

The Essentials of Modern Software Engineering

Free the Practices from the Method Prisons!

Ivar Jacobson

Ivar Jacobson International

Harold “Bud” Lawson

Lawson Konsult AB (deceased)

Pan-Wei Ng

DBS Singapore

Paul E. McMahon

PEM Systems

Michael Goedicke

Universität Duisburg-Essen

ACM Books #25



Copyright © 2019 by the Association for Computing Machinery
and Morgan & Claypool Publishers

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopy, recording, or any other except for brief quotations in printed reviews—without the prior permission of the publisher.

Designations used by companies to distinguish their products are often claimed as trademarks or registered trademarks. In all instances in which Morgan & Claypool is aware of a claim, the product names appear in initial capital or all capital letters. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

The Essentials of Modern Software Engineering: Free the Practices from the Method Prisons!

Ivar Jacobson, Harold “Bud” Lawson, Pan-Wei Ng, Paul E. McMahon, Michael Goedicke

books.acm.org

www.morganclaypoolpublishers.com

ISBN: 978-1-94748-727-7 hardcover

ISBN: 978-1-94748-724-6 paperback

ISBN: 978-1-94748-725-3 eBook

ISBN: 978-1-94748-726-0 ePub

Series ISSN: 2374-6769 print 2374-6777 electronic

DOIs:

10.1145/3277669 Book	10.1145/3277669.3277685 Part III
10.1145/3277669.3277670 Preface	10.1145/3277669.3277686 Chapter 13
10.1145/3277669.3277671 Part I	10.1145/3277669.3277687 Chapter 14
10.1145/3277669.3277672 Chapter 1	10.1145/3277669.3277688 Chapter 15
10.1145/3277669.3277673 Chapter 2	10.1145/3277669.3277689 Chapter 16
10.1145/3277669.3277674 Chapter 3	10.1145/3277669.3277690 Chapter 17
10.1145/3277669.3277675 Chapter 4	10.1145/3277669.3277691 Chapter 18
10.1145/3277669.3277676 Chapter 5	10.1145/3277669.3277692 Part IV
10.1145/3277669.3277677 Chapter 6	10.1145/3277669.3277693 Chapter 19
10.1145/3277669.3277678 Chapter 7	10.1145/3277669.3277694 Chapter 20
10.1145/3277669.3277679 Chapter 8	10.1145/3277669.3277695 Chapter 21
10.1145/3277669.3277680 Part II	10.1145/3277669.3277696 Chapter 22
10.1145/3277669.3277681 Chapter 9	10.1145/3277669.3277697 Chapter 23
10.1145/3277669.3277682 Chapter 10	10.1145/3277669.3277698 Appendix A
10.1145/3277669.3277683 Chapter 11	10.1145/3277669.3277699 References/Index/Bios
10.1145/3277669.3277684 Chapter 12	

A publication in the ACM Books series, #25

Editor in Chief: M. Tamer Özsu, *University of Waterloo*

Area Editor: Bashar Nuseibeh, *The Open University*

This book was typeset in Arnheim Pro 10/14 and Flama using ZzT_EX.

First Edition

10 9 8 7 6 5 4 3 2 1

In every block of marble I see a statue as plain as though it stood before me, shaped and perfect in attitude and action. I have only to hew away the rough walls that imprison the lovely apparition to reveal it to the other eyes as mine see it. —Michelangelo

Standing on the shoulders of a giant . . .

We are liberating the essence from the burden of the whole.

—Ivar Jacobson

Contents

Foreword by Ian Sommerville xvii

Foreword by Grady Booch xix

Preface xxi

PART I THE ESSENCE OF SOFTWARE ENGINEERING 1

Chapter 1 From Programming to Software Engineering 3

- 1.1 Beginning with Programming 4
- 1.2 Programming Is Not Software Engineering 6
- 1.3 From Internship to Industry 8
- 1.4 Journey into the Software Engineering Profession 12
 - What Should You Now Be Able to Accomplish? 15

Chapter 2 Software Engineering Methods and Practices 17

- 2.1 Software Engineering Challenges 17
- 2.2 The Rise of Software Engineering Methods and Practices 18
- 2.3 The SEMAT Initiative 28
- 2.4 Essence: The OMG Standard 29
 - What Should You Now Be Able to Accomplish? 30

Chapter 3 Essence in a Nutshell 31

- 3.1 The Ideas 32
- 3.2 Methods Are Compositions of Practices 32
- 3.3 There Is a Common Ground 34
- 3.4 Focus on the Essentials 37
- 3.5 Providing an Engaging User Experience 37
 - What Should You Now Be Able to Accomplish? 38

Chapter 4	Identifying the Key Elements of Software Engineering	41
4.1	Getting to the Basics	41
4.2	Software Engineering Is about Delivering Value to Customers	43
4.3	Software Engineering Delivers Value through a Solution	45
4.4	Software Engineering Is Also about Endeavors	48
	What Should You Now Be Able to Accomplish?	50
Chapter 5	The Language of Software Engineering	53
5.1	A Simple Practice Example	53
5.2	The Things to Work With	54
5.3	Competencies	61
5.4	Things to Do	62
5.5	Essentializing Practices	63
	What Should You Now Be Able to Accomplish?	65
Chapter 6	The Kernel of Software Engineering	67
6.1	Organizing with the Essence Kernel	67
6.2	The Essential Things to Work With: The Alphas	69
6.3	The Essential Things to Do: The Activities	72
6.4	Competencies	75
6.5	Patterns	77
	What Should You Now Be Able to Accomplish?	81
Chapter 7	Reflection on Theory	83
7.1	Where's the Theory for Software Engineering?	84
7.2	Uses of Theory	87
7.3	Essence Is a General, Descriptive Theory of Software Engineering	87
7.4	Toward a General Predictive Theory of Software Engineering	91
7.5	A Theoretical Foundation Helps You Grow	93
	What Should You Now Be Able to Accomplish?	94
	Postlude to Part I	94
	Recommended Additional Reading	95
Chapter 8	Applying Essence in the Small—Playing Serious Games	97
8.1	Progress Poker	99
8.2	Chasing the State	105
8.3	Objective Go	108
8.4	Checkpoint Construction	111

- 8.5 Reflection 113
- What Should You Now Be Able to Accomplish? 114

PART II DEVELOPING SOFTWARE WITH ESSENCE 115

Chapter 9 Kick-Starting Development Using Essence 117

- 9.1 Understand the Context Through the Lens of Essence 118
- 9.2 Agreeing on the Development Scope and Checkpoints 122
- 9.3 Agreeing on the Most Important Things to Watch 124
- What Should You Now Be Able to Accomplish? 126

Chapter 10 Developing with Essence 127

- 10.1 Planning with Essence 132
- 10.2 Doing and Checking with Essence 138
- 10.3 Adapting a Team's Way of Working with Essence 140
- 10.4 How the Kernel Helps Adapt Their Way of Working 141
- What Should You Now Be Able to Accomplish? 143

Chapter 11 The Development Journey 145

- 11.1 Visualizing the Journey 145
- 11.2 Ensuring Progress and Health 146
- 11.3 Dealing with Anomalies 148
- What Should You Now Be Able to Accomplish? 149

Chapter 12 Reflection on the Kernel 151

- 12.1 Validity of the Kernel 151
- 12.2 Applying the Kernel Effectively 151
- What Should You Now Be Able to Accomplish? 152
- Postlude 153
- Recommended Additional Reading 153

PART III SMALL-SCALE DEVELOPMENT WITH PRACTICES 155

Chapter 13 Kick-Starting Development with Practices 157

- 13.1 Understand the Context Through the Lens of Essence 158
- 13.2 Agree upon Development Scope and Checkpoints 159
- 13.3 Agree upon Practices to Apply 165

- 13.4 Agree upon the Important Things to Watch 167
- 13.5 Journey in Brief 169
 - What Should You Now Be Able to Accomplish? 170

Chapter 14 Running with Scrum 171

- 14.1 Scrum Explained 171
- 14.2 Practices Make a Software Engineering Approach Explicit and Modular 173
- 14.3 Making Scrum Explicit Using Essence 174
- 14.4 Scrum Lite Alphas 179
- 14.5 Scrum Lite Work Products 182
- 14.6 Scrum Lite Roles 184
- 14.7 Kick-Starting Scrum Lite Usage 187
- 14.8 Working with Scrum Lite 188
- 14.9 Reflecting on the Use of Scrum with Essence 198
 - What Should You Now Be Able to Accomplish? 202

Chapter 15 Running with User Story Lite 203

- 15.1 User Stories Explained 204
- 15.2 Making the User Story Lite Practice Explicit Using Essence 207
- 15.3 User Story Lite Alphas 208
- 15.4 User Story Lite Work Products 209
- 15.5 Kick-Starting User Story Lite Usage 211
- 15.6 Working with User Story Lite 211
- 15.7 The Value of the Kernel to the User Story Lite Practice 215
 - What Should You Now Be Able to Accomplish? 218

Chapter 16 Running with Use Case Lite 221

- 16.1 Use Cases Explained 222
- 16.2 Making the Use Case Lite Practice Explicit Using Essence 227
- 16.3 Use Case Lite Alphas 230
- 16.4 Use Case Lite Work Products 233
- 16.5 Kick-Starting Use Cases Lite to Solve a Problem Our Team Is Facing 237
- 16.6 Working with Use Cases and Use-Case Slices 240
- 16.7 Visualizing the Impact of Using Use Cases for the Team 244
- 16.8 Progress and Health of Use-Case Slices 245
- 16.9 User Stories and Use Cases—What Is the Difference? 246
 - What Should You Now Be Able to Accomplish? 248

Chapter 17	Running with Microservices	249
17.1	Microservices Explained	250
17.2	Making the Microservice Practice Explicit Using Essence	252
17.3	Microservices Lite	256
17.4	Microservices Lite Alphas	257
17.5	Microservices Lite Work Products	259
17.6	Microservices Lite Activities	267
17.7	Visualizing the Impact of the Microservices Lite Practice on the Team	270
17.8	Progress and Health of Microservice Development	271
	What Should You Now Be Able to Accomplish?	272
Chapter 18	Putting the Practices Together: Composition	275
18.1	What Is Composition?	276
18.2	Reflecting on the Use of Essentialized Practices	282
18.3	Powering Practices through Essentialization	283
	What Should You Now Be Able to Accomplish?	284
	Recommended Additional Reading	284
PART IV	LARGE-SCALE COMPLEX DEVELOPMENT	287
Chapter 19	What It Means to Scale	289
19.1	The Journey Continued	289
19.2	The Three Dimensions of Scaling	291
	What Should You Now Be Able to Accomplish?	294
Chapter 20	Essentializing Practices	295
20.1	Practice Sources	295
20.2	Monolithic Methods and Fragmented Practices	296
20.3	Essentializing Practices	298
20.4	Establishing a Reusable Practice Architecture	299
	What Should You Now Be Able to Accomplish?	303
Chapter 21	Scaling Up to Large and Complex Development	305
21.1	Large-Scale Methods	306
21.2	Large-Scale Development	308
21.3	Kick-Starting Large-Scale Development	309
21.4	Running Large-Scale Development	315

21.5	Value of Essence to Large-Scale Development	322
	What Should You Now Be Able to Accomplish?	324

Chapter 22 Reaching Out to Different Kinds of Development 325

22.1	From a Practice Architecture to a Method Architecture	326
22.2	Establishing a Practice Library within an Organization	328
22.3	Do Not Ignore Culture When Reaching Out	330
	What Should You Now Be Able to Accomplish?	331

Chapter 23 Reaching Out to the Future 333

23.1	Be Agile with Practices and Methods	335
23.2	The Full Team Owns Their Method	337
23.3	Focus on Method Use	337
23.4	Evolve Your Team's Method	338
	What Should You Now Be Able to Accomplish?	339
	Recommended Additional Reading	339

Appendix A A Brief History of Software and Software Engineering 341

References	349
Index	353
Author Biographies	369

Foreword by Ian Sommerville

There's some debate over whether the term *software engineering* was first coined by Margaret Hamilton at NASA in the 1960s or at the NATO conference at the end of that decade. It doesn't really matter because 50 years ago it was clear that software engineering was an idea whose time had come.

Since then, developments in software engineering have been immense. Researchers and practitioners have proposed many different methods and approaches to software engineering. These have undoubtedly improved our ability to create software, although I think it is fair to say that we sometimes don't really understand why. However, we have no basis for comparing these methods to see if they really offer anything new and we can't assess the limitations of software engineering methods without experiencing failure. Although we are a lot better at developing software than we were in the 20th century, it is still the case that many large software projects run into problems and the software is delivered late and fails to deliver the expected value.

The SEMAT initiative was established with the immense ambition to rethink software engineering. Rather than inventing another new method, however, Ivar Jacobson and his collaborators went back to first principles. They examined software engineering practice and derived a common underlying language and kernel (Essence) that could be used for discussing and describing software engineering. Essence embodies the essential rather than the accidental in software engineering and articulates new concepts such as alphas that are fundamental to every development endeavor.

Essence is not a software engineering method but you can think of it as a meta-method. You can use it to model software engineering methods and so compare them and expose their strengths and weaknesses. More importantly, perhaps, Essence can also be the starting point for a new approach to software engineering. Because of the universality of the concepts that it embodies, Essence can be used across a much wider range of domains than is possible with current methods.

It wisely separates the notion of specific practices, such as iterative development, from fundamental concepts so it can be used in a variety of settings and application domains.

The inventors of Essence understand that the value of Essence can only be realized if it is widely used. Widespread use and experience will also expose its limitations and will allow Essence to evolve and improve. This book is an important contribution to transferring knowledge about Essence from specialists to a more general audience. Although notionally aimed at students, it provides an accessible introduction to Essence for all software engineers.

Organized into four parts, the first three parts focus squarely on using Essence as a means of thinking about, planning, and describing software development. Using real but manageable examples, Parts I and II of the book cover the fundamentals of Essence and the innovative use of serious games to support software engineering. Part III explains how current practices such as user stories, use cases, Scrum, and microservices can be described using Essence and shows how their activities can be represented using the Essence notions of cards and checklists. Part IV is perhaps more speculative but offers readers a vision of how Essence can scale to support large, complex systems engineering.

Software engineering has been both facilitated and hampered by the rate of technological innovation. The need to build software for new technologies has led to huge investment in the discipline but, at the same time, has made it difficult to reflect on what software engineering really means. Now, 50 years on, Essence is an important breakthrough in understanding the meaning of software engineering. It is a key contribution to the development of our discipline, and I'm confident that this book will demonstrate the value of Essence to a wider audience. It, too, is an idea whose time has come.

Ian Sommerville

Emeritus Professor of Software Engineering at St. Andrews University, Scotland. For more than 20 years, his research was concerned with large-scale complex IT systems. He is the author of a widely used textbook on software engineering, titled *Software Engineering*, first published in 1982, with the 10th edition published in 2015.

Foreword by Grady Booch

The first computers were human; indeed, the very noun “computer” meant “one who computes or calculates” (and most often those ones were women).

My, how the world has changed.

Computing has woven itself into the interstitial spaces of society. Software-intensive systems power our cars and airplanes; they serve as our financial conduits; they track our every action; they fight our wars; they are as intimate as devices we hold close to us or even within us and as grand as the wanderers we have flung into space and that now inhabit other planets and venture to other stars. There is no other invention in the history of humanity that has such a potential to amplify us, diminish us, and perhaps even replace us.

I have often observed that the entire history of software engineering can be characterized as the rising levels of abstraction. We witness this in our programming languages, in our tools, in our frameworks, in the very ways with which we interact with software-intensive systems . . . and even in the ways in which we craft these systems. This is the world of software engineering methods.

I am proud and humbled to call myself a friend of Ivar Jacobson. The two of us, along with Jim Rumbaugh, were at the center of a sea change in the way the world develops and deploys software-intensive systems. We got some things right; we got some things wrong. But, most important, we helped to codify the best practices of software engineering in their time. Indeed, that was an incredibly vibrant time in the history of software engineering, wherein many hundreds if not thousands of others were struggling with how to codify the methods by which systems of importance could best be built.

The nature of software development has changed—as it should and as it will again—and even now we stand at an interesting crossroads in the field. Agile methods have proven themselves, certainly, but we are at the confluence of technical and economic forces that bring us again to a very vibrant point in time. As the Internet of Things brings computing to billions of devices, as computational resources grow

in unceasing abundance, and as deep learning and other forms of artificial intelligence enter the mainstream, now is the time to establish a sound foundation on which we can build the next generation of software-intensive systems that matter.

In a manner of speaking, one might say that the *essence* of Essence is its powerful mastery of the fundamental abstractions of software engineering. I saw in Ivar the seeds of Essence in the early days of working with him and Jim on the UML, and so now it is wonderful to see this work in its full flowering. What you hold in your hands (or on your computer or tablet, if you are so inclined) represents the deep thinking and broad experience of Ivar; information that you'll find approachable, understandable, and—most importantly—actionable.

Enjoy the journey; it will make a difference for the good.

Grady Booch

IBM Fellow, ACM Fellow, IEEE Fellow, recipient of the BCS Ada Lovelace Award, and IEEE Computer Pioneer.

Preface

We have developed software for many years, clearly more than 50 years. Thousands of books and many more papers have been written about how to develop software. Almost all teach one particular approach to doing it, one which the author thinks is the best way of producing great software; we say each author has canned his/her method. Most of these authors have some interesting ideas, but none can help you in all the circumstances you will be faced with when you develop software. Even the most modern books take this approach of presenting and selling “the one true way” of doing it. Unless you are a world leader ready to impose your own true way of doing it, all other top experts in the world seem to be in agreement that this proprietary approach is not the way to teach software development to students.

You now have in front of you a book that will teach you modern software engineering differently from how the subject has been taught since its infancy. On one hand, it stands on the shoulders of the experience we have gained in the last 50 years or more. On the other hand, it teaches the subject in a universal and generic way. It doesn't teach you one particular way of developing software, but it teaches you *how to create* one way of working that matches your particular situation and your needs. The resulting way of working that you create is easy to learn (intuitive), easy to adopt (by a team), easy to change (as you learn more), and fun to work with thanks to its user experience being based on games and playing cards.

It is worth repeating: This book does not primarily teach you one particular way of developing great software; rather, it teaches you how to create such a way of working that should result in great software.

How This Book Is Different from Other Software Engineering Textbooks

On the surface this book looks like most other books in software engineering (and there are many of them; some are excellent books). It describes many important

aspects of software engineering and how a typical software engineering initiative resulting in a new or improved software product takes place. However, underneath the surface, this book is fundamentally different. The things being described are selected because they are prevalent in every software engineering initiative. They are the essential things to work with, the essential things to do, and the essential competencies needed when you develop software. They are not just examples of things or typical things. They are selected because they are the things that underpin all recognized ways of developing software. The selection has been made by a group of experts from around the world representing academia, research, and industry, under the auspices of an international group called Object Management Group that gave rise to the Essence standard.¹

Essence addresses, first and foremost, a number of serious challenges we have in the software industry today, one of which is that for 50 years we have had a war between the canned methods (but there are many more challenges, which we will discuss in the book). In addressing these issues, Essence has made it possible to systematically improve the way we work, which should result in better software—faster and cheaper. However, this will have to wait to be discussed until you have gone deeper into the book.

Finally, the following summary can be repeated over and over again.

- Essence supports people when working with methods and it helps people while they actually work developing software.
- Essence is not yet another method. It is many things but not a method competing with any other method.
- It is a foundation to be used to describe methods effectively and efficiently.
- It is a thinking framework to be used when creating your method or using your method, whether it is explicit or tacit.
- It can help you in a method-agnostic way to measure progress and health in your endeavor.²
- It can help you, if you have challenges, to find root causes of the problems with your endeavor.

1. Essence has been likened to the DNA of software engineering or the periodic table in chemistry.

2. Throughout this book, except for the cases where the term *project* is more appropriate for historical reasons, we use the term *endeavor*. This is because not all software development occurs within the context of a formal project.

How This Book Can Help Students

If you are a student, this book will play a significant role in your career, because from this book you will learn the fundamentals of the complex discipline of software engineering. Even if you are not a student, you will rediscover your discipline in a way you never expected. This is no ordinary software engineering textbook. What you will learn from this book you can take with you wherever you go, for the rest of your software engineering career.

Other books will help you learn the latest technologies, practices, and methods. While you will need that kind of information as you go through your career, their value will fade over time as new technologies, practices, and methods come into play. There is nothing wrong with that. Part of our profession is continuous improvement and we encourage and expect that to go on forever.

What You Will Learn from This Book

So that you have the right expectations, we want to tell you what you can expect to learn from this book.

- You will learn what are the essentials of software engineering presented as a common ground.
- You will learn a simple, intuitive language by which you can describe specific ways of working, called practices, using the common ground as a vocabulary.
- You will learn how the common ground can be used to assess the progress and health of your software development endeavors no matter how simple or complex.
- You will learn “lite” versions of a number of practices that are popular at the time of writing this book, *but they are only meant as examples to demonstrate how to use the common ground and the language to describe practices.*
- You will learn how to improve your way of working by adding or removing practices, as and when the situation demands.
- You will learn how to improve communication with your teammates.

To be clear, this is what you won’t learn from this book.

- You will not learn any fully developed practices to be used in a real endeavor (in a commercial production environment), since what we teach here is not

intended for that purpose. To learn practices that will work in such an environment, you need to go to practice libraries such as the Ivar Jacobson International practice library (<https://practicelibrary.ivarjacobson.com/start>) or, if the practices are not yet essentialized, you will have to go to books or papers written about these practices.

- You will not learn the latest technologies, practices, and methods.

This book is about learning a foundation that underlies all practices and methods that have come and gone during the last 50 years, and all that will likely come and go over the next 50 years. What you learn from this book you can take with you, and it will continue to help you grow throughout your software engineering career.

Our Approach to Teaching in This Book

We also want to share with you a little bit about the approach to teaching software engineering that we use in this book. While we do share some of the history of software engineering in Part I and in the appendix, our general approach throughout the book is a bottom-up approach instead of a top-down one. The “user” is a young student and he/she is presented with more and more advanced use cases of software development—from small systems to large systems. Or said in another way, we present the essence of software engineering through the eyes of a young student who moves from introductory courses into the industry. This approach will help you understand how software engineering is often first viewed by new software developers and how their perceptions and understanding of software engineering grow with their experiences.

So with this brief introduction, you are now ready to start your exciting journey toward the essentials of modern software engineering. During the journey, you will pass through the following.

Part I, The Essence of Software Engineering. Here, we introduce the student to software engineering and to the Essence standard.

Part II, Applying Essence in the Small. Here, Essence is first used to carry out some simple, small, but very useful practices. They are so small that they could be called mini-practices, but we call them games—serious games. They are highly reusable when carrying out practices resulting in, for instance, software products.

Then in the rest of this part we advance the problem and consider building some real but rather small software. We make the assumption that the given team members have worked together before, so they have tacit knowledge

about the practices they use and don't need any additional explicit guidance in the form of described practices.

Part III, Small-Scale Development with Practices. We use practices defined on top of the kernel to provide further guidance to small teams.

Part IV, Large-Scale Complex Development. To describe how to develop large software systems is far too complex for a textbook of this kind. However, we do explain the impact large teams and organizations have on the practices needed and how they are applied.

Appendix, A Brief History of Software Engineering.

On our website, <http://software-engineering-essentialized.com>, you are provided with additional training material and exercises associated with each part of the book. This website will be continuously updated and will provide you with additional insight. As you gain experience, we hope you will also be able to contribute to this growing body of knowledge.

How This Book Can Free the Practices from the Method Prisons and Why This Is Important

In 1968, more than 50 years ago, the term *software engineering* was coined to address the so-called software crisis. Thousands of books have been written since then to teach the “best” method as perceived by their authors. Some of them have been very successful and inspired a huge number of teams to each create their own method. The classical faith typically espoused by all these popular methods has been that the previous popular method now has become completely out of fashion and must be replaced by a new, more fashionable method. People have been swinging with these trends and, apart from learning something new, each time they must also relearn what they already knew but with just a new spin to it.

The problem is that among all these methods there has been almost nothing shared, even if in reality much more has been shared than what separated them. What they shared was what we will call practices—some kind of mini-methods. Every method author (if very successful, each became a guru) had their own way of presenting their content so that other method authors couldn't simply reuse it. Instead, other authors had to reinvent the wheel by describing what could have been reusable—the practices—in a way that fit these other authors' presentation styles. Misunderstandings and improper improvements happened and the method war was triggered. It is still going on. Instead of “standing on one another's shoulders,” these various authors are “standing on one another's toes.”

This book will show how reusable practices can be liberated from the methods that use them—their method prisons. Free the practices from the method prisons!

Acknowledgments

Special thanks and acknowledgment goes to Svante Lidman and Ian Spence for their work on the first Essence book [Jacobson et al. 2013a], from which some pieces of text have been used, to Mira-Kajko-Mattson for her role in the original shaping of this book, to Pontus Johnson for his work on theory in Part I, Chapter 7 and to Barbora Buhnova for in particular her clear and accurate writing of the goal and the accomplishments paragraphs in each chapter of the book. All these contributions improved the clarity of the book as a whole.

The authors also want to recognize and thank all the people that worked with us in creating the OMG Essence standard and in working on its use cases. Without these individuals' work this book would never have been written:

- For founding the SEMAT (Software Engineering Method And Theory) community in 2009 and later leading it: Apart from Ivar Jacobson, the founders were Bertrand Meyer and Richard Soley. June Park chaired the SEMAT community from 2012 to 2016 and Sumeet Malhotra from 2016 until now.
- For serving as members of the Advisory Board chaired by Ivar Jacobson: Scott Ambler, Herbert Malcolm, Stephen Nadin, Burkhard Perken-Colomb.
- For supporting the foundation of the SEMAT initiative and its call for action:
 - Individuals: Pekka Abrahamsson, Scott Ambler, Victor Basili, Jean Bézivin, Robert V. Binder, Dines Bjorner, Barry Boehm, Alan W. Brown, Larry Constantine, Steve Cook, Bill Curtis, Donald Fire-smith, Erich Gamma, Carlo Ghezzi, Tom Gilb, Robert L. Glass, Ellen Gottesdiener, Martin Griss, Sam Guckenheimer, David Harel, Brian Henderson-Sellers, Watts Humphrey, Ivar Jacobson, Capers Jones, Philippe Kruchten, Harold “Bud” Lawson, Dean Leffingwell, Robert Martin, Bertrand Meyer, Paul Nielsen, James Odell, Meilir Page-Jones, Dieter Rombach, Ken Schwaber, Alec Sharp, Richard Soley, Ian Sommerville, Andrey Terekhov, Fuqing Yang, Edward Yourdon.
 - Corporations: ABB, Ericsson, Fujitsu UK, Huawei, IBM, Microsoft Spain, Munich RE, SAAB, SICS, SINTEF, Software Engineering Institute (SEI), Tata Consulting Services, Telecom Italia, City of Toronto, Wellpoint.

- Academics: Chalmers University of Technology, Florida Atlantic University, Free University of Bozen Bolzano, Fudan University, Harbin Institute of Technology, Joburg Centre for Software Engineering at Wits University, KAIST, KTH Royal Institute of Technology, National University of Colombia at Medellin, PCS—Universidade de São Paulo, Peking University, Shanghai University, Software Engineering Institute of Beihang University, Tsinghua University, University of Twente, Wuhan University.
- For developing what eventually became the Essence standard with its use cases and for driving it through the OMG standards process: Andrey Bayda, Arne Berre, Stefan Bylund, Dave Cuningham, Brian Elvesæter, Shihong Huang, Carlos Mario Zapata Jaramillo, Mira Kajko-Mattson, Prabhakar R. Karve, Tom McBride, Ashley McNeille, Winifred Menezes, Barry Myburgh, Gunnar Overgaard, Bob Palank, June Park, Cecile Peraire, Ed Seidewitz, Ed Seymour, Ian Spence, Roly Stimson, Michael Striewe.
- For organizing SEMAT Chapters around the world: Doo-Hwan Bae, Steve Chen, Zhong Chen, Barry Dwolatsky, Gorkem Giray, Washizaki Hironori, Debasish Jana, Carlos Mario Zapata Jaramillo, Pinakpani Pal, Boris Pozin.
- For co-chairing the “Software Engineering Essentialized” project with Ivar Jacobson: Pekka Abrahamsson. This project develops training material, quizzes, exercises, certification, games, essentialized practices, etc. to support teachers giving classes based on this book.

From the outset of the writing of this book, the authors were aware of the fundamental change they proposed to the education in software engineering. Therefore, they wanted the book to be meticulously reviewed before publication. The book has been reviewed in 5 phases, each being presented as a draft. About 1000 comments have been given by more than 25 reviewers and each comment has been discussed and acted upon. We are very grateful for the help we received from the following people (alphabetically ordered) in making this a book we are very proud of: Giuseppe Calavaro, A. Chamundeswari, Gökem Giray, Emanuel Grant, Debasish Jana, Eréndira Miriam Jiménez Hernández, Reyes Juárez-Ramírez, Winifred Menezes, Marcello Missiroli, Barry Myburgh, Anh Nguyen Duc, Hanna Oktaba, Don O'Neill, Gunnar Overgaard, Pinakpani Pal, Cecile Peraire, Boris Pozin, Antony Henao Roqueme, Anthony Ruocco, Vladimir Savic, Armando Augusto Cabrera Silva, Kotrappa Sirbi, Nebojsa Trninic, Hoang Truong Anh, Eray Tüzün, Murat Paşa Uysal,

Ervin Varga, Monica K. Villavicencio Cabezas, Bernd G. Wenzel, Carlos Mario Zapata Jaramillo.

As you can see from these acknowledgments, many people have contributed to where we are today with Essence and its usage. Some people have made seminal technical contributions without which we wouldn't have been able to create a kernel for software engineering. Some other people have contributed significant time and effort to move these technical contributions into a high-quality standard to be widely adopted. Some people have been instrumental in identifying the vision and leading the work through all the pitfalls that an endeavor can encounter when it is as huge as the SEMAT in fact is. Finally, some people have made huge efforts and with high passion marketed the work and the result to break through the barriers that fundamentally new ideas always face. We have not made an effort to rank all these contributions here, but we hope all these individuals are assured that we know about them and we are tremendously grateful for all they have done.

We would also like to thank the team at Windfall Software for carefully copy editing and preparing the content of this book. We are especially grateful to their professional developmental editor, who was instrumental in this endeavor and put in a huge effort to achieve this high-quality result.




PART

THE ESSENCE OF SOFTWARE ENGINEERING

We live at an exciting time in the history of computer and network technologies where software has become a dominant aspect of our everyday life. Wherever you look and wherever you turn, software is there. It is in almost everything you use and affects most everything you do. Software is in many things such as microwaves, ATMs, smart TVs, machines running vehicles, and factories, as well as being utilized in all types of organizations.

Although software provides many opportunities for improving many aspects of our society, it presents many challenges as well. One of them is development, deployment, and sustainment of high-quality software on a broad scale. Another is the challenge of utilizing technology advancements in new domains, for instance, intelligent homes and Smarter Cities. Here, the evolution of the mobile internet, apps, the internet of things (IoT), and the availability of big data and cloud computing, as well as the application of artificial intelligence and deep learning, are some of the latest “game-changers” with more still to come.

This book provides you with fundamental knowledge you will need for addressing the challenges faced in this era of rapid technology change. Part I will introduce you to software engineering through the lens of a kernel of fundamental concepts that have been provided by the Object Management Group’s standard called Essence 1. Essence is rapidly becoming a “lingua franca” for software engineering. The authors are convinced that this approach will provide a perspective that will be a lasting contribution to your knowledge base and prepare you to participate in teams that can develop and sustain high-quality software.



From Programming to Software Engineering

This chapter sets the scene with respect to the relationship between programming and software engineering. The important issue is that software engineering is much more than just programming. Of course, the running system created by an act of programming is an essential and rewarding ingredient of what the right system will become, and it is important that the reader is actually able to use and apply a programming language to create a program, at least a small one. But is it by no means everything. Thus, this chapter

- introduces the notion of software development and that it is more than just putting a program together;
- shows what additionally is needed beyond programming, i.e., shows the differences between programming, software development, and software engineering;
- shows the motivations for the discipline of software engineering;
- introduces some important elements of software engineering that actually show the differences between software engineering and programming, and shows how they relate to each other.

What is fascinating about this aspect of software development is that it is more than just programming. Rather, it is to learn the whole picture and as a software engineer to solve a problem or exploit an opportunity that the users may have.

As a new student, understanding what software engineering is about is not easy, because there is no way we can bring its realities and complexities into the student's world. Nevertheless, it is a student's responsibility to embark on this journey of learning and discovery into the world of software engineering.

Throughout this entire book, we will trace the journey of a young chap, named Smith, from his days in school learning about programming through to becoming

Sidebar 1.1 Programming

Programming is used here as a synonym for *implementation* and *coding*. From Wikipedia we quote: “Related tasks include testing, debugging, and maintaining the source code, These might be considered part of the programming process, but often the term *software development* is used for this larger process with the term *programming*, *implementation*, or *coding* reserved for the actual writing of source code.”

a software engineering professional and continuing his on-going learning process in this ever-changing and growing field. In a way, we are compressing time into the pages of this book. If you are a new student, you are considered to be the primary audience for this book. Smith will be your guide to the software engineering profession, to help you understand what software engineering is about. If you are already a software engineer by profession, or you teach and coach software engineering, you can reflect on your own personal journey in this exciting profession. As an experienced developer you will observe an exciting and fundamentally new way to understand and practice software engineering. Regardless of your current personal level of experience, through Smith’s experiences we will distill the essence of software engineering.

1.1 Beginning with Programming

The focus of our book is not about programming (see Sidebar 1.1), but about software engineering. However, understanding programming is an obvious place to start. Before we delve deeper into it, we should clarify the relation of programming to software development and to software engineering.

Thus we have chosen the following.

- *Programming* stands for the work related to implementation or coding of source code.
- *Software development* is the larger process which, apart from programming, includes working with requirements, design, test, etc.
- “*Software engineering* combines engineering techniques with software development practices” (from Wikipedia). Moving from development to engineering means more reliance on science and less on craft, which typically manifests itself in some form of description of a designated way of working and higher-level automation of work. This allows for repeatability and consistency from project to project. Engineering also means that teams, for example, learn as they work and continuously improve their way of work-

ing. Thus, stated in simple terms, *software engineering is bringing engineering discipline to software development*.

Going forward, when introducing software engineering we will mean the larger subject of “software development + engineering,” implicitly understood without specifically separating out the two parts. This will be so even if in many cases the discussion is more about the development aspect, because the approach we take is chosen to facilitate the other aspect—engineering. When we sometimes talk about software development we want to be specific and refer to the work: the activities or the practices we use. We will not further try to distinguish these terms, so the reader can in many cases see them as synonyms.

As a frequent user of applications like Facebook, Google, Snapchat, etc., whether on his laptop or his mobile, Smith knew that software forms a major component in these products. From this, Smith became strongly interested in programming and enrolled in a programming course where he started to understand what program code was and what coding was all about. More importantly, he knew that programming was not easy. There were many things he had to learn.

The very first thing Smith learned was how to write a program that displays a simple “Hello World” on his screen, but in this case, we have a “Hello Essence!”, as in Figure 1.1. Through that he learned about programming languages,

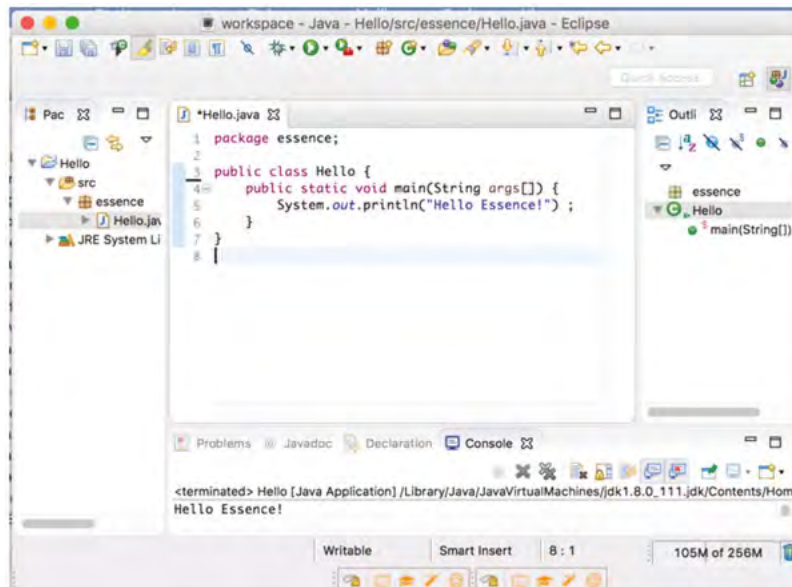


Figure 1.1 Hello Essence.

programming libraries, compilers, operating systems, processes and threads, classes, and objects. These are things in the realm of computer technology. We expect that you, through additional classes, will have learned about these things. We also expect you as a student to have some knowledge of these things as a prerequisite to reading this book. We expect that you have some knowledge of programming languages like Java and JavaScript.

1.2 Programming Is Not Software Engineering

However, Smith quickly learned that programming on its own is not software engineering. It is one thing to develop a small program, such as the “Hello Essence” program; it is a different thing to develop a commercial product.

It is true that some fantastic products such as those that gave birth to Apple, Microsoft, Facebook, Twitter, Google, and Spotify once were developed by one or a few individuals with a great vision but by just using programming as a skill. However, as the great vision has been implemented, be sure that these companies are today not relying on heroic programmers. Today, these companies have hired the top people with long experience in software engineering including great programming skills.

So, what is software engineering? Before we answer this question, we must first make it very clear that there is a remarkable difference between hacking versus professional programming. Professional programming involves clear logical thinking, beginning with the objective of the program, and refining the objective into logically constructed expressions. Indeed, the expressions are a reflection of the programmers’ thinking and analysis. Hacking on the other hand is an ad hoc trial and error to induce the desired effect. When the effect is achieved, the hacker marvels without really understanding why it worked. Professional programmers understand why and how it worked.

As such, professional programming is highly disciplined. Software engineering takes this discipline to software teams working on complex software. A typical software development endeavor involves more than one person working on a complex problem over a period of time to meet some objectives. Throughout Smith’s introductory software engineering course, he worked on several assignments, which frequently required him to work with his fellow students, and which included tasks, such as:

1. brainstorming what an event calendar app would look like;
2. writing code for a simple event calendar in a small group;
3. writing code for the event calendar app, and hosting the app on the cloud;

4. reviewing a given piece of code to find issues in it, for example bugs, and poor understandability; and
5. reviewing a fellow student's code.

Through these assignments, Smith came to several conclusions. First, there is no one true way to write code for a given problem. Writing good quality code that fellow students can understand is not easy. It often takes more than one pair of eyes to get it working and comprehensible. He learned the following.

- Testing, i.e., checking that the program behaves as intended, is not easy. There are so many paths that executing the code can follow and all have to be tested.
- Agreeing on what the application would do was challenging. Even for that simple event calendar app, Smith and his team debated quite a while before they came to a consensus on what functionality ought to be available, and how the user interface should be laid out.
- A simple application may require multiple programming languages. For example, the event calendar app would need HTML5 and JavaScript for the front end, and the Java and SQL database for the backend. Consequently, Smith found that he had to spend a significant amount of time learning and getting familiar with new programming languages and new programming frameworks. Although he endeavored to learn about all these, it was certainly not easy with the limited time that was available.
- Time management is not easy because it is hard to estimate how much time each activity will require—or when to stop fine-tuning a certain piece of code to meet time constraints of the project.

As Smith was preparing for his industry internship interview, he tried to summarize on a piece of paper, from those things he then understood, what software engineering is about, and what he had learned thus far. Smith drew what he understood many times, and he observed that he couldn't get it quite right. In the end, he settled for what is shown in Figure 1.2.

To Smith, software engineering was about taking some idea and forming a team according to the requirements. The team then transforms the requirements into a software product. To do this, the team engages in some kind of brainstorming, consensus, writing and testing code, getting to a stable structure, maintaining user satisfaction throughout, and finally delivering the software product. This requires the team to have competencies in coding, analysis, and teamwork. In addition,

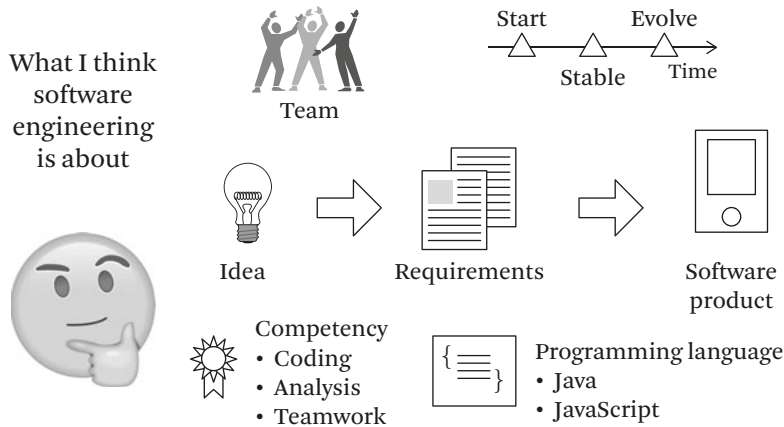


Figure 1.2 What software engineering is from the eyes of a student.

the team needs familiarity with some programming language, such as Java and JavaScript, which Smith knew. What Smith didn't yet know was that the tasks he had been given were still relatively simple tasks compared to what is typical in the software industry. Nevertheless, with this preparation, Smith marched toward his internship interview.

1.3 From Internship to Industry

With some luck, Smith managed to join the company TravelEssence as an intern trainee. Dave the interviewer saw some potential in Smith. Dave was particularly intrigued that Smith managed to draw the picture in Figure 1.2. Most students couldn't, and would get stuck if they even attempted to.

TravelEssence is a fictitious company that we will be using as an example throughout this book. TravelEssence provides online hotel booking services for travelers (see Figure 1.3). In addition, TravelEssence provides Software as a Service (SaaS) for the operation of hotels. SaaS means that the owner of the software, in this case TravelEssence, provides software as a service over the internet and the clients pay a monthly fee. Hotels can sign up and use the TravelEssence service to check-in and check-out their customers, print bills, compute taxes, etc.

Smith's stint in TravelEssence provided a whole new experience. To him, his new colleagues seemed to come from two groups: those who stated what they wanted the software to do, and those who wrote and tested the software. Figure 1.4 highlights the dramatic changes Smith experienced. While everyone seemed to speak English, they used words that he did not understand, especially the first group. As a diligent person, Smith compiled a list of some of this jargon.

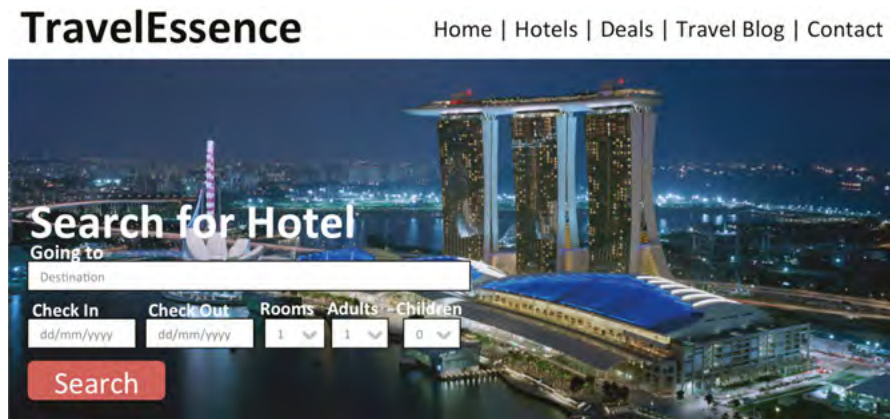


Figure 1.3 TravelEssence home page.

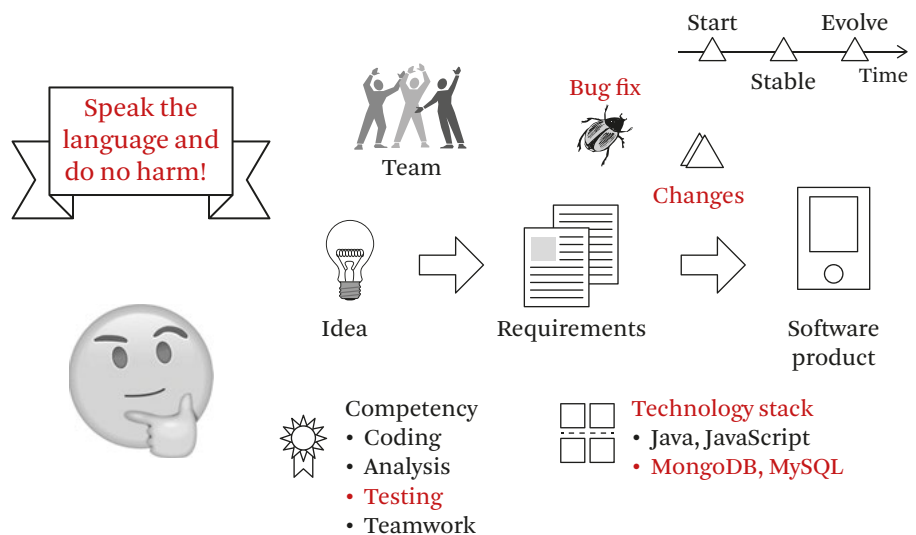


Figure 1.4 What software engineering is from the eyes of a student after internship.

Book. To sell or reserve rooms ahead of time.

No-Show. A guest who made a room reservation but did not check in.

Skipper. A guest who left with no intention of paying for the room.

PMS. Property Inventory Management System, which maintained records of items owned by the hotel such as items in each room including televisions, beds, hairdryers, etc.

POS. Point of Sale Systems (used in restaurants/outlets) that automated the sale of items and managed purchases with credit or debit cards.

It took Smith a little while to get on “speaking terms” with his new colleagues and mentors.

In his student days, Smith always wrote code from scratch, starting with an empty sheet of paper. However, at TravelEssence it was mostly about implementing enhancements to some existing code. The amount of code that Smith saw was way above the toy problems he came across as a student. His development colleagues did not trust him to make any major changes to the system. Developers in TravelEssence emphasized code reviews heavily and stressed the importance of “Do no harm” repeatedly. They would repeatedly test his understanding of terminology and their way of working. Smith felt embarrassed when he could not reply confidently. He started to understand the importance of reviewing and testing his work. After his internship, Smith attempted to summarize what he understood software engineering to be (see Figure 1.4). This was quite similar to what he thought before his internship (see Figure 1.2), but with new knowledge (indicated in red) and an emphasis on testing and doing no harm as he coded changes to the software product. Smith came to recognize the importance of knowledge in different areas, not just about the code, but also about the problem domain (in this case, about hotel management), and the technologies that were being used.

Competency not only involved analysis, coding, and teamwork, but also extensive testing to ensure that Smith did no harm. Understanding programming languages was no longer sufficient; a good working knowledge of the technology stack was critical. A technology stack is the set of software technologies, often called the building blocks, that are used to create a software product. Smith was familiar with multiple technologies that were being used including Java, JavaScript, MongoDB, and MySQL. Never mind if you do not know these specific terms.

Note: There are myriads of technology stacks available, and it is not possible for anyone to learn them all. Nevertheless, our recommendation to students is to gain familiarity with a relevant technology stack of your choice.

Smith graduated and was employed at TravelEssence. A few years later, at a get-together, Smith and his old classmates shared their newfound experiences in the real commercial world. At this occasion Smith said: “At TravelEssence even though everyone seemed to be using different terminology, and everyone did things differently, there seemed to be something common to what they were all doing.” One of his old classmates asked Smith if he could explain more, but Smith just shook his head and said, “I don’t know exactly what it is.”

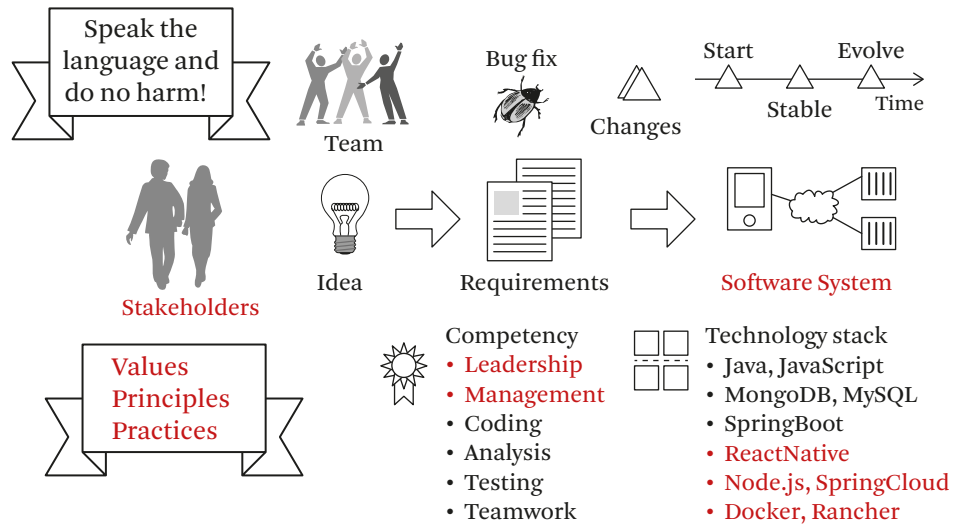


Figure 1.5 What software engineering is from the eyes of a young professional.

Some years later, Smith became a technical lead for a small group at Travel-Essence. As a technical lead he found himself continuously thinking about that discussion with his old classmates as he tried to figure out just what it was that was common about the way everyone worked at TravelEssence.

One evening the old classmates got together again. This time the discussions were a blend between technologies and people management. The old classmates were also talking more about their experiences dealing with people including their colleagues, managers, and their customers; consequently, they were talking more about the way work got done in their organizations. Managing stakeholders and their expectations became more important as they started to take on more senior positions.

After the meeting with classmates, Smith started to draw what he then thought software engineering was about (see Figure 1.5). The changes compared to Smith's internship experience are highlighted in red.

Stakeholder collaboration played an important part of Smith's work. Collaborating well involved having an agreed-on set of values, principles, and practices. These values included agreeing upon a common goal, and respecting and trusting team members, as well as being responsible and dependable. All of these values are qualities of a good and competent team player. Principles include, for instance, having frequent and regular feedback, and fixing bugs as soon as they are detected. All of

these principles identify good behaviors in a team. Practices are specific things the team will do to deliver what is expected of the team consistent with the above values and principles, as well as good quality software.

1.4 Journey into the Software Engineering Profession

Smith through his experience at TravelEssence thus far had started to appreciate the complexities involved in producing and sustaining high-quality software that meets the needs of stakeholders. He now appreciated that while programming is an important aspect, there is much more involved. It is the engineering discipline that is concerned with all aspects of the development and sustainment of software products.

Smith then reflected upon the knowledge he had attained thus far in his career. As a student with no other experience than having done some programming, it is quite difficult to understand what more is involved in software engineering. Typically, when creating a program in a course setting, the exercise starts from an idea that may have been explained in a few words: say, less than one hundred words. Based on the idea, Smith and his classmates developed a piece of software, meaning they wrote code and made sure that it worked. After the assignment they didn't need to take care of it. These assignments were small and to perform them they really did not need much engineering discipline. This situation is quite unlike what you have to do in the industry, where code written will stay around for years, passing through many hands to improve it. Here a sound approach to software engineering is a must. Otherwise, it would be impossible to collaborate and update the software with new features and bug fixes. Nevertheless, the experience in school is an important and essential beginning, even though Smith wished that it were more like the industry.

The authors of this book have all experienced, through their personal journeys, the importance of utilizing an engineering approach in providing high-quality software. Thus, we can characterize, for you, what is important in respect to software engineering.

Considering the software industry, let's put the success of Microsoft, Apple, Google, Facebook, Twitter, etc. on the side because they are so unique—relying on innovative ideas that found a vast commercial market—and programming, per se, was not the root cause of their success. In a more normal situation you will find yourself employed by a company that as part of their mission needs to develop software to support their business or to sell a product needed by potential customers. The company may be rather small or very large, and you will be part of a team. The reasons you won't be alone are many. What needs to be done is more

than what one person can do alone. If the software product is large your team will most likely not be the only one; there will be many teams that have to work in some synchronized way to achieve the objectives of your company.

As a young student having spent most of your life at school and not yet working in the industry, you may be more interested in the technologies related to software—the computer, programming languages, operating systems, etc.—and less interested in the practicalities of developing commercial software for a particular business.

However, this is going to change with this book.

First, let us consider the importance of a team. The team has a role in the company to develop some software. To do that, they need to know what the users of the software need, or in other words they need to agree on the *requirements*. In some cases, they will receive the requirements indicating that they want software that does what another piece of software does. In these cases, the team must study the other product and do something like that product or better. In other situations, someone will just tell them what to do and be with the team while they do it. In more regulated organizations, someone (or a group of people) has written a document specifying what is believed to be some or all of the requirements. Typically, people don't specify all the requirements before starting the development, but some requirements will be input to the team, so they can start doing something to show to the future users of the product. Interacting with users on intermediary results will reveal weaknesses and tell the team what they need to do next. These discussions may imply that the team has to backtrack and redo parts of what they have done and demonstrate the new results to the users. These discussions will also tell the team what more needs to be done.

Anyway, the team will in one way or the other have to understand what requirements they should use as input to the work of their team. Understanding the requirements is normally not trivial. It may take as much time or even more as it takes to program a solution. As we just stated, you will typically have to modify them and sometimes throw away some of the requirements as well as work results before the users of the software are reasonably satisfied with what they have received.

As a newcomer to software engineering but with some background in programming, you may think that working with requirements is less rewarding and less interesting than programming. Well, it is not. There is an entire discipline (*requirements engineering*) that specifies how you dig out the requirements, how you think about them to create great user experiences supported by the software, and how you

modify them to improve and sustain the software. There are requirements management tools to help you that are as interesting to work with as programming tools. There are many books and other publications on how to work with requirements, so there is a lot to learn as you advance in your career. Therefore, working with requirements is one of the things to do that is more than programming but part of software engineering.

Another thing to do that is more than programming is the *design* of the software. Design means structuring the code in such a way that it is easy to understand, easy to change to meet new requirements, easy to test, etc. You can describe your design by using elements of a programming language such as component, class, module, interface, message, etc. You can also use a visual language with symbols for such elements that have a direct correspondence in the programming language you are utilizing. In the latter case, you use a tool to draw diagrams with symbols representing, for instance, components with interfaces. In short, you express the design in a diagram form. The visual language can be quite sophisticated and allow you to not just express your design; for example, you can do quality controls using a visual language tool as well as testing the design to some extent. Doing design is as interesting and rewarding as programming and it is an important part of software engineering.

Apart from working with requirements and creating a design, there are many other things we need to do when we engineer software. We do extensive *testing* of the software; we *deploy* it on a computer so it can be executed and used. If the software we have developed is successful, we will *change* it for many years to come. In fact, most people developing software are engaged in changing existing software that has been developed, often many years ago. This means we need to deal with versions of existing software and if the software has been used at many places (even around the world) we often need to have different versions of the same original software at different locations in the world. Each version will change independent of the other existing versions. And, the complexity of the software product just continues to increase. The only way to deal with this complexity is to use tools specifically designed for its purpose: testing, deployment, version and configuration control, etc.

So, you see that software engineering is certainly much more than programming. While definitions of software engineering are always a subject of debate among professionals, the following neatly summarizes our view. *Software engineering is the application of a systematic, disciplined, and quantifiable approach to the development, testing, deployment, operation, and maintenance of software systems.*

To us, “*a systematic, disciplined, and quantifiable approach*” means it is repeatable and consistent from one project to another, with continuous improvement on the way. It means it is accompanied by some form of description of the way of working and it allows us to automate more. Software engineering includes understanding what users and other stakeholders need and transforming those needs into clear requirements that can be understood by programmers. It also includes understanding the specific technologies needed to build and to test the software. It requires teams that have the social skills to work together, so each piece of the software works with other pieces to achieve the overall goal. So, software engineering encompasses the collaboration of individuals to evolve software to achieve some goal.

Programming is very rewarding since you immediately see the impact of your work. However, as you will learn during your journey, the other activities in software engineering—requirements, design, testing, etc.—are also fascinating for similar reasons. It has been more difficult, though, to teach these other activities in a systematic and generic manner. This is due to the fact that there are so many variations of these activities and there has not been a common ground for teaching them until now as presented in this book. You will find that most students who study in the software domain have an initial desire to work with programming. However, as these people become more and more experienced they gradually move into the other areas of software engineering. This is not because programming is not important. In fact, without programming there is no product to use and sell. No, it is because they find the other areas to be more challenging; also, success in these other areas requires more experience. By essentializing software engineering as presented in this book, the full scope of the discipline will be easier to grasp and to teach.

What Should You Now Be Able to Accomplish?

After studying this chapter, you should be able to:

- explain the terms *programming*, *software development*, and *software engineering*, and how they relate to each other;
- explain the difference between professional programming and hacking;
- understand how teamwork affects the dynamics of software engineering (e.g., importance of code understandability);

- explain the importance of testing as a tool to promote safe modification of existing code;
- understand how people management blends into software engineering and why it is important to consider it;
- explain the role of requirements engineering.

In order to support your learning activities, we invite you to visit www.software-engineering-essentialized.com. There one can find additional material, exercises related to this chapter, and some questions one might encounter in an exam.

In addition to this you will find a short account of the history of software engineering in Appendix [A](#).



Software Engineering Methods and Practices

In this chapter we present how the *way of working* to develop software is organized, and to some extent what additional means are needed (e.g., notations for specifications). In particular, we

- describe the challenges in software engineering covering a wide range of aspects like how to proceed step by step, involve people, methods and practices;
- outline various key concepts of some commonly used software engineering methods created during the last four decades (i.e., waterfall methods, iterative lifecycle methods, structured methods, component methods, agile methods); and
- describe the motivation behind the initiative to create the Essence standard as a basic and extendable foundation for software engineering.

This will also take the reader briefly through the development of software engineering.

2.1 Software Engineering Challenges

From Smith's specific single-person view of software engineering, we move to take a larger worldview in this chapter and the next. We will return to Smith's journey in Chapter 4. From 2012–2014, the IEEE Spectrum published a series of blogs on IT hiccups.¹ There are all kinds of bloopers and blunders occurring in all kinds of industries, a few of which we outline here.

- According to the *New Zealand Herald*, the country's police force in February 2014 apologized for mailing over 20,000 traffic citations to the wrong drivers.

1. <http://spectrum.ieee.org/riskfactor/computing/it/it-hiccups-of-the-week>

Apparently, the NZ Transport Agency, which is responsible for automatically updating drivers' details and sending them to the police force, failed to do so from October 22 to December 16, 2013. As a result, "people who had sold their vehicles during the two-month period . . . were then incorrectly ticketed for offenses incurred by the new owners or others driving the vehicles." In New Zealand, unlike the U.S., license plates generally stay on a vehicle for its life.²

- The *Wisconsin State Journal* reported in February 2013 that "glitches" with the University of Wisconsin's controversial payroll and benefits system had resulted in US \$1.1 million in improper payments which the university would likely end up having to absorb. This was after a news report in the previous month indicated that problems with the University of Wisconsin's payroll system had resulted in \$33 million in improper payments being made over the past two years.³

These types of highlighted problems seem to be those which we can find amusing; however they are really no laughing matter if you happen to be one of the victims. What is more surprising is that the problem with these situations is that they can be prevented, but they almost inevitably do occur.

2.2 The Rise of Software Engineering Methods and Practices

Just as we have compressed Smith's journey from a young student to a seasoned software engineer in a few paragraphs, we will attempt to compress some 50 years of software engineering into a few paragraphs. We will do that with a particular perspective in mind: what resulted in the development of a common ground in software engineering—the Essence standard. A more general description of the history is available in Appendix A.

However, the complexity of software programs did not seem to be the only root cause of the so-called "software crisis." Software endeavors and product development are not just about programming; they are also about many other things such as understanding what to program, how to plan the work, how to lead the people and getting them to communicate and collaborate effectively.

2. <http://spectrum.ieee.org/riskfactor/computing/it/new-zealand-police-admits-sending-20-000-traffic-tickets-to-the-wrong-motorists>

3. <http://spectrum.ieee.org/riskfactor/computing/it/it-hiccups-of-the-week-university-of-wisconsin-loses-another-11-million-in-payroll-glitches>

For the purpose of this introductory discussion, we define a *method* as providing guidance *for all the things you need to do* when developing and sustaining software. For commercial products “*all the things*” are a lot. You need to work with clients and users to come up with “the what” the system is going to do for its users—the requirements. Further, you need to design, code, and test. However, you also need to set up a team and get them up to speed, they need to be assigned work, and they need a way of working.

These things are in themselves “mini-methods” or what many people today would call *practices*. There are *solution*-related “practices,” such as work with requirements, work with code, and conduct testing. There are *endeavor*-related practices, such as setting up a collaborative team and an efficient endeavor as well as improving capability of the people and collecting metrics. There are of course *customer*-related practices, such as making sure that what is built is what the customers really want.

The interesting discovery we made more than a decade ago was that even if the number of methods in the world was huge, it seemed that all these methods were just compositions of a much smaller collection of practices, maybe a few hundred of such practices in total. Practices are what we call *reusable* because they can be used over and over again to build different methods.

To understand how we as a software engineering community have improved our knowledge in software engineering, we provide a description of historical developments. Our purpose with this brief history is to make it easier for you to understand why Essence was developed.

2.2.1 There Are Lifecycles

From the ad hoc approach used in the early years of computing came the *waterfall* method around the 1960s; actually, it was not just one single method—it was a whole class of methods. The waterfall methods describe a software engineering project as going through a number of phases such as Requirements, Design, Implementation (Coding), and Verification (i.e., testing and bug-fixing) (see Figure 2.1).

While the waterfall methods helped to bring some discipline to software engineering, many people tried to follow the model literally, which caused serious problems especially on large complex efforts. This was because software engineering is not as simple as this linear representation indicates.

A way to describe the waterfall methods is this: What do you have once you think you have completed the requirements? Something written on “paper.” (You may have used a tool and created an electronic version of the “paper,” but the point is that it is just text and pictures.) But since it has not been used, do you know for sure

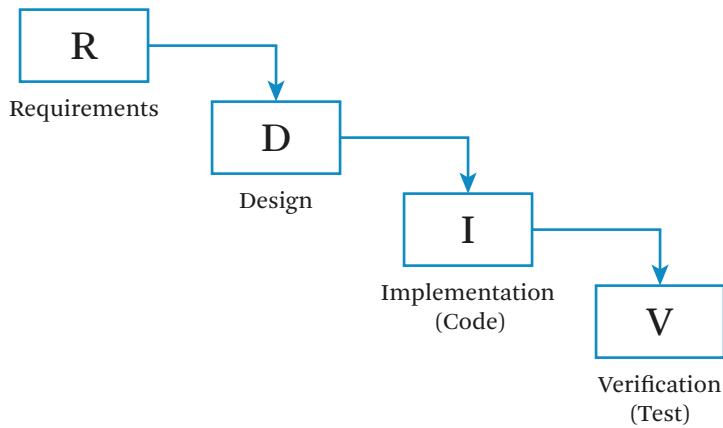


Figure 2.1 Waterfall lifecycle.

at this point if they are the right requirements? No, you don't. As soon as people start to use the product being developed based on your requirements, they almost always want to change it.

Similarly, what do you have after you have completed your design? More “paper” of what you think needs to be programmed? But are you certain that it is what your customer really intended? No, you are not. However, you can easily claim you are on schedule because you just write less and with less quality.

Even after you have programmed according to the design, you still don't know for sure. However, all of the activities you have conducted don't provide proof that what you did is correct.

Now you may feel you have done 80%. The only thing you have left is to test. At this point the endeavor almost always falls apart, because what you have to test is just too big to deal with as one piece of work. It is the code coming from all the requirements. You thought you had 20% left but now you feel you may have 80% left. This is a common well-known problem with waterfall methods.

There are some lessons learned. Believing you can specify all requirements upfront is just a myth in the vast majority of situations today. This lesson learned has led to the popularity of more iterative lifecycle methods. Iterating means you can specify some requirements and you can build something meeting these requirements, but as soon as you start to use what you have built you will know how to make it a bit better. Then you can specify some more requirements and build, and test these until you have something that you feel can be released. But to gain confidence you need to involve your users in each *iteration* to make sure what you have

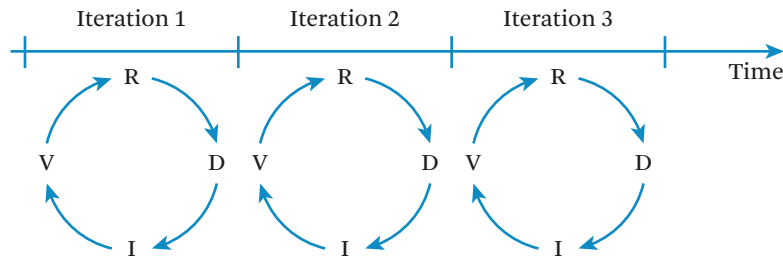


Figure 2.2 Iterative lifecycle.

provides value. These lessons gave rise at the end of the 1980s to a new lifecycle approach called iterative development, a lifecycle adopted by the agile paradigm now in fashion (see Figure 2.2).

New practices came into fashion. The old project management practices fell out of fashion and practices relying on the iterative metaphor became popular. The most prominent practice was Scrum, which started to become popular at the end of the 1990s and still is very popular. We will discuss this more deeply in Part III of the book.

2.2.2 There Are Technical Practices

Since the early days of software development, we have struggled with how to do the right things in our projects. Originally, we struggled with programming because writing code was what we obviously had to do. The other things we needed to do were ad hoc. We had no real guidelines for how to do requirements, testing, configuration management, project management, and many of these other important things.

Later new trends became popular.

2.2.2.1 The Structured Methods Era

In the late 1960s to mid-1980s, the most popular methods separated the software to be developed into the *functions* to be executed and the *data* that the functions would operate upon: the functions living in a program store and the data living in a data store. These methods were not farfetched because computers at that time had a program store, for the functions translated to code, and a data store. We will just mention two of the most popular methods at that time: SADT (Structured Analysis and Design Technique) and SA/SD (Structured Analysis/Structured Design). As a student, you really don't need to learn anything more about these methods. They

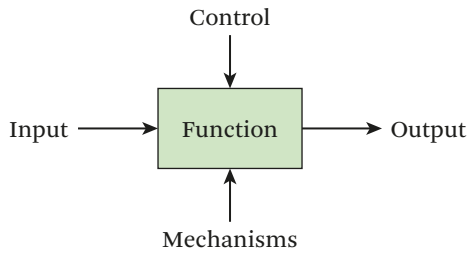


Figure 2.3 SADT basis element.

were used for all kinds of software engineering. They were not the only methods in existence. There were a large number of published methods available and around each method there were people strongly defending it. It was at this time in the history of software engineering that the *methods war* started. And, unfortunately, it has not yet finished!

Every method brought with it a large number of practices such as requirements, design, test, defect management, and the list goes on.

Each had its own blueprint notation or diagrams to describe the software from different viewpoints and with different levels of abstraction (for example, see Figure 2.3 on SADT). Tools were built to help people use the notation and to keep track of what they were doing. Some of these practices and tools were quite sophisticated. The value of these approaches was, of course, that what was designed was close to the realization—to the machine: you wrote the program separate from the way you designed your data. The problems were that programs and data are very interconnected and many programs could access and change the same data. Although many successful systems were developed applying this approach, there were far many more failures. The systems were hard to develop and even harder to change safely, and that became the Achilles’ heel for this generation of methods.

2.2.2.2 The Component Methods Era

The next method paradigm shift⁴ came in early 1980 and had its high season until the beginning of the 2000s.

In simple terms, a software system was no longer seen as having two major parts: functions and data. Instead, a system was a set of interacting elements—

4. Wikipedia: “A paradigm shift, as identified by American physicist and philosopher Thomas Kuhn, is a fundamental change in the basic concepts and experimental practices of a scientific discipline.”

Sidebar 2.1 Paradigm Shift in Detail

In more detail, this paradigm shift was inspired by a new programming metaphor—*object-oriented* programming—and the trigger was the new programming language Smalltalk. However, the key ideas behind Smalltalk were derived from an earlier programming language, Simula 67, that was released in 1967. Smalltalk and Simula 67 were fundamentally different from previous generations of programming languages in that the whole software system was a set of classes embracing its own data, instead of programs (subroutines, procedures, etc.) addressing data types in some data store. Execution of the system was carried out through the creation of objects using the classes as templates, and these objects interacted with one another through exchanging messages. This was in sharp contrast to the previous model in which a process was created when the system was triggered, and this process executed the code line by line, accessing and manipulating the concrete data in the data store. A decade later, around 1990, a complement to the idea of objects received widespread acceptance inspired, in particular, by Microsoft. We got *components*.

components (see also Sidebar 2.1). Each component had an interface connecting it with other components, and over this interface messages were communicated. Systems were developed by breaking them down into components, which collaborated with one another to provide for implementation of the requirements of the system. What was inside a component was less important as long as it provided the interfaces needed to its surrounding components. Inside a component could be program and data, or classes and objects, scripts, or old code (often called legacy code) developed many years ago. Components are still the dominating metaphor behind most modern methods. An interesting development of components that has become very popular is microservices, which we will discuss in Part III.

With components, a completely new family of methods evolved. The old methods with their practices were considered to be out of fashion and were discarded. What started to evolve were in many cases similar practices with some significant differences but with new terminology. In the early 1990s, about 30 different component methods were published. They had a lot in common, but it was almost impossible to find the commonalities since each method author created his/her own terminology.

In the second half of the 1990s, OMG (a standards body called Object Management Group) felt that it was time to at least standardize how to represent software drawings, namely notations used to develop software. This led to a task force being created to drive the development of a new standard. The work resulted in the

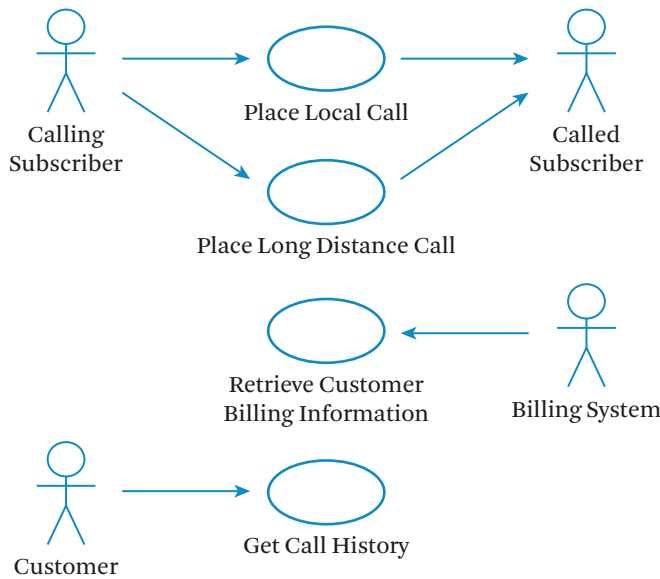


Figure 2.4 A diagram (in fact a Use-Case diagram) from the Unified Modeling Language standard.

Unified Modeling Language (UML; see Figure 2.4), which will be used later in the book. This development basically killed all methods other than the Unified Process (marketed under the name Rational Unified Process (RUP)). The Unified Process dominated the software engineering world around the year 2000. Again, a sad step, because many of the other methods had very interesting and valuable practices that could have been made available in addition to some of the Unified Process practices. However, the Unified Process became in fashion and everything else was considered out of fashion and more or less thrown out.

Over the years, many more technical practices other than the ones supported by the 30 component methods arrived. More advanced architectural practices or sets of practices, e.g., for enterprise architecture (EA), service-oriented architecture (SOA), product-line architecture (PLA), and recently architecture practices for big data, the cloud, mobile internet, and the internet of things (IoT) evolved. At the moment, it is useful to see these practices as pointers to areas of software engineering interest at a high level of abstraction: suffice it to say that EA was about large information systems for, e.g., the finance industry; SOA was organizing the software as a set of possibly optional service packages; and PLA was the counterpart of EA but for product companies, e.g., in the telecom or defense industry. More important is to know that again new methodologies grew up as mushrooms around each one

of these technology trends. With each new such trend method authors started over again and reinvented the wheel. Instead of “standing on the shoulders of giants,”⁵ they preferred to stand on another author’s toes. They redefined already adopted terminology and the methods war just continued.

2.2.2.3 The Agile Methods Era

The agile movement—often referred to just as *agile*—is now the most popular trend embraced by the whole world. Throughout the history of software engineering, experts have always been trying to improve the way software is being developed. The goal has been to compress timescales to meet the ever-changing business demands and realities. If agile were to have a starting date, one can pinpoint it to the time when 17 renowned industry experts came together and penned the words of the agile manifesto. We will present the manifesto in Part IV and how Essence contributes to agile. But for now, it suffices to say that agile involves a set of technical and people-related practices. Most important is that agile emphasizes an innovative mindset such that the agile movement continuously evolves its practices.

Agile has evolved the technical practices utilized with components. However, its success did not come from introducing many new technical practices, even if some new practices, such as continuous integration, backlog-driven development, and refactoring, became popular with agile. Continuous integration suggests that developers several times daily integrate their new code with the existing code base and verify it. Backlog-driven development means that the team keeps a backlog of requirement items to work with in coming iterations. We will discuss this practice in more detail when we discuss Scrum in Part III. Refactoring is to continuously improve existing code iteration by iteration.

Agile rather simplified what was already in use to assist working in an iterative style and providing releasable software over many smaller iterations, or sprints as Scrum calls them.

2.2.3 There Are People Practices

As strange as it may sound, the methods we employed in the early days did not pay much attention to the human factors. Everyone understood of course that software was developed by people, but very few books or papers were written about how to get people motivated and empowered in developing great software. The most

5. From Wikipedia: “The metaphor of dwarfs standing on the shoulders of giants . . . expresses the meaning of ‘discovering truth by building on previous discoveries’.”

successful method books were quite silent on the topic. It was basically assumed that in one way or the other this was the task of management.

However, this assumption changed dramatically with agile methods. Before, there was a high reliance on tools so that code could be automatically generated from design documents such as UML diagrams. Accordingly, the role of programmers was downgraded, and other roles were more prestigious, such as project managers, analysts, and architects. With agile methods programming became reevaluated as a creative job. The programmers, the people who eventually created working software, were “promoted” and coding became again a prestigious task.

With agile many new practices evolved, for instance self-organizing teams, pair programming, and daily standups.

A self-organizing team includes members who are more generalists than specialists—most know how to code even if some are experts. It is like a soccer team—everyone knows how to kick the ball even if some are better at scoring goals and someone else is better at keeping the ball out of the goal.

Pair programming means that two programmers are working side-by-side developing the same piece of code. It is expected that the code quality is improved and that the total cost will be reduced. Usually one of the two, is more senior than the other, so this is also a way to improve team competency.

Daily standup is a practice intended to reduce impediments that team members have, as well as to retain motivation. Every morning the team meets for 15 min to go through each member’s situation: what he/she has done and what he/she will be doing. Any impediments are brought up but not addressed during the meeting. The issues will be discussed in separate meetings. This practice is part of the Scrum practice discussed in Part III.

Given the impact agile has had on the empowerment of programmers, it is easy to understand that agile has become very popular. Moreover, given the positive impact agile has had on our development of software, there is no doubt it has deserved to become the latest paradigm.

2.2.4 Consequences

There is a methods war going on out there. It started 50 years ago, and it still goes on. Jokingly, we can call it the Fifty Years’ War, and there is still no truce. In fact, there are no signs that this will stop by itself.

- With every major paradigm shift such as the shift from structured methods to component methods and from the latter to the agile methods, basically the industry throws out all they know about software engineering and starts

all over with new terminology with little relation to the old. Old practices are viewed as irrelevant and new practices are hyped. To make this transition from the old to the new is extremely costly to the software industry in the form of training, coaching, and change of tooling.

- With every major technical innovation, for instance cloud computing, requiring a new set of practices, the method authors also “reinvent the wheel.” Although the costs are not as huge as in the previous point, since some of the changes are not fundamental across everything we do (it is no paradigm shift) and thus the impact is limited to, for instance, cloud development, there is still foolish waste.
- Within every software engineering trend there are many competing methods. For instance, back as early as 1990 there were about 30 competing object-oriented methods. When this book was written, there were about 10 competing methods on scaling agile to large organizations; some of the most famous ones are Scaled Agile Framework (SAFe), Disciplined Agile Delivery (DAD), Large Scale Scrum (LeSS), and Scaled Professional Scrum (SPS). They typically include some basic widely used practices such as Scrum, user stories or alternatively use cases, and continuous integration, but the method author has “improved” them—sarcastically stated. There is reuse of ideas, but not reuse of original text, so the original inventor of the practice feels he or she has been robbed of his/her work; there is no collaboration between method authors, but instead they are “at war” as competing brands.

Within these famous methods, there are some often useful practices that are specific for each one. The problem is that all these methods are monolithic, not modular, which means that you cannot easily mix and match practices from different methods. If you select one, you are more or less stuck with it. This is not what teams want, and certainly not their companies. This is, of course, what most method authors whose method is selected like, even if it was never what they intended.

Typically, every recognized method has a founding parent, sometimes more than one parent. If successful, this parent is raised to guru status. The guru more or less dictates what goes into his/her method. Thus, once you have adopted a method, you get the feeling you are in a *method prison* controlled by the guru of that method. Ivar Jacobson, together with Philippe Kruchten, was once such a guru governing the Unified Process prison. Jacobson realized that this was “the craziest thing in the world,” a situation unworthy in any industry and in particular in such a huge industry as the software industry. To eradicate such unnecessary method wars and

method prisons, the SEMAT (Software Engineering Method and Theory) initiative was founded.

2.3 The SEMAT Initiative

As of the writing of this book there are about 20 million software developers⁶ in the world and the number is growing year by year. It can be guesstimated that there are over 100,000 different methods to develop software, since basically every team has developed their own way of working even if they didn't describe it explicitly.

Over time, the number of methods is growing much faster than the number of reusable practices. There is no problem with this. In fact, this is what we want to happen, because we want every team or organization to be able to set up its own method. The problem is that until now we have not had any means to really do that. Until now, creating your own method has invited the method author(s) to reinvent everything they liked to change. This has occurred because we haven't had a solid common ground that we all agreed upon to express our ideas. We didn't have a common vocabulary to communicate with one another, and we didn't have a solid set of reusable practices from which we could start creating our own method.

In 2009, several leaders of the software engineering community came together, initiated by Ivar Jacobson, to discuss the future of software engineering. Through that, the SEMAT (Software Engineering And Theory) initiative commenced with two other leaders founding it: Bertrand Mayer and Richard Soley.

The SEMAT call for action in 2009 read as follows.

Software engineering is gravely hampered today by immature practices. Specific problems include:

- The prevalence of fads more typical of fashion industry than of an engineering discipline.
- The lack of a sound, widely accepted theoretical basis.
- The huge number of methods and method variants, with differences little understood and artificially magnified.
- The lack of credible experimental evaluation and validation.
- The split between industry practice and academic research.

6. <https://www.infoq.com/news/2014/01/IDC-software-developers>

We support a process to re-found software engineering based on a solid theory, proven principles, and best practices that:

- Include a kernel of widely agreed elements, extensible for specific uses
- Address both technology and people issues
- Are supported by industry, academia, researchers and users
- Support extension in the face of changing requirements and technology.

This call for action was signed by around 40 thought leaders in the world coming from most areas of software engineering and computer science; 20 companies and about 20 universities have signed it, and more than 2,000 individuals have supported it. You should see the “specific problems” identified earlier as evidence that the software world has severe problems. When it comes to the solution “to re-found software engineering” the keywords here are “a kernel of widely agreed elements,” which is what this book has as a focus.

It was no easy task to get professionals around the world to agree on what software engineering is about, let alone how to do it. It led, of course, to significant controversy. However, the supporters of SEMAT persevered. Never mind that the world is getting more complex, and there is no single answer, but there ought to be some common ground—a kernel.

2.4 Essence: The OMG Standard

After several years of hard work, the underlying language and kernel of software engineering was accepted in June 2014 as a standard by the OMG and it was given the name Essence. As is evident from the call for action, the SEMAT leaders realized already at the very start that a common ground of software engineering (a kernel) needed to be widely accepted. In 2011, after having worked two years together and having reached part of a proposal for a common ground, we evaluated where we were and understood that the best way to get this common ground widely accepted was to get it established as a formal standard from an accredited standards body. The choice fell on OMG. However, it took three more years to get it through the process of standardization. Based upon experience from the users of Essence, it continues to be improved by OMG through a task force assigned to this work.

In the remainder of this part of the book, we will introduce Essence, the key concepts and principles behind Essence, and the value and use cases of Essence.

This material is definitely useful for all students and professionals alike. Readers interested in learning more, please see [Jacobson et al. \[2012, 2013a, 2013b\]](#), and [Ng \[2014\]](#).

What Should You Now Be Able to Accomplish?

After studying this chapter, you should be able to:

- explain the meaning of a method (as providing guidance for all the things you need to do when developing and sustaining software);
- explain the meaning of a practice and its types (i.e., solution-related practices, endeavor-related practices, customer-related practices);
- explain the meaning of waterfall methods and their role in the history of software engineering;
- explain the iterative lifecycle methods, structured methods, component methods, and agile methods, as well as their characteristics;
- give examples of some practices (e.g., self-organizing teams, pair programming, and daily standups as examples of agile practices);
- explain the “method prison” issue discussed in the chapter; and
- explain the SEMAT initiative and the motivation behind the Essence standard.

Again we point to additional reading, exercises, and further material at www.software-engineering-essentialized.com.