (airline or railroad) booking, hotel accommodations, car rentals, amenities, facilities and others travel needs. Any single vendor cannot provide this solution. Every vendor is contributing a part of the solution. To collaborate, we need a system to cater to all dynamic and efficient searches spanning across applications and organizations.

A flexible product definition is another major requirement of the business. The product definitions change constantly to be at par with the business needs. In fact, today's travel business is in need of a flexible product definition where the product can be configured based on the search request. The search query can lead to a combination of more than one product and provide solutions to the customer at runtime. For example, if

the customer is interested in a four-day stay package at a resort, and searches for it, the search result may also combine the airplane booking to the resorts and may present the total package.

The product definitions also depend on a number of other factors such as seasons, average inventory, product definitions and categories, pricing rules, and so on. Hence, the travel agent, or call centre executive, should be able to define or configure the product in accordance with the various factors that are affecting them, or the product should adapt itself depending on the factors. For example, a local festival can be included in the package if the package is to be used during the time of the festival.

Some changes in the product may be needed to meet the personal preferences of the traveller. For example, an international cricket match can be included in the package if the traveller's PNR/profile shows his interest is sporting events. This can contribute towards personalization of search results, thus serving potential customers to the best of its capacity. There should be room for defining new products that are optionally based on some old product. But there may be a limit to the level of flexibility. This constraint comes from the business motives, goals and operations of an organization.

Search in Travel Business

Search is a major technology enabler for the travel industry. There are enterprises in the travel domain whose businesses surround only searches. In using any travel system, it is observed that every end user (internal/external) likes to place many queries on the system. These queries are expected to retrieve data from one single application, an integrated set of multiple applications or a chain of applications spanning across organizations.

Searching in the travel business is done for two purposes: either for the 'book-up' or for the 'look-up'. The 'book-up' search is for booking tickets immediately, and the 'look-up' searches are simply for seeking information and may or may not get converted to business. These searches usually deal with products such as airplane seats, rental cars, tour packages, hotel rooms, concert tickets, tickets for some sporting event, and so on. An inventory of these products is maintained and a search is also done for knowing the current status of a product. A search can be performed by any of the customers, corporate clients, retail leisure customers, resort members, travel agents or booking houses.

Glossary

Book-up Search

It is the search for booking tickets immediately.

Look-up Search

It is a search that seeks information and may or may not get converted to business.

Usually, potential travellers and travel companies search for either the contents or the inventory or a mix of the two. The inventory search is done more by corporate travellers, travel companies and service providers. The users perform either a direct search or an indirect search. While performing the direct search, users are well aware of what they want; most corporate travellers fall in this category. In an indirect search, there is a content search prior to the availability search. This is a type of a yellow page query that may or may not result in a direct search on inventory later. For example, once the destination had been identified, that is, the resort or city is identified, there is a search for a room in the resort or a search for seats in the flight to that city. If available, there may be a search for 'just the right price' for the product. This is popularly known as the pricing search. If the prices and the availability match, then the traveller may 'book-up'. There may be some delay between this 'look' and the 'book' and, hence, the book must be based on the most recent data. For this a GDS is a handy tool.

A GDS gives the most recent/real-time availability/status of the products. For example, booking for a flight is done by several agents. In order to know the latest booking status, that is, availability of seats, some connection with the centralized repository of the airline is needed. GDS provides this link. Airlines connect their inventory database to GDS. All availability queries are now forwarded through this GDS and hence the booking agents/agencies get the most recent picture and they can take decisions based on real-time data.

The GDS-based searching and booking is the most popular solution for the travel industry. But most of the GDS are private and connect only a limited number of service providers (airlines, hotels, etc.). They may not be integrating all the products and services belonging to the domain. This restricts the benefits in terms of options and centralized information that can be transferred to travellers. Since the business is dependent on the quality of the search functionality, the focus is on building elegant search solutions that will cater to the needs of all types of users, all types of travellers, all types of products and all types of searches. The focus is also on developing the progressive solutions that will be useful in the future.

Whatever may be the search type, a common complaint from most end users is that the search takes a very long time and that the search does not provide optimum matching results.

Some of the key challenges for a search are as follows:

- Handling volumes of data in a heterogeneous environment
- Complex search process spanning across applications
- Demand for optimal response time
- Ever-increasing demand for information-rich content
- Simple user-friendly interface for search request patterns

Apart from these there are many other problems to be addressed. These include data security, network complexity, network load, data presentation for dynamic searches, distribution channels to cater for responses, and so on.

IDC can be very well integrated with a search. This dual combination can provide maximum benefit to the travel industry. Today's travel solutions demand more flexible systems that are capable of managing multiple products, multiple types of users as well as personalization of search results.

Trav'mart was looking forward for such a business solution that could add value to its business and also address all these issues. They wanted to change the face of the travel business.

Problem Definition

With a firm belief that IDC is the key to a successful future, Trav'mart started planning for a search engine supporting the IDC. They planned for a complete search engine solution that will cater to the present business needs as well as pave the way for integration of so-called futuristic trends such as personalized services, configurable inventory, seamless integration between the search and booking, and so on.

Since Trav'mart was a leader in the resorts business, it wanted to develop the first consolidated inventory for all their assets

Glossary

Inventory Search

This search is used more by corporate travellers, who are sure of what they want. They are aware of the destinations and other products. The searcher's aim is to find the current status of the product. Similarly, travel companies are always keeping a check/log of the availability of this inventory. Whenever a company needs to do a booking, it does an inventory search. Hence, the search for inventory is centred on the search for availability. This is done to know if a particular product or service is available, for example, if a particular hotel accommodation is available.

Content Search

Content search is more common with leisure travellers, who wish to know about possible destinations. Here, the searcher has only a vague idea of some preferences for the destination or services he wishes to procure. This vague idea or preference is expressed in terms of some keywords. Some supplementary information from the customer's preferences stated in his profile or PNR, not directly stated by the traveller, may also be used to make the results more suitable.

Direct Search

Here the customer is well aware of the source, the destination where the customer wishes to go, the route to be followed or the time taken in the journey. Hence, the queries are direct for availability of a product or a service. In fact, most corporate travellers fall in the category of direct searches, as they are well aware of the source and destinations.

Indirect Search

Inventory search not necessarily begins and ends with the search for availability. Often, a content search may take place prior to this search. The results of the content search are used to pick some destination(s) or product(s) and then the availability of the selection is checked.

located at different geographic locations. They planned for a complete asset management system that would not only manage the inventory but would also facilitate search through the inventory and booking. They had a vision and a roadmap on how to proceed. A legacy system was already in place that served some of the purposes. Trav'mart did not want to throw away this legacy system immediately; instead, it wanted a real-time synchronization of the proposed search engine with the existing legacy system. They came up with a request for proposal (RFP) for this Web-based system that will provide the complete search solution, and hence counselling services to the intended users.

Glossary

Indirect Search (*Continued*)

Here the customer is not so sure of the destination. He may give certain choices or preferences, and based upon these choices, the search should return appropriate results. For example, a typical search scenario may be 'the place should be under 500 km, some music concert should be around or some particular food service should be available at the place'.

The requirements and expectations as stated in the RFP are as given in Exhibit A.

Exhibit A Part of RFP, stating the requirements at a broad level

Build a search engine for a travel company, which allows searching for resorts, fitting in the required criteria available for booking.

Functional requirements

1. User enters the search criteria (multiple resort IDs, date range, multiple resort amenities, multiple region hierarchy (region/sub-region/market) on the screen and submits the same
2. User defines the result sort order
3. System supports multiple levels of validations
   a. User's access control to execute the search
   b. Validation of the search criteria
4. System executes the search criteria
5. System returns the results in the required sort order
6. User can drill down on the returned results and get more details

Non-functional requirements

1. Performance of the system is a critical aspect—with 5 million records in the database, the search should take a maximum of five seconds.
2. Recommended technology is J2EE using WebLogic. Any COTS component can be suggested with proper justification.

Expectations

The stress, while defining the architecture, should be on business components and database rather than presentation. The architecture should identify the following:

1. Various layers in the architecture
2. Various business components in each layer

3. Responsibility of the component
4. Data transfer across layers
5. Considerations for performance of each of the above, as well as the database

Proposed Architecture for Trav'mart

Fact-Tree came up with a travel search engine system, a search and book solution that seemed to cater to all the needs of the user.

The search and book solutions provided the following functionalities:

- Product manager
- Content search engine
- Inventory search engine
- Virtual tour components
- Business intelligence with forecasting and planning

Product manager: The product manager has three sub-components:

- *Product definition*: This component is used to define the products. This definition is market driven and is based on various search patterns. Some of the major factors are hotel, travel, air and car rental, amenities and facilities present, events organized, seasonality, customer profile, experience profile, availability, pricing, and so on. This needs to be adaptive to searches and can be configurable runtime.
- *Product analyser*: This component analyses the product and presents the statistics and analysis results. It supports various analytical tools to provide the analysis services. The analysis can help define and forecast the life of the product. Based on various analysis and statistical techniques, a report will suggest the desirable changes in the existing products to bring new life to them.
- *Product personalization (configuration)*: Personalization is the key factor for the success of a product. Every customer should feel that the product is an exclusively designed solution for his specific needs. Though while defining the product most of the requirements or factors are considered, in actual use very few factors are applicable to the specific customer. This personalization forms an important part of the search strategy.

A product metadata repository is managed for inventory and its contents. The product manager uses the metadata inventory repository and contents to define products. The content management products support product definition as well as product configuration on the fly, based on online search patterns.

Content search engine: It consists of a content management component to provide the facility for managing the content management services and searching for contents in the inventory. This also includes the facility to submit, modify and update the contents.

Inventory search engine: This component comprises various tools and techniques that facilitate the inventory search.

Virtual tours: This component is more useful with interactive media such as iTVs. It usually forms a part of the content search, where the traveller may take a virtual tour to find more about the services or products he is interested in.

Business intelligence with forecasting: Though not directly related to a travel search engine, this component is a part of the complete solution for a travel company. This component uses various data warehousing

products and techniques that can help in forecasting and planning. For example, dimension cubes and slicing and dicing queries based on different search patterns can provide valuable analysis.

Fact-Tree travel VBU took up this challenge and came up with an appropriate architecture providing a lot of scope for reuse, maintainability and scalability. The architecture consisted of three major components, each one complementary to the others. These include

- The search component
- The caching component
- The pricing component

The architecture is shown as Figure 4.2.

The search component

A complete travel search engine should address the following search-related aspects in detail (given as Figure 4.3):

- Search request configuration
- Search request processing
- Data/content retrieval from voluminous distributed data across various applications and organizations
- Search response collaboration and presentation

The search request configuration facilitates the formation of various types of search requests, criteria classification schemes as well as search patterns. These search patterns are broadly classified into four categories—lifestyle/customer profile, product parameters, availability and pricing. These patterns are defined on the basis of market analysis and customer demands.

A search request can be mapped to one particular product or combination of products that were defined using the product manager. Based on the search request, the search processing should configure the product and provide a combo solution as a response solution. Sometimes the search cannot be mapped exactly to one product, but some nearly matching patterns can be identified. In such cases, multiple results can be provided to the customer along with a ranking.

Various search algorithms can be used based on search patterns. The search can be processed in multiple threads in parallel leading to quick results. The collaboration of all these results after matching, ranking and sorting can be published. A number of industry-standard search optimization algorithms can be used for the purpose. These include

- Artificial intelligence methods for blind search, breadth-first search, uniform cost search, depth-first search, depth-limited search, iterative deepening search.
- Inventory data segmentation can prove to be of great convenience. This can facilitate the slice and dice of data, hence leading to faster data retrieval. The search techniques should be intelligent enough to map the appropriate data segments for retrieval.

The search request may come from any channel—Web, iTV, PDA, and so on. The result display should be formatted depending on the channel in which it is to be displayed.

Caching component

Efficient data retrieval is the most critical factor in a search, as the entire response is based on how efficiently we can mine and retrieve the data. Data caching is one of the most recommended and optimized solutions for this purpose. A cache manager component will take care of all these aspects and is presented as Figure 4.4.

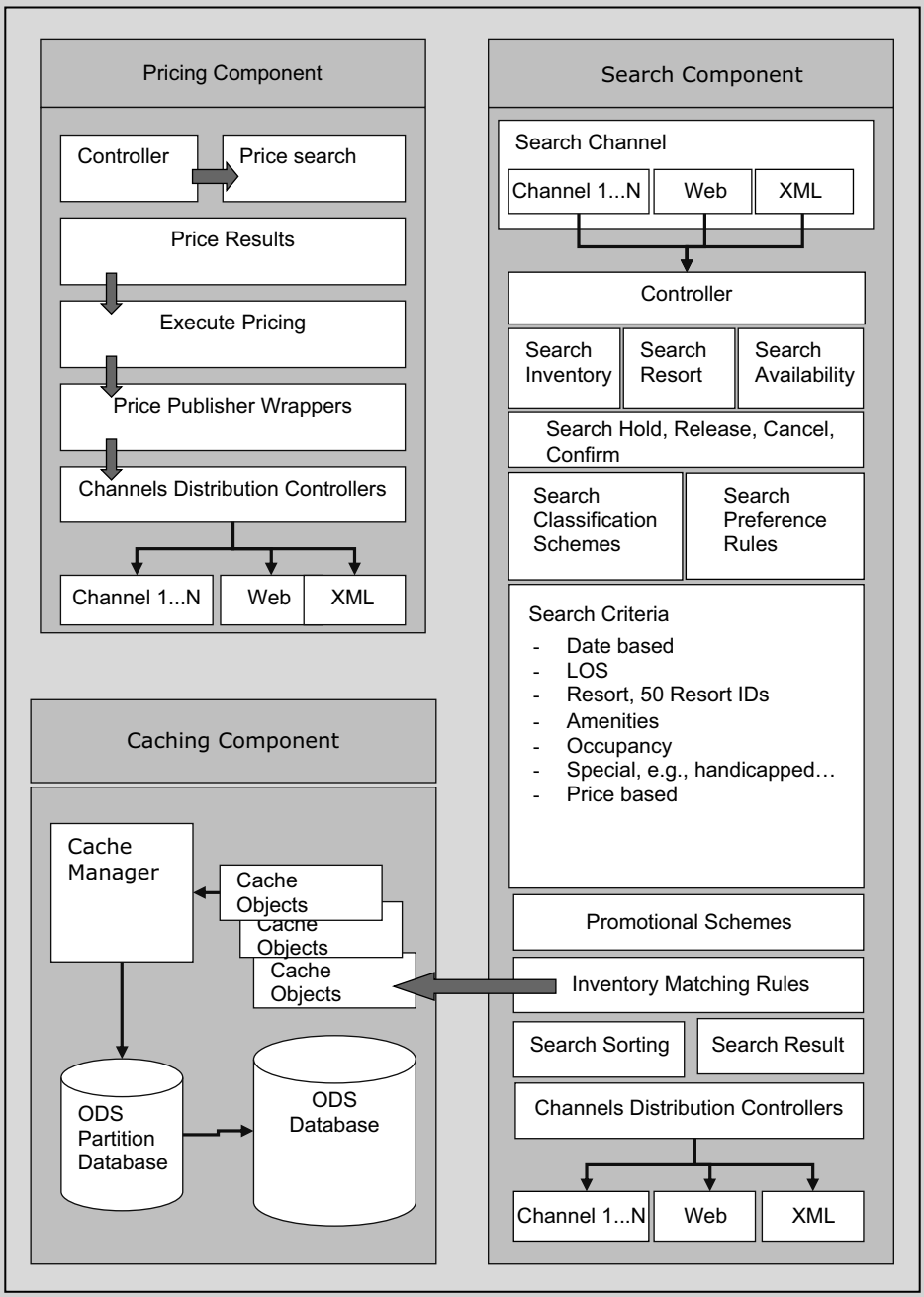


Figure 4.2 Proposed architecture for the travel search engine

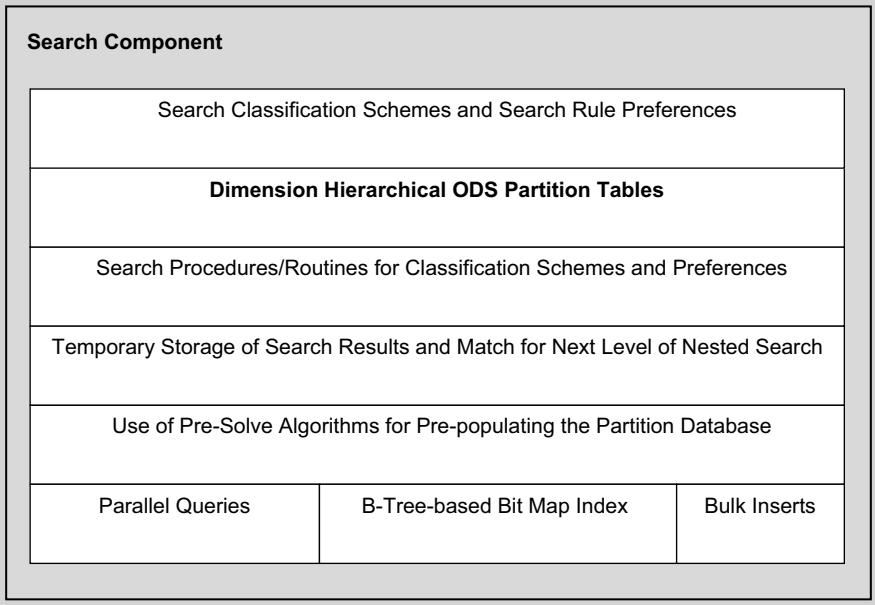


Figure 4.3 The search component for the travel search engine

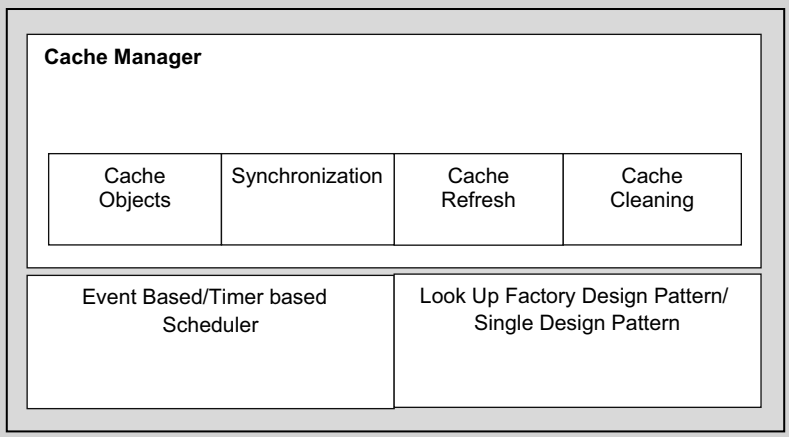


Figure 4.4 The caching component for the travel search engine

There are various caching patterns and caching engines that can be used to make a search more efficient. Some of the architectural aspects that form the essential part of the search engine are caching the product configurations on the fly, cache cleaning, cache synchronization, cash refreshing, availability of data in the cache based on the LRU algorithms and distributed caching.

Caching patterns for search response optimization

Based on various alternative solutions, some of the key caching patterns adopted for data retrieval are as mentioned below. Caching patterns can severely affect the hit-to-miss ratio as well as increase the search speed.

Caching patterns vary based on the usage and type of data we would like to cache. The various factors that need to be looked into are the demand and usage of pattern of data, load conditions of data, volume of data, whether data are static or dynamic, whether data reside in the local environment or are distributed across, and so on.

Based on all these factors, the caching patterns, their configuration as well as the implementation will vary.

Pricing component

Travel companies have a pricing model associated with each travel service they provide. This pricing model is a set of rules under which different prices are offered to the customer. For example, the hotel room charges vary depending on the season as well as the number of rooms booked at a given time. For some products, such as airplane seats, the rates offered to different customers may be modified at very short notice, like 8–10 days before the flight, in order to attract more customers, or if there are not enough customers for a particular flight. Thus, the pricing model also includes various discounts and promotion schemes for a product. The rules are stored in the form of a pricing table.

Some pricing models do the inventory search, as pricing rules depend on inventory, and the model might change its rules based on the search results. But the pricing tables need to be organized, and often search engines search through them in order to find a suitable result for the traveller's query. For example, if the customer is interested in a low-cost leisure package, then the pricing model is to be searched to find the right one within the specified budget.

Further Business Opportunities

TravelPeople, another business house, is looking forward for a similar kind of product. Apart from the usual requirements as given by Trav'mart, TravelPeople is planning for a network of some 6,000 travel agents, each specializing in a set of brands, in order to form a huge product base for content and inventory search. TravelPeople is looking for plug-ins that can be used by agents to simply download and use with its current system so that the agents get connected with the TravelPeople network and use its product and offering repository without leaving their own systems. In addition, agents should be able to customize the existing offerings in order to personalize them for the customer.

After receiving these specifications and business from TravelPeople, Travel VBU is sure that more business is in the pipeline if it can build the right architecture with reusable components and implement effective solutions for both Trav'mart and TravelPeople (both are fixed-bid projects).

Deven is the project manager for the Trav'mart project. He and his team members are enthusiastic to convert the project into the product. They have done good work previously and they feel that they can bring modifications to the current architecture and can make it suitable enough to cater to the current as well as the future needs of the travel domain and serve many other clients like Trav'mart and TravelPeople. All the clients of Fact-Tree Travel VBU, the travel business leaders, would like to make sure that they are well prepared for all the future travel needs.

Deven is also exploring the reusable COTS components for coming up with the solution. He is excited about the possibility of using state-of-the-art technology solutions, such as service-oriented architectures, intelligent-agents-based systems, distributed systems employing grid computing, as some of the options for the 'search and book solution'.

Deven is planning for an architecture trade-off analysis method (ATAM) evaluation of the current architecture to ensure conformance with all requirements, functional, non-functional and even domain requirements.

The major non-functional requirements include performance, 24 × 7 availability, maintainability, scalability and extensibility.

He has listed the purpose of evaluation as follows:

- Check for suitability of the current architecture for meeting all non-functional requirements or quality attributes of the system.
- If some quality attributes cannot be met by the current architecture, determine the changes that need to be done so that the new architecture meets all of the quality attributes.
- Determine which of the quality attributes can be met fully and which of them can be met partially.
- Determine the risks associated with each of the quality attributes.
- Come up with an improved and well-written document that captures and communicates the details of the architecture very effectively.
- Determine the suitability of the current architecture for meeting all the functional requirements of the system.

Help Deven and his team to evaluate the architecture using the ATAM or any other framework to identify the architectural requirements and to decide on the approach to come up with the complete enterprise solution that can cater to the needs of both Trav'mart and TravelPeople, and eventually develop a product that will change the face of the travel industry. Please keep the following guidelines in mind while coming up with a solution for this case study.

- The team finds that there is a huge scope for introducing techniques and technologies for handling the concept of 'configurable inventory'. One suggestion is the use of 'XML-based language for travel product definition'. Evaluate this solution for cost and benefit, the advantages it can offer and the costs involved (direct or indirect).
- The team feels that it can go for reuse of components for the 'search and book' solution. Plan a strategy for reuse. Identify the reusable components and record the arguments, for and against, while deciding on the reuse strategy and components. But they find that time is a scarce resource right now. Suggest some alternative ways for exploring the various options of reuse.
- The team feels that the search solution can be a Web-services-based system, that is, a service-oriented architecture, the latest trend in the industry. Or the search solution can be a distributed application. If one booking agent/agency is unable to provide suitable solutions to the search query, they pass it on to some other agent. If these possibilities are to be traded off, then what should be the focal points? What kind of architecture solution may prove to be most apt for the proposed system?
- The complex nature of the business brings in interesting scenarios in the search component. Since there are various types of searches, for contents, inventory, pricing, and so on, the appropriate search component has to be employed based on the context. For example, corporate travellers usually go for direct searches and, hence, may not need the caching component with such great sophistication. It would be interesting to see if the system can be divided into a set of pluggable components, where users, especially the travel agencies or booking houses, may use only a component that suits their business needs as well as their budget. If that is not possible, do we need to develop a huge system, a complete solution for all the search and book needs? You may provide your insight on this point. Discuss the possibilities, pros and cons, as well as the alternatives.
- The travel industry is changing at a very fast pace. In fact, many companies and research houses came up with various scenarios and predictions for the future. We have discussed some of those scenarios earlier (personalized services, flexible inventory, additional services, etc.). When the architecture is being implemented, how can one take into account the future needs?

POSTMORTEM

This case study exposes interesting issues relating to architecture review and evaluation. The team is executing a very ambitious project that aims to change the way the travel business runs. The challenge is to come up with an open architecture. There are many stakeholders in the project, including travel companies such as Trav'Mart and TravelPeople, travel agents, end users who plan their travel and corporate travel help-desk personnel. This architecture is like a one-time investment, for the client as well as for the developers. Clients would like to ensure that the proposed architecture is capable of fulfilling their current and future business needs. The developer organization would like to ensure that the proposed architecture will stand the test of time. The proposed system is designed to be in use for a long period and hence system administrators will ask for a system that is easy to maintain, modify and extend.

But how are they going to ensure that all these requirements will be fulfilled? Obviously, they do not want to wait for the system to be completely developed so that it can be evaluated. Hence, the option is a *toll gate* architectural review. Though we have discussed most of the fundamental aspects of architectural evaluation, there are still many questions that need to be answered and more clarity needs to be brought in. Is the architecture ripe enough to be reviewed? How will the evaluation be performed? Will the reviewer team have sufficient and necessary skills for review? The best way to answer these questions is to perform the evaluation process by following a proven review method that will guide us at each step.

Techniques for Evaluation and Review

The whole gamut of review techniques can be divided into two fundamental types:

- Questioning techniques
- Measuring techniques

The questioning techniques are relatively simple and include methods like a general discussion based on qualitative questions, evaluation through questionnaire(s), having a predefined checklist and using it for review or using some well-structured methods such as scenario-based evaluation methods. Scenario-based methods are gaining popularity and a lot of research is being carried out in these areas. Various methods such as the ATAM and the scenario-based architecture evaluation method (SAAM) are gaining popularity and have shown encouraging results.

The various measuring techniques are more technical in nature and usually involve quantitative methods like collecting answers to specific questions, collecting metrics and through simulation and prototypes.

A Review Method for Architectural Description and Architecting Process

The focus of such methods is to evaluate if the architect performs the right activities and if the resultant architecture (being represented by the description) is good enough for further use. Here, the description document is the artefact that is evaluated. For this, it is assumed that the document has been created using some standard. This standard can be 4+1 views of architecture or can be any other standard as long as it is approved and followed by the organization.

The review method

As with other evaluation methods, collection of the relevant set of documents is the first step. An evaluation team that may represent all the stakeholders is put together. The documents are read by the team members and commented upon.

The team uses some standard to evaluate the documents and check if the given architectural description represents the concerns of all the stakeholders. The standard being used as the checklist is usually a recommended practice such as the architecture description framework IEEE 1471.

This IEEE recommended practice, depicted as Figure 4.5, is for writing architectural descriptions, where the concerns of each stakeholder are represented adequately.

Often, more detailed and very specific checklists can be used for certain models or documents. The reviewing team may interview the stakeholders for clarifications, confirmations and comments or to understand the process that was followed for architecting, if needed. The team then reports back its observations, perceived risks and suggested improvements.

Whatever method is applied for the review, the typical issues that are under investigation are the following:

- Are all stakeholders considered?
- Have all requirements been identified?
- Have all quality issues been considered and selected appropriately?
- Are key concerns of all stakeholders identified and their viewpoints defined?
- Are these viewpoints rational and supported by facts?
- Are the views documented and conformance gained from the concerned stakeholder?
- What is the quality of the model/document that is being studied? Is the document organized, readable and consistent? Is it complete and free from ambiguities?
- Does the architectural solution being provided appear rational?
- Are the documents well managed?

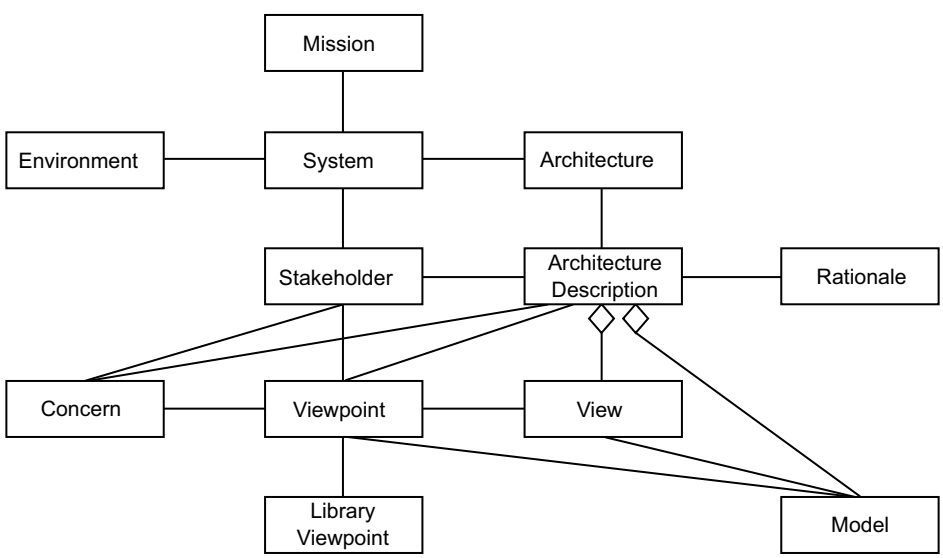


Figure 4.5 IEEE 1471-2000 recommended practice for architectural description of software-intensive system

Scenario-based Review Methods

Scenario-based methods are the most popular and advocated methods. They focus on analysing the architecture by *identifying* relevant scenarios (i.e., events/situations in a system's life cycle) and *reasoning* their impact on the system (architecture).

As is well known, a scenario is a well-described and brief narrative of the usage of a system, typically from an end user's point of view (and sometimes from the developer's point of view). The systems functionality is nothing but a collection of such scenarios.

Scenarios are also important from a completely different angle. They give very good insight into how a given architecture can meet the quality attributes. Fulfilling the scenario is a good way of measuring the readiness of the systems architecture with respect to each of the quality attributes.

Though architecture can be adequately judged on the basis of the quality attributes, understanding and expressing the quality attributes itself is a difficult task. Hence, a reverse mechanism can be used, where a scenario represents the quality attribute. As scenarios can be thought of and expressed easily, they can be used to evaluate architectures as well.

Many scenario-based methods have come into the picture recently. But most of them have been found to be restricted to a limited set of quality attributes they are suitable for. And most of them are enhanced or restricted versions of the first such method, the SAAM. There are many methods available for architecture review and evaluation (Losavio *et al.*, 2003). Besides SAAM, other popular methods that have evolved over time include

- ATAM, architecture trade-off analysis method. Though ATAM is an improvement over SAAM, ATAM is restricted to non-functional requirements only, whereas SAAM is suitable for all kinds of requirements and quality goals.
- CBAM, cost–benefit analysis method.
- ALMA, architecture-level modifiability analysis.
- FAAM, family architecture analysis method.
- ARID, active reviews for intermediate designs (specifically used for incomplete architectures).

All of these methods can be abstracted out and viewed as modified versions of a more general method that is outlined below:

- *Establish the goals of the review:* This can be to assess a single quality, compare various architectures, and so on. Select an appropriate review method as per the goals of the review.
- *Fulfil preconditions of the review method:* The usual preconditions include obtaining a right set of architectural documents, forming teams, circulating architecture details among the team members, and so on. The team should also set goal(s) and scope for the review process.
- *Establish the issues of review:* This can come through a preliminary study of the goals of review, requirements and architectural document.
- *Establish architectural description:* The architecture is described to all and clarifications provided. It is preferred to use some standard notations for architectural documents so that all those involved with the review can understand them.
- *Elicit and classify scenarios:* All the reviewers brainstorm and come with relevant scenarios that the system must satisfy or those that may impact the system. This can be done iteratively, or other heuristics can be applied to obtain the most useful and representative set of scenarios.
- *Determine the impact of the scenario on the various issues under consideration:* The interaction among the various scenarios can also be studied to check out the overall impact over the system.

The requirements and architectural descriptions can be augmented with the new information coming into the picture.

- *Overall evaluation and summarization:* The observations are summed up, trends are discovered and various issues/recommendations are finalized through consensus.
- *Report findings/recommendations/issues:* Though notes are scribed throughout the process, the final observations and findings are formally written and reported. The recommendations are given and often even the process is reported for purposes of learning and preserving knowledge.

Figure 4.6 summarizes these steps.

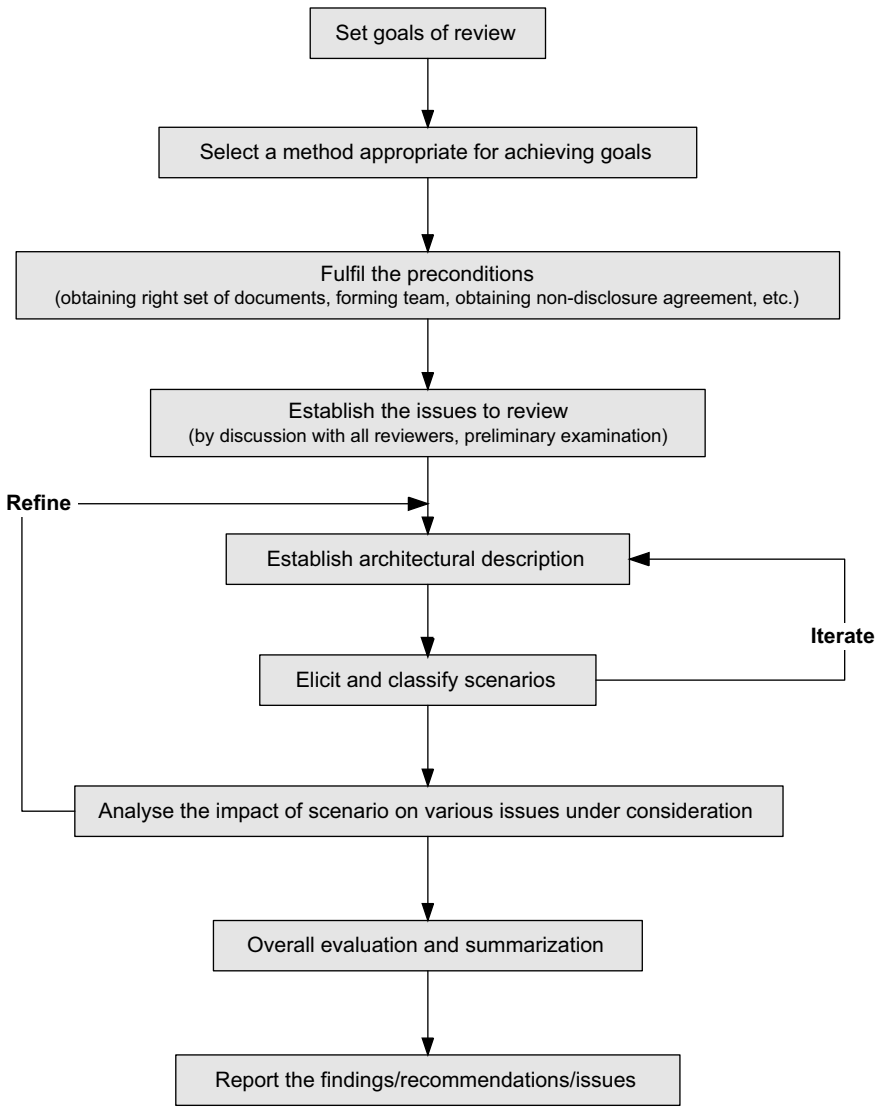


Figure 4.6 A general perspective on scenario-based evaluation methods

Most of the scenario-based methods involve the stakeholders in the evaluation process, where each stakeholder presents scenarios most relevant for his job description. Hence, the evaluation team relies on stakeholders' ability to write true, feasible and useful scenarios, though coming up with meaningful scenarios is not easy. Scenario-based methods, especially the ATAM, has become very popular in recent years and is one of the most recommended practices.

Scenario-based methods are not really scientific or formal. In addition, it is very hard to make sure that all the scenarios identified for the evaluation task are complete, comprehensive, sufficient and necessary.

Scenario-based methods depend on brainstorming, which is very popular but not necessarily effective all the time. It is possible that different subject matter experts have different opinions on the characteristics of each scenario and the business logic.

In such situations, we need a better way than brainstorming for identifying correct scenarios using failure determination methods such as anticipatory failure determination (AFD; see http://www.ideation-triz.com/AFD.asp.

CASE ANALYSIS

The travel search engine is a very ambitious product. It is clear that the team is very meticulous and wants to come up with the best possible architecture. The team has developed the basic architecture and wants to evaluate it. Hence, the team aims at validating the architecture before proceeding further.

The first important question to ask at this juncture is, 'Is this architectural description mature enough to run the evaluation process?' All we have is the logical architecture and its description. We know in detail the requirements against which we wish to evaluate the architecture. But is this sufficient? It may appear a difficult task, but not an impossible one.

Let us first establish the goal of this evaluation process. The team has explicit goals for the evaluation. Its aim is to validate the architecture to fulfil the system requirements. But many of the requirements, especially the properties such as response time, depend on the implementation details as well, and hence they are difficult to judge.

The best possible option in these circumstances is to use the domain or functional requirements as quality goals that are to be reviewed. The next task is to identity the non-functional requirements that are independent of the platform-specific or implementation details of the architecture. Take them as quality goals and check if the proposed architecture fulfils them.

The various functional requirements that are critical are

- Content and inventory search
- Collaboration with agent systems and other existing systems (legacy and other competitors)
- Pricing mechanism
- Management of inventory
- Personalized services and presentation of contents
- Presentation services such as virtual tours

The various non-functional requirements that become important here are

- Performance—Response time, number of concurrent users.
- Availability—24 × 7 availability and planned maintenance.

- Maintainability—The system will be used by various types of users, geographically distributed. The domain itself is evolving and the business rules change quickly. It should be easy to maintain, modify and evolve.
- Scalability—As the number of users are bound to increase with time.
- Security—The system will perform monetory transactions; hence, security is an important consideration.

These are the most important issues. There may be many requirements that will seem critical at times. These include the backup and restoration and connection with different types of systems (portability).

Which Method to Use?

The current system is still under development; hence, it is not possible to collect various metrics about its performance and other quality attributes. The best way to go about it seems to be to use a scenario- or checklist-based questionnaire. But checklists and questionnaires are more restrictive in this case as the opportunities for gathering the answers to questions and conducting brainstorming sessions for all stakeholders are very limited. Hence, applying scenario-based methods seems to be more appropriate. Since we have ample domain knowledge available, the scenarios can be easily built and evaluated.

Scenario-based evaluation methods usually bring forward many changes and open points that an architect must further reconsider. They are driven by the stakeholders' views as they come up with scenarios that represent their view, and the architecture is evaluated for the execution of those scenarios. Hence, the evaluation team relies on the stakeholders' ability to write true, feasible and useful scenarios. Coming up with really useful and representative scenarios is one of the most important and difficult steps in scenario-based methods. But the scenarios are also important in the sense that they help stakeholders in identification as well as proper expression of the current needs and future changes in the system. Thus, they improve the interaction between stakeholders and architects. Often, the scenario-based evaluation techniques highlight the rationale behind various architectural decisions. In fact, the methods were developed for enabling software architects to reason the system's quality aspects earlier in the development process (Ionita *et al.*, 2002).

Though a number of methods are available, we select a *modified* version of the SAAM method (Abowd *et al.*, 1997; Ionita *et al.*, 2002; Kazman *et al.*, 1994). We have modified this method to suit the needs of our case and will utilize some general quality-attribute-based evaluation measures to increase the effectiveness of our method.

Software Architecture Analysis Method

SAAM is one of the earliest scenario-based software architecture analysis methods. Though initially developed to asses the modifiability issues in mind, the various quality claims of software architectures can be evaluated with the help of scenarios. In practice, SAAM has proven useful for quickly assessing many quality attributes such as *modifiability, portability, extensibility, maintainability* and *functional coverage.*

When analysing a single architecture, the review indicates the weak or strong points, together with the points wherein the architecture fails to meet its quality requirements. If two or more different candidate architectures providing the same functionality are compared, then the review can produce a relative ranking.

Prerequisites and inputs

System quality attributes that are going to be evaluated in the review session must be addressed in a certain context. This imposed the adoption of scenarios as the descriptive means to specify and evaluate qualities. A

number of scenarios describing the interaction of a user with the system are the primary inputs to this architectural evaluation session. Besides scenarios, the system architecture description, the reference artefact on which the quality scenarios are mapped, must be available for all the participants.

Steps in the evaluation process

The process consists of six main steps, which typically are preceded by a short overview of the general business context and required functionality of the system (Ionita *et al.*, 2002). These steps include

- Step 1—Develop scenarios
- Step 2—Describe architecture(s)
- Step 3—Classify and prioritize scenarios
- Step 4—Individually evaluate indirect scenarios
- Step 5—Assess scenario interaction
- Step 6—Create an overall evaluation

Step 1—Develop scenarios

This brainstorming exercise aims at identifying the type of activities that the system must support. All stakeholders perform this exercise and each stakeholder contributes scenarios that affect him. The scenarios should capture all major uses and users of the system, all quality attributes and the associated levels that the system must reach and, most importantly, the foreseeable future changes to the system. The scenarios gathered so far can be grouped as *system scenarios*.

This exercise is usually performed iteratively. After each iteration, more architectural information is shared and hence more scenarios are gathered each time. Thus, architecture description and scenario development influence each other. The recommendation is to perform these activities in parallel. For example, some scenarios that we can develop for the travel search engine are the following:

- Time taken for completing the transaction after pressing the submit button with 54 concurrent users and a user base of 500, when it is a high volume online transaction such as a purchase order and the user is interacting through a broadband Internet connection.
- Time taken to complete the transaction when the user fills a complex Web form such as a vendor master over a 56K modem link and the system is concurrently handling some 50 users.
- What efforts will be needed when the product is being released to a non-English-speaking country?
- Will there be platform independence for the business logic layer, when the application is being ported to different operating systems and Web servers.
- Can a developer add functionality by adding sub-systems or components or simple modules? How much effort does one need?
- Will the services be available in case of failures of the Web, J2EE and database servers or the application system? Can one expect 99 per cent availability, with 24 × 7 uptime, all 52 weeks?
- What will be the authentication and access mechanism to the system for registered users? Do I need to sign into the system again and again to access various services of the system?
- Can I connect to different resources and collect rich (including audio, video and images) content about various tourist spots and add them to my database?
- Will I be able to extend the system to new information distribution systems such as iTVs and next-generation systems?

Step 2—Describe architecture(s)

The candidate architectures are presented in the second step. The description should be following a standard such as IEEE 1471-2000 or 4+1 UML views for architecture. The aim is to present an easy to understand static and dynamic representation of the system (components, their interconnections and their relation with the environment). The following can be included in the presentation:

- Driving architectural requirements
- Major functions, domain elements and data flow
- Sub-system, layers and modules that describe the system's decomposition of functionality
- Processes, threads, synchronization and events
- Hardware involved
- Architectural approaches, styles and patterns employed (linked to attributes)
- Use of COTS and other external components used
- Trace of one to three major use case scenarios including runtime resources discussion
- Architectural issues and risks with respect to meeting the driving requirements

Step 3—Classify and prioritize scenarios

The scenarios collected so far are classified as *direct* scenarios and *indirect* scenarios. A direct scenario is supported by the candidate architecture because it is based on the requirements that the system has evolved from. The direct scenarios are perfect candidates as a metric for the architecture's performance or reliability.

An indirect scenario is that sequence of events for which realization or accomplishment of the architecture must suffer minor or major changes. The prioritization of the scenarios is based on a voting procedure.

The analogy of use cases and change cases with direct scenarios and indirect scenarios make their identification easy.

Step 4—Individually evaluate indirect scenarios

In the case of a direct scenario, the architect demonstrates how the scenario would be executed by the architecture. In the case of an indirect scenario, the architect describes how the architecture would need to be changed to accommodate the scenario. For each indirect scenario, identify the architectural modifications that are needed to facilitate that scenario, together with the impacted and/or new system's components and the estimated cost and effort to implement the modification. Record all the scenarios and their impact on the architecture as well as the modifications being implied by the indirect scenarios.

It will be very useful to represent all the alternative architectural candidates in the form of a table (or as a matrix) to be able to easily determine which architecture candidate is better for a given scenario.

Step 5—Assess scenario interaction

When two or more scenarios request changes over the same component(s) of the architecture, they are said to be *interacting* with each other. In such a case, usually, a trade-off results and the architecture may need some modifications. A simple way may be to modify the affected components or divide them into sub-components in order to avoid the interaction of the different scenarios.

The impact of these modifications should also be evaluated for side effects.

Step 6—Create an overall evaluation

Finally, a weight is assigned to each scenario in terms of its relative importance to the success of the system. The weights tie back the scenario to the business goals and/or other criteria like costs, risks, time to market, and so on. Based on these scenario weights, an overall ranking can be proposed if multiple architectures are being compared. Alternatives for the most suitable architecture can be proposed, covering direct scenarios and requiring least changes in supporting indirect scenarios.

A scoring system can be used to indicate the importance of requirements and how the current architecture fulfils them. Each of the requirements is given a weight that indicates the importance of the requirement. The score is the estimate of how well the candidate architecture fulfils the requirement. An example is given in Table 4.1.

Requirement	Weight	Score	Total
Book a tour	5	5	25
Search for a tourist spot and routes from Hawaii to the spot	5	4	20
Change screen layout to suit PDA screens	3	3	9
Portability	4	3	12
Total			**66**

Table 4.1 Sample scorecard for evaluating software architecture features

The scoring can also follow the usual style of +1, −1 and 0, respectively, for present, absent and not required status of an attribute/requirement. Here, the total sum will give an objective image of the current system.

Increasing the effectiveness of your review

As mentioned earlier, we suggest a number of techniques that may be used with the basic process. One of the most time-consuming and crucial steps is to form scenarios that are meaningful, representative and helpful in the evaluation. Coming up with scenarios depend a lot upon the experience and expertise of the reviewer. It is important to understand that getting a good starting point at times becomes most difficult. The methods described below can help find meaningful scenarios.

Generate a quality attribute utility tree

The quality attributes are the driving force behind the architecture. There are four types of qualities:

- *Runtime qualities:* These qualities are seen when the system is under execution, such as functionality, security and availability.
- *Non-runtime qualities*: These are built in qualities, not depending upon the dynamic behaviour of the system. These include qualities such as modifiability, portability and testability.
- *Business qualities:* These qualities directly affect the business, such as cost and schedule for development and marketability of the product.

- *Architectural qualities:* These are the properties of the architecture process or the description. These qualities include conceptual integrity, correctness, completeness, and so on.

Though quality attributes can be the most direct way to evaluate the system, there are no universal or scalar measurements for quality attributes such as 'reliability'. Rather, there are only context-dependent measures, meaningful only in the presence of specific circumstances of execution or development. And the context is represented by the scenarios. Hence, we identify scenarios instead of quality attributes directly.

To come up with a proper set of prioritized scenarios, we suggest the development of a quality attribute tree. Starting from a node called 'utility', the quality attributes that are most important for the system and are under investigation can be added as nodes to this root. Each of these quality attributes can be further refined until we get well-defined and well-formed scenarios for each 'sub-quality attribute'. Refining the scenarios can be proceeded using the guidance provided by the ISO 9126-1 quality model (ISO/IEC, 1998), as in Figure 4.7.

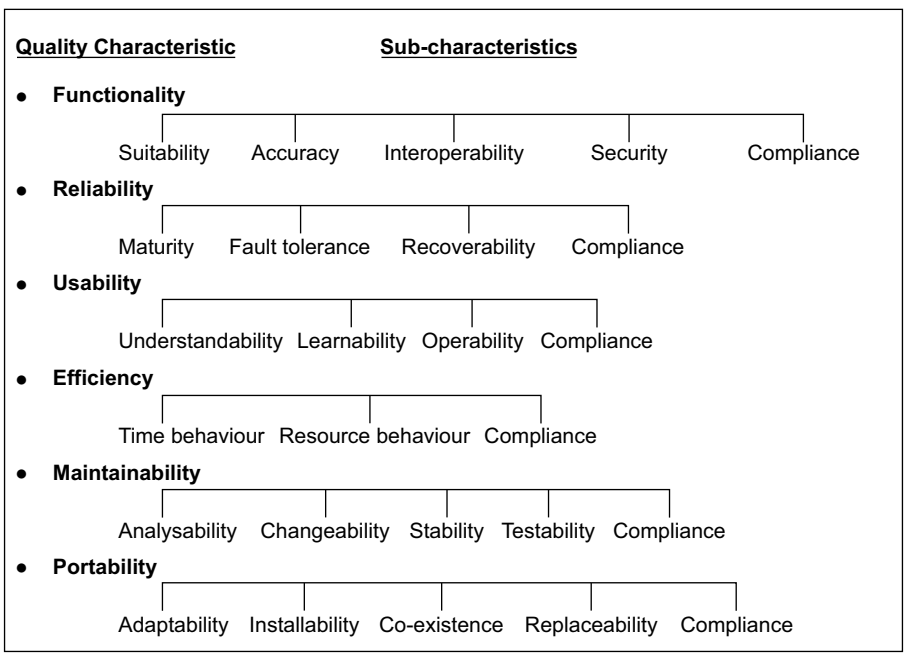


Figure 4.7 Sub-characteristics of the ISO 9126-1 quality model

Note that these sub-characteristics given by the ISO standard need to be taken only as a starting point. Typically, a modified version of the above standard gets adapted based on the expertise of the team, available information as well as the suitability of the application. The adapted sub-characteristics of the ISO quality model is further refined to several levels deep to create a utility tree for the application.

Figure 4.8 shows a sample utility tree that can be generated for the travel search engine after adapting the above-mentioned standard suitably.

This looks like a backward chaining process, but suits our purpose well.

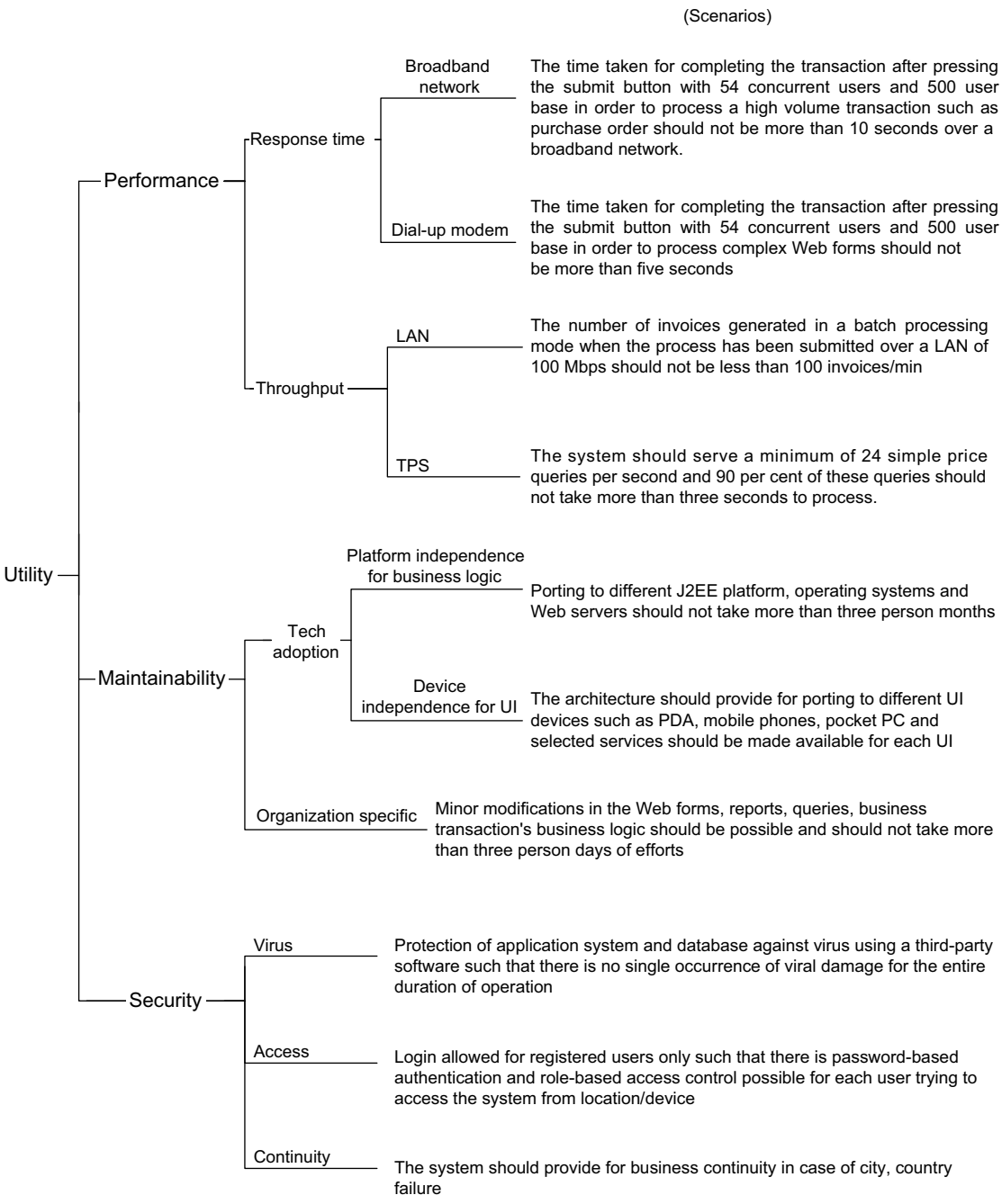


Figure 4.8 Utility tree for the travel search engine

Use quality attribute taxonomies

Another useful way to come up with scenarios is to use quality attribute taxonomies. Each of the quality attribute taxonomy gives the key concerns and the factors that affect the quality attribute. Knowing the concerns, scenarios can be refined; and knowing the factors, they can be evaluated.

Some sample taxonomies (Barbacci *et al.*, 1995, 2002) are given in Figures 4.9–4.12.

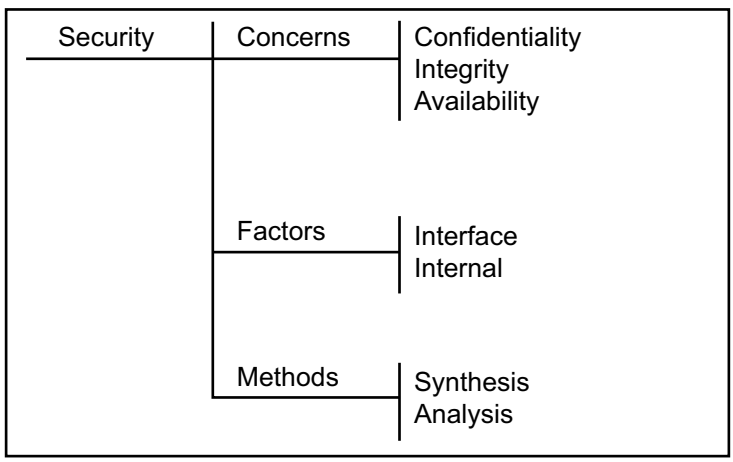


Figure 4.9 A taxonomy on security

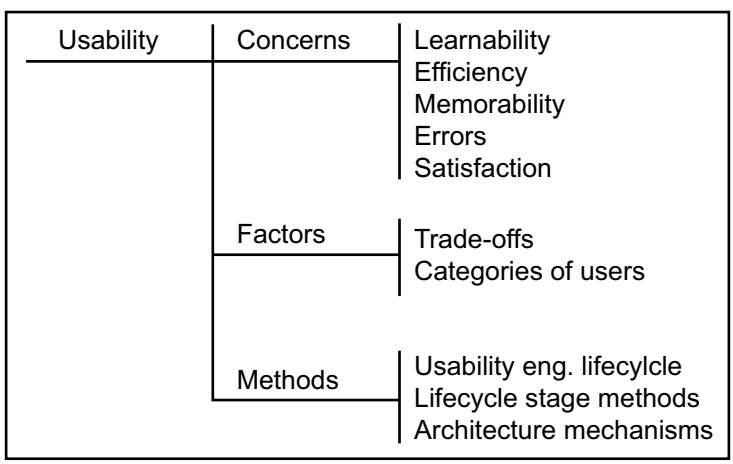


Figure 4.10 Taxonomy on software usability

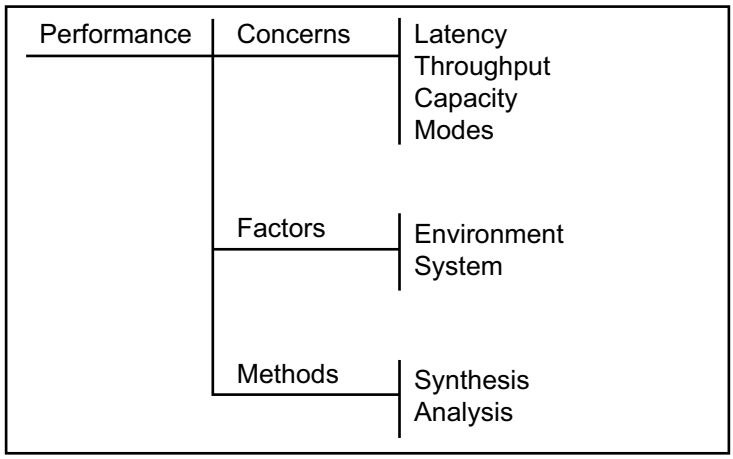


Figure 4.11 Taxonomy on performance

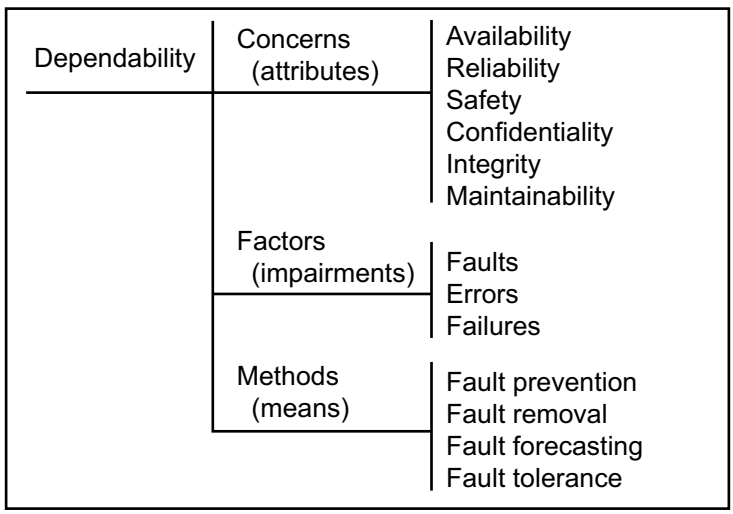


Figure 4.12 Taxonomy on dependability

Identify risks, sensitivity points and trade-offs

Though the ATAM method is developed specifically for this purpose, we suggest that whatever method is being used, the reviewers should try to classify their concerns as follows:

- *Risks:* alternatives that might create problems in the future in some quality attributes
- *Sensitivity points:* alternatives for which even a slight change makes a significant difference in a quality attribute
- *Trade-offs*: decisions affecting more than one quality attribute

They can also use some other appropriate classification. The aim is to classify so that identification of trends is easy. A careful analysis will not only show the trend exhibited by the architecture but also the architectural capabilities of the architect, thereby providing chances for improvement.

These methods improve the usefulness of the basic scenario-based evaluation method. As we did, the method should be adapted to suit particular organizational needs and available resources.

Conclusions

Evaluating architecture and finding bottlenecks and suitability of the architectural components is similar to finding out logical errors and other defects in programs early. The earlier we find them, the lesser are the costs and efforts involved. Use a suitable method (a simple checklist, questionnaire, scenarios, prototyping, simulation, metrics based, etc.) to evaluate/review the architecture.

Software architecture evaluation methods expose the specific limitations of the architecture in a very simple, straightforward and cost-effective way, which otherwise surface late in development and deployment and lead to disappointment, rework and increased costs. The various methods help us evaluate architectural decisions in light of quality attribute requirements and reveal how the various quality attributes interact with each other. They highlight the relationship between quality attributes and business drivers (e.g., affordability and time to market) that motivate them. Though the methods are not inexpensive, the cost–benefit ratios are favourable. Software architecture evaluation methods give an insight into the capability and incapability of the system architecture, which are not easily visible otherwise. Hence, their use should be encouraged.

In this chapter, we saw the rationale behind the evaluation and tried to answer the Ws (why, when, who, what) of evaluation. We also saw the basics of software architecture evaluation without binding us to just one technique. But the most essential step of this learning exercise is to apply this knowledge. This case study provides us with an opportunity to get hands-on experience that is real and represents a modern software development project scenario.

It is important to note that identifying all critical quality attributes of the application has to start from the end user's point of view and how various sub-systems within the main system interact with each other. Identifying appropriate scenarios, thus, becomes very crucial to construct the utility tree of a given application.

In this case analysis, we have used only the scenario-based method, which makes use of more of intuitive approaches like brainstorming. It does not, however, offer any assurance on whether all the scenarios are accurately captured or whether the right utility tree is built at the end of the exercise. A more important limitation of this approach is that when we consider a few important scenarios, there could be several failure points that may not be obvious for the people who are sitting in the brainstorming room.

However, we are limited by the fact that as the system is still under development, we are not left with many choices. Had it been a production system where we have enough metrics gathered, we could have used more structured methods like measuring techniques that are more technical in nature. They usually involve quantitative methods like collecting quantitative answers to specific questions, collecting metrics and using simulation and prototypes.

Note that architecture evaluation is conducted at several stages in the life cycle of the development of a software. As more architectural descriptions are available we will be able to conduct evaluation in a better way. As the knowledge about the system increases as the development and ultimately its usage progresses, we will be able to apply more sophisticated evaluation techniques.

In any case, remember that 'reviewing and evaluating architecture early in the development cycle' is one of the most effective practices in engineering software systems.

Best Practices and Key Lessons from the Case Study

- Review the architecture if the project is of 700 staff days or longer. This recommendation is based on the survey conducted by AT&T. On an average it takes 70 staff days to conduct the evaluation and the benefits are about 10 per cent of the overall project cost.

- The most suitable time to review architecture is when the requirements have been established and a proposed architecture is established. As time passes in any development effort, decisions are made about the system under construction. With every decision, modifications to the architecture become more constrained and, thereby, more expensive. Performing the architecture evaluation as soon as possible reduces the number of constraints on the outcome of the evaluation.
- Try and ensure that all possible stakeholders participate in the evaluation exercise right from the client representative to developer, system end user, system administrator, tester, and so on.
- Provide read ahead material to the review team. This may include description of the architecture and discussion of the rationale behind architectural decisions.
- Rank the quality and function requirements of the system before the evaluation begins. This will in turn guide the quality attributes to be considered on priority.
- Reiterate the scenario-finding process using the base scenarios to come up with more and refined scenarios.
- Use use cases or business and functional requirements as the starting point to identify various scenarios or checkpoints to evaluate the architecture.
- The architectural drivers can form the first set of quality attributes to be analysed.
- Industrial experience suggests that three to five is a reasonable number of quality attributes to try to accommodate.
- It is important to check the existence and soundness of system acceptance criteria.
- Each issue raised during the review should be documented.
- When evaluating performance in terms of resource utilization, ensure that the *workload information* (consisting of the number of concurrent users, request arrival rates and performance goals for current scenario), a *software performance specification* (execution paths and programs, components to be executed, the probability of execution of each component, the number of repetitions of each software component and the protocol for contention resolution used by the software component) and *the environmental information* (system characteristics like configuration and device service rates, overhead information for 'lower level' services and scheduling policies) are all available.
- Do not miss early warning signals. These include the presence of more than 25 top-level architecture components, one requirement driving the rest of the design, architecture depending upon alternatives in the operating system, use of proprietary components though standard components are available, architecture forced to match the current organization, component definitions coming from hardware division, too much of complexity (e.g., two databases, two start-up routines, two error locking procedures), exception driven design, and so on.
- If any industry standards are to be fulfilled by the system, incorporate them in the list of most important requirements.

Further Reading

Among all the resources that are currently available, in opinion, the most comprehensive one is a book written by Paul Clements, Rick Kazman and Mark Klein, *Evaluating Software Architectures: Methods and Case Studies*, published by Addison Wesley, in 2002.

Software Engineering Institute (SEI) Web site of Carnegie Mellon University has a wealth of information in this area (www.sei.cmu.edu). In fact, SEI has its own philosophy and approach to architectural evaluation. Some interesting resources from this web site are the following:

- Barbacci, M. R., Ellison, R., Lattanze, A. J., Stafford, J. A., Weinstock, C. B., Wood, W. G. *Quality Attribute Workshops*, 2nd edition. Report CMU/SEI-2002-TR-019. Pittsburgh, Pennsylvania: Software Engineering Institute, June 2002. www.sei.cmu.edu/publications/documents/02.reports/02tr019.html.

- Abowd, G., Bass, L., Clements, P., Kazman, R., Northrop, L., and Zaremski, A. *Recommended Best Industrial Practice for Software Architecture Evaluation.* Pittsburgh, Pennsylvania: Software Engineering Institute, January 1997.
- Barbacci, M. R., Klein, M. H., Longstaff, T. A., and Weinstock, C. B. *Quality Attributes*. Report CMU/SEI-95-TR-021. Pittsburgh, Pennsylvania: Software Engineering Institute, December 1995. www.sei.cmu.edu/publications/documents/95.reports/95.tr.021.html
- Kazman, R., Len, B., Gregory A., and Webb, M. SAAM: A method for analysing the properties software architectures. In: *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, May 1994, pp. 81–90.

Most of the relevant material about SEI's own methodology on architecture evaluation, The Architecture Tradeoff Analysis Method (ATAM), can be found at www.sei.cmu.edu/architecture/ata_method.html.

A good overview of scenario-based evaluation is given in the paper Scenario-Based Software Architecture Evaluation Methods: An Overview, written by Mugurel T. Ionita, Dieter K. Hammer and Henk Obbink. This paper was published in the workshop on Methods and Techniques for Software Architecture Review and Assessment at the *International Conference on Software Engineering*, Orlando, Florida, USA, May 2002.

The Software Architecture Review and Assessment (SARA) Working Group published a popular report called the SARA. Version 1.0 of this report is a result of the work done between 1999 and 2002. Philippe Kruchten is the main person behind this effort. This report is very useful for anyone who wants to understand the architectural evaluation. It can be found at: www.philippe.kruchten.com/architecture/SARAv1.pdf (earlier the same report was available at www.rational.com/media/products/rup/sara_report.pdf).

We also recommend another good article on architectural quality characteristics by Francisca Losavio *et al.* Quality characteristics for software architecture. *Journal of Object Technology*, Vol 2, no 2, March–April 2003, pp. 133–150. www.jot.fm/issues/issue_2003_03/article2.

We have mentioned earlier the limitations of the classical brianstorming methods. There are some structured approaches such as TRIZ (Russian acronym for Theory of the Solution of Inventive Problems) and I-TRIZ (Ideation TRIZ) to help in this situation. Anticipatory failure determination (AFD) is a sub-area within I-TRIZ. TRIZ, I-TRIZ and AFD are the bodies of knowledge that help in approaching the innovative solutions to problems. In particular, AFD is a method to detect, analyse and remove failure cases in systems or products or processes. Readers interested in this topic can refer to the Web site www.ideationtriz.com/ as well as the white papers available on the same Web site. For example, Phadnis and Bhalla (2006) apply these techniques to digitization of businesses.

Chapter 5

Moving from Software Architecture to Software Design—Building a Mobile Trading System

Contributors
Bhudeb Chakravarti
Vamsi STP
Kirti Garg

In most projects, the software development process appears chaotic. The two major reasons for this are inaccurate understanding of the requirements and insufficient time spent on analysing how to build the system. We address the second problem in this chapter. While exploring the solution space for a given problem and understanding the details of the proposed solution, the team usually tries to enforce some structure into the way in which each of the problem areas are being addressed and terms it as 'design'. A good design helps make the rest of the development process less chaotic and requiring less number of iterations.

What is 'good software design' all about? How does one go about creating a detailed design from the architecture? These are important questions that one has to address after the architecture is finalized. Good design is a blueprint for the software being developed. The detailed design description documents include architectures that describe the top-level design, class diagrams that illustrate the abstractions and implementation that brings out the low-level design. However, in most projects, only the high-level design document is created. The auxiliary documentation is equally important to come up with a robust design. Software design is all about thinking what one needs to build, what tools or components one needs and how all of these fit together. The lack of usability and poor design of software are reasons for many critical project failures.

In this case study, we shall look at the design of the 'Mobile Trading System' (MTS)—a software application for mobile devices that helps its users manage their finances, including their stock market transactions, from any where and at any time. The case discusses various best practices used by the software development team to come up with the design. As very few applications are simple enough not to have to worry about a proper design methodology, it also discusses usage of design methodologies and tools for developing design artefacts. Most of the system's components are interdependent and need a robust design to manage complexity. Well-designed software may help create reusable components to leverage in future projects. This case study discusses in detail different UML diagrams (such as class, sequence, and use cases), helps designers map the user requirements to software design components and helps come up with a blueprint for the system to be developed.

BACKGROUND

What Is Design?

Design is the bridging activity between gathering and implementation of software requirements that satisfies the required needs. Software design has three distinct activities:

- External—Planning externally observable characteristics of a software product, for example, external data sources, functional requirements, and so on.
- Architectural design—Architectural design refines the conceptual view of the system, decomposing it into sub-systems.
- Detailed design—Detailed design specifies the data structures, functions and algorithms needed to implement the system.

The last two activities are referred to as internal design, as they deal with specifying the internal structure and processing details of the software product.

The fundamental goal of design is to reduce the number of dependencies between modules, thus reducing the complexity of the system. This is also known as coupling; lesser the coupling the better is the design. On the other hand, higher the binding between elements within a module (known as cohesion) the better is the design. Fundamental concepts of software design include abstraction (functional, data and control), information hiding, modularity (to enhance design clarity and ease of implementation), concurrency and design

aesthetics. Apart from these concepts, the following design guidelines help organize the design activities and come up with a good design:

- Review requirements specification
- Expand the external interfaces and other artefacts developed during requirements analysis
- Refine documents (UML diagrams) and data flow diagrams developed during requirements analysis
- Record functional and data abstractions, and define visible interfaces for each
- Define modularity criteria to define the structure of the system
- Verify if the resulting structure created using one of the design techniques satisfies the requirements
- Conduct a preliminary design review
- Create data representations and redesign as and when necessary

Design Notations

The major difficulties encountered in the design phase are due to insufficient requirements, providing too many details too early and failure to consider alternative design strategies. The use of systematic design notations improves the ability to conceive, communicate and verify design. A design model preserves the structure of the system imposed by the analysis model and includes elements such as sub-systems, dependencies, classes, use case realizations, and so on. Each model must describe a specific aspect of the system under consideration. It also gives us the flexibility to fail under controlled conditions.

Visual modelling gives us an understanding of the requirements, constraints and other issues by using a common notation. Also, the interfaces between sub-systems can be identified early in the life cycle.

The different artefacts of software design include design model (hierarchy of sub-systems), design class, use case realization along with class and interaction diagrams, architectural view of the design model, deployment model (nodes and connections, mapping of classes to nodes), and so on.

In software design, representation schemes such as data flow diagrams, structure charts and pseudo-code help represent the design of the system.

In response to the growing complexity of software systems, many design methods have evolved. Broadly, these methods can be classified as follows (Sommerville, 2004):

- Top–down structured design
- Data-driven design
- Object-oriented design (OOD)

The earliest and one of the most influential techniques was the top–down design, which encouraged structural programming. The fundamental unit of decomposition in this approach is to divide a complex problem into sub-problems. Unfortunately, as the complexity of software increases the structural design fails to scale up as it does not address issues of data abstraction, information hiding and concurrency. In the data-driven approach, the system is mapped into inputs and outputs. This method fails in the case of time-critical event systems and is not suitable for object-based and object-oriented programming languages. OOD emphasizes on modelling the system as a collection of cooperating objects that are instances of classes within a class hierarchy.

Design can be defined as a disciplined approach to invent the solution of a problem, and software design helps come up with a blueprint of the system keeping in view the constraints on the system.

There is no silver bullet to come up with a good software design. The design of a complex system is often an incremental and iterative process. According to Grady Booch (2004):

OOD is a method of design encompassing the process of decomposition and a notation to depict the logical, physical as well as the static and dynamic models of a system.

The real purpose of a design is to create an architecture (including class and object structure) for implementation and common policies used by different elements of the system. The activities associated with design include the following (Booch, 2004):

- Architecture planning—Logical and physical decomposition of the system
- Tactical planning—Making decisions about common policies
- Release planning—Helps create a development plan for the system

Case Study: Mobile Trading System

It was well past office hours when David Smith stepped out of his office. He made sure that all the items in his checklist were taken care as he was leaving for a long-awaited vacation the next morning. Every winter David planned a trip to Hawaii to enjoy the sunny beaches far away from the cold winter back home at Detroit. He had planned his trip meticulously. The reservations have been made and the whole trip was planned almost six months earlier. He eagerly waited to take off the next day.

He now had some last-minute shopping to do before getting home. That is when the bad news came in. His stock broker had called him to inform that the stock market had run into some turmoil and the stock value of the shares that David had invested in were likely to touch an all-time low. David's heart sank. He wanted to know what options he had now. The stockbroker informed him that he will have to sell the stocks at the earliest. The only way to prevent a huge loss was to sell the stock completely and purchase new stocks of a company whose value is likely to increase.

David had to act fast; he now had to plan his finances without ruining the much-awaited vacation. He took out his PDA and on his way to the supermarket browsed the Internet to confirm if the stock market was truly in a state as reported by his broker. After verifying the stock prices, David connected to his bank account. He quickly reviewed his financial status before he could trade the stocks. Satisfied with the information he obtained and his analysis, he decided to sell his shares. He connected to the online trading system through his PDA and requested the transactions he wanted to carry out. He transferred money from his bank account to his trading account.

The next day David and his family took the flight to Hawaii and reached the resort where he had already made reservation. After sometime he took out his PDA to check his account and observed that all his transactions had gone through. He was a happy man now and was glad to have received updates from the stock market about the increasing prices of his new shares and felt lucky when he heard that the price of his old shares had gone down—he was just saved from a disaster!

'Thank you MTS', David said.

Fact-Tree developed the application that David had relied on to bail him out of trouble. This was a revolutionary product developed for the Stock Exchange Board (SEB) and helped Fact-Tree gain a lot of visibility in the finance domain. The product was well acclaimed among stock traders and agents. It changed the way trading got done. The customers and end users were very happy. It was one of the few software projects delivered within time and budget constraints, meeting the needs of all the stakeholders. The Fact-Tree team attributes the resounding success of its product to the robust design of the MTS. Kiran, one of the developers, had said the following:

Glossary

Online Trading

Buying and selling financial securities (stocks, bonds, options, futures) and currencies using the Internet or broker-provided Web-based proprietary software.

Wireless Trading or Mobile Trading

A special case of online trading where customers can trade via cell phones, PDAs, pagers and other hand-held devices.

A good design is work half done. Thanks to the robust design and well-documented artefacts, coding was smooth and without any glitches or ambiguities.

The Project

Last year the SEB had sent an RFP to software companies to develop an application to solve the problem of delay in customer service. They often got complaints from customers that timely information was not available or service was not fast enough. To fix these problems and provide better and faster services, the SEB wanted a system in place. After fierce bidding, Fact-Tree got the project and christened it as 'Mobile Trading System'.

The project was to develop an integrated solution designed specifically for the SEB. The primary objective was to address an important and compelling need of investors on the move, that is, making timely investment and trading decisions using their handheld devices. The system required that a link with globally reputed content providers, who are partners of the SEB, be made available. This was to provide 'Anytime, Anywhere' access to customers. The application had to provide comprehensive market information, analytical tools and recommendations—all in real time. The challenge was to integrate the ability to conduct transactions over the exchange in a secure manner through handheld devices, such as PDAs and WAP phones.

It was made clear by the client that the system had to incorporate a host of features to provide investors powerful decision-making and trading tools on their handheld devices. The analytical tools were to be designed intuitively to provide investors with easy and configurable tools for performing technical and fundamental analysis. Another requirement was that based on individual preferences investors should be able to personalize the information collected from the content providers on their handheld devices.

The application dealt with sensitive information, so it was important to enable secure trading over handheld devices. To meet this goal, end-to-end security based on PKI/WPKI concepts were to be incorporated into the solution. This was to instil confidence in all the stakeholders that

- Transactions cannot be fraudulently generated or altered
- Transactions are legally binding
- Confidentiality of private information is adequately protected

Glossary

Public Key Infrastructure

Public key infrastructure (PKI) is an arrangement that binds public keys with respective user identities by means of a certificate authority.

Wireless Public Key Infrastructure

Wireless public key infrastructure (WPKI) is a two-factor authentication scheme using mainly the mobile phone and a laptop.

The Team

Projects of this kind were not new to Fact-Tree. They had developed PDA- and Palmtop-based projects in the past and were prepared to take up this challenge. A team of eight experienced developers was put together under the leadership of Aparna, the project manager. Others in the team included one senior architect, Kishore, whose responsibility was to define the architecture of the system and maintain it throughout the development life cycle. A small team of two was brought together for testing.

Aparna was confident that the project could be completed with satisfying results. She understood the nuances of applications that ran on the mobile platform. One of her projects 'Medico' was designed to run on mobile devices. This project had won her team many accolades. The central idea of Medico was to develop a PDA-based application for a major pharmaceutical company. The application primarily was developed to help the medical representatives of the company access prices and stock details of the medicines while on the move. The application also allowed them to update a central database from any remote location, based on orders booked and sales done by the end of the day.

Aparna prepared the project plan and estimated the resources required. Fact-Tree proposed a nine-month timeframe for the development of the system. The project plan included two weeks of requirements understanding and knowledge transfer, four weeks for design, four months for coding and rest of the time was allocated for testing and deployment. SEB was comfortable with the project plan and the project was commenced.

The team embarked on the project with high spirits. It was decided that they should go for the waterfall development model using use case methodology. The waterfall model was chosen because the terrain was known and the requirements were clear. Thus, in the given scenario the waterfall method seemed to be an ideal choice. The use case method, on the other hand, is a widely used technique and was a natural choice.

Requirements Understanding and Knowledge Transfer

This phase was used to understand the requirements of the system. A four-member team led by Gopi was formed to gather requirements. The requirements were discussed with the client in detail. Since the project used the waterfall model, it was important to get a clear understanding of the requirements in the very beginning. Gopi was quite pleased with the way things were shaping up. The requirements were getting clearer. He identified that the overall purpose of the system was to

- Increase customer satisfaction
  - Providing value-added services such as 'anytime, anywhere' access to information and transaction capability
  - Increasing the number of clients
  - Increase the number of transactions executed per client
  - Facilitating faster transactions and providing timely alerts and news updates
  - Reduce transaction costs
  - Since many of the clients perform trading through handheld devices electronically rather than by phone

The team interviewed expected users of the system and used various techniques to gather the requirements. They went through the requirements to prioritize and identify the salient features that the system should offer to obtain maximum user satisfaction. After this the requirements were converted into the use cases. Rekha, one of the members in the requirements team, had expertise in use case development. She analysed the requirements to identify the actors that were interacting with the system. She discussed this with the team and came up with different usage scenarios that described how the system was likely to be used by the actors. These usage scenarios reflected the primary requirements of the system.

Some of the use cases that the team had come up with are listed below.

- Users should be able to retrieve information from content providers and store it in their handheld devices.
- Users should be able to subscribe and access services on demand and do online transactions with minimum overhead in terms of time and processes.
- Users should be allowed to access information or perform transactions only after proper authentication.
- Users should be able to personalize their profile in their handheld devices to get only relevant information and not irrelevant information.
- The system administrator should be able to add, modify and delete any user of the system and provide authorization to newly registered users.

- The system should allow the management to keep track of the usage of the system by the user and create bills based on the usage of facilities.

Based on use cases developed, the requirements team created the required use case diagrams (Figure 5.1).

They identified three actors interacting with the system: 'subscriber', the user who will subscribe to their system to avail the facility the system provides; 'system administrator', the administrator who will create and manage the user details once a client subscribes to the system, and 'manager', one who will create bills for the user depending on the use of the system.

The use cases identified are the following:

- *Login*: The authentication system to allow users to use the system.
- *Retrieve information and store*: Depicts how subscribers could retrieve information from the content server and store it in their handheld devices.
- *Perform online transaction*: Depicts how subscribers could connect to the online trading system and perform transactions.
- *Personalize profile*: Depicts how subscribers could personalize their profiles.
- *Manage users*: Activities to be performed by the system administrator to manage users in the system.
- *Generate bills*: Activities to be performed by the manager to keep track of the use of the system by subscribers and generate bills.

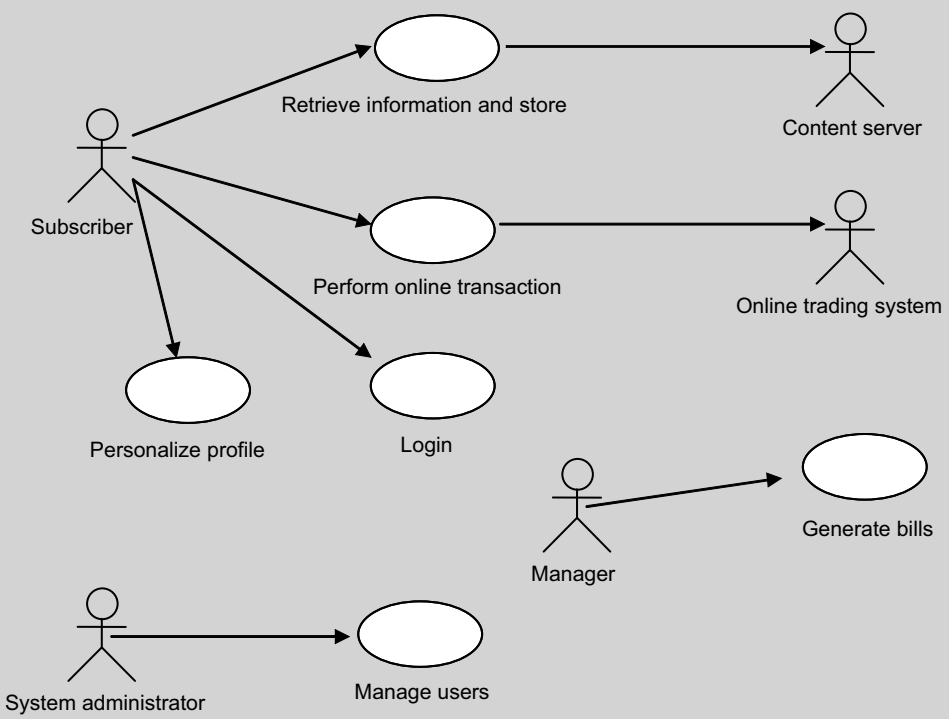


Figure 5.1 Use case diagram of Mobile Trading System

While describing the use cases through flow of events, it was observed that two external systems were required for the use cases to perform effectively. These two systems were 'content server' and 'online trading system'. They were also considered as actors in the system and were depicted in the use case diagram.

The team was quite happy with the way things were progressing. They used Rational Rose case tool to document the use case analysis. Rekha was excited about the whole exercise and commented the following:

> Of all the times I've done use case modelling this was one of the most exciting. The task was challenging but the team was bent upon finishing it on time. I thoroughly enjoyed all those long discussions running into late hours!

Once the actors, use cases and the use case diagram were decided, the next activity was to identify the flows of events for each use case. Gopi and his team interviewed users and had long discussions with client representatives to finalize the flow of events for each use case. They created the use case specification document for each use case that was identified in the system. These documents were also included in the user requirements documents and delivered to the client. The client reviewed the documents and was very happy with the contents submitted by the development team (see Exhibit A). They approved the same without any major changes. The requirement phase was declared as completed and signed off to move ahead with the next phase.

Exhibit A Use case specification document (partial) for a few use cases as delivered to the client

Use case ID: UC1	**Use case name: Retrieve information and store**
Description	This use case retrieves financial data from the content providers' network using Content Interface and stores the data at the Content database. This use case is a continuous one and starts at the time of the start up of the system and continues till the system is terminated or an interrupt is used to terminate the process.
Actors	Subscriber
Pre-conditions	NA
Post-conditions	NA
Frequency of use	NA
Normal course of events	The subscriber initiates the use case when he/she wants to retrieve information from the content provider. The normal course of events for this use case is as follows: 1. The subscriber initiates the retrieval process 2. The system initiates the Information Retrieval Controller (IRC) 3. IRC connects to the content provider through Content Interface and gets confirmation 4. The system connects to the content database and gets confirmation

(Continued)

Use case ID: UC1	Use case name: Retrieve information and store
	5. IRC gets the query parameters from the subscriber and creates statement 6. IRC retrieves data from content server in specified format 7. IRC converts the format of the data to a defined format 8. IRC stores the data in a handheld device
Alternative courses	**In** step 6, if IRC finds that connectivity to the database is lost, then it connects the system to the content database and gets confirmation. After completion of this event, it goes back to the normal event, that is, step 7.
Associated use case diagram	The use case diagram is enclosed
Exceptions	If the system fails to execute the use case for any reason, it should pop-up a message stating the same. In that case, the use case should be initiated again.

Use case ID: UC2	Use case name: Login
Description	This use case allows the subscriber to log into the system and use the system after verification. The subscriber cannot initiate any other use case till he has not successfully completed this use case.
Actors	Subscriber
Pre-conditions	NA
Post-conditions	NA
Frequency of use	Whenever subscriber starts to use the system
Normal course of events	When the subscriber initiates the client software installed on his handheld device, the use case starts. The normal course of events is as follows: 1. The system displays the login screen 2. The subscriber enters the username and password and clicks submit 3. The authentication controller verifies the subscriber details 4. The main menu is displayed at the subscriber's device

(Continued)

Use case ID: UC2	Use case name: Login
Alternative courses	**In** step 4, if the authentication server finds the information sent by the subscriber is not valid, it performs the following: 1. The system sends a message that the subscriber cannot be logged into the system 2. The message is displayed on the subscriber's handheld device along with the login screen
Associated use case diagram	The use case diagram is enclosed.

Use case ID: UC3	Use case name: Personalize Profile
Description	This use case allows the subscriber to set his preferences on personalized settings so that only the relevant information is displayed on his handheld device.
Actors	Subscriber
Pre-conditions	NA
Post-conditions	NA
Frequency of use:	Whenever the subscriber wants to view information.
Normal course of events	The normal course of events is as follows: 1. The subscriber logs in to the Web site. The system displays the 'Personalize Screen' 2. The subscriber selects the 'Subscribe to Service' option 3. The system displays the 'Subscription Screen' with the subscribed services and available services. The subscriber can add or remove any of the services 4. The system modifies the user information 5. The subscriber selects the 'Customize Services' option for the selected service. The system displays the customization options available for that service. The subscriber submits modification. The system updates user information 6. The subscriber selects the 'Change Password' option. The system displays the 'Change Password' screen. The subscriber submits the modification. The system updates user information
Alternative courses	NA
Associated use case diagram	The use case diagram is enclosed.

The Design Phase

The design phase of the project started with much enthusiasm. The team knew that a good design was important to improve the quality of service and would ease the development efforts and time. So under the guidance of Kishore, the architect, the team explored different ways to design the system. Since the underlying idea was to come up with a strong design, it was decided that some object-oriented analysis and design would be followed.

The first activity the team did was to develop the overall system block diagram (Figure 5.2).

Once the system block diagram was defined, the team prepared a use case specification document detailing all the use cases, actors, pre- and post-conditions, alternative courses, and so on. After identifying the appropriate usage scenarios, the team came up with a high-level design for the system keeping in mind that the system had to be highly scalable and extendable to other handheld devices with minimum cost extension in the future.

The next activity the development team did was to draw the sequence diagram. Kishore and Rekha were experienced in object-oriented analysis and design and had the knowledge of identifying boundary, entity and control classes from the use case flow of events that would be required to draw the sequence diagram.

All the sequence diagrams were created and drawn using Rational Rose. By now the team had got a clear understanding of the scope of the system from the use case diagram and the interactions between different entities in the sequence diagram. Now it was time to map these real-world objects to software, to group them into classes. These classes could be represented during the coding phase using any of the chosen languages for development. So the team now got busy in designing the static structural model of the system and identifying the relationships between the classes.

Glossary

Entity, Control and Boundary Classes

Entity classes: Model actual entities in the real world

Control classes: Model conceptual sets of functions that operate on the entities

Boundary classes: Model the interfaces that the system provides to access these operations

The team identified the boundary classes, control classes and entity classes. Some of the team members were new to this mode of design, but as Kiran pointed out later:

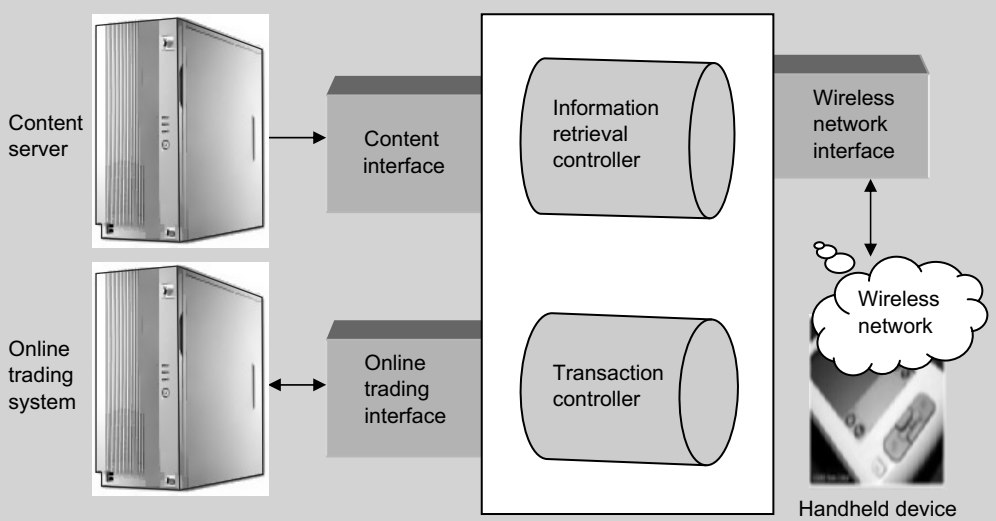


Figure 5.2 Block diagram of the mobile trading platform

Coming up with classes was a tricky job but not entirely new. In many ways class diagrams are similar to entity relationship diagrams, a technique that we have used in many other projects during design.

Aparna reminded the team that MTS was to be designed as a generalized, user friendly and click and view system that provides real-time financial market data on PDAs or other handheld devices. So the team needed to specifically design the software for the investment community whose members will want to retrieve real-time financial data, analyse them and transact them with high security while on the move. This became one of the major design goals. In another team discussion, the team came up with a valid consideration that went on to become another important design objective to be met to ensure customer satisfaction. The idea was to customize the presentation for easy viewing and help in quick decision making. Several other important design goals surfaced after a series of discussions and extremely productive brainstorming sessions. Gopi rightly pointed out the following in one such discussion:

One of the objectives for the system must be that the application should be highly scalable and must be extendable to other handheld devices with minimal cost extension. This is keeping in mind the influx of new WAP-enabled and wireless gadgets that keep flooding the market periodically.

It was important to make the design flexible enough so that it could be extended in the future to accommodate other devices. This idea went on to become one of the salient features of the system. The team members threw in all the ideas they had during the discussions. No idea was considered silly unless it was unspoken. This open environment encouraged a strong set of design goals to be formulated. With these design goals in mind, Kishore discussed with the team and came up with the complete design of the system (see Exhibit B).

Aparna was pleased with the outcome of the high-level design phase. She was convinced based on her experience that this robust design was the stepping stone for the success of her project. This marked the end of the high-level design phase. The seed was sown for a very successful system to take shape.

Exhibit B Sequence diagram and top-level class diagram of the Login use case (Figure 5.3-5.5)

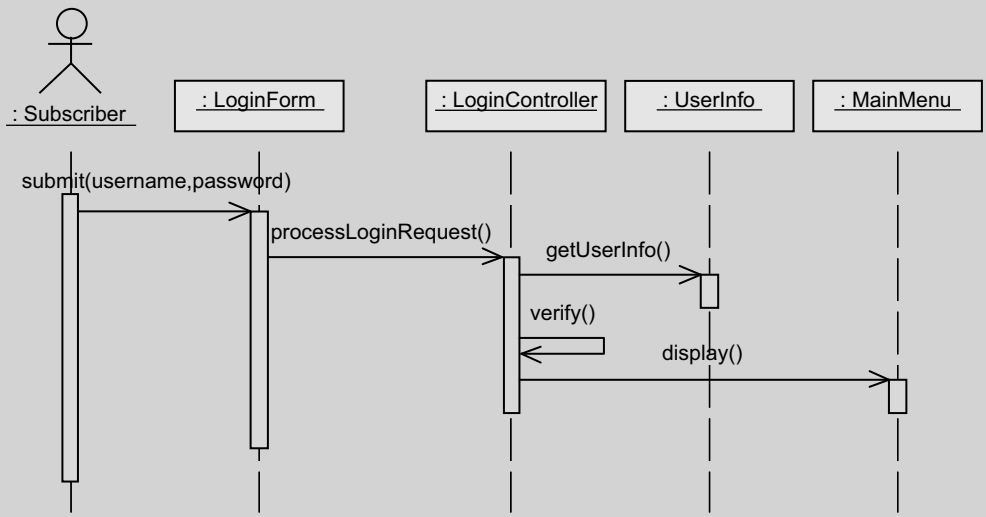


Figure 5.3 Sequence diagram of Login use case (basic flow)

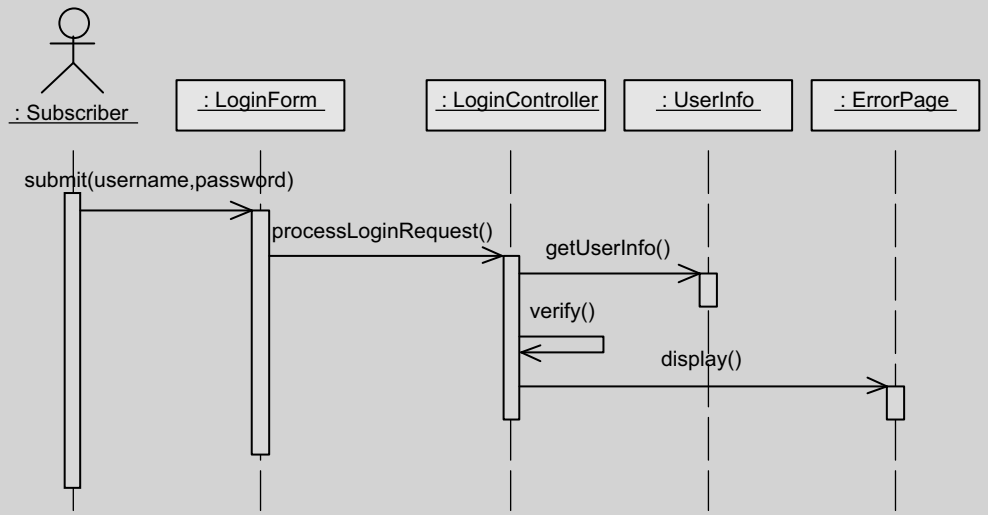


Figure 5.4 Sequence diagram of Login use case (alternative flow—login failed)

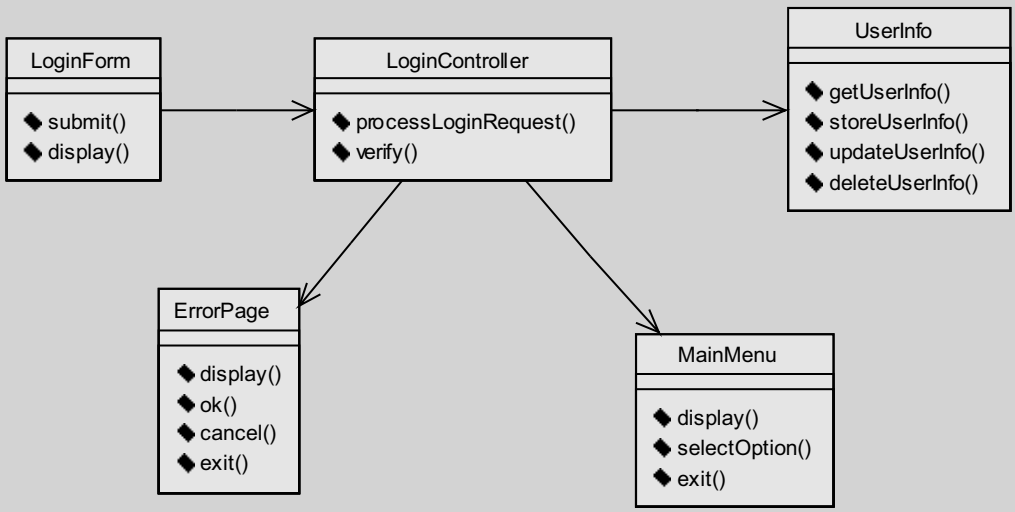


Figure 5.5 Class diagram (VOPC) of Login use case

Solve this Case

1. Based on the case given above, can you now come up with your own design for the given system?
2. Create the complete use case Specification document for all the requirements.
3. Draw the sequence diagrams for the remaining use cases.

4. Draw the class diagrams (VOPC) from the sequence diagrams that you have drawn in the previous question.
5. Draw a package diagram for the class diagrams you have drawn.
6. Identify the components that can be considered for this case.
7. Group the classes for each component.

POSTMORTEM

The case study discusses at length several design issues that came up during the development of the MTS. For example, the application had to run on handheld devices providing 'anytime, anywhere' access to users. Also, the application had to provide comprehensive market information, analytical tools and recommendations. The team moved carefully from architecture to detailed design considering several issues such as ability to conduct secure transactions over all the exchanges through different types of handheld devices. Thus, the requirements were interesting and are representative of the new genre of requirements, as seen in this age of information and communication. Under such conditions, it is preferred that the team moves forward in a guided manner, from requirements to architecture, and create a detailed design from architecture. The MTS team was careful to use standard methodologies such as the use case methodology to come up with the design and its artefacts. This was a good move as it provided a common language to communicate design decisions and system requirements.

But is the process of design as easy as its description? What are the issues that become prominent, at times? How do we move from architecture to detailed design? What are the tools and techniques available to a designer on this journey? Can there be a well-defined process that, when followed, ensures a 'good' or a 'perfect' design? We will try to seek answers to these questions.

The Design Process

Developing a solution is a challenging task. The software designer needs to acquire a degree of domain knowledge to undertake the design task.

There are two phases in the process of designing. In the first phase, the software designer decomposes the solution into a set of large and complex software components that interact with each other. This is called *high-level design*. The details of the components and the interactions are at a very abstract level. In the second phase, each of these high-level components is further detailed with their attributes and methods. The interactions between these components and sub-components are also specified. At the end of second phase, the so-called *low-level design* is ready. During the low-level design phase, it is possible to identify all the objects (either new or existing) that can be reused and group them into various object libraries.

A simple design process could be followed to make life easier for the designers. According to David Budgen (2004), a usual design process consists of the following steps:

- Postulate a solution clarifying the nature of requirements.
- Build a model of the solution: Construct both black box and white box models of the problem. The black box model describes the external functionality and behaviour of the system as a whole, without any reference to how this is to be achieved. On the other hand, a white box model describes the internal workings of a system.

- Evaluate the model against the original requirement: Validate the solution including use of prototypes.
- Elaborate the model to produce a detailed specification of the solution: Use suitable forms of software for implementation of the design plan.

Design models allow the designers to explore the potential limitations of the solution, its behaviour and its structure. A design method can be regarded as providing a procedural description of how to go about the task of producing a design solution for a given problem. There are several software design methods that support building initial abstract models to explore ideas. The major components of such design methods are representation, process and heuristics.

Due to the abstract nature of software, identifying and specifying all the needs of the system can be problematic. As a result, the solution space may be ill-defined and give raise to the following problems:

- Design is not self-consistent
- Design and specifications are not consistent
- Design and requirements are not consistent

These inconsistencies must be identified and dealt with early in the design phase. The design should be flexible to accommodate important changes in design decisions and plan for reuse. Also, the design stage is the ideal time to plan for maintenance of the system.

Communicating the design decisions is an important activity of the design process. All the design decisions must be recorded and transferred to the team.

Design notations come to aid here. Many standard notations for representing design are available, with UML being the most prominent.

Moving from Architecture to Design

The process of designing the architecture typically starts with formulating problems that need to be solved and ends with a conceptual solution. The design process starts with this high-level solution and fills all the details that are required before constructing the software system. The architectural design is concerned with the overall form of the solution to be adopted, such as what the major elements are and how they interact. Detailed design, on the other hand, is concerned with developing descriptions of the elements identified in the architectural design phase and the interactions between these elements. To move to detailed design, we need to map the architectural design decisions to the logical design. After the candidate architecture is defined, we need to analyse the behaviour of the system and design the components and database accordingly. The idea is to move from architectural model to detailed design. The architectural model is used to describe the intended form that the system will take and is very abstract in nature, while the design model gives a concrete shape to the elements of the architectural model.

Theoretically, one should not tamper with business objects unless absolutely necessary in order to maintain their basic properties and behaviour.

To move from architecture to detailed design we need to identify design elements and their relationships. This process includes the following (Budgen, 2004):

- Defining system context
- Identifying the modules
- Describing the components and connectors

Step 1: Defining System Context

Defining the system from an external perspective helps establish the overall purpose of the system. It is related to understanding the problem domain that the system addresses. This utilizes the abstraction design technique and allows focusing on a particular level of detail without having to focus on all aspects of the system. The input for this step is the initial requirements list. The external behaviour of the system is mapped to the interfaces of the system. Each interface represents some coherent sub-set of system functions. A use case diagram is a common way to depict system context. An initial model of the enterprise context of the system can be created by diagramming the existing business processes, artefacts, people and existing systems.

Step 2: Identifying the Modules

In this phase, we identify discrete units that contain component definitions. This step also utilizes the abstraction design technique and applies design operators to split the system into module types. These design operators include the following:

- *Decomposition*: Separating functionality into distinct components that have well-defined interfaces, that is, separating the system into sub-systems. It is used to achieve a variety of quality attributes. Two types of decomposition are part/whole and generalization/specialization.
- *Replication*: Also known as redundancy, replication is the operation of duplicating a component to enhance its reliability and performance. Redundancy (identical copies of a component) and N-version programming (several different implementations of same functionality) are two types of runtime replication.
- *Compression*: It is the opposite of decomposition. It involves merging of components into a single component or removing layers or interfaces between components. It involves coupling and improves performance by removing indirection.
- *Abstraction*: It hides the implementation details. This makes components portable with respect to the component that has been abstracted and can be used to improve adaptability.
- *Resource sharing*: This refers to encapsulating data or services in order to share them among multiple independent components. This is useful when a resource is scarce and enhances portability and modifiability. For example, repository of user information can be a shared resource.

Step 3: Describing the Components and Connectors

The last step involves creating descriptions of instances of module types in runtime configurations. Identification of modules and components is the central activity of software architecture design. Many of the quality attributes are embodied in the components and their connectors. Components typically refer to runtime instances of software units. Connectors can refer to software components or mechanisms of communication. Some examples of connectors are user interface design and common data representation, where two applications are connected by shared data that are stored in a file or database.

The rational workflow for design can be sought as an example. It gives a view of how to move to design from the use cases:

- Use *case analysis*: Coming up with possible scenarios.
- *Use case design*: Identifying actors and use cases. Establishing the functionality of the prospective product.
- *Sub-system design*: A general understanding of how the individual parts work together to provide the desired functionality.

- *Class design*: Determine necessary classes and their data. Also discover inter-relationships among classes.

If we consider the OOD or the use case method of designing, then the design process will take the following course of action as per the steps given above.

Guided by the architectural decisions, a detailed design effort takes place. It addresses details such as the specification of all classes, including the necessary implementation attributes, their detailed interfaces and pseudo-code or plain text descriptions of the operation. The specification should be detailed enough that, together with model diagrams, it provides all the necessary coding information. In many automated software production processes, one can generate code skeletons from object-oriented diagrams.

Guided by architectural specifications, the design technically expands and adapts the analysis result. Further, detailed use case specification documents detailing all the use cases, actors, pre- and post-conditions and alternative courses can be prepared. The next step would be to identify boundary, entity and control classes from the use case flow of events in order to come up with the sequence diagrams.

The next logical step is to come up with the class diagrams that describe the static structure of the system, including the object classes, their internal structures and the relationships in which they participate. This is followed by object design, which is the conceptual representation of an implementation of the solution to a business problem. The object design, typically represented with a class diagram, is technology dependent and specific to architecture, computer language, screen and report layouts and other real-world restrictions. It is during this stage in the life cycle that the implementation of each class, association, attribute and operation is determined.

Thus, OOD builds on the products developed by refining candidate objects into classes, defining message protocols for all objects, defining data structures and procedures and mapping these into an object-oriented programming language. OOD groups these classes into packages. A package is a general purpose mechanism for organizing elements into groups. Packages help organize the model under development and are a unit of configuration management.

In order to produce a good design, it becomes important to assess the quality of both static and dynamic attributes of the system. The design attributes include simplicity, modularity (coupling and cohesion), information hiding, and so on. These design attributes provide a set of measurable characteristics of the entities and provide mapping between the abstract idea and the identifiable features of the actual design.

No discussion on design is complete without a mention of design patterns. A pattern describes a recurring problem, the core of the solution to the problem that can be reused many times. Design patterns provide templates and encourage reuse. Examples of some design patterns include proxy and chain of responsibility. There are design anti-patterns too that describe the pitfalls to avoid by capturing the unsuccessful attempts at design solutions (Jacobson *et al*., 2002).

Characteristics of a Good Design

To improve the quality of the design process, it is important to provide inputs to design activities. Technical and managerial review and prototyping can be used to provide these inputs. Design reviews and inspections can be effective ways to asses the design quality. Parnas and Weiss (1985) identify eight requirements of a 'good design':

- *Well structured*: Consistent with chosen properties
- *Simple*: To the extent of 'as simple as possible, but no simpler'
- *Efficient*: Providing functions that can be computed using existing resources
- *Adequate*: Meeting the stated requirements
- *Flexible*: Able to accommodate likely changes in the requirements

- *Practical*: Module interfaces should provide the required facilities
- *Ability to implement*: Using current and available hardware and software technology
- *Standardized*: Using well-defined and familiar notation for any documentation

CASE ANALYSIS

A detailed analysis of the case shows that the team worked in a rather systematic manner to move from requirements to architecture and then to design.

The architecture for the MTS is shown in Figure 5.6. The content provider and bank have a server that provides an interface to aggregate content from leading global content providers. It

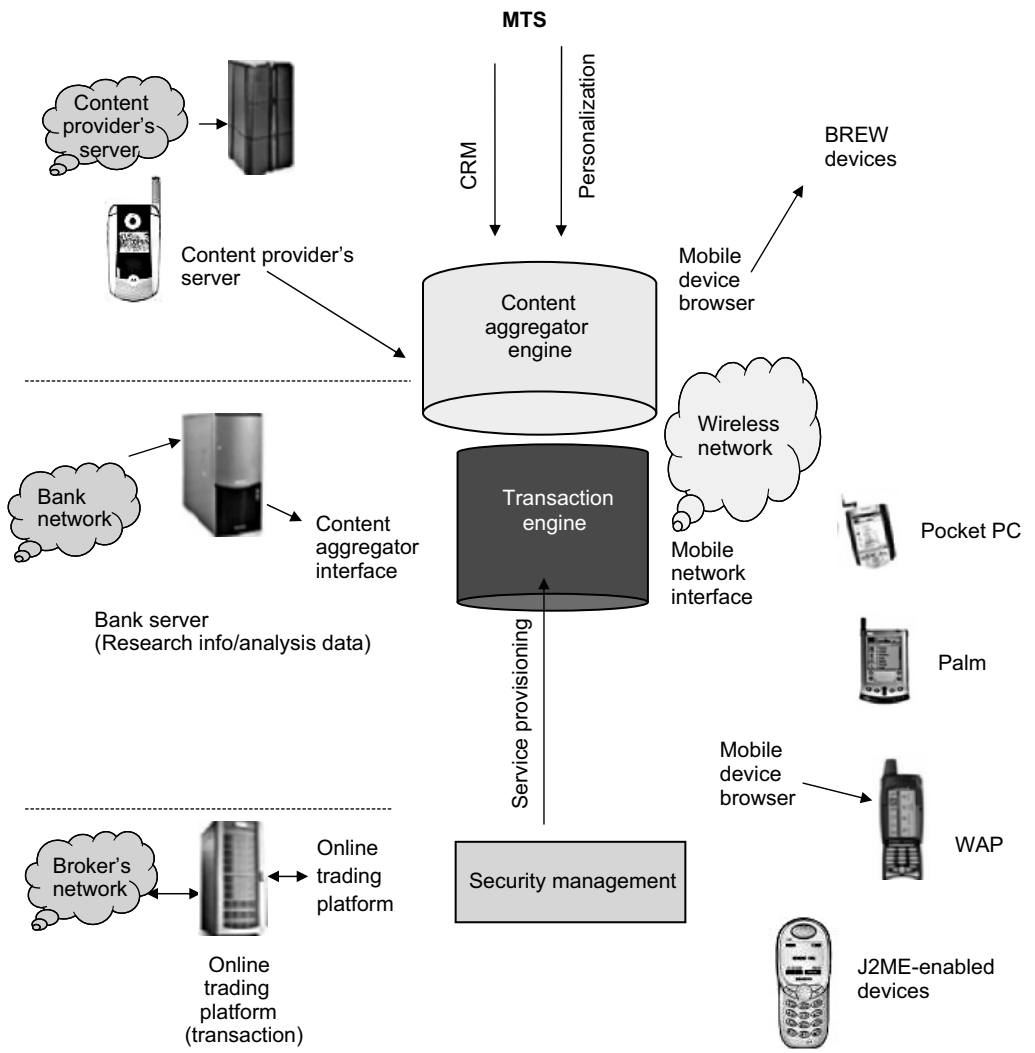


Figure 5.6 Mobile trading platform

has open interfaces that provide interfaces with legacy databases of various enterprises. On the broker's network resides the online trading platform interface (OTPI), which connects the system to the online trading platform used by securities companies. Between this and the client devices there is a security management system that protects the services and transactions that are being provided and monitors alerts raised, administration services and content aggregation. Over this there is a mobile network interface that helps the system communicate through the mobile network. This interface is available for different mobile networks such as GSM, GPRS and CDMA. Mobile device browser is the presentation software on the handheld device and provides the click screens to the user.

The team moved in a structured manner to come up with the design. Object-oriented analysis and design techniques helped model the system as a set of cooperating and individual objects as instances of a class. The job included the identification of classes and objects, identification of their semantics, identification of their relationships and specifying class and object interfaces and implementation.

As the first step, the team captured the usage scenarios of the system. This identification of the major use cases and external systems helped the team understand how the system is being used, its interactions and system requirements. The detailed use case specifications explained the system flow very well and gave a deeper understanding of the usage scenarios. This proper and thorough understanding of the requirements is the *first* step towards a good design.

The major use cases that were identified include the following:

- Login
- Retrieve information and store
- Perform online transaction
- Personalize profile
- Manage users
- Generate bills

External systems were also identified during this process. These included online trading platform, content provider's server and the bank's server.

Use Case Specification

During the detailed design we produce detailed artefacts that specify what the system is going to be like, which are captured in a use case specification. The team developed such a specification for the mobile trading platform. It gives details about all the use cases in the system and describes the usage scenario, the entities using the functionality of the use case to interact with the system, the pre- and post-conditions, normal and alternative flow of events, associated use case and sequence diagrams, exceptions, assumptions, and so on.

For example, the 'generate alert and sound' use case describes what happens when alerts are generated, who can generate them and when. Also, the use case specification gives details about the normal flow and alternative flow when alerts are generated.

Use case ID: UC2	Use case name: Generate alert and send
Description	This use case checks the user database if any alert is requested by the subscriber. If there is any alert sent by the subscriber, it generates a message as per the subscriber's request and sends the message to the subscriber.
Actors	System
Pre-conditions	The system is already connected to the user database during the start up.
Post-conditions	NA
Frequency of use	NA
Normal course of events	When the system is switched on, the system initiates this use case. The normal course of events is as follows: 1. The system gets the time from the system clock 2. The system checks whether any subscriber has set any alert 3. It reads the alert and gets the content information from the content database 4. System checks the alert condition and creates a message as per the request from the subscriber 5. It sends the message to the subscriber through the mobile network interface 6. It repeats steps 3-5 till all the alerts are read and taken care of
Alternative courses	In step 4, if the alert condition is not satisfied, system skips steps 4 and 5.
Associated use case diagrams/ sequence diagrams/ collaboration diagrams	The use case diagram is created in a Rose model called 'MTS.mdl'. The sequence diagrams are created under the package 'Logical view\Use case realization' within the use case realization called 'Generate alert'. The sequence diagram is shown in the annexure.
Exceptions	If the system fails to execute the use case for any reason, it should pop-up a message stating the same. In that case, the use case should be initiated again
Includes	NA
Special requirements	NA
Assumptions	NA
Notes and issues	NA

The 'Get information' use case explains how the subscriber can use the system to retrieve information and what the flow of events is when such a request is made.

Use case ID: UC4	**Use case name: Get information**
Description	This use case retrieves financial data, research data, analysis information, recommendations, and so on from the content database and displays them on the handheld device of the subscriber.
Actors	Subscriber
Pre-conditions	Login use case should be completed successfully by the subscriber before using this use case.
Post-conditions	NA
Frequency of use	NA
Normal course of events	When the subscriber has logged in, he can initiate this use case. The normal course of events is as follows: 1. The subscriber selects his choice on the main menu 2. The mobile device browser (MDB) interprets the request and sends it to the system through the mobile network interface. 3. The content aggregation engine (CAE) retrieves the data as per the request from the customer 4. CAE sends the information to the subscriber's device 5. CAE sends the information to the call data record (CDR) controller to create and store a CDR 6. MDB displays the information on the handheld device
Alternative courses	NA
Associated use case diagrams/ sequence diagrams/ collaboration diagrams	The use case diagram is created in a Rose model called 'MTS.mdl'. The sequence diagrams are created under the package 'Logical view\Use case realization' within the use case realization called 'Get information'. The sequence diagram is shown in the Annexure.
Exceptions	NA
Includes	NA
Special requirements	NA
Assumptions	NA
Notes and issues	NA

The following use case specification gives details of how transactions are performed via the online trading interface. Again the preconditions and basic and alternative flow of events are described, giving an understanding of what course of action the system should take.

Use case ID: UC6	**Use case name: Transact financial entity**
Description	This use case allows the subscriber to perform any transaction through the OTPI of MET. The OTPI is connected to the online trading platform of the enterprise, which in turn allows the user the platform to do secure transactions.
Actors	Subscriber
Pre-conditions	Subscriber should complete the 'Login' use case before using this use case.
Post-conditions	NA
Frequency of use	Subscriber can use this case whenever he wants to perform a transaction.
Normal course of events	The normal course of events is as follows: 1. The subscriber selects the option 'Sell' or 'Buy' on the main menu 2. MDB formats the request and sends it to the 'transaction engine (TNE)' through the mobile network interface 3. TNE sends the request to the online trading platform of the enterprise for the transaction 4. TNE sends a confirmation to the subscriber after receiving confirmation from the online trading platform 5. TNE sends the transaction information to the CDR controller to create and store a CDR
Alternative courses	During step 4, if the TNE does not receive the confirmation, it sends a message to the subscriber stating that the transaction is not successful.
Associated use case diagrams/ sequence diagrams/ collaboration diagrams	The use case diagram is created in a Rose model called 'MTS.mdl'. The sequence diagrams are created under the package 'Logical view\ Use case realization' within the use case realization called 'Transact financial entity'. The sequence diagram is shown in the Annexure.
Exceptions	NA
Includes	NA
Special requirements	NA
Assumptions	NA
Notes and issues	NA

To specify the 'Manage system' use case consider all the activities that the system administrator is required to do. The normal action would be to login to his account and perform the necessary functions via the interface provided. The following use case specification provides these details:

Use case ID: UC7	**Use case name: Manage system**
Description	This use case allows the system administrator to monitor and maintain the system.
Actors	System administrator
Pre-conditions	NA
Post-conditions	NA
Frequency of use	NA
Normal course of events	The normal course of events is as follows: 1. The system administrator performs login. The system displays the 'Admin Menu' 2. The system administrator selects the option 3. The system performs the action
Alternative courses	NA
Associated use case diagrams/ sequence diagrams/ collaboration diagrams	The use case diagram is created in a Rose model called 'MTS.mdl'. The sequence diagrams are created under the package 'Logical view\ Use case realization' within the use case realization called 'Manage system'. The sequence diagram is shown in the Annexure.
Exceptions	NA
Includes	NA
Special requirements	NA
Assumptions	NA
Notes and issues	NA

As the next step, the use case behaviour should be spread out to the classes. For this we can perform the following activities for each use case flow of events:

- Identify analysis classes
- Allocate use case responsibilities to analysis classes
- Model analysis class interactions in interaction diagrams

Corresponding to each of these major components in the system, analysis classes were defined, which included classes for the following:

- Mobile network interface
- Web server and application server

- Database server
- Mobile device browser
- Remote access server
- Administration server
- OTPI and CAI

Analysis classes were identified for each actor/use case pair. Analysis classes are later grouped and regrouped to come up with actual packages, components, sub-systems, and so on.

Three types of classes were defined: entity class, boundary class and control class. Boundary classes interact with external systems. They are the intermediates between the interface and classes outside the system and are environment dependent. The classes outside the system can be of the following types:

- User interface classes
- System interface classes
- Device interface classes

There is only one boundary class per actor/use case pair.

Control classes represent the actual use case behaviour. There is one control class per use case. The entity classes represent the key abstractions of the system and stores the related information.

Figure 5.7 shows the purpose of each of these class types.

To allocate the responsibilities to the classes, we can use the following guidelines:

- Behaviour that involves communication with an actor should be assigned to *boundary classes*.
- Behaviour that involves the data encapsulated within the abstraction should be assigned to *entity classes*.
- Behaviour specific to a use case or part of a very important flow of events should be assigned to *control classes*.

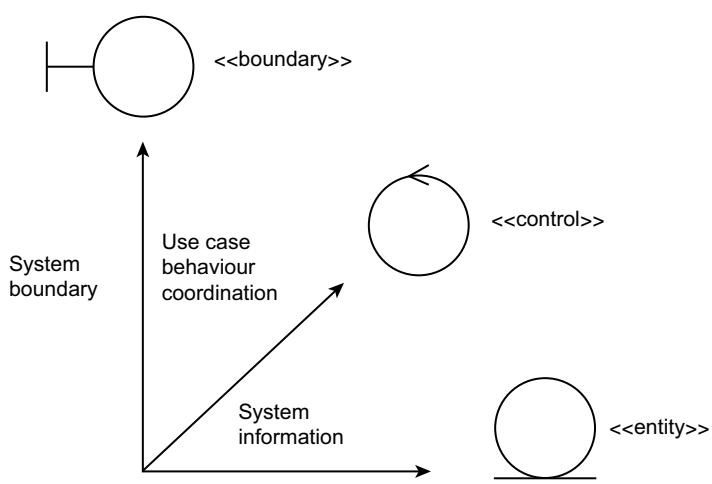


Figure 5.7 Analysis classes for a software system

As the next step, the interaction among the classes should be modelled in the form of interaction diagrams. Sequence diagrams and class collaboration diagrams (from the UML notation) fit into this. Next, the team created sequence diagrams for the use cases. These were created both for the normal and alternative flows that describe the other options available, in case the normal flow of events does not take place.

Sequence Diagrams

Figures 5.8 and 5.9 are sequence diagrams that were created during the detailed design phase. The 'Personalize profile' sequence diagram shows the steps that the user takes to personalize his profile. Similarly, the 'Retrieve information' diagram details the steps to retrieve and store information from the content server.

As the last step, the team grouped the real-world objects illustrated in the sequence diagrams into classes and packages. Hence, the classes were re-grouped and re-distributed as packages. The redundant classes were removed. Further, the packages can be grouped as sub-components, where each sub-component can be a component as described by the architecture.

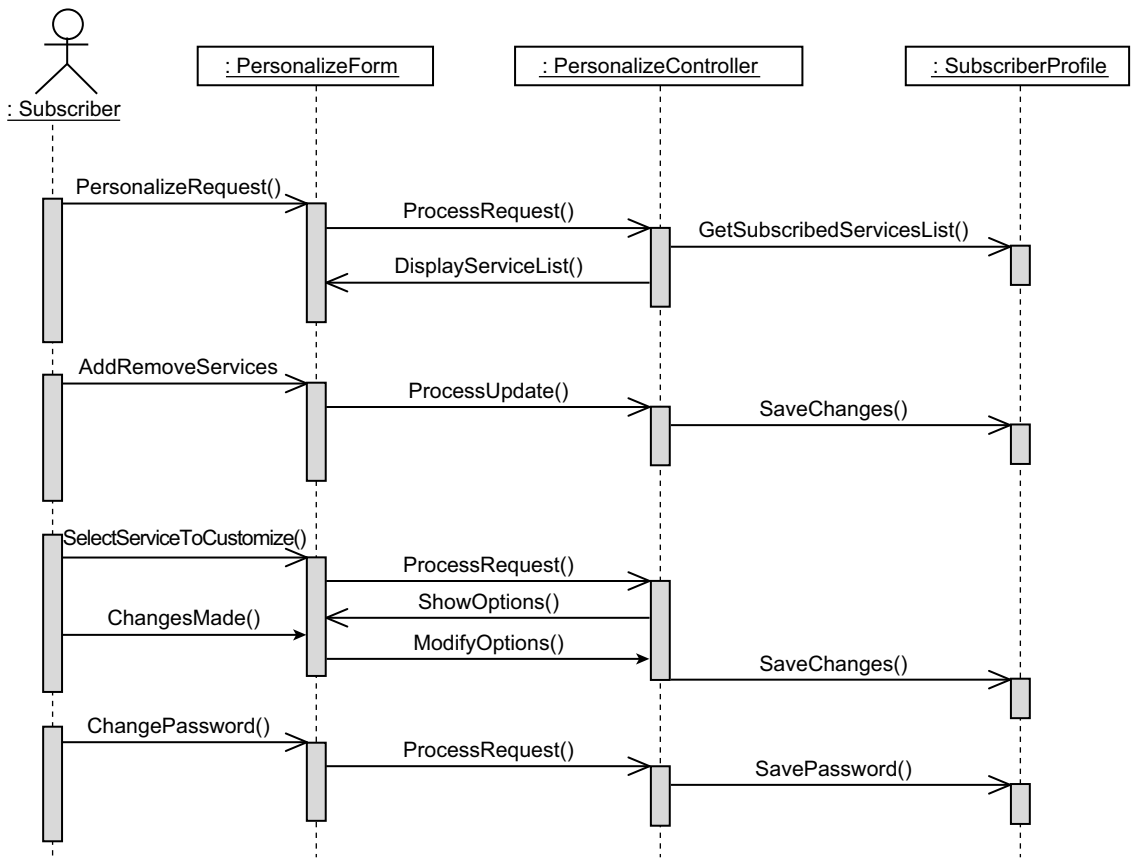


Figure 5.8 Sequence diagram for the 'Personalize profile' use case

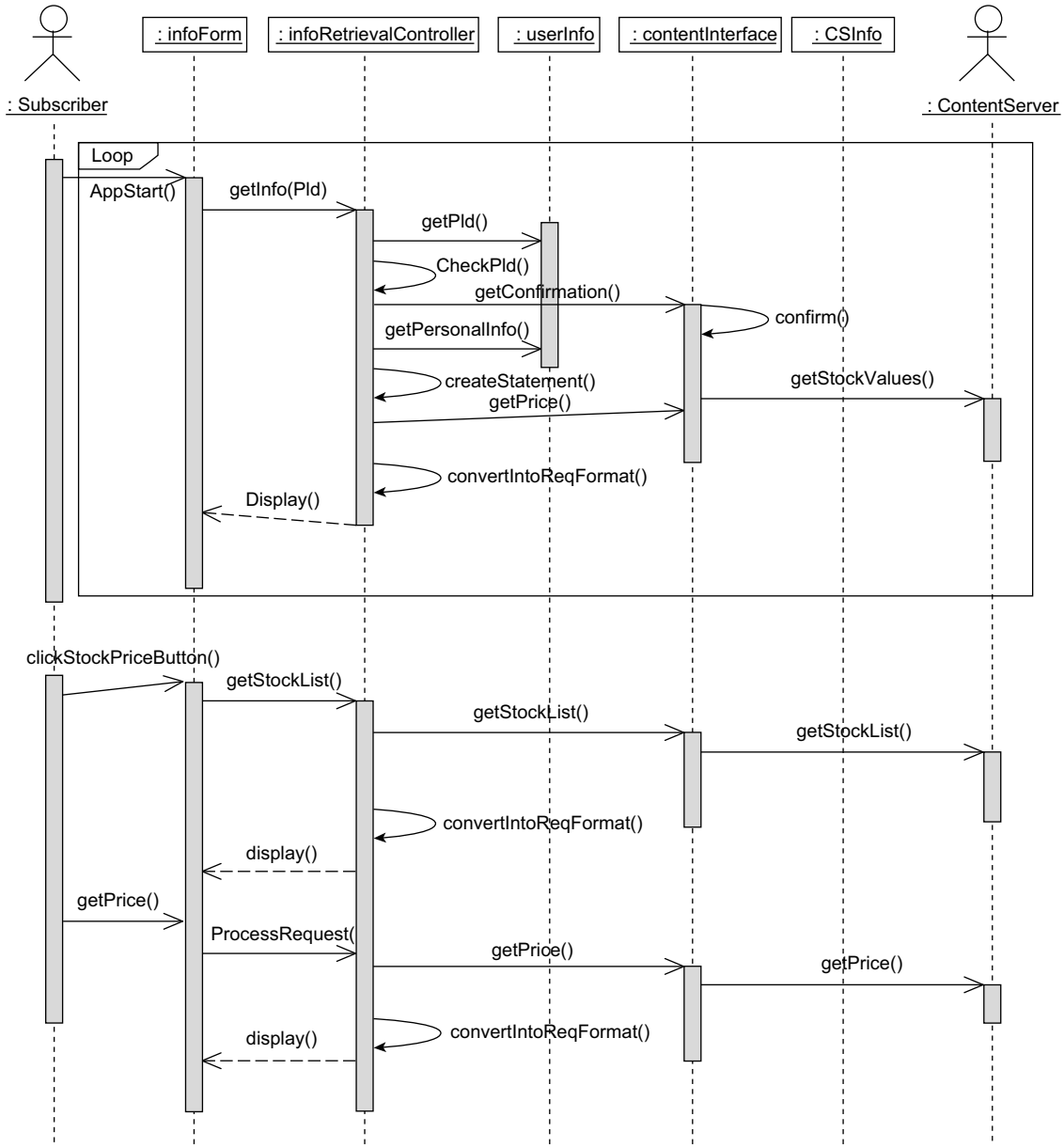


Figure 5.9 Sequence diagram for the 'Retrieve and store information' use case

Class Diagrams

The two class diagrams shown in Figures 5.10 and 5.11 give a mapping of the system to real-world components. Once we have the class diagrams we are ready to construct the actual system.

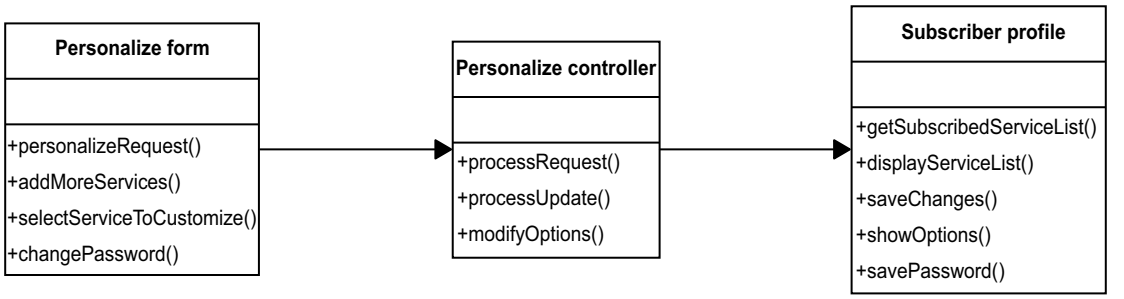


Figure 5.10 Class diagram for the 'Personalize form' use case

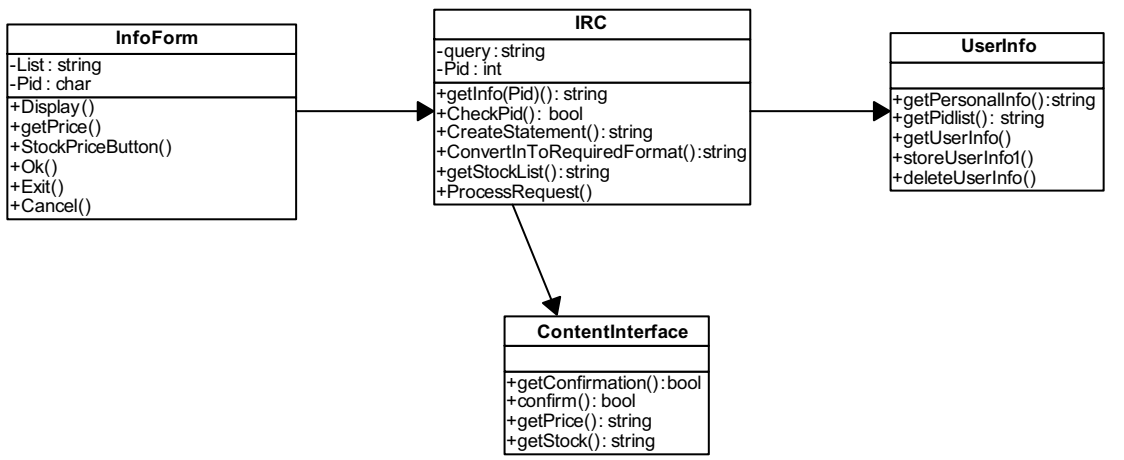


Figure 5.11 Class diagram for the 'Retrieve and store information' use case

Conclusions

The architecture of the system acts as a blueprint for the design phase. It is often represented as a set of interconnected modules or sub-systems that are organizational structures that structure the source code. Representing the architecture in terms of design elements helps in finding the interdependencies that govern the interfaces between modules.

Design is a very important phase in the software development life cycle for several reasons:

- Thinking in advance always helps (and it is cheap!)
- Cannot add quality at the end: this is in contrast to reliance on testing, more effective and much cheaper
- Makes delegation and teamwork possible
- Design flaws affect user: incoherent, inflexible and hard to use software
- Design flaws affect developer: poor interfaces, bugs multiply and hard to add new features

Design activities are usually classified into two stages: preliminary (or high level) design and detailed. At the high-level design we first identify modules, control relationships among modules and interfaces among modules.

The outcome of high-level design is a program structure (or software architecture). Several notations are available to represent high-level design, usually a tree-like diagram called structure chart. For each module, design data structure and algorithm, a high-level design maps functions into modules such that

- Each module has high cohesion
- Coupling among modules is as low as possible
- Modules are organized in a neat hierarchy

The outcome of detailed design is module specification. A good design should implement all functionalities of the system correctly. It should be easily understandable, efficient and easily amenable to change, that is, easily maintainable. Understanding a design is a major issue as it determines the quality of the design; a design that is easy to understand is also easy to maintain and change. Unless a design is easy to understand, tremendous effort is needed to maintain it. A design having modules with high fan-out numbers is not a good design and lacks cohesion.

The characteristics of a good conceptual design include the following: it is in the customer's language with no technical jargon, describes system functions, is independent of implementation up to a point and is linked to requirements.

Designing is not an ad hoc process. We can make the design process systematic and structured. This will add to the features of design by reducing errors and ambiguity, improving the readability and, most importantly, will ensure that the design is consistent and complete.

Best Practices and Key Lessons from the Case Study

- A good system design is based on a sound conceptual model (architecture). A system design that has no conceptual structure and little logic to its organization is ultimately going to be unsuccessful. Good architecture will address all the requirements of the system at the right level of abstraction.
- Consider the exceptional scenarios that lead to system failure in the design.
- A good conceptual model is easy to communicate: Ease of communication of the conceptual model on which the design is based is essential for successful solutions.
- Keep it simple and practice information hiding: Minimizing complexity and information hiding is perhaps the most powerful design heuristic because it explicitly focuses on hiding details.
- Design for re-use: Re-factor classes to distribute responsibilities and behaviour.
- To come up with a robust design there needs to be clarity of requirements.
- Selecting a proper methodology for development based on the requirements and resources is the first step towards a good design.
- Prioritizing requirements and charting use cases help produce a sound conceptual model.
- Using case tools like Rational Rose to document the use case analysis aids in faster development and is useful to produce standard artefacts that ease communication of design issues and objectives.
- From the block diagram of the system and the use case flow of events the boundary, entity and control classes can be identified to draw the sequence diagrams.
- Class diagrams help map real-world entities to software. They present the static structural model of the system and help identify the relationships between the classes.
- The design for the system must be flexible enough to be able to accommodate future changes.
- An open environment in the team is recommended and user interaction is required to come with a strong set of relevant goals to aid design.
- Good design is seldom arrived through a single-step procedure. It is arrived through a series of steps and iterations.

- Important design considerations include the following:
  - Component independence
  - Coupling (degree of inter-dependence)
  - Cohesion (degree of internal relatedness)
  - Exception identification and handling
  - Fault prevention and tolerance
  - Active considerations (code to periodically check, react, etc.)
  - Passive considerations (code to assert, throw exceptions, etc.)

Further Reading

There are several good books on general software design. Some of the books we recommend are *Object Oriented Analysis and Design With Applications* by Grady Booch (Pearson Education) and *Software Design* by David Budgen (Pearson Education).

The software design concepts are also well covered in some of the undergraduate books on software engineering, including *Software Engineering* (7th edition) by Ian Sommerville (Addison Wesley) and *Software Engineering Concepts* by Richard Fairley (McGraw Hill).

Software Engineering Institute has some useful resources on this topic. Refer www.sei.cmu.edu/str/descriptions/oodesign.html for details.

Design patterns can be learnt from several sources. We have given some pointers at the end of the first chapter. *Design Patterns* by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (Addison-Wesley) is the original and is still one of the best books on this topic.

The Unified Software Development Process by Ivar Jacobson, Grady Booch and James Rumbaugh (Pearson Education) gives good design fundamentals using UML.

We were also benefited by an old article written by D. L. Parnas and D. Weiss on Active Design Reviews: Principles and Practices (Proceedings of Eighth International Conference on Software Engineering, 1985).

Chapter 6

Component-based Development: Portal of Universal Bank

Contributors
Inderneel Sikdar
Srikanth
Kirti Garg

The science of developing software applications using components, popularly known as 'component-based software engineering' (CBSE), has grown from concept stage to a practical possibility to an essential way of developing software. With the emergence of several standards such as UML and with platforms such as J2EE and .Net supporting component-based development (CBD), it has become very easy to come up with complex and high-quality software applications very quickly. CBSE provides us a way to faster, better and cheaper software not just till the first customer shipment of the application but for its entire life.

Commercial off-the-shelf (COTS) components have been increasingly finding their way into software application development. The marketplace for COTS is increasingly getting better and bigger. However, it is very important to exercise caution while adapting CBD. The rules of the game differ for CBSE when compared with a regular (build from scratch) software development life cycle (SDLC).

In this chapter, we present a case study of Universal Bank. The task is to bring all its (about 600) Web sites, both internal and external, under the umbrella of one platform and framework for easy maintenance, extensibility and security. The expected cost savings are also huge. In other words, the task involved re-engineering, using a common set of components, a huge integration exercise of a large number of portals and Web sites that were previously developed on multiple platforms and technologies with their own style and structure. This case study brings out the importance, approach and the process of CBD and addresses many related meta-issues. For this case study, unlike previous ones, there is no explicit solution outline given at the end. Various challenges and the ways to overcome them are discussed throughout the case study. We also provide more background information on CBD in the later part of the chapter.

Broadly, the major learning goals for this chapter are the following:

- To make readers look at the component-based software development model as a viable model that can be used for software development.
- To help readers decide when the CBD approach makes sense and how to go about choosing the right solution.
- To make readers see the cost of software development and how that alters when the CBD approach is used.
- Finally, to explain how to develop CBD components that can be reused.

BACKGROUND

Why Component-based Development?

CBD focuses on building large software systems by integrating previously existing software components. This approach can potentially be used to reduce software development costs, assemble systems rapidly and reduce the maintenance burden associated with the support and upgrade of large systems.

This approach is based on the assumption that certain parts of large software systems reappear with sufficient regularity that common parts should be written once rather than many times and that common systems should be assembled through reuse rather than rewritten many times.

Consider the amount of time it takes to build industrially deployable software

Under such a scenario, it is important to understand that code reuse might be beneficial for saving time and expense incurred by 're-inventing the wheel', since the time needed to investigate, design, code and test gets significantly reduced in this process.

This principle, if you notice, is very fundamental to software engineering. It is used everywhere from code libraries to reusable functions that everyone keeps handy. The same principle is extended to include entire components that were either written by a third party or were written for earlier software in-house and are now reused.

Software	Time of development
OS/360 (IBM)	5,000 man years
JDK 1.5 (Sun)	2 years 7 months after JDK 1.4 (number of contributors not known)
Windows 2000 (Microsoft)	1 year 8 months after Windows 1998 (number of contributors not known)

Developing component-based systems is becoming feasible due to the following reasons:

- The increase in the quality and variety of available components
- Economic pressures to reduce system development and maintenance costs
- The emergence of component integration technology (e.g., Object Request Broker)
- The increasing amount of existing software in organizations that can be reused in new systems

CBD shifts the development emphasis from programming software to composing software systems, which itself is a massive task for large-scale projects.

Origin of CBD

CBD embodies the 'buy, do not build' philosophy made popular by Fred Brooks. While this trend has been encouraged for many years for all kinds of systems development, especially in the US Department of Defense, it was only in the early 1990s that the practice caught up in the industry.

There are two important developments that lead to the adoption of CBD:

- Availability of high-quality COTS products
- Evolution of open systems

Commercial-off-the-Shelf Components

The COTS components refer to software products that are ready to buy and use from a commercial vendor. It means that one can choose to buy, from a catalogue, components that are useful directly in building new systems. If one is able to identify such components and successfully use them in new applications or software systems, they are typically more reliable and save time, effort and money.

So far only well-understood components such as database systems, Web servers, application servers, e-mail and messaging systems and some office automation components such as word processing programs, spreadsheets and time management tools are available in the form of COTS components. These types of components are time tested and hence more reliable.

Open Systems

Open systems form a collection of interacting software and hardware that are designed to satisfy stated needs, with the interface specification of components provided. They must be fully defined, with specifications available to the public and maintained according to group consensus. OpenOffice is an example of open systems as against the formerly closed StarOffice on which it was based.

Open systems are based on market-accepted standards in terms of stable interfaces and hence makes them more adaptable when compared with custom-made and more closed systems. Any changes or advances in technology can be accommodated more easily when open systems are used.

Open systems need not be based on COTS components, and COTS components need not be based on open systems, but they could be based on each other. For example, the well-known PDF format is an open system, but it is not based on COTS architecture. IBM WebSphere application server is an example of COTS-based system that is not an open system. Apache Web server is an example of an open system that is based on COTS architecture.

Together, these two factors propelled the use of pre-existing components in newer projects.

Some Examples of Cost of 'From the Scratch' Development

It is a general perception that CBD in general and using COTS components in particular will save significant amounts of money. It happens because we will save on the development time as well as cost of developing the equivalent components from scratch.

This perception has come across over the last couple of decades due to some famous examples, some of which are listed here.

Netscape browser rewrite

We have discussed the story of the rise and fall of Netscape in Chapter 1. This story again becomes relevant here. The Netscape 6.0 browser came out in 2000. There never was a version 5.0. The previous major release, version 4.0, was released almost three years before that. The long waiting period was caused by Netscape's decision to write the entire browser code from scratch. However, during this time Internet Explorer ate into the user base of Netscape and eventually became the de facto browser on all PCs.

Microsoft Vista (earlier code named as Longhorn)

Microsoft projected Longhorn as a completely new operating system development paradigm almost immediately after the release of Windows XP in 2001. It involved a complete redoing of file system, user interface, access rights, explorer, and so on. The project, which was later renamed as Vista, was finally released in 2007. This is the longest period in which Microsoft had not released a major operating system, and Microsoft had only just survived by doing incremental updates to its existing operating systems, and finally choosing to keep lots of its current code base.

Firefox browser

This was essentially a fork on the pre-existing Mozilla browser, which in turn was based on the original Netscape browser. By adding features to an existing code base, the browser, which is only 2 years old, is today the second most frequently used browser (with close to 12 per cent penetration) after Internet Explorer. Microsoft delivered Internet Explorer, version 7, almost 5 years after they delivered Internet Explorer, version 6. Firefox managed to release FF2 within 2 years of releasing the first beta release.

All these and many more incidents have managed to convince industry leaders to look at CBD as an alternative that is not only viable but also competitive. Wherever it is not critical to build for themselves, people are willing to look at available solutions, and wherever it makes commercial sense, people are willing to pay for the same as well. In fact, many companies are now making money by making CBD components along with other full-fledged products.

Case Study: Component-based Development for Universal Bank Portal

'What may be the right approach for this project? Should we go ground up or should we go the components way?' These were some of the questions troubling Udaya, project manager at Fact-Tree. Udaya was going through the requirements documents of Universal Bank's integration project that Fact-Tree bagged amidst strong competition. It was a 'once in a lifetime' kind of project. Over the years, Fact-Tree had done tremendously well in terms of handling complex projects and had carved a niche for itself in the software industry. It had always been considered as a pioneer in adopting new methods of development and setting standards for the industry. This project was being considered as another chance to confirm their eminent position.

Universal Bank: A Software Maze

True to its name, Universal Bank has branches all over the world. It has been operating since almost a century and hence has quite a strong customer base. It has vast financial information gathered over the century, and it uses several mechanisms to manage it. The bank is a pioneer in adopting technology and in the process has acquired and adopted several records management and enterprise information systems such as SAP and PeopleSoft.

These enterprise information systems are used to manage financial information. These are closed systems, inaccessible from outside the specific system. The bank then needed a framework to publish this information on to its intranet, the external Web and to its partner and to provide a common platform for its country offices and its 2000 plus field staff to access this information.

Universal Bank had allowed its branch offices to manage their own Web sites. The purpose was to share the operational information across the offices and with their partners. Currently, this was a whopping number, 600 Web sites with more than 200,000 pages.

Also, each branch was independent and had selected its own Web site development approach and technologies. This approach had its own troubles, for example, the inconsistent user interfaces. Information sharing across various enterprise systems was difficult as there was no institution-wide cataloguing of information or any other search technique to extract the right information.

Now Universal Bank was planning to transform the intranet from a basic information source into a full-blown corporate portal, offering personalization features and providing a platform for Web-based applications.

With the current system, this plan was nothing more than a plan. The main drawback was that many of its constituent banks has used different technologies to build their Web sites, with 50 per cent of its banks using Lotus Notes, 40 per cent HTML, and the rest PHP, ASP, and so on.

The bank needed a common technical infrastructure for its intranet and public Web site to publish varied information on to its intranet, its external Web and to its partners. Thus, they decided to integrate all the 600 Web sites into a common one. The major requirements included the following:

- Create an easily maintainable structure
- Integrate navigation and search so that information is easier to find
- Reduce development time

The project was awarded to Fact-Tree, owing to its track record in integration projects.

Glossary

Records Management Systems

A record management system (RMS) is a software package that can be used to identify, track, store, circulate and dispose records of an enterprise. In other words, an RMS can take care of the entire life cycle of the records.

Enterprise Information Systems

Enterprise information systems provide support for large volumes of enterprise-class data with high quality of service and security to help enterprises in managing their business processes. Several applications such as content management systems and customer relationship management systems fall under this category.

Fact-Tree Gets to Work

Fact-Tree had undertaken several such integration projects earlier. But all agreed that this one was the most prestigious and complex, with 600 Web sites to be integrated. The management decided not to take any chances and be careful from day one. They had been given three months to prove the worth of Fact-Tree.

Since the project demanded utmost importance and planning to complete in the given time frame, Dravid, the business manager of Fact-Tree, wanted every step to be carried out with clockwork precision. He had been in the industry for enough time to recognize that planning, control and feedback are the mechanisms that help tame the bull. He assembled his experienced project managers with proven track records to draw a clear plan and development strategy for the Universal Bank project.

The project was planned in the following manner:

Stage	Time
Requirements gathering and analysis	Two weeks
Design	Two weeks
Implementation	One month
Testing	Three weeks
Deployment	One week

The development had eight internal milestones. Bi-weekly status-check meetings were proposed. Any delay in milestone achievement by a single day was to be reported and corrective measures were to be taken.

The Users and the Requirements

A team of three members was sent on-site to gather requirements, to study about the existing sites and their usability and document the same. They spent 10 days to identify the requirements. The following were identified as the critical parties involved:

- *End users*: They are the people for whom the content is created and they are the ones who will access the Web site pages via a browser such as Internet Explorer, Mozilla, Firefox or Netscape.
- *Business users*: They will create and publish the contents on their Web sites using ePublish tool (explained later).
- *Site developers*: They will be configuring the Web site using Site Manager tool (explained later).

The following specific requirements were identified:

- *Structuring of the bank's Web sites*: The structuring was to be in terms of content, look and feel:
  - The requirement is to have a uniform look throughout all the Web sites, both internal and external, and all the portals. This will enable user comfort in browsing through any content at any site.
  - The basic requirement in any Web site is to have left navigation panel, header and footer. Any site can choose to have header and footer with standards and optionally choose the left navigation panel.
- *Information sharing across the bank's sites*: Although information sharing is possible even with independently developed Web applications, the biggest bottleneck is that every application has its

own access control mechanism and this increases inconsistency and overhead in terms of allowing users access to specific information.

- *Publishing of bank's information and content management*: Each branch was evaluated on what applications made most sense, and some applications that needed to be globally deployed were short listed. Such applications were expected to increase in their user base and this made end-user management more complex. Going for a new component-based system will allow consolidation of all these resources into one single place and end-user training will become simple in terms of using tools such as Site Manager and ePublish.
- *Integration and sharing of information from existing bank implementations*: The new system should still allow access to some of the existing sites that are not going to be migrated immediately by providing possibility to access the corresponding URL as part of the content without much effort. A common repository for cataloguing and meta-tagging searching content is a big challenge in existing applications as each of these is based on different architectures and systems. In the new scheme, search capabilities should be built using strong capabilities of relation databases in the industry.
- *The application should be multi-language compliant*: The application should have multilingual capabilities and should allow enhancing the capability to other languages.
- *Strong infrastructure support for Web site migration, hosting and maintenance***:** Infrastructure capabilities to cater to the migration of all the existing Web sites to the new system should withstand the amount of load in terms of Web application servers, operating system and database servers.
- *Integrated authoring system and workflow*: The new system should provide a seamless and powerful environment for content creators. Non-technical people create the contents of the pages being hosted. These authors should be able to have simple user interfaces to publish content without knowledge on HTML. Authors should be capable of creating many cross-links between pages, and these changes must be stable against restructuring. The requirement is to have a strong de-centralized content creation, which should rely heavily on a powerful workflow model that can be easily customized and is resilient against organizational change.
- *Meta-data creation*: The new system should capture meta-data (creator, subject, keywords, etc.) that are critical when managing a large content repository. This should also include keyword indexes, subject taxonomies and topic maps. Each site has its own meta-data, and as of now, there are no common meta-data though efforts are on to create a single repository reference.
- *Version control, archiving and security*: Strict version control is necessary for legal accountability, backup and disaster recovery. A simple but powerful interface must be provided for these features. Adequate security levels and audit trails must be in place to protect the integrity of the content.
- *Integration with external systems*: A content management system (CMS) is typically only one of a number of systems used to present information on the intranet or Web site. It is very important to note that the CMS should be seamlessly integrated with the rest of the business systems within the enterprise. Otherwise the significance of the CMS is lost.
- The integration with the existing system should be based on open or industry standards.
- Completion of the requirements gathering phase brought relief to all. The client was satisfied that no requirements were missing, and the team was happy that since all requirements were clear and there was no scope for confusion, they could follow the linear waterfall development as planned.

Glossary

Authoring Systems

Authoring systems allow people even without programming skills to create software or content that may be rich in terms of graphics and other effects, which is not otherwise possible without programming knowledge.

Application Architecture and Design

The project was named Internet Services Program (ISP) and aimed to bring the entire Internet and intranet sites of Universal Bank under a common framework to achieve easy site maintenance and to reduce site development time. The design started ahead of the planned schedule. The first five days were devoted to the architecture and the remaining period of the two weeks was assigned for low-level design activities. The team defined and developed a framework for site development and content management. To cater to the aforementioned needs, a framework (Figure 6.1) was defined that aimed to unify the bank's intranet and the external Web sites. This involved provisions for searching, accessing and aggregating the bank's operational information spread across more than 600 independent Web sites. A content management and publishing tool, named as ePublish, was to be developed to manage and publish onto the bank's intranet and external Web sites. The entire application was divided into eight modules as listed below:

- Site Manager
- Presentation Engine (PE) and PE Controller
- Content Management Framework (CMF)
- ePublish
- Forms Engine

Glossary

Taxonomies

Taxonomy is a classification of concepts in a structured manner. All the concepts are connected with other concepts using specific relationships (such as 'is-a' relationship).

Topic Maps

Topics maps is a standard way (ISO/IEC13250:2003) of representing knowledge for the purposes of interchange and findability. A topic map can represent knowledge or information using topics (any concept or an entity), associations or relations between these topics and their occurrences or instances.

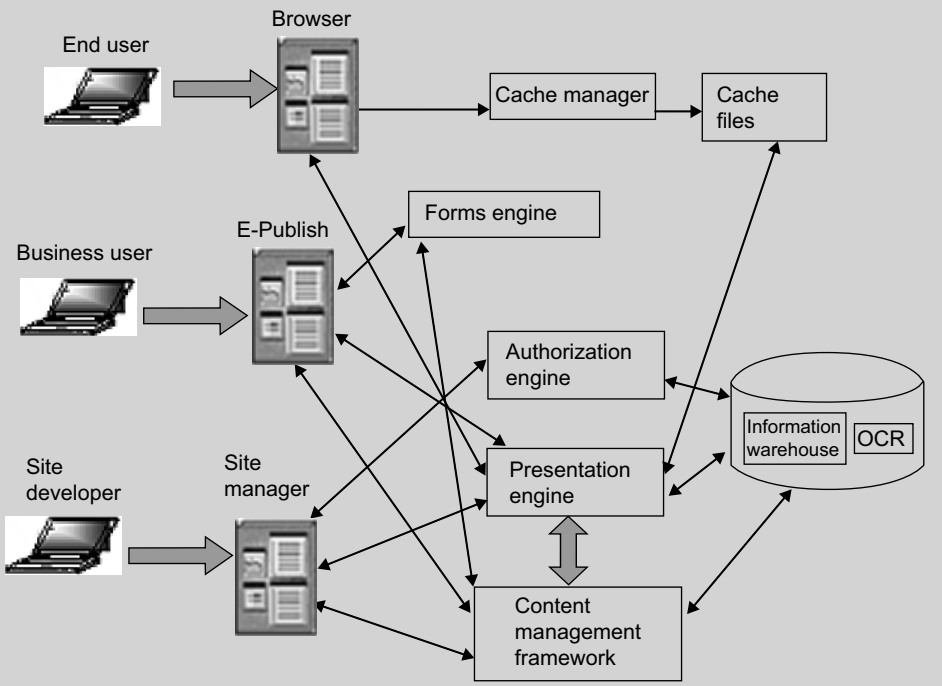


Figure 6.1 Logical view of the ISP framework

- Authorization Engine
- Cache Manager
- Information Warehouse (IW) and Central Control Repository (CCR)

Each of these is explained in greater detail below.

The team worked on the system architecture and came with the given architecture (Figure 6.2). The system architecture consists of several layers and engines.

The functions of each module were defined as given below:

Site Manager

The site manager should facilitate creation and publishing of Web sites on a Web server and should be made viewable through a Web browser. It should provide graphical user interfaces for creating and configuring the pages for a site. The look and feel and configuration settings for the pages could be modelled or changed to display desired information. Also, it should provide user administration features for the sites, that is, users could be created and assigned with different roles such as site administrator and site developer at site levels.

Presentation Engine

The PE should provide developers or Web masters a framework for developing Web-based applications, Web sites or portals. It could include components such as page, page instance, templates and widgets.

Each page should be designed as a child object of the site. And each site might have any number of pages, which in turn could have a number of child pages. Each child page could be associated with a set of widgets to render the output. The page instance could be used to add additional parameters to the widgets, customize it according to the requirements and also to retrieve the target URL. Each page should be associated with a standard template. The templates could be predefined and could be changed when required

Glossary

Widgets

Widgets are a set of components similar to personalization of portals (e.g., my.yahoo.com components such as search, header, calendar, banner).

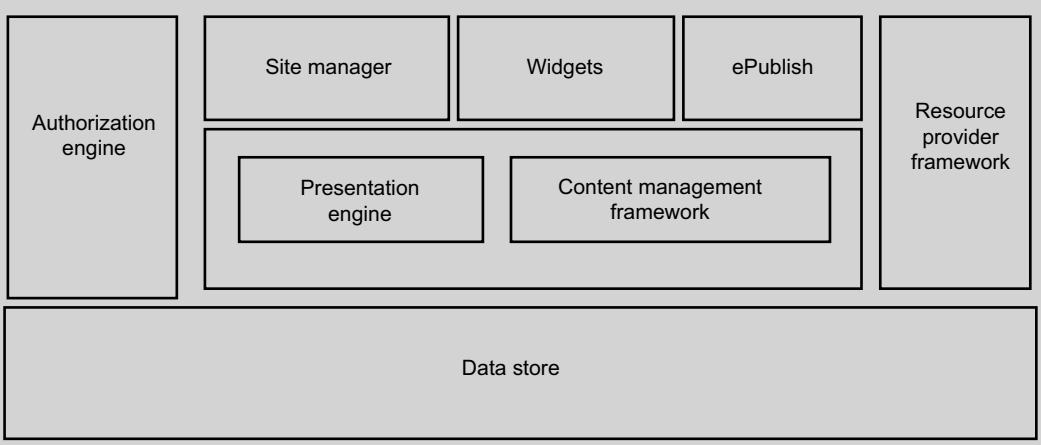


Figure 6.2 Block diagram for the ISP

and it should serve as an interface. Widgets encapsulate presentation logic to render HTML and contain very little business processing logic. They rely upon the business objects to do business processing that retrieves data in objects. The widgets can be configured from the page. Each page can be associated with any number of widgets.

PE Controller: The PE controller was used to serve the request coming from the end user. It would authenticate the user and would render the page needed to the Web browser and was part of PE.

Content Management Framework

The CMF should allow the user to create and maintain dynamic Web site content. It provides content objects and application program interfaces (APIs) to access dynamic content, which is used for building content-oriented Web sites. The backend systems store the information, which could be accessed from the IW and from the CCR. ePublish uses the CMF for publishing the content and the site manager uses it to create a template for content manipulation. The CMF API layer would interface with the IW and the CCR for extracting the information and creating the content. The CMF was responsible for populating the various content types identified by Universal Bank from its various repository sources. One of the main objectives of the ISP framework was to provide a powerful search capability for Universal Bank, both on the Internet sites and the intranet. In order to achieve this, cataloguing of information that goes on these sites becomes important. For this purpose, each site should have a set of content types and related meta-data defined and captured. Content types are used for classification of content. Content types are groups of attributes that form a template for content manipulation. The content types are stored in Folder, which was meant as a logical container of content types.

ePublish

- ePublish supports rules-based content publishing that allows the site owners to control the publishing process and give options to change the permission of the user and site properties. It allows business users to create menus where each menu item could be mapped to a page. It consisted of the following modules:
  - *Design selection*: Branding images, left navigation and dynamic highlights
  - *Administration*: Site properties, user management and folder properties
  - *Content*: HTML folders, image and resources folder.

Forms Engine

The forms engine should handle the content management application and generate the content input form dynamically depending upon the input template. The content input form generates the forms based on the input template and the related attributes. The form engine should be built to communicate with the widget library, business objects and the data access layer. Invoking the form engine, the user should be authorized by the authorization engine. On successful authorization, the form engine would interface with the CMF API layer to generate the page. Also, the form engine through the CMF API should be able to communicate with the database and retrieve the content pertaining to the content type that was clicked.

Authorization Engine

The authorization engine should perform the function of login, user validation and assigning user-level privileges for accessing the Web sites from the site manager.

- *Login*: The supplied user ID and password should be validated, and if they are correct, the user-level attributes would be set. The session is then maintained through cookies for navigating the entire Web site till logging out.

- *Roles/privileges in the site manager*: The authorization engine is responsible for assigning the user-level privileges for accessing and performing certain actions on the Web sites and pages at the site manager level. The user list should be obtained from the server, and for the list of selected users, the required rights are assigned. The rights might include create/edit/delete the site, page and content related to the sites. Appropriate menus are enabled/disabled based on the rules set when the user logs in.
- *The CCR*: The CCR should act as a data access layer between the CMF and the IW. The CMF uses this interface to access the information from the IW. For content management, the content types are grouped under a folder. The association of content types with the folder and the process of creating, deleting the folders is also done here.

Cache Manager

The function of the cache manager is to cache the Web pages pertaining to a particular Web site that the browser requested. To enable this, the content delivery server (CDS) would be connected to the HTTP server. The CDS looks for the cached page in a physical folder location. If the Web page is available, it would serve the cached page to the browser. If the Web page is not available, the request will be forwarded to the application server, which will then generate the required page and render it to the browser and will also place a copy of the page in the physical folder. The deployment diagram is given in Figure 6.3. A server will cache the Web pages and will serve the cached Web pages to the browser.

Information Warehouse and Central Control Repository

There would be a logical separation at the database level for cataloguing the data. The IW would hold the data that were coming from different sources such as SAP, People soft, IRIS, CDS. The CCR holds the data related to the ISP framework, which includes Web content and data pertaining to the PE.

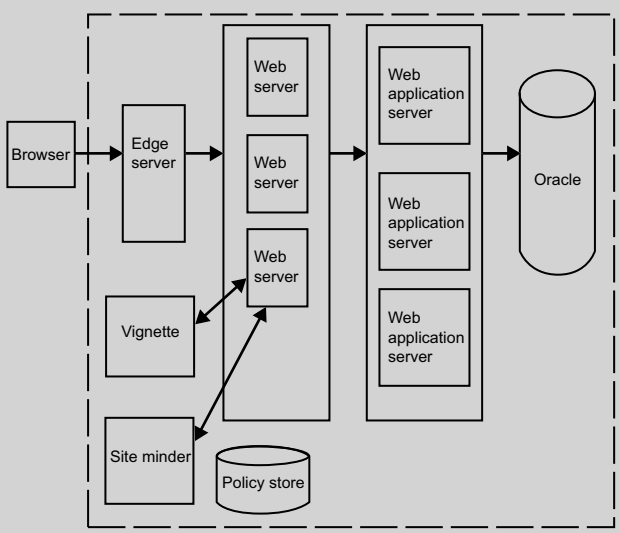


Figure 6.3 Deployment view for the ISP

It was decided to move some sites as a sample into the framework defined. Then other sites and content were to be migrated onto the framework in a phased manner.

Are We Going Right?

While discussing the requirements and coming up with the architecture, the team grew conscious of the fact that similar work had happened earlier. Some commented that parts from previous projects would have fitted here well. Everyone started talking about their experience with previous projects and the kind of advantage that experience was bringing to the Universal Bank project.

Udaya had been contemplating the idea of 'reuse' since long, since the day she went through the requirements documents. Thinking that the time was right, she floated the idea in the next status meeting.

She put forth her idea that instead of starting the development from scratch, the team could utilize component reuse. She was enthusiastic that as several projects with similar application and scope were done earlier, it was very much possible to use the components already built for those projects. She supported her argument by stating that frequently occurring components of a large software system could be developed once as a component, made a part of a component library and reused. Further, assembling the components rather than developing them over and over could help build similar systems. 'This is like the subroutines that we use in writing programs'.

Some of the team members acknowledged the idea and added from their experience on earlier projects that they also found that they are repeatedly doing the same things and often contemplated on creating a library of reusable code. However, due to project and time pressure, and the fact that previous clients had not been very appreciative, this idea did not materialize.

Udaya started listing some other advantages of reuse and components-based development. She emphasized that reuse would lead to reduced time to market due to the major efficiencies and economies of scale. It would simplify the process by reducing the total weight of the development. She said, 'Imagine the ease in maintenance'!

She further explained that problem identification would be quicker, as due to behaviour isolation, errors could be attributed to specific components. The overall maintenance costs would reduce through quick replacement of components without new code creation. She also said the following: 'While testing and maintaining, the surprises will be eliminated, as through reuse you will be utilizing tested and known functionality backed by clear specifications and interfaces'.

Visualizing the idea was not that easy. Some of the members had doubts in their minds. They asked if it was actually possible to construct complex systems by assembling them from a catalogue of reusable software components.

Jimmy, one of the project leaders, put forward his doubts. His argument against the effectiveness of code reuse was that reuse of code was not going to be cost effective as being promised, as coding took only 10-15 or at most 20 per cent of the development time. Hence, how was reuse actually going to cut costs effectively? 'What about the effort and cost requirements for building reusable components'?

Udaya replied that not just the development costs but also maintenance costs would go down. And since maintenance was one of the costliest activities, this would mean overall reduction in cost.

Jimmy, who was trying to digest the information as it was coming, further asked, 'How will it affect the existing development processes'?

'Oh! It should not be a lot I guess', replied Udaya.

She continued, 'There has to be a change in the software development process as well. The entire outlook needs to be different. There is no way we can do a big bang integration at the end of the implementation cycle—the integration has to be planned and must start early and has to be managed continually till the end of the development phase'.

Chandra, another project leader, was interested in knowing if there were any proven results available?

'Oh yes!, I know of at least three', Said Udaya and she started listing them. Deep Space Network Program at the NASA Jet Propulsion Laboratory, Lewis Mission at NASA's Goddard Space Center and Boeing's new 777 aircraft with four million lines of COTS software!

Everyone agreed to the idea that reuse of code and test cases would bring immense reduction in effort and time requirements. But for reuse of components, or architecture, while people did appreciate the idea, they were a little apprehensive. Doubts lingered in their minds.

Jimmy, was worried about the cost and time involvements with component development. He wanted to know about the total cost of ownership of the components and the potential payback? He was not sure if the management was willing to incur the added expenses associated with creating reusable software components.

Udaya answered that this approach could potentially be used to reduce software development costs, assemble systems rapidly and reduce the maintenance burden associated with the support and upgrade of large systems. 'In many of the earlier projects, we have worked on portal and Web site integration. In this project also, there are many sites and portals to integrate, which means that the reuse level of the component is going to be quite high. Hence, this might be the right time for the management to go for CBD, which certainly has costs, but there should also be the returns and long-term benefits'.

Chandra asked, 'And what about domains for which we do not have any readily available components, but for which there do exist options in the market'?

'Then, we shall look at those options and evaluate if they are suitable for our needs', said Udaya. 'Incidentally, this notion of looking at pre-existing components in the market is called components off-the-shelf or COTS', added Udaya. 'The caveat here is that the client should be OK using the COTS features, for that we will check with Universal Bank'.

The team felt that though the idea could be tested, it was sceptical about the success of the project, as the development method might have to be changed altogether. As a company, Fact-Tree had always encouraged new ways of doing old things, and Dravid, the business manager, thought that success in this approach would mean a lot to the company and asked whether Udaya could take up the project and do it in her proposed way. She accepted the challenge and the project began!

CBD Development—Baby Steps

A development team consisting of 16 members was created, which would be later divided as per the project requirements and modules identified. The team was high in spirit as they were going to work on the new concept of 'component reuse' to build the product.

Udaya thought that without a detailed plan the project would go awry since except requirements gathering, the rest of the SDLC was to be carried out only offshore (Figure 6.4). Hence, she created a project plan containing all the activities that had to be done. The software life cycle for this development model was identified as follows:

- Requirements analysis and definition
- Specification
  - Component selection and evaluation
  - System design using the components
- Provisioning
  - Make sure that the components are able to provide required functionality

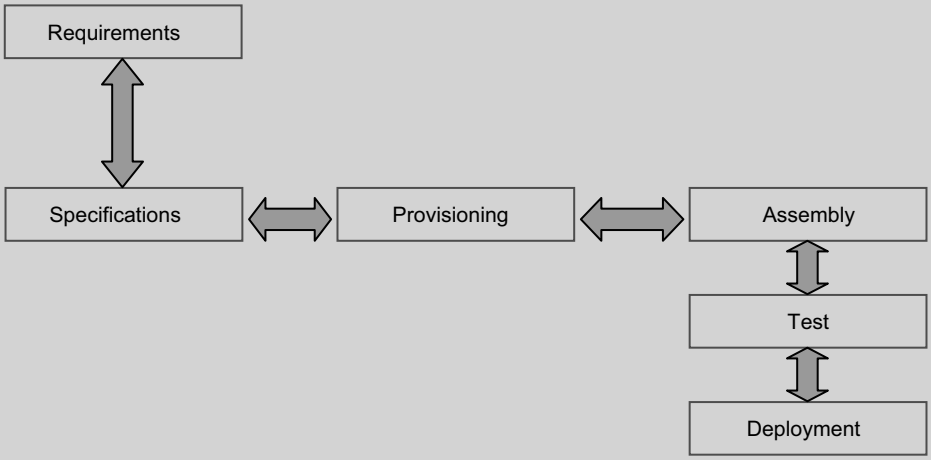


Figure 6.4 SDLC of the component-based development modelW

- Assembly
  - System integration using components
  - Writing glue code to make components to interface with each other
- Test
  - Verification and validation
- Deployment

This model was quite different from a conventional SDLC, which they would have otherwise followed (see Exhibit A).

The advantage that Udaya hoped to get out of this exercise was development of artefacts that the team could leverage at a later stage in some other project that which had similar requirements.

Exhibit A Traditional SDLC

The following are the stages in the traditional SDLC represented in the well-known waterfall model (Figure 6.5)

- Identification of business needs and constraints
- Elicitation and collection of requirements
- Architectural design
- Detailed design
- Implementation
- Testing
- Deployment

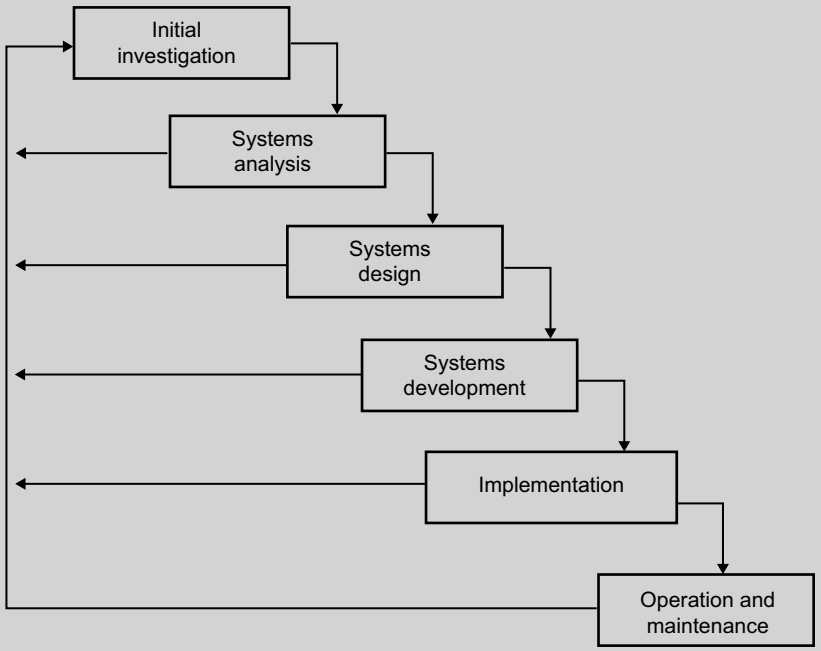


Figure 6.5 Conventional SDLC

Domain Analysis

Each team leader went on to do domain analysis for his or her module to find out the components that could be extracted from the earlier projects. The intent of domain engineering was to find, assemble, list and distribute a set of software that could have applicability to existing and future software. Also, if the reusable components did not exist, the team leaders were given options either to make use of COTS components available in the market or to construct the same to suit the application.

Each team leader faced different sorts of challenges to come up with the reusable components to suit his or her own module in particular and to suit the project application in general. They had a very tough time sorting the components from the earlier projects to locate the component with nearly the same application.

'Without proper record of existing reusable components, and creation of a reuse library, how can we find the component for reuse', commented a frustrated team leader.

The Need for Structured Development

The team found that components differed in complexity, scope, functionality, infrastructure and skill sets needed to use them. Hence, to manage them further, they grouped them in three categories: graphical user interface (GUI), service and domain components.

- GUI components included buttons, slidersand widgets used to build the user interface.

- The service components included database access components, messaging and transaction services, system integration services, and so on. These used additional infrastructure or systems to perform their functions (e.g., messaging platforms, workflow systems). Though more sophisticated than GUI components, these did not provide the full functionality by themselves.
- Domain components, also known as business components in general and 'reusable' domain components in particular, were the focus.

It took them nearly a week to come up with the list of reusable components, which then had to be modified to suit the current application. The short-listed candidates for component reuse were the following:

- Meta-data components
- Search components
- Entities at the database level to avoid any additional effort spent on normalization
- Caching
- Authentication and authorization
- Integration components that get data from external systems

Some of the team leaders decided to purchase the components that were not available or the available component did not fit well with the current application. As many of them had firsthand knowledge about the selection of the COTS components, they got the product from the COTS market and made it ready—actually patched it—to ensure that the modules functioned as expected. Some of the team leaders who were quite sceptical about adopting COTS components went on to build the component from scratch to suit the functions of their module.

For example, Jimmy, who was in charge of selecting the application server, ran through a list of tentative COTS candidates. He narrowed it down to WebSphere, BEA, Oracle Application Server, Tomcat Serer, and Sun Java System Application Server. He then ran a variety of tests to benchmark each of these against each other. The list of tests is shown in Exhibit C. Based on these tests, he decided to recommend the use of WebSphere.

Udaya worked with Dravid to identify how a component could be demonstrated by taking through the entire CBD lifecycle. She short-listed the GUI widgets for this (see Table 6.1).

Stage	Work on/with the component in that stage
Requirement analysis	A widget was required that would pull data from the database based on some input parameters. Another widget was needed that would display the content information extracted from the database and present it in a tabular format or as requested by the user.
Specification	The DAO Retrieve widget matched the criteria for widget for data retrieval from database, and thus the DAO Retrieve widget was selected for the CBD. The DAO widget is a non-display widget and provides other widgets with the backend support of retrieving information from the database based on some input parameters provided by other widgets. The Display table widget also was added to CBD because it provided the functionality of presenting the required information in tabular format or user-requested format takIng the backend help of the DAO Retrieve widget.

(Continued)

Stage	Work on/with the component in that stage
Provisioning	Fact-Tree was to develop both the widgets in question internally. The DAO was something that Fact-Tree had developed for Universal Bank in an earlier project. The other widget could also be developed similarly.
Assembly	Both the widgets were to be integrated using a tool called Site Manager. Tests showed that they could be well integrated with the project. The widget team was asked to write the glue code needed for the same and also provide good documentation for both the widgets for future use.
Testing	The working of the integrated code (integration of both the widgets with the existing project) was to be tested in detail to get satisfactory results. The testing process for this is explained in great detail in Exhibit B.
Deployment	It was hoped that with all above processes done carefully, both the widgets could be deployed in the project successfully.

Table 6.1 Widgets for various stages in the SDLC

Exhibit B Test cases for DAO retrieve and display table widgets

Test case ID	Test case name	Test case description	Test steps		Test case status	Test priority
			Step	Expected		
DAOR0	Validate input parameters	To verify whether the input parameters are taken correctly or not	Using the display table widget, enter any input and check the output	The required data are displayed in the display table in the browser	Design	High
DAOR02	Validate database connection	To verify whether the widget is able to connect to the database or not	Pass the database parameters to the DAO retrieve widget to connect to the database	The widget after connecting to the database should confirm the connection establishment	Design	High

(Continued)

Test case ID	Test case name	Test case description	Test steps		Test case status	Test priority
			Step	Expected		
DAOR03	Validate data retrieval from the database	To verify whether the widget is able to retrieve data from the database	Request some data from the database using the DAO retrieve widget by passing parameters using some other widget	The requested data should be retrieved from the database and manipulated as per requirement	Design	High
DAOR04	Validate non-display property	To verify that the widget is a non-display widget and takes the help of some other widget for data interactions	Try configuring the DAO widget independently without using any other widget	The configuration must ask for some other widget for which the DAO retrieve widget acts as a back-end and also there is no display of data only through this widget (non-display widget). It must use another widget for data manipulations	Design	High
DT05	Validate displaying output	To verify that the display table widget is able to pre sent the output in a formatted manner	Use the display table widget for displaying some information in the output by specifying the required format for display	Based on the information that has been given to the display table widget, the output must be generated in the required format(may be as a table or any other user-specified format)	Design	High

(Continued)

Test case ID	Test case name	Test case description	Test steps		Test case status	Test priority
			Step	Expected		
DT06	Validate data retrieval from database	To verify that the display table widget is able to pass parameters to get data from database	Use the display table widget for passing parameters for retrieving data from database by using some widget such as DAO retrieve widget	The required information from the database must be extracted using a data retrieval widget such as DAO retrieve widget and the data are outputted in the required format	Design	High
DT07	Validate hyperlink facility	To verify that the display table widget can be configured to make link locations for images, anchor tags, and so on	Use the display table widget for linking different elements to the display table, for example, specify a link in the display table to some image	When the link is clicked or when the link is followed, the specified image must be seen in the output	Design	High

In addition to these test cases, Udaya listed other criteria for success, such as response time, time to load, and visual display on different browser versions for these GUI widgets.

Exhibit C Process (matrix) description of the component selection used for the application servers

Benefits	Description	Features	Web-Sphere	BEA	Oracle Application Server	Tomcat Server	Sun Java System Application server
Certified J2EE1.3 Compatible	Implements latest J2EE specifications	Servlets (version 2.3)	×	×	×	×	×
		JSP (1.2)	×	×	×	×	×
		JDBC (2.0)	×	×	×	×	×
		JNDI (1.2.1)	×	×	×	×	
		JTA (1.0.1a)	×	×	×	×	×
		JMX (1.0)	×	×	×	×	×
		J2EE Connection Architecture (1.0)	×	×	×		×
		EJB Container (2.0)	×	×	×		×
		JMS (1.0.2b)	×	×	×		
Developer productivity	Getting applications developed and deployed in the shortest possible time and at the lowest total cost	Graphical deployment management with builder	×	×	×		
		Ant-based build tools	×	×	×		×
		Hot deployment	×	×	×	×	×
		Hot modification	×	×	×	×	×
		Version compatibility	×	×	×		×

(Continued)

Benefits	Description	Features	Web-Sphere	BEA	Oracle Application Server	Tomcat Server	Sun Java System Application server
Production-ready reliability	Ensuring that applications are up and meeting customer needs	Can be used with most H/W load balancers	×	×			
		Web application failover to backup database via JDBC connection pooling	×	×	×	×	×
		Web application load balancing via JDBC connection pooling	×	×	×	×	
Enterprise strength reliability	Adds premium clustering capabilities to enable automatic failover, replication and additional load balancing schemes	In-memory replication of EJB state	×			×	
		In-memory replication of servlet session state	×	×		×	
		Stateful session EJB failover	×			×	
		Clusterwide JNDI naming service	×	×		×	
		Two-phase deployment of clustered applications	×	×		×	

(Continued)

Benefits	Description	Features	Web-Sphere	BEA	Oracle Application Server	Tomcat Server	Sun Java System Application server
High performance	Makes applications run fast	Web caching (static HTML, Servlet, JSP, full page, URL-based, file type-based)	×	×	×	×	
		JSP tag caching	×	×	×	×	
		Premium EJB caching with multicast update and invalidation across clusters	×			×	×
		Thread pooling, Connection pooling, Multipools	×	×	×	×	×
		RMI optimizations		×	×	×	×
		Performance packs	×				
Manageability	Simplified and granular application management and monitoring capabilities	Web-based console, providing monitoring and configuration of instances, resources and applications	×	×	×	×	×
		Wizard-based domain and cluster configuration	×	×		×	×

(Continued)

Benefits	Description	Features	Web-Sphere	BEA	Oracle Application Server	Tomcat Server	Sun Java System Application server
		Administration of replication groups, 'single-image view' cluster administration		×		×	×
		Extensible—customers can build or customize their management console	×	×	×	×	×
		Programmatic management via JMX		×	×	×	×
		SNMP support	×	×	×	×	×
Security	Controls access and provides authentication and authorization services	HTTPS, SSL, JAAS, X.509			×	×	×
		Pluggable security modules for authentication, authorization, auditing and PKI management	×	×	×	×	×
		Built-in LDAP-based data store for all profile and entitlement data	×	×	×	×	
		Roles-based, dynamic-rules-driven access authorization engine		×	×	×	

(Continued)

Benefits	Description	Features	Web-Sphere	BEA	Oracle Application Server	Tomcat Server	Sun Java System Application server
		Graphical security policy administration editor	×		×	×	×
		Automatic conversion of existing access control list (ACLs)	×	×	×	×	×
Web services	Distributed application functionality offered as a service over the Web or intranet	Support for SOAP, XML, JAX-RPC, UDDI and WSDL	×	×	×	×	
		Asynchronous, high performing and secure Web services	×		×	×	×
		Interoperable with .NET, MS Web Services tool kit	×	×	×	×	×
		Deploy Web services, deploy Java classes and EJBs as Web services	×		×	×	×
Enterprise messaging	JMS messaging	Point-to-point, publish-subscribe, durable subscriber, XA compliant, XML messaging	×		×	×	×

(Continued)

Benefits	Description	Features	Web-Sphere	BEA	Oracle Application Server	Tomcat Server	Sun Java System Application server
		Asynchronous data processing with message-driven beans	×		×	×	
		Plug and play with third-party messaging providers, including MQSeries	×		×	×	
Premium enterprise messaging	High availability and scalability message infrastructure	JMS functionality distributed across clusters, highly available connection factories and destinations	×			×	×
Distributed transaction management	Transaction management maintains consistency of user data and enforces concurrency rules to access different system components	Two-phase commit protocol support. Leverages core BEA Tuxedo transaction expertise	×		×	×	
		Advanced transaction services including fault tolerance and transaction monitoring	×		×	×	×
		Transaction recovery service	×	×		×	×

(Continued)

Benefits	Description	Features	Web-Sphere	BEA	Oracle Application Server	Tomcat Server	Sun Java System Application server
Built-in Web server	Provides manageability with a fully integrated Web server and servlet container	HTTP, Servlets, JSP, CGI, etc.	×	×	×	×	×
		Plug-ins for Apache, IIS, iPlanet	×	×	×	×	×
		Virtual hosting	×	×	×		×
System integration	Compatibility and interoperability with most major hardware, software, protocols, environments and architectures	J2EE connector architecture	×		×	×	
		Bi-directional integration	×		×	×	×
		Interoperability with Microsoft COM+	×	×	×	×	
		Interoperability with CORBA through IIOP	×	×	×	×	
		Pluggable messaging infrastructure allows integration with other JMS and non-JMS messaging solutions	×		×	×	

(Continued)

Benefits	Description	Features	Web-Sphere	BEA	Oracle Appli-cation Server	Tomcat Server	Sun Java System Appli-cation server
		Pluggable security allows integration with other security solutions	×	×	×	×	
Heterogene-ous platform support	Ensures port-ability of your application and access to existing systems	Certified on most major hardware and operating sys-tems including Unix, Linux, Windows and mainframe	×	×	×	×	

Udaya assembled all the team leads at the end of the week. Everyone believed that the project might be over at the end of two months, well before the specified time of three months.

As they were trying to integrate, an array of problems surfaced. There were many interface mismatches between the modules and among the modules. Many of the COTS products do not function as expected when integrated with other modules, but they functioned when tested in isolation.

When tested, algorithms and code inspections did not matched with the function that was to be performed after integration. They found it difficult to adopt the reusable component to suit the requirement.

Chandra suggested that the technique of wrapping—which modifies the reusable components to suit the current application by re-writing the code internally—might be followed. She even explained the technique using the example of resource provider (see Exhibit D).

Though the team tried to develop wrappers for the components, they were not successful as they clearly lacked the experience of developing wrappers. The team found that they lacked well-defined infrastructure to assemble the components into systems. The system lacked interoperability and portability. All were at their wit's end about where they really went wrong and were perplexed. Udaya was pretty much confused but did not lose hope. She asked the team members to re-align the CBD model and work on a systematic method to integrate the components and then to build the application as a whole.

Exhibit D Resource provider

Resource provider is given as an example for the adaptor or bridge that was created and used.

Resource provider is module that connects all enterprise systems (comparable to Java Connection Architecture), such as SAP, Iris, Sybase. Connecting to each of the legacy systems could be painful so they designed an adopter that gives a uniform interface.

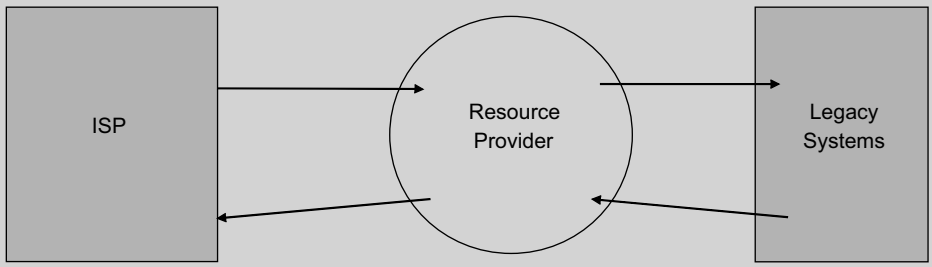


Figure 6.6 Resource provider sketch

Resource provider is an external system interface adapter developed for ISP to retrieve data from the legacy system. Legacy systems were considered to be potentially problematic by many software engineers for several reasons. These systems were often hard to maintain, improve and expand because there was a general lack of understanding of the system (Figure 6.6).

So the resource provider is used to retrieve data from such difficult data systems.

Key features of Resource provider

- Based on industry standard technologies, such as XML, XSL
- Separation of developer roles
- Ease of development, debugging and deployment including local development and debugging, logging, dynamic reconfiguration
- Overall architecture is based on separating the model (XML), view (XSL) and the controller
- Linkages between system components are loosely coupled
- Components are independent and replaceable

Roles

The overall framework is designed to separate roles based on knowledge of various tiers and roles. This allows work and activities to proceed in parallel and allows different skill sets to contribute to the overall solution without requiring everybody to be trained in everything. The roles include the following:

- Enterprise developers
- Presentation
- Assembler

Testing and development

Provides the ability to develop in the local environment on all the tiers (ABAP, Java on workstation, XSL). Provides extensive logging implemented using Apache log4j package with logging levels configurable on a per class/package basis.

Data integration

Resource provider uses message-oriented semantics to get the relevant data and data transfer is done in all parts of the system by XML and generation of XML is done by the back-end. Transformations and caching are implemented on the front-end resource provider layer.

XML generation

XML documents are generated at the back-end. Resource provider will send the request to the back-end. The back-end will generate the XML by executing the stored procedures based on the input. Additional capabilities of this model are ability to provide pagination and caching.

XML at Java layer

- RP provides another facility to generate the XML using XMLHelper class based on the plain data from the back-end
- XML generation is based on only string and not on DOM objects

High-level classes in resource provider (Figure 6.7)

ResourceProviderFactory

getResourceProvider(String Service)

ResourceProvider(interface)

getReader(String qStr)

ResourceProvider implemented by

1. ResourceProviderFile — For file access
2. ResourceProviderProjectDB — For Project DB access
3. ResourceProviderSAP — For SAP API access
4. ResourceProviderSybase — For sybase access

Sample usage of resource provider

```
//get the factory
ResourceProviderFactory factory = new ResourceProviderFactory();
//set the resource string. ** SAMPLE ONLY **
String resource = 'SAP:IN_FNAME=Z_XML_BAPI_GET_COMPANY_CODES';
//get a provider
ResourceProvider rp = factory.getResourceProvider(resources);
//get the reader
Reader isr = rp.getReader(resource);
```

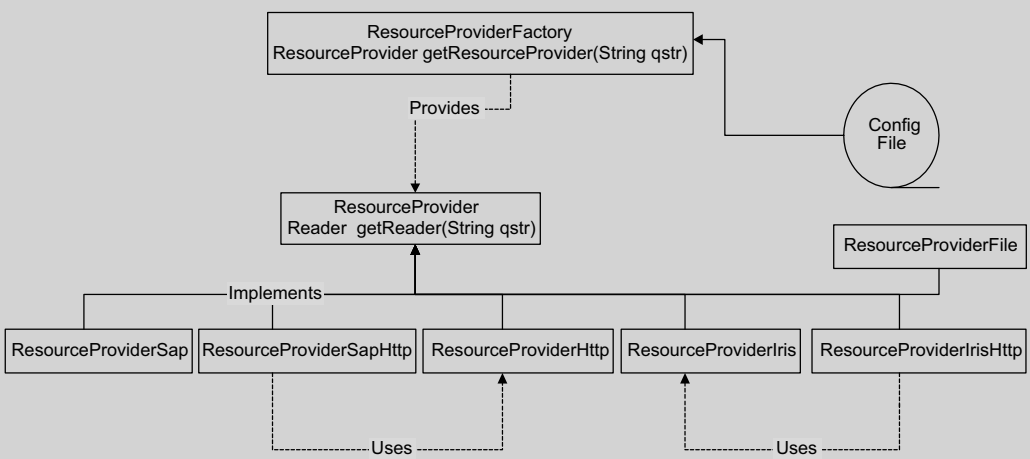


Figure 6.7 High-level classes in resource provider

New resource provider implementation

- Create a class that implements the interface *ResourceProvider* and also extends *ResourceProvider-Abstract*
- Implement the getReader(String) method
- Use the string to make decisions like which business object to call and what method to call
- Business object then calls the DAO object that will request for a connection from a ConnectionProvider(need to be written)
- Call the stored procedure which returns string (xml)
- Construct StringReader reader = new StringReader(xmlString)
- Return the reader

More Issues

It is said that 'CBD is all about building complex systems by reusing existing components rather than re-inventing the wheel'. It is important to understand the pros and cons of CBD as inferred from this case.

The project team was methodical in requirements gathering, analysis and designing the architecture. However, we understand from this case study that something went wrong. It is important to understand where the team went wrong and also come up with suggestions about how this should have been done.

It is believed that 'domain engineering in CBD will become as important as software engineering over the next decade'. Do you think that the project team has followed good domain analysis? Suggest how you would have advised them to go about systematically in carrying out the domain analysis process.

The success of CBD partially lies in properly selecting the reusable component. Do you think the team had done it right? What factors do you think are vital in selecting the components for reuse from the following:

- Already existing projects?
- COTS market?

POSTMORTEM

This case study brings out various CBD issues and gives some solutions. Unlike previous case studies where we have left many details for postmortem and case analysis sections, in this case study we have discussed solutions for various problems in the case study itself. For example, the widgets required at various stages of the SDLC (Table 6.1), test cases (Exhibit B) and evaluation matrix (Exhibit C) for some components were presented in the case study itself. We also discussed the development of an adaptor called 'resource developer' (Exhibit D). To solve the final problem given in the case study, we need to discuss more aspects of the CBD. We first discuss the CBD approach, and how it differs from the conventional waterfall model. We then present some characteristics of the CBD and discuss success factors and challenges of adapting the CBD. With this background the reader is advised to revisit the problems posed in the context of a given case study and come out with an answer. We will not be explicitly discussing the solution for this case study as it is more of a synthesis of exhibits and concepts already discussed. More details on relevant concepts are discussed in the remaining part of the chapter.

THE COMPONENT-BASED DEVELOPMENT APPROACH

There are two sides of the CBD approach:

- System development with components
- Component development

It is very important to understand that these two are different.

System development with components focuses on the identification of reusable entities and relations between them, beginning from the system requirements and from the availability of components already existing.

Component development is the development of the components that are then available for others to use. So component development needs the 'producer's perspective' while system development with components needs the 'consumer's perspective'.

How Different is the CBD Approach?

The choice of the CBD approach is based on a desire to reuse existing solutions, which must be planned.

It needs to be understood and remembered that unless the organization emphasizes reuse as an approach to be aggressively pursued by developers, it is hard to get tangible benefits from this. The approach needs to be multi-pronged:

- *Emphasize the reuse mantra*: Make all the developers aware that they should reuse whenever possible.
- *Emphasize the need for good documentation*: The CBD approach depends on not having to tweak the black box that should 'just work'! For that to happen, the team developing that black box needs to provide documentation of all sorts, use cases, supported platforms, dependencies, and so on. A portal needs to be provided for hosting these documents. They should be easily searchable.
- An especially successful module needs to be nurtured further, with periodic bug fixes, upgrades, and so on. Provide adequate resources to maintain and support the component.
- The organization also needs to realize that management and development issues are different for CBD and COTS systems.
- CBD orientation is required at every stage in the SDLC.

This approach is quite different from the conventional SDLC, where once the inputs have been taken from the end user every thing else is developed in-house. This leads to what is called the 'not-invented-here' syndrome, where people try to write/test/deploy every part of the solution all by themselves. This is especially expensive when the problem in question is well defined and previously solved by someone else (in a different team in the same company or outside the company).

The entire process of system development—requirements, design, implementation, testing and maintenance—will undergo many radical changes when a system incorporates multiple COTS components.

By committing to just one part of the 'new paradigm' (i.e., simply buying some commercial products), it is vital to realize that this will usually demand a commitment to all parts. This impacts the SDLC significantly. Compare the traditional waterfall model of SDLC with the CBD model (Figures 6.8 and 6.9).

We have discussed in the case study as to how CBD differs from the normal SDLC. Here, we provide more details on the entire CDB life cycle process. We are aware that there are six major stages in

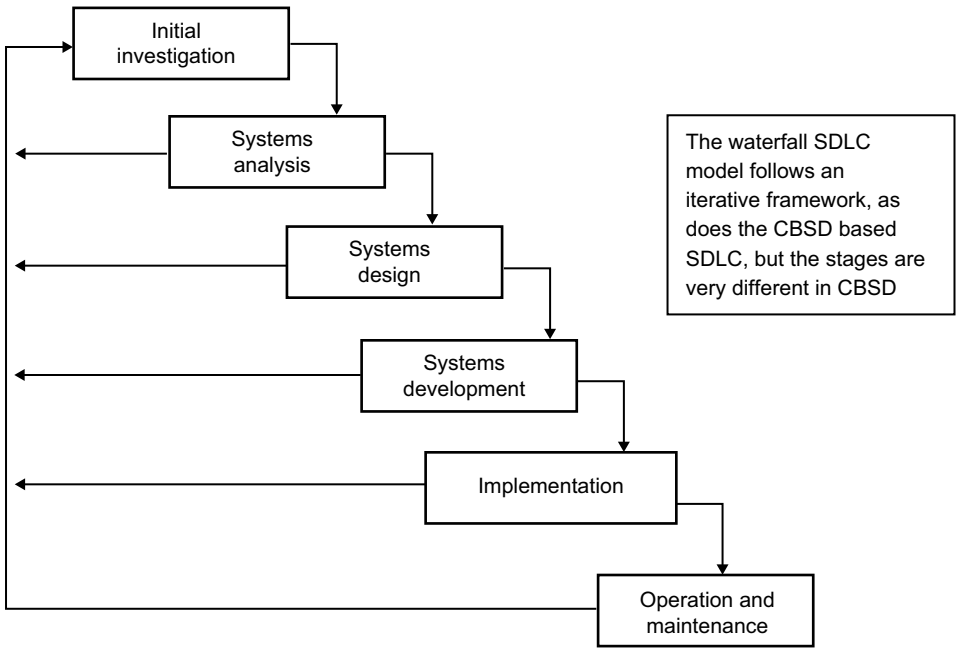


Figure 6.8 Traditional waterfall SDLC

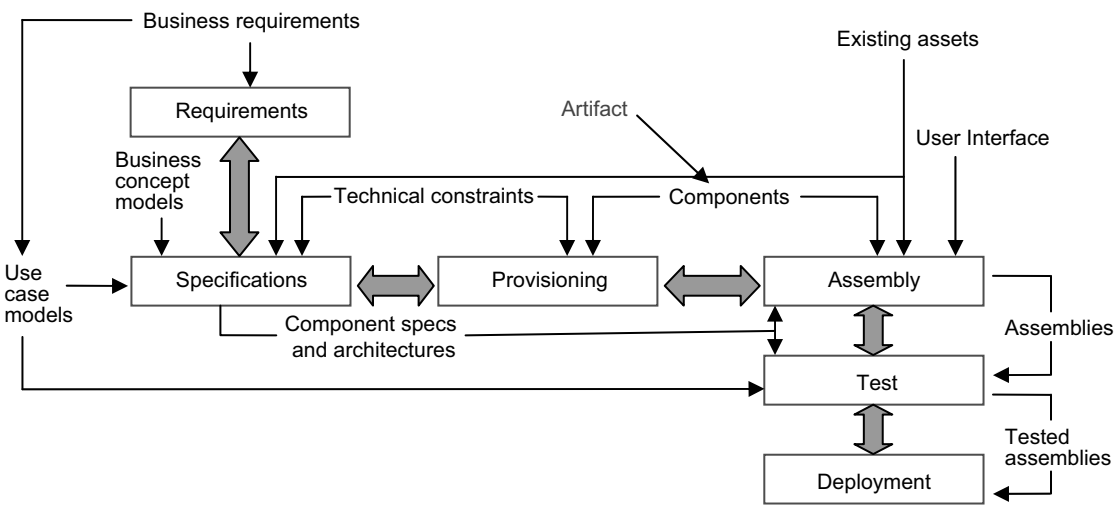


Figure 6.9 Process in CBD SDLC

the CDB lifecycle: requirements, specifications, provisioning, assembling, testing and deployment. We have described these stages and tried to understand the Universal Bank case study in terms of these stages (see Table 1).

Once the business requirements are captured, they are fed into the requirements phases. During the requirements phase, these are analysed thoroughly so that they can be transformed into more concrete and quantitative specifications later on in the specifications phase. The business concept models also help in the specifications phase. In addition, the business requirements are also used to create use case models to represent the requirements in a more structured manner. These use case models help in the specifications phase as well as the testing phase. Based on the platform we choose, there may be certain technical constraints that influence the decisions being made in the specifications phase. The same constraints can also influence the provisioning phase. At the end of provisioning phase, there are a set of candidate components that are ready to be assembled in assembly phase.

The specifications phase is responsible for producing the specifications of various components that are required to achieve the functionality of the proposed software system as well as the overall architecture. These component specifications and architecture will be required in the assembly phase as well in the testing phase.

During assembly, one common practice is to look at the existing assets, which are already available components available in the organization's component repository, and see if any of them can be useful. Sometimes, based on the existing assets, the specification of the system can be changed to make the system cost effective. It is also important to note that this should not change the requirements. Here, it is very important to understand the difference between the requirements and specifications. You are only changing the means of achieving the functionality but not the functionality of the system. In the assembly phase, the user interface issues are sorted out and the proper user interface components are built glued together with other components.

During the assembly phase, several possible and promising assemblies of components are worked out with different combinations of candidate components. It is a common understanding that unless various combinations are tried out, we will not know which combinations gives best results.

There are four major activities that characterize the CBD approach from the consumer point of view:

- Component qualification
- Component adaptation
- Component assembly
- Component evolution

Component qualification: It is important to have a process in place that will help us in identifying the right components for the current need. This process cannot start when the need arises but starts when the CBD approach is adapted. When several components exist and as and when new components enter the marketplace, this process helps us in selecting the right components for a specific need.

Discovering components and *evaluating* them for future use are two major steps in the component qualification process. In the discovery phase, various properties of a component including component functionality, component reliability, predictability, usability and available interfaces are studied. It is also important to look at some of the 'non-technical' properties, such as who is the vendor, what is vendor's market share, what is the vendor's past, present and predicted business performance and what is the process maturity of the vendor. Though the evaluation phase is not as clearly defined, there are a few techniques such as the International Standards Organization (ISO). An ISO technique is applicable for any product evaluation. We have described one evaluation and selection criterion in Exhibit C (component selection matrix). Most mature organizations define their own evaluation criteria involving one or more of the following

approaches: paper-based studies of the components, discussion with other users of those components and benchmarking using a pilot study or prototyping.

Component adaptation: When components are created, one major criterion is the coverage—how generic is the component and the wide range of applications in which this component can be used. However, when these components are used in a specific context, coverage criteria do not always help because we want the component to work in this specific context in an optimal manner. Hence, components must be adapted or re-interfaced to ensure that there are minimum conflicts among components. We can use the white box, grey box or black box approach based on the degree to which a component's internal structure is accessible.

The black box approach can treat the component as a binary executable; the grey box approach plays with various interfaces of APIs provided by the component; the white box has access to source code so that the component can be completely re-engineered to suit the current context. However, note that the flexibility offered by these components is inversely proportional to the ease of maintenance; that is, if you rip open the component and make significant changes, you are possibly affecting its maintenance and robustness severely.

Component assembly: Once all the required components are selected, we must have an integration platform that binds all the individual components to form a proposed system. Many architectural styles can be used to create this platform. An integration infrastructure can also be used that makes use of a database as a centralized controlling unit for all operational data coming from various components. Even more sophisticated architectural styles such as black board architecture or message bus can also be used. Platforms such as common object request broker architecture (CORBA), J2EE or .Net platforms provide such infrastructure.

System evolution: Component-based systems are relatively easier to maintain or upgrade because the unit of change is a component. Ideally, when a component goes bad, to repair an error or to enhance the functionality of a component, all that we need to do is to swap another equivalent or enhanced component. Here, the assumption is that the components are pluggable units. This is a popular view but at the same time is a very optimistic and simplistic view of system evolution. In practice, replacing a component with another component can be a very difficult task as the new component may provide very different interfaces. Once the interfaces are adapted, a thorough testing needs to be done to make sure that the new components work well with the rest of the system.

Developer (Producer) and Consumer Perspectives on CBD

It is important for the producer of a component to keep in mind a few important aspects while building components, which include business goal, functionality of the proposed component, the maintenance policy, expected lifetime of the component and the application in which this component will be used.

The business goal of the overall system is important because it will help in identifying the exact problem to be solved keeping in mind the customer needs as well the how it will contribute in increasing the revenue for the customer.

The functionality of the component is expected to be clearly defined along with the scope of the component as well as all the component features that need to be supported. In addition to the functionality aspect, the maintenance policy should also be in place. A typical maintenance policy takes into consideration aspects such as who will be the users of this component and how are they are going to use it, what kind of APIs need to be provided, how often should the component releases be made, what are the release dependencies, how many developers are needed to make sure the release cycles are maintained, and so on.

Components work with other components and depend on other components to provide the functionality of the system. The characteristics of other components such as the life expectancy, release cycles and their dependencies will matter in making design decisions while building a component. Finally, the environment in which the component will be used needs to be taken into consideration. This includes the operation systems, hardware configuration, whether the component will be connected over the network or in the standalone machine along with other components, whether it will be placed on the server or on the client side and what would happen when the other components are replaced with newer versions.

As we are moving from object technologies (object-oriented languages such as SmallTalk, Java, etc.) to component technologies (J2EE, .Net), the landscape of software development is undergoing significant changes. Most of the current CBD platforms provide an easier way to package object implementations and make the use of these components convenient. Hence, creators of components should be aware of the need, advantages and disadvantages of creating components in specific platforms.

From the component consumer perspective, as the given component is going to be used in the context of building a bigger system, it is important to consider the life cycle of the component in relation to the life cycle of the overall system. There are several concerns in this context:

- *Release cycle of the component*: Can the component producer will be able to deliver newer versions in synchronization with rest of the system?
- *Behaviour of the component*: If there is a change or replacement of the component, how will that affect the other components in the system that work with this component?
- *Compatibility of the component*: If the component cannot be changed or a new version is not available, and rest of the system has to be changed, will the existing component be compatible with the new system and other components?

Why CBD—Cost and Use

The essential reason for a CBD approach is partially to save money. Constant re-inventing the wheel is costly in terms of people needed to maintain it, upgrade it, and so on. The cost is even more burdensome if the feature in question is not a core expertise. In such a scenario, it is a more optimal solution to find suitable components and use the same. Due to behaviour isolation, errors could be attributed to specific components. The overall maintenance costs would reduce through quick replacement of components without new code creation.

The more fundamental reasons for using CBD relate to issues such as rapidity of system deployment, timely maintenance and ease of modernization, all of which are vital for a modern computer-based system. Reuse can lead to reduced time to market due to major efficiencies and economies of scale.

However, immediate cost savings from CBD may in many cases be minimal or illusory. It may even be true, especially in the short term, that a shift to use of CBD can lead to higher costs. This is often due to

- Cost of buying COTS components
- Deployment-related issues

From the consumer's viewpoint, the benefit of moving to the CBD model lies in the fact that if they can identify a component that works for them on a given platform + hardware for a given set of users in a specific setup, someone else (the owner of that component) is guaranteeing compatibility, bug fixes, upgrades, and so on.

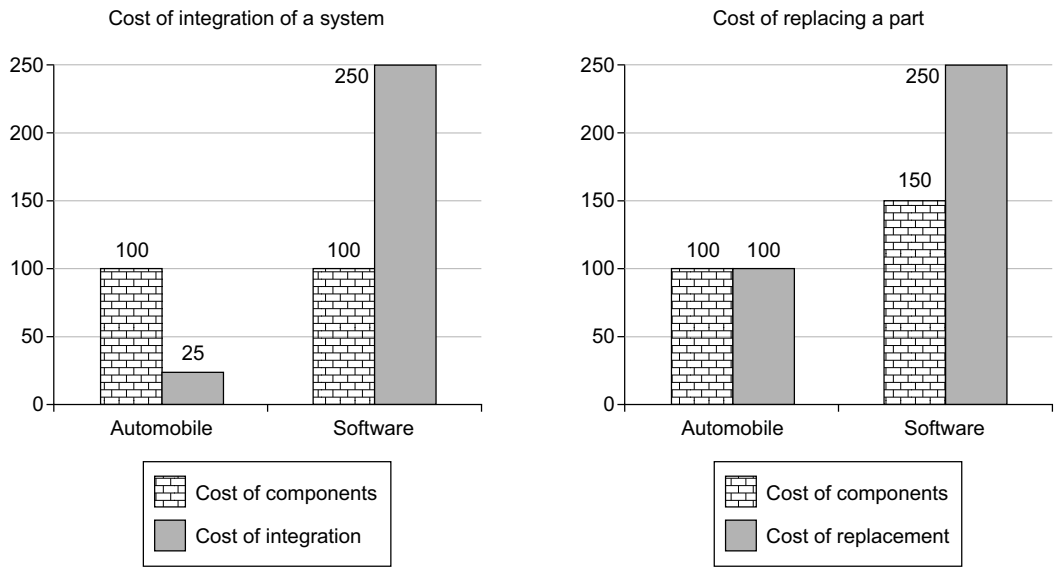


Figure 6.10 An illustrative visual for commonsense observation (and not based on any scientific survey). Source: http://cbsdf.com/ps_blog/plug-n-play.htm

Especially, if either the commercial COTS approach or the company's internal component approach is selected, these guarantees are more solid. This significantly reduces the total cost of ownership for the consumer (Figure 6.10).

From the producer's viewpoint, if a customer commits to a component being developed, then it will mean steady support revenue and often being able to convince them for an upgrade.

SUCCESS FACTORS OF CBD

As has been mentioned earlier, the success of CBD depends on how applications, systems and components are purposefully designed and structured to ensure that they fit together and that they cover the needs of the family or domain.

A well-defined, layered, modular structure and various design and implementation standards (such as UML and pattern catalogs) will be of great help.

The premise of CBD lies in supporting low coupling and high cohesion, leading to better software architecture. The very fact that you have to make components automatically results in low coupling and high cohesion, which makes better modularization and hence better systems that are easy to develop and maintain.

Another feature of CBD is layered component infrastructure. Each application is represented as built using lower-level component systems such as applications, business-specific components, middleware components and system software components.

The layered architecture and component infrastructure provide components and a roadmap to help each person in the organization understand and apply the desired engineering practice. The component architecture supports platform-independent interfaces, providing openness and flexibility. Different implementations of the same interfaces can be plug-in compatible.

CHALLENGES TO ADOPTING CBD

Component mismatch happens to be the single most difficult change to circumvent when it comes to adopting CBD. The software market is filled with options when it comes to components. The real problem lies in identifying a set of components that each fulfils a specific task for you and gels together well enough to be industrially deployed.

Sometimes, these choices are well known. For example, the combination of LINUX, Apache, MySQL and Perl is a tried and tested one. Sometimes, they are not. For example, what open source implementation of Java Messaging Service goes well with Oracle Ebiz suite?

In such scenarios, it takes a lot of experience to be able to identify the set that will work for you. The problem increases in complexity if you have already integrated your applications with the components and then some change at the vendor end breaks your code.

Inconsistent Component Integration

We may often run into situations where a particular version of a given component is specifically needed to be compatible with some other component. This is often going to prevent you from leveraging the latest features of the latest versions of that component.

As shown in Figure 6.11, product P makes use of components A and B. Component A in turn makes use of component B. When product P moves from version V1 to V2, components A and B are also moved from version V1 to V2. In this scenario, it is likely that product P has upgraded its components but version V2 of component A did not upgrade component B. This results in inconsistent component integration. Thus, this scenario can prevent your product P from leveraging the new features in component B by forcing you to stick to version V1.

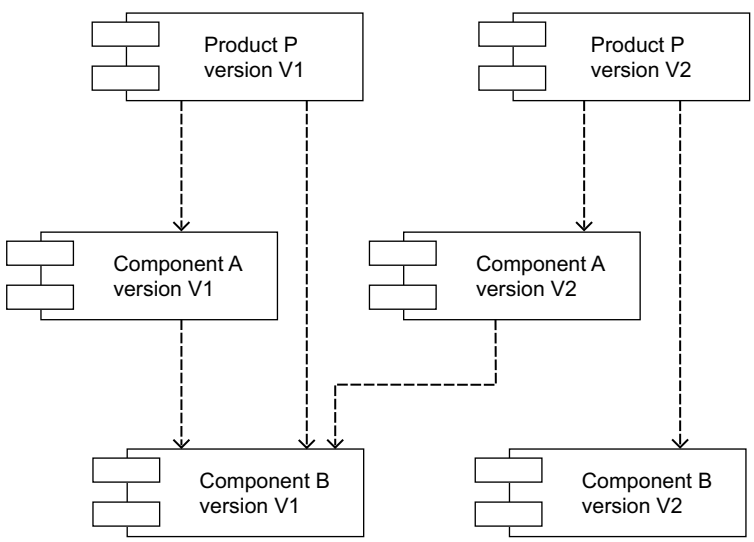


Figure 6.11 Component A depends on version V1 for B

Component Mismatches

This can happen when two separate components selected for integration are not compatible with each other.

This can be due to architectural reasons (Apache worked better on Linux for a long time than it did on Windows!). Anyone attempting to integrate two different components with different architectural requirements can run into situations where each works well standalone but behaves badly when coupled together.

This can also be due to interface mismatch, wherein two components may choose to provide interfaces that have gaps between them. An image editor may allow only jpeg files to be modified, while the Web hosting applications could support more file types. Trying to club these two may not succeed, since the image conversion routines are missing.

There are several ways to deal with architectural mismatches when using CBD. First of all, one can take care of avoiding mismatches, then employ techniques to detect the mismatches if any and finally repair the mismatches when found.

How to avoid mismatches

An architect can take care from the beginning to avoid mismatches. Obviously, this is most economical way of dealing with the problem. Conventional ways of dealing with this problem is to design interfaces and make them available by stating all possible assumptions. This is a disciplined approach but is not practical in many occasions. Some component systems are based on layered interfaces. That is, the inner components have fixed interfaces that talk to outer components. As we move towards outer layers, the flexibility, sophistication and the number of interfaces increases. The inner interfaces are typically private interfaces and the outer ones tend to be more public.

What to do in case of a mismatch

In case a mismatch is detected, the integrating team needs to evaluate their options.

It is called the 'avoid–detect–repair' cycle. If they have alternative components that will plug in better, they should go for them. This is called the avoid phase. In case replacement is not an option, you need to identify the exact cause of mismatch. This will mean understanding what pieces of the interface are missing, which component expects data in which form and how can you mitigate the missing pieces.

Lastly, you have the repair phase, where you must code the missing pieces yourself.

Techniques for repairing mismatched interfaces

The following can be used for repairing mismatches

- Wrappers
- Bridges
- Mediators

Wrapper: For a given set of interfaces of a component, if none of them are suitable we can build wrappers for the available interfaces so that the client accesses the component through this alternative interface. The steps involved in creating wrappers are as follows:

- Translating an element of a component interface into an alternative element
- Hiding the element of a component interface
- Preserving without change an element of a component's interface

Bridge: This will translate between some requirements of an arbitrary component to provide some assumptions of some other arbitrary component. The main difference between a wrapper and a bridge is that a bridge is independent of any particular component and must be explicitly invoked by an external agent and it is usually a transient process. For example, a bridge that translates from postscript to PDF or vice versa (this bridge is independent of components).

Mediator: The properties of both bridges and wrappers can be found in mediators. They incorporate planning function that in effect results in run time determination of the translation. A mediator becomes an explicit component in the architecture. Mediators are like bridges with planning logic. For example, intelligent data fusion (like sensor that generates high volume of high-fidelity data, which need processing at run time), or if there are bridges from say data format D0 to D1 and from D1 to D2, one can write a mediator that translates from D0 to D2 (as in Unix pipes and filters).

Conclusions

CBD is here to stay and is a way of life for many software developers. Companies will gradually need to evolve by recognizing this fact and adjusting and adapting themselves to the same. This will involve cultural changes, a greater understanding of market dynamics, legal issues, newer tools and technologies, corresponding measurement metrics and product certification methodologies.

It is a fact that the current state of software management is inadequate. As competition in software markets increases, companies will be driven to adopt CBD. Companies will realize that they need to invest in CDSB or risk losing their competitive edge. These investments will include funding projects, training (including special degree programs in component-based SDLC at core universities for working managers) and development.

Creation and adoption of a component marketplace is not going to happen only because of the efforts of technical people, but will be created by all sets of people involved in delivering software to the market. In such a marketplace, each of the component models (EJB, COM+, CORBA, etc.) will co-exist. The key would be strict adherence to standards and differentiation by the features they add to a commonly accepted core standard.

Looking at different kinds of components (very small to very big), the marketplace seems to support two kinds of components: components at the level of design patterns and slightly higher and larger platform specific frameworks (.Net and J2EE).

Measurement and metrics will focus on an empirical approach to the practice of CBD. Standards will emerge in developing, interfacing, documenting and third-party evaluations (certification). ANSI/UL 1998 is an example of such a standard (e.g., the document that describes risk analysis).

It is also important to mention here about an initiative by the Object Management Group (OMG) called reusable assets specification (RAS), which tried to come with a standard for component repositories to specify the reusable assets. This initiative was then partnered by Rational Corporation (now merged with IBM), IBM, Microsoft and ComponentSource. RAS defines various types and categories of reusable assets and how these reusable assets should be specified and packaged. It uses UML and XML as related and dependent standards. Core RAS and Profiles are the two major categories defined by RAS. All fundamental elements of reusable assets are defined as Core RAS, and Profiles are the extensions of Core RAS elements. RAS enjoyed attention of the community but unfortunately after 2005 it has not made much progress.

In CBD, the quality of the overall system needs special attention. The quality of the entire system needs to be maintained even if the system is built using mostly COTS components. Since the design, implementation and interaction mechanisms of individual components used are not under the control of the architect, the quality practices in CBD needs to be different. The requirements engineering process needs to be more flexible in CBD. We need to identify critical requirements and constraints and other requirements and constraints should be allowed to be more flexible to play with. It means modification of requirements to suit the available components. In addition, to control quality, several backup

options and multiple contingencies need to be worked out before hand as part of the CBD strategy. It means coming up with multiple assemblies and making use of the one that is most optimal with an option to get back to one of the other worked out assemblies.

Best Practices and Key Lessons from the Case Study

Keeping a couple of things in mind and adjusting your development process based on that can be very useful in getting things done painlessly.

- It is important to distinguish between component development and system development with components. The former is influenced by the perspective of the vendor, producer, or developer and the latter is influenced by the perspective of the consumer.
- Saving money is only the partial reason for taking the CBD approach. The more important reasons to use CBD are rapid system deployment, easy maintenance and extensibility. The immediate cost savings from CBD may in many cases be minimal. It is even possible that, in the short term, a shift to use of CBD can lead to higher costs.
- It is important to focus on low coupling and high cohesion while designing the components.
- Be on high alert while designing the interfaces between the service components and domain components.
- In layered architectures with CBD focus, it is important to make sure that components in the lower layers are also pluggable. It means that when a component in a lower layer needs to be replaced, the rest of the system should not fall apart.
- CBD orientation is required at every stage in the software development life cycle.
- Parameterized and negotiated interfaces avoid component interface mismatches and hence they need to be considered while designing the architecture.
- When an architectural mismatch occurs, choose a wrapper, a bridge or a mediator in an appropriate manner without taking a hit either on performance or on flexibility.
- It is important to know the stability of the component vendor, not just the technical sophistication. Sometimes, non-technical criteria overweigh the technical criteria in selecting the components.
- The CBD approach demands that the systems are designed and engineering to accommodate the component marketplace imperatives.
- It is good to define separate roles for scouting for right components in the marketplace, for independently evaluating the components and for system integration. If all these roles are played by the same person or team, it is possible that either inefficiency or bias will creep into the process.
- Practices for qualifying available products as components that are 'fit for use' in a particular system. This usually means being constantly vigilant about the dynamics of the software industry.
- Techniques for engineering and designing software so that market-supplied products can be integrated into the system more predictably and with reduced risk of system failure, performance reduction or increased costs. This means that engineering and design techniques will allow qualified components to be adapted, assembled and updated easily. These techniques usually involve using architecture, infrastructure and middleware.
- A technique (framework) for evaluating technology so that managers acquire emerging technology at the optimal stage and retire products based on eroding technology before the erosion has a negative impact on the system and the organization.

❖ A system for acquiring products and managing the organization to maximize benefits of CBS practice by sustaining technology component qualification, evaluation and improved engineering and design practices throughout the organization and across many systems.

Further Reading

There are a few good reference books on component-based development. *Component Based Software Engineering—Putting the Pieces Together* by George T. Heineman, William T. Councill (editors) published by Pearson Education (Addison Wesley) in 2001 is a great source of very important articles on CBD. Another important book is *Objects, Components and Frameworks with UML—The Catalysis Approach* by Desmond Francis D'Souza and Alan Cameron Wills published by Addison Wesley in 1999. *Realizing eBusiness with Components* (by Allen), *UML Components: A Simple Process for Specifying Component Based Software* (by Daniels Cheesman) and *Component Software* (by Clemens Szyperski) are also good books published by Addison Wesley.

There are many good articles available on the Web to get insight into component-based development. It will be a good idea to read the articles at the following addresses and write a critique:

- www.devsource.com/article2/0,1895,2020837,00.asp
- www.joelonsoftware.com/articles/fog0000000069.html
- http://it.slashdot.org/it/06/09/27/2111214.shtml
- http://cbsdf.com/ps_blog/plug-n-play.htm

One can gauge the size and variety of the component marketplace by checking out the Web site of ComponentSource (www.componentsource.com/index.html). This portal also contains information regarding CBD.

Software Engineering Institute of Carnegie Mellon University has very useful information on component-based software development. See www.sei.cmu.edu/str/descriptions/cbsd.html for more information.

Reusable Asset Specification, OMG Available Specification, Version 2.2, formally released in November 2005 can be found at www.omg.org/docs/formal/05-11-02.pdf.

Chapter 7

Emerging Trends in Software Architecture

Software Architecture Discipline—Past, Present and Future
Reusability and Reusable Services
Service-oriented Architecture
Dimensions of Future Software Architecture
Critical Software Architecture Elements
Conclusions
Further Reading

Contributor
Bhudeb Chakravarti

Software architecture is a fast-growing discipline. In this book, we have looked at various aspects of software architecture through a set of case studies. We have seen how software architecture is only paid a lip service in many projects. We have tried to distinguish between marchitectures and architectures. We have seen how a given architecture should be re-factored so that many of the performance and other non-functional issues can be fixed. We then looked at architectural evaluation in detail. We have also seen how to move from software architecture to software design. In Chapter 6, we discussed about how component-based software engineering helps us create faster, better and cheaper software.

The topics we have discussed in this book by no means cover all important aspects of software architecture. In fact, software architecture has become a very important sub-field of software engineering in the recent past. There is a lot of activity in this area in terms of development of new techniques, tactics, frameworks and theories. It is hard to keep track of the progress.

In this chapter, we discuss some important trends in software architecture. We look at the past, present and future trends of software architecture before discussing some of the emerging trends in detail.

There is a transition of IT systems from being backroom operations to becoming business critical. Due to this transition, there is a criticality of service management. One major trend in software architecture is service-oriented architecture and its related technologies such as Web services, enterprise service bus, software as a service and grid computing.

The field of software architecture is at present at a crossroad, wherein it is important to redefine the dimensions that we should consider to take it to a new peak. It is important to understand the dimensions of future software architecture. We will have some discussion on some of these dimensions in the later part of the chapter.

If we need to bet on one aspect of future aspects of software architecture, it will be on reusability at various levels of software development. Exploiting, employing and enjoying the benefits of reusability in software services industry is a major promise we see in the coming future. In our opinion, adapting software product lines (we would like to call them as software service lines) by the software services industry will be the most interesting and important change one can anticipate in the coming future.

SOFTWARE ARCHITECTURE DISCIPLINE—PAST, PRESENT AND FUTURE

Since the late 1980s, software architecture became the most important part of a software development life cycle. Starting from the broad organizational structure of the system, software architecture, at present, encompasses a wide set of notations, modelling languages, tools and a number of analysis techniques. The increasing use of object orientation in software development provided a base for using component-based architecture. The benefits of a built-in framework and reuse of components provided substantial incentive to use component-based architectures even when they are not ideal for the systems being developed.

A critical issue in design and construction of any complex software system is its architecture. A good architecture can ensure that a system will satisfy key requirements in areas such as performance, reliability, portability, scalability and interoperability. A bad architecture can be disastrous.

The first reference to *software architecture* occurred in 1969 at a conference on software engineering techniques organized by NATO (Kruchten *et al*., 2006). Some of the pioneers in the software architecture field, such as Tony Hoare, Edsger Dijkstra, Alan Perils, Per Brinch Hansen, Friedrich Bauer and Niklaus Wirth, attended the meeting. Since then and until the 1980s, software architecture mainly dealt with system architecture, that is, the physical structure of computer systems. The concept of software architecture as a discipline started to emerge in 1990. A 1991 article by Winston W. Royce and Walker Royce was the first to position software architecture between technology and process (Royce and Royce, 1991).

Even though articles were published in the open literature in the 1990s, software architecture and architectural styles remained under a veil of mystery. Garlan and Shaw (1993) first mentioned some architectural styles in their seminal paper. The authors mentioned in that paper that most systems typically involve some combination of several styles. There are 12 styles explicitly mentioned by the authors. Some of the architectural styles were extensively discussed in Chapter 1. The original list of architectural styles mentioned by Shaw and Garlan are the following:

- Layered
- Distributed processes and threads
- Pipes and filters
- Object oriented
- Main program/sub-routines
- Repositories
- Event based (implicit invocation)
- Rule based
- State transition based
- Process control (feedback)
- Domain specific
- Heterogeneous

Each of these styles does not describe a clear smooth continuum, that is, there is no clear set of attributes that enables each style to make definitive choices to distinguish it from other styles. Some styles focus on certain attributes that are not addressed by other styles. For example, the layered style is concerned with constraining the topology of connections, whereas the repository style merely says that a central repository is at the heart of the system.

The field of software architecture started to flourish in the mid-1990s. A number of professionals and academicians started publishing articles and books in this field. The first book on software architecture was by Witt *et al.* (1994). Then, Software Engineering Institute (SEI) published an article on software architecture analysis method (SAAM), the first in the SEI series of methods (Kazman *et al.*, 1994). Several software professionals in large organizations such as IBM, Siemens, Nokia, Philips and Nortel started research in this field and several approaches and design patterns emerged from their works. One of the notable approaches that became very popular was the 4 + 1 view model of architecture (Kruchten, 1995).

Since then software architecture has crossed a number of crossroads. The professionals working in this field collectively created the IFIP Working Group and the Worldwide Institute of Software Architects. New methods such as BAPO and ATAM emerged and consolidated along with SAAM (Clements *et al.*, 2002). Different architectural languages were introduced such as XML-based ADL that provide support for broad sharing of architectural models. Joining forces with the reuse and product families communities, software product lines became a sort of sub-discipline, attracting the attention of large manufacturing companies.

Researchers are at present working on model-driven architecture with UML, which has become a defunct standard. Some researchers are also working on architectural decisions, requirements for bringing architecture as part of the initial business model and testing of architectural models at an early stage of the development life cycle. Studies are also being to make architectural analysis a part of agile software development. Architecture of quality attributes and quality of architecture are going to become the most important factors in the coming years.

Next-Generation Systems

In the past, software architecture dealt with well-defined systems having well-defined interfaces for each of their components. In present scenario, due to the pressure of time to market, the components are being forced into the market without well-defined interfaces. System designers are also using a number of independent components either developed by them for previous systems or directly purchased from the market. The systems are also becoming more complex with diversified functions and stringent quality requirements. So it is time to replace the phrase 'time to market' with 'time to quality systems'.

At present, there is no common understanding or standards on quality attributes. Only the quality requirements described in the business model and the performance-related quantitative requirements are considered when assessing the quality of the system. The quality attributes of the domain-specific requirements are still not included in the evaluation of the system. Although there are methods for analysing specific quality attributes, these are typically done in isolation. It is thus important to identify and list out the quality of a system under development and analyse the quality attributes in an integrated manner. It would also be a focused area to define the quality attributes in a quantitative manner and demonstrate success in achieving them.

Qualities of system can be categorized into four parts:

- Runtime qualities
- Non-runtime qualities
- Architectural qualities
- Business qualities

Runtime quality attributes emerge from the execution of a deployed system. Quality scenarios of runtime attributes relate to specific instances of runtime execution. Some of the runtime quality attributes are performance, reliability, security, availability, usability, functionality and interoperability. These quality sets are most important for the execution of the system and should be given due consideration while architecting the system.

Non-runtime quality attributes relate to the development, deployment or operation of the system. Quality scenarios that cater to non-runtime attributes are expressed in terms of system lifetime activities. Some of the non-runtime quality attributes are reusability, maintainability, testability, scalability, integrity and configurability.

Other than system quality attributes, the architect should also take care of architectural quality attributes and business quality attributes. Architectural quality attributes relate to the quality of the architecture proposed for the system. The correctness or completeness of the architecture, whether the system can be built from the architecture being proposed and whether the architecture proposes an integrated theme are some architectural quality attributes.

Business quality attributes include the cost and benefit of the system. The marketability or usefulness in the organization should be considered while architecting the system. The rollout time to develop and release the system, the lifetime of the system and the requirements of integrating with legacy systems are some business quality attributes.

Software architecture encompasses the quality attributes of the system and brings about a standard way of achieving them. The traceability between architectural decisions and quality requirements will help developers to take care of the quality attributes in a systematic manner. Architecture will also enable to tune the quality attributes as per the requirements and ensure easy implementation. In the coming years, the inclusion of quality requirements in architectural design will become mandatory for complex systems.

Though much study is done on eliciting requirements based on the architectural needs of the system, not much work is done on defining the architect's role in identifying architectural requirements as a part of functional requirements. Thus, it is important to identify the strategy and methods to include functionalities into architectural components. It is also important to allocate quality attributes into architectural abstraction.

A number of research works are ongoing to create architecting methods for smooth transition from requirements to the architecture and finally to development. For example, Brandozzi and Perry (2003) have suggested a new process called 'Preskriptor', which is based on an architectural descriptor language. This process systematically ensures that the requirements related to architecture are being satisfied. Another method called component bus system and properties was presented by Egyed *et al.* (2001), which expresses requirements in a form that is more closely related to architecture.

In this method, the functional requirements are categorized based on various properties to group them in such a way to identify whether they should be implemented as components, bus or system properties. Liu and Easterbrook (2003) extended this method by introducing a rule-based framework that allows for requirements architecture mappings to be automated wherever possible. Liu and Mei (2003) viewed the mapping from requirements to architecture in terms of features, essentially a higher-level abstraction of a set of relevant requirements as perceived by users (or customers).

Another important aspect of architectural design is to consider risk assessment.

The strategy of including architectural requirements depends on organizational culture, global factors and the negotiation capability of the architect. It is important to have a holistic approach to consider the global issues such as the environment in which the system is being built, flexibility and rigidity of requirements, external technological factors, and so on. Architecture decision points should consist of tactics to resolve the issues related to the quality attributes of the system. Architectural patterns should be created and used to realize those tactics that satisfy the quality issues. There might be some conflicts or trade-offs between different architectural components or decision points, which should be resolved through effective negotiation.

REUSABILITY AND REUSABLE SERVICES

The major challenge of a software architect today is to improve the production rate of software systems. Reusable components are one of the solutions in this area. Though the original idea of component-based development was the direct fallout of hardware components, it could not reach the height of reusability that electronics engineering achieved through integrated circuits. One of the reasons was the simplicity of software components. Software components are considered as simple units in the complete system, and developers considered it much easier to develop them than integrate them with their other components. It has been observed that though some of the components are readily available, developers re-develop them as it is much easier to do so. They do not consider it as waste of time to write the component again as it takes a very little time to develop it. But if the application under development requires hundreds of thousands of such components, the total time taken becomes a deciding factor.

Reuse Maturity Levels Needed in an Organization

In his interesting book *Confessions of a Used Program Salesman: Institutionalizing Software Reuse*, Will Tracz (1995) discusses the reuse ratio in organizations at various levels of maturity (see Table 7.1).

Maturity levels	Reuse ratio	Requirements
No reuse	–20 to 20%	Business as usual
	Smart people	
Salvaging	10–50%	Smart people
	Depends on luck	
	Maintenance problems	
Planned reuse	30–40%	Reuse library
	Management support	
	Incentives	
Systematic reuse	50–70%	Reuse library
	Reuse process	
	Reuse metrics	
	Education	
Domain-oriented reuse	80–90%	Domain analysis
	Application generators	
	Architecture	

Table 7.1 Reuse ratio in organizations

If an organization does not plan for reuse, it can achieve a reuse ratio of around 20% by accident. This is typically done because there are smart people in the organization who are able to make use of the existing resources and components to come up with new systems. However, notice that the reuse ratio also can swing to the negative side (up to –20 per cent), because when people are not lucky they end up developing a system from scratch after investing significant amount of time on trying make use of existing resources.

If there is a previous project that is similar to the current project, many organizations end up making use of what was done earlier, that is, *salvaging* previous work. Depending on the similarity between the projects, the reuse ratio can vary between 10 and 50 per cent. This is wide range, which means that the luck factor plays a major role. Again, we need smart people who are able to see the similarity and are able to adapt to the new requirements. One major problem is that since this reuse is not really planned the projects at the end of their life cycle may require to heavy maintenance costs.

In more mature organizations, where there is some investment into *planned reuse* and management support, the reuse ratio is higher, varying between 30 and 40 per cent. These numbers are much more consistent when compared with the previous situation. These kinds of organizations typically maintain a reuse library and management provides incentives to those who follow the philosophy of reuse.

Systematic reuse is possible in addition to planned reuse if there is a process that makes sure that optimal reuse is achieved. This kind of process includes checking the marketplace continuously for new reusable components, various metrics relating to the components and pilot implementations to evaluate these components so that the users of these components have in-house knowledge about the strengths and

weaknesses of the new components. If an organization matures to have a systematic reuse in place, the reuse ratio can be very high—between 50 and 70 per cent. It means that more than half of the system is developed using already existing components. In addition to management support, we need to constantly educate developers about reuse.

The best reuse ratios are possible if organizations focus on building reusable domain components. The state-of-the-art software development almost mastered the reuse in software engineering aspects. Reuse in domain engineering is what we are headed for. If we achieve this we can get up to 90 per cent reuse ratio. Domain analysis, application generators and reference architectures are some of the prerequisites for achieving this kind of ratio.

We will discuss how we can move from lower levels of reuse ratio to higher levels. Though reusability has become a punch line for every IT organization, the cultural and business barriers to reuse are just too large. Despite the efforts of the IT manager, the architects and the development team members, it is a tough job to reuse business logic than components or objects. One of the solutions to have emerged to take care of this problem is reusable services. To make a reusable service successful, we should identify the services that are difficult to develop and need expensive development and knowledgeable resources.

Krueger (1992) proposed a framework to evaluate software reusability with the following four aspects (see also Zhu, 2005):

- *Abstraction*: A reusable component should have a specific level of abstraction that helps software developers avoid sifting for details.
- *Selection*: A reusable component should be easy to locate, compare and select.
- *Specialization*: A group of reusable components should be generalized and the generalized form should be easy to be specialized to an application.
- *Integration*: There must be an integration framework to help reusable components be installed and integrated into a newly developed application.

The business logic can be published in the form of a service in such a way that it can be discovered, matched, composed, executed, verified and monitored in real time and at runtime. This brings about a new paradigm of computing called *Software as a Service*.

Software as a Service

Software as a Service (SaaS) has become increasingly relevant to both enterprise and SMB customers and has the potential to impact the entire IT industry. Although the market size for SaaS was relatively small in 2005, to the tune of approximately US$ 6 billion, as per a McKinsey analysis report, it is expected to grow by more than 20 per cent annually. There are many factors that are driving this paradigm shift.

Some of the primary factors of choosing SaaS are listed below.

Need for frequent updates

Due to the constant changes in the business model, in the business environment and in the technology front, it has become necessary to provide frequent updates of the applications that have been supplied to customers. This is a painful process and involves much effort and higher cost. Sometimes, the cost is transferred to the customer in the form of maintenance fees. In SaaS, as there is one application installed globally and no locally installed software, companies avoid the headache of software upgrades or aging technology. In case software updates are required, they are sent automatically to clients, providing seamless, hassle-free updates. By centrally upgrading one version of the application, every user can access the same latest capability.

Provide secured environment

With a large number of Internet crimes occurring, it is very important to provide a secure environment to users for usage of any application. Many companies do not have the technical skill set or operational capabilities to monitor and respond to security threats to their networks. IT companies that build and provide SaaS applications have first-hand knowledge in tackling security issues, have the necessary hardware and software components and understand the importance of maintaining appropriate control over access to client data. While architecting SaaS applications, security is always considered to be the primary requirement as the applications are developed to be used over the Internet. Security measures added into an application as an add-on service is never as good as security built into the architecture from the beginning. SaaS providers actually offer better data security and better application reliability than in-house operations, because the data centres are up-to-date with security management and have built-in redundancy.

Drive for reduced total cost of operation

The implementation of a typical system includes data consolidation, security, a tailored interface and complex reporting needs. With a SaaS application, the system along with the data are stored, backed up and secured by the vendor. So there is a possibility to reduce the IT personnel costs significantly if not eliminating the entire IT team. Additionally, as the application is being maintained centrally by a team of experienced professionals, the training and operational costs are reduced drastically. SaaS models improve efficiency by enabling vendors to interactively collaborate with clients during implementation. Incorporating client feedback and providing immediate results is far easier with a SaaS application than with other platforms.

The primary advantages of the SaaS model is that the focus of the IT engineers will be shifted more towards the domain than the software engineering nuances. The companies can focus on the business domain, business models, cost model and the return on their investment. They can have a very thin IT department and can depend on SaaS providers for their IT needs. They can select the applications they need from the SaaS applications available in the market and use them as required. The best example is salesforce.com. A number of companies worldwide are using this service for their sales management. It provides all the support required by senior management for analysing their sales data, trend analysis and keeping track of their sales target.

SaaS is also viewed as 'do it yourself development'. This implies that software professionals will have to become more of domain experts (on a lighter vein, considering that other professionals are keen to become software engineers and software engineers are going to become experts in other domains).

One of the major challenges of using the SaaS model is the integration of third-party applications. A reasonable amount of an organization's IT budget is spent on interfaces and integration. Whenever a system is modified due to change in requirements or environment, the interfaces need to be rebuilt. It is a complex task when there are a large number of interfaces in a system. Addressing integration concerns has been particularly a key concern to enterprises and they are shifting to the more reliable integrated option 'service-oriented architecture'. Thus, larger companies still look at SaaS as a quick way to solve a business problem and not as an application to replace something they already have in their portfolio.

SERVICE-ORIENTED ARCHITECTURE

The term *service-oriented architecture* (SOA) was coined in a research paper by Gartner analyst Yefim V. Natis (1996) as follows:

SOA is a software architecture that starts with an interface definition and builds the entire application topology as a topology of interfaces, interface implementations and interface calls.

He also mentioned that SOA is better named as *interface-oriented architecture*.

Much before SOA became popular, software professionals were using distributed object technologies such as DCOM and CORBA to create a set of design principles, guidelines and best practices. These distributed objects, developed in the form of components, used to be referred as *services*. It was like using object-oriented concepts in client–server architectures.

SOA has many definitions, which are widely different and a bit confusing to the software architecture fraternity. The World Wide Web consortium represents it as follows (Booth *et al.*, 2004):

- An SOA is a form of distributed systems architecture, where the provider and the requestor of the service are interacted through message exchange. The implementation of the process including its internal structures and processes are abstracted and kept hidden. The service is usually described in terms of those information that are exposed through the service definition. The services are designed in such a way that they provide only few functions and can be accessed through a platform-neutral standardized format (such as XML string).
- A service is an abstract resource that represents a capability of performing tasks that form a coherent functionality from the point of view of provider entities and requester entities. To be used, a service must be realized by a concrete provider agent.

The definition mentioned by W3C is very generic in nature. Another definition, which is much specific and inspired by Sayed Hashimi (2003), states that a service is an exposed piece of functionality that is dynamically located and invoked. A service is self-contained and maintains its own states and the interface contract to the service is platform independent.

SOA supports three primary functions to provide services to consumers:

- Create an application as a service, describe the interfaces and publish the service.
- Discover a service already published.
- Consume the service using the interfaces.

SOA is an *architectural style* that emphasizes implementation of components as modular *services* that can be discovered and used by clients. Services are defined by the messages through which they can be communicated and used. Services are published in the service registry with the details of their capabilities, interfaces, supporting protocols and policies. They do not publish the implementation details such as programming language and the platform on which it is hosted as they are of no concern to clients.

Though a service is usually used individually and supports a particular functionality, it may be integrated with other services to provide higher-level services. This promotes reuse of services and allows the architect to use them in an orchestrated manner.

As per Adrian Trenaman (2005), SOA is an *architectural style* for client–server applications. Distinguishing features of SOA include the following:

- An SOA consists of clients and servers (services) connected by a communication sub-system known as a service 'bus'.
- Services are defined using formal interfaces (contracts) that hide implementation details from the client.
- The bus supports both point-to-point and messaging styles of communication and supports enterprise qualities of service such as security and transactions.
- Services are often reused by a number of applications.

In an SOA, services exhibit the following properties:

- Its interface is defined using a formal contract (e.g., WSDL or IDL). The service implementation details are unknown to the client; the only connection (or *coupling*) between the client and server is the contract.
- The service is self-contained and can be run with or without other services present.
- Services may be used by other services.
- The interface is stored in a well-known repository that prospective users can browse.

In SOA, the contracts between the service providers and consumers are formal. The implementation of the services are abstracted and isolated from the consumer.

SOA is essentially a collection of self-contained, pluggable, loosely coupled services that have well-defined interfaces and functionalities with little side effect. A service is thus a function that is self-contained and immune to the context or state of other services. These services can communicate with each other either by explicit messages that are descriptive and not instructive or by using master services coordinating or aggregating activities, for example, in a workflow.

The consumer–provider role is abstract and the precise relationship relies on the context of the specific problem. SOA achieves loose coupling among interacting software agents by employing two architectural constraints: (a) a small set of simple and ubiquitous interfaces to all participating software agents and (b) the interfaces being universally available for all providers and consumers.

Challenges in Implementing SOA

The primary challenge of considering SOA is to get a collective agreement from the stakeholders to use SOA as the enterprise framework for delivering services. The success of implementing SOA is based on the availability of services that are useful to the users of the services. It is, thus, very important for organizations to adopt service orientation as the key principle for their IT solutions.

Another challenge of SOA implementation is integration of services available in different parts of the organization. All the services should be placed in a centralized environment so that one service can access the others easily. As the services can be modified or new services can be added at any point in time, it is necessary that the services be discovered dynamically. The services should be able to expose their definitions and allow easy access.

SOA enables an integrated environment for all the services being offered by the organization. It is, thus, important to have an efficient governance structure for the orchestrations of the services. Governance is required not only for the management of the services but also for the management of the resources, such as business processes, business intelligence, hardware and shared software components, used to provide the services.

While defining the services, it is necessary to decompose the business processes to identify common processes that can be reused by different services. Reuse of services or business processes help the organization to achieve higher return on investment. It would be a challenging task while implementing SOA to identify the business processes that are used by many services. These services can be designed as elementary services and implemented as shared services.

It is also important to change the mindset of the people involved in implementing SOA in any organization. To make the SOA implementation a success, it is important to motivate all the stakeholders with information on the usefulness of SOA and create a roadmap with proper change management plan.

Web Services

Most people are familiar with accessing the Internet through a Web browser. When a user requests a Web page, the request is handled by a remote Web server, which returns the information in *hypertext markup language* (HTML)—a form that allows the browser to present it using a selection of fonts, colours and pictures, all factors that make it more useful and appealing.

Web services are distributed software components that provide information to *applications* rather than to humans through an application-oriented interface. The information is structured using *extensible markup language* (XML) so that it can be parsed and processed easily rather than being formatted for display. In a Web-based retail operation, for example, Web services that may be running on widely separated servers might provide account management, inventory control, shopping cart and credit card authorization services, all of which may be invoked multiple times in the course of a single purchase.

As Web services are based on a service strategy that uses common communication protocol to facilitate interaction between the service provider and the client of the service, they do not depend on any specific platform and can support a distributed heterogeneous environment.

SOA and Web Services

We initially described SOA without mentioning Web services, and vice versa. This is because they are orthogonal: service orientation is an architectural style, while Web services are an implementation technology. The two can be used together, and they frequently are, but they are not mutually dependent.

For example, although it is widely considered to be a distributed-computing solution, SOA can be applied to advantage in a single system, where services might be individual processes with well-defined interfaces that communicate using local channels, or in a self-contained cluster, where they might communicate across a high-speed interconnect.

Similarly, while Web services are well suited as the basis for a service-oriented environment, there is nothing in their definition that *requires* them to embody the SOA principles. While statelessness is often held up as a key characteristic of Web services, there is no technical reason that they should be stateless—statelessness would be a design choice of the developer, which may be dictated by the architectural style of the environment in which the service is intended to participate.

Grid Computing

Grid computing is a form of distributed computing in which the use of disparate resources such as compute nodes, storage, applications and data often spread across different physical locations and administrative domains and is optimized through virtualization and collective management.

Grids are often classified as either *compute* grids, which emphasize the shared use of computational resources, or *data* grids, which support federation, integration and mining of data resources. These distinctions mostly dictate the type of hardware infrastructure needed to support the grid; for example, nodes on a data grid may need to provide high-bandwidth network access, while a grid whose primary use is for long-running parallel applications is more in need of high-performance computational nodes. Other classifications, such as *hallway*, *campus*, *enterprise* and *global grid*, indicate the scale or other properties of the grid. In practice, grids tend to display characteristics of various functional classifications and their scale tends to increase over time, so the distinctions are often blurred.

Although grid computing was conceived in research organizations to support computer-intensive scientific applications and to share large research databases, enterprises of all types and sizes are beginning to recognize the technology as a foundation for flexible management and use of their internal IT resources, enabling them to better meet business objectives such as minimized cost and increased agility and collaboration.

Grid computing is thus the foundation for collaborative high-performance computing and data sharing for the adaptive enterprise and for the vision of utility computing, wherein computing resources will become pay-per-use commodities. While the precise definition is debated, common characteristics of grids are that they tend to be large and widely distributed, require decentralized management, comprise numerous heterogeneous resources and have a transient user population. Hence, grids exemplify the need for a highly scalable, reliable, platform-independent architecture that supports secure operation and standardized interfaces for common functions.

Service-Oriented Grids

Grids are facilitated by the use of grid middleware: software components and protocols that provide the required controlled access to resources. To date, grids have been built using, for the most part, either ad hoc public components or proprietary technologies. While various public and commercial solutions have been successful in their niche areas, each has its strengths and limitations, and they have limited potential as the basis for future-generation grids, which will need to be highly scalable and interoperable to meet the needs of global enterprises.

In recent years, however, it has become clear that there is considerable overlap between the goals of grid computing and the benefits of an SOA based on Web services; the rapid advances in Web services technology and standards have thus provided an evolutionary path from the 'stovepipe' architecture of current grids to the standardized, service-oriented, enterprise-class grid of the future.

The user submits a job request to the job-submit service, which locates the grid's scheduling service and passes on the request. The scheduler locates and contacts the service that represents the requested application, and requests details of the resources that are required for the job. It then queries the registry to find all suitable resources and contacts them individually via their associated services to determine availability. If sufficient resources are available, the scheduler selects the best available set and passes their details to the application service with a request to begin execution; otherwise, it queues the job and runs it when the required resources become available. Once the job has been completed, the application service reports the result to the scheduler, which notifies the job-submit service. The job-submit service, in turn, notifies the user.

Note that this example is simplified for clarity: a real enterprise-class grid would involve much more sophisticated operation than is shown here, with the net result being a high degree of automation and optimization in the use of resources across the grid's domain.

Enterprise Service Bus

The clients and servers in an SOA communicate via a common communication framework called a 'service bus'. The term *service bus* is derived from *computer bus* where the computer components such as CPU, RAM and I/O are connected through the wire mess. Similarly, in SOA, each component provides an interface to the bus, which allows it to communicate with a large number of different kinds of components. One component places a message on the bus and another component receives it and serves that request.

The term *service bus* is used metaphorically to describe a sub-system that connects clients and servers using the same underlying protocols and technologies. The service bus is a single communication sub-system used by all the clients and services in an SOA (Figure 7.1).

A service bus usually includes system services that provide clients and servers with commonly required functionality:

- Service naming and lookup—for finding services at runtime
- Service registry—for storing service contracts

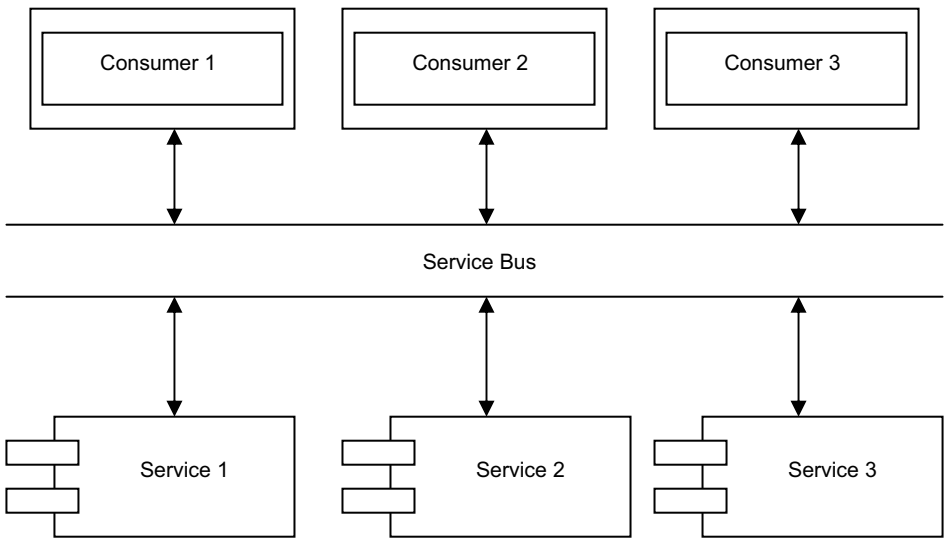


Figure 7.1 The service bus in SOA

- Security
- Transactions
- Service management
- Messaging

Typically, the services are implemented using the same communication infrastructure that clients and *normal* servers use.

The term *enterprise service bus* (ESB) first emerged around 2003 and has become synonymous with SOA. It has been hijacked by marketers and vendors and liberally redefined to suit their product sets. The 'enterprise' in ESB stresses that the product has features such as security and transactions and is suitable for use in a large-scale enterprise. ESBs provide support for contract-based services using either direct (synchronous point-to-point) or indirect (asynchronous messaging) communication paradigms.

Though technologies such as CORBA and DCOM have provided service-bus functionality for some time now, the industry has settled on the following rule-of-thumb: 'if it doesn't use XML, then it's not an ESB'. To qualify as an ESB, a product *must* use the following Web services technologies:

- XSD and WSDL for contract definition
- SOAP and XML for protocol payload
- UDDI for service registry

As Adrian Trenaman (2005) quips, 'Perhaps ESB is a misnomer; it should have been called XML Services Bus or Web Services Bus'.

An ESB typically provides the following functionalities:

- Data modelling with XML schema
- Interface modelling with WSDL

- Client and server development tools (code generation from WSDL, libraries, IDE support, etc.)
- Synchronous point-to-point communication using SOAP/HTTP
- Asynchronous messaging using SOAP as a payload over a messaging protocol that supports persistence of messages
- Message payload transformation using XSLT

The ESB provides core infrastructure and tools for SOA. Many ESB components such as Registry and JMS are pluggable. Some of the open source ESB choices available include the following:

- *Celti*x: Hosted on ObjectWeb, supported commercially by IONA Technologies
- *OpenES*B: Hosted on java.net
- *Mul*e: Hosted on codehaus.org
- *ServiceMi*x: Hosted by LogicBlaze

Each of these ESBs typically provides a default JMS messaging provider. The registry system for WSDL contracts is defined by the UDDI standard. Some of the open source implementation of registry system are the following:

- *jUDDI*: Hosted by Apache (http://ws.apache.org/juddi)
- *Ruddi*: Hosted by InspireIT (www.inspireit.biz)
- *Nsure UDDI server*: Hosted by Novell (http://developer.novell.com/uddi)

ESB uses WSDL for interface definition, which provides a clean separation of interface versus location (host name, port, etc.) Interfaces are defined using WSDL data types using XSD. ESB *typically* uses SOAP as a messaging payload. SOAP 'wraps' XML messages in an XML wrapper. Messages are transmitted as plain text.
ESB can support the following messaging styles:

- One way (using SOAP/HTTP or SOAP/JMS)
- Request response (using SOAP/HTTP or SOAP/JMS)
- Document oriented (using SOAP/HTTP or SOAP/JMS)
- Publish subscribe (using SOAP/JMS)

Thus, we observe that ESB offers great advantages in terms of technology adoption and acceptance of the underlying XML core.

Business Trends

A Gartner Research publication (Schulte *et al.*, 2005) mentions that the introduction of Web services and the resulting popularization of SOA caused a major upheaval in this market. Prior to 2002, integration backbones were largely proprietary, and many application developers were unknowledgeable about SOA and event-driven design. From 2002 to 2004, a new type of middleware, ESBs, came into the market to exploit Web services standards and the growing interest in SOA. Simultaneously, the original integration suite suppliers enhanced their products to adopt Web services standards, business process management and other features. The platform middleware suppliers also broadened their offerings, adding support for Web services, portals, message-oriented middleware, integration and other features to their application servers to create a new product category, the APS. In 2005, all three types of products addressed customer needs for the integration backbone, but their relative strengths vary because of their diverse backgrounds and company strategies.

Another Gartner report (Barnes *et al.*, 2005) mentions that among global organizations across all geographies, business trends are driving an expanded focus beyond the stability and reliability of operations and processes. It has been observed that organizations are increasingly focusing on agility and flexibility of their operation while simultaneously improving overall process visibility and consistency. SOA is being eyed as one of the tools to provide the desired flexibility as a group of services; once designed and implemented, SOA can be integrated and used to create new business processes. So the emphasis of organizations shifts towards process definition, visibility and control.

The increased focus on SOA will create a series of changes in the current operating model of many IT organizations that are not organized to support this new model. The impact will be especially acute across the following areas of IT governance (Gaugham, 2006):

- *Investment planning*: SOA's compartmentalized development model will force a change in how we build the business case for new and existing investments, how we prioritize them, and how we measure benefits since most current portfolio management processes are designed to support long-term projects and will not support the higher volumes of investment candidates.
- *Software development*: SOA-based development is more like an assembly line, where the developers will be working on discrete services and assembling them with existing internal services or even third-party services. Testing models will also need to change to support distributed composite applications that may span multiple companies' service repositories.
- *Change management*: SOA will create an environment in which change is more dynamic since you can update one part of the composite application without breaking the whole.
- *Maintenance and operations*: A composite application based on SOA will be distributed across the organizations and even outside the firewall with little or no control on the performance characteristics of the service. So it will change the way we manage the services and new methods based on service line architectures (SLAs) to be innovated to achieve the goal.
- *Security*: SOA adoption will call for a new model for application security that will be required to take care of security aspects of services offered as well as the integrity requirements of composite applications.

To take care of the above-mentioned impacts organizations should evaluate the maturity of important processes that span across organizations, identify the best practices and improve the effectiveness of core processes. They should also clearly define a strong enterprise architecture to ensure that they can maintain the agility and flexibility of SOA while adapting to meet changing business requirements.

Though the technology required for SOA is very important, the cultural, procedural and organizational changes that are required to be implemented effectively will ultimately determine the success of SOA in most organizations. Another important factor to be considered is identifying the business scenarios in which SOA can be beneficial.

Though SOA is fundamentally about business process agility, it can also address some other business issues such as inability to respond to business demands for new application functionality, processes or information access, inconsistent or inadequate information and data quality and inefficient IT resource utilization (Barnes et al., 2005). It also enables organizations to reuse established applications cost effectively. Organizations get greater capability to modify, extend or improve individual services independently in cases where there is no change to the underlying contract. SOA helps business owners to take a more direct, active role in the design of IT systems. This enables IT systems to be more accurate in reflecting business processes and improves IT systems' capability to adapt as processes change.

As SOA maximizes the reuse of applications, it helps in saving costs in developing new components and maintenance of developed components. As the consumers of the services are more concerned about

the service contract and the service delivery and not about the service implementation, it is possible to test the service implementation independently and consistently, thus reducing the cost and time required for software testing.

Though SOA has a number of benefits to the business community, there are a number of business challenges that should be considered before adopting SOA. The most important challenge is non-availability of mature resources with SOA experience and expertise. In most of the cases, organizations need to spend a lot of time and cost to build a team with SOA expertise and develop the guidelines and process for implementing SOA. Organizations also need to invest more on network infrastructure as the requirements for an SOA environment are more complex.

As per a recent Gartner report (Schulte and Abrams, 2006), SOA will be used, in some part, in more than 50 per cent of new mission-critical applications and business processes designed in 2007, and in more than 80 per cent by 2010 (with a probability of .7).

DIMENSIONS OF FUTURE SOFTWARE ARCHITECTURE

The software architecture field is at present at a crossroad, where it is important to redefine the dimensions that we should consider to take it to a new peak. In the working session of the IEEE Conference on Software Architecture in January 2007 (WICSA, 2007), the topic was discussed and the dimensions discussed below were identified.

Codification and Socialization

These are the processes by which an architect communicates architectural ideas to the stakeholder community. Codification refers to the strategy based on explicit reuse of architectural knowledge and solutions, while socialization refers to the strategy based on knowledge creation, sharing and transfer. Codification and socialization are complimentary in nature and they should be used in such a way that they reinforce each other.

For example, let us consider the 'elevator pitch' scenario: what if, by chance, you wind up in an elevator with a key person, such as the CIO, and you have 20 seconds to convince him or her that your ideas are worth exploring further? If you have a strong idea that has been proven and used, use the codification strategy and explain clearly how your idea can make a change and add value to the business. If you want to try a new idea, then use socialization strategy and ask feedback from the listener. Based on the feedback you have received from an earlier listener, make the ideas more consistent, more innovative and simpler to understand.

Though the choice of using codification or socialization strategy depends on the maturity of the solution and the business unit, it is observed that a combination of both has been successfully used in many industries among their business units.

Handling Quality Attributes

It is very important to identify the quality attributes and handle them in designing the architecture of the system. Though quality attributes linked to business and engineering needs are specified, all quality attributes are not yet quantitatively available. Usually, the quality requirements are plugged-in later during the coding phase. It is important to consider them during architectural design. Researchers are working on understanding domain quality attributes and ways to achieve them. Generation of quality standards and certification process is also on the cards. Another important factor is to use traceability between the quality attributes and architectural decision points and provide analysis of the architectural quality needs.

Architectural Automation

There are around 50 different architecture specification languages available, but none of them allows the automatic conversion to the building blocks. UML tried to achieve a part of it providing diagrams and tools, but it is not sufficient for architects or designers. Building blocks with defined quality attributes for specific domains should be created and architectural language to be used for the automation.

Architectural Responsibility

It is important to define the role of an architect and specify the degree to which an architect is responsible. In most organizations, the role of architects is consulting. They give their ideas and suggest some styles and patterns. They are usually not held responsible if the system faces any problem in the future. If the system development succeeds, the architect is not given the recognition. So, a clear definition of an architect's role and authority has to be established and adhered to.

Accidental versus Intentional Architects

It is observed that in most cases architects are selected on the basis of their development/programming experience. They usually do not have any formal training on architecting systems. They also do not have a clear career path, so they want to learn more on project management than on architecting systems. It is, thus, important to create a career path for architects so that they intentionally chose this career path and excel in their work rather than being accidentally put in this role.

Domain Versatility and Adaptability

Often, the architectural design depends on the technology and platform to be used for the development of the system. The domain is not considered at all. The architect also does not have much domain knowledge. It is, thus, important for the architect to have domain knowledge and the maturity to link the domain information with architectural goals and techniques.

Technology Dependency

Technology is a tool to help the architect to design the system efficiently. Architecture should not be constrained by technology. Architects should specify the abstract solution without necessarily having any bindings to existing technology. Technology should not restrict the architect while designing the system. Actually, the architect should be fearless enough to architect the system such that it will influence future technology.

Inter-disciplinary Architecture

At present, an architecture is designed based on a single discipline, which is fully understood by the architect who is comfortable with the efficiency of the design. It is important to architect the system considering multiple disciplines fully integrated into a single system. The architecture should meet the perspective of all the stakeholders.

CRITICAL SOFTWARE ARCHITECTURE ELEMENTS

In any discipline, fashions, terminology, platforms, methods and frameworks, which are the means to achieve the goal of building a complex software system that is usable, faster, better and cheaper, might

change but there are always some critical elements that remain important. Service orientation is based on some such critical elements. They include serviceability, adaptability and productivity. For the software architecture field, we believe that these critical elements based on service orientation are very important along with performance, reliability and availability, but they are not as well understood.

In this article, we would like to discuss serviceability, adaptability and productivity from the SOA point of view (that is, how software architecture can be serviced, adapted and efficiently produced). Our prediction is that in the future these elements will become more important as the complexity of the software we are developing increases and the environment in which we operate changes.

Criticality of Serviceability

The importance of serviceability is growing as the systems being developed are transitioning from being backroom operations to becoming business critical. Here, serviceability means all the users of a given system are appropriately assisted by the system itself in achieving their goals and the functionality expected by the system is provided. These users include not just the so-called end customers but also the other users, including those who provide services to their customers using this system and the users responsible for maintaining the system by making sure the system runs appropriately. System administrators help maintain users, data and processing of the system. This list includes those who take backups, those who add/delete users of the system and those who watch the system during peak usage times. Programmers help enhance the system as the environment in which the system operates gets changed.

All of these users service the system so that it continues to serve end customers. In other words, the functional requirements of these users are different from functional requirements of end users. For example, a 'programmer user' whose responsibility is to enhance the system has a requirement that the system should support easy debugging and its architecture should allow adding new features with ease. Similarly, a person who maintains user data should be able to easily add, revoke and view the privileges of all the users. Most often, what are normally perceived as non-functional requirements, such as maintainability, reliability and availability, will become functional requirements of these special users.

In this context, it is very important to define the term *service* properly. At the minimum level, each service should have the following:

- A clearly defined set of consumers or the customers or the users
- A clear expectation of what functionality the service provides
- A concrete understanding of the quality of service that is guaranteed

The guaranteed quality of service needs some attention here. Every service provided by an IT system and the IT system has a whole should clearly specify what the customer should expect. It should not hide this aspect. This is an explicit commitment a service will make to those who use this service. Each service should be built in such a way that it honours its commitment, and in case it cannot honour it commitment, it should know how to manage that situation. IT systems should be engineering in such a way that this serviceability aspect is built into them. It implies that the system's architecture has to support easy serviceability and that the quality of service has to be engineered into the system. Performance engineering for software development (Smith, 1990) shows how performance (performance-oriented architecture) challenges are identified and addressed. Similarly, if other architectural drivers such as security (architecture for security) are identified, the challenges related to security needs to be identified and addressed.

Enterprises of today are increasingly service centric in their dealings with customers as well as with their suppliers and other partners. As an example, a banking customer today more or less acts as his or her own teller in the course of an online or ATM-based transaction. An unwieldy application might be enough with a trained employee, but not when it is directly accessed by a customer. With the customer being a

direct consumer of IT services exposed by an enterprise, it is important to ensure that the service quality as promised to the customer is maintained.

It is very important to understand that an enterprise that is service centric is service centric all the way down.

A business service consumed by the customers of the enterprise is realized by a combination of systems and processes within the enterprise. For the service-level commitments of this service to be met in an efficient fashion, these commitments must translate into a set of appropriate and aligned commitments to any part of the enterprise involved in its delivery.

As an example, consider the service commitments one would expect from a hospital: patient records available all the time and at any time, critical systems available all the time, the formalities of completing a patient admission must be over within a small period of time, and so on. This is the quality-level demanded from the hospitals business systems.

These business functions are realized by a set of processes that include workflows with human, non-IT and IT system participation. These systems are further realized as software systems that finally run on a set of infrastructure. For the business commitment to be maintained, the entire ecosystem that supports the business function must be aware of and must be designed to support that commitment. Hence, if patient records need to be available all the time, the storage system inherits an SLA that demands that it is available all the time and that information inside it cannot be corrupted and that it is truly persistent. It also means that the storage system must be backed up, but the backup cannot take the storage system, as seen by the rest of the environment, offline.

Thus, the design of each component of the enterprise in the ecosystem that supports this business function needs to be done so that the delivery of the business service commitment is achieved in the most efficient fashion. The only way this can be done is if the entire ecosystem is modelled as a set of services. The business service is composed of a set of other services. Each service has stringent service-level commitments that it must honour to its consumers.

Criticality of Adaptability

Whatever the reason, be it changes in the requirements from the customer, changes in the environment in which the system operates or the need to create newer applications and modify existing applications, the need to build adaptable systems cannot be overemphasized. Adaptability aids in software being evolved over a period of time in a natural manner.

There are various models to tackle the adaptability aspect of the software systems, especially from the software architecture point of view. Unless the software architecture itself is adaptable the final software system cannot be made adaptable. Evaluation at the software architecture level is the highest level of abstraction one can aim to achieve. If a software system is designed in the most adaptable manner, it should be a self-managing system. In other words, adaptable programs change their behaviour automatically with the changes in the environment they operate.

However, adaptability is a tough problem. There is a classic debate between being generic and being specific. If the system is built to be very generic, it suffers from software bloating. In other words, there will be huge amount of code that will perhaps never be used because most of it is written to make the system more extendable or adaptable in case there are future requirements. This phenomenon not only makes the software inefficient in terms of its speed and other performance aspects but also it makes maintaining the software very hard, which actually contradicts the very goal of making it more evolvable. On the other hand, if you make the software very specific, though it enjoys advantages of high performance and low or no-fat code, it suffers from being non-adaptable to new environments. It is also important to come up with adaptability metrics so that adaptability requirements can be verified at a later and appropriate point in time.

There are various categories of adaptability solutions: architecture-based solutions, component-based solutions, code-based solutions, generic algorithmic techniques, dynamic adaptation techniques and adaptation methodologies (Subramanian, 2003). Without going into details of each of these categories, we would like to state that among these architecture-based techniques offer the highest level of abstraction. We will focus on the architecture-based approach to adaptability. In order to make high-level abstraction of the architecture possible, it is important to treat adaptability as one of the important non-functional requirements from the beginning.

An adaptable system should be capable of

- Detecting any changes in the environment (environment change recognition)
- Determining the changes to be made in the original system based on the changes detected in the environment (system change recognition)
- Effect these changes in the original system to create a new system (system change)

In other words, the requirement of adaptability is decomposed into several sub-goals. This approach is influenced by the non-functional requirement approach to the adaptability problem as proposed in Subramanian (2003). In this approach, decomposition is based on knowledge available (from a knowledge base) about a class of systems. Based on this knowledge about sub-goals, methods for refining these sub-goals can be developed and put back in the knowledge base. This is essentially a knowledge-based approach very similar in its spirit to QFD or the house of quality model proposed by Hauser and Clausing (1988).

Criticality of Productivity

The architectural productivity is concerned about providing tools, methods and frameworks for an architect so that he or she can conceive systems that can be built quickly without compromising on the quality of the solution.

We can see better tools in the marketplace from vendors such as IBM Rational, Borland and Telelogic. These tools are much more helpful and useful for architects than what was available in the past. Of course, there is still a lot of scope to make them assist architects in more appropriate ways. As methods and other infrastructure improve, these tools automatically get enriched.

More than the tools, development platforms such as J2EE, .Net and Symbian/Series 60 offer more assistance to the architecture community and that too at a more rapid pace. These are, in a sense, platforms with built-in architectures. An architect needs to extend them for a given application. Of course, the disadvantage is that the architect needs to move very close to platform-dependent skills and knowledge. The architect needs to have complete insight about the capabilities provided by these platforms in order to build applications on top of them. Most of these platforms have XML-based or SOAP-based communication between various layers of the architecture.

From 1994 to 1998, there was much emphasis on architecture description languages (ADLs) among the research community in software architecture. We have seen several ADLs including Wright, ACME, Darwin, Rapide, C2, Koala and UML. Of these, only a few survived and proved to be useful. Koala and UML (note that many ADL purists do not agree that UML is an ADL) are perhaps more successful in terms of being useful in practice. ACME found more visibility among the research community. After the late 1990s, there was very little emphasis on ADLs except ADML in 2000 and UML 2.0 more recently. However, the body of knowledge created because of this activity was very useful in creating various methods and platforms within the software architecture field and indirectly aiding productivity goals.

Coming to methods in software architecture, several of them became very popular including SAAM, which was developed in 1995, RUP (1996), ATAM (1998), ATAM 2 (2003) and ADD (2003–2004). We have discussed some of them in detail in previous chapters. These methods have changed the way architects

approach a problem, making it more systematic and predictable, thus having a tremendous impact on the productivity goal of the software architecture.

However, the best bet to increase productivity is the systematic approach to reuse. The productivity increases by increasing the reuse in software development. Whether it is component-based development or service orientation, reuse in its various manifestations will enhance the productivity in a very certain way if our approach is systematic.

Software product lines or product line architectures achieve this goal in a very systematic way. Clements and Northop (2001) defined SPL as follows:

> A software product line is a set of software intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.

Essentially, SPL is a strategic reuse approach. Starting from requirements analysis till building individual components, product lines amortize the investment into development of *core assets* for a given product line. The core assets are built for various phases of software development in the following order:

- Requirements and requirements analysis
- Domain model
- Software architecture and design
- Performance engineering
- Documentation
- Test plans, test cases and data
- People: their knowledge and skills
- Processes, methods and tools
- Budgets, schedules and work plans
- Components

If reuse is brought in earlier phases it will be more beneficial. There are three key concepts in software product lines:

- *Core assets*: Product lines emphasize on making use of a common asset base in the architecture phase. This is very similar to what we discussed in the component-based development life cycle. There is an emphasis on developing the core assets. Typically, there are attached processes for this phase, such as maintaining the core asset repository and adding and retiring assets over a period of time.
- *Production*: The production or actual implementation part of the software is guided by the production plan that aids developers in gluing together all the components put together by the architecture. The production plan considers what the existing assets are and what the scope of the product line is.
- *Management*: To have a successful product line, managing various resources is essential. Management here includes achieving the right organizational structure, allocating resources, coordinating and supervising, providing training, rewarding employees appropriately, developing and communicating an acquisition strategy, managing external interfaces and creating and implementing a product line adoption plan.

These three concepts or activities are interrelated and highly iterative. For a successful product line, each of these is essential, as is the blending of all three. Also, it is important to note that there is no 'first' activity. Each of them should start at the same time and independently. However, there is a strong feedback loop between the core assets and the products. Strong management at multiple levels is needed throughout.

There are three approaches to product lines:

- *Proactive approach*: In this approach, the core assets/components are developed first by investing in their development upfront. It might take longer for the first product to appear but later other products in the product line come to market quickly.
- *Reactive approach*: Here the core assets are developed from existing products. The earlier products perhaps were developed without the product line in common. However, by studying them, core assets that are common across all or most of these products can be pulled together. This activity has to be guided by the product line architecture. Using this approach, one can start a product line with a low cost compared with other approaches.
- *Incremental approach*: Using this approach involves the development of a part of the core asset base (architecture + a few components). Using this part, the initial few products are developed and released. These initial products help in extending the core asset base and this process continues in an iterative and incremental manner. This approach can be very beneficial if a good roadmap is created initially.

After we decide on the approach to take for the development of the SPL, we need to identify the practice areas. The book on software product lines by Clements and Northop (2001) identified 29 practice areas categorizing them into software engineering, technical management and organizational management. The software engineering practice areas include architecture development, architecture evaluation, COTS utilization, and so on. Technical management involves practice areas such as scoping, technical planning, tool support, configuration management, data collection, metrics and tracking. Business management includes business case development, customer interface management and market analysis.

For many organizations, focusing on 29 practice areas at the same time is not possible. There may be different practice areas that can be identified with respect to the context of an organization than those listed in the above source. Organizations need to figure out which practice areas are more important to initiate. One way to tackle this problem or resolve this conflict is to employ patterns as a way of expressing common context and problem-solution pairs.

For example, 'what to build' pattern or 'cold start' pattern gives us the contexts of a typical organization where there is a dilemma of how to initiate a product line process. This pattern also gives solutions to tackle various issues such as organizational planning, launching and institutionalizing, structuring the organization, funding, organizational risk management, customer interface management, developing an acquisition strategy, operations and training.

Software Service Lines or Service Line Architectures

We have discussed the major aspects of product lines to examine the possibility of applying them to the software service industry. We would like to call them as software service lines (SSLs) or service line architectures.

The emphasis here is to achieve reusability of business solutions through SSLs. This area is still at its infancy. Our purpose is to initiate the thought process in the area so that if a reader is interested he or she can take up the challenge of making inroads into this very important area.

Currently, many service companies are initiationg most projects from scratch. There is hardly any reusability across the groups. In a typical enterprise application development, business process modelling goes through several refinements and finally gets implemented using either tightly coupled components or loosely coupled services. Due to increasing demands, it is more viable to enhance reusable assets and maintain profitable margins. There should be a movement towards moving software service companies from labour-based model to asset-based model.

In one sense, software reuse in services setting might be like an oxymoron. The services industry typically gets paid by the man hour or man month. More time spent on the project results in higher payment. Some people think that if we are reducing the time on development by reusing core assets, it might act negatively on the bottom line. In addition there are also the following issues:

- How to protect core assets and prior IP when everything developed should be given to the customer?
- When to focus on developing core assets when everyone is busy working for various customers?
- How do they negotiate value for prior (or background) knowledge brought into the project?

While these are very important questions, there are certainly ways in which these can be addressed. A lot depends on the customer and the organizational policy. The important goal is to bring in more value to the customer. The service organizations also need to go up in the value chain by offering critical and important services in a cost-effective manner. The emphasis should be on increasing per employee productivity (and hence the revenue) than focusing on recruiting more and more people into the organization in order to grow.

In the services industry, software development differs from product development in various ways. For example the way one acquires requirements is different. The sources and methods of acquisition are different. The way one analyses requirements would also differ. The way a customer is involved in creating the end product also differs significantly, as in a product there is no single and specific customer for whom a product gets designed, whereas in services, every project typically begins and ends with a single customer. So, the intended end user of the output is different for both cases.

However, thinking from a different perspective, there is no thin line separating products and services. Often, this difference gets blurred. In one case, the product is the process and in the other case the process is the product. In services, the speed at which the system gets developed is very important for the customer and hence for the developer. In a competitive world, cycle time is one of the more critical performance measures. Customers always demand faster, better and cheaper solutions.

As an analogy, we can consider the automobile industry to illustrate the difference between product lines and service lines. The assembly line plants to manufacture different cars can be compared with product lines. Note that cars of different models but belonging to the same product line can be manufactured in a single plant. But these plants are fixed. For example, Toyota's Camry, Corolla and Avalon can come from the same plant as they belong to same product line.

Customizable assembly plants can be compared with service lines. Here, the plant configuration can be changed depending on the type of vehicle to manufacture. For example, the plant should be equipped to change its configuration if the product line changes. This means that the same plant can be used even if the product line changes but with a change in configuration. For example, imagine the same machinery producing Camry, Corolla, Land Cruiser and RAV4 in the same Toyota plant.

In this dynamic situation, it is very important to measure the quality of such service lines. Treat each of the reusable assets in the service line as an engineered object within an enterprise. Thus, their quality should be measured in the same way as the quality of other engineered objects is measured. That is, in terms of their quality attributes such as reliability, availability, performance and supportability.

Reliability is a measure of its completeness, correctness and consistency. Its performance is a measure of its responsiveness, throughput and efficiency. Its supportability is a measure of how the asset is designed to be supported, how it handles its exceptions and how the asset is maintained and upgraded with minimal impact to its users. As we discussed before, supportability also defines the reporting requirements on service-level accomplishment against commitments to the consumers of the service.

While discussing product lines we said that core assets are developed at various stages of the software life cycle. All of the core asset areas are also applicable for service lines. In addition to them, the following core asset areas are important to amortize the investments in service lines:

- Customer discovery
- Customer engagement
- Presales support
- Delivery excellence
- Vertical businesses (domain areas such as insurance and telecom)
- Horizontal businesses (platform areas such as SAP, J2EE, .Net)
- Support businesses (such as learning, enterprise systems)

Since product lines are relatively well studied and practiced, service lines can make use of the product lines knowledge body in defining scope, variation points and evaluation aspects. Scoping will determine what will be part of the core asset base and will not be. Variation points will tell us which locations within service lines, where different applications built using the same service line, are expected to vary. Evaluation of the service lines will be challenge as there are no formal models available yet. It will be interesting to see what kind of metrics and parameters will be evolving in order to evaluate the effectiveness of service lines.

In order to adapt a service line we need a visionary champion within the organization with sound knowledge of reuse models, service scoping and product lines. Over a period of time, we expect this to transform into a specialized discipline by itself. For each business unit within the service organization, either proactive or reactive or iterative approach can be used based on the appropriateness. Another requirement of any unit ready to adapt service lines is that it should be technology ready. It means readiness to embrace newer tools, methods and processes that aid the service line practice.

IBM has initiated and actively promoted a related area called service sciences, management and engineering (SSME). It is an interdisciplinary approach to design and implement service systems that are complex in nature and deals with people and technologies that provide value for others. SSME is a combination of science, management and engineering applied together. Various organizations such as Berkley, Arizona State University, CMU, MIT, Oxford, Tsing Hua, NCSU, Georgia Tech, San Jose State University and Stanford are already part of this initiative. This is essentially to promote research and practice in a service centric manner.

It will be interesting to see how this new area will shape itself in aiding the service industry.

Conclusions

Software architecture as a discipline has arrived. We have titles and roles in organizations with the keyword *architect*. We have more than 50 books written on software architecture and numerous articles. Many universities are teaching courses on software architecture. There are several conferences and workshops such as WICSA and SPLC that are regular events. Now no architect can start a design from scratch because there are several developmental platforms offering pre-created architectures such as J2EE, .Net and Symbian along with communication or data interchange

standards such as XML and SOAP. One needs to follow the rules already given by these platforms, which is making the life of an architect easier in one way and hard in other way—easy because there is so much reuse of already proven platform components of the architecture and hard because there is so much of platform dependency.

Gone are the days when we designed, implemented and delivered the system to the customer and had a sigh of relief. Customer shipment is only the beginning of the real game in terms of system development. Actual constraints, requirements and quality issues will surface only after this stage. Hence, the important factors that give strength to a system include the openness, serviceability and extensibility of the architecture. This is about moving beyond software architecture and building a solution that will adapt itself to the changing environment and continue to be useful.

We also see a future trend that can be called *lightweight software architecture*, similar to the idea of lightweight or agile programming practices. This comes from the similar context that gave birth to extreme programming, aspect-oriented programming and other agile methodologies. Since the market to be served is highly dynamic, and as the lessons from practice show that changes are normal, we expect that the stability of the solution is very hard. In this dynamic market with a changing solution space the architecture must be lightweight. However, there are lots of issues that need to be resolved. For example, the concept of *weight of architecture* has to be determined in a more formal and clear way. Currently, this term is being used in an informal way. The benefit of lightweight architectures and how to create lightweight architectures is not very clearly known yet.

In summary, we can expect tremendous growth in the field of software architecture. Service orientation is expected to take its true meaning and will be practiced at many levels in an organization. There is also a prediction that UML will continue to dominate the next decade of software architecture models and frameworks (Kruchten *et al*., 2006). It is also predicted that model-driven architectures will dominate software architecture research whereas the work on ADLs will reach a plateau. Material on software architecture including case studies and text books will be created, and more tools will be available to help architects in the design process. Whatever happens, it is an important and promising area in software engineering. Budding architects need to be alert and watch out for developments.

Further Reading

David Garlan (2000) wrote a very influential article 'Software Architecture: A Roadmap' (available in *The Future of Software Engineering*, Anthony Finkelstein (Ed.), ACM Press, 2000, ISBN 1-58113-253-0.) that discusses a few important research as well as practice trends of software architecture. Some of the speculations made in this paper have already come true. Balancing buy-versus-build options, network-centric computing and pervasive computing are speculated as major future challenges.

Philippe Kruchten, Henk Obbink and Judith Stafford were guest editors of the special issue (March/April 2006) of *IEEE Software Journal* on 'The Past, Present, and Future of Software Architecture' (Kruchten *et al*., 2006). This issue has a collection of very interesting articles. The editorial article lists some of the best text books, articles, conferences and events in the field of architecture.

Service-oriented architecture is now very popular, with several books and other literature being available. The following are our recommendations:
A series of three books by Thomas Erl from Prentice Hall is very popular:

- *SOA: Principles of Service Design*
- *Service-Oriented Architecture: Concepts, Technology, and Design*
- *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*
  Please check the Web site of this series at www.soabooks.com.
  We also recommend the following online resources to learn more about SOA:
- SOA Master Class: www.soamasterclass.com
- Thomas Erl's: www.soaprinciples.com

- 10 Things to Think About When Building the Perfect SOA: www.linthicumgroup.com/10_Things_to_Think_About_When_Building_the_Perfect_SOA.pdf
- 12 Steps towards SOA: www.linthicumgroup.com/12_Steps_towards_SOA.pdf
- Practitioners guide: http://dev2dev.bea.com/pub/a/2006/09/soa-practitioners-guide.html
- Wikipedia: http://en.wikipedia.org/wiki/Service-oriented_architecture
- IBM SOA: www-306.ibm.com/software/solutions/soa
- www.service-architecture.com

Nary Subramanian and his advisor Lawrence Chung wrote a couple of popular technical papers on software adaptability: 'Adaptable Software Architecture Generation Using the NFR Approach' (Subramanian, 2003) and 'Process-Oriented Metrics for Software Architecture Adaptability' (Subramanian and Chung, 2003). They give a new approach to increase the adaptability of software architecture.

For information on Service Sciences, Management and Engineering you can refer to the IBM Web site (www.research.ibm.com/ssme).

References

Abowd, G., Bass, L., Kazman, R., and Webb, M. (1994). SAAM: A method for analyzing the properties of software architectures (pp. 81–90). In *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, 16–21 May. Los Alamitos, CA: IEEE Computer Society Press.

Abowd, G., Bass, L., Clements, P., Northrop, L., and Zaremski, A. (1997). *Recommended Best Industrial Practice for Software Architecture Evaluation*. Pittsburgh, PA: Software Engineering Institute.

Barbacci, M. R., Klein, M. H., Longstaff, T. A., and Weinstock, C. B. (1995). *Quality Attributes*. Report CMU/SEI-95-TR-021. Pittsburgh, PA: Software Engineering Institute. www.sei.cmu.edu/publications/documents/95.reports/95.tr.021.html

Barbacci, M. R., Ellison, R., Lattanze, A. J., Stafford, J. A., Weinstock, C. B., and Wood, W. G. (2002). *Quality Attribute Workshops* (2nd ed.). Report CMU/SEI-2002-TR-019. Pittsburgh, PA: Software Engineering Institute. www.sei.cmu.edu/publications/documents/02.reports/02tr019.html.

Barnes, M., Sholler, D., and Malinverno, P. (2005). *Benefits and Challenges of SOA in Business Terms* (Gartner Research Publication No. G00130078). Stamford, CT: Gartner Inc.

Bass, L., Clements, P., and Kazman, R. (2003). *Software Architecture in Practice* (2nd ed.). Upper Saddle River, NJ: Pearson Education.

Booch, G. (2004). *Object Oriented Analysis and Design with Applications*. Upper Saddle River, NJ: Pearson Education.

Booch, G., Rumbaugh, J., and Jacobson, I. (1999). *The Unified Modeling Language User Guide*. Boston, MA: Addison-Wesley.

Booth, D., et al. (Eds.) (2004). *W3C Working Group Note 11: Web Services Architecture*. World Wide Web Consortium (W3C), February. www.w3.org/TR/ws-arch/#stakeholder.

Brandozzi, M., and Perry, D. E. (2003). *From Goal-Oriented Requirements to Architectural Prescriptions: The Preskriptor Process*. STRAW Workshop.

Brooks, F. (1975). *The Mythical Man-Month—Essays on Software Engineering*. Reading, MA: Addison-Wesley.

Brown, W. J., et al. (1998). *Anti Patterns—Refactoring Software, Architectures, and Projects in Crisis*. New York: John Wiley.

Brown, W. J., McCormick, H. W., and Thomas, S. W. (1999). *Anti-Patterns and Patterns in Software Configuration Management*. New York: John Wiley.

Brown, W. J., McCormick, H. W., and Thomas, S. W. (2000). *Anti-Patterns in Project Management*. New York: John Wiley.

Budgen, D. (2004). *Software Design*. Upper Saddle River, NJ: Pearson Education.

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

Checkland, P. (1999). *Systems Thinking, Systems Practice: Includes a 30-Year Retrospective*. New York: John Wiley.

Choi, H., and Yeom, K. (2002). An approach to software architecture evaluation with the 4+1 view model of architecture. In: *Proceedings of the Ninth Asia-Pacific Software Engineering Conference*, Korea.

Clements, P. (1995). Understanding architectural influences and decisions in large-system projects (Invited talk). In: *First International Workshop on Architectures for Software Systems*, Seattle, WA, April.

Clements, P. (1996). A survey of architectural description languages. In: *Proceedings of the Eighth International Workshop on Software Specification and Design*, Paderborn, Germany, March.

Clements, P., and Northrop, L. (1996). *Software Architecture: An Executive Overview*. Technical Report

CMU/SEI-96-TR-003 ESC-TR-96-003. Pittsburgh, PA: Software Engineering Institute.

Clements, P., and Northrop, L. (2001). *Software Product Lines*. Reading, MA: Addison-Wesley.

Clements, P., Bass, L., Kazman, F., and Abowd, G. (1995). Predicting software quality by architecture-level evaluation (pp. 485–498). In: *Proceedings of the Fifth International Conference on Software Quality*, Austin, TX, 23–26 October. Austin, TX: American Society for Quality Control, Software Division.

Clements, P., Kazman, R., and Klein, M. (2002). *Evaluating Software Architecture*. Reading, MA: Addison-Wesley.

Cohen, L. (1995). *Quality Function Deployment: How to Make QFD Work for You*. Boston, MA: Addison-Wesley.

Dudney, B., Asbury, S., Krozak, J., and Wittkopf, K. (2003). *J2EE Anti-Patterns*. New York: John Wiley.

Egyed, A., Grunbacher, P., and Medvidovic, N. (2001). *Refinement and Evolution Issues in Bridging Requirements and Architecture—The CBSP Approach*. STRAW Workshop.

Evans, G. (2004). Getting from use cases to code. *Rational Edge*. www-128.ibm.com/developerworks/rational/library/5383.html.

Garlan, D., and Shaw, M. (1993). *An Introduction to Software Architecture*. Advances in Software Engineering and Knowledge Engineering Series. Singapore: World Scientific.

Gaugham, D. (2006). *SOA Changes the Nature of Innovation for IT*. AMR Research Article, 9 March.

Gharajedaghi, J. (1999). *Systems Thinking: Managing Chaos and Complexity: A Platform for Designing Business Architecture*. Boston, MA: Butterworth-Heinemann.

Hashimi, S. (2003). *Service-Oriented Architecture Explained*. Cambridge, MA: O'Reilly ON Dotnet. com. www.ondotnet.com/pub/a/dotnet/2003/08/18/soa_explained.html.

Hauser, J. R., and Clausing, D. (1988). The house of quality. *Harvard Business Review*, May–June, 63–73.

Ionita, M. T., et al. (2002). *Scenario-Based Software Architecture Evaluation Methods: An Overview*. Eindhoven, the Netherlands: Department Software Architectures, Philips Research.

Jackson, M. A. (1995). *Software Requirements & Specifications*. New York: ACM Press.

Jackson, M. A. (2001). *Problem Frames: Analysing and Structuring Software Development Problems*. New York: ACM Press.

Jacobson, I., Booch, G., and Rumbaugh, J. (2002). The Unified Software Development Process. Upper Saddle River, NJ: Pearson Education.

Kärpijoki, V. (2001). *Anti-Patterns: A Seminar on Design Patterns*. Helsinki University of Technology, Helsinki, Finland.

Kazman, R., Bass, L., Abowd, G., and Webb, M. (1994). SAAM: A method for analyzing the properties software architectures (pp. 81–90). In: *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, May.

Kazman, F., Bass, l., Abowd, G., and Clements, P. (1995). An architectural analysis case study: Internet information systems (Invited talk). In: *First International Workshop on Architectures for Software Systems*, Seattle, WA, April.

Kruchten, P. (1995). Architectural blueprints—The "4+1" view model of software architecture. *IEEE Software*, 12(6), 42–50.

Kruchten, P. (2003). *Rational Unified Process: An Introduction* (3rd ed.). Boston, MA: Addison-Wesley.

Kruchten, P., Obbink, H., and Stafford, J. (2006). The past, present and future of software architecture. *IEEE Software*, 23(2), 22–30.

Krueger, C. W. (1992). Software reuse. *ACM Computing Survey*, 24(2), 131–183.

Liu, D., and Mei, H. (2003). *Mapping Requirements to Software Architecture by Feature-orientation*. STRAW workshop.

Liu, W.-Q., and Easterbrook, S. (2003). *Eliciting Architectural Decisions from Requirements Using a Rule-based Framework*. STRAW workshop.

Muller, G. (2003). *Architectural Reasoning: Balancing Genericity and Specificity*. Eindhoven, The Netherlands: Embedded Systems Institute.

Natis, Y. V. (1996). *Service Oriented Architectures, Part 1*. SSA Research Note SPA-401-068, 12 April.

Phadnis, S., and Bhalla, A. (2006). *I-TRIZ and the Digitization of Your Business*. A White Paper by QAI Innovation Practice, January.

Parnas, D. (1976). On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2(1), 1–9.

Parnas, D. L., and Weiss, D. (1985). Active design reviews: principles and practices. In: *Proceedings of the Eighth International Conference on Software Engineering*, August.

Polya, G. (1956). *How to Solve It: A New Aspect of Mathematical Method* (2nd ed.). Princeton, NJ: Princeton University Press.

Rechtin, E. (1991). *Systems Architecting: Creating and Building Complex Systems*. Upper Saddle River, NJ: Prentice Hall.

RM-ODP (2007). www.rm-odp.net/, a resource page for "Reference model: Open distributed processing framework".

Royce, W. E., and Royce, W. (1991). Software architecture: integrating process and technology. *Quest*, 14(1), 2–15.

Schulte, R. W., and Abrams, C. (2006). *SOA Overview and Guide to SOA Research* (Gartner Research Publication No. G00144894). Stamford, CT: Gartner Inc.

Schulte, R. W., et al. (2005). *Magic Quadrant for Integration Backbone Software, 1H05* (Gartner Research Publication No. G00127186). Stamford, CT: Gartner Inc.

Shaw, M., and Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline*. Upper Saddle River, NJ: Prentice Hall.

Sherwood, D. (2002). *Seeing the Forest for the Trees: A Manager's Guide to Applying Systems Thinking*. London: Nicholas Brealy.

Smith, C. U. (1990). *Performance Engineering of Software Systems*. Boston, MA: Addison-Wesley.

Smith, C. U., and Williams, L. G. (2002). *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Boston, MA: Addison-Wesley.

Sommerville, I. (2004). *Software Engineering* (7th ed.). Boston, MA: Addison-Wesley.

Subramanian, N. (2003). Adaptable software architecture generation using the NFR approach. PhD Thesis, University of Texas at Dallas, May.

Subramanian, N., and Chung, L. (2003). Process-oriented metrics for software architecture adaptability. In: *Proceedings of the Sixth International Workshop on Principles of Software Evolution (IWPSE'03)*.

Tracz, W. (1995). *Confessions of a Used Program Salesman: Institutionalizing Software Reuse* (1st ed.). Reading, MA: Addison-Wesley.

Trenaman, A. (2005). *Using Open Source Software for SOA*. IONA Technology Presentation. http://objectwebcon06.objectweb.org/xwiki/bin/download/Main/DetailedSession/A-Trenaman-Celtix.pdf.

WISCA (2007). http://ieeexplore.ieee.org/xpl/tocresult.jsp?isnumber=4077011&isYear=2007.

Witt, B., Baker, F. T., and Merritt, E. (1994). *Software Architecture and Design: Principles, Models and Methods*. New York: Van Nostrand Reinhold.

Zhu, H. (2005). Building reusable components with service-oriented architectures (pp. 96–101). In: *IEEE International Conference on Information Reuse and Integration*, IRI-2005.

FURTHER READING

ACM/IEEE-CS Joint Task Force on Computing Curricula, Software Engineering. (2004). *Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*. New York: Association for Computing Machinery. www.acm.org/education/curricula.html.

Allen, A. (2000). *Realizing eBusiness with Components*. Reading, MA: Addison-Wesley.

Apple-Architecture. http://developer.apple.com/documentation/Cocoa/Conceptual/AppArchitecture/Concepts/MVC.html.

Asbury, S., Krozak, J., and Wittkopf, K. (2003). *J2EE Anti-Patterns*. New York: John Wiley.

Barbacci, M. R. (1998). *Software Quality Attributes and Architecture Tradeoffs*. Pittsburgh, PA: Software Engineering Institute.

Beckman, K., Coulter, N., Khajenoori, S., and Mead, N. R. (1997). Collaborations: Closing the industry–academia gap. *IEEE Software*, 14(6), 49–57.

Bhattchargya, K., and Johnson, R. A. (1977). *Statistical Concepts and Methods*. New York: John Wiley.

Bloom, B. S. (1956). *Taxonomy of Educational Objectives, Handbook I: The Cognitive Domain*. New York: David McKay Inc.

Boehm, B. W., et al. (1998). A stakeholder win-win approach to software engineering education. In: Balci, O. (Ed.), *Annals of Software Engineering* (pp. 295–321). Amsterdam, The Netherlands: Baltzer Science Publishers.

Bonwell, C., and Eison, J. (1991). *Active Learning: Creating Excitement in the Classroom*. ASHE-ERIC Higher Education Report No. 1. Washington, DC: George Washington University, School of Education and Human Development.

Bromley, D. B. (1990). Academic contributions to psychological counseling: I. A philosophy of science for the study of individual cases. *Counseling Psychology Quarterly*, 3(3), 299–307.

Butler, S. (1999). A client/server case study for software engineering students. In: *Proceedings of the 12th Conference on Software Engineering Education and Training*, New Orleans, LA, 6–8 March.

Carrington, D., McEniery, B., and Johnston, D. (2001). *PSPSM in the Large Class*. Charlotte, NC: CSEET.

Cheesman, J., and Daniels, J. (2000). *UML Components: A Simple Process for Specifying Component Based Software*. Reading, MA: Addison-Wesley.

Ciancarini, P., and Mascolo, C. (1998). Using formal methods for teaching software engineering: A

tool-based approach. *Annals of Software Engineering*, 21, 433–453.

Clancy, M., and Linn, M. (1992). Case studies in the classroom. *SIGCSE Bulletin*, 24(1), 220–224.

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

Cohen, L. (1995). *Quality Function Deployment*. Upper Saddle River, NJ: Prentice Hall.

Collins, A., Brown, J. S., and Newman, S. E. (1989). Cognitive apprenticeship: Teaching the craft of reading, writing, and mathematics. In: Resnick, L. B. (Ed.), *Cognition and Instruction: Issues and Agendas*. Hillsdale, NJ: Lawrence Erlbaum.

Crook, R., Ince, D., Lin, L., and Nuseibeh, B. (2002). Security requirements engineering: When anti-requirements hit the fan. In: *Proceedings of IEEE International Requirements Engineering Conference (RE'02)*, Essen, Germany.

Cusumano, M. A., and Yoffie, D. B. (2002). *Competing on Internet Time: Lessons from Netscape and Its Battle with Microsoft*. New York: Free Press.

D'Souza, D. F., and Cameron Wills, A. (1999). *Objects, Components and Frameworks with UML—The Catalysis Approach*. Reading, MA: Addison-Wesley.

Dalcher, D. (2003). Stories and histories: case study research (and beyond) in information systems failures. In: Whitman, M., and Woszczynski, A. (Eds.), *Handbook for Information Systems Research* (pp. 305–322). Cluj, Romania: Idea Publishing.

Darren, D., and Drevin, L. (2003). Learning from information systems failures by using narrative and antenarrative methods (pp. 137–142). In: *Proceedings of SAICSIT 2003.*

Dawson, R. (2000). Twenty dirty tricks to train software engineers (pp. 209–218). In: Proceedings of the 22nd International Conference on Software Engineering. New York: Association for Computing Machinery.

Emily, O., and van der Andre, H. (2002). Towards game-based simulation as a method of teaching software engineering. In: *Frontiers in Education Conference*, Boston, MA, November.

Fairley, R. (1985). *Software Engineering Concepts*. New York: McGraw Hill.

Ford, G., and Gibbs, N. E. (1996). *A Mature Profession of Software Engineering*. CMU/SEI-96-TR-004. Pittsburgh, PA: Software Engineering Institute.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). Design Patterns. Reading, MA: Addison-Wesley.

Garg, K., and Varma, V. (2007). A study of the effectiveness of case study approach in software engineering education. In: *Proceedings of the 20th Conference on Software Engineering Education and Training (CSEET 2007)*, Dublin, July.

Graff, M. G., and Wyk, K. (2003). *Secure Coding—Principles and Practices*. Cambridge, MA: O'Reilly.

Haley, C. B., Laney, R., Moffett, J., and Nuseibeh, B. (2003). Using trust assumptions in security requirements engineering. In: *Proceedings of 2nd Internal iTrust Workshop on Trust Management in Dynamic Open Systems*, London, September.

Heineman, G. T., and Councill, W. T. (Eds.) (2001). *Component Based Software Engineering—Putting the Pieces Together*. Upper Saddle River, NJ: Pearson Education.

Hilburn, T., et al. (2006). A case study project for software engineering education. In: *36th SEE/IEEE Frontiers in Education Conference*, San Diego.

Howard, M., and LeBlanc, D. (2002). *Writing Secure Code* (2nd ed.). Redmond, WA: Microsoft Press.

ISO/IEC (1998). *FCD 9126-1.2: Information Technology—Software Product Quality. Part 1: Quality Model*.

Java-Sun. http://java.sun.com/blueprints/patterns/MVC-detailed.html.

Jona, K. (2000). *Rethinking the Design of Online Courses*. Keynote Paper, ASCILITE 2000.

Kitchenham, B., Pickard, L. M., and Pfleeger, S. L. (1995). Case studies for method and tool evaluation. *IEEE Software*, 12(4), 52–62.

Losavio, F., et al. (2003). Quality characteristics for software architecture. *Journal of Object Technology*, 2(2), 133–150. www.jot.fm/issues/issue_2003_03/article2.

Mahanti, R., and Mahanti, P. K. (2005). Software engineering education from Indian perspective (pp. 111–117). In: *Proceedings of the 18th Conference on Software Engineering Education & Training*, 18–20 April. Washington, DC: IEEE Computer Society.

Mayr, H. (1997). Teaching software engineering by means of a "virtual enterprise". In *Proceedings of the 10th Conference on Software Engineering*. Washington, DC: IEEE Computer Society.

McDermott, J., and Fox, C. (1999). Using abuse case models for security requirements analysis. In: *Proceedings 15th IEEE Annual Computer Security Applications Conference*. Washington, DC: IEEE Computer Society.

McLoughlin, C., and Luca, J. (2000). Learning through self-direction: The influence of task design in team

based professional knowledge building in an online environment. In: *ASCILITE 2000 Annual Conference*, Southern Cross University, Coffs Harbour, Australia. www.ascilite.org.au/conferences/coffs00/papers/catherine_mcloughlin.pdf.

Morphy, E. (2006). SaaS Apps gaining ground as implementation, ROI concerns dwindle. *CRM Buyer*. www.crmbuyer.com/story/52441.html.

NASSCOM Education Initiatives. www.nasscom.in/Nasscom/templates/LandingPage.aspx?id=5794.

NASSCOM McKinsey Study (2005). *Executive summary on Extending India's Leadership of the Global IT and BPO Industries*. www.nasscom.in/Nasscom/templates/NormalPage.aspx?id=4969.

National Research Council (1999). *How People Learn: Brain, Mind, Experience and School.* Washington, DC: National Academy Press.

O'Brien, L., Bass, L., and Merson, P. (2005). *Quality Attributes and Service-Oriented Architectures*. Pittsburgh, PA: Software Engineering Institute.

Oliver, R. (2000). *When Teaching Meets Learning: Design Principles and Strategies for Web-based Learning Environments that Support Knowledge Construction*. Edith Cowan University, Australia.

OMG-RAS (2005). *Reusable Asset Specification, Version 2.2*. Available at www.omg.org/docs/formal/05-11-02.pdf.

Perfeng. www.perfeng.com/.

Pfleeger, C. P., and Pfleeger, S. L. (2002). *Security in Computing*. Upper Saddle River, NJ: Prentice Hall.

Piessens, F., and Joosen, W. Security Task Force, Department of Computer Science, Katholieke Universiteit Leuven. http://securitytf.cs.kuleuven.ac.be/.

Polajnar, D., and Polajnar, J. (2004). Teaching software engineering through real projects. In: *Western Canadian Conference on Computing Education*.

Raju, P. K., and Sankar, C. S. (1999). Teaching real-world issues through case studies. *Journal of Engineering Education*, 88(4), 501–508.

Remenyi, D., Williams, B., Money, A., and Swartz, E. (2002). *Doing Research in Business and Management: An Introduction to Process and Method*. London: Sage.

Schank, R., Jona, K., and Bareiss, R. (2003). Every curriculum tells a story. In: *Proceedings of the International Conference on Information Technology: Research and Education*.

SEI-CMU-OOD. www.sei.cmu.edu/str/descriptions/oodesign.html.

Senge P. (1990). *The Fifth Discipline: The Art and Practice of the Learning Organization*. New York: Currency Doubleday.

Senge, P. M., Kleiner, A., Roberts, C., Ross, R. B., and Smith, B. J. (1994). *The Fifth Discipline Fieldbook: Strategies and Tools for Building a Learning Organization*. New York: Doubleday.

Shaw M. (1990). Prospects for an engineering discipline of software. *IEEE Software*, 7(6), 15–24.

Shaw M. (2000). Software engineering education: A roadmap (pp. 371–380). In: Finkelstein, A. (Ed.), *Proceedings of the Conference on the Future of Software Engineering*. New York: ACM Press.

Shaw, M. (Ed.) (2005). *Software Engineering for the 21st Century: A Basis for Rethinking the Curriculum*. Technical Report CMU-ISRI-05-108. Pittsburgh, PA: Software Engineering Institute.

Shaw, M., Herbsleb, J. D., and Ozkaya I. (2005). Deciding what to design: Closing a gap in software engineering education (Invited paper). In: *Education and Training Track of 27th International Conference on Software Engineering (ICSE 2005)*, May.

Sims-Knight, J. E., and Upchurch, R. L. (1992). Teaching object-oriented design to nonprogrammers: A progress report. In: *Proceedings of OOPSLA-92 Educators' Symposium*, Vancouver, British Columbia, Canada.

Sindre G., and Opdahl A. L. (2001). Templates for misuse case description. In: *Proceedings of 7th International Workshop on Requirements Engineering, Foundation for Software Quality (REFSQ'2001)*, Switzerland, 4–5 June.

Sivan, A., et al. An implementation of active learning & its effect on the quality of student learning. *Innovations in Education and Training International*, Vol. 3.

Software Architecture Review and Assessment (SARA) Report, Version 1.0. www.rational.com/media/products/rup/sara_report.pdf.

Software Engineering Coordinating Committee of ACM and IEEE. *Guide to the Software Engineering Body of Knowledge*. www.swebok.org.

Software Engineering Institute of Carnegie Mellon University. *Component-based Software Development*. Pittsburgh, PA: Software Engineering Institute. www.sei.cmu.edu/str/descriptions/cbsd.html.

SWEBOK (2004). *Guide to the Software Engineering Body of Knowledge*. A project of the IEEE Computer Society, Professional Practices Committee.

Szyperski, C. (2002). *Component Software*. Reading, MA: Addison-Wesley.

Tockey, S. R. (1999). Recommended skills and knowledge for software engineers (pp. 168–176). In: *Proceedings of the 12th International Conference on Software Engineering Education & Training*.

Varma, V., and Garg, K. (2005). Case studies: The potential teaching instruments for software engineering education. In: *Proceedings of the 5th International Conference on Software Quality*, Australia, 19–20 September.

Varma, V., Garg, K., and Vamsi, S. T. P. (2005). A case study approach towards software engineering education. Accepted in *Software Engineering Research and Practice*, 2005, Las Vegas, June.

Veraart, V. E., and Wright, S. L. (1996). Experience with a process-driven approach to software engineering education. In: *1996 International Conference on Software Engineering: Education and Practice (SE: EP '96)*, 24–27 January.

Weinberg, G. M. (2001). *An Introduction to General Systems thinking* (Silver Anniversary Edition). New York: Dorset House.

Whitehead, A. N. (1929). *The Aims of Education*. New York: Macmillan.

Wohlin, C., & Regnell, B. (1999). Achieving industrial relevance in software engineering education (pp. 16–25). In: *Proceedings of the 12th Conference on Software Engineering Education and Training*.

Wu, D., and Hiltz, S. R. (2004). Predicting learning from asynchronous online discussions. *Journal of Asynchronous Learning Networks*, 8(2), 139–152.

Index