Object-Oriented Software Engineering

Conquering Complex and Changing Systems

Bernd Bruegge & Allen H. Dutoit

Carnegie Mellon University
School of Computer Science
Pittsburgh, USA

Technische Universitaet Muenchen
Institut fuer Informatik
Munich, Germany

Preface

The K2 towers at 8.611 meters in the Karakorum range of the western Himalayas. It is the second highest peak of the world and is considered the most difficult 8000er to climb. An expedition to the K2 typically lasts several months in the summer when the weather is most favorable. Even in summer, snow storms are frequent. An expedition requires thousands of pounds of equipment, including climbing gear, severe weather protection gear, tents, food, communication equipment, and pay and shoes for hundreds of porters. Planning such an expedition takes a significant amount of time in the life of a climber and requires dozens of participants in supporting roles. Once on site, many unexpected events, such as avalanches, porter strikes, or equipment failures will force the climbers to adapt, find new solutions, or retreat. The success rate for expeditions to the K2 is currently less than 40%.

The United States National Airspace System (NAS) monitors and controls air traffic in the United States. The NAS includes more than 18,300 airports, 21 air route traffic control centers, and over 460 control towers. This adds up to more than 34,000 pieces of equipment, including radars, communication switches, radios, computer systems, and displays. The current infrastructure is aging rapidly. The computers supporting the 21 air route traffic control centers, for example, are IBM 3083 mainframes which date back to the early eighties. In 1996, the US government initiated a program to modernize the NAS infrastructure, including improvements such as satellite navigation, digital controller/pilot communications, and a higher degree of automation in controlling the air routes, the order in which aircrafts land, and control of ground traffic as aircrafts move from and to the runways. Modernizing such a complex infrastructure, however, can only be done incrementally. Consequently, while new components offering new functionality are introduced, older components still need to be supported. For example, during the transition period, a controller will have to be able to use both analog and digital voice channels to communicate with pilots. Finally, the modernization of the NAS coincides with a dramatic increase in global air traffic, predicted to double within the next 10 to 15 years. The previous modernizing effort of the NAS, called the Advanced Automation System (AAS), was suspended in 1994 because of software related problems, after missing its initial deadline by several years and exceeding its budget by several billions of dollars.

Both of the above examples discuss complex systems, where external conditions can trigger unexpected changes. Complexity puts the problem beyond the control of any single individual. Change forces participants to move away from well known solutions and to invent new solutions. In both examples, several participants need to cooperate and develop new techniques to address these challenges. Failure to do so results in the failure to reach the goal.

This book is about conquering complex and changing software systems.

The theme

The application domain (mountain expedition planning, air traffic control, financial systems, word processing) usually includes many concepts that software developers are not familiar with. The solution domain (user interface toolkits, wireless communication, middleware, database management systems, transaction processing systems, wearable computers, etc.) is often immature and provides developers with many competing implementation technologies. Consequently, the system and the development project is complex, involving many different components, tools, methods, and people.

As developers learn more about the application domain from their users, they update the requirements of the system. As developers learn more about emerging technologies or about the limitations of current technologies, they adapt the system design and implementation. As quality control finds defects in the system and users request new features, developers modify the system and its associated work products, resulting in continuous change.

Complexity and change represent challenges which make it impossible for any single person to control the system and its evolution. If controlled improperly, complexity and change defeat the solution before its release, even if the goal is in sight. Too many mistakes in the interpretation of the application domain make the solution useless for the users, forcing a retreat from the route or the market. Immature or incompatible implementation technologies result in poor reliability and delays. Failure to handle change introduces new defects in the system and degrades performance beyond usability.

This book reflects more than ten years of building systems and of teaching software engineering project courses. We have observed that students are taught programming and software engineering techniques in isolation, often using small problems as examples. As a result, they are able to solve well defined problems efficiently, but are overwhelmed by the complexity of their first real development experience, when many different techniques and tools need to be used and different people need to collaborate. Reacting to this state of affairs, the typical undergraduate curriculum now often includes a software engineering project course, organized as a single development project.

We wrote this book with such a project course in mind. This book can be used, however, in other situations as well, such as short and intensive workshops or short term R&D projects. We use examples from real systems and examine the interaction between state-of-the art techniques, such as UML (Unified Modeling Language), Java-based technologies, design patterns, design rationale, configuration management, and quality control. Moreover, we discuss project management related issues that are related to these techniques and their impact on complexity and change.

The principles

We teach software engineering following five principles.

Practical experience. We believe that software engineering education must be linked with practical experience. Understanding complexity can only be gained by working with a complex system; that is, a system that no single student can completely understand.

Problem solving. We believe that software engineering education must be based on problem solving. Consequently, there are no right or wrong solutions, only solutions that are better or worse relative to stated criteria. Although we survey existing solutions to real problems and encourage their reuse, we also encourage criticism and the improvement of standard solutions.

Limited resources. If we have sufficient time and resources, we could perhaps build the ideal system. There are several problems with such a situation. First, it is not realistic. Second, even if we had sufficient resources, if the original problem rapidly changes during the development, we would eventually deliver a system solving the wrong problem. As a result we assume that our problem solving process is limited in terms of resources. The acute awareness of scare resources moreover encourages a component-based approach, reuse of knowledge, design, and code. In other words, we support an engineering approach to software development.

Interdisciplinarity. Software engineering is an interdisciplinary field. It requires contributions from areas spanning electrical and computer engineering, computer science, business administration, graphics design, industrial design, architecture, theater, and writing. Software engineering is an applied field. When trying to understand and model the application domain, developers interact regularly with others, including users and clients, some of whom know little about software development. This requires viewing and approaching the system from multiple perspectives and terminologies.

Communication. Even if developers built software for developers only, they would still need to communicate among themselves. As developers we cannot afford the luxury of being able to communicate only with our peers. We need to communicate alternatives, articulate solutions, negotiate trade-offs, review, and criticize others' work. A large number of failures in software engineering projects can be traced to the communication of inaccurate information or to missing information. We must learn to communicate with all project participants, including, most importantly, the client and the end users.

These five principles are the basis for this book. They encourage and enable the reader to address complex and changing problems with practical and state-of-the-art solutions.

The book

This book is based on object-oriented techniques applied to software engineering. It is neither a general software engineering book which surveys all available methods nor a programming book about algorithms and data structures. Instead, we focus on a limited set of techniques and explain their application in a reasonably complex environment, such as a multi-team development project including twenty to sixty participants. Consequently, this book also reflects our biases, our strengths, and our weaknesses. We hope, nevertheless, that all readers will find something they can use. The book is structured into twelve chapters organized into four parts which can be taught as a semester long course.

Part I, *Getting started*, includes three chapters. In this part, we focus on the basic skills necessary for a developer to function in a software engineering context.

- In Chapter 1, *Introduction to Software Engineering*, we describe the difference between programming and software engineering, the current challenges in our discipline, and basic definitions of concepts we use throughout the book.
- In Chapter 2, *Modeling with UML*, we describe the basic elements of a modeling language, UML (Unified Modeling Language), used in object-oriented techniques. We present modeling as a technique for dealing with complexity. This chapter teaches the reader how to read and understand UML diagrams. Subsequent chapters teach the reader how to build UML diagrams to model various aspects of the system. We use UML throughout the book to model a variety of artifacts, from software systems, to processes and work products.
- In Chapter 3, *Project Communication*, we discuss the single most critical activity that developers perform. Developers and managers spend more than half of their time communicating with others, either face-to-face, via email, groupware, video conference, or written documents. While modeling deals with complexity, communication deals with change. We describe the main means of communications, their application, and discuss what constitutes effective communication.

In Part II, *Dealing with complexity*, we focus on methods and technologies which enable developers to specify, design, and implement complex systems.

- In Chapter 4, *Requirements Elicitation*, and Chapter 5, *Requirements Analysis*, we describe the definition of the system from the users' point of view. During requirements elicitation, developers determine the functionality users need and a usable way of delivering it. During requirements analysis, developers formalize this knowledge and ensure its completeness and consistency. We focus on how UML is used to deal with application domain complexity.
- In Chapter 6, *System Design*, we describe the definition of the system from the developers' point of view. During this phase, developers define their design goals

and decompose the system into subsystems. They address global issues such as the mapping of the system onto hardware, the storage of persistent data, and global control flow. We focus on how developers can use design patterns, components, and UML to deal with solution domain complexity.

- In Chapter 7, *Object Design*, we describe the detailed modeling and construction activities related with the solution domain. We refine the requirements and system models and specify precisely the classes that constitute the system. In this chapter, we focus on UML's Object Constraint Language for specifying class interfaces.

In Part III, *Dealing with change*, we focus on methods and technologies which support the control, assessment, and implementation of changes throughout the life cycle.

- In Chapter 8, *Rationale*, we describe the capture of design decisions and their justifications. The models we develop during requirements elicitation, requirements analysis, and system design help us deal with complexity, by providing us with different perspectives on *what* the system should be doing and *how* it should do it. To be able to deal with change, we need also to know *why* the system is the way it is. Capturing design decisions, the evaluated alternatives, and their argumentation enables us to access the rationale of the system.
- In Chapter 9, *Testing*, we describe the validation of system behavior against the system models. Testing activities include unit testing, integration testing, and system testing. We describe several testing techniques such as whitebox, blackbox, path testing, state-based testing, and inspections. Testing detects faults in the system, including those introduced during changes to the system or its requirements.
- In Chapter 10, *Software Configuration Management*, we describe techniques and tools for modeling the project history. Configuration management complements rationale in helping us deal with change. Version management records the evolution of the system. Release management ensures consistency and quality across the components of a release. Change management ensures that modifications to the system are consistent with project goals.
- In Chapter 11, *Project Management*, we describe techniques necessary for initiating a software development project, tracking its progress, and dealing with risks and unplanned events. We focus on organizations, roles, and management activities, that allow a large number of participants to collaborate and deliver a high quality system within planned constraints.

In Part IV, *Starting over*, we revisit the concepts we described in the previous chapters from a process perspective. In Chapter 12, *Software Life Cycle*, we describe software life cycles, such as Boehm's Spiral Model and the Unified Software Development Process, which provide an abstract model of development activities. In this chapter, we also describe the Capability Maturity Model, which is used for assessing the maturity of organizations. We conclude with thoughts about how complexity and change impact the software life cycle.

The topics above are strongly interrelated. To emphasize their relationships, we select an iterative approach. Each chapter consists of five sections. In the first section, we introduce the issues relevant to the topic with an illustrative example. In the second section, we describe briefly the activities of the topic. In the third section, we explain the basic concepts of the topic with simple examples. In the fourth section, we detail the technical activities with examples from real systems. Finally, we describe management activities and discuss typical trade-off. By repeating and elaborating on the same concepts using increasingly complex examples, we hope to provide the reader with an operational knowledge of object-oriented software engineering.

The courses

We wrote this book to support a semester long, software engineering project course for seniors or graduate students. We assume that students have some experience with a programming language such as C, C++, Ada, or Java. We expect that students have the necessary problem solving skills to attack technical problems but we do not expect that they have been exposed yet to complex or changing situations characteristic of system development. This book, however, can also be used for other types of courses, such as short intensive, technical or managerial courses.

Project and senior level courses. A project course should include all the chapters of the book, roughly in the same order. An instructor may consider teaching early in the course introductory project management concepts from Chapter 11, *Project Management* such that students become familiar with planning and status reporting.

Introductory level course. An introductory course with homeworks should focus on the first three sections of each chapter. The fourth section can be used as material for homeworks and can simulate the building of a mini system using paper for UML diagrams, documents, and code.

Short technical course. This book can also be used for a short intensive course geared towards professionals. A technical course focusing on UML and object-oriented methods could use the chapter sequence 1, 2, 4, 5, 6, 7, 8, 9, covering all development phases from requirements elicitation to testing. An advanced course would also include Chapter 10, *Software Configuration Management*.

Short management course. This book can also be used for a short intensive course geared towards managers. A management course focusing on managerial aspects such as communication, risk management, rationale, maturity models, and UML could use the chapter sequence 1, 2, 11, 3, 4, 8, 10, 12.