Model-Driven Development with Executable UML

Dragan Milicev

wrox™

Model-Driven Development with Executable UML

Dragan Milicev

WILEY

Wiley Publishing, Inc.

Model-Driven Development with Executable UML

Published by
Wiley Publishing, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Published by Wiley Publishing, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-0-470-48163-9

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Library of Congress Control Number: 2009927339

About the Author

Dragan Milicev, PhD, is an associate professor at the Department of Computer Science at the University of Belgrade, School of Electrical Engineering. He is the founder and CTO of Serbian Object Laboratories d.o.o. (SOL, `www.sol.rs`), a software development company specializing in building software development tools using model-driven technology, as well as in building custom applications and systems. With 25 years of experience in building complex software systems, he has served as the chief software architect, project manager, or consultant in more than 20 academic and international industrial projects. Of note is the fact that he was Chief Software Architect and Project Manager for most of SOL's projects and all its products: SOLoist, a rapid application model-driven development framework for information systems; SOL UML Visual Debugger, one of the world's first UML visual debuggers, designed for the Poseidon for UML modeling tool; and SOL Java Visual Debugger, a plug-in for Eclipse that enables modeling of test object structures using UML object diagrams. He has published papers in some of the most prestigious scientific and professional journals and magazines, contributing to the theory and practice of model-driven development and UML. He is the author of three previous books on C++, object-oriented programming, and UML, published in Serbia. You may contact him at `dmilicev@etf.rs`.

Acknowledgments

I would like to express my great debt of gratitude to Bran Selic, one of the pioneers and most respected authorities on model-driven software engineering and key contributors to UML, whose careful and thorough review of the manuscript and invaluable comments helped improve this book's structure and technical accuracy.

Special thanks to my colleagues from SOL who have been involved in the development of SOLoist and its application to many industrial projects. Their contribution to the implementation of many ideas presented in this book, as well as their help in making the concepts indeed pragmatic and effective for industrial systems, is deeply appreciated.

I would like to thank my research assistants, as well as my graduate and undergraduate students from the University of Belgrade, School of Electrical Engineering, Department of Computer Science, who took part in the research and development of some ideas and concepts presented in the book. They also offered many valuable comments on the early drafts of the manuscript.

Many thanks to the companies where I have served as a consultant. I have enjoyed the opportunity to brainstorm with colleagues and discuss many ideas presented in this book, as well as the opportunity to apply the ideas in practice.

The work on the development and implementation of the approach presented in this book is partially supported by the Serbian Ministry of Science, under the national program for technology development, grant TR-13001.

Finally, I would like to express my boundless gratitude for forbearance, understanding, and support to my beloved daughter, Mina, sons Miloš and Jovan, and wife, Snežana, to whom I dedicate this book.

Preface

Logical complexity of software systems is one of the main factors causing problems and errors in their planning, design, development, testing, deployment, maintenance, and use. There is a common understanding that building complex software systems requires careful planning, good architectural design, and well-controlled development processes. Many good books and papers, as well as all software engineering curricula, address this issue and yet, many software projects fail, miss their deadlines, or exceed their budgets. Building or maintaining a complex system (be it software or not) is always connected to a risk of mistakes and missed requirements, because humans (who are supposed to build the system) are intrinsically prone to errors when handling too many details and interrelated components at a time.

However, logical complexity is not completely inherent to software systems. On one hand, there is an inevitable component of complexity that is inherent to the very problem domain a software system deals with. The term *essential complexity* refers to that part of logical complexity inherent to the problem domain, and not introduced by a solution or the implementation technology used for it. Essential complexity is, thus, the "natural" part of complexity that cannot be removed and will exist in every solution to a problem, simply because a simple solution to the problem does not exist. However, essential complexity stands in contrast to *accidental complexity*, which arises purely from the implementation technology, tools, and methods applied in a solution. While essential complexity is unavoidable by any approach chosen to solve a problem, accidental complexity is caused by that very approach.

One of the main tasks in software engineering as a discipline is to discover means to minimize accidental complexity. Accidental complexity is to be minimized in any good software architecture, design, and implementation.

Sometimes, accidental complexity can be caused by mistakes such as ineffective planning or project management, or a low priority placed on a project. However, some accidental complexity always occurs as a result of solving any problem. For example, the complexity caused by out-of-memory errors in many programs is an accidental complexity that occurs because someone decided to use a computer to solve the problem [Wiki].

Another significant cause of accidental complexity is a mismatching or immature technology or process selected for the development of a software system. If the available technology (including the language used for software development) requires the developer to write more words or perform more actions to specify a design decision than is really necessary, the artifacts of the development will be accidentally complex. Using an assembly language to implement a non-trivial algorithm, and using a file system interface to build a database application are simple, extreme examples of such mismatching technology. A less obvious example of such accidental complexity is when some code has to be written to specify that a relationship between two objects has to be established when one object is dragged and dropped on the other object. This can be done in an easier and more direct way, by demonstration.

For that reason, raising the level of abstraction of the technology used for development of software systems, and doing so in a way that it better matches the problem domain of those systems, is one of the basic means for guarding against accidental complexity. Raising the level of abstraction is one of the main characteristics of the evolution of software engineering as a discipline. As Bran Selic once said, "There has been no revolution in software engineering since the invention of a compiler." In other words, once we understood that we did not have to talk to the computer in the language its hardware understands, but rather we can do it in a language that is more suitable for us, and which can be automatically translated into the language of the machine, we made the most significant breakthrough in software engineering. Everything since then has basically been all about raising the level of abstraction of the language used to program machines.

The point of raising the level of abstraction is to achieve better *expressiveness*. By using a language that better matches the problem you want to solve, you can say more "facts" in fewer "words." This also means that you can do more with less work. In addition, written words do not have to be the only way to communicate with the machine. Pictures (such as diagrams), motions, and sounds (for example, spoken words) have already been a mode of communication between humans and computer programs, so they can be in software development, too.

This book contributes to the technology of developing one of many kinds of software systems, and proposes a technique that can improve development efficiency by raising the level of abstraction and reducing accidental complexity.

Model-driven development is one approach to raising the level of abstraction that has been successfully exploited for more than a decade. Its basic premise is to use models instead of (solely) code to specify software. Models are generally nonlinear forms, as opposed to code that is inherently linear.[1] *Non-linear* means that models consist of elements that are interrelated in a manner that is freer than a simple sequence where each element can have (at most) two adjacent elements. For that reason, models are usually rendered using visual notations, such as diagrams, instead of pure text.

The software development approach described in this book is model-driven.

The Unified Modeling Language (UML) is a standard language that is used for modeling software. It was proposed in the mid-1990s, and was first standardized in 1997. It is a general-purpose language aimed at modeling all kinds of software systems.

The approach described in this book uses UML as the modeling language.[2] The book follows the definitions and specifications given in the reference [UML2].

However, the scope of this book does not cover all kinds of software systems. Instead, it is limited to one special kind of software systems known as *information systems*. The introductory part of this book defines what is precisely meant by this term. In short, this book focuses on all those applications that have the following properties:

- **A complex conceptual underpinning** — The applications rely on rather rich sets of concepts, properties, and relationships from their problem domains.

[1]Note that code is a sequential form, because it represents a string of characters. Machines and humans read code in a sequential order, one character after another. To improve its readability, machines render code in two-dimensional viewports, but it is still inherently sequential.

[2]As of this writing, the latest UML standard is version 2.2. This book describes this version of UML and is based on the reference [UML2].

- **Large-scale dynamic instantiation** — During exploitation, the applications manipulate large spaces of instances of their concepts and relationships. These instances are dynamically created, modified, retrieved, queried, presented, and deleted. They are traditionally referred to as *data objects*.
- **Persistence of the run-time space** — The applications rely on what is conventionally called a *database* behind.
- **Interactivity** — The applications intensively interact with users and/or other systems to accomplish their purpose through user or machine interfaces.

This book focuses on model-driven development of information systems using UML.

UML is not, however, a fully formal language. This means that its semantics are not defined in an unambiguous way in all its elements. For that reason, UML cannot be used as a language in the same way as traditional programming languages, in which a specification of a software system can be unambiguously interpreted by machines (for example, compiled and executed). In order to be such, a language must have formal, unambiguous semantics — that is, a unique interpretation of each of its concepts that is supposed to have run-time effects.

In addition, UML is a general-purpose modeling language that can be *profiled* for a specific domain of problems. For example, standard UML leaves many so-called *semantic variation points*, which allow a profile to interpret certain language concepts in several ways. A profile can also reduce the set of the language concepts used in a particular problem domain, or extend the semantics of the concepts in a controlled way. This way, a profile can customize the standard language so that it becomes fully formal and, thus, executable. A model built in such a profile represents the implementation of the software at the same time, and, because it can be executed, is not just an informal sketch of the design.

This book proposes and describes one new executable profile of UML for the described application domain. It is but one of several existing profiles of UML with formal and executable semantics, specifically tailored for the domain of information systems.[3]

On one hand, the relational paradigm has been proven and widely accepted as the underpinning technology for building information systems. On the other hand, as another software-development paradigm with significantly more abstract and expressive concepts, object orientation has been successfully used for decades in programming. UML is one of the languages based on the object paradigm.

The marriage of object orientation with information systems development has been predominantly accomplished by using object-oriented programming (OOP) languages to implement behavior (or the so-called *business logic*) upon the underlying relational database, possibly accessed through a data persistence layer that performs object-to-relational mapping. This approach has partially replaced the use of fourth-generation programming languages that directly fit into the relational paradigm. At its current stage of technical development, this widely used approach suffers from discontinuities in development caused by incomplete or informal coupling of the object with the relational paradigm.

[3]For that reason, the term "executable UML" does not refer to any particular executable version of UML, but is rather a generic term that denotes any formal and executable specialization of standard UML. One such executable specialization of standard UML is presented in this book.

This book discusses the problems of these technologies, how they affect development, and how they can be overcome.

In short, this book explores the following:

- A technology for rapid development of one kind of applications referred to as information systems
- The use of the object paradigm and model-driven development of information systems
- One executable profile of UML for model-driven development of information systems

Following are the goals of this book:

- To provide an in-depth tutorial on model-driven development and UML for building information systems
- To show how information systems can be understood better and developed more efficiently by using the object paradigm, model-driven development, and a profile of UML that is formal and executable (rather than the relational paradigm or its incomplete coupling with object orientation)

Note that this book is *not* any of the following:

- **A tutorial on, a reference specification of, or a textbook about the entire general-purpose UML** — This book does cover a major part of UML, but there are still parts of UML that are not covered. Instead, the book focuses on the concepts and parts of UML that are most likely to be needed in building information systems.
- **A complete tutorial on the object paradigm or any traditional OOP language** — However, this book does describe the fundamental concepts of object orientation.
- **A complete tutorial on information systems or all the related technologies** — Part V of this book does, however, provide a condensed recapitulation of the main facts about information systems and the technology of their building, including their architectures, the relational paradigm, entity-relationship, structured analysis, and SQL.
- **A complete textbook on the development process of software systems in general, and information systems in particular** — Part IV of this book does, however, provide a quick practical guide to the proposed development method.
- **A book that describes patterns or other techniques and building blocks for building information systems** — This book does not teach how to build information systems through the use of complex, integrated examples and case studies. Instead, it teaches concepts and principles, using many small, simple, and particular examples for illustration.

Whom This Book Is For

This book will be useful to software practitioners who analyze, specify, design, model, develop, or test information systems. This book is for those who want to improve their knowledge and productivity by exploiting model-driven rapid application development with an executable profile of UML. Readers who might benefit include system analysts, system and software architects, designers, developers, and testers.

The book will also be interesting to researchers who want to explore new software development strategies, methods, and metaphors, especially model-driven development and programming by demonstration. The book introduces some new concepts and ideas that could be interesting to explore further.

This book can also be used as a textbook for higher-education courses on information systems, model-driven software engineering, and UML.

The reader's prior knowledge of the object paradigm or any of the OOP languages is a plus, but is not necessary. This book gradually introduces the basic concepts and principles of object orientation.

Similarly, prior knowledge of the relational paradigm or any of the relational database management systems (RDBMSs) and SQL is not essential, although it is desirable. Part V of the book summarizes these topics for those who are not familiar with them. On the other hand, readers familiar only with these topics will experience a paradigm shift.

Finally, prior knowledge of UML is not needed at all. The book is a complete beginner's tutorial of (a profile of) UML. However, experienced users of UML will also benefit from clarification of many vague concepts of UML and their semantics.

The prerequisite for reading this book is general knowledge of programming. Knowledge and experience in building information systems is a plus, although not essential.

How This Book Is Structured

The book is divided in the following parts:

- **"Introduction" (Part I, Chapters 1–3)** — This part quickly introduces information systems. It then elaborates on traditional technologies of development of information systems and their advantages and drawbacks. This part clearly indicates the main issues with the widespread use of traditional paradigms for building information systems (most notably, relational modeling or entity-relationship modeling), or with incomplete coupling of object orientation (and OOP languages) with relational modeling. The analysis provides the motivation for the approach presented in the book.
- **"An Overview of OOIS UML" (Part II, Chapters 4–6)** — This part is a quick overview of the executable profile of UML proposed in this book, referred to as the *OOIS UML profile*. This part quickly presents the main concepts and ideas that will be described in more detail later in the book.
- **"Concepts" (Part III, Chapters 7–16)** — This central part of the book thoroughly explains the concepts of OOIS UML and their semantics.
- **"Method" (Part IV, Chapters 17–19)** — This part provides a quick guide to the proposed method for applying the OOIS UML profile for building information systems.
- **"Supplemental" (Part V, Chapters 20–24)** — This part provides auxiliary tutorial material for the traditional technology that is widely used for building information systems nowadays, and that is not essential for understanding the main parts of the book. The supplement includes a summary of the general characteristics of information systems, some basics of software engineering processes, the relational paradigm, entity-relationship modeling, structured analysis, and general principles of the object paradigm. These tutorials are provided for the convenience

of the interested readers who are not familiar with these topics and traditional technologies, or as quick reminders for those who are experienced with them.

If you are familiar with the notion of information systems and the traditional technology of their development (including relational databases and entity-relationship), you can simply read the book from its beginning. In the first three chapters of the book, you will find an analysis of the issues that you have probably faced in your work. You will also find explanations of the causes of the issues, while the central part of the book will provide solutions.

If you are not familiar with these traditional technologies, you can still start reading from the beginning. However, you can also skip to the supplement to gain some basic knowledge of the technology you do not know well. However, this knowledge is not essential for understanding of the main part of the book.

Finally, if you are just eager to see what this book is all about, and the new and original information contained herein, simply read Part II and you will get the main idea. Then you can go back or forward as you like.

About the Supporting Software and the Accompanying Site

The method described in this book can be applied even without full-fledged tool support. The author has taken part in several successful industrial projects where only customized off-the-shelf or ad hoc developed tools were used to partially support some activities in the approach (such as UML modeling tools, customized code generators, and object-to-relational mapping frameworks). Even without full-fledged tool support, the proposed approach can boost the productivity and improve the quality of the produced software because of the raised level of abstraction, better expressiveness of the modeling language and its semantics, clear architecture of the software system, and a well-controlled development method.

However, obviously, the full benefit of the proposed approach can be reaped only with strong and full-fledged support of computer-based tools. There can be many different implementations of the proposed UML profile with the appropriate tool support. The author is the inventor and has served as the chief architect of one such tool, named SOLoist[4], which has been developed for and successfully applied to a wide variety of industrial projects since 2000, and which supports many concepts described in this book.

See `www.ooisuml.org` for more discussion about the profile and the method presented in this book, their open issues and further improvements, as well as their implementations and applications to real-world projects.

Conventions

To help you get the most from the text and keep track of what's happening, we've used a number of conventions throughout the book.

[4]SOLoist is a trademark of Serbian Object Laboratories d.o.o. (SOL)

- Each section of the book ends with a summary enclosed in a box like this.

As for styles in the text:

- We *highlight* new terms and important words when we introduce them.
- We show keyboard strokes like this: Ctrl+A.
- We show filenames, URLs, and code within the text like so: `persistence.properties`.
- We present code in two different ways:

```
We use a monofont type with no highlighting for most code examples.
```

```
We use gray highlighting and underlining to emphasize code that is of particular
importance in the present context.
```

Errata

We make every effort to ensure that there are no errors in the text or in the code. However, no one is perfect, and mistakes do occur. If you find an error in one of our books (such as a spelling mistake or faulty model fragment), we would be very grateful for your feedback. By sending in errata you may save another reader hours of frustration and, at the same time, you will be helping us to provide even higher quality information.

To find the errata page for this book, go to `http://www.wrox.com` and locate the title using the Search box or one of the title lists. Then, on the book details page, click the Book Errata link. On this page you can view all errata that has been submitted for this book and posted by Wrox editors. A complete book list (including links to each book's errata) is also available at `www.wrox.com/misc-pages/booklist.shtml`.

If you don't spot "your" error on the Book Errata page, go to `www.wrox.com/contact/techsupport.shtml` and complete the form there to send us the error you have found. We'll check the information and, if appropriate, post a message to the book's errata page and fix the problem in subsequent editions of the book.

p2p.wrox.com

For author and peer discussion, join the P2P forums at `p2p.wrox.com`. The forums are a Web-based system for you to post messages relating to Wrox books and related technologies, and to interact with other readers and technology users. The forums offer a subscription feature to e-mail you topics of interest of your choosing when new posts are made to the forums. Wrox authors, editors, other industry experts, and your fellow readers are present on these forums.

At `http://p2p.wrox.com`, you will find a number of different forums that will help you not only as you read this book, but also as you develop your own applications. To join the forums, just follow these steps:

1. Go to p2p.wrox.com and click the Register link.
2. Read the terms of use and click Agree.
3. Complete the required information to join, as well as any optional information you wish to provide, and click Submit.
4. You will receive an e-mail with information describing how to verify your account and complete the joining process.

You can read messages in the forums without joining P2P. However, in order to post your own messages, you must join.

Once you join, you can post new messages and respond to messages other users post. You can read messages at any time on the Web. If you would like to have new messages from a particular forum emailed to you, click the Subscribe to this Forum icon by the forum name in the forum listing.

For more information about how to use the Wrox P2P, be sure to read the P2P FAQs for answers to questions about how the forum software works, as well as many common questions specific to P2P and Wrox books. To read the FAQs, click the FAQ link on any P2P page.

Contents