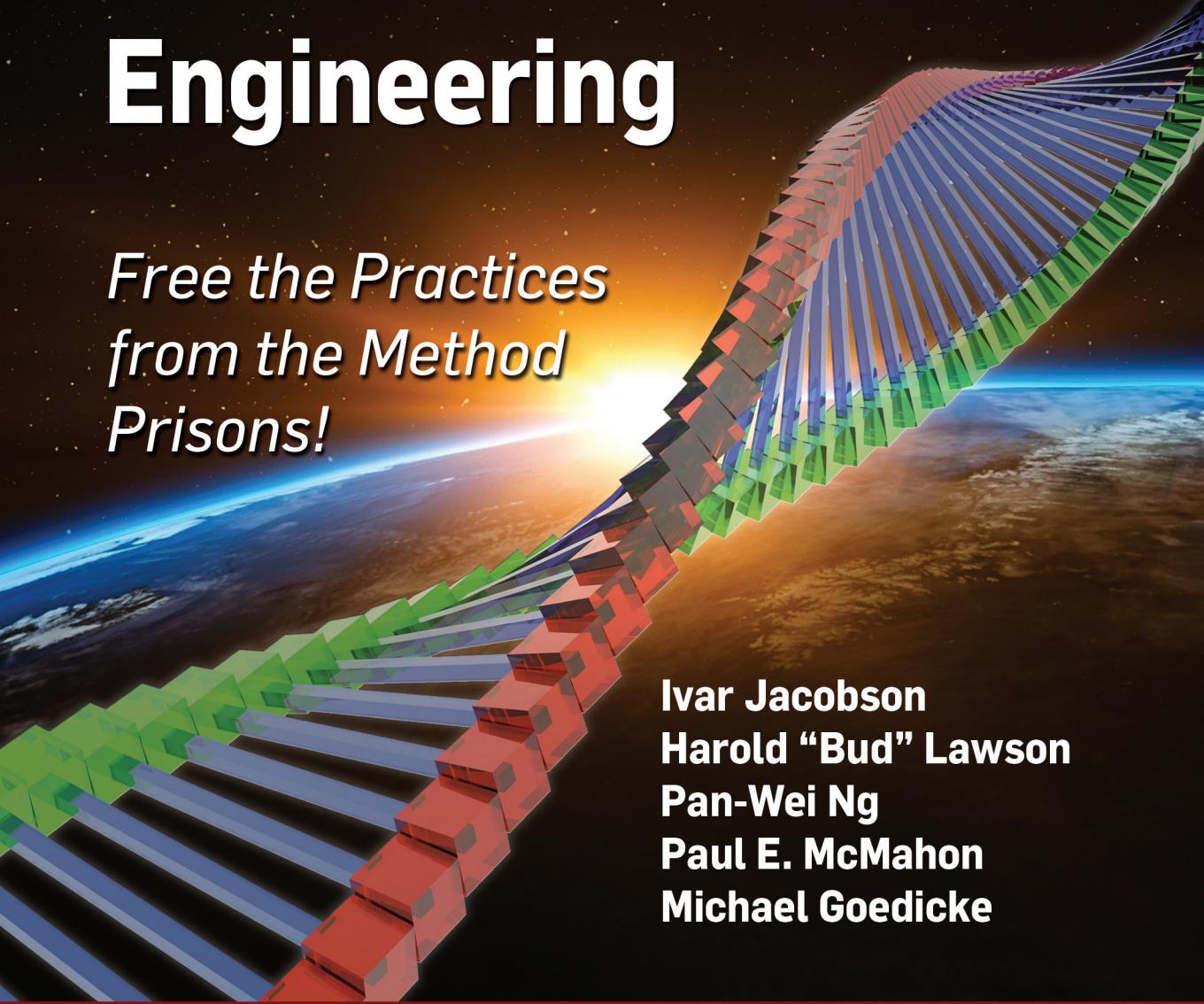


The Essentials of Modern Software Engineering

*Free the Practices
from the Method
Prisons!*



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



The Essentials of Modern Software Engineering

ACM Books

Editor in Chief

M. Tamer Özsu, *University of Waterloo*

ACM Books is a new series of high-quality books for the computer science community, published by ACM in collaboration with Morgan & Claypool Publishers. ACM Books publications are widely distributed in both print and digital formats through booksellers and to libraries (and library consortia) and individual ACM members via the ACM Digital Library platform.

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*

2019

Concurrency: The Works of Leslie Lamport

Dahlia Malkhi, *VMware Research and Calibra*

2019

Data Cleaning

Ihab F. Ilyas, *University of Waterloo*

Xu Chu, *Georgia Institute of Technology*

2019

Conversational UX Design: A Practitioner’s Guide to the Natural Conversation Framework

Robert J. Moore, *IBM Research-Almaden*

Raphael Arar, *IBM Research-Almaden*

2019

Heterogeneous Computing: Hardware and Software Perspectives

Mohamed Zahran, *New York University*

2019

Hardness of Approximation Between P and NP

Aviad Rubinstein, *Stanford University*

2019

**The Handbook of Multimodal-Multisensor Interfaces, Volume 3:
Language Processing, Software, Commercialization, and Emerging Directions**

Editors: Sharon Oviatt, *Monash University*
Björn Schuller, *University of Augsburg and Imperial College London*
Philip R. Cohen, *Monash University*
Daniel Sonntag, *German Research Center for Artificial Intelligence (DFKI)*
Gerasimos Potamianos, *University of Thessaly*
Antonio Krüger, *Saarland University and German Research Center for Artificial Intelligence (DFKI)*
2019

Making Databases Work: The Pragmatic Wisdom of Michael Stonebraker

Editor: Michael L. Brodie
2018

**The Handbook of Multimodal-Multisensor Interfaces, Volume 2:
Signal Processing, Architectures, and Detection of Emotion and Cognition**

Editors: Sharon Oviatt, *Monash University*
Björn Schuller, *University of Augsburg and Imperial College London*
Philip R. Cohen, *Monash University*
Daniel Sonntag, *German Research Center for Artificial Intelligence (DFKI)*
Gerasimos Potamianos, *University of Thessaly*
Antonio Krüger, *Saarland University and German Research Center for Artificial Intelligence (DFKI)*
2018

Declarative Logic Programming: Theory, Systems, and Applications

Editors: Michael Kifer, *Stony Brook University*
Yanhong Annie Liu, *Stony Brook University*
2018

The Sparse Fourier Transform: Theory and Practice

Haitham Hassanieh, *University of Illinois at Urbana-Champaign*
2018

The Continuing Arms Race: Code-Reuse Attacks and Defenses

Editors: Per Larsen, *Immunant, Inc.*
Ahmad-Reza Sadeghi, *Technische Universität Darmstadt*
2018

Frontiers of Multimedia Research

Editor: Shih-Fu Chang, *Columbia University*
2018

Shared-Memory Parallelism Can Be Simple, Fast, and Scalable

Julian Shun, *University of California, Berkeley*
2017

Computational Prediction of Protein Complexes from Protein Interaction Networks

Sriganesh Srihari, *The University of Queensland Institute for Molecular Bioscience*

Chern Han Yong, *Duke-National University of Singapore Medical School*

Limsoon Wong, *National University of Singapore*

2017

The Handbook of Multimodal-Multisensor Interfaces, Volume 1: Foundations, User Modeling, and Common Modality Combinations

Editors: Sharon Oviatt, *Incaa Designs*

Björn Schuller, *University of Passau and Imperial College London*

Philip R. Cohen, *Voicebox Technologies*

Daniel Sonntag, *German Research Center for Artificial Intelligence (DFKI)*

Gerasimos Potamianos, *University of Thessaly*

Antonio Krüger, *Saarland University and German Research Center for Artificial Intelligence (DFKI)*

2017

Communities of Computing: Computer Science and Society in the ACM

Thomas J. Misa, Editor, *University of Minnesota*

2017

Text Data Management and Analysis: A Practical Introduction to Information Retrieval and Text Mining

ChengXiang Zhai, *University of Illinois at Urbana-Champaign*

Sean Massung, *University of Illinois at Urbana-Champaign*

2016

An Architecture for Fast and General Data Processing on Large Clusters

Matei Zaharia, *Stanford University*

2016

Reactive Internet Programming: State Chart XML in Action

Franck Barbier, *University of Pau, France*

2016

Verified Functional Programming in Agda

Aaron Stump, *The University of Iowa*

2016

The VR Book: Human-Centered Design for Virtual Reality

Jason Jerald, *NextGen Interactions*

2016

Ada's Legacy: Cultures of Computing from the Victorian to the Digital Age

Robin Hammerman, *Stevens Institute of Technology*

Andrew L. Russell, *Stevens Institute of Technology*

2016

Edmund Berkeley and the Social Responsibility of Computer Professionals

Bernadette Longo, *New Jersey Institute of Technology*

2015

Candidate Multilinear Maps

Sanjam Garg, *University of California, Berkeley*

2015

Smarter Than Their Machines: Oral Histories of Pioneers in Interactive Computing

John Cullinane, *Northeastern University; Mossavar-Rahmani Center for Business
and Government, John F. Kennedy School of Government, Harvard University*

2015

A Framework for Scientific Discovery through Video Games

Seth Cooper, *University of Washington*

2014

**Trust Extension as a Mechanism for Secure Code Execution on Commodity
Computers**

Bryan Jeffrey Parno, *Microsoft Research*

2014

Embracing Interference in Wireless Systems

Shyamnath Gollakota, *University of Washington*

2014

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 Arnhem Pro 10/14 and Flama using ZzTeX.

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
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 Perkens-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 Firesmith, 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.

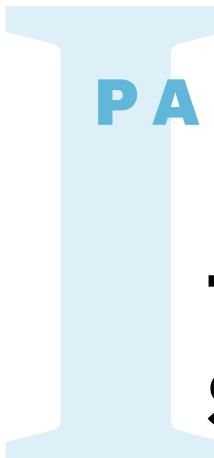
- Academics: Chalmers University of Technology, Florida Atlantic University, Free University of Bozen Bolzano, Fudan University, Harbin Institute of Technology, Joburg Centre for Software Engineering at Wits University, KAIST, KTH Royal Institute of Technology, National University of Colombia at Medellin, 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 Cunningham, 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 Calavarro, A. Chamundeswari, Görkem Giray, Emanuel Grant, Debasish Jana, Eréndira Miriam Jiménez Hernandez, 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.



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 und 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,

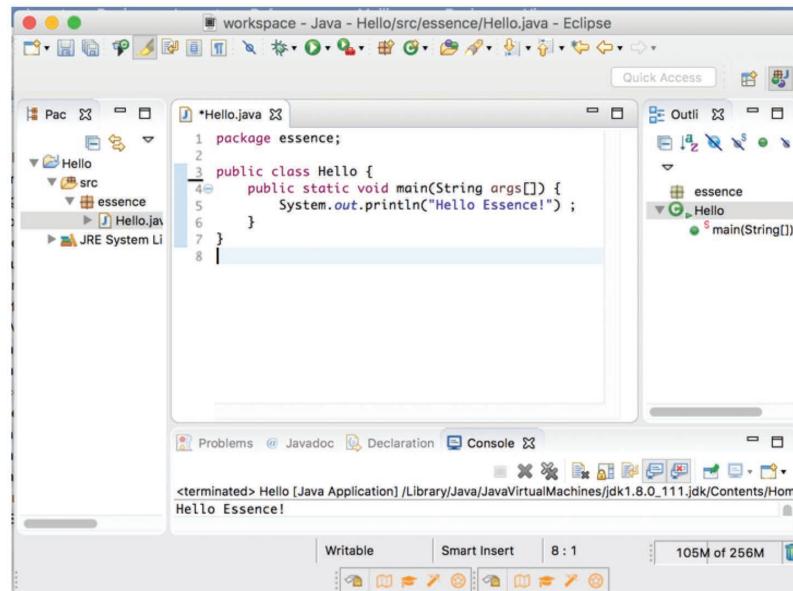


Figure 1.1 Hello Essence.

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,

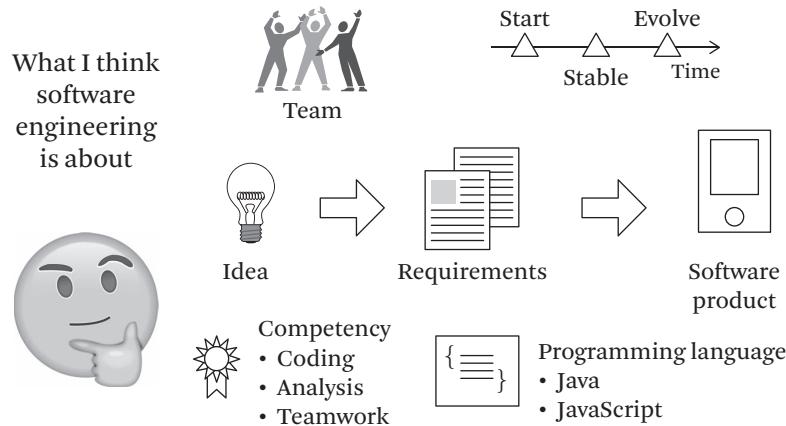


Figure 1.2 What software engineering is from the eyes of a student.

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.

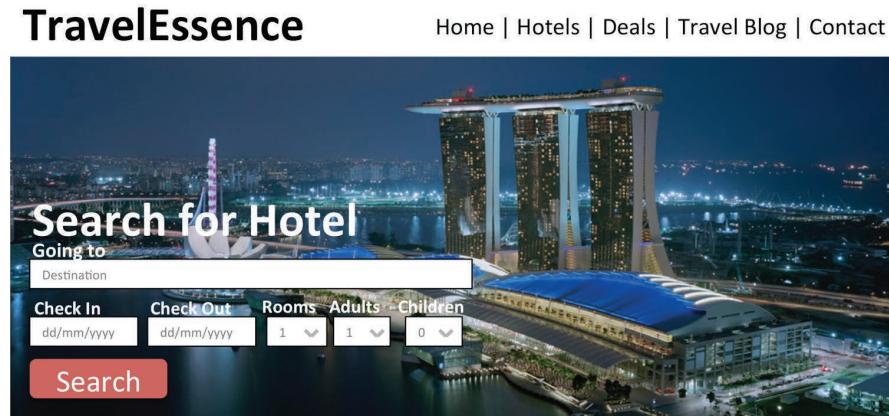


Figure 1.3 TravelEssence home page.

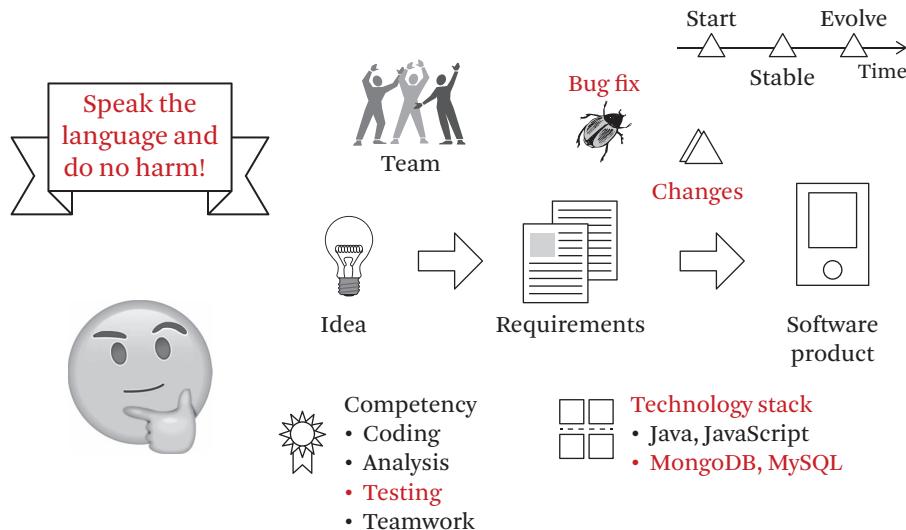


Figure 1.4 What software engineering is from the eyes of a student after internship.

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.”

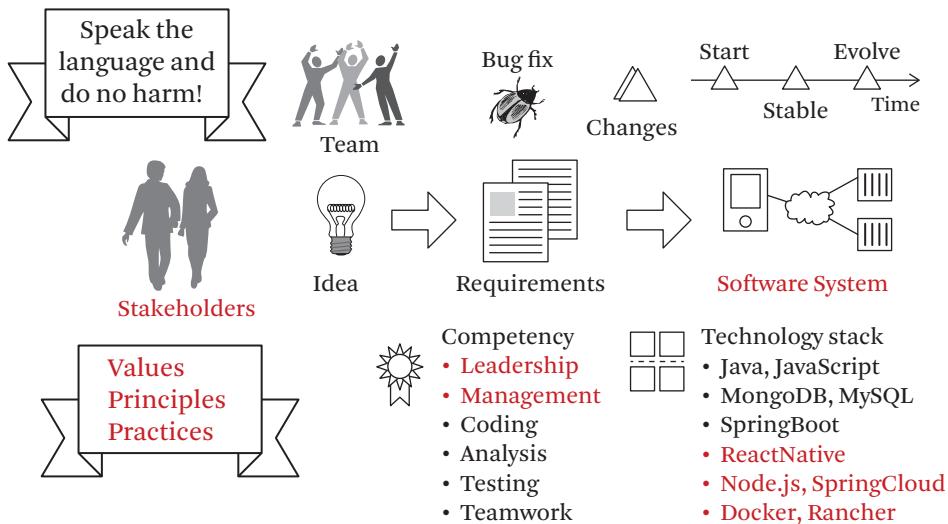


Figure 1.5 What software engineering is from the eyes of a young professional.

Some years later, Smith became a technical lead for a small group at TravelEssence. 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?

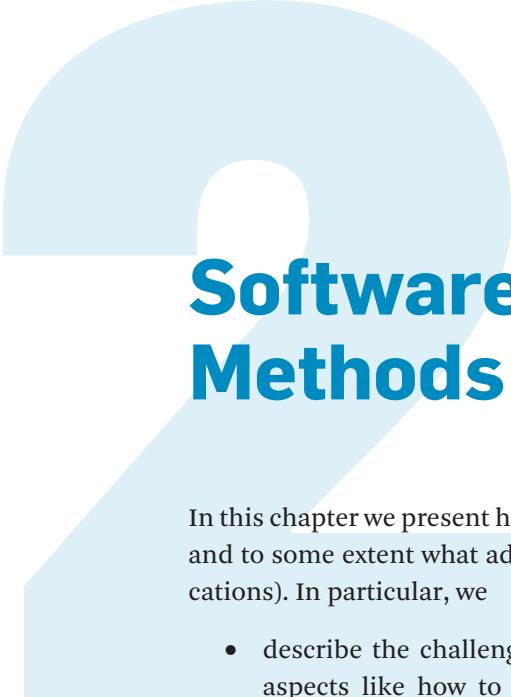
After studying this chapter, you should be able to:

- 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

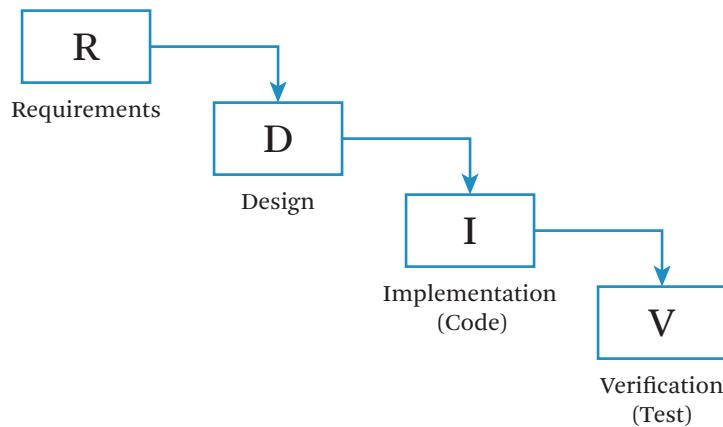


Figure 2.1 Waterfall lifecycle.

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 up-front 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

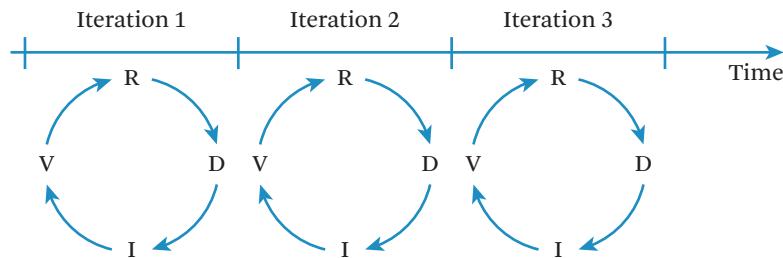


Figure 2.2 Iterative lifecycle.

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

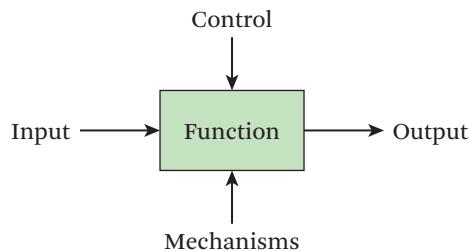


Figure 2.3 SADT basis element.

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

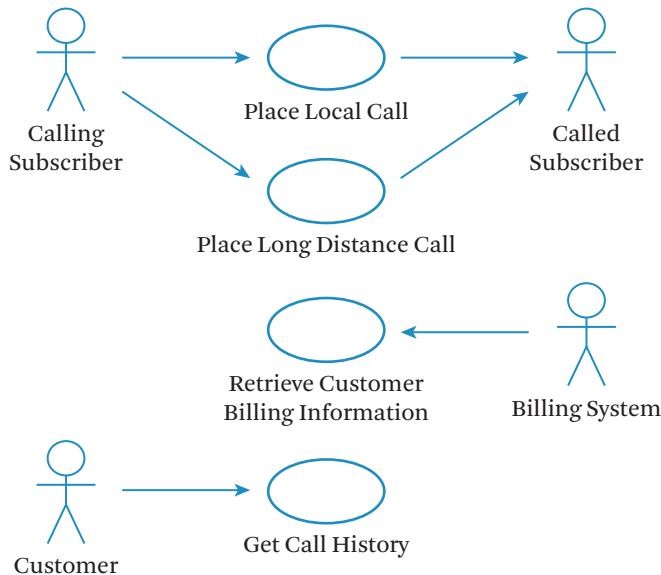


Figure 2.4 A diagram (in fact a Use-Case diagram) from the Unified Modeling Language standard.

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 Meyer 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.

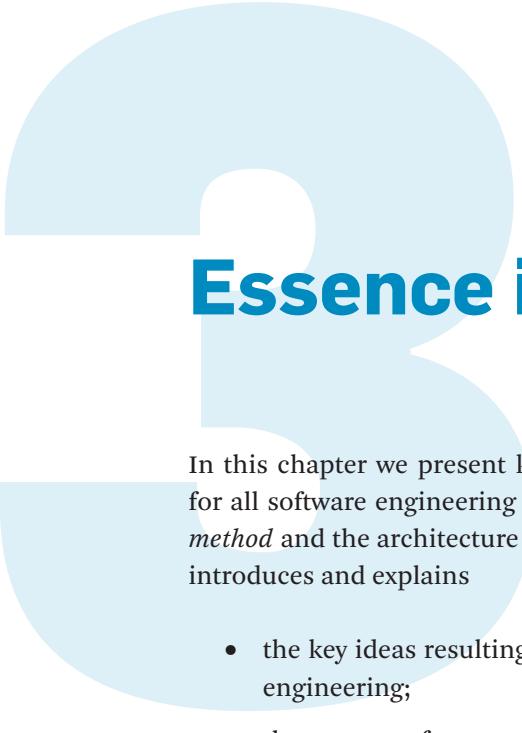
This material is definitely useful for all students and professionals alike. Readers interested in learning more, please see [Jacobson et al. \[2012, 2013a, 2013b\]](#), and [Ng \[2014\]](#).

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.



Essence in a Nutshell

In this chapter we present key ideas resulting in Essence as a common ground for all software engineering methods. Basic concepts of Essence, the notion of a *method* and the architecture of Essence are introduced. In particular, this chapter introduces and explains

- the key ideas resulting in Essence and why they are important for software engineering;
- the concept of composition as a way to combine practices to form a method to guide a development endeavor;
- the common ground Essence as consisting of a language and a kernel of essential elements;
- the concepts of practices and methods built on top of Essence forming a method architecture; and
- the concept of cards as a means to give the abstract elements of Essence an attractive and tangible user experience.

This chapter shows that the main idea behind Essence is to focus on the most important things to think about when developing software (focus on the essentials only).

Essence became a standard in 2014 as a response to “the craziest thing in the world” presented in Chapters 1 and 2.

In this chapter, we will present some key ideas used as input when designing Essence. We will also separately motivate each idea and describe how it has been realized with Essence.

3.1 The Ideas

Essence relies on the following insights.

- Methods are compositions of practices.
 - There are a huge number of methods (guesstimated to be $> 100,000$) in the world, some of which are recognized and have a large user base.
 - There are only a few hundred reusable practices in the world. With n practices the number of theoretically possible combinations of these practices can quickly grow very large.
- There is a common ground, or a kernel, shared among all these methods and practices.
- Focus on the essentials is needed when providing guidelines for a method or practice.
- Providing an engaging user experience is possible when teaching and learning methods and practices.

3.2 Methods Are Compositions of Practices

As explained in Chapter 2, a method is created with the intention to guide the software development team(s) through everything they need to do during the development process: that is, giving them all the practices they need. A practice is like a mini-method in that it guides a team in how to carry out one particular thing in their work. For instance, “requirements management” is a potential practice dealing with what a software system should do. It is obviously not all you need to do when you develop software; you need many other such practices, for instance, “design, implement, and test,” “organize your team,” “perform project management,” etc. For small endeavors, it is not uncommon that you need a dozen such mini-methods/practices.

Because a method is attempting to give complete guidance for the work, it relies on a composition of practices. This is an operation merging two or more practices to form the method. Such a composition operation has to be defined mathematically in order to be unambiguous and precise. It has to be specified by a method expert with the objective to resolve potential overlaps and conflicts among the practices, if there are any. Usually most practices can be composed easily by setting them side

by side because there are no overlaps and conflicts, but in some cases, these have to be taken care of.

This is because, while practices are separate, they are not independent. They are not like components that have interfaces over which communication/interoperation will happen. They also share elements, so let us briefly look at what these might be. Inside a practice are, for instance, guidelines for activities that a user (e.g., a developer) is supposed to perform, and guidelines for work products (e.g., components) that a user is expected to produce. Although two practices may share the same work product, they contribute separate guidelines to this work product, and composing these two practices, resolving potential overlaps and conflicts, will require that you specify how the contributions must be combined in a meaningful and constructive way.

However, not just methods are compositions, but also practices can be compositions of smaller practices. Scrum, for instance, can be seen as a composition of three smaller practices: Daily Standup, Backlog-Driven Development, and Retrospective. (We will discuss these later when we come to Scrum in Part III.)

We will come back later to compositions when we have more details about practices in our knowledge bag. (If you want to have a peek into more on compositions now, take a look at Chapter 19.)

What eventually becomes a method or a practice is just a practical decision. To reiterate, a method is closer to the complete guidance you need whereas a practice (composed or not) is just one aspect (or several) of what you need to guide the team(s) to deal with all the “things” they need to deal with when developing software. An individual can create one or a few practices based on experience, but a method is always too big to be created by one individual without “borrowing” practices from others. We say “borrowing” within quotes, because it is an act without consent of the originator. Practices are successful because of the knowledge they provide, whereas methods are usually branded (like RUP, SAFe, DAD, Nexus, Less) and success is more about great marketing than about knowledge being provided.

When we say that a practice guides a team, we mean it is described one way or another to indicate what to do. How explicit a practice should be, i.e., how detailed the descriptions should be, depends on two factors: capabilities and background.

Capability. Capability refers to team members’ ability, based upon the knowledge they already have, to figure things out for themselves. Team members with high skill and capability need only a few reminders and examples to

		Tacit practices sufficient	Explicit practices needed
High			
Capability	Low	Tacit practices with coaching	Explicit practices with coaching

Common Different
Background

Figure 3.1 How explicit practices depend on capability and background.

get going. Others may need training and coaching to learn how to apply a practice effectively.

Background. If the team has worked together using a practice in the past or have gone through the same training, then they have a shared background. In this case, practices can be tacit. On the other hand, if team members have been using different practices, e.g., some have been using traditional requirements specifications while others have been using user story (a more modern way of dealing with requirements), then they have different backgrounds. In this case, practices should be described to avoid miscommunication.

How these two factors interact and influence the form that your practices should take is summarized in Figure 3.1.

As an example, in the case where a team's requirements challenges relate to different backgrounds and members do not know that much about requirements collaboration techniques, the team needs explicit practices and some coaching which a more experienced team member can provide out of the box. Additional factors to be considered, contributing to the need for practices, include the size of the team and how its members are geographically distributed.

3.3

There Is a Common Ground

Using a common ground as a basis for presenting guidelines for all practices will make it easier to teach, learn, use, and modify practices and easier to compare practices described using the same common ground.

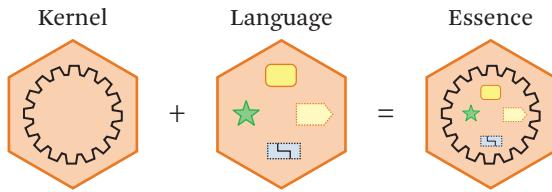


Figure 3.2 Essence and its parts.

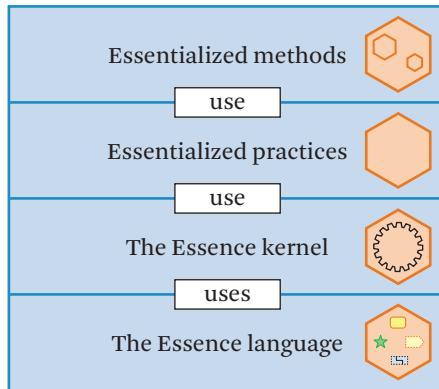


Figure 3.3 Essence method architecture.

Figure 3.2 illustrates Essence as this common ground, providing both a language and a kernel of software engineering.

The Essence language is very simple, intuitive, and practical, as we will show later in this section.

As previously described, it was left to the software engineering community to contribute practices, which can then be composed to form methods. Figure 3.3 depicts the relationships between methods composed using practices, which are described using the Essence kernel and the Essence language. As you can see in Figure 3.3, the notation used in the Essence language for practices is the hexagon, and for methods it is the hexagon enclosing two minor hexagons.

The practices are *essentialized*, meaning they are described using Essence—the Essence kernel and the Essence language. Consequently, the methods we will describe are also *essentialized*. In Part III you will see many examples of essentialized practices.

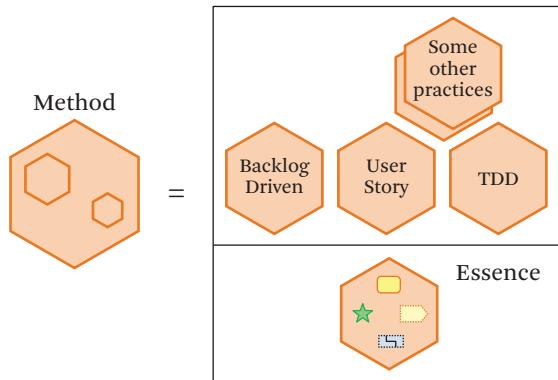


Figure 3.4 A method is a composition of practices on top of the kernel.

Essentializing not only means that the method/practice is described using Essence; it also focuses the description of the method/practice on what is essential. It doesn't mean changing the intent of the practice or the method. This provides significant value. We as a community can create libraries of practices coming from many different methods. Teams can mix and match practices from many methods to obtain a method they want. If you have an idea for a new practice, you can just focus on essentializing that practice and add it to a practice library for others to select; you don't need to "reinvent the wheel" to create your own method (see, e.g., Figure 3.4). This liberates that practice from monolithic methods, and it will open up the method prisons and let companies and teams get out to an open world.

As mentioned earlier, a team usually faces a number of challenges and will need the guidance of several practices. Starting with the kernel, a team can select a number of practices and support tools to make up its way-of-working. The set of practices that they select for their way-of-working is their method.

When learning a new practice or method, perhaps about use cases, or user stories, it is sometimes difficult to see how it will fit with your current way-of-working. By basing the practices on a common ground, you can easily relate new practices to what you already have. You learn the kernel once and then you just focus on what is different with each new practice.

Even if there are many different methods (every team or at the least every organization has one), they are not as different as it may seem. Examples of common practices are user story, test-driven development (TD), and backlog driven development. These terms may not mean much to you now, but in Part III we will give meaning to them. Right now, this combination serves just as an example of the relationship between a method and its practices.

Sidebar 3.1 How Much Does a Developer Need to Know About Methods?

You may be asking yourself at this point, do I really need to care about all of this method theory? Remember, a method is a description of the team's way of working and it provides help and guidance to the team as they perform their tasks. What the kernel does is help you structure what you do in a way that supports incremental evolution of the software system. In other words, it puts you in control of the way you work and provides the mechanism to change it.

The idea of describing practices on top of the kernel is a key theme of this book. A further discussion of how they are formed this way is found in Part III (see also Sidebar 3.1).

3.4 Focus on the Essentials

Our experience is that developers rarely have the time and interest to read detailed methods or practices. Starting to learn the essentials gets teams ready to start working significantly earlier than if they first have to learn "all" there is to say about the subject.

These essentials are usually just a small fraction of what an expert knows about a subject—we suggest 5%—but with the essentials you can participate in conversations about your work without having all the details. It helps to grow T-shaped people—people who have expertise in a particular field, but also broad knowledge across other fields. Such people are what the industry needs as work becomes more multi-disciplinary. Once you have learned the essentials it is relatively straightforward to find out more by yourself from different sources.

Some teams and organizations need more than the essentials, so different levels of details must be made optional.

3.5 Providing an Engaging User Experience

Many books have been written to teach software engineering. Some people interested in learning about ideas and trends in this space read these books, but the vast majority of software development practitioners don't—not even if they have bought the books. Thus, textbooks are not the best way to teach practices to practitioners. Modern method-authors have taken a different approach by presenting their methods as a navigable website.

Essence has taken a different approach by providing a hands-on, tangible user experience focused on supporting software professionals as they carry out their work. For example, the kernel can be accessed (and actually touched) through the

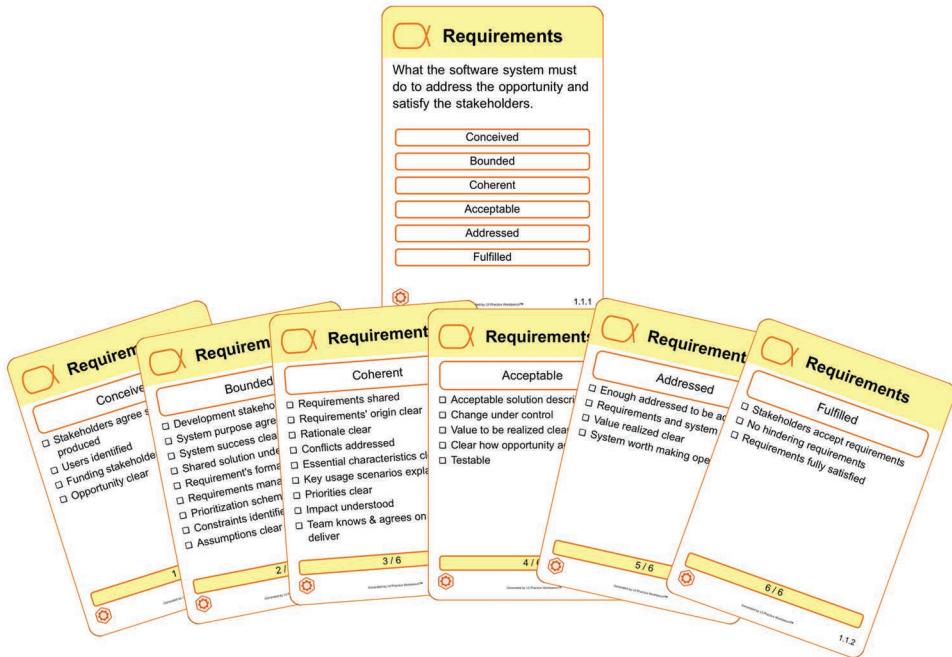


Figure 3.5 Cards make the kernel and practices tangible.

use of cards (see Figure 3.5). The cards provide concise reminders and cues for team members as they go about their daily tasks. By providing practical check-lists and prompts, as opposed to conceptual discussions, the kernel becomes something the team uses on a daily basis. This is a fundamental difference from traditional approaches, which tend to emphasize method description over method use and tend to be consulted only by people new to the team.

Cards have proven to be a lightweight and practical way to remember, but also to use in practice in a team. They make the kernel and the practices easy to digest and use. For this reason, throughout the book we will use cards to present elements in the kernel and in practices.

What Should You Now Be Able to Accomplish?

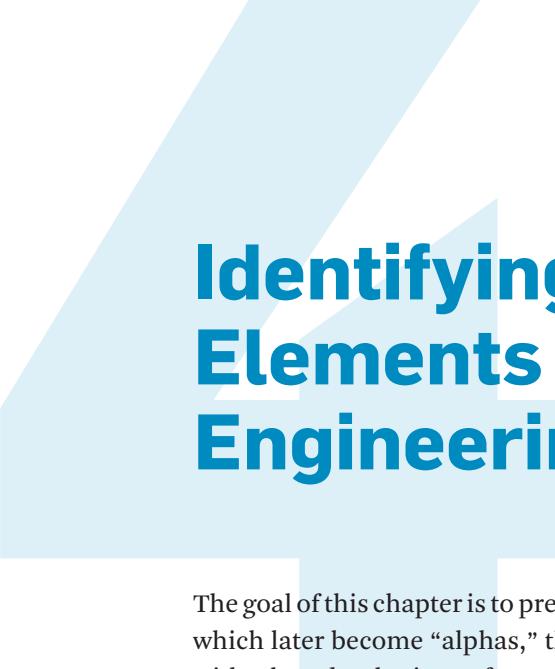
After studying this chapter, you should be able to

- name the key elements of Essence;
- distinguish between a practice and a method (give some examples of both);

- explain the concept of composition as a key technique to build methods using practices (and to support extensibility in Essence);
- explain the concept of tacit vs. explicit practices;
- explain the role of capability and background in deciding how explicit a practice should be; and
- explain the layered architecture of Essence and its elements.

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

Given that the reader is now equipped with ability to distinguish essential (i.e., the important) steps/aspects/decisions from those of minor importance, more knowledge can now be gained by proceeding in a given project and by working on the project with other people/stakeholders involved.



Identifying the Key Elements of Software Engineering

The goal of this chapter is to present the key elements of the development endeavor, which later become “alphas,” the building blocks of Essence—the things we work with when developing software. In this chapter, we discuss

- the key elements within software engineering that deliver value to the customer;
- the key elements in software engineering that are related to the targeted solution and development endeavor; and
- the role and importance of different stakeholders, requirements, and team composition.

This knowledge will help us to lay out a model of software engineering with areas of concern and key elements, which will create the basis for our understanding of Essence. To understand this model in practical application, we now rejoin Smith in his journey into software development.

4.1

Getting to the Basics

After Smith had been working in the software industry for several years, he had his fair share of ups and downs. He wished there were more ups than downs. Without a doubt, software engineering is a creative process, but Smith had come to recognize that there are some fundamental basics—some things to be mindful of, to avoid making unnecessary mistakes.

Smith's colleagues also recognized that, but they had difficulty articulating these fundamentals due to their different backgrounds, experience, and, consequently, the different terms they used. It seemed that every time a new team was formed, members had to go through a "storming and norming" process to iron out these terms before starting to deal with the challenges.

If you have been in the software industry for some time, you can empathize with Smith. For students new to software engineering, we want you to appreciate the complexities of a software development endeavor as you read this chapter and compare that against the complexities of your class, or of project assignments that you have worked on.

Essence was developed to help people like Smith and companies like Travel-Essence who rely heavily on software to run their business. What the contributors to Essence did was to lay down the foundation of software engineering for folks like Smith and yourself to cut short this startup period and ensure health and speed as your software development endeavors progress. The term health is discussed and defined later on for the area of software development. See, for example, Chapter 11 for a more detailed discussion.

Let's begin with some commonly used terms found in software engineering, which we will briefly introduce in *italics*. Regardless of size and complexity, all software development endeavors involve the following facets (see Figure 4.1):

- There are *customers* with needs to be met.
 - Someone has a problem or *opportunity* to address.
 - There are *stakeholders* who use and/or benefit from the solution produced, and some of these will fund the endeavor.
- There is a *solution* to be delivered.
 - There are certain *requirements* to be met.
 - A *software system* of one form or another will be developed.
- There is an *endeavor* to be undertaken.
 - The *work* must be initiated.
 - An empowered *team* of competent people must be formed, with an appropriate *way of working*.

These terms map out what software engineering is about. When something goes wrong, it is normally an issue with one or more of these facets. The way we handle these issues has a direct impact on the success or failure of the endeavor. We will

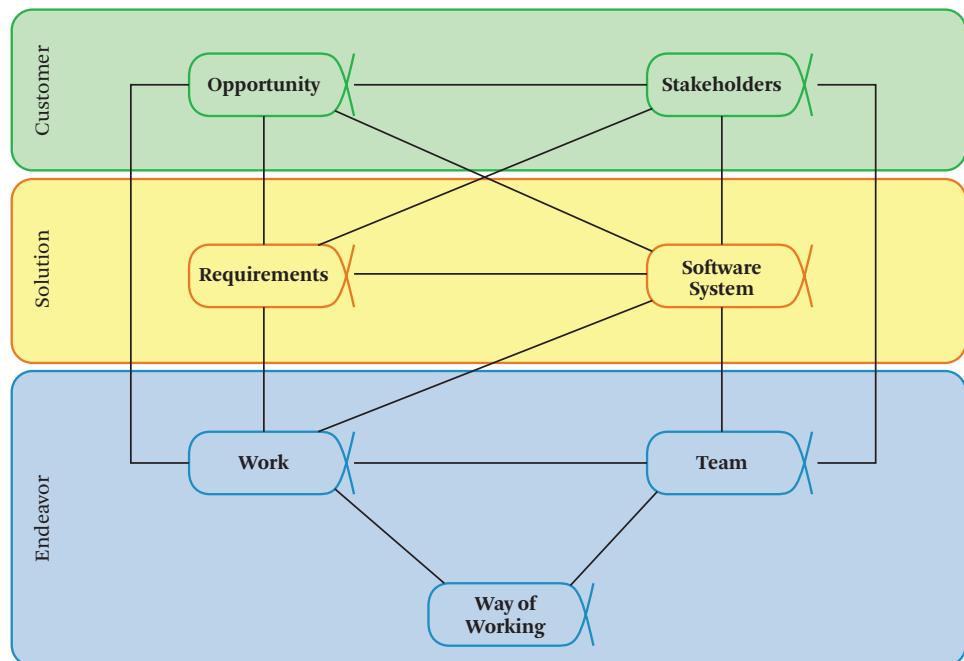


Figure 4.1 The things involved in all development endeavors.

now look at each of these facets in turn. Later, in Chapter 6, we will once more discuss issues and their relationships.

4.2

Software Engineering Is about Delivering Value to Customers

First, software engineering is about delivering value to customers. This value can be in improvements to existing capabilities or in providing new capabilities that are needed by the customer. (In our TravelEssence mode, customers are the users. They can be travelers or travel agents who make reservations on behalf of actual travelers.) Different customers would have different needs. If the endeavor does not deliver value, then it is a waste of time and money. As the saying goes, life is too short to build something that nobody wants!

4.2.1 Opportunity

Every endeavor is an opportunity for success or failure. It is therefore very important to understand what the opportunity is about. Perhaps you have heard of Airbnb. Airbnb started out in 2008 with two men, Brian Chesky and Joe Gebbia, who were

struggling to pay rent. They came up with the idea of renting out three airbeds on their living-room floor and providing their guests with breakfast. It turned out that during that time, there was an event going on in their city and many participants weren't able to book accommodations. Brian and Joe realized they were onto something. To cut the story short, Airbnb became a 1.3 billion USD business in 2016.

However, not all businesses grow and are successful like that. In fact, far more companies do not make it, and miss many opportunities. In fact, there has been a 90% failure rate for startups.¹ Many successful companies become failures due to missed opportunities. Thus, understanding opportunities is critical.

An opportunity is a chance to do something, including fixing an existing problem. In our context, an opportunity involves building or enhancing software to meet a need. Regardless of what the opportunity is, it can either succeed or fail. It can deliver real value, or it could be something that nobody wants.

As an example, our TravelEssence model revealed that customers like to engage travel staff because of the recommendations that the staff provides. The opportunity here is that if TravelEssence can provide recommendations online through a software solution, it can provide better service to customers, thereby shortening the time customers need to make a purchase decision. Of course, whether this opportunity is truly viable depends on many factors.

Thus, when working with an opportunity it is important to continuously evaluate the viability of the opportunity as it gets implemented.

- When the opportunity is first conceived, some due diligence is necessary to determine if it truly addresses a real need or a real new idea that customers are willing to pay money for.
- It would likely be the case that different solution options are available to address the opportunity, and some difficult choices will have to be made.
- When the solution goes live, it normally takes some time before the benefits become visible to customers.

4.2.2 Stakeholders

For opportunities to be taken up, there must be some people involved in the decision. The name we have for those individuals, organizations, or groups is *stakeholders*. Stakeholders who have some interest or concern in the system being developed are known as external stakeholders; those interested in the development

1. <https://www.forbes.com/sites/neilpatel/2015/01/16/90-of-startups-will-fail-heres-what-you-need-to-know-about-the-10/#43f76e846679>

endeavor itself are called internal stakeholders. In our TravelEssence case, internal stakeholders include a development team assembled to develop the new services for travelers, along with key managers in the organization who need to agree to the new venture. Examples of external stakeholders being affected by the system include a manager in our TravelEssence organization who needs to agree to fund the new software effort, or a traveler who might benefit by using the new services.

One of the biggest challenges in a development endeavor is getting stakeholders to agree. Before that can occur, they must first be involved in some way. The worst thing that could happen is that when all is said and done, someone says, “This is not what we want.” This happens too often.

Thus, it is really important early in the endeavor to:

- understand who the stakeholders are and what their concerns are;
- ensure that they are adequately represented and involved in the process; and
- ensure that they are satisfied with the evolving solution.

4.3

Software Engineering Delivers Value through a Solution

What sets a software development endeavor apart from other endeavors is that the *solution* that addresses the opportunity is via a good piece of software. Nobody wants a poor-quality product. Customers’ needs are ever evolving, so the solution needs to evolve as well, and for that to happen it has to be extensible. This extensibility applies to both the *requirements* of the solution and the *software system* that realizes the requirements.

In TravelEssence, the requirements for the solution cover different usage scenarios for different kinds of customers (e.g., new travelers, frequent travelers, corporate travelers, agents, etc.). The software system involves a mix of mobile applications and a cloud-based backend.

4.3.1 Requirements

Requirements provide the stakeholders’ view of what they expect the software system to provide. They indicate what the software system must do, but do not explicitly express how it must be done. They identify the value of the system in respect to the opportunity and indicate how the opportunity will be pursued via the creation of the system.

As such, requirements serve like a medium between the opportunity and the software system. Among the biggest challenges software teams face are changing requirements. Usually, there is more than one stakeholder in an endeavor, and stakeholders will of course have different and even conflicting preferences and

opinions. Even if there is only one stakeholder, he/she might have different opinions at different times. Moreover, the software system will evolve together with the requirements. What we see affects what we want. Thus, requirement changes are inevitable because the team's understanding of what's needed will evolve. What we want to prevent is unnecessary miscommunication and misunderstanding.

At TravelEssence, Smith encountered this when working on a discount program. The team had thought that this enhancement would be very simple. However, the stakeholders had different ideas on how long the program should last, which group of users the discount program should focus on, the impact of the discount program on reservation rates, etc. They had wanted to launch the discount program within a month's time, yet there was a great deal of debate even to the very last hour.

Thus, how a team works with requirements is absolutely crucial, with principles like:

- ensuring that requirements are continuously anchored to the opportunity;
- organizing the requirements in a way that facilitates understanding and resolves conflicting requirements;
- ensuring that the requirements are testable, i.e., that one can verify that the software system does indeed fulfill the requirements without ambiguity; and
- using the requirements to drive the development of the software system. In fact, code needs to be well structured and easy to relate back to the requirements. Progress is measured in terms of how many of the requirements have been completed.

4.3.2 Software System

The primary outcome of a software endeavor is of course the software system itself. This software system can come in one of many different forms. It could be the app running on your mobile phone; it could be embedded in your air conditioner; it could help you register for your undergraduate program; it could tally election votes. It could run on a single machine or be distributed on a server farm in data centers or across a vast network as in the Internet today.

Whatever the case, there are three important characteristics of software systems necessary before they can be of value to users and stakeholders: functionality, quality, and extensibility.

The need to have *functionality* is obvious. Software systems are designed and built to make the lives of humans easier. They each must serve some functions, which are derived from the software system's requirements.

The need for *quality* is easy to understand. Nobody likes a software system that is of poor quality. You do not want your word processor to crash when you are finishing your report, especially if you have not saved your work. You want your social media posts to be instantaneous. Thus, quality attributes like reliability and performance are important. Other qualities, such as ease of use or a rich user experience, are becoming more important as software systems become more ubiquitous. Of course, the extent of quality needed depends on the context itself. This again can be derived from the software system's requirements.

The third characteristic is that of being *extensible*. It can be said that this is another aspect of quality, but we want to call this out separately. Software engineering is about changing and evolving the software system, from one version to the next, giving it more and more functionality to serve its users. This evolution occurs over time as a series of increments of more functionality, where every increment is described by more requirements. This is illustrated in Smith's job at TravelEssence, which involves introducing changes to the existing travel reservation software system when TravelEssence introduces new discount programs, initiates membership subscription incentives, integrates with new accommodation providers, etc.

There are several important aspects of this evolution. First, it does not merely entail hacking or patching code into the software system. Otherwise, as the software system grows in size, it will be harder to add new functionality. Consequently, teams often organize software systems into interconnecting parts known as components. Each component realizes part of the requirements and has a well-defined scope and interface. As a student, the lessons you will learn about object orientation, etc. are about organizing the software system into manageable components.

Second, code needs to be well structured and easy to relate back to the requirements. Just as the requirements will evolve, the software system needs to be extensible to such changes.

Third, the evolution involves transitioning the software system across different environments, from the developer's machines, to some test environment, to what is known as the production environment, where actual users will be using the software system. It is not unusual to find that software that works on the developer's machines will have defects (or bugs) in the test or production environment. Many senior developers get irritated when they hear novices say: "But it works on my machine." A developer's job is not done until the system works well in the production environment. A quality software system must:

- have a design that is a solution to the problem and agreed to;
- have demonstrated critical interfaces;

- be usable, adding value to stakeholders; and
- have operational support in place.

4.4

Software Engineering Is Also about Endeavors

An endeavor is any action that we take with the purpose of achieving an objective, which in our case means both delivering value according to the given opportunity and satisfying stakeholders. Within software engineering, an endeavor involves a conscious and concerted effort of developing software from the beginning to the end. It is a purposeful activity conducted by a software team that achieves its goals by doing work according to a particular way of working.

4.4.1 Team

Software engineering involves the application of many diverse competencies and skills in a manner similar to a sports team. As such, a team typically involves several people and has a profound effect upon any development endeavor. Without the team there will be no software.

Good teamwork is essential for high performance. It creates a synergy, where the combined effect of the team is much greater than the sum of individual efforts. But getting to a high-performance state does not come naturally; instead, it results from a deliberate attempt to succeed.

To obtain this high level of performance, the team members should reflect on the way they work together and how they focus on the team goal.

Teams need to:

- be formed with enough people to start the work;
- be composed of personnel possessing the right competencies/skills;
- work together in a collaborative way; and
- continuously adapt to their changing environment.

When working in TravelEssence, Smith belonged to a development team. Although members of Smith's team had slightly different skill sets, they collaborated to achieve the team's goal together. Smith was particularly focused on backend technologies (i.e., the part of the software system running on the cloud), whereas Grace, a colleague, focused on front-end JavaScript and React Native technologies. (Since these technologies are not the emphasis of this book, you don't need to know them. Moreover, technologies change rather quickly. Instead, what we want to

emphasize is that effective teams have to address the opportunity presented to them to satisfy stakeholders.)

4.4.2 Work

When a team comes together to do the work of making the opportunity a reality in the software system they build, the purpose of all their efforts is to achieve a particular goal. In general, there is a limited amount of time to reach this goal: they must get things done fast but with high quality. The team members must be able to prepare, coordinate, track, and complete (stop) their work. Success in this has a profound effect upon meeting commitments and delivering value to stakeholders. Thus, the team members must understand how to perform their work and recognize if the work is progressing in a satisfactory manner.

Doing the work, then, involves:

- getting prepared;
- communicating the work to be done;
- ensuring that progress and risk are under control; and
- providing closure to the work.

In TravelEssence, Smith and his team managed their work through a backlog. The backlog is a list of things to do, which originated from requirements. They communicated regularly with their stakeholders to make sure that their backlog was accurate, up-to-date, and represented the stakeholders' priorities. In this way, they could focus on getting the right things done.

4.4.3 Way of Working

A team can perform their work in different ways and this may lead to different results. It can be performed in an ad hoc manner, meaning that you decide how to work while doing the work. For instance, while cooking a soup, you may not follow a recipe—you might decide on the fly which ingredients to use and in what order to mix and cook them together. When following an ad hoc way of working, the result may or may not be of high quality. This depends on many factors: among them, the skill of the people involved and the number of people involved in the process.

All of us are acquainted with the saying “too many cooks spoil the broth.” If too many cooks cook the soup in an ad hoc manner, the soup won’t taste good. Translating the analogy to software, if too many people participate in agreeing on how to do the job, the job will probably not be done well. There are many reasons

for this. One of them is that each person has his/her own idea of how to conduct the job and, often, they do not work in an orchestrated manner.

Smith's team addressed this issue by regularly looking at their prioritized backlog. They made sure that they correctly understood the scope of each item in the backlog, checking with stakeholders and getting feedback from them. Smith's team regularly examined their method or, in other words, their way of working. If things did not seem right, they made changes.

It is therefore important for team members to come into some kind of consensus regarding their way of working. Disagreements about the way of working are significant barriers to team performance. You would think that coming to an agreement would be easy. The truth of the matter is that it is not. On a small scale, within a single team, there is still a need for members to agree on the foundations and principles, followed by specific practices and tools. This would of course need to be adapted to the team's context, and evolve as the environment and needs change.

The way of working must:

- include a foundation of key practices and tools;
- be used by all the team members; and
- be improved by the team when needed.

In an industry scale, one of the things we hope to achieve through Essence is to simplify the process of reaching a common agreement. We do that at this scale by identifying a common ground or a kernel and having a way to deal with diversity of approaches. In the subsequent chapters, we will discuss the approach taken by Essence in greater detail.

In this chapter, we have introduced the following terms: opportunity, stakeholders, requirements, software system, work, team, and way of working. Essence will give these terms greater rigor and provide guidance to software teams on how to build a stronger foundation to achieve their goals.

What Should You Now Be Able to Accomplish?

After studying this chapter, you should be able to:

- list and explain the things involved in all development endeavors, related to the customer (i.e., opportunity, stakeholders), solution (i.e., requirements, software system), and endeavor (i.e., work, team, way of working);
- give examples of different types of stakeholders, together with their interests and concerns;

- explain the mediating role of requirements;
- name and explain the three key characteristics of software systems (i.e., functionality, quality, and extensibility); and
- explain what makes a good team.

Understanding the facets of software engineering covered in this chapter provides an overview of the main core of Essence. This core may seem fairly abstract at this point, but as you read on, you will recognize all these facets in the Essence alphas, and be able to apply them in more and more practical and detailed ways.

The Language of Software Engineering

The goal of this chapter is to present how the concepts discussed in the previous chapter are expressed as a language similar to how a programming language is expressed in computer science. Thus, we learn the concepts of alpha, alpha state, work product, activity, activity space, competency, and pattern, and how these concepts are expressed. We will show in this chapter

- the language constructs of Essence;
- the role of alpha states in two alphas and related checklists;
- the meaning and benefits of essentializing a practice; and
- the concept of cards as a practical way to use the various language elements of Essence.

What you learn here provides the necessary handles to use Essence in practice!

5.1

A Simple Practice Example

Essence provides a precise and actionable language to describe software engineering practices. Just as there are constructs like nouns and verbs in English, there are constructs in the Essence language in the form of shapes and icons. Just as a child learns a language by first using sentences without understanding the underlying grammar, we will introduce the Essence language through a simple pair programming practice. We choose this very simple practice because it is easily understandable even for students new to software engineering. We will introduce more sophisticated ones in later parts of the book.

We describe this programming practice as follows.

- The purpose of this practice is to produce higher quality code. In this case, we assume that the concept of code quality is understandable to the different members of the team.

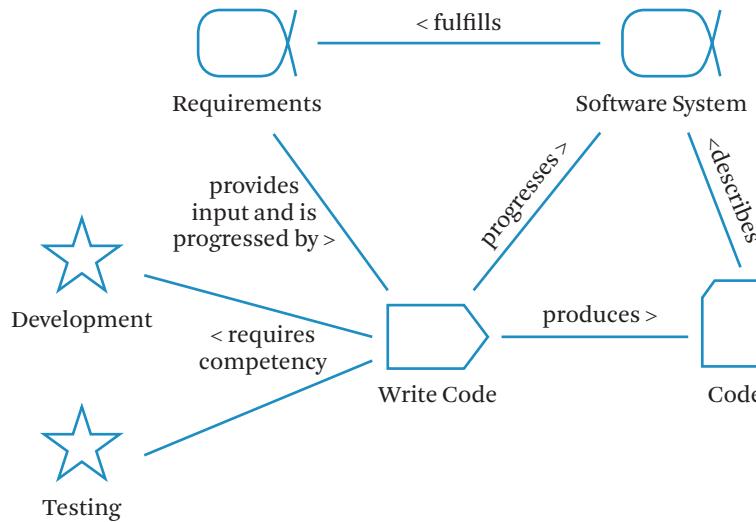


Figure 5.1 Simple programming practice (pair programming) described using Essence language.

- Two persons (students) work in pairs to turn requirements into a software system by writing code together.
- Writing code is part of implementing the system.

Essence uses shapes and icons to communicate the Things to Work With, the Things to Do, and Competencies. We shall describe each of these categories in turn and at the same time demonstrate how Essence provides explicit and actionable guidance for them, delivered through associated checklists and guidelines.

Figure 5.1 shows the elements in our simple programming practice. The shapes and icons are the constructs, i.e., the “nouns” and “verbs,” in the Essence language.

These different shapes and icons each have different meanings, as quickly enumerated in Table 5.1. As the text continues, we will go into greater depth for each one of them and explain their significance.

5.2

The Things to Work With

The “things to work with” in our programming practice are requirements, software systems, and code. If you compare the icons in Figure 5.1 with their definitions in Table 5.1, you will see that two of these are alphas, but one is a work product.

Essence distinguishes between elements of health and progress, which are called alphas, versus elements of documentation, which are known as work products. Alphas are the important intangible things we work with when conduct-

Table 5.1 Essence language as used in our simple programming practice

Element Type	Syntax	Meaning of Element Type
Alpha		An essential element of the development endeavor that is relevant to an assessment of the progress and health of the endeavor.
Work Product		A tangible thing that practitioners produce when conducting software engineering activities.
Activity		A thing that practitioners do.
Competency		An ability, capability, attainment, knowledge, or skill necessary to do a certain kind of work.

ing software engineering activities. They have states to help practitioners evaluate the progress and health of their endeavor. Work products, in contrast, are the tangible things that practitioners produce when conducting software engineering activities, such as requirement specifications, design models, and code. In TravelEssence, for example, the work products include user story index cards, use case narratives, and deployment scripts. As a student, your programming assignment worksheet is an example of a tangible work product.

Alphas are not work products. Alphas are elements of the development process that we want to track the progress of. An alpha is not tangible by itself, but it is understood or described by the work products that are associated with it. Their *states* describe how far in the lifecycle these aspects of development have progressed.

To illustrate this, in our programming practice example, Requirements and Software System¹ are alphas. Taking Requirements as our first instance: there will always be requirements for a software development endeavor, regardless of whether

1. In this book we have adopted a convention for the use of lowercase vs. uppercase letters. Lowercase letters are default for Essence language constructs such as practice, alpha, alpha state, alpha card, work product, and level of detail. However, when referring to names of elements defined in the Essence language we capitalize each word. Examples are names of alphas (Requirements, Software System), alpha states (Requirements: Conceived), work products (Use Case Narrative), and level of detail (Outlined). Italics are used independently, only to provide emphasis. When referring to instances of these elements we use lower case letters, for instance if we refer to the requirements of the Travel Exchange system, we use lower case letters.

you document them or not, or how you document them (e.g., as requirements items, test cases, user manuals, etc.). In some cases, the requirements for an endeavor may just exist in the heads of people. However, the alpha may be evidenced by providing one or more descriptions; that is, by attaching work products to the alpha. This is not always needed but very often desirable, in particular for larger endeavors.

Software System is another example of an alpha. Similarly, there will always be a software system, regardless of techniques used and documents produced. Software System too has a work product, which is code. Code is the physical thing that you as a developer write. The computer processes the Code into a Software System. The Software System itself (the alpha) is intangible, whereas the Code (the work product) is tangible.

We will develop these concepts throughout the book, but for now focus in a bit more on each of these differentiated Things to Work With.

5.2.1 Alphas

Alphas, again, are aspects of a software endeavor whose evolution we want to understand, monitor, direct, and control. Why are they called alphas? The developers of Essence, in searching for a word to describe these important key elements, chose the word alpha, which has been used to denote important elements such as a position in a social hierarchy or the first (often the brightest) star in a constellation (examples from Wikipedia). The alphas bound the work to be accomplished in progressing toward the provisioning of the software system. Alphas are portrayed as an icon that is a stylized variant of the first letter of the Greek alphabet (see Figure 5.1 and Table 5.1).

Simply put, alphas are the most important things you must attend to and progress in order to be successful in a development endeavor.

Although an alpha is commonly understood or evidenced by one or more associated work products (which thus describe particular aspects of it), it is not tangible by itself, so there must at least be tacit knowledge linked to each alpha. One form of this knowledge is the alpha's state.

5.2.2 Alpha States

Alphas have states that describe progression through a lifecycle. As an example, Essence defines the states of the Requirements alpha as follows.

Conceived. The need for a new system has been agreed upon.

Bounded. The purpose and theme of the new system are clear.

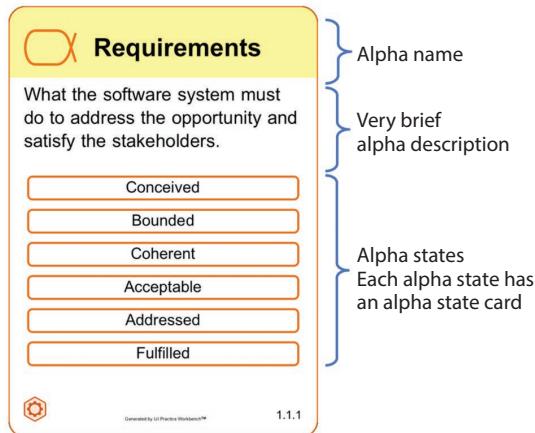


Figure 5.2 Requirements alpha card.

Coherent. The requirements provide a consistent description of the essential characteristics of the new system.

Acceptable. The requirements describe a system that is acceptable to the stakeholders.

Addressed. Enough of the requirements have been addressed to satisfy the need for a new system in a way that is acceptable to the stakeholders.

Fulfilled. The requirements have been addressed to fully satisfy the need for a new system.

To help remember the states of the Requirements alpha, Essence provides a poker card description, as shown in Figure 5.2. As we will discuss later, such poker-sized cards will also serve as teaching tools, and even make software engineering more fun through games.

Let's consider the layout of the card. At the top left, you see the icon representing an alpha. This distinguishes it as an alpha card rather than a work product card, competency card, or any other element. The top of the card also highlights the name of the element—in this case, the Requirements—followed by a brief description of the alpha and the states of the alpha. (This is a very concise overview of the requirements alpha. For more details, refer to the Essence specification.)

Cards are also available for each alpha state, as shown in Figure 5.3, which shows six cards, one for each of the Requirements alpha states. The layout of each of

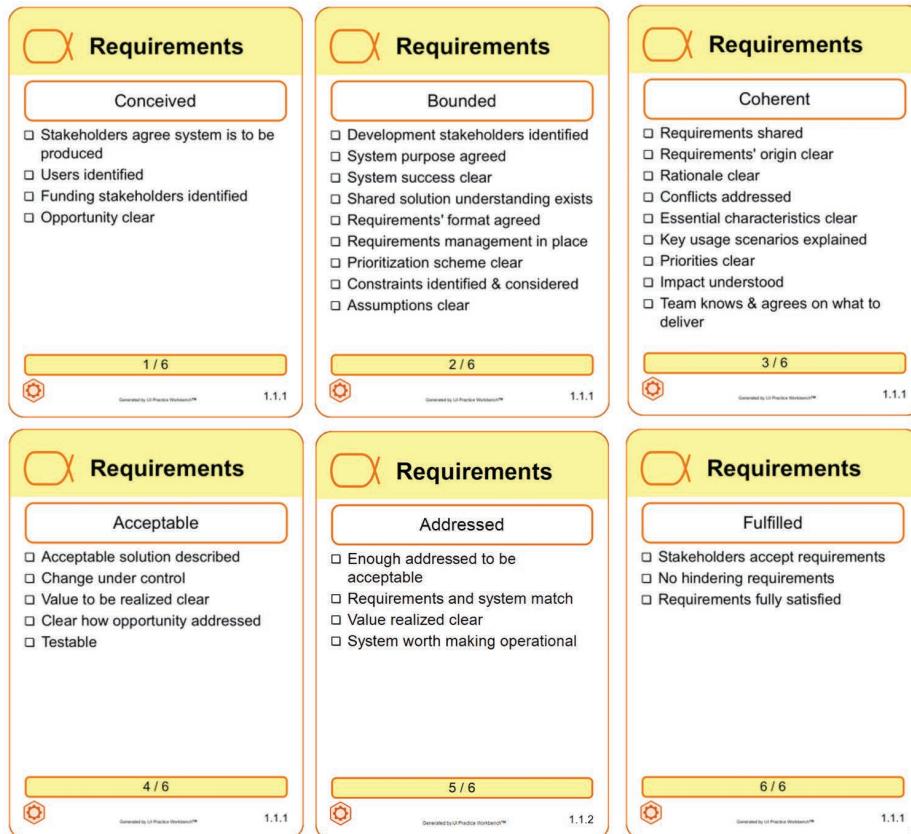


Figure 5.3 Requirements alpha state cards.

these cards comprises the alpha icon and . . . by the standard. (These abbreviations make it possible to fit the checklists on the cards for easy, quick reference. If you do not find them to be fully understandable, you should refer to the full checklists in the Essence Quick Reference Guide, available for download from www.software-engineering-essentialized.com.)

The other alpha in our example, the Software System, has the states shown in Figure 5.4.

The states are defined on the basis of an incremental risk-driven approach to building the Software System, first by establishing a sound architecture, and then by demonstrating key decisions about the Software System. These states are summarized as follows.

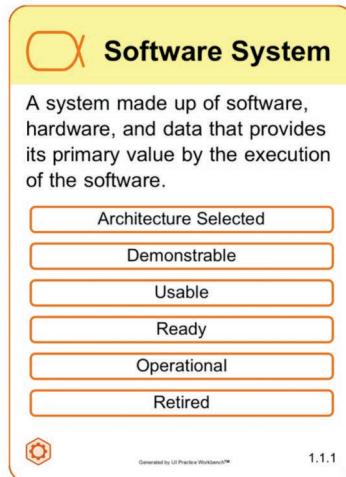


Figure 5.4 Software System alpha card.

Architecture Selected. Key decisions about the Software System have been made. For instance, the most important system elements and their interfaces are agreed upon.

Demonstrable. Key use of the Software System has been demonstrated and agreed.

Usable. The Software System is usable from the point of view of its users.

Ready. The Software System has sufficient quality for deployment to production, and the production environment is ready.

Operational. The Software System is operating well in the production environment.

Retired. The Software System is retired and replaced by a new version of the Software System, or by a separate Software System.

5.2.3 Work Products

Work products, again, are tangible things such as documents and reports, and may provide evidence to verify the achievement of alpha states. An example of a work product for the Requirements alpha might be some kind of a requirements specification—a description of the software system to be built. When a complete and accepted requirements document has been developed, it can be one way to

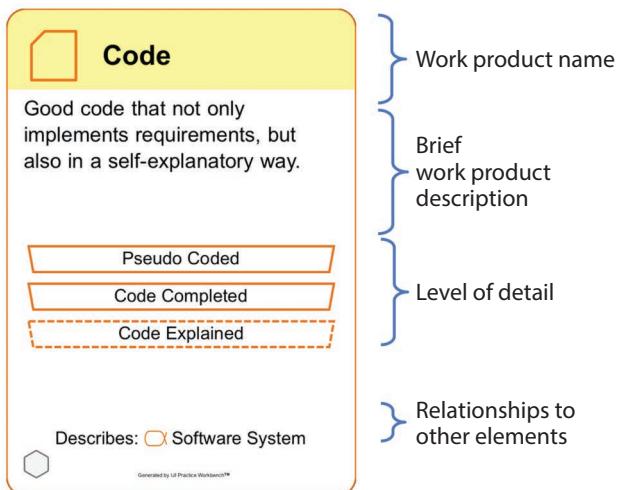


Figure 5.5 The Code work product card.

confirm having achieved certain checklists within a state of the Requirements alpha. Yet the fact that you have a document is not necessarily a sufficient condition to prove evidence of state achievement. Historically, documentation has not always provided an accurate measurement of progress. Thus, you may reach the same state without any documentation or with very brief documentation, as long as the checklist for that state has been achieved satisfactorily.

Code is the example of a work product for our programming practice. We provide a concise description of this work product in the form of a poker-sized card, as shown in Figure 5.5.

Note that the work products you produce do not belong to the common ground represented by the Essence standard. They are dependent on how you want to work (which practices you want to use). Thus, Essence does not specify exactly which work products are to be developed, but it does precisely specify what work products are, how you represent them, and what you can do with them.

Work products can have *different levels of detail*. In a development endeavor, the degree of detail needed in work products can vary greatly, depending on many factors such as past history of team members working together, customer requirements, regulatory requirements (e.g., regulation for software validation of medical devices), and organizational policies.

In TravelEssence, for example, Smith's team expressed requirements through use case narratives, which we will cover in Part III. These also have different levels of

detail. For simple requirements, Smith's team did not need as many specifics, but relied on more direct communication with their stakeholders. However, for more sophisticated requirements involving complicated calculation and decision rules, Smith had more complete explanations in his use case narratives.

Practitioners often have trouble figuring out how detailed the various work products they produce should be. Explicit practices, which we will cover in later parts of the book, can provide guidance on this.

5.3 Competencies

The team members applying the programming practice must, of course, be able to program; that is, they must have a development competency. Competencies are the abilities needed when applying a practice. Often, software teams struggle because they don't have all the abilities needed for the task they have been given. In these situations, a coach can help by explaining different ways the practitioner can address the problem, such as learning something that is missing in their competencies.

Figure 5.6 shows the card for the Development competency. Similar in format to the alpha and work product cards, the top of a competency card has the competency icon (a star) and the competency name followed by a very brief description of the competency. Then a list of competency levels follows.

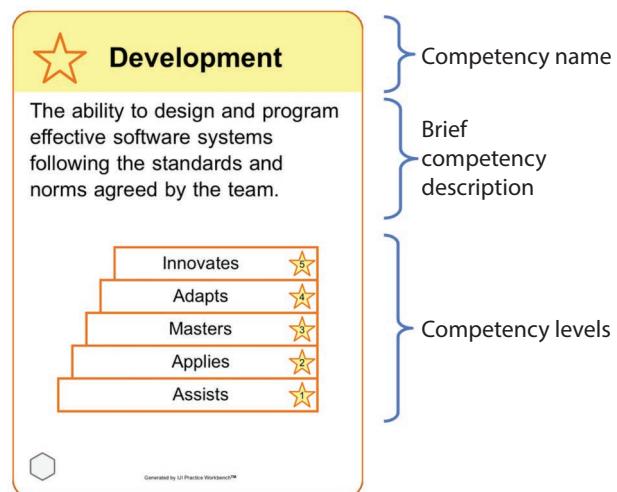


Figure 5.6 The Development competency card.

The Development competency has five levels of achievement.

- 1. Assists.** Demonstrates a basic understanding of the concepts and can follow instructions.
- 2. Applies.** Able to apply the concepts in simple contexts by routinely applying the experience gained so far.
- 3. Masters.** Able to apply the concepts in most contexts and has the experience to work without supervision.
- 4. Adapts.** Able to apply judgment on when and how to apply the concepts to more complex contexts. Can enable others to apply the concepts.
- 5. Innovates.** A recognized expert, able to extend the concepts to new contexts and inspire others.

Teams should be encouraged to conduct a self-assessment of their competencies and compare the results to the competencies they believe they need to accomplish their specific endeavor. This useful exercise can help software teams objectively determine any competency gaps they may have, which they can raise to management for resolution before serious problems occur that could hurt their team's performance.

5.4 Things to Do

To make progress in a development endeavor, or for that matter, any endeavor, all participants must do something. In our programming practice, this concerns writing code. The Essence language calls this an activity.

5.4.1 Activities

Activities are things that practitioners do, such as holding a meeting, analyzing a requirement, writing code, testing, or peer review. Similar to the challenges practitioners often face with determining the appropriate level of detail in work products, they also often struggle to determine the appropriate degree of detail or formality with an activity, or exactly how to go about conducting it. This is another motivation to specify explicit practices as they can provide guidance to practitioners in selecting appropriate activities, as well as in how to go about conducting each activity. A practice may include several activities that are specific to the practice being described. Thus, activities are specific and not standard—they are not a part of Essence. Figure 5.7 shows the activity card for Write Code. Its icon is an arrowed pentagon. (The *activity space* concept will be described in the next section.)

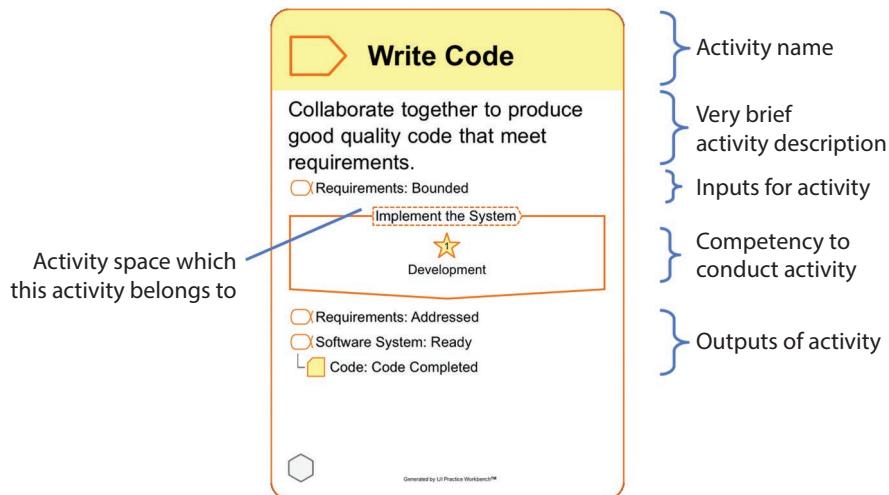


Figure 5.7 The Write Code activity card.

An activity is always bound to a specific practice and cannot “float around” among the practices. If you find an activity that needs to be reused by many practices, then you may want to create a separate practice including this activity. The alternative is that you decide not to reuse it, but each practice that potentially could have reused it will have to keep its own copy of that activity. Changes to one copy of it will not impact the other copies; they will just change independently of one another, which means no reuse.

5.5 Essentializing Practices

What we have just presented, using the Essence tools of diagrams (such as Figure 5.1, descriptions, and checklist cards), is a simple approach to concisely describe a practice, in this case a simple programming practice. We call the approach *essentialization*. Essentialization makes use of the Essence language, which we summarize in Figure 5.8.

As you can see, the language contains two more elements—activity space and pattern—than have been introduced thus far. See Table 5.2. These elements will be described further in the next chapter. The syntax and shape of these elements are defined in [Object Management Group, Essence \[2015\]](#).

To summarize, alphas are the essential things we work with. Each alpha defines a lifecycle, moving from one alpha state to another. Work products are the tangible

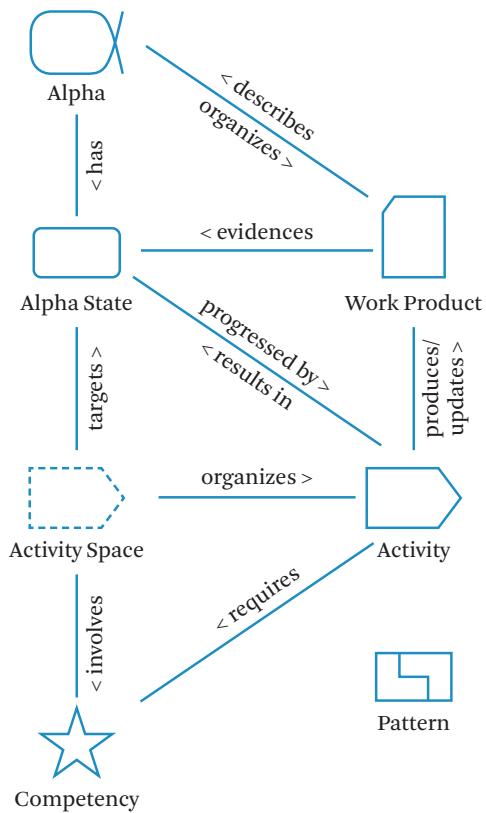


Figure 5.8 Elements of the Essence language and their relationships.

Table 5.2 Additional elements in the Essence language

Element Type	Syntax	Meaning of Element Type
Activity Space		A placeholder for something to do in the development endeavor. A placeholder may consist of zero to many activities.
Pattern		An arrangement of other elements represented in the language.

things that describe an alpha and give evidence for its alpha states. Activity is required to produce or update a work product. Activity spaces organize activities. To conduct an activity requires specific competencies. Patterns are solutions to typical problems. An example of a pattern is a role that is a solution to the problem of assigning work.

Essentializing a practice follows the following steps.

Identify the elements. First, build a list of elements that make up a practice.

The output is essentially a diagram like that in Figure 5.1 at the beginning of this chapter.

Draft the relationships between the elements and the outline of each element.

At this point, the cards are created.

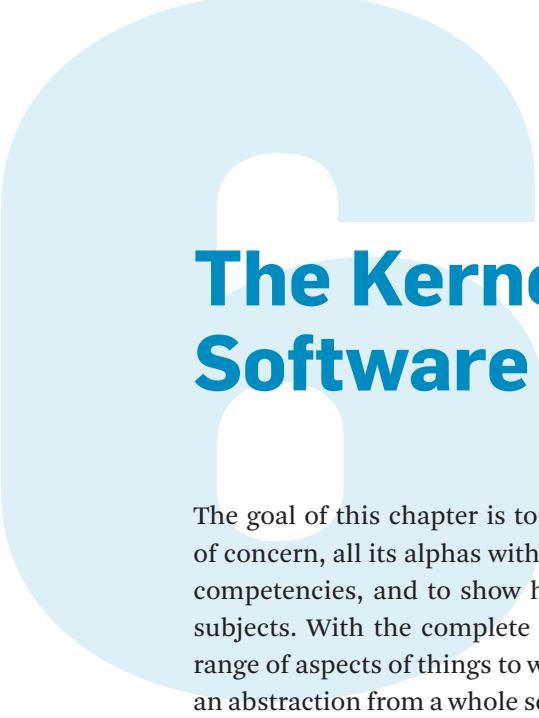
Provide further details. Usually, the cards will be supplemented with additional guidelines, hints and tips, examples, and references to other resources, such as articles and books.

In the rest of this book, especially in Part III, we will provide more examples of essentialized practices.

What Should You Now Be Able to Accomplish?

After studying this chapter, you should be able to:

- explain the concepts of alpha, alpha state, and work product, and how they are expressed;
- identify the concepts of activity and activity space (how they are expressed is shown in later chapters);
- identify the concepts of competency and pattern (how they are expressed will be shown in later chapters);
- explain the difference between an alpha and work product;
- explain the concept of alpha with an example (e.g., the Requirements alpha and its states) and the way these are expressed;
- explain the type of information present in Essence cards;
- explain the benefits of checklists in software engineering; and
- list the steps of essentializing a practice.



The Kernel of Software Engineering

The goal of this chapter is to present the complete Essence kernel with its areas of concern, all its alphas with states and their relationships, its activity spaces, its competencies, and to show how the pattern construct can be used for different subjects. With the complete picture presented, we can go on to map the whole range of aspects of things to work with, things to do, and so on. As a map is always an abstraction from a whole set of details, this chapter will provide first a bird's eye view. Specifically, we show here

- the basic organization of the kernel, based on three areas of concern;
- the way in which the various alphas are related to these areas of concern;
- all the alphas, activity spaces, and other elements in the kernel;
- the way in which activities relate to progress with respect to the alpha states;
- all the competencies in the kernel, plus insight on why competencies are important in software engineering; and
- how patterns can be used.

In Chapter 5, we introduced the Essence language of software engineering through a simple programming practice. In this chapter, we summarize the elements of software engineering as viewed through the lens of Essence.

6.1 Organizing with the Essence Kernel

Essence takes a structured approach in organizing the elements of software engineering. First, it organizes elements into three discrete *areas of concern*, each

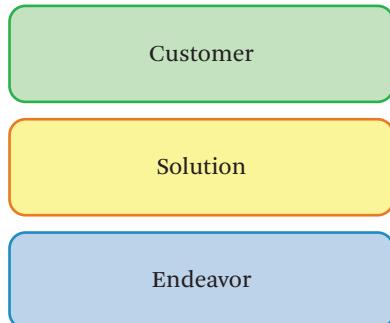


Figure 6.1 The three areas of concern.

focusing on a specific dimension of software development, as portrayed in Figure 6.1.

Customer. This area of concern contains everything to do with the actual use and exploitation of the software system to be produced.

Solution. This area of concern contains everything related to the specification and development of the software system.

Endeavor. This area of concern contains everything related to the development team and the way that they approach their work.

The kernel elements are fundamentally of four kinds:

1. The essential things to work with: the *alphas*
2. The essential things to do: the *activity spaces*
3. The essential capabilities needed: the *competencies*
4. The essential arrangements of elements: the *patterns*

Finding the right elements is crucial. They must be universally acceptable, significant, relevant, and guided by the notion that “In anything at all, perfection is finally attained not when there is no longer anything to add, but when there is no longer anything to take away . . . ”¹ When the originators of Essence set out to identify kernel elements, we took a conservative approach. We adopted only elements

1. Antoine de Saint-Exupéry, *Wind, Sand and Stars* (2002), trans. by Lewis Galantiere, Harcourt, p. 42.

we were sure would always be found in all types of software system endeavors. For instance, the element architecture was discussed as a kernel element. The idea was that while for many systems it is critical to identify an architecture, there are many simpler systems where architecture is not an issue. Thus, since it is not common to all endeavors, architecture is not a concern that every endeavor has to face, so it didn't qualify as a kernel element.

6.2

The Essential Things to Work With: The Alphas

While it is true that Essence attempts to provide a concise list of terms commonly found in software engineering, it is not like a static dictionary that you read. Essence is dynamic and actionable. By that we mean you can make it come alive. As we have already begun to show, you can use it in real situations to run an endeavor (e.g., a project). It includes the essential elements prevalent in every endeavor, such as Requirements, Software System, Team, and Work. These elements have states representing progress and health, so as the endeavor moves forward, these elements progress from state to state. The alphas in Essence are shown in Figure 6.2. The reader is already familiar with these terms to some degree and we go into more detail here now.

In the Customer area of concern, the team needs to understand the stakeholders' needs and the opportunity to be addressed.

Opportunity. The set of circumstances that makes it appropriate to develop or change a software system. The opportunity articulates the reason for the creation of the new, or changed, software system. It represents the team's shared understanding of the stakeholders' needs and helps shape the requirements for the new software system by providing justification for its development.

Stakeholders. The people, groups, or organizations that affect or are affected by a software system. The stakeholders provide the opportunity and are the source of the requirements and funding for the software system. The team members are also stakeholders. As much stakeholder involvement as possible throughout a software engineering endeavor is important to support the team and ensure that an acceptable software system is produced.

In the Solution area of concern, the team needs to establish a shared understanding of the requirements, and then implement, build, test, deploy, and support a software system that fulfills them.

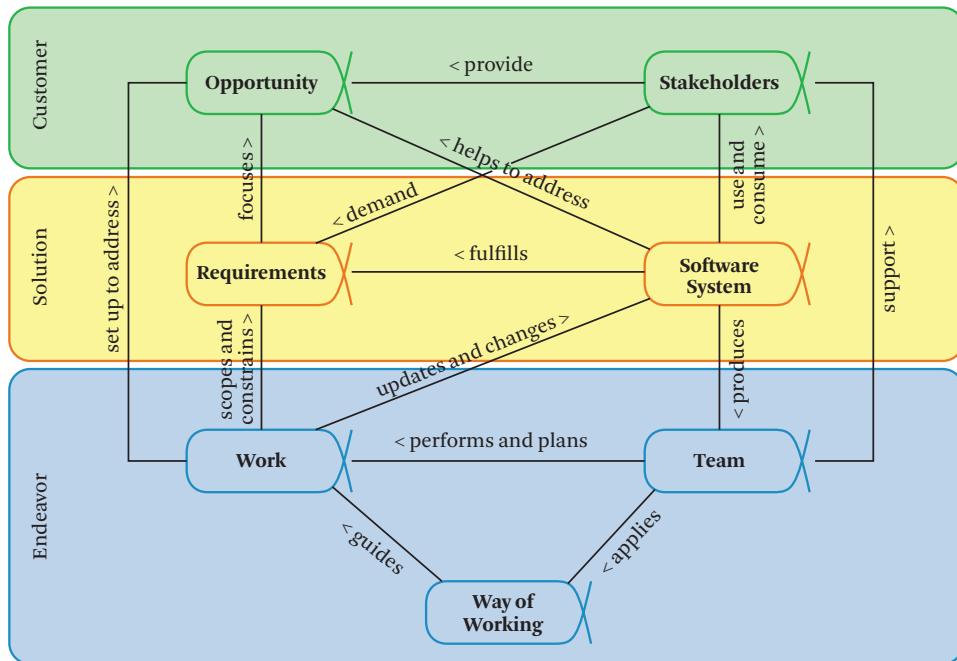


Figure 6.2 The Essence alphas and their relationships.

Requirements. What the software system must do to address the opportunity and satisfy the stakeholders. It is important to discover what is needed from the software system, share this understanding among the stakeholders and the team members, and use it to drive the development and testing of the new system.

Software System. A system made up of software, hardware, and data that provides its primary value by the execution of the software. The primary product of any development endeavor, a software system can be part of a larger software, hardware, business, or social system solution.

In the Endeavor area of concern, the team and its way of working have to be formed, and the work has to be done.

Team. A group of people actively engaged in the development, maintenance, delivery, or support of a specific software system. The team plans and performs the work needed to create, update, and/or change or retire the software system.

Work. Activity involving mental or physical effort done in order to achieve a result. In the context of software engineering, work is everything that the team does to meet the goals of producing a software system matching the requirements, and addressing the opportunity, that has been presented by the stakeholders. The work is guided by the practices that make up the team's way of working.

Way of Working. The tailored set of practices and tools used by a team to guide and support their work. The team evolves their way of working alongside their understanding of their mission and their working environment. As their work proceeds, they continuously reflect on their way of working and adapt it as necessary to their current context.

All alphas are vital during development endeavors. Thanks to their states, one may think that they may live their own lives. This is partly true. All alphas are related to one another and they complement each other by addressing their own aspects of the development endeavor.

As illustrated in Figure 6.2, the relationships are shown with arrows and there are many of them. They may be read as follows: Stakeholders provide Opportunity, which then helps to identify Requirements and focuses on the most important ones. These Requirements are then fulfilled by implementing a Software System. The Software System implementation addresses the Opportunity, and it is used and consumed by Stakeholders.

- It is the Team that produces the Software System by doing Work. The Work is set up to address the Opportunity and it implies updating and changing the Software System. Work is guided by a Way of Working that is applied by the Team while performing its Work. The Team is continuously supported by Stakeholders who provide feedback about the Software System to the Team.
- The OMG standard defines the states for each kernel alpha, as shown in Figure 6.3. The details of each state can be found in the Essence standard, and we will not go deeper into each of them in this chapter. You should later be able to download them from the book website.

So, now you are equipped with knowledge of things to work with that will in the following two sections be related to things you do and the competencies that are needed.

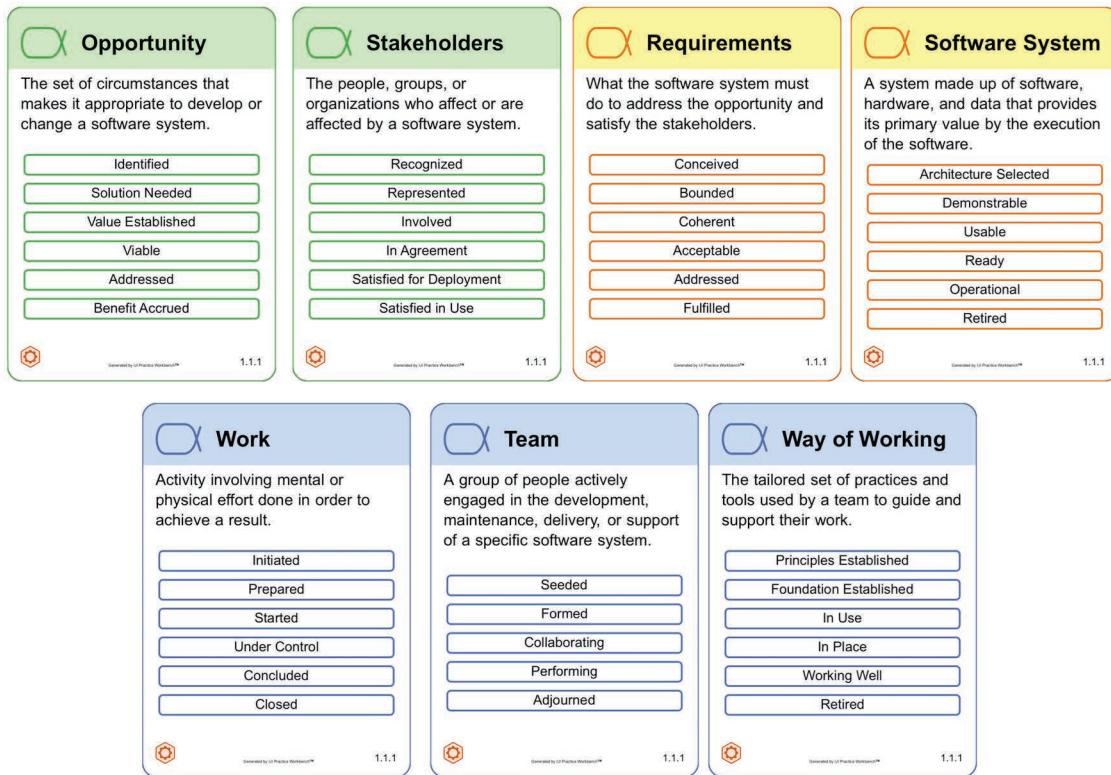


Figure 6.3 Essence kernel alpha states cards.

6.3

The Essential Things to Do: The Activities

In every development endeavor, you carry out a number of *activities*. Examples of activities include agreeing on a user story with a product owner, demonstrating the system to a customer representative, and estimating work. Essence as such does not define any activities (how your team goes about capturing and communicating the requirements can be very different from team to team). However, Essence defines a number of so-called *activity spaces*. You can think of activity spaces as generic (i.e., method-independent) placeholders for specific activities that will be added later, on top of the kernel. Since the activity spaces are generic, they can be standardized and are thus part of the Essence standard. The activity spaces are packages used to group activities that are related to one another. This helps to keep them organized, which in turn makes them easier for practitioners to find and use when they face common challenges where a practice could help.

The following *activity spaces* provide guidance to you and your team on the things to do in each of the three areas of concern as the team marches toward

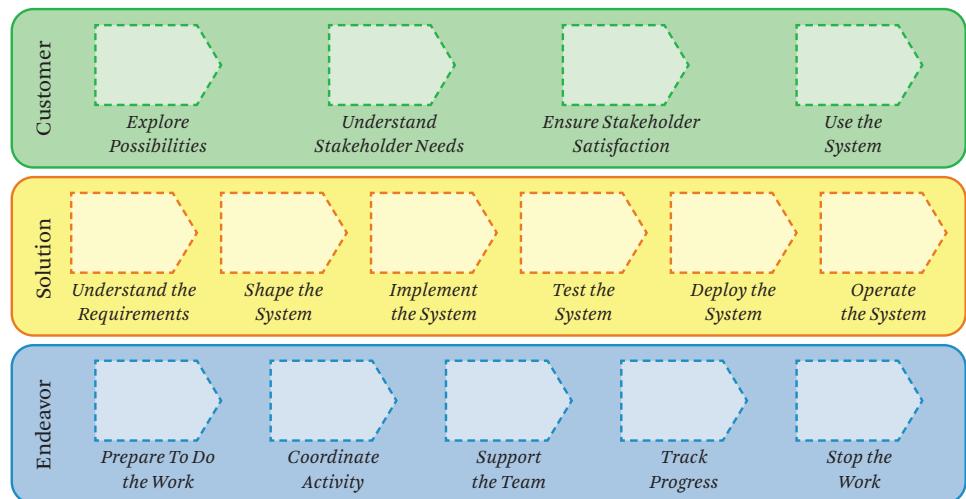


Figure 6.4 Essence activity spaces (from the Essence standard).

achieving the states specified in the alphas. The activity spaces of Essence are shown in Figure 6.4, where they are denoted by dashed arrowed pentagons and organized into the areas of concern.

Activity space cards have very similar contents to activity cards. Figure 6.5 shows an example of an activity space card. Going back to the simple programming example in the previous chapter, the Write Code activity would be allocated to the Implement the System activity space. The dashed symbol in the upper left-hand corner of the card, as in Figure 6.4, is the symbol for an activity space.

In addition to being placeholders for specific activities, the activity spaces represent the essential things that have to be done to develop software. They provide general descriptions of the challenges a team faces when developing, maintaining, and supporting software systems, and the kinds of things that the team will do to meet them. Each activity space then can (via the practices) be extended with concrete activities that progress one or more alphas—either alphas in the kernel or practice-specific alphas.

In the top row of Figure 6.4 there are activity spaces to understand the opportunity, and to support and involve the stakeholders.

Explore Possibilities. Explore the possibilities presented by the creation of a new or improved software system. This includes the analysis of the opportunity and the identification of the stakeholders.

Understand Stakeholder Needs. Engage with the stakeholders to understand their needs and ensure that the right results are produced. This includes identifying and working with the stakeholder representatives to progress the opportunity.

Ensure Stakeholder Satisfaction. Share the results of the development work with the stakeholders to gain their acceptance of the system produced and verify that the opportunity has been addressed.

Use the System. Observe the use of the system in a live environment and how it benefits the stakeholders.

In the middle row of Figure 6.4 there are activity spaces to develop an appropriate solution to exploit the opportunity and satisfy the stakeholders.

Understand the Requirements. Establish a shared understanding of what the system to be produced must do.

Shape the System. Form and structure, i.e., shape the system so that it is easy to develop, change, and maintain, and can cope with current and expected future demands. This includes the architecting and overall design of the system to be produced.

Implement the System. Build a system by implementing, testing, and integrating one or more system elements. This includes bug fixing and unit testing (see also Figure 6.5).

Test the System. Verify that the system produced meets the stakeholders' requirements.

Deploy the System. Take the tested system and make it available for use outside the development team.

Operate the System. Support the use of the software system in the live environment.

In the bottom row of Figure 6.4 there are activity spaces to form a team and to progress the work in line with the agreed way of working.

Prepare to Do the Work. Set up the team and its working environment. Understand and commit to completing the work.

Coordinate Activity. Coordinate and direct the team's work. This includes all ongoing planning and re-planning of the work, and re-shaping of the team.

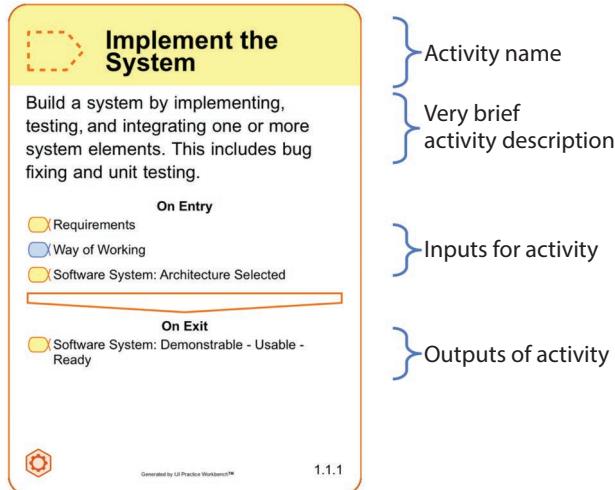


Figure 6.5 Implement the System activity space card.

Support the Team. Help the team members to help themselves, collaborate, and improve their way of working.

Track Progress. Measure and assess the progress made by the team.

Stop the Work. Shut down the development endeavor and handover of the team's responsibilities.

To make Figure 6.4 easy to read, the activity spaces are shown in a sequence from left to right in each row. The sequence indicates the order in which things are finished and not necessarily the order in which they are started. For example, you can start shaping the system before you have finished understanding the requirements, but you can't be sure you have finished shaping the system until you have finished understanding the requirements.

6.4 Competencies

To participate in a software endeavor you need to have competency in different areas. You need competency relevant to the specific tasks you are working on, but also other competencies to understand what your teammates are working on.

Competencies are defined in the kernel and can be thought of as generic containers for specific skills. Specific skills (e.g., Java programming) are not part of the kernel because such skills are not essential on *all* development endeavors. But

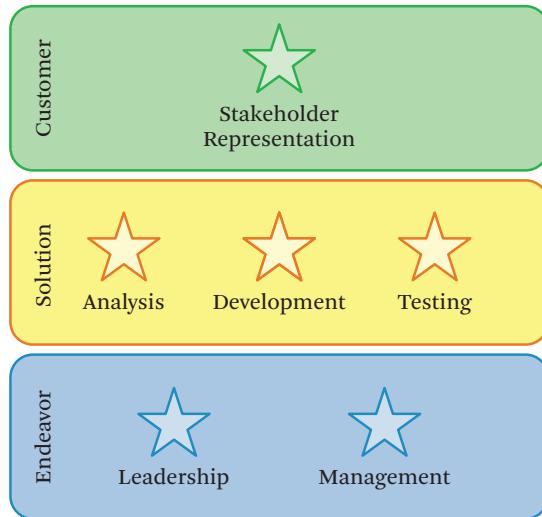


Figure 6.6 The kernel competencies.

competency is always required, and it will be up to the individual teams to identify the specific skills needed for their particular software endeavor.

A common problem on software endeavors is not being aware of the gap between the competencies needed and those that are available. The kernel approach will raise the visibility of this gap. Each of the competencies has a competency level, and this is the same across all of the kernel competencies. Thus, the levels that we presented in Section 5.3 are the same for all six competencies in Figure 6.6.

In the Customer area of concern, the team has to be able to demonstrate a clear understanding of the business and technical aspects of their domain and have the ability to accurately communicate the views of their stakeholders. This requires the following competencies to be available to the team.

Stakeholder Representation. This competency encapsulates the ability to gather, communicate, and balance the needs of other stakeholders, and accurately represent their views.

In the Solution area of concern, the team has to be able to capture and analyze the requirements and build and operate a software system that fulfills them. This requires the following competencies to be available to the team.

Analysis. This competency encapsulates the ability to understand opportunities and their related stakeholder needs, and to transform them into an agreed upon and consistent set of requirements.

Development. This competency encapsulates the ability to design, program, and code effective and efficient software systems following the standards and norms agreed upon by the team.

Testing. This competency encapsulates the ability to test a system, and verify that it is usable and that it meets the requirements.

In the Endeavor area of concern, the team has to be able to organize itself and manage its workload. This requires the following competencies to be available to the team.

Leadership. This competency enables a person to inspire and motivate a group of people to achieve a successful conclusion to their work and to meet their objectives.

Management. This competency encapsulates the ability to coordinate, plan, and track the work done by a team.

6.5 Patterns

Patterns are generic solutions to typical problems. In our daily life, we see patterns every day. In a classroom, we expect to have the teacher in front, with rows of desks and chairs for students. Such a pattern is designed for the teacher to transmit knowledge as efficiently as possible. Some classrooms are designed such that students are arranged in circles for greater discussion and discovery. In this case, the arrangement (i.e., pattern) is for the purpose of enhancing collaboration between students. Both arrangements are examples of patterns. There are many different kinds of patterns in software engineering. Examples of typical problems include how to set objective checkpoints, or how to conduct analysis and design. These logically lead to guidelines you may decide to give when describing a practice. Another typical problem to address might be how to assign responsibilities associated with work assignments to team members. This problem can be solved by defining a role pattern, as described in the next section. Patterns are optional elements (not a required element of a practice definition) that may be associated with any other language element. In Figure 6.7, we see a pattern named Student Pairs with a brief description. This pattern could be used to help two students who are working closely together on the same piece of code.

6.5.1 Roles

Many kinds of work require more than one competency. For instance, when developing software, you need more than just a Development competency. You also need to know how to find out what to develop, which requires an Analysis competency. A

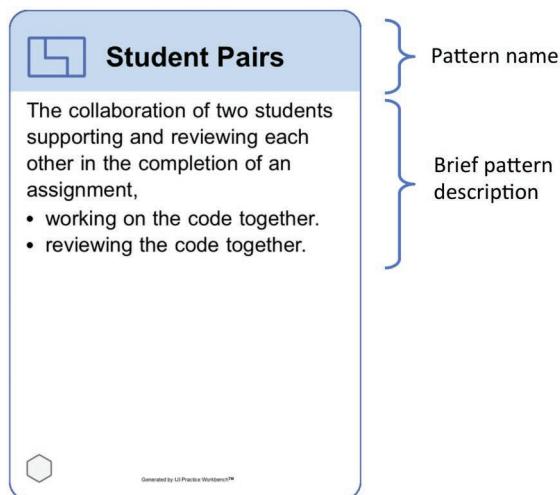


Figure 6.7 Student Pairs pattern card.

common way to ensure that an individual assigned this work has these competencies is to define a special kind of pattern, called a role. A role designates not only a set of specific responsibilities, but also the competencies required to fulfill them. A role can also specify a minimum level of each competency needed to do the job effectively.

When someone is given a role, she/he should have the needed competencies to succeed. If not, he/she should be offered additional training and coaching to carry out the role successfully.

6.5.2 The Checkpoint Pattern

As another example, a pattern could be used to define a checkpoint. They are frequently used within all types of development endeavors. A *checkpoint* is a set of criteria to be achieved at a specific point in time in a development endeavor. Checkpoints are key points in the lifecycle of a software endeavor where an important decision must be made. They are used by organizations as points where they stop and think about whether it makes sense to proceed or not, dependent on whether there is or isn't faith in what has been done. A decision could also be made to move forward but to go in a different direction because of a known problem or risk.

All of this is simply expressed in Essence by a set of alpha states that must have been achieved in order to pass the checkpoint. This pattern can be reused for other similar endeavors trying to get to the same checkpoint.

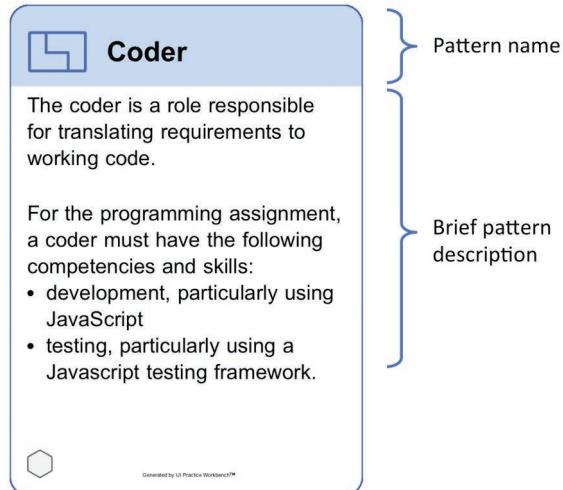


Figure 6.8 Coder role (pattern) card.

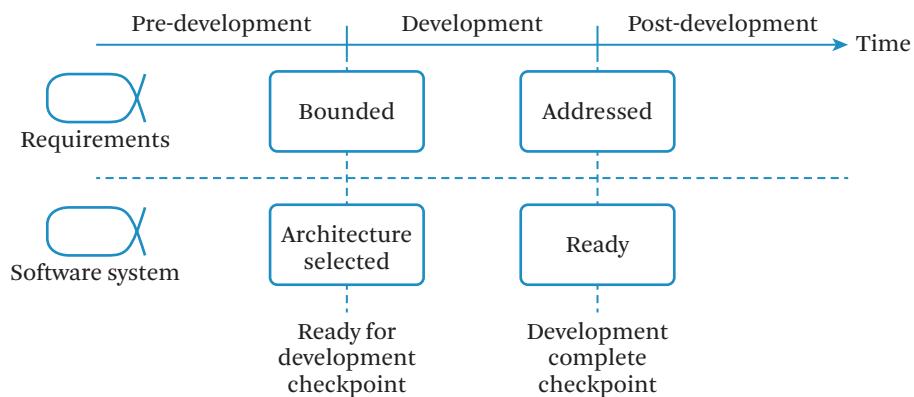


Figure 6.9 A checkpoint pattern example.

In the simplest case, the timeline of a development endeavor can be divided into three phases, pre-development, development, and post-development (see Figure 6.9). In the figure, the rounded boxes are states associated with the alphas.

Pre-development could be defined as the phase when the team and the stakeholders are determining what they need to do. The end of the pre-development phase needs to include any essential items that must be achieved before the team can begin developing the software system. One example might be funding approval to proceed with development.

Development is the phase when the team works collaboratively to create a software system that addresses the agreed-on needs and objectives. The phase ends when the software has reached a point where it can be used by real users.

Post-development is the phase when the software system is delivered to the user community and the software is used with real clients.

In this example, there are two checkpoints: Ready for Development and Development Complete. The criteria for these two checkpoints are expressed using alpha states. For example, the Ready for Development checkpoint is defined as Requirements: Bounded (an abbreviation of “Requirements alpha: Bounded state”) and Software System: Architecture Selected. The Development Complete checkpoint is defined as Requirements: Addressed, and Software System: Ready (for production).

Note that different kinds of development will define the checkpoints differently. For example, a development endeavor that has life-threatening consequences should it fail will have different checkpoints for exploratory development (for instance, developing a new product) than for maintenance (updates after delivery of the first version of the system).

Figure 6.10 provides a simple example of a checkpoint pattern in our programming practice. In this simple programming assignment, we just need the Software System to be demonstrable and the Requirements to be addressed.

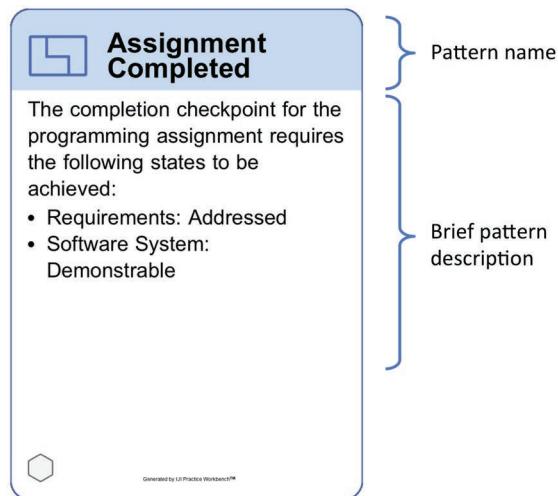


Figure 6.10 A checkpoint pattern card.

What Should You Now Be Able to Accomplish?

After studying this chapter, you should be able to:

- explain the kernel and its three areas of concern;
- explain how the various alphas are related to each other and mapped onto the areas of concern;
- identify the relationship between alphas and alpha states;
- explain in principle how progress is made by performing activities;
- identify the competencies;
- explain how patterns can be used to express frequently occurring combinations of Essence elements (alphas, activities, etc.) in software engineering;
- explain what makes Essence dynamic and actionable;
- explain the concept of activity spaces; and
- give an example of a checkpoint.

Now the basic picture of software engineering has been sketched completely. More detail will come in later chapters of this book. The concrete “instantiation” of these elements in a particular software development endeavor will apply these.

Reflection on Theory

The goal of this chapter is to present the role that Essence plays in working toward a theory of software engineering. Theoretical foundations bring technology to new heights. We have seen that in every engineering discipline, and it would apply for software engineering as well. In this chapter, we discuss

- that software engineering is still on its way toward appropriate theory;
- the need for a software engineering theory and the expected benefits of it;
- the differences between descriptive and predictive theories;
- the role Essence plays as a descriptive theory; and
- the role Tarpit plays as an example of a predictive theory.

The intent is to encourage readers to research and evolve a theory that helps teams and organizations deliver software much better.

In Chapter 2, we said that the Software Engineering Method and Theory (SEMAT) initiative called for

. . . 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.

Our work on Essence has provided a kernel that is able to represent any software engineering method, which we will demonstrate in later parts of this book. Essence is gaining traction in both industry and academia. But what about theory? What about the theoretical foundation behind software engineering?

In the paper “Where’s the Theory for Software Engineering?” by [Johnson et al. \[2012\]](#), these important questions are discussed, and with the permission of the authors, we reprint this discussion in its entirety as Section 7.1.

7.1

Where’s the Theory for Software Engineering?

Most academic disciplines are very concerned with their theories. Standard textbooks in subjects ranging from optics to circuit theory to psychology to organizational theory to international relations either present one single theory as the subject’s core or discuss a limited set of alternative theories to better explain the discipline’s essence to its students. A prime example is the central role of Maxwell’s equations in the subject of electrical engineering. It’s difficult to fathom what electrical engineering would be today without those four concise equations. A quite different example is the contested Domino theory, which heavily influenced American foreign policy in the 1950s to 1980s by speculating that one nation’s embrace of communism would entail the conversion of surrounding countries in a domino effect. Even though electrical engineering and political science are different in almost all respects, they’re both highly interested and invested in their theories.

7.1.1 What Is Software Engineering Theory?

Software engineering, however, isn’t concerned with the topic of theory—and if asked, the community surely couldn’t give a coherent answer about which is the most important one. Candidates might include theories with significant scope, such as formal systems theory, decision theory, organization theory, or theory of cognition. Collections of propositions might also be suggested, such as Alan Davis’s *201 Principles of Software Development*, Frederick P. Brooks’s propositions in *The Mythical Man-Month*, or SWEBOk. Specialized models such as Cocomo might also be candidates. We suspect that you’ll disagree with most of these proposals, but that just proves our point about the lack of consensus. Still, why are so many other fields explicit with their theories while software engineering is not?

Before discussing this question, we need to—in impossibly few words—describe what we mean by that multifaceted word “theory.” The best definition comes from a thoughtful article published in *Management Information Systems Quarterly*. According to author Shirley Gregor, there are many definitions for the term, but most theories share three characteristics: they attempt to generalize local observations and data into more abstract and universal knowledge; they generally have an interest in causality (cause and effect); and they often aim to explain or predict a phenomenon. Considering the purpose of theory, Gregor proposes four goals. The

first is to simply describe the studied phenomenon; SWEBOk could serve as an example. The second goal is to explain the how, why, and when of the topic; theory of cognition, for example, is aimed at explaining the human mind's limitations. The third goal is to not only explain what has already happened but also to predict what will happen next; in software engineering, Cocomo attempts to predict the cost of software projects. The fourth goal of theory according to Gregor is to prescribe how to act based on predictions; Alan Davis's 201 principles exemplify this goal.

7.1.2 Three Arguments

Returning to the main question—why the software engineering community seems so uninterested in discussing its theories—we can imagine three arguments: software engineering doesn't need theory, software engineering already has all the theory it needs, and software engineering can't have any significant, defining theories. We don't believe that these arguments are valid, but let's consider them individually.

7.1.2.1 Software Engineering Doesn't Need Theory

Software engineering is doing fine without explicit theories, so why change a winning formula? First, software engineering *isn't* doing fine. Numerous reports about failed IT projects have surfaced for decades now. Second, all engineering fields need theory. To build something good, you must understand the how, why, and when of building materials and structures. Indeed, you have to predict in the design stage the qualities of the end product if you want to avoid the painstaking labor of trial and error. In the words of Kurt Lewin, “There is nothing so practical as a good theory.” Third, for the many software engineering researchers employed at universities around the world, a researcher without a theory is like a gardener without a garden. According to philosopher Thomas Kuhn, the maturity of scientific disciplines can be measured by the unity of their theories. In the most established disciplines engaged in what Kuhn calls normal science, a paradigmatic theory defines a whole field (for example, Maxwell's equations, Einstein's theory of relativity, and Darwin's theory of natural selection). In a less mature phase, called pre-paradigm, a small number of theories, typically with ambitious explanatory scopes, compete for academic hegemony. This is the case in psychology, where cognitive theories challenge psychodynamic theories, and in international relations, where realist and liberalist theories battle for dominance. Kuhn doesn't offer a name for the phase before the pre-paradigm, in which there exists a large number of unrelated theories, because he considers this something less than science.

7.1.2.2 Software Engineering Already Has Its Theory

A discipline's significant theories should be able to provide answers to that discipline's significant questions. Considering software engineering, one of the most hotly debated questions concerns the choice of software engineering method. Although there are many opinions on the subject, we can name very few theories that attempt to answer the question. And to the extent that such theories exist, they aren't, as in other disciplines, given names, presented in textbooks, or debated at conferences. The same goes for other significant questions of software engineering, such as which programming language to use, how to specify system requirements, and so on. Note that many proposed software engineering methods, programming languages, and requirements specification languages exist, but very few theories explain why or predict that one method or language would be preferable to another, under given conditions.

7.1.2.3 Software Engineering Can't Have a Theory

Software engineering is a practical engineering discipline without scientific ambitions where rules of thumb and guidelines assume the role of theory. We can counter this argument by reiterating the tight connection between engineering and science. A typical definition of engineering is the one found in *Encyclopedia Britannica*: "the application of science to . . . the uses of humankind." Thus, no engineering without science. Second, it isn't true that there is no theory in the software engineering community. In a sense, theory is abundant. To the previously mentioned propositions, we could add Kent Beck's suggestion that the change cost curve could be logarithmic rather than exponential (*Extreme Programming Explained*, Addison-Wesley, 1999), Parnas's principle of information hiding (*On the Criteria to Be Used in Decomposing Systems into Modules*), Conway's law, Dijkstra's theory of cognitive limits as presented in the classical article "Go to statement considered harmful" (*Comm. ACM*, 1968), stepwise refinement, and so on. But all of these theories are small and most are casual, proposed by the author but rarely subjected to extended studies, and they explain only a limited set of phenomena. Furthermore, most of these theories aren't subject to serious academic discussion; they aren't evaluated or compared with respect to traditional criteria of theoretical quality such as consistency, correctness, comprehensiveness, and preciseness.

As should be evident by now, we don't believe that there's any rational reason for the lack of theoretical focus in software engineering. It's surely historical; born in the hurly burly of software practice, explanation and prediction were often merely glanced through the car window in the race between problem and solution. Today, tens of thousands of software engineering researchers are employed in the uni-

versities of the world, spending innumerable man-hours on software engineering research, but theory is still on the sidelines. To our knowledge, very few explicit attempts propose general theories of software engineering.

This has to change: without the predictive and prescriptive support of theory, software engineering would be relegated to the horribly costly design process of trial and error. With theory, we rise from the drudgery of random action into the sphere of intentional design. Software engineering is already full of implicit theory. We just need to bring it out into the open and subject it to the serious scientific treatment it deserves.

(This ends the quote from the paper “Where’s the Theory for Software Engineering?” [Johnson et al. 2012].)

7.2

Uses of Theory

Theory is generally used to (i) describe a phenomenon of interest and (ii) explain and predict that phenomenon. Prediction is of importance for engineering, as prediction allows engineers to build artifacts with predictable features and properties. Without prediction, engineering becomes no more efficient than a random walk. Prediction and explanation, however, require the capability to describe the salient features of the phenomenon in question. Therefore, description precedes prediction.

To describe something, a language is needed. In the case of mechanics, that language includes primitives such as mass, force, and acceleration. Such a language can then be used to express causal relationships, e.g., “if the force increases, then so will the acceleration,” or “if the mass increases, then the acceleration decreases.”

7.3

Essence Is a General, Descriptive Theory of Software Engineering

Unfortunately, as we explained previously, there is currently no widely accepted predictive general theory of software engineering. However, Essence takes the first step by proposing a coherent, general, *descriptive* theory of software engineering (i.e., a language of software engineering). The concepts included in Essence were designed to capture the most important features of the various phenomena of software engineering.

As a descriptive theory, Essence can be used to describe and facilitate discussion of future predictive theories of software engineering. However, Essence is not strictly limited to description; it does make a number of general claims regarding the relationships between concepts. For instance, the states of alphas have a certain

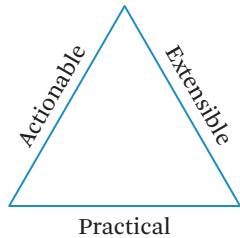


Figure 7.1 Guiding principles behind Essence.

order, excluding the movement between non-neighboring states. But a complete consideration of the causality between concepts and thus prediction is beyond the current version of Essence.

7.3.1 A Common Ground for a Descriptive Software Engineering Theory

What Essence has provided is a common ground for software engineering. This is an important idea, and it is worth taking a few moments to think about why establishing the common ground in this way is so important. More than just a conceptual model, as you will see through the practical examples in this book, the kernel provides

- a thinking framework for teams to reason about the progress they are making and the health of their endeavors;
- a framework for teams to assemble and continuously improve their way of working;
- the common ground for improved communication, standardized measurement, and the sharing of best practice;
- a foundation for an accessible, interoperable method and practice definitions; and
- most importantly, a way to help teams understand where they are and what they should do next.

What is it that makes the kernel anything more than just a conceptual model of software engineering? What is it that's really new here? This can be summarized as the three guiding principles shown in Figure 7.1.

7.3.2 Essence Is Practical

Perhaps the most important feature of the kernel is the way it is used in practice. Traditional approaches to software engineering methods tend to focus on support-

ing process engineers or quality engineers. The kernel, in contrast, is a hands-on, tangible thinking framework focused on supporting software professionals as they carry out their work.

For example, the kernel is tangible and deployed through the use of cards. The cards provide concise reminders and cues for team members as they go about their daily tasks. By providing practical checklists and prompts, as opposed to conceptual discussions, the kernel becomes something the team utilizes on a daily basis. This is a fundamental difference from traditional approaches, which tend to overemphasize method description as opposed to method use.

7.3.3 Essence Is Actionable

A unique feature of the kernel is the way that the “things to work with” are handled. These are captured as alphas, rather than work products (such as documents). An alpha is an essential element of the development endeavor; one that is relevant to an assessment of its progress and health. As shown in Figure 6.3, Essence has identified seven alphas. The alphas are characterized by a straightforward set of states that represent their progress and health. As an example, the Software System in Figure 6.3 moves through six states. Each state has a checklist, which specifies the criteria needed to reach the state. It is these states that make the kernel actionable and enable it to guide the behavior of development teams.

The kernel presents software engineering not as a linear process but as a network of collaborating elements: elements that need to be balanced and maintained to allow teams to progress effectively and efficiently, eliminate waste, and develop great software. The alphas in the kernel provide an overall framework for driving and progressing development efforts, regardless of the practices applied or the software engineering philosophy followed.

As practices are added to the kernel, additional alphas will be added to represent the things that either drive the progress of the kernel alphas or inhibit and prevent progress from being made. For example, the requirements will not be addressed as a whole but will be progressed requirement item by requirement item. It is the progress of the individual requirement items that will drive or inhibit the progress and health of the requirements. The requirement items could be of many different types; for example, they could be features, user stories, or use case slices, all of which can be represented as alphas and have their states tracked. The benefit of relating these smaller items to the coarser-grained kernel elements is that it allows the tracking of the health of the endeavor as a whole. This provides a necessary balance to the lower-level tracking of the individual items enabling teams to understand and optimize their way of working.

7.3.4 The Kernel Is Extensible

Another unique feature of the kernel is the way it can be extended to support different kinds of development (e.g., new development, legacy enhancements, in-house development, off-shore, software product lines, etc.). The kernel allows you to add practices, such as user story, use case, component-based development, pair-programming, daily standup meetings, and self-organizing teams, to build the methods you need. For example, different methods could be assembled for in-house and outsourced development, or for the development of safety-critical embedded systems and back office reporting systems.

The key idea here is that of practice separation. While the term “practice” has been widely used in the industry for many years, the kernel has a specific approach to the handling and sharing of practices. Practices are presented as distinct, separate, modular units, which a team can choose to use or not to use. This contrasts with traditional approaches that treat software engineering as a soup of indistinguishable practices and lead teams to dump the good with the bad when they move from one method to another.

7.3.5 How Does the Kernel Relate to Agile and Other Existing Approaches?

As we will show in more detail later in the book, the kernel can be used with all the currently popular management and technical practices including, for instance, Scrum, Kanban, risk-driven, iterative, waterfall, use case–driven development, acceptance test–driven development, continuous integration, and test-driven development. It will help teams embarking on the development of new and innovative software products and teams involved in enhancing and maintaining mature established software products. It will help all sizes of teams, from one-man bands to 1,000-strong software engineering programs.

For example, the kernel supports the values of the Agile Manifesto. With its focus on checklists and results, and its inherent practice independence, it values individuals and interactions over processes and tools. With its focus on the needs of professional development teams, it values teamwork and fulfilling team responsibilities over the following of methods.

The kernel doesn’t in any way compete with existing methods, be they agile or anything else. On the contrary, the kernel is agnostic to a team’s chosen method. Even if you have already chosen or are using a particular method, the kernel can still help you. Regardless of the method used, as Robert Martin has pointed out in his Foreword to [Jacobson et al. \[2013a\]](#), projects—even agile ones—can get out of kilter, and when they do teams need to know more. This is where the real value of

the kernel can be found. It can guide a team in the actions to take to get back on course, to extend their method, or address a critical gap in their way of working. At all times, it focuses on the needs of the software professional and values the “use of methods” over “the description of method definitions” (as has been the norm in the past).

The kernel doesn’t just support modern best practices; it also recognizes that a vast amount of software has already been developed and needs to be maintained. The software will live for decades and it will have to be maintained in an efficient way. This means the way you work with this software will have to evolve alongside the software itself. New practices will need to be introduced in a way that complements the ones already in use. The kernel provides the mechanisms to migrate legacy methods from monolithic waterfall approaches to more modern agile ones and beyond, in an evolutionary way. It allows you to change your legacy methods practice-by-practice whilst maintaining and improving the teams’ ability to deliver.

7.4 Toward a General Predictive Theory of Software Engineering

As mentioned above, a general (predictive) theory of software engineering ought to be able to provide answers to the discipline’s significant questions. Considering software engineering, there are a number of such questions, such as which method to deploy under what circumstances. There are several questions of similar importance, and there are numerous subquestions to each overarching one. Three more concrete examples of questions that a general theory of software engineering should be able to answer include these: (i) What is the explanation of Brooks’s Law? (ii) Are domain-specific languages here to stay? and (iii) Is continuous integration worth the effort? The first one is explanatory, and the remaining two are predictive.

There is currently no widely accepted general theory, but there are a few contenders. Here we outline one, called the Tarpit theory. This theory is described in greater detail in [Johnson and Ekstedt \[2016\]](#); here we can provide only a brief sketch.

The theory’s name stems from the communicative tragedy that, according to the theory, constitutes the core of the software engineering problem. On the one hand, people have a strong desire to explain to the evermore ubiquitous computers what we would like them to do. Computers, on the other hand, are obediently awaiting our instructions, and once received, those instructions will be performed with remarkable loyalty and zeal. But despite the eagerness of the two parties, the

relationship is a frustrated one, because it has proved strangely difficult to establish efficient means of communication between people and computers. Currently, almost twenty million software developers are employed worldwide, with the sole task to explain to the computers what it is that we really want from them. In the proposed theory's interpretation, Alan Perlis's famous epigram "Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy" refers precisely to the communicative intractability that separates the wondrous capabilities of the Turing machine from the many desires of humankind.

The theory is grounded in four subfields:

1. the theory of *software*, on the structure and meaning of computer programs;
2. the theory of *design*, on the process of decision-making;
3. the theory of *cognition*, on the limitations of human (developers') minds; and
4. the theory of *organization*, on the possibilities and limits of human cooperation.

In its most skeletal form, the Tarpit theory can be reduced to four propositions.

1. The goal of *software engineering* is to create *programs* that, when *executed* by a computer, result in *behavior* that is of *utility* to some *stakeholder*.
2. The two first challenges for *developers* are to (a) *make* appropriate *design decisions* and to (b) *translate* between *languages*.
3. These challenges appear because the *human mind* suffers from *limited cognitive capabilities*.
4. The *cognitive limitations* of a single *developer* can be partially mitigated by adding more *team members*, but this generates a third challenge, namely that of *team coordination*.

The authors claim that the theory is able to answer many of the most pertinent questions in software engineering, including the three questions above, which are discussed further in [Johnson and Ekstedt \[2016\]](#).

The Tarpit theory and Essence integrate nicely. Many concepts are identical or easily translated, such as *endeavor*, *stakeholder*, and *team*. Other concepts constitute the extension of each theory with the concepts of the other. Jointly, the integrated theory gains the descriptive and practical power of Essence and the predictive power of Tarpit.

7.5

A Theoretical Foundation Helps You Grow

The main concepts and principles of Essence have been presented and you will learn more about its application as we follow Smith's experiences in the remainder of the book. But how will Essence help you as you proceed in *your* career?

First and foremost, you will be provided with a common ground for understanding the scope as well as the application of software engineering methods and practices that will be of great benefit during your career. Certainly, as new practices and methods arrive on the scene, you will be prepared to easily integrate them into your own body of knowledge.

As Essence is accepted by software engineering companies around the world (a trend that is already happening), you will be able to reuse your knowledge when moving from working with one software system to working with another one, without having to learn a new method that utilizes different terminology introduced by its founding gurus. This was normally not possible before Essence adoption. In short, what Essence gives you is a means by which developers can become reflective practitioners and think about their own ways of working outside of commercial frameworks.

Once you become an experienced software professional, you will see additional values. For example, Essence provides guidance to help you assess the progress and health of your development endeavors, evaluate your current practices, and improve your way of working. It will also help you to improve communication, move more easily between teams, and adopt new ideas.

By providing a practice-independent foundation for the definition of software methods, the kernel also has the power to completely transform the way that methods are defined, and practices are shared. For example, by allowing teams to mix and match practices from different sources to build and improve their way of working, the kernel addresses two of the key methodological problems facing the industry. Teams are no longer trapped by their methods; they can continuously improve their way of working by adding or removing essentialized practices when their situation demands. Methodologists no longer need to waste their time describing complete methods: they can easily describe their new ideas in a concise and reusable way.

As a software engineering student, you may first learn the mechanics of software engineering (e.g., writing code, writing tests, and planning a project). But as you progress as a software professional, you will need to gradually develop your understanding and interpretation of this discipline. You will try to make sense of this world. In other words, you will start to form your own theory. As detailed previously, the Essence kernel provides a universe of discourse. It gives you a language to

describe phenomena, thus facilitating the discussion of future predictive theories of software engineering.

We hope that as you read the subsequent pages of the book, you will reflect on the observed phenomena. If you are already a software engineering professional, we encourage you to compare described phenomena in Smith's story with what you experience in the real world, and determine the value of the kernel in presenting the facts. As a researcher, we certainly encourage you to use the Essence kernel to frame your research hypothesis, findings, and conclusions. Regardless of who you are, the Essence kernel is a springboard toward more mature software engineering practices and a more mature software engineering discipline.

What Should You Now Be Able to Accomplish?

After studying this chapter, you should be able to

- explain what is expected from a software engineering theory and what in general are the purpose and goals of a theory;
- present arguments why software engineering needs a theory;
- explain the connection of a software engineering theory and predictability (give examples to support the explanation);
- explain the difference between descriptive and predictive theory;
- explain the role and key characteristics of Essence (i.e., that it is practical, actionable, extensible) in the context of a software engineering theory; and
- explain the propositions of the Tarpit theory.

Postlude to Part I

In Part I of the book, we introduced Essence, its motivation, its purpose, its principles, its language, and its kernel. In the remaining parts of the book, we will demonstrate how Essence helps you and your teams collaborate more effectively.

Many of us know how to develop software, but we have different ideas about how to go about doing it. Some of us know more than others and can figure it out; some need more guidance, or just enough to get started. We make the distinction between explicit knowledge and tacit (implicit) knowledge. Explicit knowledge is structured and expressed clearly, whereas tacit knowledge is unstructured and not written down—you just keep it in your head. It is knowledge emerging from experience and discussions. Software engineering needs a combination of explicit and tacit knowledge. It is not possible to be fully explicit and prescribe everything

a team needs to do, because every development is different and involves creative problem solving, which is by nature emerging as more experience is obtained. Thus, we need to strike a balance between the use of these two kinds of knowledge.

We take you on a journey in the rest of this book, describing more and more advanced use cases of Essence.

Recommended Additional Reading

This book is about the essentials of software engineering through the lens of Essence, the standard. There is, of course, more to be learned than what is presented here. In the following you will find references that at the time of writing this book are well suited as additional reading. At the website¹ that accompanies the book you will find more references related to software engineering in general, and also updates to the following references as such become available.

- Object Management Group, Essence—Kernel and Language for Software Engineering Methods (Essence) [[OMG Essence Specification 2014](#)]. This is the source for the Essence standard, where you may want to go to find more details and a formal definition of the Essence language. In this book here, we present only a subset of the standard, selected to be useful in everyday life. The standard is not set in stone; it undergoes continued improvements and it will usually be updated yearly.
- I. Jacobson, P.-W. Ng, P. E. McMahon, I. Spence, and S. Lidman, *The Essence of Software Engineering: Applying the SEMAT Kernel* [[Jacobson et al. 2013a](#)]. This is the first book on Essence written primarily for software development professionals. It provides the underlying ideas behind the Essence kernel, an overview of the kernel, and examples of how teams can use the kernel to run a software endeavor. This reference should be used when the reader is looking for details related to the ideas behind the kernel and more details on how the kernel can be used by teams.
- I. Jacobson, P.-W. Ng, P. E. McMahon, I. Spence and S. Lidman, “The essence of software engineering: The SEMAT kernel” [[Jacobson et al. 2012](#)]. This paper discusses the background, motivation, and overview of the Essence kernel. This reference should be used when the reader is looking for a short synopsis of Essence.

1. <http://www.software-engineering-essentialized.com/>.

- I. Jacobson, I. Spence, and P.-W. Ng, Agile and SEMAT: Perfect partners [[Jacobson et al. 2013b](#)]. This paper discusses how the Agile ideas and Essence are related. Essence is a tool to work with methods in an agile way. This reference should be used by readers that are using an Agile approach and want to understand more about how Essence can help their endeavor.
- P.-W. Ng, Theory based software engineering with the SEMAT kernel: Preliminary investigation and experiences [[Ng 2014](#)].
- P.-W. Ng, Integrating software engineering theory and practice using Essence: A case study [[Ng 2015](#)].
- P. E. McMahon, A thinking framework to power software development team performance [[McMahon 2015](#)]. This paper discusses specific features of Essence that can help teams improve performance in ways previous frameworks have fallen short. This reference should be consulted by teams that are using other frameworks and are looking for ways to improve their performance.
- P. Johnson, M. Ekstedt, I. Jacobson, Where's the theory for software engineering? [[Johnson et al. 2012](#)].
- P. Johnson and M. Ekstedt, The Tarpit—A general theory of software engineering [[Johnson and Ekstedt 2016](#)]. This paper describes the Tarpit general theory of software engineering.



Applying Essence in the Small—Playing Serious Games

The goal of this chapter is to introduce the concept of Essence cards and card games as a facilitation tool in a variety of settings and purposes: for instance, to ease the process of reaching a consensus within a team, or to promote discussion related to the health and progress of their endeavors. In this chapter, the following topics are introduced and discussed:

- the elements of cards and the benefits of card games as a facilitation tool in team communication;
- the benefits of card games as a learning mechanism not only for understanding the software engineering process and method, but also for developing basic mental abilities such as perception, attention, and decision making;
- the connection between four specific card games and the issues in software engineering that they help to solve; and
- the dynamics of state change during the software engineering process, including the fact that alphas rarely progress independently of each other (and rather progress in “waves”).

In Part I, we illustrated how the Essence kernel and practice elements can be represented as poker-sized cards. A card provides a concise description of the most important information about its element. Additional details are available in complementary guidelines.

Cards are handy and can act as reminders to practitioners. In addition, since they look like playing cards, it is natural to use them to play different games to

help the teams achieve goals when developing software. Teams use the cards to play games as facilitation tools in a variety of settings and purposes: for example, to help them obtain a consensus about their work. Cards are also a good way to introduce the kernel and practice elements to people who are new to Essence.

Software development is a highly intellectual and collaborative endeavor, as described in Part I. To achieve good results, most of the work is performed and planned by teams where the team performance is strongly dependent upon effective communication, common understanding, and trust. All team members must understand the endeavor purpose and its benefits, as well as its problems, and resolve any conflicts within limited time. For this reason, most of the serious games utilized in software engineering are collaborative in nature.

In this chapter, Essence is used in a tacit manner without explicitly described practices on top of it. However, we will also introduce some simple, small, but very useful techniques to facilitate working together within a team. We call these techniques games—serious games. Serious games may be entertaining, but their purpose is beyond entertainment. By simulating lifelike events, serious games aim at achieving some specific goal. Usually, this goal is to solve a particular real-world problem or to learn something new. Games also provide a powerful tool to help develop many different skills, not the least of which include basic mental abilities such as perception, attention, and decision making. Furthermore, in the context of this book, games can be highly reusable aids when carrying out multiple practices.

Again, these games are cooperative, rather than competitive. Thus, team members—or, as we will call them here, players—play in groups with one another, rather than against each other in a winner-take-all style. Serious games can get teams communicating in a way that transcends differences in background, experience, and perceptions.

To achieve common goals and maximal results, the players must express their thoughts clearly, listen to one another, share information and resources, learn from one another, identify solutions, negotiate, and make common decisions. Playing serious games with the Essence cards is one way to help team members observe how their teammates reason, and help them advance their own reasoning in a structured and systematic way. This is because by using natural words used by most development teams in naming the alphas, states, and checklists, the cards stimulate a team to discuss the issues related to the health and progress of their own endeavors.

In addition, once the team members understand where they are, they can use the cards to look ahead at states and checklists not yet achieved, thus stimulating discussion on what is most important to do next.

This chapter introduces four card games¹ that use the alpha state cards:

1. Progress Poker
2. Chasing the State
3. Objective Go
4. Checkpoint Construction

The card games in this chapter represent only a sample of what can be played. By identifying the purpose behind those games included here, and investigating other games and techniques (see [Alpha State Card Games \[2018\]](#), [Ng \[2013\]](#)), you will understand more of the goals that can be achieved by using the Essence kernel to facilitate work within the team.

8.1

Progress Poker

One of the most important questions teams often face is “Are we done?”—referring to a particular piece of work being completed. This has resulted in deep discussions over the definition of “done.” While there are several definitions of done, ours relates to the movement of an alpha from one state to another state. Take, for example, the Software System alpha, which over the lifecycle of a software system moves through six different states. The definition of done here is related to when you have done everything to qualify for a state change.

What does it take, for example, to move from Architecture Selected to Demonstrable?

Let’s now look at the individual checklist items in Figure 8.1. Take, for instance, the item, “Key architectural characteristics demonstrated.” Is the meaning of this checklist item clear? Well, some people would say they know what it means, but within a team, members can make several interpretations. One team member may say that this means that the key architectural characteristics have been agreed to and demonstrated to the team members, while another may think it means the agreement and demonstration must involve external stakeholders. It is true that the checklist items do not provide a precise definition. If they did, they would likely

1. <http://www.ivarjacobson.com/publications/brochure/alpha-state-card-games>

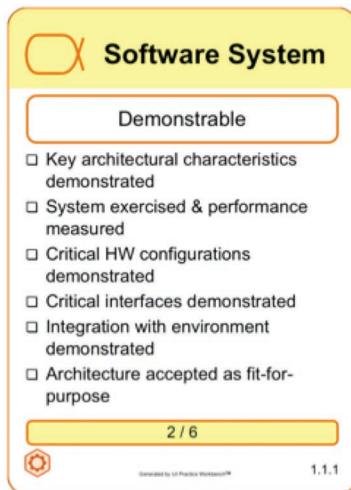


Figure 8.1 Software System: Demonstrable alpha state card.

be unintelligible to most developers. The items are not unambiguous, but they provide a hint of what needs to be done. They are subject to interpretation by the team members, who may each have different opinions on their meaning. One way to reach an agreement is by playing Progress Poker.

Progress Poker is a game played to facilitate discussion about and achieve understanding of the current state of a particular alpha. It is played one alpha at a time. This is a very important game that is reusable in many different situations. For instance, if you want to know where the team is in their endeavor, you play this game for each alpha and then you will know very precisely the progress the team has achieved. Just as in poker itself, this game's tools are a deck of cards (in this case a deck for each player), a table, and a set of rules and procedures. However, in contrast to the original game of poker, where the goal is to win, Progress Poker is a consensus-based game. Its main objective is to ensure that everybody is on the same page. To play Progress Poker, you need the Alpha Overview card and the Alpha State cards for the particular alpha whose current state the team is trying to understand. Figure 8.2 shows these cards for the Requirements alpha. (Note: Again, unlike the original game of poker, Progress Poker requires a deck of cards for each participant, and all players play all rounds.)

Here, there is no single winner. The winner is the whole team, and the winning hand is the team's common agreement on the endeavor status.

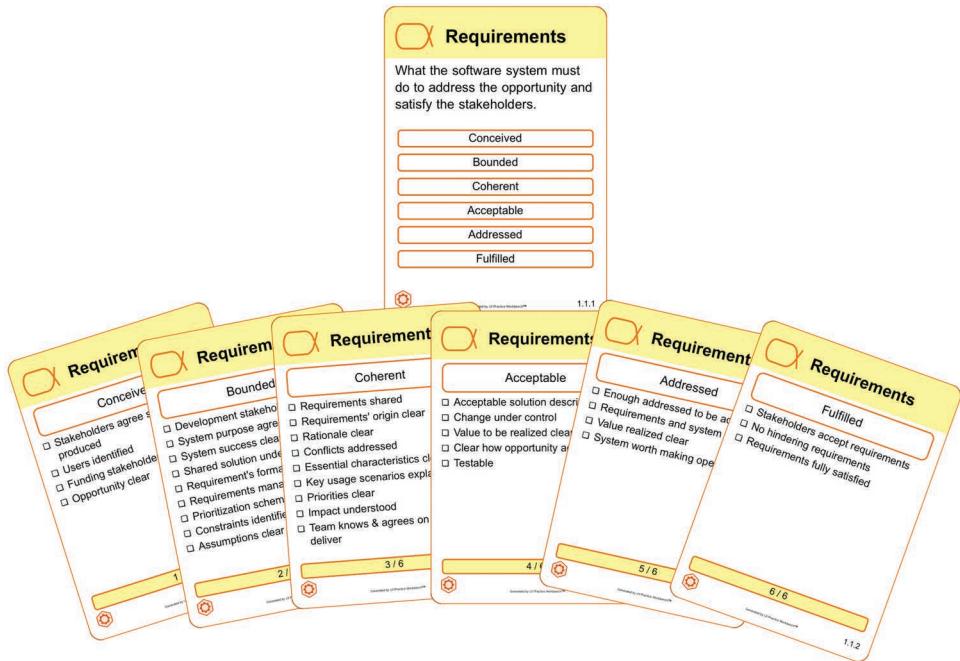


Figure 8.2 Tools needed to play Progress Poker (alpha card and alpha state cards).

Progress Poker may be played by any number of players. However, experience has indicated that it is most effective in teams consisting of three to nine players. When you have less than three players, often there are not enough different viewpoints to make the game worthwhile; when a team gets larger than nine, there is an increased risk of not reaching consensus. There is no fixed duration of the game: for teams familiar with the states and checklists, it may only take a few minutes to play. The game ends as soon as a consensus has been reached on the current state that has been achieved for a particular alpha.

Let us now sit in on a game of Progress Poker. Our players, the team members, gather around the table. To communicate that the Requirements alpha is under consideration, they place its alpha card in the center of the table, just as shown in Figure 8.2. Each player then uses his/her own set of state cards for that alpha, and identify from their own set the card that they think best represents the current state. True to the game's name, they should keep a poker face—i.e., keep the card of their choice confidential. After having selected his/her card, each player should place it face down on the table and wait until all team members have made their own

choices. By doing so, they make sure that everyone's initial opinion is not affected by anyone else's opinion. After everyone has chosen an alpha state card, all players turn their chosen state cards face up at the same time, and compare the results.

The results of playing Progress Poker may vary. In this simplest case, all team members have chosen the Bounded state. This means they have achieved a consensus among the players and, therefore, have the same understanding of the endeavor status. So they do not need to continue playing Progress Poker for the Requirement alpha, and the game is over.

Quite often, though, the players have different opinions about the state of the alpha under consideration, so the cards that they have chosen will differ. To reach an agreement, the players have to discuss their choices. Usually, those with the least and the most advanced states should start the discussion. They have to explain and motivate why they have chosen these alpha states. This, in turn, may lead to further discussions revealing the details of the endeavor status, and the next round of Progress Poker may be played. The number of rounds required depends on how soon the team reaches consensus. It may take three rounds for the team to arrive at a common agreement. In some cases, it may be necessary to create an explicit list of requirements to resolve the issue. Again, all players take part in all the rounds of the game, and the winning hand here is the agreement of the entire team. That is only achieved after everybody has put down the same state card.

A variant of Progress Poker can be played, in which a single team member plays the game alone, and then explains their results to the rest of the team. When playing this variant of the game, it is recommended that after the results are shared, the team leader encourages the team to provide feedback as to whether they agree with the assessment or not. This game can be effective at stimulating communication and collaboration in organizations that have cultures where decisions and direction are expected to come primarily from those in authority.

8.1.1 Benefits of Progress Poker

One of the benefits of Progress Poker is that everyone on the team gets involved, since each team member is forced to make an assessment and explain their views when their assessment differs from those of their teammates. Each team member must think through and talk about why they assessed the state the way they did. This helps teams avoid making decisions that are not rational, and it avoids the situation where just a few team members' decisions drive the team. The game also helps to ensure that all checklist items are considered.

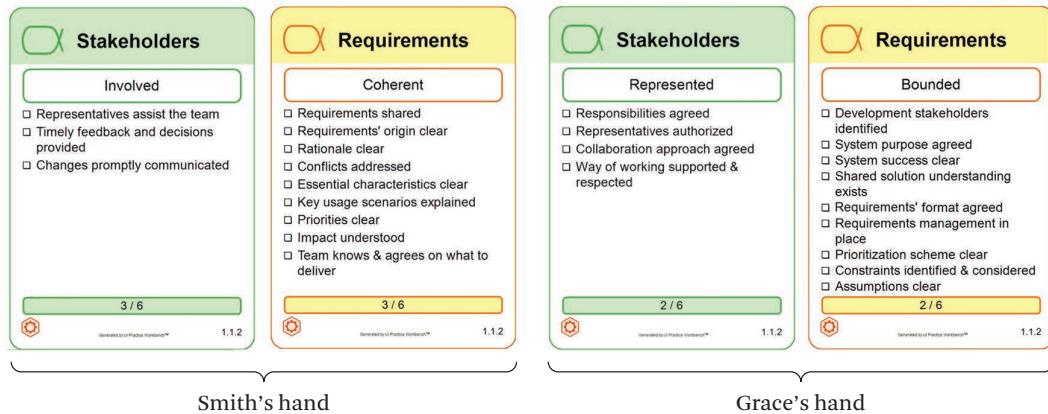


Figure 8.3 Differing views on state of Stakeholders and Requirements.

8.1.2 Example of TravelEssence Team Playing Progress Poker

We will now return to Smith and his team members. After learning about Essence and the alpha state cards, Smith was eager to share and apply what he had learned.

Smith and his team had been assigned to work on providing some new functionality for TravelEssence, specifically a recommendation engine for travelers. Based upon discussions with Angela, the business analyst, Smith found out that the goal of this recommendation engine was to recommend hotels and discount deals to travelers based on their travel history, TravelEssence's sales promotions, and so on.

Tom, Joel, and Grace were members of Smith's team. After his brief introduction to the ideas behind the alpha state cards, the team played Progress Poker seven times—one for each alpha—to determine their current state of development. They were already initially in agreement for all the alphas apart from the Stakeholders and Requirements alphas² (see Figure 8.3).

Smith's team couldn't immediately agree on what the Stakeholder and Requirements states were.

- Smith thought the Stakeholders were quite well represented and the members were actively involved helping the team. For example, he had been

2. If the team had been more experienced, they could have discovered this information without playing Progress Poker.

talking to Angela, who as a business analyst was a key stakeholder, about the requirements for the recommendation engine, and Angela had shared with Smith what she had learned from her analysis. Therefore, Smith deemed the Stakeholders state to be Involved.

- Smith thought the Requirements were fairly clear because of the work Angela had done. Therefore, he assessed the Requirements to be in the Coherent state.

However, other team members begged to differ.

- Grace pointed out that in the past business analysts frequently did not represent stakeholders well. They would say one thing, and when it was close to delivery, some higher-level authority would say something quite different. This resulted in unplanned rework late in the endeavor. Therefore, Grace saw the Stakeholders as Represented, but not Involved.
- Grace also pointed out that it was not clear how the new requirements would affect the existing functionality of the Hotel Management System (HMS). Therefore, Grace saw the Requirements as Bounded, but not Coherent.

Smith agreed that while Angela had completed some relevant analysis, she had not yet gone back to the customer stakeholders to gain their agreement, and that this created a risk to the endeavor. As a result, the entire team agreed that the Stakeholders alpha had achieved the Represented state, but they still had some work to do to get to the Involved state, and the Requirements had achieved the Bounded state, but more work was needed to get to the Coherent state.

Smith wanted to obtain a wider common understanding of the current state and what they needed to do to progress the endeavor. He rounded up his team again, as well as Angela, and this time asked Angela to play the Progress Poker game variant with his team in attendance. After a quick introduction, Smith asked Angela to lay out the cards, and he asked her which alpha states she thought had been achieved and what their current states were.

Angela was known to be rather terse when it came to requirements. She would think that the team had understood her when in fact they did not. Smith and his team members held their breath, as Angela shifted the cards around.

Surprisingly, Angela also recognized that Stakeholder involvement was not sufficient because she had not gone back to the customer stakeholders to gain their agreement, and that Requirements needed some more work. The source of the re-

uirements was Dave, who was Angela's boss. Dave needed more involvement, not just Angela. So in the end everyone agreed that the Stakeholders were Represented, but not Involved, and the Requirements were Bounded, but not yet Coherent.

8.2

Chasing the State

With Progress Poker you can obtain consensus within a team on which state an alpha has reached. You can of course also use Progress Poker for all alphas and agree on which states they all have. This must normally be determined many times during an endeavor. You need to find out where you are in the work. However, often teams are in agreement on which states most of the alphas have without having to play Progress Poker; they just look at the cards for each alpha and agree on which state has been achieved. A faster way to achieve team agreement on where they are for all the alphas is accomplished by playing the Chasing the State game.

This game is initiated by laying out all of one alpha's cards on a table. To the very left is the Alpha Overview card with a picture of all the states of the alpha. To the right are all the alpha state cards with the first state card on the left and the last state card on the right. See Figure 8.4.

Now the first card for the Stakeholder alpha is discussed. The team studies the first Stakeholder card (left) and agrees that all criteria are fulfilled, meaning that state Recognized has been achieved. See Figure 8.5.

As a consequence, that card is moved to the left on the table, as in Figure 8.6.

The game continues and the second Stakeholder state card is examined. The team agrees that this state has also been achieved, so that card is also moved to the left, close to the first state card. Thus, the third state card is studied. Here the team agrees that the criteria are not fulfilled, so this card is not moved; it stays where it is. Now we have the position in Figure 8.7.

Chasing the State continues on through the Opportunity alpha, the Requirements alpha, etc. until the Work alpha. In this example we ended up with the situation portrayed in Figure 8.8.

The state of the endeavor can be described as follows: Stakeholders, 2; Opportunity, 2; Requirements, 2; Software System, 1; Team, 2; Way of Working, 1; and Work, 1.

In this particular game it is assumed that everything goes smoothly and the team can easily agree upon the states that have been achieved. That is not always the case, so if the team can't easily agree, they can play Progress Poker for the particular alpha that is not easy to agree upon.

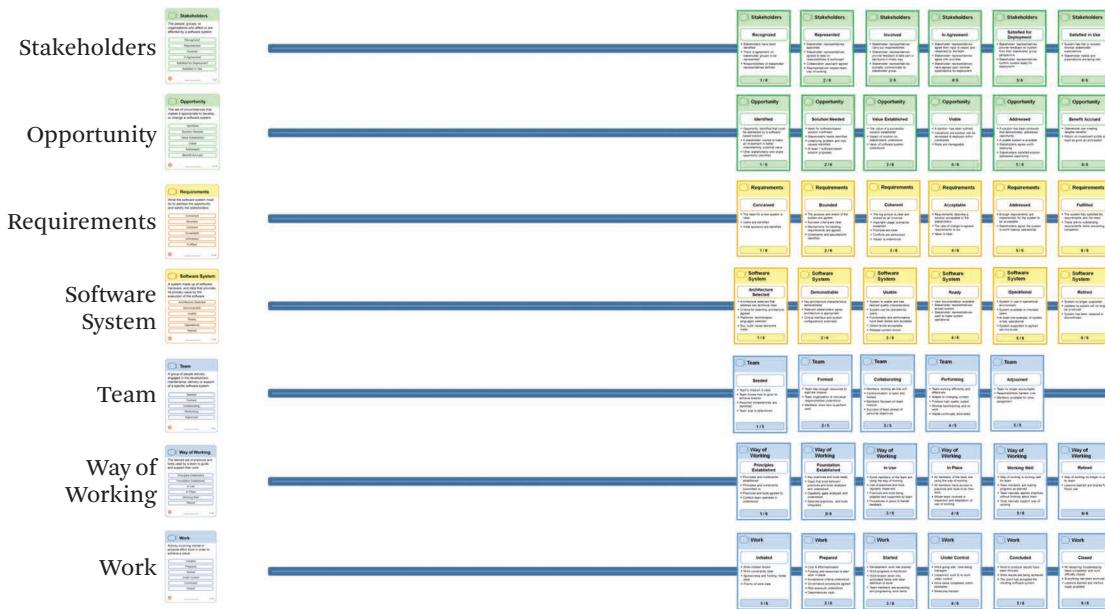


Figure 8.4 Initial position for playing the Chasing the State game.

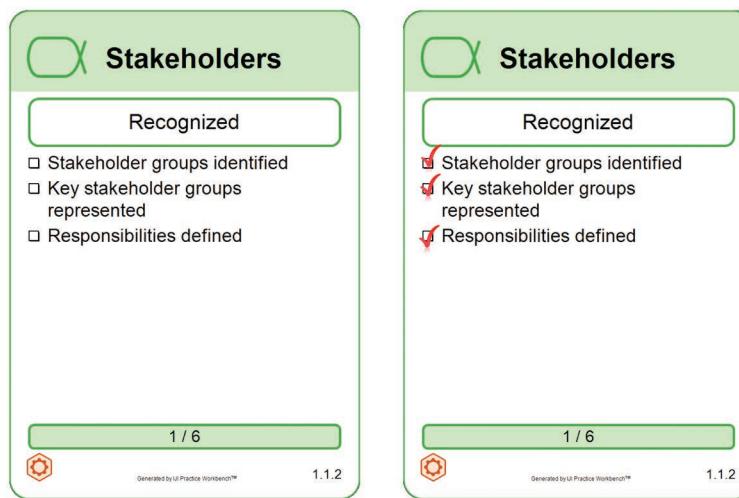


Figure 8.5 Stakeholder: Recognized before and after discussion.

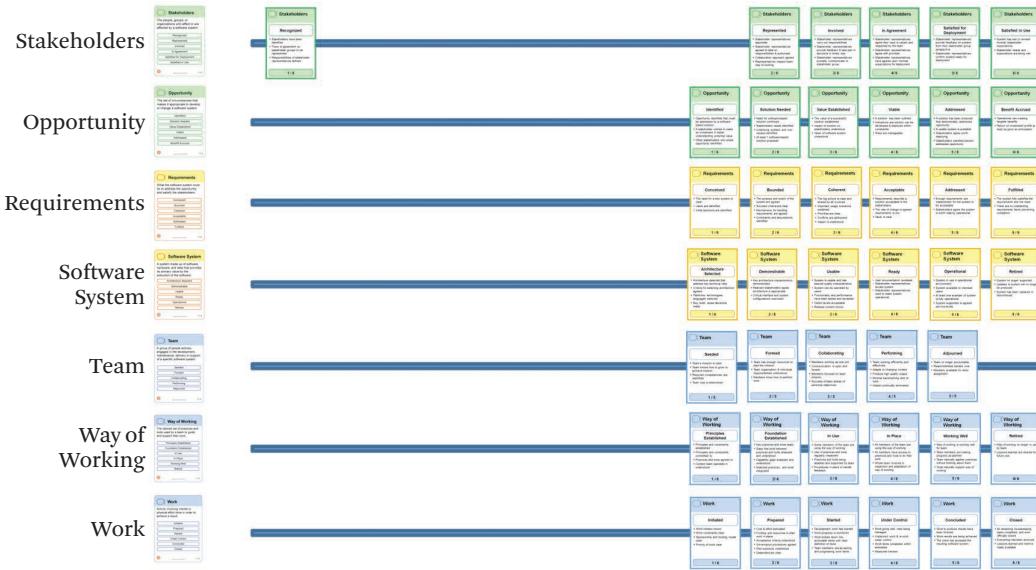


Figure 8.6 The Stakeholder alpha has reached its first state.

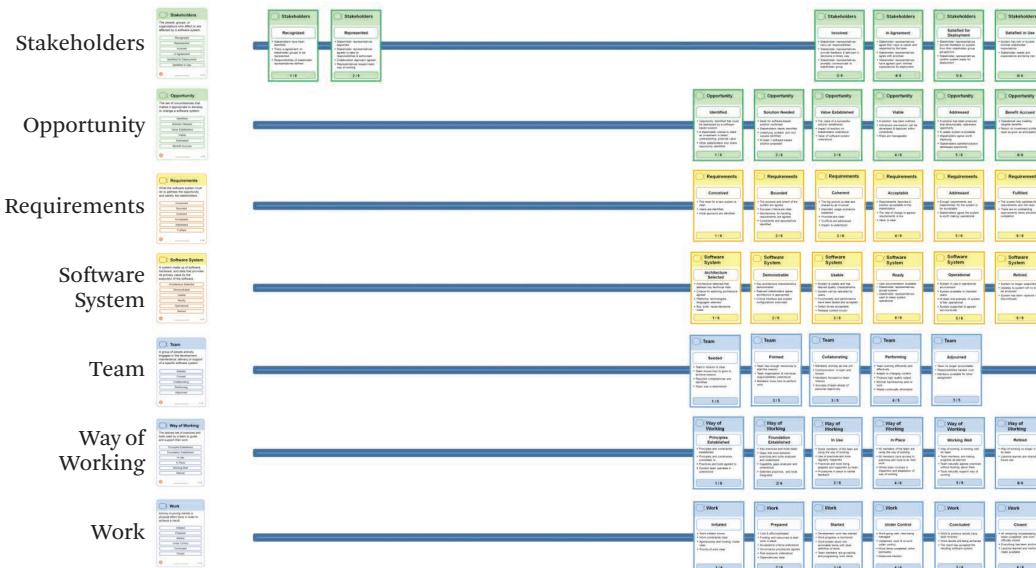


Figure 8.7 The current state of the Stakeholder alpha is agreed to be state 2.

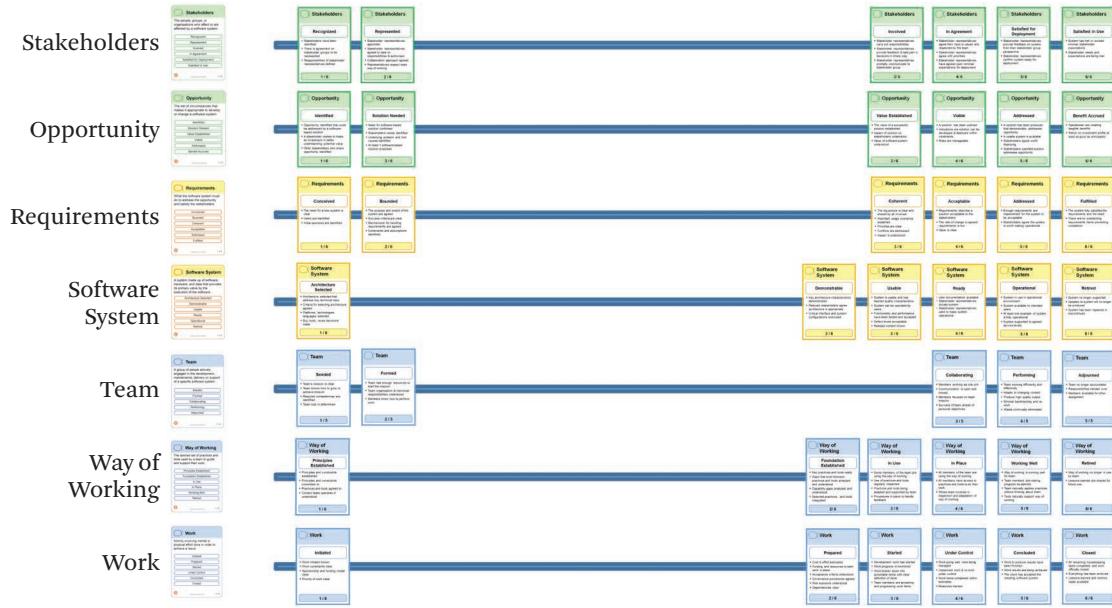


Figure 8.8 The current states for all alphas have been identified.

8.3 Objective Go

The Objective Go game is played to agree upon where you need to go next. To know where to go next, you of course need to know where you are. This game is played after you have assessed the current states of all the alphas. Thus, it is usually played after you have played the Chasing the State game. Let us therefore take the start position for the game as in Figure 8.8. In this position, the team asks the question “Which is the next step we should take to progress the endeavor?” or, in other words, “What are the next set of alpha states we should achieve?” The team may decide that their objective is to move to the next state for all seven alphas, or they may decide that the next step should be to focus on just one or a few of the alphas to progress to the their next states.

An experienced team would not deal with one alpha at a time but instead take a holistic view and agree on which alphas to progress next. It would be a mistake to think that alphas progress independently of each other. In fact, we have learned through experience that the alphas often progress in “waves” that cross multiple alphas. A simple example is that the Requirements alpha cannot be progressed without also progressing the Stakeholders alpha; to achieve the Coherent state of Requirements you need to have Stakeholders: Involved; see Figure 8.9.

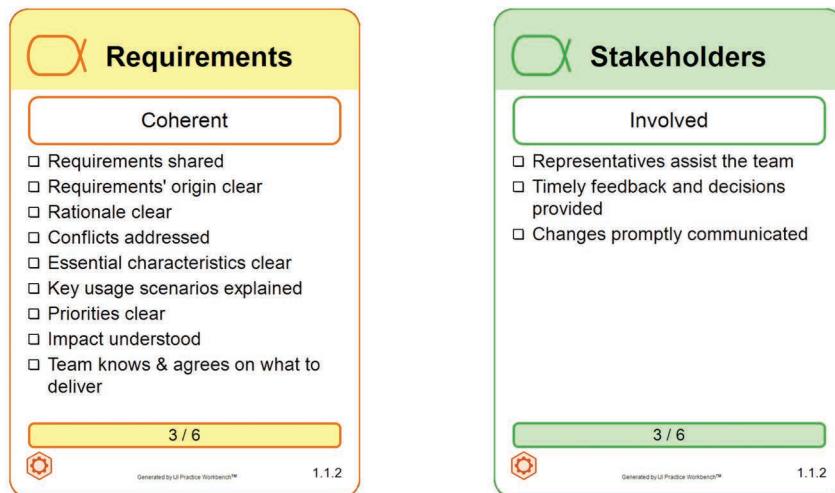


Figure 8.9 Requirements and Stakeholders Alpha Wave

Let us assume, for example, that a team agrees that one of their next objectives is to reach the Software System: Demonstrable state. This means the team agrees that there are checklist item(s) in the Demonstrable state that have not been achieved; see Figure 8.10.

As an example, let's look at the checklist item, "Key architectural characteristics demonstrated," in Figure 8.10. The team may have differing opinions on the interpretation of this state, and they may ultimately agree that they have not achieved this checklist item. Suppose this is the case and the reason is that the team feels they have not demonstrated a key performance requirement to a key stakeholder. The team will then discuss and agree on what to do next: in this example, conduct a demonstration for that key stakeholder; see Figure 8.11.

In a similar way, the team looks at each alpha deemed interesting to progress in the next step. For each alpha, they discuss the next state that should be achieved and which checklist items for that state are not yet achieved. Once they have agreed upon where they want to go next, they also discuss what tasks they need to do to get there. For each alpha the team has agreed to progress to a higher state, its corresponding state card is moved to the middle of the table, as in Figure 8.12.

In our example, the team agrees that the next step's are to move to Stakeholders: Involved, Software System: Demonstrable; Way of Working: Foundation Established; and Work: Prepared.

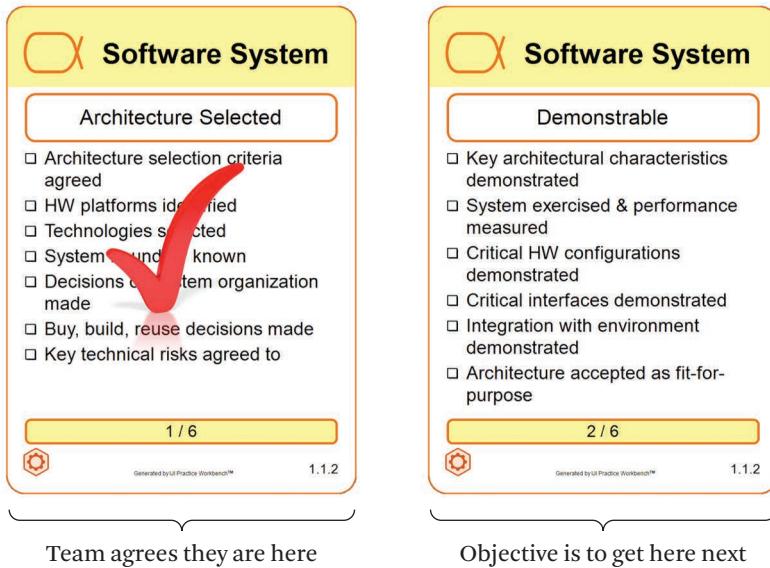


Figure 8.10 The Software System alpha—where the team is and where they need to go next.

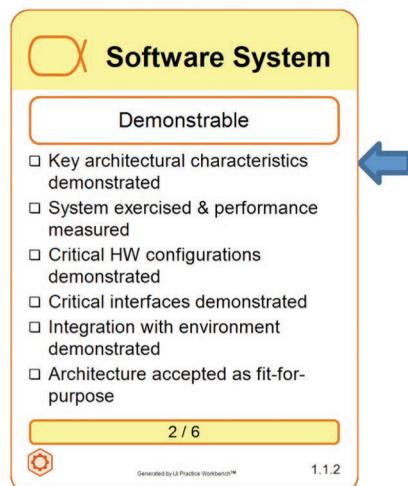


Figure 8.11 Team agrees that demonstration to a key stakeholder is needed.



Figure 8.12 The next step is represented by the cards in the middle of the table.

8.4 Checkpoint Construction

Usually, organizations have defined lifecycles that consist of phases separated by checkpoints. Checkpoints are intentionally independent of the practices a team agrees to use, because one of their main purposes is to assess the endeavor from different viewpoints such as value, funding, and readiness. In this sense, checkpoints can be viewed as critical points in the lifecycle of an endeavor where the definition of “done” for the phases needs to be specified. At each checkpoint, a decision is made whether to proceed to the next phase or not. A checkpoint can be defined using alpha states, as we have shown in Section 6.5.2.

Since an endeavor can have many teams working in parallel, to synchronize between the teams, they usually all need to have the same checkpoints. Thus, the checkpoints for an endeavor are normally specified by the stakeholders of the whole endeavor and not by every team participating in the endeavor. Therefore, this game is played by the stakeholder team, or a few of the key stakeholders. In this section,

the team to which we refer is the stakeholder team—a few key stakeholder members that can represent the views of the stakeholders.

Checkpoint Construction is played to gain consensus on the checkpoints in an endeavor's lifecycle. To illustrate the use of this technique, we will use the same simple lifecycle example as in Section 6.5.2. The lifecycle of the development endeavor comprises three phases: pre-development, development, and post-development.

Let us now play the Checkpoint Construction game. Our players, the stakeholder team members, gather around the table. The game is played for one checkpoint and in two rounds. One team member accepts the role of facilitator and lays out the seven Alpha Overview cards on the table. The facilitator next describes the checkpoint being considered. Let us assume we are going to specify the Ready for Development checkpoint.

In the first round, each team member considers each of the seven alphas and decides which ones should be considered as part of the checkpoint. They each jot down their choices. Then the facilitator for each Alpha Overview card asks the team whether that alpha should be considered in the checkpoint. Each player responds to that question using a thumbs up/thumbs down. Thus, in this round the team just agrees on which alphas should be considered for the checkpoint. Let us refer back to the TravelEssence example where we assume that after going through all seven alphas, the team agrees all alphas should be considered except for the Opportunity alpha, which Angela was handling (see Figure 8.13).

Now, the second round is played. The facilitator lays out all of the alpha state cards horizontally across the table for all of the selected alphas to be considered for the checkpoint. Each player considers the set of states for each alpha and, without informing the other players, he/she identifies the state he/she believes the alpha needs to be in to pass the checkpoint. When everyone is ready, each of the players simultaneously raise a hand with the number of fingers indicating the state he/she believes the alpha needs to be in to pass the checkpoint. A closed fist is used to indicate the sixth state. If all players have selected the same state, there is consensus. If not, the players with the least and most advanced states explain their reasoning. After discussion, the players again simultaneously raise their hands, indicating the states they have selected. This step is continued until consensus is reached.

Once the state is agreed on, the facilitator leads the group through a discussion of potential additional checklist items to be added for this checkpoint. In this way, the generic checklist items on the cards can be tailored to the context of the specific endeavor.

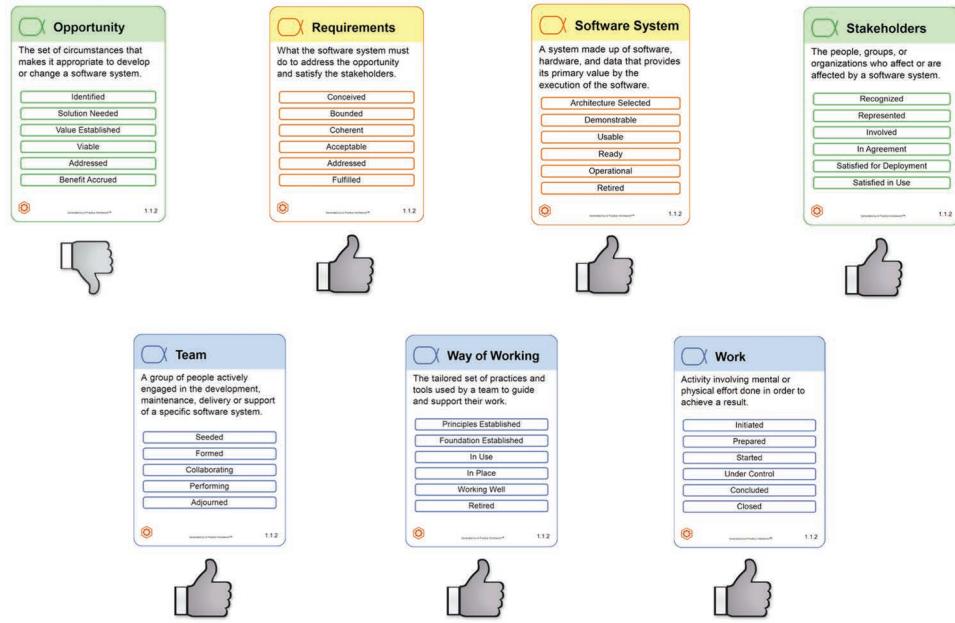


Figure 8.13 Results of team playing first round of Checkpoint Construction game.

We have now showed how Essence can be utilized to help define a checkpoint. By applying the Checkpoint Construction game several times, a whole lifecycle (such as the one in Section 6.5.2) can be defined.

8.5

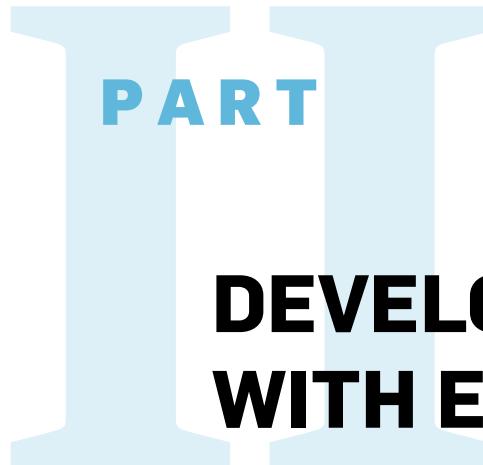
Reflection

What we as authors have found is that serious games using Essence can provide effective facilitation techniques. Ideas are never absent when knowledgeable workers come together. Cards provide a good avenue to bring these ideas to reality very quickly. They engage all members of the team, not just the most vocal or the most experienced or competent. The fact that the states and their checkpoints are not unambiguous is not just negative; it results in engaging discussions, which help the team think about issues that might not occur to them from just their own personal experiences. They need to agree what those issues mean to their endeavor. Ultimately, this helps the team address issues and risks early, before they become major problems. This not only helps keep the endeavor on a healthy course, but also helps team members learn to collaborate effectively, as well as bringing the team together.

What Should You Now Be Able to Accomplish?

After studying this chapter, you should be able to

- name the elements of the information on Essence cards and explain their significance;
- explain how the cards can be used to gain a shared understanding within a team;
- name the key principles behind the card games, and explain their key characteristics and role in the process (e.g., the focus on cooperation rather than competition);
- explain the four introduced card games (Progress Poker, Chasing the State, Objective Go, and Checkpoint Construction), together with the issues they help to solve and examples of their usage; and
- explain the key concepts of the software engineering process important for the card games, such as the state change, checkpoints, and typical lifecycle phases.



PART

DEVELOPING SOFTWARE WITH ESSENCE

The goal of this part of the book is to demonstrate how Essence can help a team run a relatively simple software development endeavor under these circumstances:

- The requirements and consequently the related software system are relatively simple.
- The team is small (only a few members).
- The team members have worked together before, know one another, and know how to work together, so there is not much need for written guidelines.

Although simple, the foundation behind all software development endeavors is the same. This sample endeavor does not have the clutter and noise often found in many more complex endeavors, and therefore the team can focus upon the essence of software engineering.

In such simple situations, teams can rely upon tacit knowledge and little explicit knowledge. (Tacit and explicit knowledge were explained at some length in Chapter 7.) The Essence kernel provides the bare essentials to get teams started and make progress, which you will learn about in this part of the book.

Part II will cover the following objectives:

- How to use Essence to describe some reusable “mini-practices” called games.
- How to kick-start software development using Essence only.

- How to plan the work, do the work, check the work, and adapt the way the team works.
- How to visualize progress and health, and detect anomalies.
- How to appreciate the need to make practices explicit and modular when facing more complex situations.



Kick-Starting Development Using Essence

What we have presented thus far (in Part I) provides an overview of how to productively utilize the Essence kernel. As we progress in this part of the book, we will provide greater detail on how the Essence kernel helped Smith and his team with the development of their recommendation engine.

In this chapter specifically, through the story of TravelEssence, we show

- the role the Essence kernel can play in helping a team, during a real development endeavor, to stay focused on the most important things;
- the importance of stakeholders' involvement, clarify of requirements, and the scope of the software system;
- the notion and role of opportunity;
- the work breakdown in relation to the agreed way of working;
- the importance of integrating the work of all members of the team;
- the steps to set up Essence for real usage;
- the application of card games step-by-step, including the involvement of discussion, argumentation, and voting; and
- the concept of combining games into a well-aligned flow during the development process.

In our story, Smith used the Essence framework to help his team ask the right questions and get pointed in the right direction. To get started, his team needed to know where they were and what they needed to head toward.

The Essence kernel, together with some of the games we saw in Part I, provides the tools to do just that.

Getting started with Essence involves the following steps:

1. understanding the context through the lens of Essence;
2. agreeing on the development scope and checkpoints, including where the endeavor begins and ends; and
3. agreeing on the most important things to watch.

9.1

Understand the Context Through the Lens of Essence

Software development is really a problem-solving endeavor. Problems can exist in any dimension of software engineering.

- First of all, software engineering is a result of recognizing some problem or an opportunity. As an example, when Apple invented the iPhone Steve Jobs saw an opportunity to improve the way people communicated.
- There can be many sources of problems. Some of them may be related to the software system itself that is being built or has been built; for example, if the software system already exists, you may not fully understand the original requirements and/or the solution that led to the existing software system.
- Perhaps you have problems related to getting stakeholders involved; for example, you may need to talk to users of the software system to understand their problems better, but they may not have time to talk to you.
- Perhaps you have problems related to getting your team to communicate; for example, a less experienced developer may not be getting the guidance needed from a more experienced developer who is very busy.

Like all problem solving, everything begins with understanding the problem, which may include multiple related problems, each found in the different dimensions of software engineering. Examples of these dimensions include stakeholders, requirements, the software system, and the team. You have to understand the requirements for the software system, but you also have to understand the needs of the stakeholders. Essence helps you interpret your context by showing where your problems are in each of these dimensions of software engineering. For example, the alphas help lead us to ask questions about the development endeavor, and they help us collect useful information pertaining to each alpha.

We consider the needs of the development team that has been assigned to an endeavor within the company TravelEssence. As you recall, TravelEssence is a



Figure 9.1 Understand context with Essence.

leading travel service provider that targets both leisure and business travelers, as discussed in Part I.

Smith and his very small team came together and started capturing what they knew about the endeavor using some sticky notes and the Essence alphas; the result can be seen in Figure 9.1. (The text on the notes in the figure is only a sample, intended to help you get a sense of the kind of information associated with each alpha. Some of the information in the next sections has been used in previous chapters, but here it is grouped all together to form a complete picture of the endeavor Smith and his team were assigned.)

From the Perspective of the Customer Area of Concern

Stakeholders. The stakeholders in this endeavor included Angela and Dave from the Digital Transformation Group within the company. This group was tasked with employing digital transformation to expand the company's business. Digital transformation is the use of technology to radically improve a company's performance through changes in business models and improvements to customer experience (in this example, adding features to a mobile plug-in to help frequent travelers). In particular, Angela was leading this digital transformation effort in collaboration with Smith and his team. Dave was the Chief Digital Officer (CDO) in the company and Angela's boss.

Opportunity. Frequent travelers were already logging information about their trips and sharing their experiences to social media sites, such as Facebook and Instagram. Thus, TravelEssence already had a significant amount of

data about travelers. This situation created an opportunity for TravelEssence to generate more business by using traveler data from repeat customers to attract new customers.

From the Perspective of the Solution Area of Concern

Requirements. The specific requirements for the new endeavor included analysis of traveler data to identify trends and relationships and to recommend more exciting travel options to travelers. A measure of success would be an increase in customers accessing these options (e.g., clicks on the options by potential customers looking for more information).

Software System. TravelEssence already had a mobile application (app) that was cloud-based, which means their customers did not need to download and install any software on their own digital devices to access the service. Therefore, Smith and his team only needed to develop a simple plug-in (i.e., addition to the software) that would allow customers to view the recommendations on the already existing TravelEssence mobile app. Figure 9.2 shows the additions needed to the existing mobile app and to the existing cloud service to provide the recommendation enhancements. Some of the team members would also need to learn how to develop software utilizing a software architectural style that uses small independent processes to communicate. This architectural approach is referred to as microservices. (You will learn in Part III how to express practices using the Essence language and, specifically, you will learn one way to describe microservices as a practice.)

From the Perspective of the Endeavor Area of Concern

Work. Smith and his team were asked by management to deliver a working demo of the new product in one month.

Team. Smith's development team comprised himself and three other developers, namely Tom, Joel, and Grace, all of whom were familiar with mobile app development. Tom and Joel had used microservices before, but this technology was new to Grace.

Way of Working. Smith and his team would use the facilities of the Essence kernel as a convenient way to evaluate their endeavor's progress and health. The practices in their way of working were not explicitly described. They would use the alphas, states, and checklists of the kernel to help them figure out where their problems were and where they would need to focus

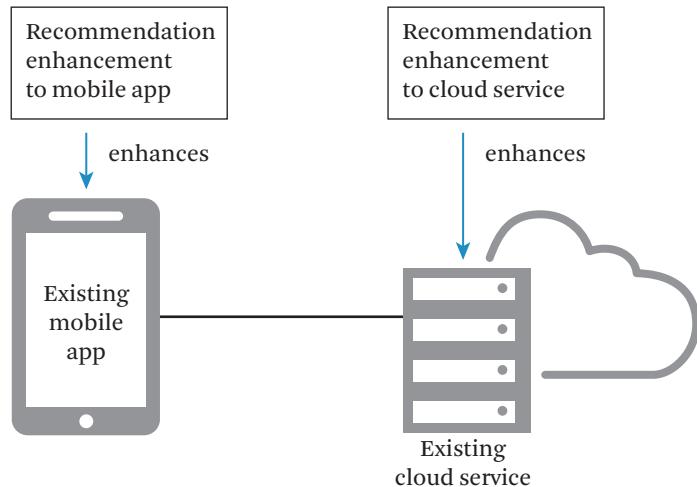


Figure 9.2 Enhancement to the Software System to achieve recommendations.

their attention as their endeavor evolved. They would also use the kernel to help them decide about the activities they needed to conduct to help solve the problems they identified. The team referred to this as “vanilla” Essence because it included no explicit practices as extensions to the kernel. They just used the alphas, states, and checklist items to help their team initiate discussions leading to the actions agreed upon for each of the team members.

The team had initial experience using the games as described earlier. This helped to gain an overview of the requirements, endeavor, etc. Their plan was to learn more about Essence and the games, using common sense and experience to guide them. In this way they would evolve their understanding of the different ways to use both the games and the kernel alphas. Below we will describe a few examples of what resulted from this approach. For example, we will describe how the team decided to add some additional alphas, referred to as “subordinate” alphas or sub-alphas for brevity. As another example, we will also describe how the team applied a number of the games previously discussed to help them run their endeavor and conduct progress assessments. Later, in Part III you will learn a different way to use the kernel when we discuss how the team extended it beyond the games and the sub-alphas when the scope of their endeavor became more complex.

9.2

Agreeing on the Development Scope and Checkpoints

The states of the Essence kernel alphas, together with each state's checklists, can provide a straightforward way for a team to gain agreement about preconditions for starting development and on the criteria for completing development. One way to help you understand how the kernel can do this is to review the Checkpoint Construction game discussed in Chapter 8.

Smith started the game by laying out on a table the state cards for all seven alphas. Smith then said to his team, "We need to define two key checkpoints, which we will name 'Ready for Development' and 'Development is Complete'." (See Figure 9.3.)

Smith continued, saying, "Let's start by selecting the alphas that need to be inspected at our 'Ready for Development' checkpoint. Then we need to conduct a second round where we agree on the states of each of these alphas that need to be reached to say we have achieved this checkpoint. This checkpoint must be reached before we can formally start the development." (See Figure 9.4.)

After all the team members had considered all the alphas and noted to themselves which alphas they thought should be included, Smith said, "OK, let's start with the Requirements alpha. Using a thumbs-up, or thumbs-down, who thinks this alpha should be included in our 'Ready for Development' checkpoint?" After the team members all voted and reached agreement, Smith proceeded to lead the team through the voting for the other six alphas. As you can see in Figure 9.4, the team members agreed that all seven alphas should be included in their 'Ready for Development' checkpoint.

Smith then led the team through the second round, where each team member voted on their choice for which state of each alpha needed to be achieved to meet their 'Ready for Development' checkpoint. During the second round, Grace disagreed with her teammates, who all felt the Work alpha only needed to achieve the Initiated state. Grace said, "I cannot start my tasks until the funding is approved, and that checklist is in the Work alpha Prepared state." Tom quickly replied, "We

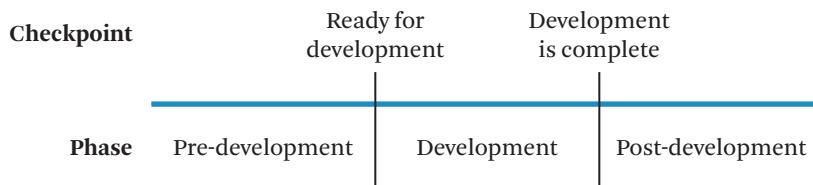


Figure 9.3 Checkpoints and phases for enhancement of TravelEssence.

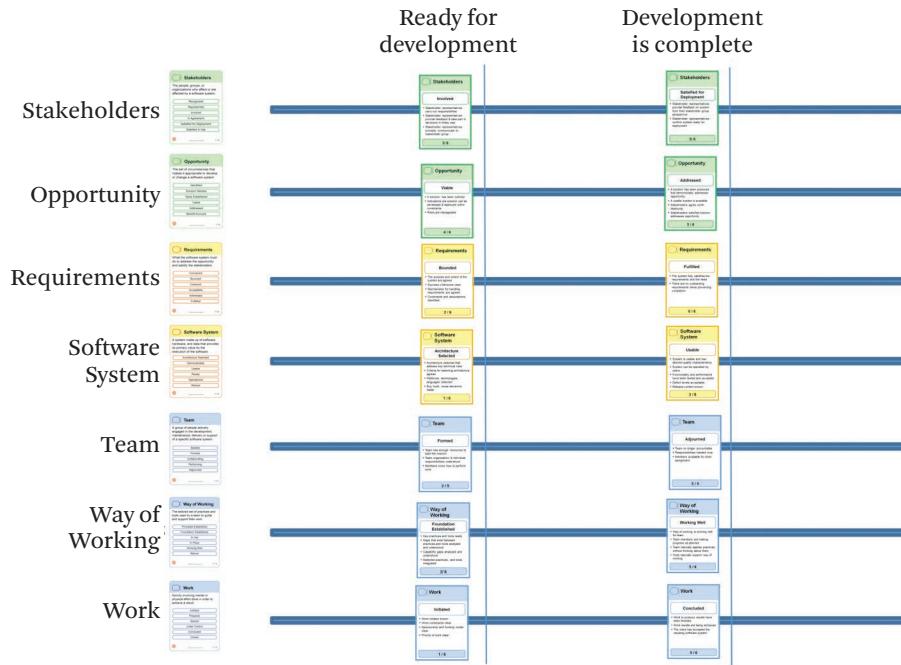


Figure 9.4 Defining the two key checkpoints using alpha state cards.

can't wait for formal management funding approval because the schedule is too aggressive and our organizational approval process is way too slow." Smith then responded, "Yes, Tom, you are right, but Grace has a very important point. We can't work without funding approval. So I will go and talk to Grace's manager and explain the situation. I am sure we can get approval to start while we wait for formal funding approval." Everyone agreed with Smith's approach to solve this dilemma.

As shown in the figure, the team agreed that they needed to get the key Stakeholders Involved, and they needed to establish the value of the Opportunity and show that it was Viable. They also needed to agree that the Requirements were Bounded and that Software System had reached Architecture Selected so that the key technical risks had been addressed. They then agreed that the Work had to be Initiated, the Team Formed, and they had to agree that their Way of Working had reached the Foundation Established state. After the states were covered, the team discussed the checklists associated with each state and reached agreement on any additional checklist items they thought should be included. Once this discussion concluded, Smith next led the team through a similar process to define the 'Development is Complete' checkpoint (see Figure 9.4).

Sidebar 9.1 Sub-Alphas

When using the kernel, it is unlikely that you will progress the alphas as a single unit. In each case you will drive the progress of the alpha by progressing smaller parts of the alpha. For example, the Requirements will be progressed by progressing individual requirement items. Requirement Item is an example of what we refer to as a sub-alpha to Requirements. Sub-alphas are alphas in their own right that help to move forward or slow the progress of the kernel alphas. As an example of slowing progress, Defect could be a sub-alpha of the Software System alpha that slows the progress of the Software System kernel alpha.

In contrast, Requirement Item is a sub-alpha that helps to move forward the progress of the kernel Requirements alpha. The Requirement Item sub-alpha has states with checklists just like the kernel alphas that can help practitioners when assessing the state of the sub-alpha. Now you may be wondering whether or not these sub-alphas, such as Requirement Item or Defect, are part of the kernel. The answer is that they are not part of the Essence kernel because they are not always needed—they are not essential. Depending on your specific endeavor and the practices your team agrees to use, sub-alphas may or may not be needed. You will learn more about practices in Part III where you will also see more examples of sub-alphas that are important to monitor once you have chosen to use certain explicit practices.

9.3

Agreeing on the Most Important Things to Watch

The team agreed that watching just requirements was too coarse for their endeavor because it would not be able to show them progress on a day-to-day basis. Often, the Essence kernel alphas need to be broken down into smaller items to measure progress.

Tom said, “In order to accurately measure our progress with requirements, we need more than just the Requirements alpha.”

Angela, Smith, and the team therefore agreed that they would track requirement items, defects, and issues that would occur during development. The team’s work at this level could be reported each day. They agreed to use a simple spreadsheet to track progress for these items.

After the team agreed that they had reached their ‘Ready for Development’ checkpoint, they decided to play the Chase the State game and the Objective Go game (see Sections 8.2 and 8.3) to determine where they were and what they should focus on next. Smith handed each team member a set of Alpha State cards. Smith then placed the Requirements Alpha Overview card in the center of the table. Each team member thought about which state they believed they were in and then placed that state card face down on the table. When everyone was ready, they turned their

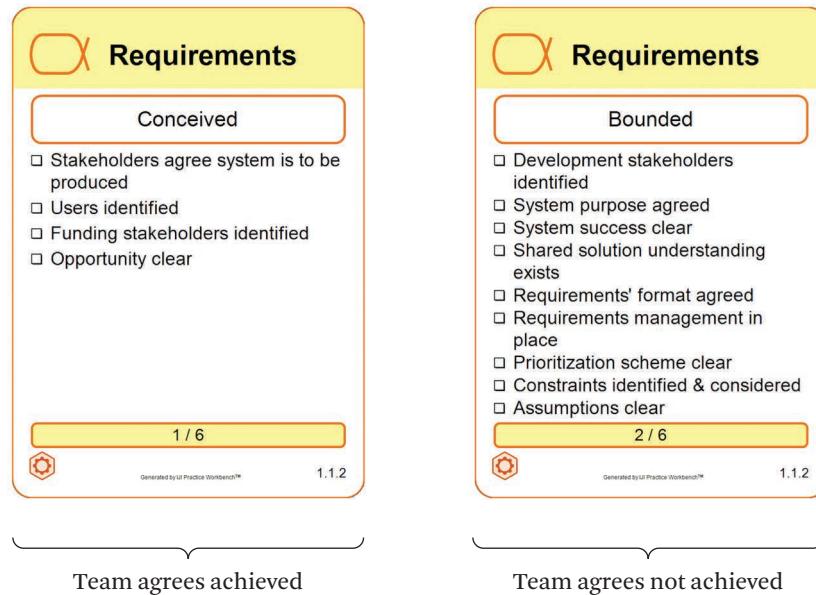


Figure 9.5 Requirements: Conceived and Bounded state cards.

cards face up. As a result of playing this game, the team agreed that they had already achieved the Requirements: Conceived state because they knew who their stakeholders were, and they knew which stakeholders would fund the new system. However, they had not yet achieved the Requirements: Bounded state. This was because the team agreed they had not yet achieved a clear understanding of what success meant for the new system. See Figure 9.5. (Note: In the interest of brevity we are not discussing every checklist item in these states.)

They also knew from playing the Checkpoint Construction game (Section 8.4) earlier that they needed to get to Requirements: Bounded and to Way of Working: Foundation Established states to get to their agreed-to 'Ready for Development' checkpoint. The team discussed the way they planned to work, and agreed that they would work iteratively, which meant they would provide frequent software deliveries to the customer. This discussion led to their agreement that they had reached the Way of Working: Foundation Established state. To achieve the Requirements: Bounded state, Angela, Smith, and the team sketched out the requirement items that would be part of their first-month delivery (see Figure 9.6).

Having agreed on the Requirement Item list in Figure 9.6, the team agreed that they had reached Requirements: Bounded. As can be seen in Figure 9.4, the team



Figure 9.6 Requirement item list for the enhancement of TravelExchange.

also needed to agree on the work that needed to be done to achieve the other five kernel alpha states for that checkpoint—Stakeholders: Involved; Opportunity: Viable; Software System: Architecture Selected; Work: Initiated; Team: Formed; and Way of Working: Foundation Established. This is an example of what we mean by describing software engineering as multidimensional, and what we mean by stating that Essence helps the team stay focused on the most important things throughout their endeavor. In Chapter 10 we discuss how the team reached agreement on additional needed work.

What Should You Now Be Able to Accomplish?

After studying this chapter, you should be able to

- explain the steps involved when getting started with Essence (i.e., understanding the context, agreeing on scope and checkpoints, and agreeing on the things to watch);
- explain how to use Essence’s areas of concern to understand the context of the problem or opportunity an endeavor addresses;
- for a sample endeavor, name the related stakeholders, the opportunity, requirements, software system, work, team, way of working; and
- explain the concept of sub-alphas and when they are useful, including examples of sub-alphas.

Developing with Essence

The goal of this chapter is to demonstrate how Essence can help teams find the best approach to development endeavors via selection of appropriate practices, mini-practices, and tools. Specifically, we show

- situations within software endeavor that require the team to be ready to resolve minor challenges during software development, because not everything runs smoothly in real endeavors;
- the importance of communication and teamwork during software engineering; and
- that “essentializing software engineering” means representing the way your team is working using the Essence language and the Essence kernel common ground.

Furthermore, using the example of iterative development performed by Smith and his team, we will also show

- what typically happens during parts of the development cycle that include planning, doing, checking, and adapting;
- the role of Progress Poker, Chase the State, and Objective Go when used in development planning; and
- the mechanisms to update the cards (by hand) when needed during the process.

To achieve the Way of Working: Foundation Established state, Smith’s team needed to agree on their approach to development, which included selecting their key practices or mini-practices and tools. They agreed to work in an agile way, splitting up the whole endeavor into smaller mini-endeavors, each mini-endeavor resulting in progressing the work on the software to be built. This approach is called iterative development and each mini-endeavor is called an *iteration*. An iteration is a fixed period of time in which the team develops a stable piece of a software system.

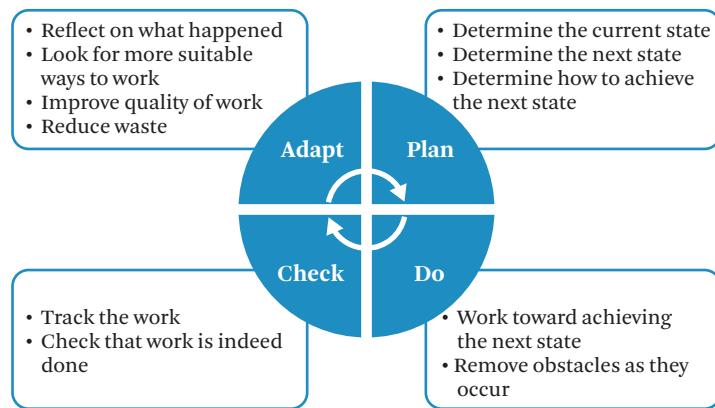


Figure 10.1 Plan-Do-Check-Adapt cycle.

The length of an iteration is typically two to four weeks and can involve all kinds of activities, including requirements gathering and the deployment of the resultant software system.

Developing your software iteratively is like taking a journey in your car. You need to know where you are, where you are heading, how much fuel you have, and how much further you have to go before reaching your destination. You adapt to the road conditions, traffic, and weather as you drive. You are continuously planning, doing, checking, and adapting¹ (see Figure 10.1). This is how Smith and his team ran their iterations. Each iteration was set to be one week in duration.

Plan. First, Smith's team would ascertain the current state of the whole endeavor by determining the current state of each of the alphas. Before their first weekly iteration, they decided to start by playing the Progress Poker game (Section 8.1). But after having done it for the first two alphas they continued to play the Chase the State (Section 8.2) game for the other five alphas. So Smith began by placing the first Alpha Overview card in his deck in the middle of the table. Each team member thought about which state they believed that the Requirements alpha was in and then placed the selected state card face down on the table. When everyone was ready, they turned the cards face up. After the team discussed the results and reached agreement on the state of the Requirements alpha, the team members

1. We modified Deming's PDCA cycle (<https://en.wikipedia.org/wiki/PDCA>), replacing Act with Adapt, as this is more descriptive of the intent.

picked up their cards. Together they had agreed that the Requirements had reached the Coherent state.

Smith then selected the Software System alpha and placed its Alpha Overview card in the middle of the table. He and his team repeated the same steps with this alpha and agreed it had reached the Architecture Selected state.

It turned out that the team wasn't needing to discuss too much to obtain consensus; they found that they were in agreement immediately. So Smith suggested instead to play the Chase the State game (Section 8.2) next. This open discussion would more quickly reveal the state for each of the remaining five alphas.

For the Stakeholder alpha, although the team agreed that they knew Dave and Angela were their stakeholders, they still had not yet agreed on how they would get them involved. Thus, the Stakeholder alpha continued to be in Represented state. For the Opportunity alpha, the team agreed that they had now achieved the Solution Needed state, but no one thought they had achieved the Value Established state. At this point, Joel and Tom started to discuss how they might go about convincing senior management of the value of the endeavor, but Smith quickly interrupted, saying, "Wait. We are just conducting the game now to agree where we are. Let's hold off discussing which states we need to focus on next and how we will achieve them. We will do that after we have fully assessed where we are now and can then decide what is most important to do next." Everyone agreed and they continued with Chasing the State to reach agreement on the other alphas (Work: Prepared, Team: Formed, and Way of Working: Foundation Established); see Figure 10.2.

After finishing the games, knowing where they were, they discussed and agreed to what alphas and states to progress in the coming iteration. This was done by the team playing the Objective Go game (see Section 8.3), Smith decided to start the Objective Go game saying, "We now know what state we are in for each of the seven kernel alphas. Now we need to agree on which states to focus on achieving in our next iteration." Joel decided to cut to the chase and asked, "So does everyone agree that the most important thing in the next iteration is to convince management of the value of our endeavor?" All of the team members nodded in agreement. Smith then said, "This means we have concurred that we need to achieve the Opportunity: Value Established state in the next iteration."

The team continued to discuss, and reached agreement on which states in the other alphas they needed to achieve in the next iteration. For one thing, they needed to get their key stakeholders, Dave and Angela, involved (meaning reaching the Stakeholder: Involved state). They also agreed that the requirements for the upcoming iteration were coherent, which means they were consistent, and that

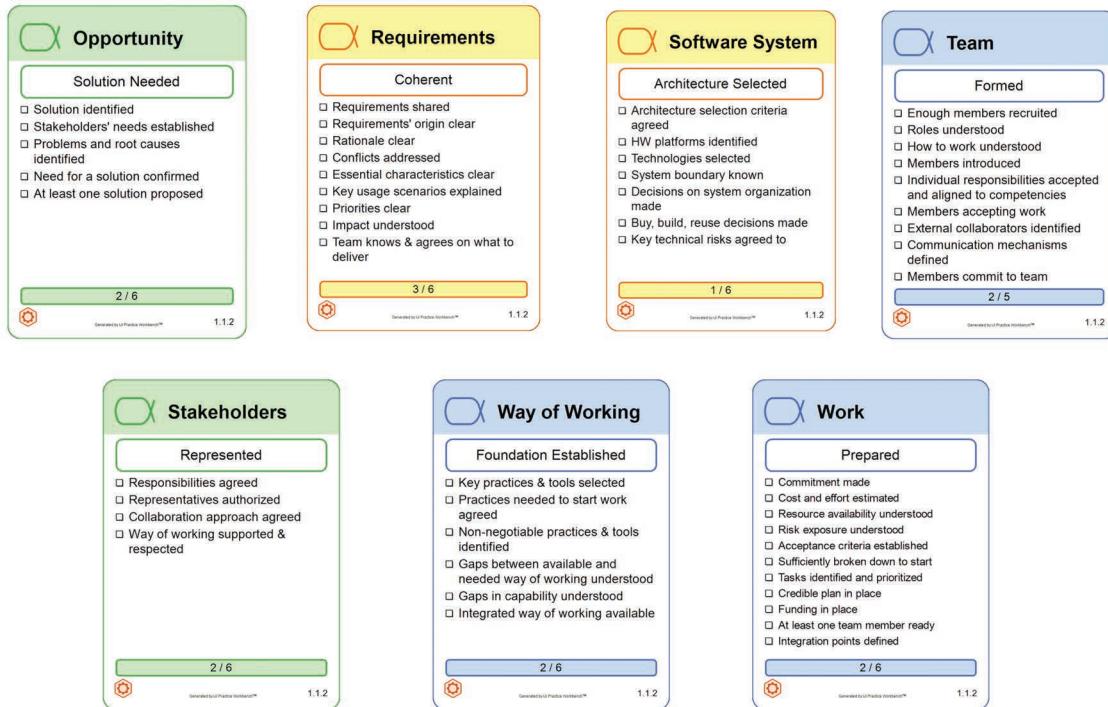


Figure 10.2 The alpha states agreed on after playing the Progress Poker and Chasing the State games.

they needed to address them by demonstrating the implementation of those requirements to Dave and Angela. (That is, they needed to achieve the Requirements: Addressed state.) They also agreed the team needed to be collaborating, the work must be started, and the way of working needed to be in use; see Figure 10.3.

Together they then planned how to achieve the target states by identifying which tasks, if completed, would achieve these states. For example, one specific part of the work they needed to do was to set up a meeting with Dave and Angela to discuss how they would get them involved. They also knew they needed to set up a test environment.

This all enabled them to connect their detailed day-to-day work with the progress of the endeavor as a whole. If the effort to complete the tasks exceeded that available in the iteration, then it would take more than a single iteration to achieve the objectives and the target states. This means the team would need to break their tasks down further and agree on the pieces to complete in the current iteration. For instance, they knew they could not get all four requirement items done in the first

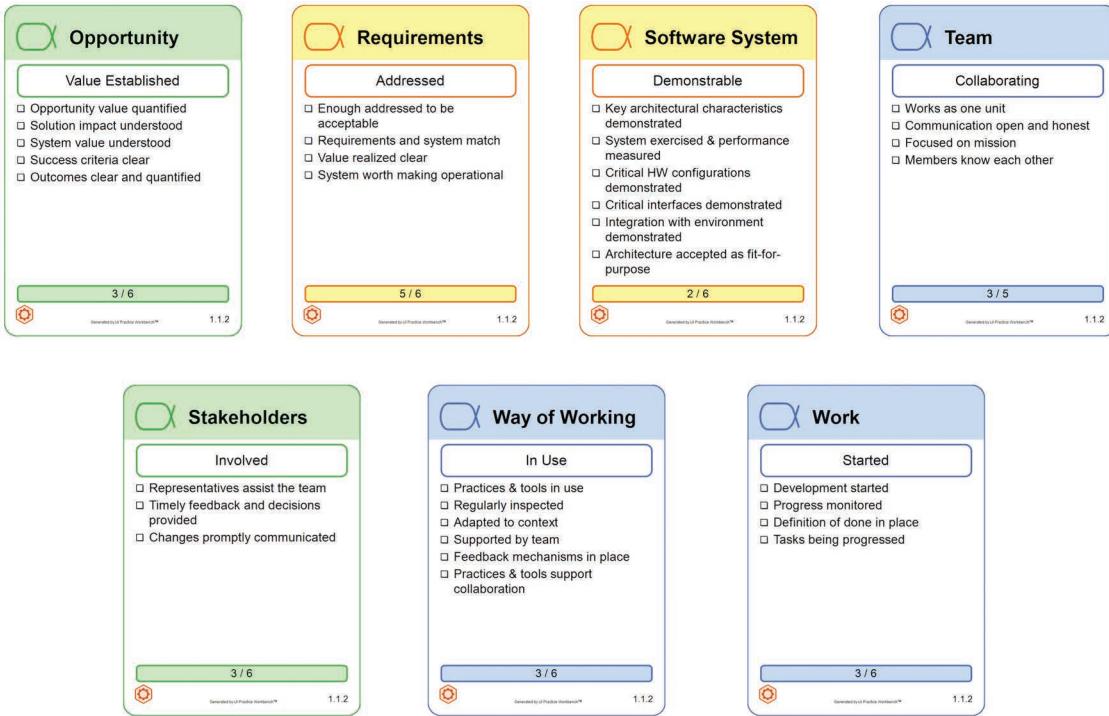


Figure 10.3 Result after applying the Objective Go game.

iteration, so they concurred to work on just the first three (for the four requirements items mentioned here see Figure 9.6 above).

Do. Each iteration, Smith's team worked on the identified tasks to progress the endeavor as a whole toward the target states. This involved setting up environments, discussing requirements, capturing agreements, writing code, testing, and so on; see Figure 10.4.

Check. Smith's team tracked the objectives and tasks to make sure that they were completing what they had planned as they used their agreed-on way of working. The team discussed the healthiness of all alphas; unhealthy meant the alpha had a checklist that had not yet been met but should have been met, or that the checklist had previously been met, but was no longer met due to some change in the condition of their endeavor. The team placed green stickers next to the alpha state cards they had agreed represented healthy alphas. They also agreed that anyone could

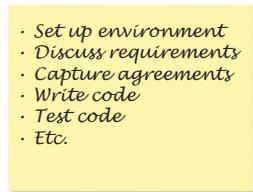


Figure 10.4 Task list.

place a red sticker next to a card if they felt during the iteration that an alpha had become unhealthy.

Adapt. Smith's team reviewed their way of working, identified obstacles, and found better or more suitable ways of doing things. This often resulted in changes to their plans and their way of working.

10.1

Planning with Essence

When you plan an iteration, the alphas can help you understand where you are and where to go next. By aligning the objectives for each iteration, Smith's team made sure that they progressed in a balanced and consistent way. The alphas helped by reminding them what was essential for success as the team decided what was most important to focus upon next.

As mentioned, Smith and his team agreed that they would work in cycles where each iteration was one week. It was the first day (Monday) of the first iteration week. Using Essence, they reviewed their current state and the states they had previously agreed that they wanted to achieve by the end of the first iteration. Based on that, they identified a set of tasks. In what follows, you will find task descriptions that were agreed on by the team after discussing each alpha, the state they had achieved, and the next target state(s). The first state on the left in each diagram is the current state, and the state(s) listed after the arrow is (are) the target state(s).

Sometimes we have more than one target state for a given alpha. This is because teams often work to achieve checklist items in more than one state at the same time. When a team works iteratively, they often are working to get some of the requirement items both coherent and addressed in the same iteration. For example, when our TravelEssence team was working on Req-Item #1, they needed to get the code to actually produce recommendations on hotels, and they needed to get clarification on how far from the traveler's current location they should search for possible hotels.

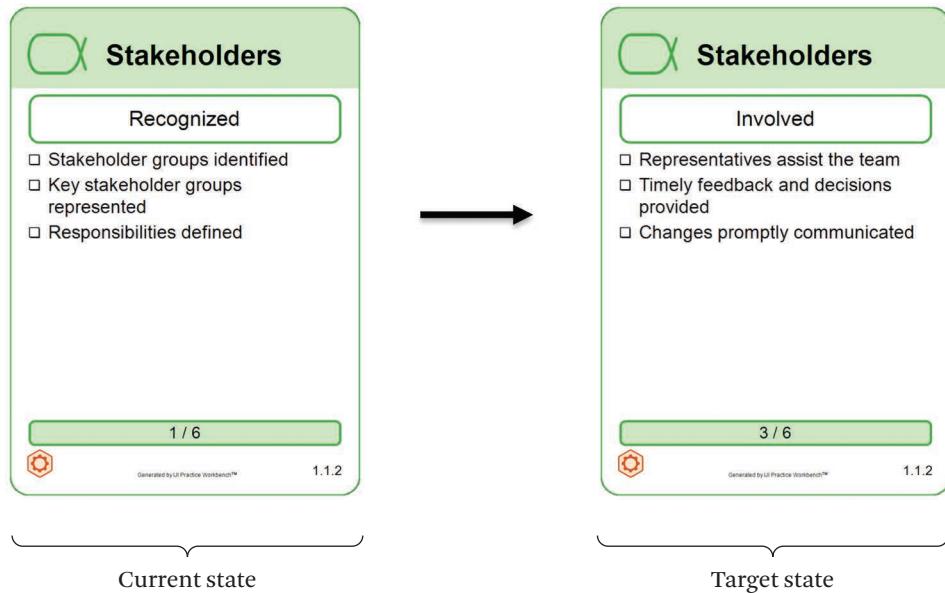


Figure 10.5 Stakeholders current and target states.

As we established in Chapter 6, the alphas fall into three areas of concern: Customer (indicated by green cards and notes), Solution (indicated by yellow cards and notes), and Endeavor (indicated by blue cards and notes). We will look at the planning in these terms; first we will consider it from the perspective of the Customer area of concern, comprising the Stakeholders and Opportunity alphas.

Stakeholders: Recognized → Involved (see Figure 10.5).

The team had identified two of their key stakeholders as Dave and Angela, and that there was not yet agreement on their involvement and commitment. As mentioned, they had agreed to set up a meeting with Dave and Angela to clarify their involvement.

(Task: Stakeholder involvement meeting)

Opportunity: Solution Needed → Value Established (see Figure 10.6).

A solution was needed to exploit the travelers' data, but what the solution required was still up in the air. The value of that solution still needed to be established to convince senior management at TravelEssence to move forward and fund the effort. The team's immediate priority was to set up a test environment where they could quickly experiment with different ideas for using the travelers' existing data



Figure 10.6 Opportunity current and target states.

to generate more traveler interest, leading to increased revenue. This could help them quantify the value of the new system to Dave and Angela.

(Task: Experiment with different ideas to increase business.)

Next, we look at planning from the perspective of the Solution area of concern, comprising the Requirements and Software System alphas.

Requirements: Conceived → Coherent, Addressed (see Figure 10.7).

The team determined two target states for this alpha: Coherent and Addressed. To achieve their objectives in the current iteration, they would need not only to get three requirement items into the Coherent state, but also move these requirement items to the Addressed state. In the discussion that follows, you will see how the team could work toward these two target states at the same time.

Smith already had a simple requirement list. In the first iteration, Smith and his team would attempt to complete the following requirement items:

Req-Item #1. System generates recommendations for a traveler

Req-Item #2. Mobile plug-in displays recommendations

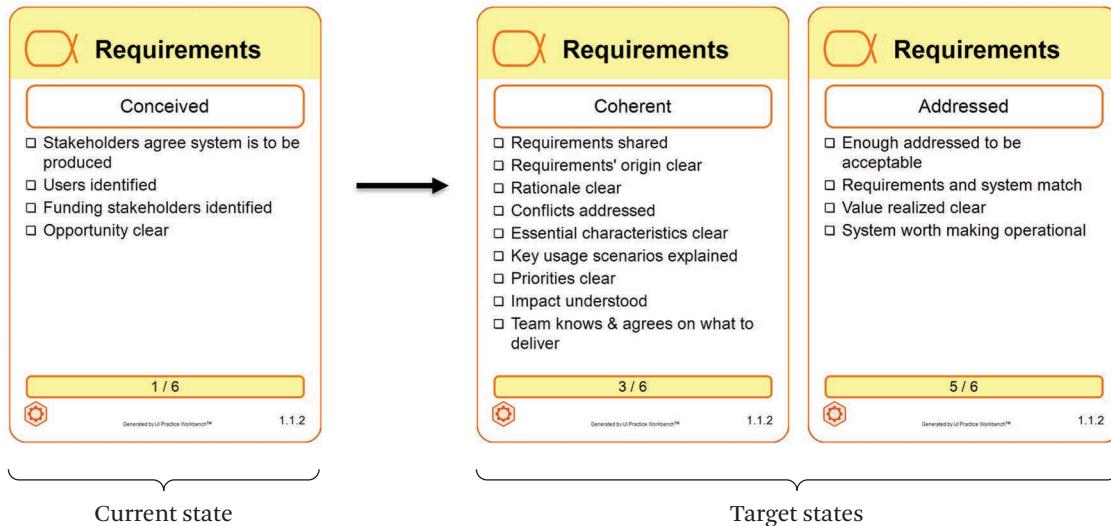


Figure 10.7 Requirements current and target states.

Req-Item #3. System handles user's selection to view or discard recommendations

Of course, some details needed clarification, such as finding the algorithm (calculation formula) to generate a recommendation, and determining which target set of travelers they would use as a test data set. This meant the next target state for requirements was to get to the Coherent state by working on these issues. The team agreed that this would be achieved by Smith collaborating with Angela to reach agreement on the plan. Once agreed, the team would need to move forward quickly to address these requirements for the upcoming planned demonstration (discussed below under Software System). This means they needed to move their requirement items to the Addressed state.

(Task: Smith to work with Angela to reach agreement on recommendation algorithm, and which set of travelers they would use as their test data set)

Software System: Architecture Selected → Demonstrable (see Figure 10.8).

To get to the Demonstrable state, the team would need to code, test, and integrate critical parts of the system and demonstrate the results to Angela. The team agreed to work on their respective requirement items and integrate their work by Wednesday evening to be ready for a demo to Angela by Friday.

(Task: Team members work on implementing their respective requirement items)

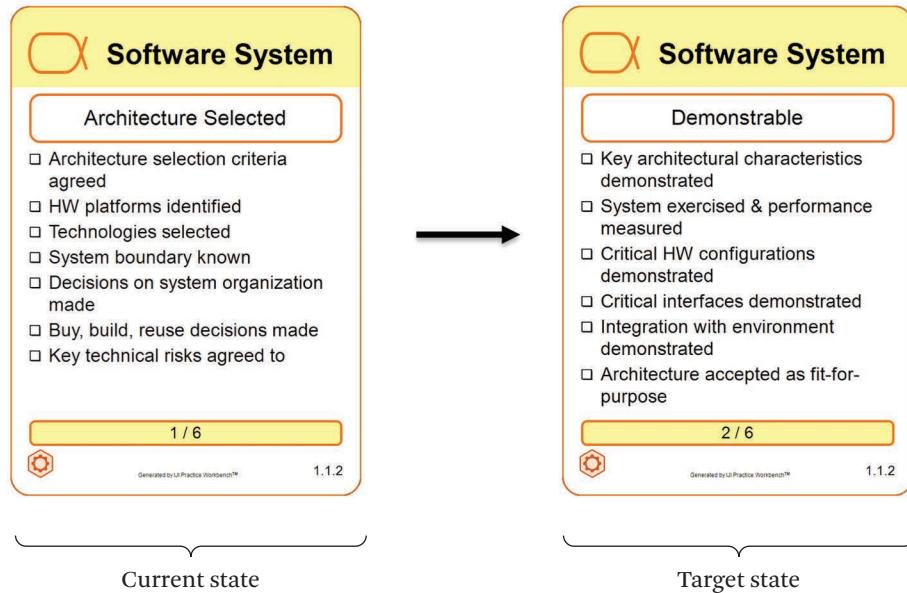


Figure 10.8 Software System current and target states.

Last, we look at this iteration's planning from the perspective of the Endeavor area of concern, comprising the Work, Team, and Way of Working alphas.

Work: Initiated → Prepared, Started (see Figure 10.9).

To get to the Work: Prepared state, the team needed to make sure their tasks were broken down into sufficiently small pieces to fit in the agreed iteration, understand any related risks, and be sure they had a credible plan in place that extended beyond the current iteration. While Tom, Joel, and Grace focused on the planning for the first iteration, Smith reviewed the work the team had agreed to do. As part of Req-Item #1, the team had discussed providing recommendations for both hotels and restaurants, but Smith decided this was too much for the first iteration and suggested the team limit the work for now to just providing hotel recommendations.

(Task: Team breaks work down to fit in iteration)

Team: Formed → Collaborating (see Figure 10.10).

Smith's team members had successfully worked together before. They each knew their responsibilities and how they would work together, but the team had not yet showed that it was working as one cohesive unit. By setting the goal of integrating

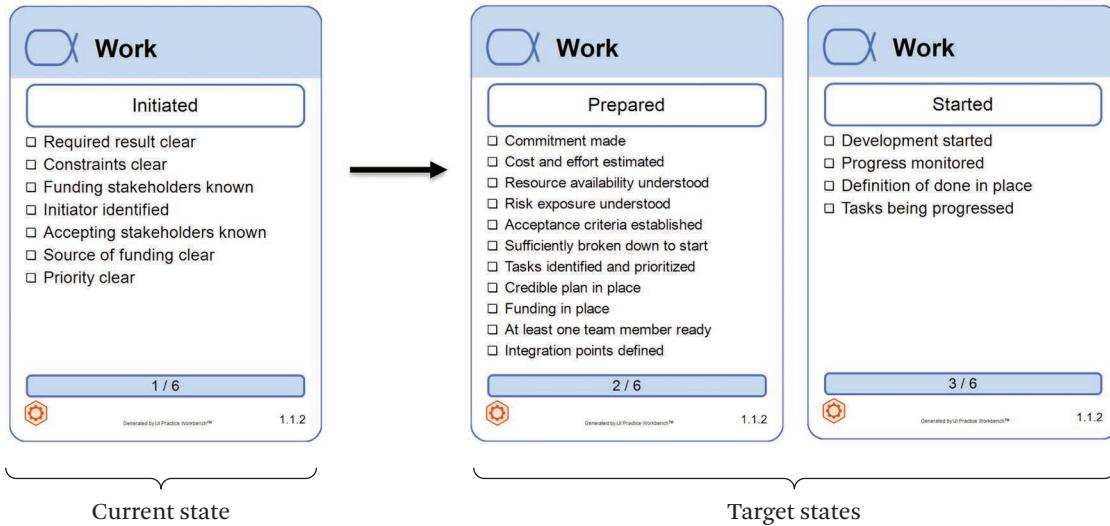


Figure 10.9 Work current and target states.

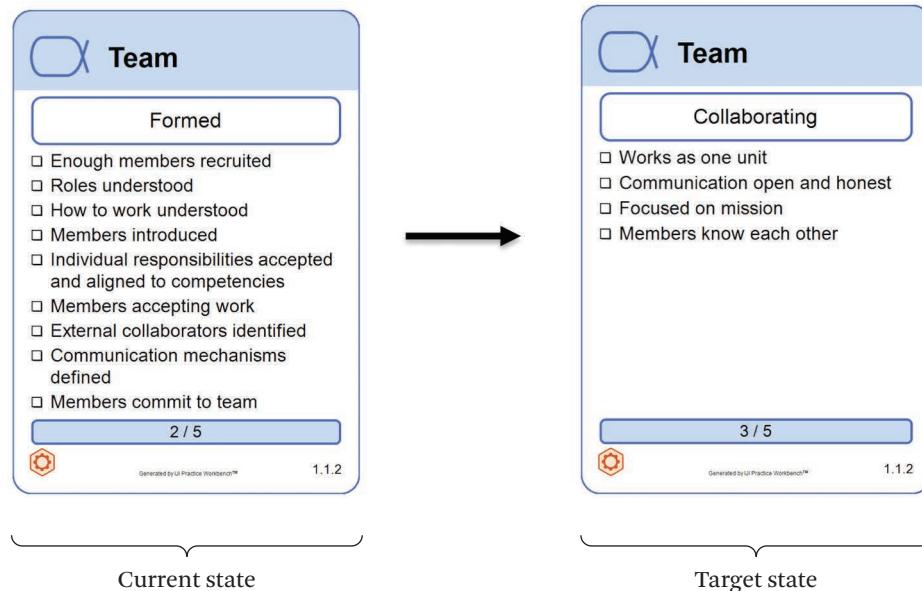


Figure 10.10 Team current and target states.

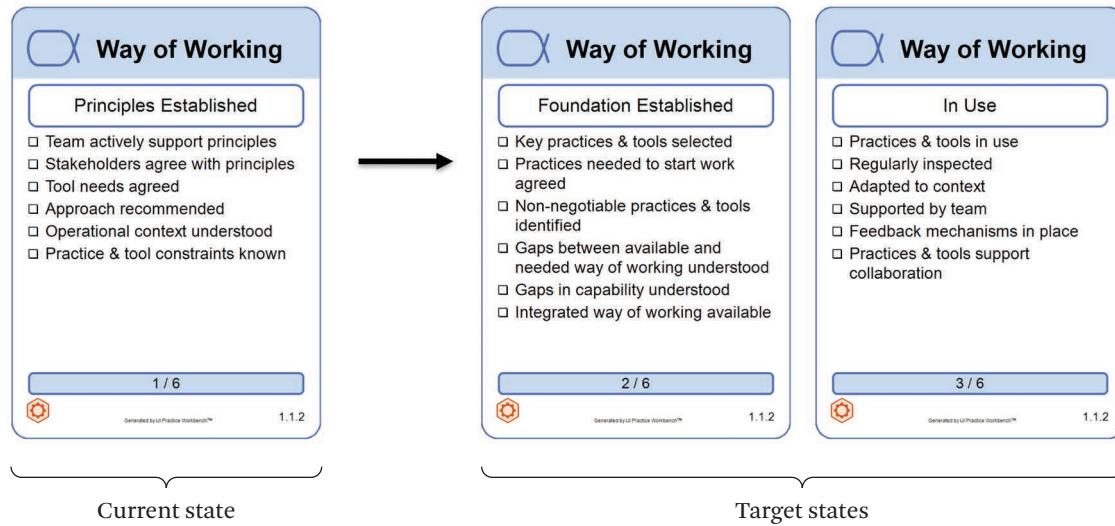


Figure 10.11 Way of Working current and target states.

their work by Wednesday, they would be able to verify that they were working as one cohesive unit (Task: Integrate work by Wednesday).

Way of Working: Principles Established → Foundation Established, In Use (see Figure 10.11).

To get to Foundation Established, the team needed to establish a development and test environment. Tom agreed to set up the development environment that included the team's chosen repository version control tool and the test environment. Grace agreed to prepare the test environment and supporting scripts. Following the setup of the environment, the team would start to use it to complete their tasks during the first iteration (Task: Establish development and test environment).

10.2

Doing and Checking with Essence

With the goals (expressed as target alpha states) and the tasks identified, Smith's team proceeded to work on their respective tasks.

Smith and his team members were co-located: sitting near each other in the work place. Angela's work area was located on a different floor, near Dave. Travel-Essence had an internal corporate chat application for collaboration, so all of them could access each other when needed. In general, work went rather smoothly, as the members were familiar with each other and the technology they were using.

On Friday afternoon, they did indeed achieve their goals and demonstrated the implementation of the designated requirement items to Angela. They reviewed their health and progress by playing the Chase the State game again and comparing the results to the previous time. In the following results, the target state for each alpha is listed in parentheses, with the actual results of the team's effort described afterward.

From the perspective of the Customer area of concern:

Stakeholders (Involved). Following the Friday demo, Smith had a side meeting with Angela and Dave, where they discussed and agreed to their involvement in future demonstrations.

Opportunity (Value Established). The Friday demonstration was successful and convinced Dave and Angela that the system could potentially produce significant user interest, leading to increased business. As a result, Dave was ready to move forward and fund the effort.

From the perspective of the Solution area of concern:

Requirements (Coherent, Addressed). The team had successfully clarified the open issues related to the agreed requirements, and then they successfully addressed those requirements in the Friday demonstration.

Software System (Demonstrated). The team successfully demonstrated the critical parts of the system that had been agreed to for the Friday demonstration.

From the perspective of the Endeavor area of concern:

Work (Prepared, Started). The team had successfully broken their agreed tasks down for the first iteration, understood the risks, and moved forward coding, testing, and integrating the pieces in preparation for the Friday demonstration.

Team (Collaborating). The team successfully integrated their work on Wednesday in preparation for the Friday demo. This activity verified that the team was working as one consistent unit.

Way of Working (Foundation Established, In Use). The team had successfully gotten their environment set up and used it during the first iteration to complete the work for the Friday demonstration.

10.3

Adapting a Team's Way of Working with Essence

The kernel clearly helped the team capture and apply the essence of software engineering. It reminded them to

- involve key stakeholders;
- think about the opportunity;
- break the work down to fit in the agreed way of working;
- think about risk;
- clarify requirements;
- integrate each team member's work with teammates' work; and
- focus on the most important things first.

But there will still always be better ways of doing things. So after the successful Friday demonstration, the team decided to discuss what went well, what did not go so well, and how they could do better during their next iteration.

During this discussion, Smith reminded the team of their agreed-on target alpha states from the first iteration; see Figure 10.3.

Smith then asked the following questions:

- What went well with our planning, doing, and checking related to the above alpha states?
- What did not go well with our planning, doing, and checking related to these alpha states?
- What can we do better with our planning, doing, and checking related to the alpha states?

Joel said, "We achieved a successful demonstration and now Dave is going to fund our full endeavor, so that certainly went well."

Tom said, "The way to achieve the Requirements: Addressed state was not clear to me at the start of the iteration. I learned that I had to talk to Angela and get her to agree to the requirement items to be implemented. I didn't understand this just by looking at the state checklist."

Grace said, "Actually, Smith, for me to do my job better, I would like to have better guidance regarding how to work on a requirement item."

Smith first considered Tom's request. That was easy; all Smith had to do was to supplement the state checklist with some additional guidance. He scribbled two

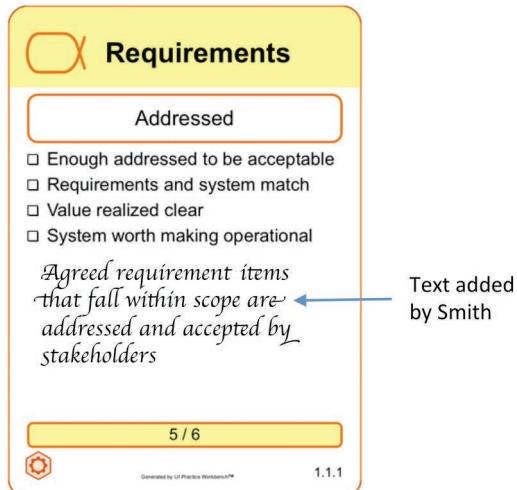


Figure 10.12 Additional guidance beyond the standard on achieving a state.

lines of text onto the Requirements card as follows (see Figure 10.12):

- gain agreement on requirement items that are within scope of Addressed; and
- implement these requirement items.

These notes were additional guidance on how to achieve the Addressed state.

Then Smith considered Grace's request. This request was not as easy. It meant making the way of working on all requirement items more explicit. We will discuss how to do this in Part III.

The way of working affects all team members, and every team member can contribute. This is another area where the kernel is useful. By talking about the alpha states after their successful demonstration, as we have just observed, the team came up with a number of good ideas on how they could do better during the next iteration.

10.4 How the Kernel Helps Adapt Their Way of Working

The kernel helps teams adapt their way of working in multiple ways.

10.4.1 Helping a Team Reason about Their Way of Working

First, it helps a team reason about their way of working and decide if there are improvements they should make.

Developers who come straight from an educational program often know more about programming than developing software, and more about developing software than working as a team and improving their way of working. Because their experience is limited, they often need a little help. The alphas and their states can help a team reason about their way of working as they try to improve.

When reviewing their way of working, we make the alpha states visible to the team members to help them think about their “process.” If you are conducting a review of your way of working for an iteration, you only need to make visible those states relevant to the current iteration (i.e., the iteration’s target states). This helps the team stay focused on reviewing just their way of working related to that iteration.

By visualizing the states, a mental transition takes place. The team is now looking at the “process.” We then look at each state specifically, and ask the same questions.

- What went well during this iteration, and have we achieved this alpha state?
- What did not go well during this iteration, and do we know what is keeping us from achieving this alpha state?
- What can we do better in the next iteration that will help us achieve this alpha state?

10.4.2 Making Changes to the Way of Working

The daily contact your team has with the alpha states (and hence the kernel) help you find simple improvements to adapt your team’s way of working. This may mean adding additional items to the alpha state checklist to meet your team’s needs. Teams can also define new alphas or add checklists to help team members, such as the text Smith added to Requirements: Addressed to help Tom. How to extend the kernel elements further with more explicit practices is discussed in Part III.

Keep in mind that the team should only add information to the kernel elements and their checklist items. Changing the information that is already there would undermine the value that we gain through the use of a standard kernel. The standard is the basis for essentializing methods and practices, a subject discussed in Part I. You will learn more about essentializing in Part III, namely how to express explicit practices using the Essence language.

You can just think of “essentializing software engineering” as representing the way your team is working using the Essence language and the Essence kernel common ground. This can help your team understand more clearly what missing elements they need to focus on in order to be successful in their endeavor.

What Should You Now Be Able to Accomplish?

After studying this chapter, you should be able to

- explain the terms iteration and iterative development;
- explain the activities involved within iterative development (i.e., planning, doing, checking, and adapting);
- explain the use of individual card games within the planning, doing, checking, and adapting parts of the iteration cycle, and be able to apply them;
- explain the role of discussion and team agreement during these activities;
- explain the role of “healthy alpha” and its use in examining alpha states; and
- give examples of the “Way of Working” and means to adapt them during development, meeting the needs of the team and the endeavor.

The Development Journey

In Chapter 10, we showed how Smith's team conducted one single iteration using Essence through the Plan-Do-Check-Adapt cycle. In this chapter, we will walk through how Smith's team's journey would continue to successful completion and discuss how Essence helps the team ensure progress and health. Specifically, we show

- the importance of tracking progress during a development endeavor; and
- the ways Essence facilitates progress tracking, via surfacing all of the essential dimensions so that an accurate progress assessment is visible and timely actions can be taken.

11.1 Visualizing the Journey

Let's look at Smith's team's development progression in terms of how its requirement items evolve across their journey. Figure 11.1 shows what is referred to as a cumulative flow diagram. Such a diagram provides a visual display of how the requirement items progress. It has time on its horizontal axis, and the distribution of requirement items across different states on the vertical axis. In this figure, you can see the states of the Requirement Item sub-alpha (Identified, Described, Implemented, and Verified). The team can use these states to help assess their progress and health in the iterations after the first one.

Smith's team's list of requirement items was not static. Not every item was identified upfront as shown on the y-axis. Instead, items were added at the end of each iteration cycle until the end of the third iteration. You can also see from this diagram that some items were verified in the first iteration. Those are the three requirement items discussed in Chapter 9 that the team planned to work on in the first iteration and demonstrate to Angela. As the team continued into iteration 2, three more requirement items were identified, and more requirement items were implemented and verified by demonstrating them to Angela. In Part III, we will discuss how the team used an explicit requirements practice (user stories or use

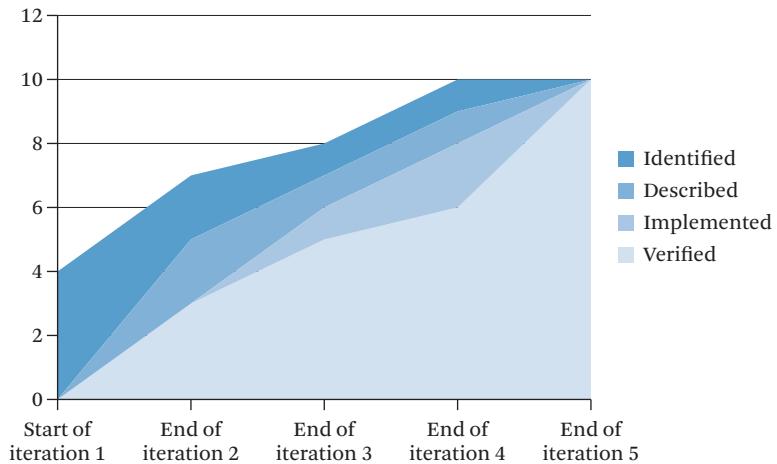


Figure 11.1 Cumulative flow diagram.

cases) as their endeavor started to become more complex, and they realized they needed to improve their requirements approach.

11.2 Ensuring Progress and Health

Most practitioners are concerned about progress: how much stuff gets developed, how many defects are found, how many defects are fixed, and so on. But the health of the development endeavor includes far more than these individual progress measures. The Titanic was progressing very well until it hit an iceberg and the rest was history. Similarly, often software endeavors appear to be progressing very well when limiting your view to only a few dimensions. The role of Essence alphas is to bring all of the essential dimensions to the surface so that an accurate progress assessment is always visible and timely actions can be taken when necessary.

Table 11.1 shows the numerical evolution of the Essence kernel alpha states. By numerical, we mean that each number represents the *n*th state of the kernel alpha: e.g., state 1 of the Requirements alpha is Conceived, state 2 is Bounded, and so on. The column “Start of Iteration 1” represents the time when Smith’s team received the work. “End of Iteration 1” represents the end of the first iteration, which we discussed in detail earlier (see Chapter 9). The complete story is four iterations.

For example, in our simple story in Chapter 9, the team was able to get their key stakeholders (Dave and Angela) involved and therefore achieved the Stakeholders:

Table 11.1 Kernel state evolution

	Start of Iteration 1	End of Iteration 1	End of Iteration 2	End of Iteration 3	End of Iteration 4	Target
Stakeholders	1	3	4	4	5	5
Opportunity	2	3	3	3	3	3
Requirements	1	2	5	5	6	6
Software System	1	2	2	4	4	4
Work	1	3	4	4	5	5
Team	2	3	4	4	4	4
Way of Working	1	3	3	4	5	5

Note: The numbers in the table indicate the achieved state.

Involved state (state #3) in iteration 1. The team also successfully addressed all three requirement items that were agreed to for the first iteration. However, Smith's team agreed to apply Essence to the complete four iterations when assessing their progress. This means that although they completed all the requirement items they planned to address in iteration 1, they did not assess their state of Requirements to have achieved the Addressed state at that point, because they had not addressed all of the requirement items needed for the full four iterations. Although the Requirements alpha doesn't seem to progress in iteration 1, by introducing requirement-item sub-alphas the team can track more accurately the progress of each requirement item and thus of the requirements as a whole.

In the simple case, we described the states as all having progressed forward throughout the four iterations. But when new requirement items are added in later iterations, it is possible that the requirements alpha state may fall back to a previous state. This is because the new requirement items may not have achieved the same state as the original requirement items. For example, they may not have achieved the Coherent state (state #3) or even the Bounded state (state #2). So the states of the alphas do not always move forward linearly as depicted in Figure 11.1 and in our simple example.

We can also picture the evolution of the kernel alpha states as a radar diagram [Graziotin and Abrahamsson 2013] (see Figure 11.2).

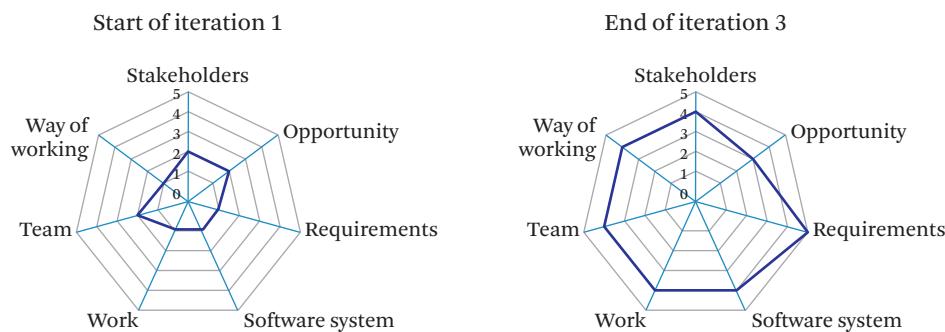


Figure 11.2 Radar diagrams for the start of iteration 1 and the end of iteration 3.

We see the endeavor evolved gradually with alphas progressing from lower states to higher states. If a particular axis in the radar diagram (i.e., an alpha such as Software System) is progressing slower than projected, this prompts the team to consider what is getting in the way and what needs to be done to keep progressing as planned. One possible way is to reduce the scope of the requirements or look for a different approach to implement the software system so that it can be validated earlier.

Thus, the individual alphas progress one by one and together the alphas progress in *waves*. In order to make progress for the whole endeavor, the alphas need to progress in a balanced way. For instance, you cannot progress the Software System alpha without also progressing the Requirements alpha. These parallel progressions in waves are critical for the success of the endeavor.

11.3

Dealing with Anomalies

Well, the reality is that things don't always go according to plan. This means that once you reach a certain alpha state, things can happen to cause your endeavor to fall back. For example, stakeholders who were involved at the start of the endeavor may stop participating because of other priorities, and even though your work may be under control one day, the next day you may discover new risks that you never expected, and teams that are working well together collaborating may lose seasoned team members and instead gain new members with less experience. All of these kinds of situations can cause your endeavor to fall back. This is why teams should periodically use the Essence kernel alphas to provide a very straightforward health check.

What Should You Now Be Able to Accomplish?

After studying this chapter, you should be able to

- name different aspects of software engineering that are worthy of progress checking (e.g., requirements met, artifacts developed, defects found, defects fixed);
- explain what the cumulative flow diagram is and how it can be used to track progress;
- explain the means Essence provides to facilitate progress tracking; and
- explain what the radar diagram is and how it can be used to track progress balance among different alphas.

Reflection on the Kernel

The goal of this chapter is to reflect on general observations on software engineering using Essence and its kernel, beyond the example discussed thus far. In this chapter, we show

- that software engineering is essentially concerned with addressing complexity when developing software;
- that each alpha addresses the complexity of a development endeavor from a different perspective; and
- that each alpha/dimension is associated with its own lifecycle, and underlined with its own theory.

12.1

Validity of the Kernel

What we have shown in this part is one single case of using the kernel, as we walked through how Smith's team progressed through all its states. We can say that the kernel is valid for this case, but does it work for other potential cases? To validate that, we would need to apply the kernel over a broad range of endeavors. As a matter of fact, we authors have done this both through academic classes and real-world industrial projects. As of 2017, we have never received any proposal for removing any elements from the kernel. However, we have received proposals to add elements, and such additions will probably happen as time goes by, but in the interest of being conservative, they must be very well motivated.

12.2

Applying the Kernel Effectively

There are two important approaches that Smith applied to run his endeavor effectively: the kernel alphas and the facilitation games using cards.

Let's revisit alphas. Development endeavors are complex and multi-dimensional. Each alpha addresses this complexity from a particular dimension, and readers should be able to recall and name these seven dimensions quite readily (Opportunity, Stakeholders, Requirements, Software System, Work, Team, and

Way of Working). Each comprises a lifecycle of its own. Each dimension is based on some underlying theory. For example, the Team alpha is based on the Tuckman model of team formation and performance; the Stakeholder alpha is based on works on stakeholder engagement [Ng 2015]; and so on.

Even though the alphas are separate, they are not independent. They are just different views of the same development process. As such, in an endeavor the kernel alpha states progress in waves, as discussed in Section 11.2. We have to make moves that take the endeavor from one set of states to another set of states, and the balance is achieved by doing enough on each dimension to be able to progress all dimensions. This wave-like progression acts as a reference for detecting anomalies, as we discussed in Section 11.3. If some dimension—e.g., Software System or Stakeholder—progresses much more slowly than other dimensions, it would be a cause for concern. Waterfall projects can fall into the error of producing a demonstrable Software System too late. Agile projects may fall into the trap of not seeking early Stakeholder feedback and consensus. Thus, the kernel alphas provide a simple yet holistic progress and health check.

The second approach is the facilitation games we introduced in Chapter 8. Development is a complex process with participants coming from different backgrounds and having different intent; they thereby could have different ideas of how best to run the endeavor. Although the kernel alpha state cards and checklists provide descriptions of what each alpha entails, there is still important work needed to ensure that alphas are being used and understood. As such, the cards are important consensus facilitation tools.

Software engineering is cooperative, and having consensus among its participants is crucial for success. Just having a prescribed kernel alpha definition is not sufficient; we need to get these definitions into the hands of the participants and to have them used in everyday conversations. The cooperative games discussed in this part of the book help build this consensus, aligning different people's understanding and thereby working toward a common goal, which is the success of the endeavor. (There are other games supporting the actual development work, for information on these, see the Recommended Additional Reading at the end of this chapter.)

What Should You Now Be Able to Accomplish?

After studying this chapter, you should be able to

- explain using examples that although alphas are separate, they are not independent;

- explain how anomalies in the endeavor can be detected via the disbalance in alpha progress;
- name potential software engineering anomalies in waterfall and agile projects, and explain how Essence can help to detect them;
- name ways to integrate Essence into software development process; and
- perform card games you now know in the context of a method you know, so that the card play smoothly integrates into daily team conversations.

Postlude

In Part II of this book, we showed how a team conducted a development endeavor using minimum explicit knowledge. (This knowledge is captured in the Essence kernel, and in particular the alpha state cards.) However, the team's way of working was not static. For example, they added an additional sub-alpha (Requirement Item), and they added some additional checklists to a state (Requirements: Addressed) to help a team member understand when it was achieved.

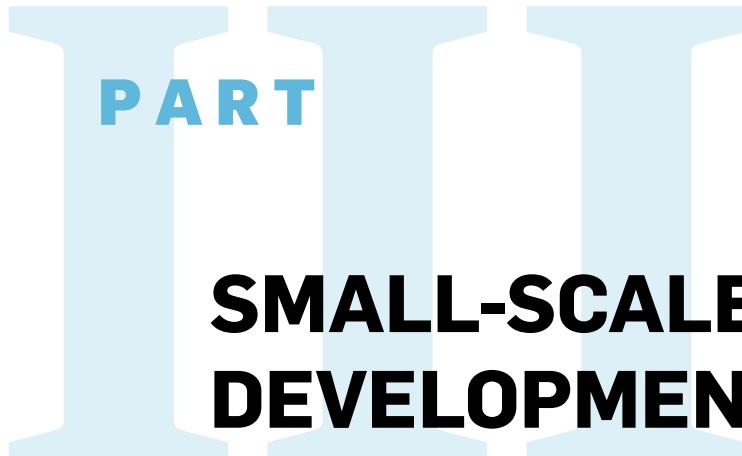
As we described, teams have to add practices to the kernel to get a complete way of working. Often, many teams don't describe these but keep them tacit. When such teams grow in numbers, and as new members join and others leave the team, it becomes quite difficult, especially for the new members, to understand exactly how the team functions.

Even if a team did make their way of working explicit, it is another thing to structure it in a way that is easy for the team to use and improve as they learn. So, how does one accomplish this? In Part III we will demonstrate how to achieve this goal through the idea of explicit practices.

Recommended Additional Reading

- Alpha State Card Games [[Alpha State Card Games 2018](#)]. Here you will find the games described in Chapters 8 and 9 developed in some more detail. You will also find additional games to support your team's software development endeavors.
- P-W. Ng, Making software engineering education structured, relevant, and engaging through gaming and simulation [[Ng 2013](#)].
- D. Graziotin and P. Abrahamsson, A web-based modeling tool for the SEMAT Essence theory of software engineering [[Graziotin and Abrahamsson 2013](#)].

- P. E. McMahon, A thinking framework to power software development team performance [[McMahon 2015](#)].
- P.-W. Ng, Integrating software engineering theory and practice using Essence: A case study [[Ng 2015](#)].



PART

SMALL-SCALE DEVELOPMENT WITH PRACTICES

In Part II, we introduced how Essence helps teams to evaluate their current state, to plan, and to progress toward achieving the next state. However, how to actually get to the next state is not shown by merely using the Essence kernel. This instead requires tacit or explicit knowledge that you need to have or acquire on top of what the kernel gives you. (In this case, tacit means it is known by the team, but not necessarily written down. Explicit means it is written down.) When facing simple development endeavors, like the one described in Part II, you just need to rely on tacit knowledge. However, in other situations—due to team members coming from different backgrounds, experiences, and/or competencies—it is useful to have some tangible and explicit practices for them to fall back on. Essence helps here, as well. The Essence language allows you to define practices on top of the Essence kernel, providing explicit guidance to teams on how to get to the next goal state(s) or how to maintain the health of the current state(s). By “on top of” we mean using Essence, the kernel elements, and the language as a vocabulary to describe your team’s specific practices, or in other words using Essence to essentialize practices. Moreover, teams usually need many practices and these need to be merged or composed to remove overlaps and conflicts. Thus, the Essence language includes a composition mechanism that is described later in this part to allow teams to create a method including many practices of their choice.

In this part, we will motivate the need for explicit practices and discuss how explicit practices that are described on top of Essence are used by small development teams to help solve the common challenges they often face. As an example, we have chosen a team that uses an agile approach, but we could have chosen a team that uses a more traditional approach, such as a waterfall approach. Agile development, of course, is the mainstream approach to delivering software at the time of writing this book and is utilized in many organizations. Agile development is not just a method, but rather it is a mindset, with principles as well as practices. Over the years, common agile practices have been codified (i.e., written down), and we explain in this part how to modularize them as practices using Essence. In particular, you will learn how a small agile team makes use of *Scrum*, *User Stories*, *Use Cases*, and *Microservices* practices to solve specific challenges they face during their development. This development typically takes several weeks to complete, and we will see how this team evolves their way of working practice-by-practice while working on a list of changes. Now we consider a new and different situation with respect to the TravelEssence development team. We consider it now enlarged with new persons. Also we start with a slightly different development situation which is summarized below (see Chapter 13 and Table 13.1).

Part III will cover the following objectives.

- Appreciate what practices are, and the types of challenges teams often face where practices can help. We will describe practices including Scrum, User Stories, Use Cases, and Microservices.
- Appreciate the value that representing practices in an essentialized form provides in helping you find the right practices for your team.

Kick-Starting Development with Practices

In Part II, we saw how a small team kick-started development using Essence. The goal of this chapter is to extend the process with the use of explicit practices to help the team with particular challenges they face. Specifically, in this chapter we show

- the elements behind kick-starting development with explicit practices, and
- the role of these elements and—at an elementary level—the roles of four popular practices detailed later in the book (i.e., Scrum, User Stories, Use Cases, and Microservices).

Furthermore, we demonstrate through our example how a real endeavor can progress as more people and stakeholders become involved. We reflect on the state of our simple endeavor after the demonstration discussed in Part II, and move forward from there. We will learn how the conditions have changed and what effect that has had on the state.

Now that our kick-start process will involve not just tacitly agreeing on the things to monitor as we saw earlier, but agreeing on explicit practices to apply, the kick-start sequence has changed. It is now as follows.

1. Understand the context through the lens of Essence.
2. Agree upon development scope and checkpoints (where it begins and where it ends).
3. Agree upon practices to apply.
4. Agree upon the important things to watch.

In the remainder of this chapter, we will explain what each of these items means, and we will describe how teams can use practices effectively in development. As

as this part of the book progresses, we will see firsthand through TravelEssence's team the value explicit practices can bring to them over the simpler tacit practices they have used so far.

13.1 Understand the Context Through the Lens of Essence

We have already explained in Chapter 9 why it is important to understand the context of a development endeavor and how the Essence kernel alpha states can help. What we are illustrating in this part is similar, except that the context is different.

Table 13.1 shows a mapping of each Essence kernel alpha to its state at this point in time, and the rationale for the team's assessment of these current states.

Table 13.1 Development context through the lens of Essence

Alpha	State achieved	Rationale for achieving the state
Stakeholders	Involved	Cheryl, Dave, and Angela are key stakeholders in the endeavor. The state is achieved because they were actively involved in helping the team achieve a successful demo.
Opportunity	Value Established	Achieved this state because the team had a successful demo supporting the objectives of the Digital Transformation Group.
Requirements	Bounded	Achieved this state because the team had successfully gotten the key stakeholders involved and those key stakeholders had reached a shared understanding of the extent of the proposed solution.
Software System	Architecture Selected	Achieved this state because they had made their decision to use the existing proven mobile app, and to use an architecture approach referred to as microservices to host their recommendation engine.
Work	Initiated	Achieved this state because all the team members had agreed that the source of their funding, and the stakeholders who would fund the work, were clear.
Way of Working	Working Well	Initially tacit agreed practices as discussed in Part II worked well for the team, but as we shall see the team eventually evolved to the more explicit practices of Scrum, User Story, Use Case and Microservice due to changes in their endeavor as it progressed.

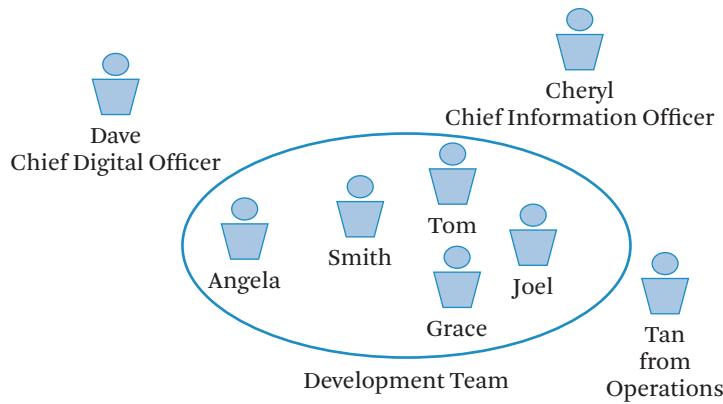


Figure 13.1 Participants in our story.

As pointed out above, we now consider a new and different situation with respect to the TravelEssence. Following the team's successful demonstration, Dave, the Chief Digital Officer and head of the Digital Transformation Group (DTG), decided to move ahead to the next phase of this initiative, expanding the scope and vision of the endeavor. As a result, there would now be more people involved.

We have already introduced Dave, Angela, Smith, Tom, Joel, and Grace. We will now introduce the new characters involved in the expanded initiative (see Figure 13.1). Cheryl was the Chief Information Officer (CIO), responsible for all IT systems operations including development and enhancement of each IT system. Developers like Smith, Tom, Joel, and Grace worked for Cheryl, as did our other new character Tan (we will find out more about his role in Part IV), from Operations.

As we move forward, we will learn how the endeavor conditions changed due to this and other alterations in the context, and how this affected the way the team assessed the alphas' achieved states.

13.2

Agree upon Development Scope and Checkpoints

As we have stated previously, a team needs to know where it is going. In Chapter 9, we demonstrated how the Essence kernel alpha states can be used to discuss and come to an agreement on what should be achieved by a checkpoint,¹ such as the internal demo our team successfully produced. In general, if a team has enough knowledge to accurately plan the road forward, it would look ahead and identify several checkpoints, including a release roadmap.

1. Another commonly used word for checkpoint is milestone.

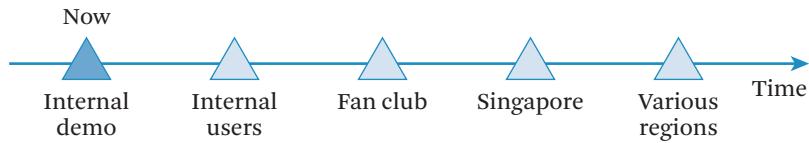


Figure 13.2 Release roadmap.

Dave and Angela discussed how TravelEssence would introduce the recommendation engine to their travelers. They agreed on an incremental approach, starting with a small number of internal users and gradually rolling the product out to travelers across various regions of the world, as depicted in Figure 13.2.

Internal users were primarily folks from the sales and marketing department with whom Joel interacted. These users traveled as part of their jobs and were excellent candidates for reviewing the mobile application before introducing it to real users.

The first set of actual users outside of TravelEssence would be its fan club. This included folks who were frequent and loyal users of the app and services provided by TravelEssence. After the fan club, the first target set of users would be frequent travelers to and through Singapore. TravelEssence had been approached by the Singapore Tourism Board and had been working with them. Knowing that the Singapore Tourism Board had been actively seeking out collaboration with travel providers, TravelEssence executives agreed together that this was a great opportunity.

With the first milestone (the demo) having been successfully achieved, it was now time for the team to set its sights on its release to the internal users. While the Friday afternoon demo to Angela had been a success, the team received feedback on areas that Angela felt should be improved before the next release. So, to get started, the team decided to review the current state of the whole endeavor once again by playing the Chasing the State game to determine where they were, and then playing the Objective Go game, as discussed in Chapter 8.

Below you will find a sampling of the results of the team discussions in playing the game, and what the team agreed to be the next focus states to be achieved for the upcoming release to the internal users.

As usual, we begin with alphas in the customer area of concern.

Stakeholders: Recognized → Involved (see Figure 13.3).

While the team had agreed that they had achieved the Stakeholders: Involved state during the internal demo, they realized that they had new stakeholders joining the

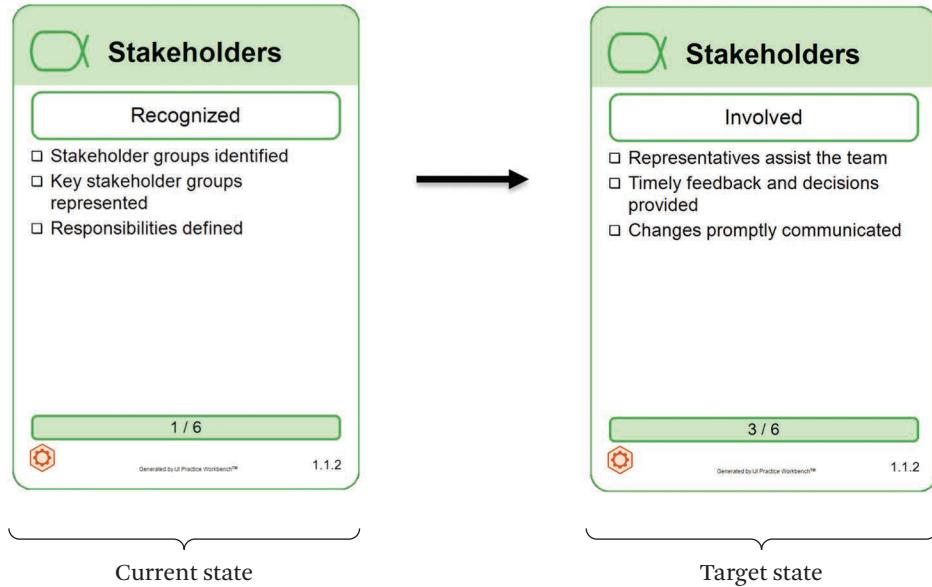


Figure 13.3 Stakeholders current and target states.

endeavor (i.e., Cheryl, marketing and sales) and so now they would need to gain new agreement on these stakeholders' involvement and commitment.

Opportunity: Value Established → Value Established (see Figure 13.4).

There would be no change in state for the Opportunity, except that the team would have greater confidence about the value of the recommendation functionality.

Next, we examine the alphas in the solution area of concern.

Requirements: Conceived → Coherent, Addressed (see Figure 13.5).

While the team had achieved their goal for the internal demo, they knew there would be new requirements to be added during the next iteration, as well as a need to address the specific issues that Angela had raised during the internal demo.

Software System: Demonstrable → Usable (see Figure 13.6).

Although the team had successfully demonstrated critical parts of the system to Angela, that demonstration had also uncovered multiple defects that would need to be fixed before the system was usable. The team also knew there was more functionality that the marketing department felt would need to be added before the product was ready to be shown to real customers.

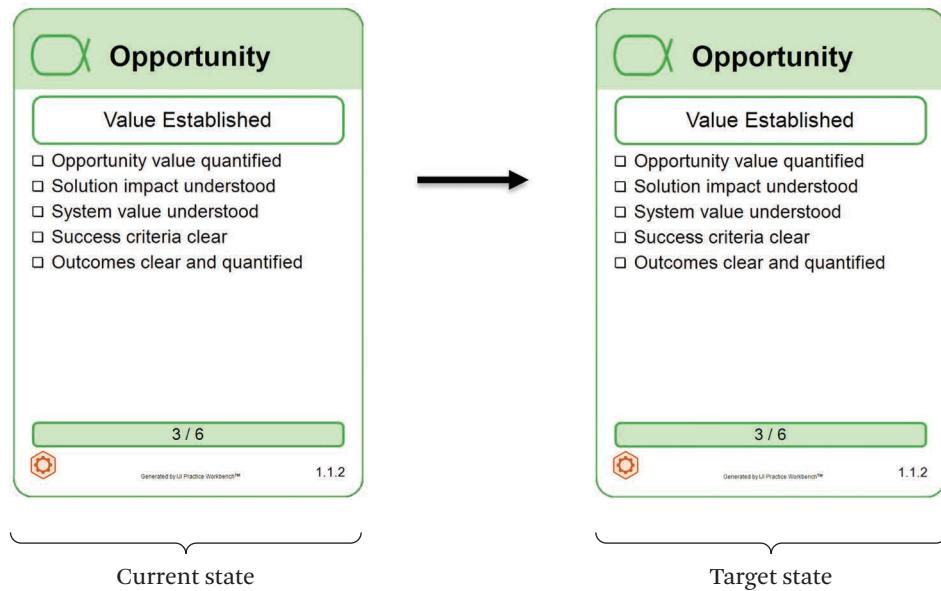


Figure 13.4 Opportunity current and target states.

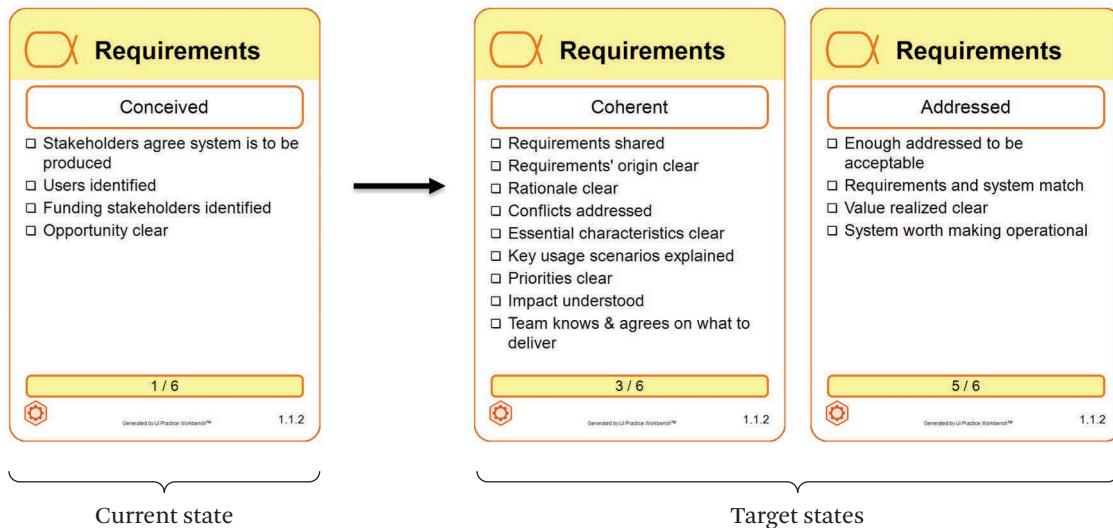


Figure 13.5 Requirements current and target states.

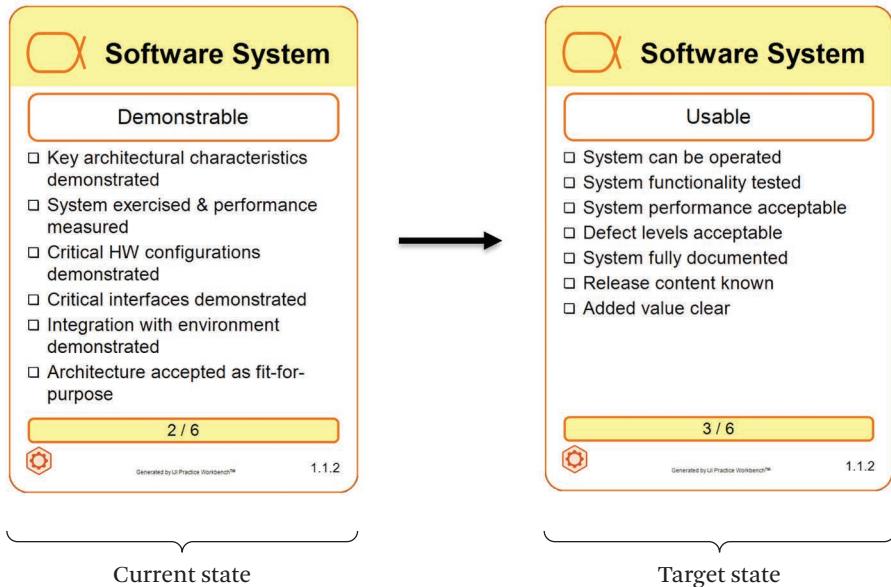


Figure 13.6 Software System current and target states.

Last, we examine the alphas in the endeavor area of concern.

Work: Initiated → Prepared (see Figure 13.7).

Smith's team had to quickly drill down the scope of the endeavor and come up with a credible plan and schedule for the initial release to the internal users and fan club.

Team: Collaborating → Performing (see Figure 13.8).

While Smith's team members had worked together before, during the internal demonstration, they knew there were areas for which they could improve both their practices and tools, and also that there would be additional team members to work with during the upcoming iteration.

Way of Working: Principles Established → Foundation Established, In Use (see Figure 13.9).

Although Smith's team had successfully established a development and test environment during the internal demo, the team members realized that there had been some miscommunications in the process of conducting certain activities. As a result, they agreed that they would need more explicit practices to make sure everyone understood and agreed to how the team conducted these activities.

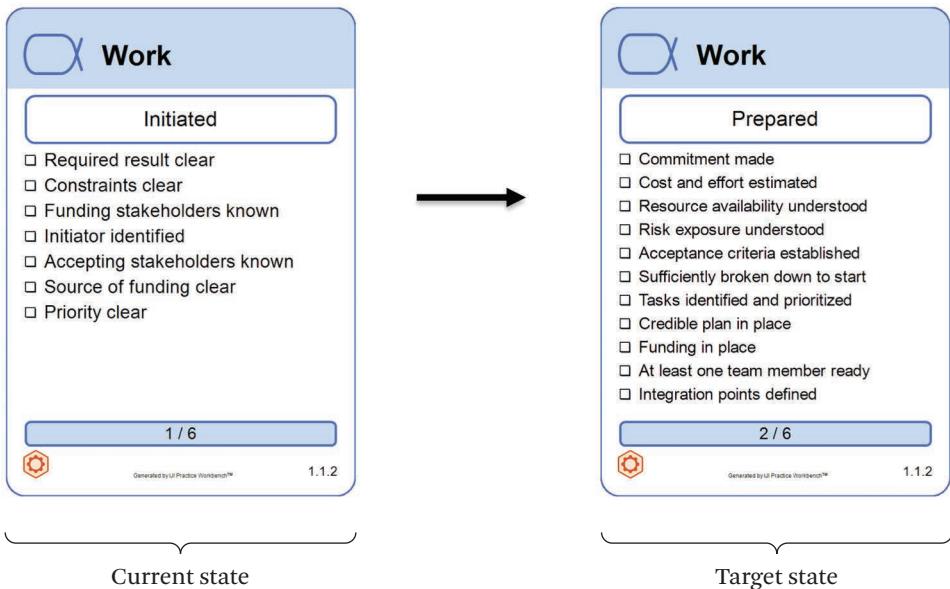


Figure 13.7 Work current and target states.

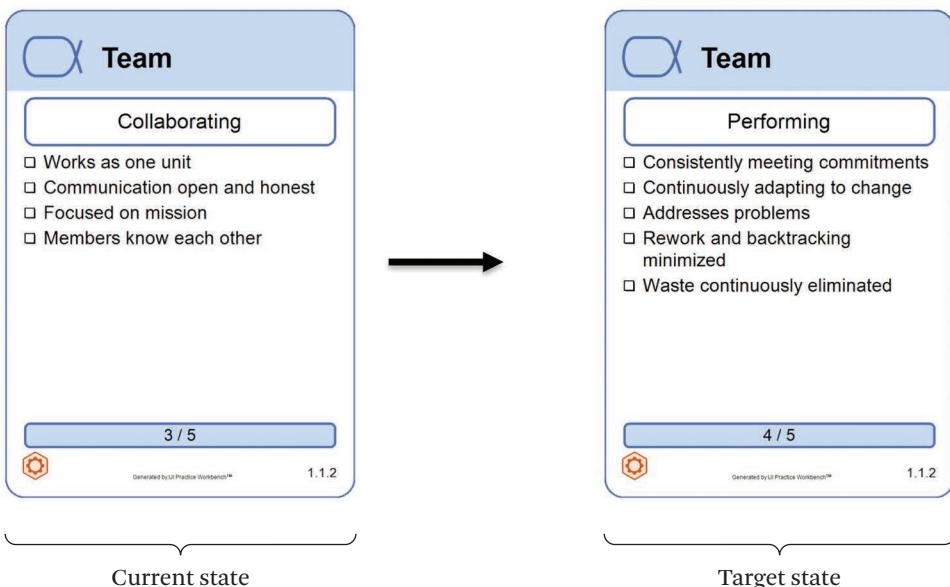


Figure 13.8 Team current and target states.

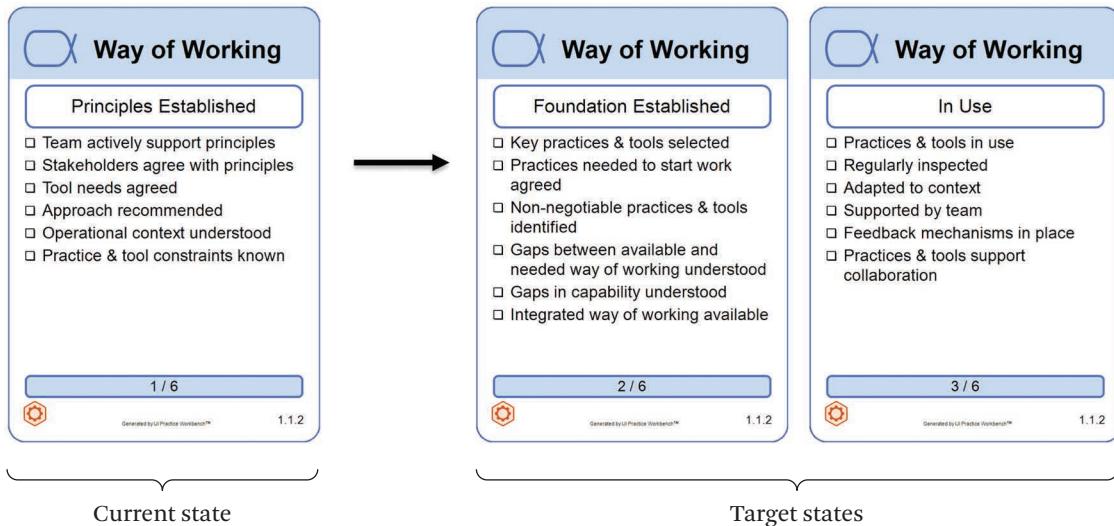


Figure 13.9 Way of Working current and target states.

13.3

Agree upon Practices to Apply

Unlike the previous two steps, which just use the Essence kernel as we did in Part II, this is a third and additional step. In this step, we are explicitly selecting a practice to apply. We assume that there exists a library of practices from which a development team can pick to address its challenges. We also assume that the team is knowledgeable in selecting practices, or that there is a convenient and easy way to select them. Of course, if a team does not possess this knowledge, there would be a facilitator or coach to help the team. Let's assume for this part of the book that Smith is such a knowledgeable coach.

Smith's development team was now no longer working on a demo or proof of concept. They would be working as part of a larger development team and as such would be using practices mandated for development.

Cheryl (the CIO) had, after a series of successful pilots, mandated that Scrum and either User Stories or Use Cases be employed by all development teams. By "pilots" we mean endeavors where certain practices are tested to determine if they will work well in an organization before deploying the practices formally across the full organization.

Scrum is a popular practice to help teams collaborate and work effectively in an iterative manner. To develop software, however, Scrum is not enough. We need practices to come up with the right requirements, and to design, code, and test

these requirements. Since Scrum itself suggests an iterative way of working, this applies to requirements, design and test—a small requirement to start with, then design and test to fulfill it, and when that is working well, more requirements can be addressed, etc.

When it comes to requirements, we will demonstrate that development can be done in different ways by applying different requirements practices. We will first utilize the user story practice and then the use case practice. These practices are not equivalent; they address requirements breakdown and other aspects of requirements differently. However, in this book we won't create a “methods war” because our intent is not to tell our readers which practices or methods we may think are better. Rather, our intent is to explain how you can do your own comparison and reach your own conclusions when using Essence-based practices.

Many Scrum teams utilize user stories to help them understand and agree to the requirements. A user story is a written description of a story that describes functionality that will be of value to a user of a software system. User stories are first written without all the needed details. The details are fleshed out in discussions between the team and the stakeholders. By encouraging informal conversations, the real needs of the customer surface, along with the details of the requirements. Direct communication between development team members and users can be an effective practice to help ensure that the requirements that eventually get written down are in fact the real requirements needed by the users. We will elaborate on the user story practice in Chapter 15.

Some teams decide that the short informal descriptions that are created with user stories are not sufficient to help them flesh out the requirements details. In such situations, use cases provide one possible alternative. Use cases help teams capture and validate the completeness of requirements. Use cases provide a diagram that then gives an overview of all the use cases in the system being built and their interactions with one or more users of the system. This diagram also helps some teams evolve their requirements by clarifying what the system will do, and what will be done by the users of the system. We will discuss use cases in greater detail in Chapter 16.

In our story, after the development team discussed what worked well and what didn't work so well after the internal demo, they realized they needed a more disciplined approach to capturing requirements. They started out with a plan to utilize user stories; however, they later decided that use cases better met their needs.

After some discussion, the team also decided to use microservices to help them evolve the software system. Microservices are small independent processes that communicate with each other through well-defined interfaces. By building the

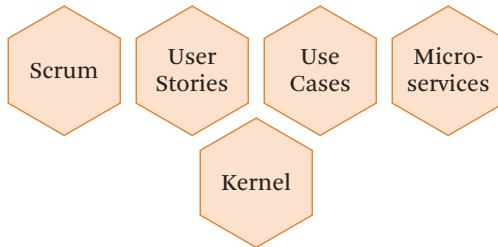


Figure 13.10 Practices for TravelEssence teams.

recommendation engine as a separate microservice rather than as a new module within the legacy system in TravelEssence, they would be able to experiment and evolve the new functionality with less risk of causing new defects in the legacy system. We will elaborate on the microservices practice in Chapter 17.

Thus, for the next release to internal users, Smith and his team agreed to use these practices in addition to the kernel.

You will learn more about how teams can use Scrum, user stories, use cases, and microservices as explicit practices later on in Chapters 14–17.

We want to reiterate that the practices in Figure 13.10 are examples to illustrate how essentialized practices can help teams do their work. They are a mixture of collaboration, engineering, and technical practices. For example, scrum is about team collaborations. User stories and use cases are requirements engineering practices (any decent requirements engineering book will discuss these two practices). Microservices is a highly technical practice, and we would deem it a rather advanced one, with which even rather senior software practitioners are learning and experimenting as we write this book.

As a software professional, you will come in contact with many other practices. There are simply too many for us to even attempt to enumerate in this book. We believe that in due time, popular practices will be essentialized, and as a student or professional who has a good understanding of Essence, you will learn these practices quickly. We will discuss more about how practices help teams grow and become more competent when we conclude this part of the book in Chapter 18.

13.4

Agree upon the Important Things to Watch

This fourth step to kick-start development is similar to that in Part II, except that we have explicitly selected practices in addition to the Essence kernel. Recall that the important things to watch out for in a development endeavor are the alphas. Of

Table 13.2 Practices that TravelEssence applied

Practice	Description	Things to Watch (alphas)
Scrum	A practice for the iterative development of software systems working off a backlog.	Sprint Product Backlog Item
User stories	A way to capture functionality that will be of value to a user of a software system.	User Story
Use cases	All of the ways of using a system to achieve a particular goal for a particular user.	Use Case Use Case Slice
Microservices	A software architecture style that uses small independent processes to communicate.	Microservice

course, the kernel has already defined the seven universal alphas that you should now know well, plus definitions of states and checklists as reminders to evaluate the progress and health of these things, and hence that of the development endeavor.

However, the alphas from the kernel are not the only things to watch. There are others. In Part II, we saw that Smith's team added other things to watch as well, specifically Requirement Items. These kinds of additions are sub-alphas. In general, the practice you apply will explicitly call out specific things to watch out for, as we see in our TravelEssence story. So let's return to the story and continue to watch what Smith's team is doing.

Having agreed on the practices to apply, Smith's team now had inputs as to the important things (i.e., sub-alphas) to watch to ensure health and progress (see Table 13.2).

Further explanation will be provided in subsequent chapters, but we want to provide a look ahead, to alert you to some important things to watch for in the practices that we are going to introduce.

- Scrum is a practice for iterative development where each iteration, or time-box, is called a sprint. The sprint is an alpha, something we need to watch. Scrum guides teams to complete work items in a backlog. These work items, known as Product Backlog Items (PBIs) using Scrum terminology, can also be treated as alphas.
- User Stories is a practice about expressing requirement items more succinctly, focusing on values. Specific user stories can also be viewed as sub-

alphas, similar to treating Requirement Items as sub-alphas of Requirements.

- Use Cases is a practice that [Jacobson et al. 2011, 2016] helps teams identify and organize requirements in the form of use cases and use case slices. A use case slice is a part of a use case that is broken down to an appropriately sized piece of work for the development team to tackle. Specific use cases can be viewed as sub-alphas of Requirements and specific use case slices can be viewed as sub-alphas of a specific use case. This is discussed further in Chapter 16.
- Microservices is a practice that helps teams break down a complex software system into a set of cooperating small independent modules, each with its own purpose and each with its own well-defined interface to other modules. Specific microservices can be viewed as sub-alphas of the Software System kernel alpha and monitored as alphas.

From the previous discussion, it should be very clear that alphas are very important things in a development endeavor to help teams understand progress and health. The kernel calls out universal alphas explicitly, while practices call out practice-specific alphas. It is important to identify the right alphas, because there is a cost to making something an alpha due to the need to then explicitly assess and track its state. As an example, small teams that have experience working together but lean budgets may decide to track their progress and related risks through tacit knowledge only, whereas large teams working on more complex efforts are more likely to see the need and payback for explicitly tracking progress and key risks as alphas.

It is also important to identify the right states and the right checklists for each alpha, so your team can assess their progress and health. It is all these explicit practices your team agrees upon to use that help your team progress your alphas through their states by achieving the checklists.

13.5

Journey in Brief

In the remaining chapters in this part of the book, we will walk through the Travel-Essence team's journey of applying the more explicit practices their CIO requires, as well as Microservices, as their endeavor becomes more complex. The chapters will run as follows:

- Chapter 14—Running with Scrum
- Chapter 15—Running with User Story Lite

- Chapter 16—Running with Use Case Lite
- Chapter 17—Running with Microservices

In each chapter, we will briefly present an overview of one practice, the problems it solves, and how TravelEssence adopted and applied the practice and the benefits they achieved, along with the benefits achieved by representing each practice in an essentialized form.

What Should You Now Be Able to Accomplish?

After studying this chapter, you should be able to

- explain the challenges that might arise as the endeavor progresses (e.g., miscommunication at different levels if the practices are not explicit);
- name practices using examples that might be considered during the endeavor (e.g., Scrum or Microservices practices as used in our example) and explain the relation among them, if any; and
- give the related description and name the alphas for the practices discussed in this chapter.

Running with Scrum

The goal of this chapter is to introduce the Scrum practice, including its elements in the Essence language, and use the TravelEssence example to demonstrate the benefits of Scrum compared to the development approach used earlier in the book. In this chapter, the reader will be introduced to

- the Scrum practice and its elements;
- the relationships between the Scrum elements, activity flows, and to their relationships with kernel elements (i.e., Work, Requirements, Software System);
- the simplified version of the Scrum practice (called Scrum Lite) in a real endeavor, including the obstacles and challenges that might arise; and
- how kernel activity spaces are covered by the Scrum Lite practice.

As mentioned previously, Cheryl (the CIO) had mandated after a series of successful endeavors, that Scrum and either user stories or use cases be employed by all development teams. One reason why organizations often mandate specific practices and tools is to simplify training and communication. Recall that a practice is defined to be *a repeatable approach to doing something with a specific purpose in mind*. By making a practice explicit we improve communication, reducing the chances of someone misunderstanding how the practice is intended to be carried out.

Scrum is perhaps the most popular agile practice at the time of this writing. Jeff Sutherland and Ken Schwaber created Scrum to get teams to work iteratively and to collaborate more effectively by following a number of practical and proven activities.

14.1

Scrum Explained

Figure 14.1 shows a big picture overview of Scrum [Schwaber and Sutherland 2016]. It provides explicit guidance related to how a small development team with about

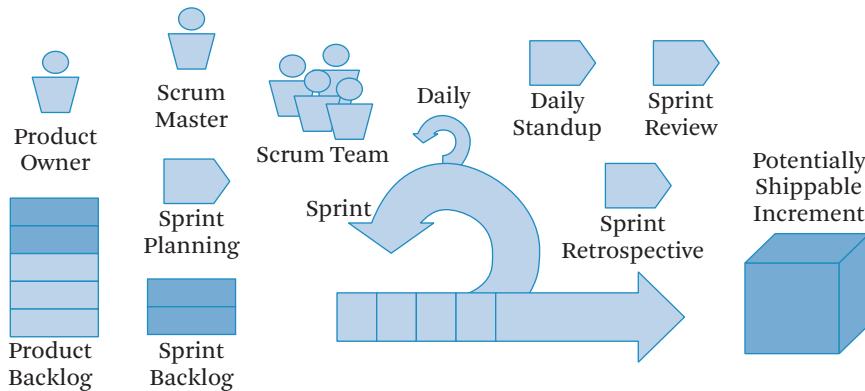


Figure 14.1 Scrum big picture.

7 plus/minus 2 team members can work together. Henceforth, we will refer to that team as a Scrum team. All the work that the team might have to do is first placed in an ordered list—the product backlog. The product backlog is maintained with the most important items near the top of the list. The things in the product backlog are called *Product Backlog Items* (PBIs). A PBI can be a piece of a requirement, something the team can do to improve themselves, or defects that they will have to fix.

The heart of Scrum is the sprint, a fixed-length period of time, usually one to four weeks, during which the team meets a certain goal, which includes producing a potentially shippable increment of the product to be developed. The PBIs to be performed in a sprint are selected through the sprint planning activity where the team, together with the product owner (PO), agrees on the highest prioritized PBIs to be worked on in the upcoming sprint. These PBIs are moved from the product backlog to the sprint backlog. This activity is done on the first day of each sprint by the full Scrum team working together to determine what can be delivered and how it can be delivered in the agreed-to sprint time period. There are two parts to the sprint planning activity. During the first part, the PO explains to the team the goals of the sprint and the Product Backlog Items that, if implemented, would achieve the sprint goal.

The goal gives the Scrum team some flexibility regarding the functionality implemented within the sprint. During the second part of the meeting, the team forecasts the PBIs it will achieve and determines their agreed-upon sprint goal. Those PBIs are then moved to the sprint backlog (i.e., the agreed work to be done in the current sprint).

Each day during the sprint, the team meets to synchronize their work and create a plan for the next 24 hours. This is called the daily scrum (or daily standup, as shown in Figure 14.1) and is limited to 15 minutes. At the daily scrum, each team member explains what he/she did since the last meeting, what s/he plans to do today, and what is getting in her/his way preventing her/him from meeting the sprint goal. Solutions to problems are not discussed in the daily scrum. A separate meeting is arranged to dive deeper into problems when necessary.

At the end of the sprint, the team conducts a sprint review activity with key stakeholders to review the product in its current version (referred to as a potentially shippable increment) they have produced. At this review, stakeholders may also identify product improvements (PBIs) that will be placed in the product backlog. At the end of each sprint, the team holds a sprint retrospective activity. The sprint retrospective is an opportunity for the Scrum team to agree on improvement to their way of working, to be implemented in the next sprint.

There are three major roles in Scrum, namely the PO, the Scrum master, and developers. The PO is responsible for feeding the product backlog based on his/her interaction with customers and users. The PO is also responsible for prioritizing the PBIs. The team members (i.e., developers) are responsible for estimating the effort for implementing each PBI.

The Scrum master role is something unique to Scrum. The Scrum master is a servant leader, a person who facilitates the Scrum activities and motivates the team members to follow the Scrum activities.

While the Scrum activities we have just described are fairly simple, teams often tailor them based on their own situations. This is one reason why capturing a team's agreed-to way of working as a set of explicit practices can help team members—especially new and less experienced team members—understand what activities are expected, what options they have in carrying out these activities, and how much detail is expected in any related work products.

14.2 Practices Make a Software Engineering Approach Explicit and Modular

Scrum is in essence a practice, or rather, a set of practices. Briefly speaking, a practice is about doing stuff in a certain way to address certain problems, and with Scrum, it is about teams improving team collaborations and performance. We will take a slight detour to explain how Essence captures practices in an explicit way and thereby provides practical guidance to teams. We will in a short moment capture the Essence of Scrum and demonstrate how Smith's team applied Scrum.

The word “practice” is an overloaded word, meaning different things to different people. The Essence specification provides a specific definition: *a practice is a repeatable approach to doing something with a specific purpose in mind*. A practice provides a systematic and verifiable way of addressing a particular aspect of the work at hand. It has a clear goal expressed in terms of the results its application will achieve. It provides guidance not only to instruct practitioners in what is to be done to achieve the goal, but also to ensure that the goal is understood and to verify that it has been achieved.

As such, a practice provides a proven way of approaching or addressing a problem. It is something that has been done before, can be successfully communicated to others, and can be applied repeatedly, producing consistent results.

14.3

Making Scrum Explicit Using Essence

Scrum can be represented as a practice that is a set of activities to help teams conduct iterative development in a highly collaborative manner. In Part IV of this book, we will show a different way to represent Scrum as a composition of multiple practices: that is, a product ownership practice, a backlog practice, an iterative development practice, and a retrospective practice. Thus, Scrum is not simple, and indeed, the Scrum Guide [Schwaber and Sutherland 2016] calls Scrum a process framework. For the purpose of this part of the book, we choose to use a simplified version of Scrum, which we represent as a single practice that we call Scrum Lite.¹ While our Scrum Lite includes what we have assessed to be the important elements of Scrum, we do not include a discussion on the ideas of Scrum or all of the responsibilities of all of the Scrum roles, nor do we include all of the characteristics of a Scrum Team.

Different practice authors will have different ways to express their key concepts. So, Scrum will be expressed in one way (as depicted in Figure 14.1), user stories another way, use cases another way, and microservices yet another way. These authors haven’t used a common ground, because they haven’t had one. They use different terms for the same thing, and the same term may have different meaning in their practices. Some authors deal with this problem by re-describing what the original author developed, but they don’t use a standard like Essence; rather, they create their own terms. This doesn’t create a collaborative environment between the people who have contributed their ideas to the world. This is one of the serious problems Essence addresses—under Essence, every author uses the same standard

1. Based on version 03.2015 of the Scrum Essentials practice originally developed by Ivar Jacobson International. Used and adapted with permission.

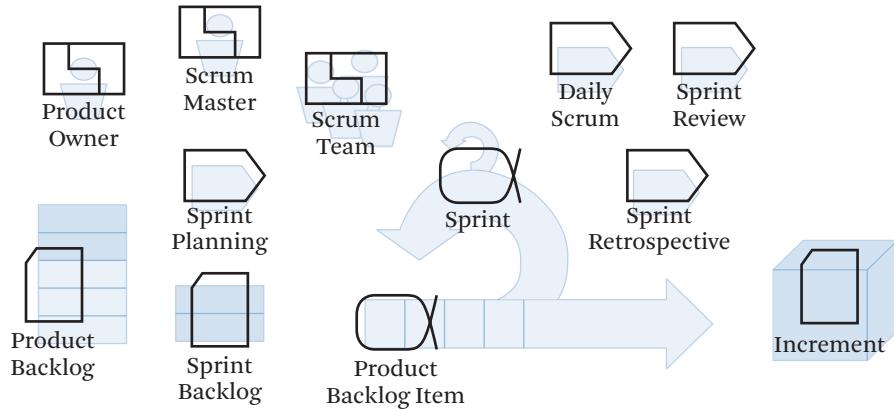


Figure 14.2 Scrum big picture mapped to the Essence language.

language. It becomes significantly easier for students to learn many practices. Once they have learned one practice, they will while learning a new practice recognize that there is a lot in common, even if the new practice covers a different topic than the first practice. And the more practices they learn, the easier it becomes to learn something new.

We can redraw the Scrum Lite big picture in Figure 14.1 using the Essence language, as shown in Figure 14.2. So, what you see in this figure is the Scrum big-picture elements represented using the Essence language symbols. To start, the figure reminds us that PO, Scrum Master, and Scrum Team roles are represented as patterns in the Essence language. And Sprint Planning, Daily Scrum, Sprint Review, and Sprint Retrospective are activities. Product Backlog, Sprint Backlog, and Increment are work products. And finally, Sprint and Product Backlog Item are alphas in the Essence language.

A complete model of the Scrum practice is shown in Figure 14.3. It is a useful diagram in that it shows

1. relationships between the elements in the practice, such as the relationships between Sprint, Sprint Backlog, Product Backlog, PBI, and Increment;
2. activity flows, such as from Sprint Planning to Daily Scrum to Sprint Review and Sprint Retrospectives; and
3. relationships with kernel elements, such as between Sprint and Work, PBI and Requirements, and Increment and Software System.

Thus, Figure 14.3 not only shows the relationships between elements in the practice, in this case the Scrum practice, but also the relationship with the kernel, and

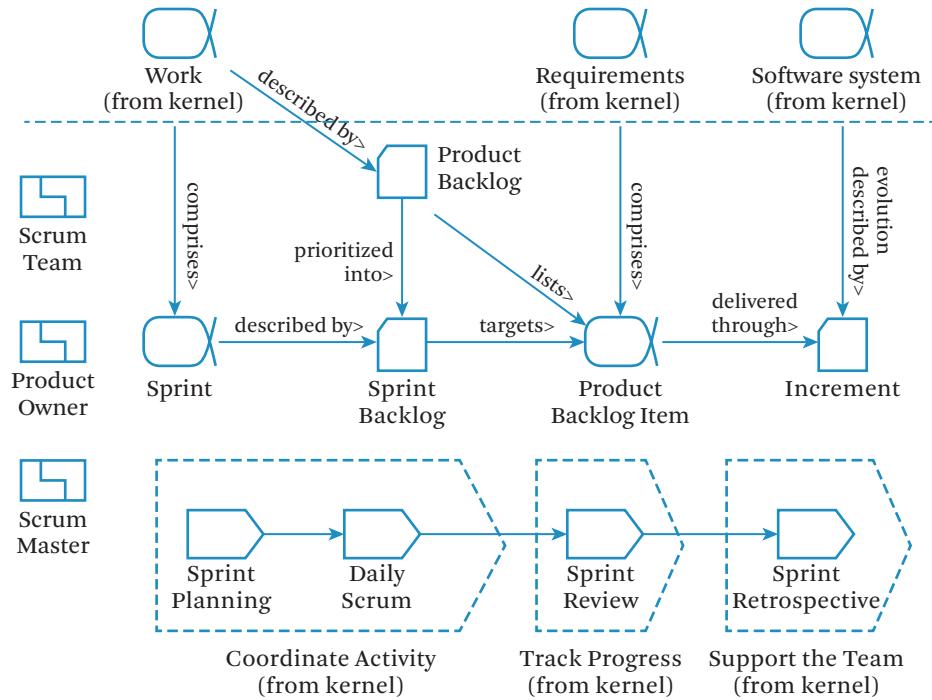


Figure 14.3 Scrum Lite practice expressed in the Essence language.

through that you are better able to understand how Scrum provides more guidance on top of what is available in the kernel.

An important concept highlighted in Scrum is the “Definition of Done” (DoD). The DoD is a clear and concise list of criteria that each PBI must satisfy for the team to call it complete. As an example, a PBI may include DoD criteria such as:

- sufficiently tested;
- accepted by the PO;
- source code checked in; and
- associated documents (e.g., user manuals) updated.

The DoD must apply to all items in the backlog. It can be considered a contract between the Scrum team and the PO. The purpose of DoD is similar to the purpose of the checklists of alpha states. They provide an explicit definition of what the team members must do. DoD as provided through the alpha state checklists is available

for PBI and Increments, and in general any alpha or work product. Alpha states go one step further by defining the completion criteria for each state.

Table 14.1 provides a summary of the elements in the Scrum Lite practice. Before we look in on the TravelEssence team's use of them, we will establish the framework for these elements.

Table 14.1 Elements of Scrum Lite practice

Element	Type	Description
Sprint	Alpha	A time-box (e.g., fixed length of time) of one month or less during which a “Done,” usable and potentially shippable product increment is created. A new sprint starts immediately after the conclusion of the previous sprint.
Product Backlog Item	Alpha	A change to be made to the product (in a future release, for example, a feature, user story, requirement, enhancement, or fix).
Sprint Backlog	Work product	The set of PBIs selected for the sprint, plus a plan for delivering the Increment and realizing the sprint goal. The sprint backlog makes visible all of the work the development team identifies as necessary to fulfill the sprint goal.
Product Backlog	Work product	A priority-ordered list of everything that might be needed in the product: the single source of requirements for any changes to be made to the product. The items in the product backlog are known as PBIs.
Increment	Work product	The sum of all PBIs completed during a sprint and those items completed during all previous sprints. The increment must be “Done,” which means the software system it describes must be usable and meet the Definition of Done (DoD). When a PBI or an increment is described as “Done,” everyone must understand what “Done” means. Although this varies significantly per Scrum team, members must have a shared understanding of what it means for work to be complete, to ensure transparency. This is the definition of done for the Scrum team and is used to assess when work is complete on the product Increment. (Work completed includes only items that meet the team’s agreed-to DoD.)

Table 14.1 (*continued*)

Element	Type	Description
Daily Scrum	Activity	The team meets every day, same time and place, to assess progress, synchronize activity, and raise any issues that are getting in their way and require action to resolve. The meeting is time-boxed, typically to 15 minutes.
Sprint Planning	Activity	Deciding what can be delivered in the sprint's increment and how the work needed to deliver the agreement will be achieved.
Sprint Review	Activity	A time-boxed review of the outcomes of the sprint, used to gather feedback and discuss what should be done next.
Sprint Retrospective	Activity	The whole team meets at the end of the sprint to reflect on its way of working. Improvements are identified and prioritized, and actions agreed. At the next retrospective, the results are evaluated.
Product Owner	Pattern	<p>The PO is responsible for maximizing the value of the product and the work of the development team. How this is done may vary widely across organizations, Scrum teams, and individuals. The PO is the sole person responsible for managing the product backlog.</p> <p>Product backlog management includes clearly expressing PBIs:</p> <ul style="list-style-type: none"> • ordering the items in the product backlog to best achieve goals and missions; • optimizing (maximizing) the value of the work the development team performs; • ensuring that the product backlog is visible, transparent, and clear to all, and shows what the Scrum team will work on next; and • ensuring the development team understands items in the product backlog to the level needed.
Scrum Master	Pattern	The Scrum master is responsible for ensuring that Scrum is understood and enacted. The Scrum master is a servant leader for the Scrum team. Among other things, the Scrum master helps to:

Table 14.1 (*continued*)

Element	Type	Description
		<ul style="list-style-type: none"> • guide Scrum activities; • remove impediments; • ensure everyone understands Scrum; and • make certain all members of the Scrum team understand the need for clear and concise product backlog items.
Scrum Team	Pattern	The Scrum team consists of a PO, the development team, and a Scrum master. Scrum teams deliver products iteratively and incrementally, maximizing opportunities for feedback on how they are doing and self-improvement. The best Scrum team size is small enough to remain nimble and large enough to complete all significant work within a sprint.

14.4

Scrum Lite Alphas

In this section, we will introduce each of the alphas that comprise the Scrum Lite practice. These alphas are the Sprint alpha and the PBI alpha.

14.4.1 Sprint

As mentioned, a sprint is a time-box (e.g., fixed length of time) whereby some useful work is completed.

In the Scrum guide [Schwaber and Sutherland 2016], there are no explicit alpha states defined. However, we find that alpha states are very useful for teams who are new to Scrum. They help teams understand what they must do to prepare for their sprints and for each activity in a sprint. In our Scrum Lite practice the Sprint alpha states are as follows (see also Figure 14.4 below).

Scheduled. The start and end dates for the sprint are scheduled and all team members are aware of the dates; there are sufficient backlog items in the product backlog, and the backlog items have been prioritized.

Planned. The goals to be achieved in the sprint have been agreed, including the specific backlog items within the scope of the Sprint, and the risks have been identified and the team has agreed how to mitigate them.

Reviewed. The outcome of the sprint has been reviewed. This includes reviewing the product increment (such as the recommendation engine), and the team's way of working. This includes reviewing which specific backlog items

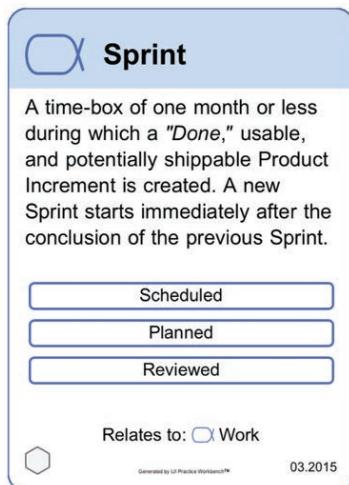


Figure 14.4 The Sprint alpha card.

have been completed, which ones were not completed, why they were not completed, how the team can improve their way of working, and what improvement actions they can take.

In companies, these Sprint alpha states will often be tacit. However, for students new to Scrum, these states and checklists are very useful; new students have not learned the essence of Scrum and can't easily figure it out on their own. By providing the checklists, a less experienced developer would be guided away from making unnecessary mistakes. Figure 14.5 shows the alpha state checklists.

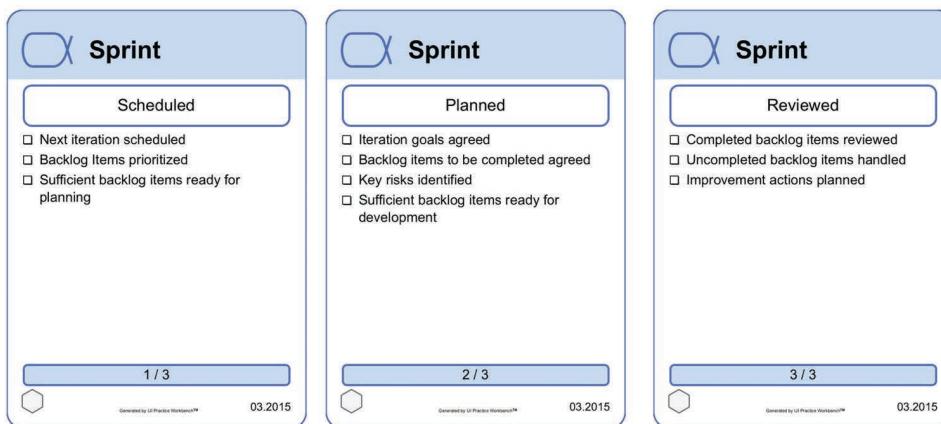


Figure 14.5 The Sprint alpha state cards.

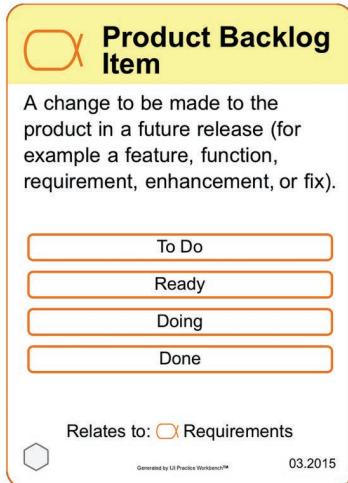


Figure 14.6 The Product Backlog Item alpha card.

These simple checklists are also a good source of information for learning and as reminders even for experienced developers, who can still manage to forget what they previously learned. However, being too prescriptive might prevent the practitioner from thinking on his/her own, and instead treating the checklists as the word of God. Like all things, this is about striking a balance between what should be left tacit and what should be made explicit. In our Scrum Lite practice, we intentionally did not add a fourth state, for Retrospective Completed, to the Sprint alpha. The value of such a state is something each organization can decide based on its own situation, including the competencies of its people.

14.4.2 Product Backlog Item

A Product Backlog Item, or PBI, is a change to be made to the product in a future release (for example, a feature, user story, requirement, enhancement, or fix).

The alpha has the following states identified in Figure 14.6.

To Do. It has been agreed that the PBI needs to be completed within the next sprint. The scope and completion criteria of the PBI are clear.

Ready. The team works together with the product owner to agree on how they should go about completing the PBI.

Doing. At this state, the team is working on the item and bringing it to completion.

Done. The Product Backlog Item has been completed.

14.5 Scrum Lite Work Products

The Scrum Lite practice comprises the following work products:

- Product Backlog,
- Sprint Backlog, and
- Increment.

14.5.1 Product Backlog

A product backlog work product is a priority-ordered list of everything that might be needed in a product. It is the single source of requirements for any changes to be made to the product (see Figure 14.7). The items in the product backlog are known as PBIs.

For endeavors at this point, there is only one level of detail in a product backlog:

Items Ordered. The product backlog Items are captured in the product backlog, which can be in the form of a spreadsheet or within some backlog management tool. They are ordered according to their priority, so that high priority ones can be selected for the next sprint backlog.

For more sophisticated endeavors, there might be more levels of detail. For example, the team might want to describe rationales for prioritizing the PBIs, so that team members can avoid unnecessary debates.

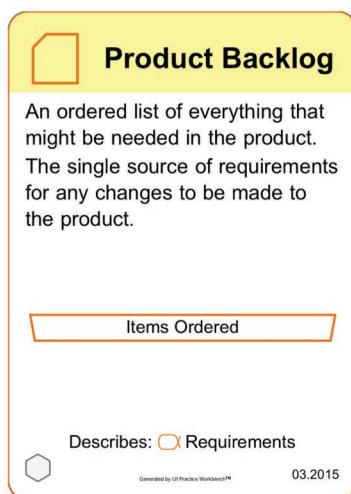


Figure 14.7 The Product Backlog work product card.

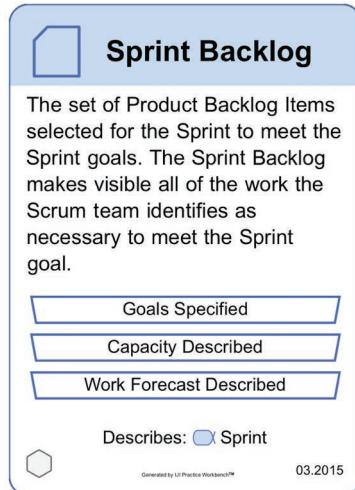


Figure 14.8 The Sprint Backlog work product card.

14.5.2 Sprint Backlog

A sprint backlog work product is the set of PBIs selected for a sprint. It also includes a plan for delivering an increment realizing the agreed-upon sprint goal. A sprint backlog makes visible all of the work the development team identifies as necessary to meet the sprint goal.

The Sprint Backlog comprises the following levels of detail (see also Figure 14.8).

Goals Specified. The sprint goal is clearly stated and sets the target for the team members.

Capacity Described. The amount of work the team can perform is estimated. In this way, the team can determine if it has too much or too little work in the sprint.

Work Forecast Described. The team agrees on product backlog items that can be completed within the sprint, as well as target dates they expect to complete within the sprint.

14.5.3 Increment

An increment is the sum of all product backlog items completed during a sprint and those items completed during all previous sprints.

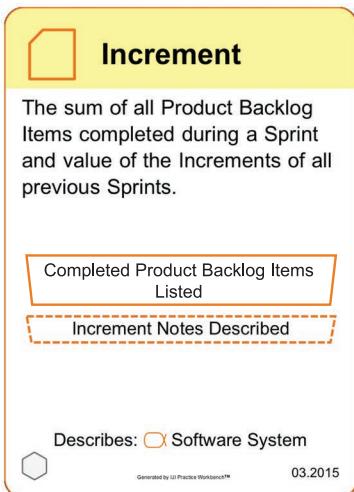


Figure 14.9 The Increment work product card.

The Increment work product has the following levels of detail (see also Figure 14.9).

Completed PBIs Listed. The PBIs that make up the Increment are clearly listed.

Increment Notes Described. Further information about the Increment is provided, such as environments in which the increment can work, known issues, and so on. By environment we mean which browser version, which operating system, and the like. The specific content has to be agreed upon by the team.

14.6

Scrum Lite Roles

Scrum Lite explicitly identifies two roles, namely the PO and Scrum Master. A role is a list of responsibilities that one or more people accept. The individuals serving as PO and Scrum Master and the rest of the team members form the Scrum team. Essence allows you to model roles and team organization as patterns.

In Part I, you learned that Essence provides a concise representation of patterns as poker-sized cards. Figure 14.10 shows the PO pattern card, which comprises the most important information about the Product Owner's responsibilities. The card shows that the PO is responsible for managing the product backlog, ensuring each item is clear to the team members, and making certain that the product backlog is visible to the team. The card also shows that the PO is responsible to ensure

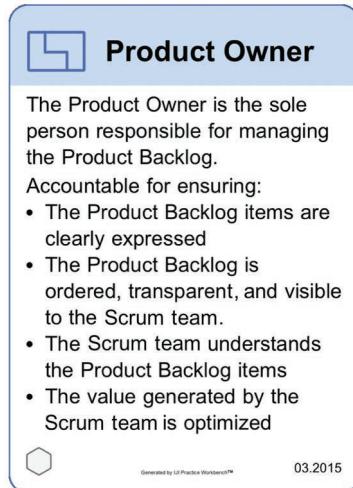


Figure 14.10 The Product Owner pattern card.

the value generated by the Scrum team is optimized, which means the work of the Scrum team provides value toward achieving the goal of the sprint.

The Scrum Master role is also represented by a pattern, as in Figure 14.11. The individual acting as the Scrum Master coaches the team as they conduct the Scrum activities. When team members face impediments, such as unclear backlog items, he/she works to remove the impediment. As an example, Smith’s team faced an impediment with using a library, which required licensing fees. If the licensing were not resolved, Smith’s team might have to rewrite a part of the software. Smith discussed the problem with the company legal representative and finance officer. After discussion, they agreed to use an alternative library that didn’t require licensing fees, which resolved the impediment.

The third pattern in Scrum Lite is the Scrum team, which is a team pattern in Figure 14.12. The Scrum team consists of members, two of which play the roles of a PO and a Scrum Master. Scrum teams are self-organizing, which means no one outside the team tells the team how to achieve the goal of each sprint. The team includes people with the needed competencies to accomplish all the required work. This is sometimes referred to as a cross-functional team.

These cards can be used by Scrum team members by placing them on a board or having them carry them in their pockets where they can be easily accessed as quick reminders of their agreed-to responsibilities.

Thus, Angela agreed to be the PO, and Smith agreed to take on the Scrum Master role for the development team.

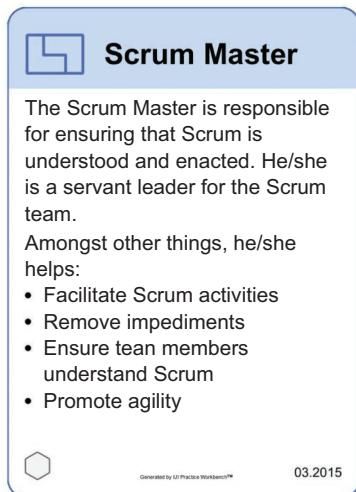


Figure 14.11 The Scrum Master pattern card.



Figure 14.12 The Scrum Team pattern card.

14.7

Kick-Starting Scrum Lite Usage

When Smith's team discussed how they would apply Scrum Lite, there were many questions. Tom, a rather senior and always vocal developer, asked, "How would using Scrum Lite differ from what we did when we delivered the demo? We have been providing demos to Angela on a weekly basis and we adapted our plans based on her feedback."

Smith replied, "Yes indeed, that is true, but Scrum Lite does have some things we can learn that can help us improve the way we are currently working. We did not do Daily Scrums to highlight problems early and improve our team communication. We did not actively manage our Requirement Items in a backlog. We did not define and assign roles with explicit responsibilities, such as Product Owner and Scrum Master. This might have been OK when we were producing the demo for Dave and Angela. But now that we are working toward a live product, and our endeavor has greater visibility to management, we need a way to ensure our team operates with appropriate discipline. We also did not really prioritize our work when we did the demo, nor did we estimate the effort." Making Scrum activities explicit helps team members apply the practices they have agreed to use. The explicit definitions act as reminders to team members. Also, by using Essence to describe practices, we will be able to see any gaps in our practices that we need to fix. This is because when we express practices using the Essence language, we can see which kernel alphas are being progressed, and we can see which ones are not being affected. If an alpha is not being affected by a certain practice, it should lead the team to ask if they have another practice that is helping them progress that alpha. They may or may not need an explicit practice for every alpha, but the team should discuss this and decide based on their specific situation. Once the team has agreed to the set of explicit practices they need, the cards can be used as visible reminders.

Tom asked, "How will the alpha state cards be used now that we are using Scrum?"

Grace replied, "I think one way would be to use the Health Monitor game to help us keep the status of each increment visible to our development team. We can easily incorporate this game into our sprint planning, review, and retrospective activities by keeping the green stickers and red stickers up to date on the board with our alpha state cards.² This can remind us to discuss problems as soon as possible."

All in all, the team members were eager to use Essence and the cards along with Scrum to help learn from their experiences. They agreed to move forward with Scrum as summarized in Table 14.2.

2. Refer to Chapter 10 for information on green and red stickers.

Table 14.2 Adopting Scrum Lite in TravelEssence

Scrum Element	How the TravelEssence team did it
Product Owner	Angela would be the PO for the team.
Scrum Master	Smith would be the Scrum Master for the team.
Sprint	Smith's team agreed that they would iterate on a weekly time-box.
Sprint Planning	Weekly (every Monday morning)
Daily Scrum	Every morning (Tuesday, Wednesday, Thursday)—note that Mondays and Fridays were for sprint planning and reviews, and for retrospectives.
Sprint Review	Weekly (every Friday afternoon)
Sprint Retrospective	Weekly (Friday afternoon after sprint review)

14.8

Working with Scrum Lite

Smith, in his role as Scrum Master, made sure that the activities in Scrum Lite were observed in order to get the team into a working rhythm useful for nurturing good habits.

As we have outlined, working with Scrum Lite involves activities like:

- Sprint Planning,
- Daily Scrum,
- Sprint Review, and
- Sprint Retrospective.

We will further explain these activities as we see how Smith's team performs them.

14.8.1 Sprint Planning

As it does with alphas, work products, and patterns, Essence also presents the key information about activities through poker-sized cards. Figure 14.13 shows a Sprint Planning activity card.

Smith's team started using Scrum on top of Essence on one Monday morning with Sprint Planning. This was all about deciding what priority items from the Product Backlog should go into the current Sprint Backlog.

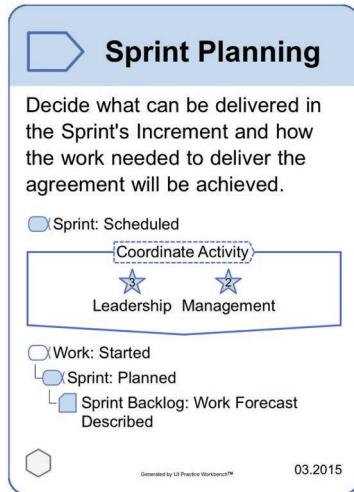


Figure 14.13 Sprint Planning activity card.

But this meant more than just picking various items from the Product Backlog and moving them to the Sprint Backlog. It was important for the team to ask some critical questions about the items they were considering.

- “Are the items we’re selecting for this sprint properly prepared?” Properly prepared means that the items are broken down into small enough sub-items to be completed by the team within the time available for the next Sprint. This means there must be enough information about each selected backlog item for the team to estimate the effort needed to complete it. You might be wondering how the team knew to ask this question. Some of the questions the team asked may come from an explicit practice defined on top of the kernel, such as the Sprint Planning activity that is part of the Scrum practice. However, other questions may be based on the kernel itself. For example, note that a Sprint is a sub-alpha to the Essence Work alpha (refer to Figure 14.4). The Work alpha contains a state named Prepared. Achieving this state means all pre-conditions for starting the work have been met. One of the checklists in this state includes the item “The work is broken down sufficiently for productive work to start.” This checklist item should remind team members to ask if the backlog items have been properly prepared, regardless of the practices they have agreed to use.

- “There are situations when such questions as above cannot be answered due to unclear/insufficient/ambiguous information in the backlog. An indication for this might be that the team is not able to reliably estimate the effort.” This meant the team needed to go back to the product owner and get more information, or reject that item as not being ready to be worked on in the sprint.
- “Has the team considered their capacity when deciding if they can commit to the proposed items to complete in this sprint?” This question comes specifically from the sprint planning activity within the Scrum practice. This means each team member needs to consider how much time they have to work on the endeavor and if they believe that is enough time to complete the items the team is committing to complete in the sprint. Note that in Figure 14.12 we saw that the Scrum team has the responsibility to be self-organizing. This means they are responsible for estimating and committing to the work to be completed within a sprint. If the team had chosen to use a different practice than Scrum, this responsibility might have fallen on a different person (for instance, a manager) in the organization. This is because another practice might define its roles differently than Scrum.

The preceding questions provide good rationale why some teams need explicit practices and some don’t. Teams need to ask questions such as the following.

- “Are our team members experienced enough to know to ask key questions?” If the team does not have adequate experience, they need to decide if other team members can help, or if they need to ask for more help from outside the team. “Do they need explicit reminders, such as checklist reminders, to conduct a proper disciplined sprint planning session?” The answer to this question often depends on how many people on the team have previous experience conducting Sprint Planning. In some cases, explicit practices may be sufficient, but if most or all of the team members are new to Scrum, then a coach may also be needed to guide them through their first few Sprints. Explicit practices can provide these critical reminders to teams that have less experienced practitioners or who have never worked together before, but sometimes a coach is needed to properly set the expectations of their new team.

Both Product Backlog and Sprint Backlog have simple poker-sized cards to describe them. Figure 14.8 shows a Sprint Backlog work product card, which identifies what a team agrees to work on for a sprint. Note that this card can also provide

Table 14.3 Product Backlog

Item	Product Backlog Item	Originator	Priority	Estimate
1	Set up group of internal users	Angela	High	2
2	Add toggle for recommendations feature based on user group.	Angela	High	1
3	Run series of tests to ensure the toggle for recommendations feature does not result in performance degradation.	Angela	High	2
4	Add introductory user screen for recommendation functionality.	Angela	Medium	3
5	Develop the pre-processing data needed for the recommendation algorithm.	Smith	Low	5

reminders of activities the team should be conducting while producing the work product. For example, the team needs to agree on the goals for the sprint, and it needs to estimate its capacity when it is developing its sprint backlog. In the past, often there was considerable effort put into producing lengthy documentation that wasn't used. This often occurred because practitioners tasked with producing this documentation did not have clear guidance on what should be included, along with how much detail. Work product cards provide a simple way to communicate what practitioners need to know to produce useful documentation that achieves the intent of the work product.

As mentioned earlier, the team's first target was to work toward a version of the system suitable for internal users. Angela participated as the Product Owner and she quickly identified a number of product backlog items, as shown in Table 14.3.

Since Smith's expertise and background was in the technical solutions area, he identified some key items on the product backlog, focusing on the technical issues. He said, "We need to get going on the recommendations and the user screen. These are both high-priority items." This was not meant to imply that they only needed technical issues on the backlog. As we have explained earlier, the product backlog should include all work the team has to do from the stakeholders' perspective. This will make the Product Backlog the "single source of truth" for the whole team. Everything that the team might ever need to do must eventually be added to the Product Backlog. However, at this point in the TravelEssence endeavor, the product backlog was not a complete list. It contained a number of items the team knew they

needed to do, but certainly not everything needed to create a new release. This is the way many endeavors get started.

The product backlog evolves as a team progresses through the sprints. The team learns more about what requirements are needed as the endeavor progresses. This is how many endeavors evolve. The TravelEssence team knew from their previous work with Essence (during the internal demo) the value of the alpha checklists, especially in regard to helping them get their Work to the Under Control state. As shown above, for example, checklists from the Work alpha had already reminded the team of the need to make sure they had sufficiently broken the work down. This helped the team confidently estimate the work to fit within its agreed sprint.

During the sprint planning session Joel said, “I don’t think we have enough information to estimate the work involved in Item #2. Angela, can you explain more about what you mean by the toggle for recommendations based on user group?” After further discussion, the team felt they understood that all Angela wanted was a way to turn the recommendation on or off for particular user groups. This conditional check could be easily achieved just by adding several lines of code. Thus, they now understood the backlog item well enough to make an estimate (see Table 14.3).

What we have just described demonstrates how, by applying the Sprint Planning activity that is part of the Scrum practice, the team members were better able to understand the work products.

14.8.2 Daily Scrum

The Daily Scrum is a simple activity that Scrum teams conduct every day. There are only a few guiding principles required (Figure 14.14), such as keeping the meeting to 15 minutes, having only the developers speak, and keeping the focus on answering the three main questions (what did I do since the last daily scrum, what do I plan to do next, and what obstacles am I facing).

When conducting the daily scrum, Smith’s team met at the same time and same place each day. Keeping the meeting to just 15 minutes forced Smith’s team members to be as brief as possible, and focused the team just on answering the three questions. As each team member answered the questions, the others listened, and if they heard something with which they could help, they agreed to talk further after the meeting with just the smaller group that needed to be involved in the discussion. In this way, team communication was enhanced while minimizing the time lost by all team members in attending the meeting. When someone identified an obstacle that was keeping them from getting their work done, Smith, the Scrum Master, accepted an action to work the issue, but sometimes other team members stepped up and helped when they knew how to solve the problem. Still, they didn’t

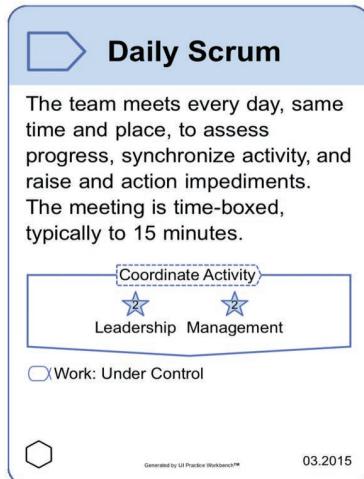


Figure 14.14 The Daily Scrum activity card.

discuss the solution in the daily scrum because they didn't want to take up the time of the other team members who didn't need to be involved in solving the problem. The daily scrum helped the team keep the Work under control. (In Figure 14.14, at the bottom of the card, the result of the activity is that the Work alpha will be in the Under Control state. When no state is specified as input to the activity it means that the daily scrum is done in the same state.)

During one daily scrum, Tom said, "I have been working since our last daily scrum on Item #4, the introductory user screen, and I am having some trouble getting it to work." Joel replied, "Tom, I have worked on introductory user screens before, so I will give you some help right after the daily scrum." From an Essence perspective, we can see from Joel's reply how their daily scrum is helping to progress the *Team* alpha's Collaborating state checklist items: "The team is working as one cohesive unit," and "Communication within the team is open and honest."

The value in documenting these simple guidelines for a daily scrum as checklist items in an activity is that they serve as reminders to the team that can help them conduct the daily scrum consistently. These checklist items can likewise be used during training and coaching sessions. They are also useful for bringing new hires on board.

14.8.3 Sprint Review

The Sprint Review is a review of the product by the stakeholders (see Figure 14.15). The focus of this review should be on demonstrating what the team produced based

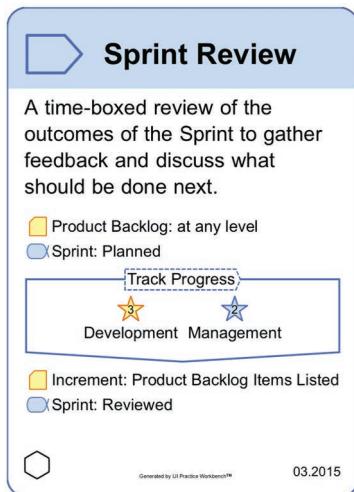


Figure 14.15 The Sprint Review activity card.

on what they committed to produce at the previous sprint planning session. At TravelEssence Angela, as the product owner, led this meeting, but she was supported by the other team members who worked on the changes related to the current sprint. To keep the meeting focused, Angela started each sprint review by going over the sprint backlog items that the team agreed to work on during the previous sprint. Then each item was demonstrated, and Angela asked the stakeholders in attendance if they agreed that each sprint backlog item that was committed to was achieved. Only sprint backlog items that were completed during the sprint were demonstrated. If something was partially completed, its demonstration was put off until the next sprint, when it could be fully demonstrated. Scrum teams do not take “partial credit” for completing part of a backlog item. If committed sprint backlog items are not completed, the product owner explains this during the sprint review and explains the plan to address the missing item. The sprint review is also an opportunity for the team members to get valuable feedback from the stakeholders. This occurs primarily at the end of the sprint review, when the product owner asks the stakeholders if they feel that the goal of the sprint was achieved. However, often stakeholders provide feedback throughout the review. In this case, at the end of the sprint review the PO summarized the result of the review and the actions the team was taking out of the review to address in a future sprint review. Input to the review is the product backlog that is used to ensure the review focuses on the items the team committed to complete. The card also shows that sprint alpha

is in the Planned state prior to the Sprint Review activity, and is progressed to the Reviewed state as an output of a successful review. Moreover, it shows that the increment work product is updated with new product backlog Items as an output of the review.

Often, when Scrum teams operate with only tacit practices, the sprint review can lose its focus, with stakeholders bringing up issues that were never planned as part of the sprint, or team members discussing the method they are following rather than the product they have produced. The value in adding a simple Sprint Review activity, or at least adding checklists, is that these checklist reminders can help the team to recall their agreed-to activities related to the sprint review. It can also help to bring new people on board, similar to what we just discussed with the daily scrum.

One Friday at TravelEssence, Angela, as the PO, started out the sprint review by explaining to the stakeholders who had come—Cheryl and Dave, since this sprint was focused on an internal release—the product backlog items that the team had committed to for this sprint, and what they were going to see demonstrated. Angela then asked Grace, Tom, and Joel to demonstrate what they had accomplished during the sprint. At the conclusion, Angela explained how they had not been able to fully implement the user screen because they ran into a few issues, but they felt they did meet their commitments for achieving the toggle for recommendations. She then asked the stakeholders in attendance for their feedback, and whether they thought the goal of the sprint had been achieved.

It is worth noting that often explicit practices and their associated activities can help teams progress multiple Essence alphas at the same time. For example, the sprint review activity just discussed, as part of the Scrum Lite practice, helped the team progress the Essence Stakeholder alpha's Involved state, checklist item "The stakeholder representatives provide feedback and take part in decision making in a timely manner." This was seen with the involvement and feedback provided at the sprint review by the stakeholders Cheryl and Dave. As another example, we can also observe from the sprint review activity how the Essence Work alpha's Under Control state, checklist items "Estimates are revised to reflect the team performance" and "Measures are available to show progress and velocity" were achieved by the team's discussion with the stakeholders on the issues they encountered and the tasks they successfully completed.

14.8.4 Sprint Retrospective

The purpose of a sprint retrospective is for the team to review how they are doing on their endeavor from the perspective of their agreed-to method, and to agree to

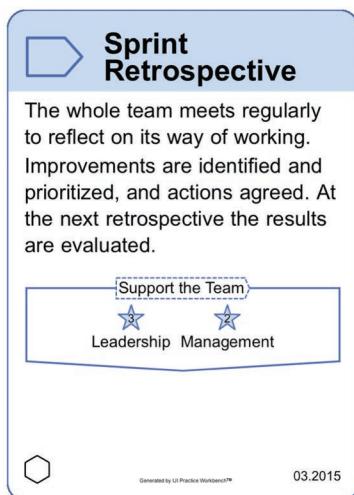


Figure 14.16 The Sprint Retrospective activity card.

improvements to their method to implement in the next sprint. The results of these improvements can be tacit or explicit, which means they may or may not require changes to practice descriptions.

There are many techniques available for conducting sprint retrospectives. There are even books that have been written just focused on this subject alone [Derby and Larsen 2006]. The value of the Sprint Retrospective is to get feedback from your development team on what is working well and what isn't working well, and get agreement with the team on what they can do differently during the next sprint to improve their method (see Figure 14.16). The Sprint Retrospective can also help to guide teams with decisions related to how to move forward with the team's suggested improvements.

A sprint retrospective could be represented in the Essence language as an activity within a larger practice, such as Scrum, or as a practice itself. For example, many organizations break their retrospectives out as a separate practice and include in the practice criteria to help teams select practical improvements that can be implemented within the next sprint. An example of such criteria are referred to as the SMART criteria, which stands for Small, Measurable, Attainable, Relevant, and Testable. These attributes are intended to be used by teams to help them assess if their agreed to improvements can be implemented within the next sprint. For instance, a team member could ask the following questions in regard to a proposed improvement.

- Is this improvement *small* enough for the team to implement it the next sprint?
- Is this improvement *attainable*? If all the team members do not have the needed skills to implement the improvement, then it might not be attainable.
- Is the improvement *relevant*? There are lots of improvements that teams could decide to make that might not be relevant to helping the team achieve their goal.
- Is the improvement *testable*? We need to know how we are going to test the improvement to say whether we achieved it or not.

In our TravelEssence case, Smith started the sprint retrospective by asking all the team members to jot down on yellow sticky notes things they thought went well during the sprint and things they thought didn't go well and could be improved. Smith then collected all sticky notes and grouped the ones that seemed to be related under larger sticky notes that said "working well" and "not working well." He then highlighted to the team that everyone felt the team was working well with respect to communication. (It is a good idea to always start a sprint retrospective by sharing with the team what is working well so the team doesn't feel like they are always just focusing on negative things.) Smith then moved over to where he had grouped a number of yellow sticky notes under the heading "not working well." He pointed out that a number of team members felt that the team could do better in future sprint planning sessions.

Smith said, "Grace, one of your sticky notes says 'sprint tasks unclear'. Could you explain what you mean by that?" Grace replied, "Sure. I don't think we are breaking our sprint tasks down enough and describing enough about what needs to be done to complete each task." The team then used the Essence Work alpha's Started state, checklist item "The work is being broken down into actionable work items with clear definition of done" to stimulate their retrospective discussion related to breaking work down. Tom said, "I agree with you, Grace. I think we need to spend a little more time discussing and agreeing what 'Done' means for each of our tasks before we commit to the sprint work." The team agreed that this would be an improvement area they would work on during the next sprint. Specifically, they made it a point to break down the work into items each with clear acceptance criteria. This was one of the reasons why they moved to user stories, which will be fully described in Chapter 15.

14.9

Reflecting on the Use of Scrum with Essence

As we saw in Part II, Essence by itself can help teams who are experienced, or are working on a simple endeavor where their practices are tacit (i.e., not written down). When your endeavor is more complex or more people are involved who have never worked together, then there is increased risk of miscommunication of what activities the team members are expected to carry out and the degree of detail expected in work products produced. There is also increased risk that different team members will assess their progress and risks differently, leading to confusion and inaccurate progress reporting to stakeholders.

14.9.1 Adding Explicit Practices

Scrum essentialization helps you to focus on the essentials of Scrum in two important ways:

1. Calling out the most important parts of the practice.
2. Making explicit what these important parts are.

For example, there are many books written about Scrum, but how do you explain Scrum quickly to a team new to Scrum? You will inevitably need to choose what to talk about and what to ignore. It is worth repeating that in this chapter, we have done that, and we call what we want to talk about Scrum Lite. Next, you have to describe the result in a way that is easy to understand and not misinterpreted by the team. We do that by essentializing Scrum Lite. Recall that when we use the term “essentialized” we mean you have described your practice using the Essence kernel and language.

Thus, an essentialized Scrum Lite can provide guidance in the essentials of conducting certain activities such as Sprint Retrospective and Sprint Review. There are many different ways teams can conduct a sprint retrospective [[Derby and Larsen 2006](#)] that goes beyond what is essential. As an example, they can ask each team member to write down what went well and what didn’t go well during the last sprint. This helps the team decide where they can improve for the next sprint.

14.9.2 Visualizing the Impact of the Scrum Lite Practice

Let’s see how essentialization helps teams see more clearly where gaps exist in their own practices, using the Essence kernel as a reference. This leads them to see where specific new explicit practices may be needed or improvements made to their existing explicit practices.

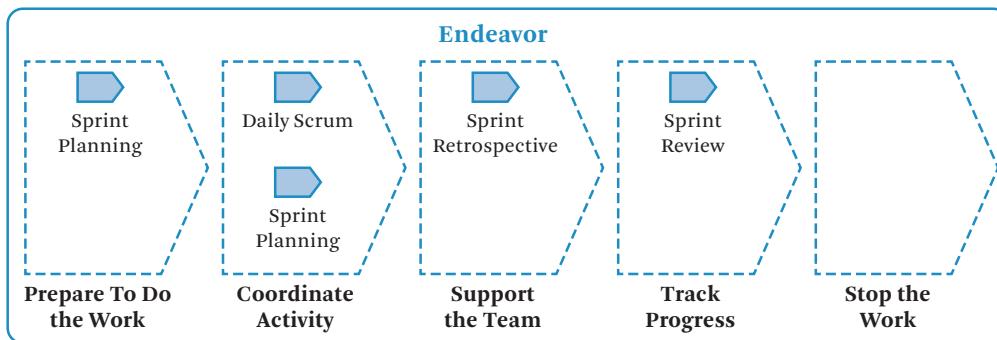


Figure 14.17 Endeavor activity spaces partially filled with Scrum Lite activities.

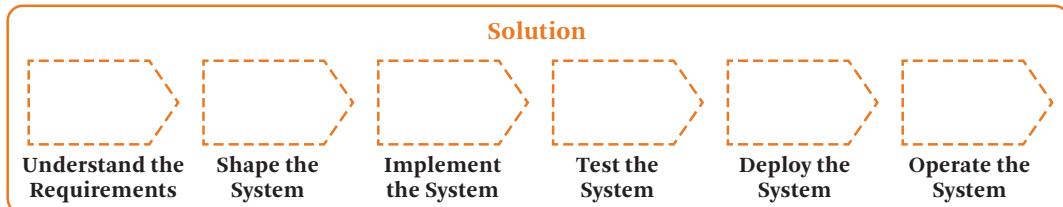


Figure 14.18 Solution activity spaces (not addressed by Scrum Lite).

To demonstrate what we mean, Figure 14.17 shows all the activity spaces in the endeavor area of concern. It also shows the activity spaces that are populated by Scrum Lite activities. Note that the Sprint Planning activity occupies two activity spaces: Prepare to Do the Work and Coordinate Activity. Sprint planning has the dual purpose of creating an initial product backlog and a sprint backlog from which the team can start working, and iteratively updating the product and sprint backlogs to keep the team going.

As can be seen, Scrum Lite only occupies four out of the five activity spaces in the endeavor area of concern. In particular, Stop the Work is empty. This activity space is about concluding the work. These are outside the scope of the Scrum Lite practice and thus show the team where they have gaps.

Scrum Lite also does not provide any guidance on other areas of concerns (i.e., the customer and solution areas)—nor does Scrum. As an example, Smith and his team did have issues with how to work effectively with regard to activities in the solution area of concern (see Figure 14.18) and hence needed some explicit guidance. Because the team quickly observed gaps in their requirements practice, it led them to apply further practices: User Story Lite, and then Use Case Lite.

14.9.3 Value of Being Precise

The idea of practices is not new. It has been around for maybe 50 years. As we explained early in Part I, and will review at length here, in the past, we have seen two major problems. First, practices have been applied in an ad hoc manner. Sometimes different authors use different words to mean the same thing. Or perhaps, a single word has more than one meaning, or two words may overlap, but have some major discrepancy at the same time. This meant that practitioners had to relearn the vocabulary when moving from working with one practice to working with another. What the Essence standard attempts to do is to eradicate this kind of unnecessary confusion. The second problem is that sometimes practices and methods are too rigorous. For example, some quite successful methods were described using thousands of pages. The intention was good, but it didn't work in practice. There is a law of nature: "Most people don't read big books." Thus, when developing Essence, the following points were considered for how individuals in the team/organization should learn and improve.

1. Focus on the essentials. This is the gist of a practice, maybe 5% of what an expert knows of the practice but enough to participate in a team.
2. Use a very simple intuitive visual language to describe guidance of practices, alphas, activities, etc.
3. Provide a simple way to access additional guidelines, for example using links to books, teaching materials, and tools from the essentials.
4. Keep practices separate but make them composable with other practices to form methods.
5. Keep the practices in a library and let the teams that develop software improve them easily and quickly when they do retrospectives.

Regarding the first three points above, what has been done is to be precise and explicit, and to give the essentials a certain level of rigor using the Essence language. The goal is to facilitate more effective communications and guidance, so that team members can have more time to do the actual work.

This precision is made possible not only through the language and its usage but also by focusing on the essentials. Thus, practice descriptions following this approach are very concise and intuitive to practitioners. It serves as a reminder, and helps teams get started with conversations. The Scrum Lite practice is just 12 cards (see Figure 14.19), one card for each element in Figure 14.3.

With an understanding of the Essence kernel as presented in Part II, a practitioner only needs to have a small deck of 12 cards to begin his/her journey under-

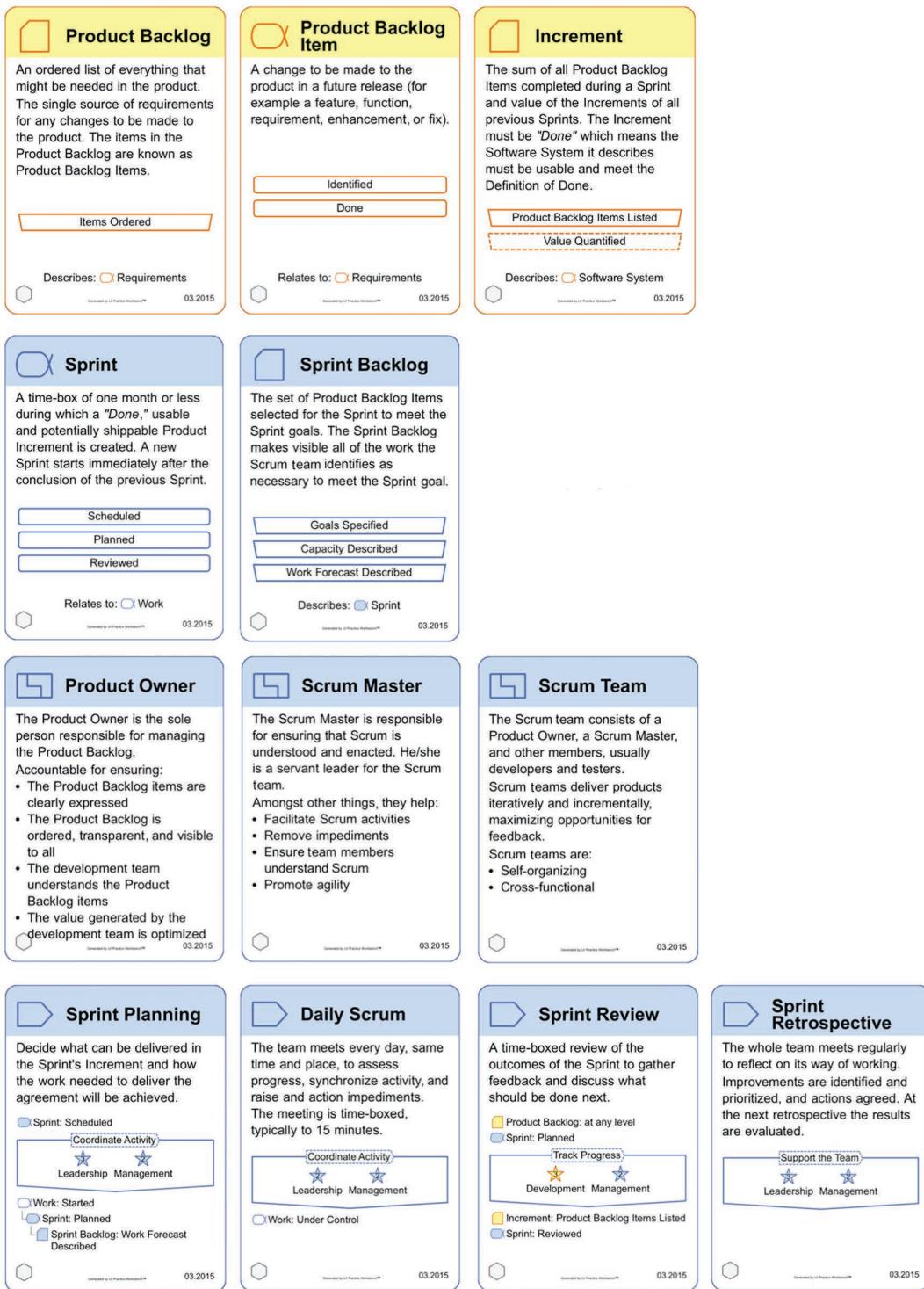


Figure 14.19 Scrum Lite as a deck of 12 cards.

standing, learning, and applying a new practice. If you want to give more guidance to the practitioners, more detailed descriptions can be made available. Associated with each card there could be a 2–4 page document with further details, guidance, hints, tips, and common mistakes. All the cards mentioned in this book are available for download on the book’s website at <http://semat.org/web/book/>.

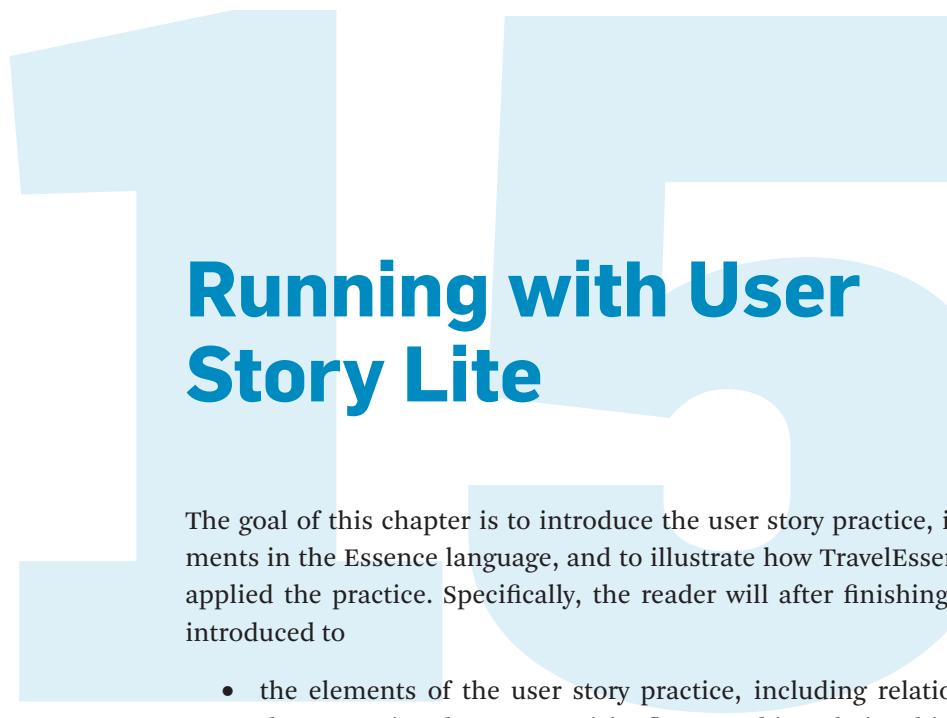
Regarding the last two points above, Essence provides a mechanism for teams to select and compose practices. Our work with practices has yielded a sizeable set of them. We will discuss this further in Part IV, when we talk about how Essence can scale to large organizations.

So again, our recommendation to software engineering students is to learn the kernel, learn the language, symbols, and agreed terms, to lay the foundation for learning and comparing the multitude of practices in software engineering. In the remainder of this part of the book, we will demonstrate how Smith’s team—with the help of precise and explicit practices—embarked on their journey to successful software delivery and personal growth.

What Should You Now Be Able to Accomplish?

After studying this chapter, you should be able to

- explain the benefits of the Scrum practice;
- explain how TravelEssence adopted and applied Scrum and what benefits they achieved, together with the benefits implied by using the Scrum practice in an essentialized form;
- explain why organizations often mandate specific practices and tools;
- list and explain the alphas, work products, activities, and patterns of Scrum;
- explain the concept “Definition of Done” used in Scrum;
- apply Scrum Lite practice;
- name relevant questions to ask during development with Scrum (e.g., “Were the items selected for this sprint properly prepared?”); and
- explain SMART criteria for a practice (what do the letters stand for?).



Running with User Story Lite

The goal of this chapter is to introduce the user story practice, including its elements in the Essence language, and to illustrate how TravelEssence adopted and applied the practice. Specifically, the reader will after finishing this chapter be introduced to

- the elements of the user story practice, including relationships between the respective elements, activity flows, and its relationship with the kernel elements (in TravelEssence's case, only Requirements and Work);
- quality criteria for each user story and the decisions they drive along the practice;
- the elements and structure of a simplified version of the user story practice (called User Story Lite) in a real endeavor, including the obstacles and challenges that might arise; and
- the coverage of kernel solution activity spaces by the User Story Lite practice.

In this chapter, we describe how Smith's team started to apply user stories in their work. User stories have the benefit of getting the team to think, inquire, and understand the value of what they do from the point of view of their users. The user story practice is a popular practice, in particular for small teams. It originated from Extreme Programming (XP), a lightweight, efficient, low-risk way to develop software [Beck 1999]. XP was in turn inspired by use cases from 1992. The User Story Lite practice is a simplified version of the user story practice, created just for the readers of this book.¹

1. Based on version 2017.01 of the User Story Essentials practice originally published by Ivar Jacobson International, © 2015–2017 Ivar Jacobson International SA. Used and adapted with permission.

15.1

User Stories Explained

A user story [Cohn 2004] describes functionality in the system we are building that is valuable to a user of a system. User stories are based on an approach that was proven successful back in the 1990s and earlier, where, rather than write lengthy requirements documents, informal discussions were conducted between the user of the system and the developer. A user story includes a written description that is utilized when discussing the story, along with tests to help communicate what is needed to complete the story. By complete we mean everything that has been agreed upon that will achieve the user's need. The idea of user stories is to provide a way to facilitate discussion to help clarify who a piece of functionality is for—i.e., a role—and how it benefits that role. A user story is often captured on a 3×5 index card with a very concise format or template as follows:

As a <role, or type of user>, I want to <list here the function you want the system to do>, so that <list here the objective you want to achieve>.

An example could be: “As a bank customer I want to have a direct deposit capability so that my employer can electronically send me my paycheck.” This template helps to ensure that the “Who,” “What,” and “Why” are all considered and captured:

- Who will get the value?
- What do we need to achieve?
- Why are we doing it?

(Note that this concept of role is different from the concept of roles we defined earlier (Section 14.6) where a role meant a list of responsibilities that one or more members of the team accept. The role of a bank customer within a user story is with respect to the system being developed, whereas roles within the Scrum Lite practice such as Scrum Master and PO are with respect to a development endeavor.) User story cards, of course, do not provide everything that a user needs. They are placeholders used to remind the team of the need to conduct conversations with the users. The purpose of the conversations is to flesh out the details. These additional details can be added to the card, or they can be captured through additional stories. Again, the primary value of user stories is that they get a conversation going between the development team and the user.

Figure 15.1 shows the idea of applying user stories, and a simple way of remembering what a user story comprises.

Card. A succinct headline description, as captured on a story card.

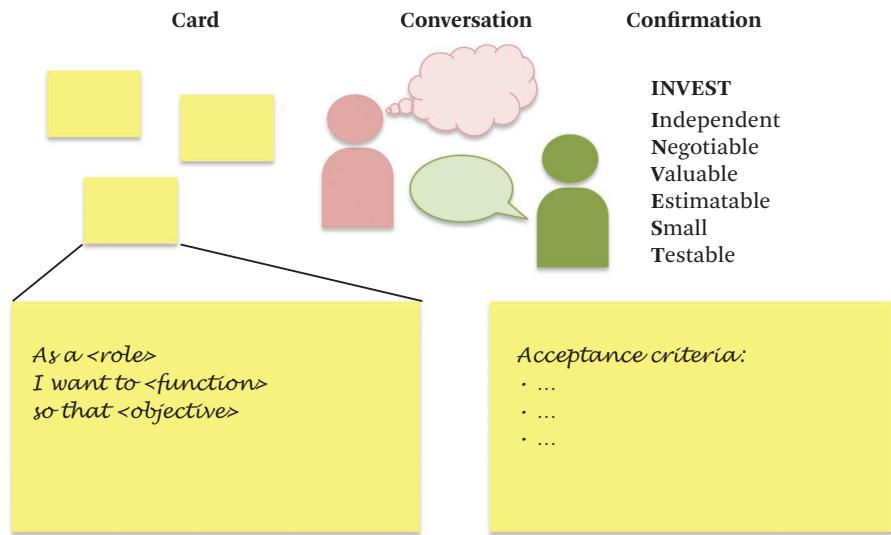


Figure 15.1 User Story practice big picture.

Conversation. The discussion between actual users of the proposed system and developers about what is needed to converge on the best solution.

Confirmation. Acceptance criteria, captured as bullet-point statements, which can be captured on the back of the story card.

To write a good user story it is useful to apply the INVEST criteria, which is an acronym for Independent, Negotiable, Valuable, Estimatable, Small, and Testable. Each of these six criteria items is discussed below.

Independent. User stories should be *independent* of each other so they each can be developed separately.

Negotiable. At least part of the reason for promoting a conversation when using user stories is to support give and take between the user and developers. To do this, user stories should be written in a way that allows them to be *negotiable*. Negotiation promotes understanding and commitment.

Valuable. A user story should be *valuable* to the user. The conversation can help team members understand the real intent of a requirement and the value each story brings to the user. One way to help ensure each story has this value is to engage the user in actually writing the story.

Estimatable. A user story should be *estimatable*. As team members and users work together on user stories, the goal is for enough details to emerge to

allow the developer to estimate the work effort required to implement the story.

Small. User stories should be *small*. Often, when stories are first written they are too large to fit within a given iteration and therefore must be split into smaller stories. These large stories that are too large to fit within an iteration are often referred to as epics. Through the conversations held between developers and users, the needed smaller stories emerge and are agreed upon.

Testable. An important criterion to keep in mind for a good story is that when completed it should be *testable*. Writing the tests first help ensure the story is testable and helps ensure both the user and the developer are in agreement on what it means to complete the story.

One question that often arises for beginners when using user stories is:

But why do we need the “so that” clause in a user story?

One of the reasons the “so that” clause is added to this format is so the developers understand the end objective of the user. This helps to support evolutionary requirements development, by which we mean that the requirements may evolve as we learn more about the available options and needs of the user. This also keeps the developer’s options open in providing alternative solutions. Refer to Figure 15.2 and Table 15.1 for a summary of User Story Lite practice.

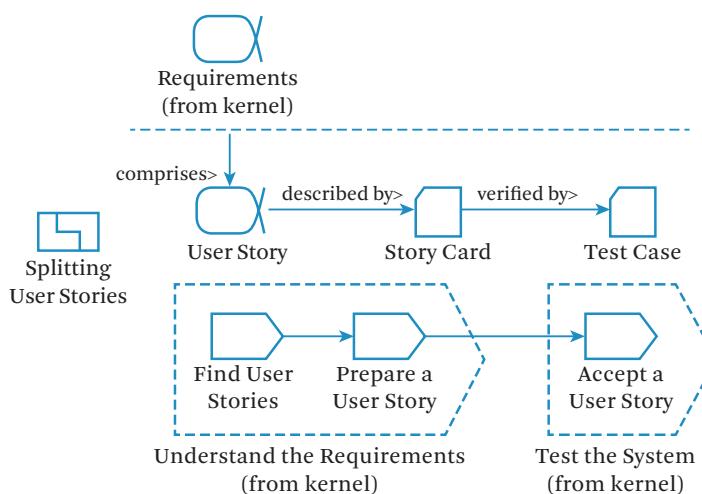


Figure 15.2 User Story practice expressed in the Essence language.

Table 15.1 Elements of User Story Lite

Element	Type	Description
User Story	Alpha	Something that a software system could be extended to do, expressed in terms of the value that it will provide to a user of the system.
Story Card	Work Product	An index card, or equivalent, that captures the essential details of a user story.
Test Case	Work Product	Defines test inputs and expected results to evaluate whether a user story is fully and correctly implemented.
Find User Stories	Activity	Identify things of value that a software system could do. Capture these as simple and succinct headline descriptions on story cards.
Prepare a user story	Activity	A user story is prepared for development by discussion with users to build understanding and refinement of its acceptance criteria and test cases.
Accept a User Story	Activity	The user story implementation is evolved in close collaboration with the customer/user until it is acceptable to and accepted by the customer/user representative.
Splitting User Stories	Pattern	Small things get done faster. In agile development there is a continuous and relentless drive to reduce the size of user stories by splitting bigger stories into smaller ones. The key is to ensure that each story delivers value: * Splits should support meaningful user interactions, no matter how small or “specialized” (think “thin” end-to-end journey with each split providing value to the user).

15.2

Making the User Story Lite Practice Explicit Using Essence

Just as we did in the previous chapter on Scrum, we can be very explicit about how the user story practice guides the team by understanding how user stories and various elements surrounding user stories are related. Figure 15.2 expresses the user story practice using the Essence language.

From Figure 15.2, it is clear that this practice is a way to decompose complex Requirements into sub-alphas—the User Story alpha. Each user story is described

by a story card and is verified through a test case. The User Story Lite practice has several activities:

- Find User Stories;
- Prepare a User Story; and
- Accept a User Story.

We will exemplify how Smith's team applies these activities shortly. Figure 15.2 also shows one pattern, Splitting User Stories, to help teams ease development.

When you compare this with Scrum Lite in Chapter 14, it is obvious that this User Story Lite practice is simpler than that of Scrum Lite. Not only does User Story Lite have fewer elements than Scrum Lite, it also relates to fewer elements in the kernel: in this case, only the Requirements alpha. Thus, a team applying a user story practice alone should consider other practices that provide explicit guidance on how to progress the other kernel alphas, such as Opportunity, Work, etc.

15.3 User Story Lite Alphas

15.3.1 User Story

A user story is something that a software system could be extended to do, expressed in terms of the value that it will provide to a user of the software system.

A user story usually progresses through the following states (see also Figure 15.3).



Figure 15.3 User Story alpha card.

Identified. The user story is identified with its value clearly expressed. It is placed in the team's product backlog.

Ready for Development. The team discusses the details of the user story such that members are clear on what is involved in fulfilling the requirements behind the user story. This might involve details about user interfaces, implementation details, and so on.

In Progress. At this state, the team is working on fulfilling the user story.

Verified. The user story is verified by a qualified user representative, such as a product owner.

15.4

User Story Lite Work Products

The work products in the user story Lite practice are the Story Card, and the Test Case for each user story.

15.4.1 Story Card

A story card is an index card, or equivalent, that captures the essentials of a user story.

A user story can be expressed at different levels of detail.

Value Expressed. The value of the user story is clearly expressed, such as using the common format described above.

Acceptance Criteria Listed. The acceptance criteria for the fulfillment of the user story are clearly expressed.

Conversation Captured. The discussions the team has about the user story are captured so that the team understands more clearly the requirements for the user story and the rationale behind its details. These discussions are usually verbal, but can be written on the story card itself or recorded by some electronic means (see Figure 15.4).

15.4.2 Test Case

A test case defines test inputs and expected results to evaluate whether a user story is fully and correctly implemented.

A test case has several levels of detail (see also Figure 15.5).

Acceptance Criteria Captured. The different possible ways for testing the user story are captured.

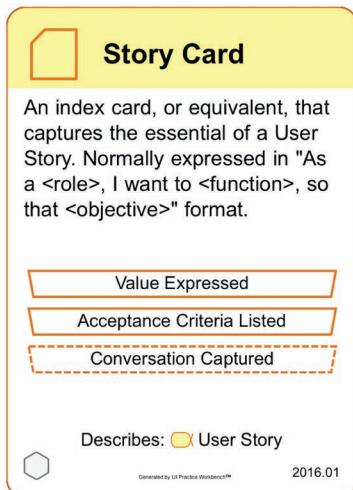


Figure 15.4 Story Card work product card.

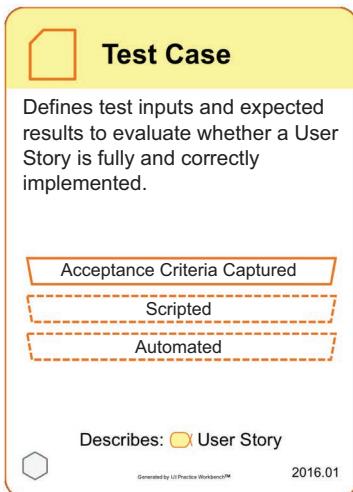


Figure 15.5 Test Case work product card.

Scripted. The step-by-step procedure for testing and accepting the user story is available. This also necessitates the preparation of test data and test environment used when executing the test case.

Automated. The test case is automated and can be executed with little or no intervention.

15.5

Kick-Starting User Story Lite Usage

There were two primary challenges our TravelEssence development team faced that led them to decide to try User Stories Lite in their endeavor. First, Smith's team members sometimes found themselves wondering about the purpose of the system they were developing. This often resulted in animated discussions with Angela. So, instead of just enumerating PBIs, Angela recognized that by investing a little time in developing PBIs into a user story format, the resulting requirements would help the team better understand the purpose of the system they were developing. This would also help Angela when discussing the system with other stakeholders, such as Dave.

The second challenge the development team often faced was that product backlog items were sometimes too large to fit within a single sprint/iteration. Smith had heard that the User Story Lite practice could help them with both of these challenges and so the team decided to try out this practice to see if it could help solve their challenges.

15.6

Working with User Story Lite

Working with User Story Lite involved several activities (see Figure 15.6). First, the team needed to find User Stories, prepare each User Story for development, and then accept the implementation of the User Story. (The actual implementation (i.e., writing and testing code) is outside the scope of the User Story Lite practice we are describing in this section; it is expected to be addressed by another practice. Later

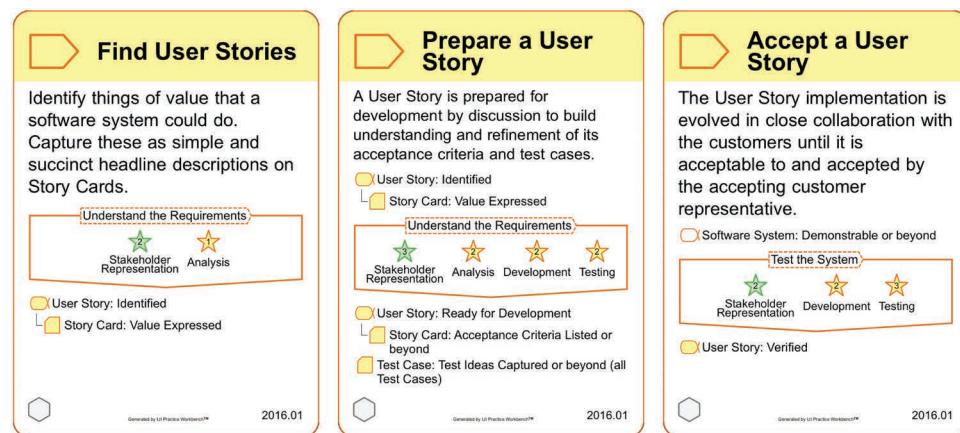


Figure 15.6 User Story Lite activity cards.

in Chapter 17, we will show how a microservice practice can be used to accomplish this.)

15.6.1 Find User Stories

Angela and the team were discussing which PBIs they would target for the next iteration. Among them were the following three backlog items:

- improve algorithms to rank destinations according to traveler-specific preference;
- improve algorithms to rank destinations according to general popularity of destinations; and
- collect user data from users and analyze them.

15.6.2 Prepare a User Story

Having agreed to the User Story Lite practice, the team proceeded to prepare each story for development in the next iteration. The preparation involved some detailed discussion.

Tom was quick to highlight that the purpose and scope of the above items were not clear. For example, the team was not clear on the acceptance criteria for improving the algorithms. They were also unclear about the purpose of collecting and analyzing user data, and hence the scope of this backlog item.

Smith explained the idea of the User Story Lite practice to Angela. She was quick to grasp the problem the team was facing, and understood how this practice could help. Together as a team, they expressed the User Stories as shown in Figure 15.7.

<p>As a traveler, I want to have destinations I like to be ranked higher than other destinations so that it is easier for me to find them.</p> <p>Acceptance criteria:</p> <ol style="list-style-type: none"> 1. A visited destination ranks higher than a non-visited one. 2. A “liked” destination ranks higher than a “non-liked” one. 	<p>As a traveler, I want to have popular destinations ranked higher than other destinations so that it is easier for me to find them.</p> <p>Acceptance criteria:</p> <ol style="list-style-type: none"> 1. Each destination visited by a traveler will be given a higher score. 2. Each destination liked by a traveler will be given a higher score. 	<p>As TravelEssence promotion staff, I want to track the actions on the recommendation list so that I can improve the quality of the recommendation and user experience.</p> <p>Acceptance criteria:</p> <ol style="list-style-type: none"> 1. Count the clicks, likes, and booking on each recommendation destination by specific traveler and travelers in general. 2. Trend chart by day, week, month of top N destinations.
---	--	--

Figure 15.7 User Story examples.

Tom, Joel, and Grace were much happier with the User Story format as depicted in Figure 15.7 compared to what they had earlier (see Sections 10.1 and 14.4.2), as this format helped them better understand the purpose of the system they were developing. Furthermore, the added detail helped them estimate each story and ensure each one was small enough to fit into an iteration.

Angela mentioned that expressing the requirements in this User Story format demanded more effort from her, but after some discussion, she agreed that this small upfront investment was worthwhile because it made her think in more detail about what she wanted. For example, in the first and second stories in Figure 15.7, the agreed-on acceptance criteria made clear to the team what Angela would accept for improved algorithms. In the third story, because it specified “count clicks” and created a “trend chart,” the team understood better what Angela would accept for the data to be collected and how she expected it to be analyzed. The user stories would also help Angela when explaining to Dave, her boss, the specific requirements that the team would be focusing on in the next sprint. Note that these were not the only three user stories they were delivering. There were others, but for brevity, we limit our discussion to these three.

The development of each story would involve designing user interfaces, writing code (user interface code, back-end processing code, and database code), preparing test data, and testing the code according to the acceptance criteria. So, in general, completing one user story was not something each member could do in a day, especially if it involved new functionality, rather than a simple modification of some existing functionality. (Keep in mind that explaining how the team conducts their implementation—code and testing—is outside the scope of our User Story Lite practice.)

15.6.3 Applying the Splitting User Stories Pattern

As part of preparing the stories for development, the team proceeded to split each user story that was too large into smaller stories that were more aligned with the INVEST criteria (see Section 15.1), especially the *small* and *testable* criteria (see also Figure 15.8).

In general, having smaller stories with clear test criteria makes each story easier to complete, which rewards team members with a sense of achievement and improves team member progress assessments.

As an example, Figure 15.9 shows how the first user story was split into three smaller ones. The team members took the guidance from the Splitting User Stories pattern for approaches to accomplish the splitting, ensuring that the smaller stories were testable all the time.



Figure 15.8 Splitting User Stories pattern card.

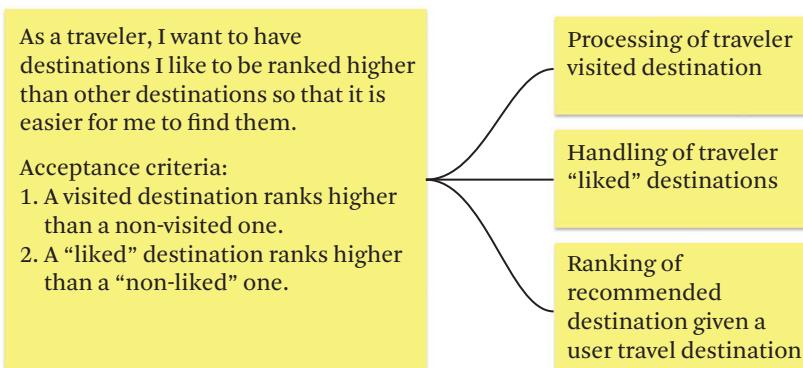


Figure 15.9 Splitting a user story.

15.6.4 Accept a Story

The team worked on the user stories within the current iteration. They made it a point to have their acceptance criteria expressed clearly. This investment paid off, as developers had a clearer idea what had to be done. They found that it was not easy to specify acceptance criteria at the same time as they described the story, because they were not yet sure what was really needed. Nevertheless, they felt that doing their best to split the stories was the right thing to do. Over the course of the

delivery of each user story, they regularly communicated with Angela and with each other regarding its details. The result was reduced disagreements when accepting the story.

Angela continued to work closely with the development team using their agreed-to Scrum Lite practice. She also participated in the acceptance of each user story. Whenever issues arose during the sprint, she worked with the team to refine the acceptance criteria.

15.7

The Value of the Kernel to the User Story Lite Practice

By describing the User Story Lite practice in an essentialized form (e.g., activity cards showing relationships to alphas), the team could see which alphas were being progressed and where their Requirements practice still had weaknesses. Specifically, the team recognized that their User Story Lite practice helped them achieve the following Essence kernel alpha states.

- Requirements alpha: Bounded and Coherent state
- Work alpha: Prepared state
- Requirements alpha: Acceptable state

The explicit activities in the User Story Lite practice directly supported the team in achieving key checklists within the Requirements alpha: Bounded and Coherent states. For example, the User Story practice encouraged stakeholders and team members to discuss and to agree on the purpose of the new system, as well as helping everyone involved to achieve a shared understanding of the extent of the proposed system. Furthermore, discussions helped both the team members and stakeholders to work through issues related to potentially conflicting requirements (see checklist items in Figure 15.10).

Achieving the Work alpha: Prepared state was helped because the User Story Lite practice encourage the splitting of each story in order to break the requirements down into tasks that the team could estimate and commit to completing within a single Sprint (see Figure 15.11).

The explicit activities in the User Story Lite practice next directly supported the team in achieving key checklists in the Requirements alpha: Acceptable state. For example, it encouraged the team and Angela to agree together on acceptance criteria, which reminded them of the importance of describing clear test steps that would lead to an acceptable solution (see highlighted checklist item in Figure 15.12).

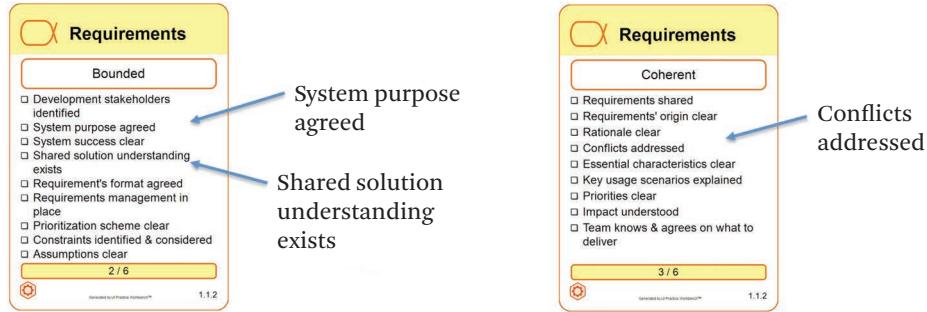


Figure 15.10 Requirements Alpha: Bounded and Coherent alpha state cards.

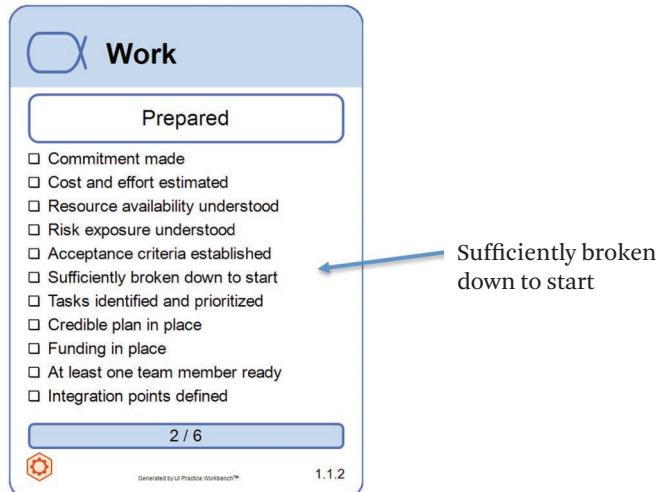


Figure 15.11 Work: Prepared alpha state card.

15.7.1 Visualizing the Impact of the User Story Lite Practice

While the User Story Lite practice helped the team progress two Essence kernel alphas, it did not solve all the challenges the development team faced with regard to satisfying Angela and progressing these alphas. After some discussion, the team began to realize that the User Story Lite practice had a number of weaknesses that was holding them back from fully achieving the Requirement alpha: Coherent and Acceptable states. For example, the informal nature of the User Story format left too much room for ambiguity in the requirements, and the team realized they were having trouble seeing the “big picture” and how new requirements would fit into that big picture. This led them to realize that they needed more help than the User

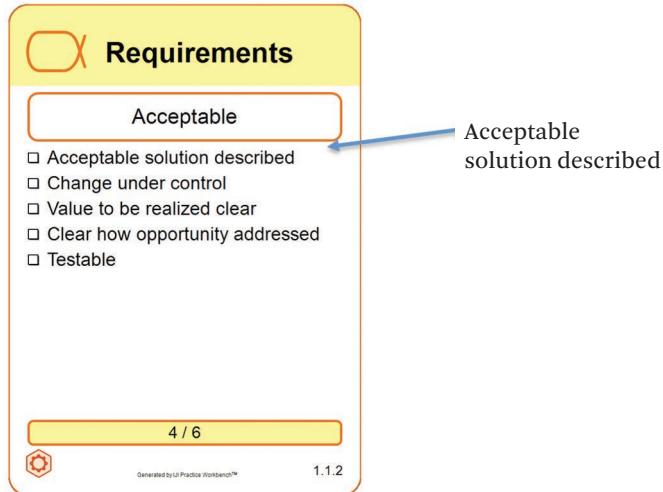


Figure 15.12 Requirements: Acceptable alpha state card.

Story Lite practice was providing when it came to structuring and documenting the stories within the overall system.

Smith said that he had heard that the weaknesses the team had found in their use of the User Story Lite practice could be addressed if the team considered migrating to use cases. As a result, the team agreed to study the Use Case Lite practice, which we will present in the next chapter.

The first thing they did is to compare the two practices and their coverage. We will present their comparison later, once we have introduced Use Case Lite in the next chapter. Here, we will discuss only that provided by the User Story Lite practice (see Figure 15.13).

The three activities in User Story Lite only cover two activity spaces. In particular, there is no activity that covers the Shape the System activity space. This is the activity space that deals with the structure of the solution area, including the structure of requirements and the structure of the software system. That was precisely what Smith's team indicated when they said they have trouble seeing the “big picture.” They had a list of user stories, but not how all the stories were related to one another. They could not see the entire shape of the software system. In the next chapter, we will present use cases as a way to deal with this gap.

We want to point out here again to the reader that it is not our intent in this book to create arguments or explain why one practice may be better than another (e.g.,

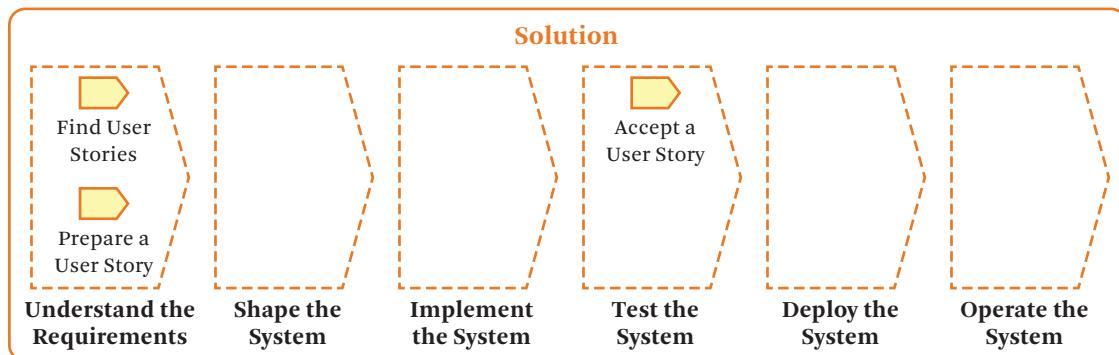


Figure 15.13 User Story Lite coverage of kernel solution activity spaces.

use cases vs. user stories). Our intent in this book is to help the reader understand the value of expressing practices in an essentialized form. Essentialization can aid teams in discussing their own endeavor situations, leading to appropriate decisions.

By looking at their practices through the lens of Essence, the team was able to see the strengths of their current agreed practices, as well as the weaknesses. For example, when the team was still small and they had just a few requirements, the User Story Lite practice worked well for them. But as their requirements grew further, and new team members were added, they realized they needed another approach to describe the big picture, and see how all the requirements fit into that big picture. By having an open and honest discussion about this, the team was able to agree that it would be an improvement to migrate to use cases. In the next chapter, we will discuss what the team learned as they did this, and how it helped them with their current challenges.

What Should You Now Be Able to Accomplish?

After studying this chapter, you should be able to

- explain the “Who,” “What,” and “Why” of user stories;
- explain the purpose of Card, Conversation, and Confirmation within a user story;
- explain the INVEST criteria;
- explain why we need the “so that” clause in a user story;

- explain the purpose of the User Story Lite practice and the problems it solves;
- explain how TravelEssence adopted and applied User Story Lite and the benefits they achieved, together with the benefits implied by using the User Story Lite practice in an essentialized form; and
- list and explain the alphas, work products, activities, and patterns of User Story Lite.

Running with Use Case Lite

The goal of this chapter is to introduce the Use Case Lite practice and its elements, and to demonstrate its application within the TravelEssence journey. In this chapter, the reader will be shown

- the differences between use cases and user stories;
- the elements of the use case practice, including relationships between elements, activity flows, and relationships with kernel elements;
- the concept of use-case slices and their benefit when use cases are used together with Scrum;
- the importance of monitoring progress and health of use cases and use-case slices;
- a simplified version of the use case practice (called Use Case Lite) in a real endeavor, together with the obstacles and challenges that might arise; and
- the coverage of kernel solution activity spaces by the Use Case Lite practice.

In this chapter, we will see why and how Smith's team started applying use cases in their development. The use case practice is a requirements analysis technique that has been widely used in modern software engineering since its introduction by Ivar Jacobson in 1987 [[Jacobson 1987](#)]. In fact, Ivar Jacobson introduced a larger practice, Use-Case Driven Development (UCDD) that extends the use case practice beyond requirements analysis to driving design, implementation, and testing. Since its introduction, the use case practice has been widely adopted in basically all industries and inspired the user story practice we presented in the previous chapter. In fact, the use case idea has become so widespread that the term "use case" has become a normal English expression used to understand the usages of virtually anything. The use case practice has evolved since 1987 and in its turn

has become inspired by the lightness of the user story practice, making it practical to use in all kinds of endeavors and in particular in agile endeavors [Jacobson et al. 2011, 2016]. Similar to what we did with Scrum and user stories, in this chapter we will describe a simplified version of the use case practice, which we refer to as Use Case Lite. It is designed for this book only and to be exchangeable with the user story practice; the better way to handle use cases is provided by the UCDD practice.

Smith's team had already been working with product backlog items as part of their Scrum explicit practice, as previously described. These product backlog items frequently had relationships between them that were not evident from the simple product backlog list. For example, some product backlog items provided basic functionality (meaning high-level requirements that do not include details), while other product backlog items provided additional extended functionality. Use cases can help teams understand the bigger picture and how product backlog items are related.

16.1

Use Cases Explained

A *use case* is all the ways of using a system to achieve a particular goal for a particular user. Use cases [Bittner and Spence 2003] help teams understand how user needs and requirements affect the behavior of the system. Often, at the start of a software endeavor, development teams are given a requirements specification which is essentially a narrative that supposedly captures the requirements for the system to be built. As shown in Chapter 15, user stories can help flesh out missing requirements by encouraging informal discussion between developers and users. However, often these informal discussions lead to user stories that are fragmented—there are too many of them, it is not clear how they make up something more complete—and they lack structure. This can become a significant problem especially for large complex systems. One approach that has been used to help address this problem is developing larger stories first, which are referred to as epics (see Section 15.1). This is an area, though, where use cases may be more helpful to teams. Use cases provide a systematic way to organize requirements. This structure makes it easier for teams to conduct analysis and orchestrate facets such as user interface (UI) design, service design, implementation, tests, and so on.

In the Unified Modeling Language [Booch et al. 2005], the relationships between users and use cases are represented in what is referred to as a use-case model—a model of the use cases within a system. Since users are not always human but can

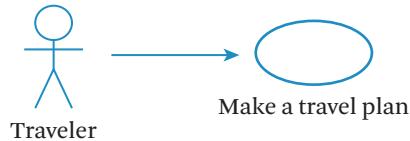


Figure 16.1 Use-Case Model example: “Make a Travel Plan.”

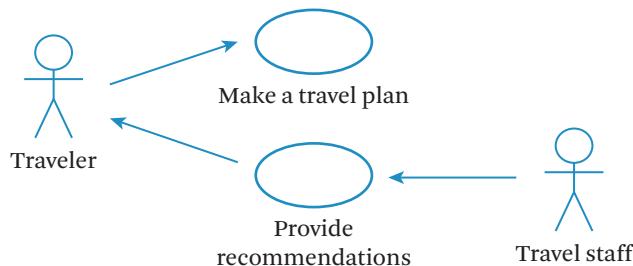


Figure 16.2 Use-Case Model for TravelEssence providing travel recommendations.

also be other systems, we use a more general term than users and speak instead about actors.

As you will recall, TravelEssence was a leading travel service provider that targeted both leisure and business travelers. One of their customer-facing systems, a travel portal, had the use case shown in Figure 16.1 (among others, but right now we will pay our attention to only this one).

The stick figure is the UML symbol for an actor and the oval is the UML symbol for a use case. A use-case model provides a visual representation of the software system, which is a very useful way for brainstorming its overall scope. What we show in Figure 16.1 is a use-case model with only one actor and one use case. In real systems, there are many actors and use cases. Figure 16.2 shows the actors and use cases for the development endeavor our TravelEssence team had just embarked upon with the following requirement items implemented:

- Req-Item #1.** System generates recommendations for a traveler
- Req-Item #2.** Mobile plug-in displays recommendations
- Req-Item #3.** System handles user’s selection to view or discard recommendations
- Req-Item #4.** System tracks recommendation success rate

Basic Flow:

1. Traveler provides travel details (travel dates and destination).
2. System searches and displays hotels for the travel dates and destination.
3. Traveler selects a hotel and room type.
4. System computes and displays available rooms and prices.
5. System makes a tentative reservation for the traveler at the selected hotel.

Alternate Flows:

- A1. Travel plan with multiple destinations.
- A2. Travel plan with a single destination but non-consecutive dates.
- A3. Travel plan with non-consecutive dates and multiple destinations.

Figure 16.3 Use-Case Narrative example: “Make a Travel Plan.”

The reader should not jump to the conclusion that use cases are just about diagrams. Use cases include the actual functionality and behavior of the system. Each use case is described in a use-case narrative. A use-case narrative provides a textual description of the sequence of interactions between the actor and the system. It also describes what the system does as a response to each message from the actor. This response includes internal actions as well as the sending of messages back to the actor or to other actors (which could be other systems). Figure 16.3 shows an example use-case narrative for the “Make a travel plan” use case. The use-case narrative is usually separated into two parts, referred to as the basic flow and the alternate flows. The basic flow describes a normal use of the described use case, often called the *happy day scenario*. In our TravelEssence case, this would be making a travel plan. The basic flow is worded in a sequence of steps you would expect to encounter when using or testing the system.

The alternate flows (there may be a single or multiple alternate flows) are variations of the basic flow to deal with more specific cases. These variations can be enhancements, special cases, etc. In this case, we have three alternative paths A1–A3 (see Figure 16.3). Their steps should be listed, but for brevity, we did not show them in the referenced figure.

As another example, the use-case narrative for “Provide travel recommendations” is depicted in Figure 16.4. The basic flow enumerates the sequence of interactions between the software system and the user to provide travel recommendations to the user (i.e., the traveler).

Basic Flow:

1. Traveler verifies travel details (travel dates and destination).
2. Traveler requests recommendations.
3. System provides list of recommendations.
4. Traveler browses recommendations.
5. Traveler selects and views a recommendation.

Alternate Flows:

- A1. Recommendations of different entities
 - a. Hotel
 - b. Place of Interest
- A2. Recommendations
 - a. Recommendations based on popularity rating
 - b. Recommendations based on pricing
 - c. Weighting function (preference indicator) for the above parameters
- A3. Recommendation request trigger
 - a. User initiated
 - b. System triggered
- A4. Sorting of recommendations
 - a. Sorting based on prices

Figure 16.4 Use-Case Narrative: “Provide Travel Recommendations.”

When trying to understand requirements, or anything, we start by understanding the heart of the matter, before diving into details. Use cases are structured to help with this kind of thinking. This same idea works when you are implementing a system; you start building the skeleton before fleshing out the details. The basic flow acts as the skeleton. If you compare the use case approach to capturing requirements with that of user stories in Chapter 15, it should be clear that the use case approach provides more structure through the separation of basic and alternate flows, in addition to other features we will describe. This structure also makes the requirements easier to understand, especially for endeavors that are large and complex.

Keep in mind that what you find in this chapter is just a brief introduction to the use case approach. In Section 16.9, we have made a more complete comparison. All of the practices we present in this book are example practices. We want the reader to understand how they can be represented in an essentialized form and what value essentialization brings to comparing practices helping you make the best decision

given your own situation. If you are interested in learning more, we recommend you study this approach in detail [Jacobson et al. 2011].

16.1.1 Slicing Use Cases

Use cases can help solve one of the key problems with incremental development that many teams face. That is the fact that the functionality of the entire software system can become fragmented into product backlog items scattered across all the iterations throughout the evolution of the software system. As such, it is often challenging to understand what the system can do at any point in time, or answer questions about the impact of a new product backlog item on the current software system. Use cases provide an approach for putting all these product backlog items into context from the user's point of view. A use case often contains too much functionality to be developed in a single iteration, such as a single sprint when using Scrum. That is why a use case is split up into a number of smaller parts that are referred to as *use-case slices*. These use-case slices taken together represent the whole use case. Each use-case slice should provide some value to its users, so they should typically include a sequence of activities and not just represent a user interface part or a business rule.

For example, as explained, the “Make a travel plan” use case has a basic flow and three alternate flows. So, you might implement the basic flow first as one use-case slice, before implementing the alternate flows as subsequent use-case slices. Recall that in Chapter 15, we discussed an example for splitting user stories. With use cases, the way a use-case narrative is expressed (partitioned into basic and alternate flows) facilitates splitting up the use case into smaller use-case slices. These can then be placed into the backlog for development (see Jacobson et al. 2011, 2016).

A *use-case slice* is a slice through a use case that is meaningful to describe, design, implement, and test in one go. It doesn't need to by itself give value to a user, but together with all other slices of the same use case, the value is achieved. For example, the basic flow of a use case is a good candidate to become an early use-case slice. Additional slices can then be added to complete the whole use case later. The slicing mechanism is very flexible, enabling you to create slices as big or small as you need to drive your development. The use-case slices include more than just the requirements. They also slice through all the other aspects of the system—e.g., user experience (user interface), architecture, design, code, test—and its documentation. Thinking about and developing a software system through slices in this way helps you create the right system. The slices provide a way to link the requirements to the parts of the system that implement them, the tests used to verify that the requirements have been imple-

mented successfully, and the release and endeavor plans that direct the development work.

The Use Case Lite practice that we discuss below provides a scalable, agile practice that utilizes use cases to capture the functionality of a software system and test them to ensure the system fulfills them.

16.2

Making the Use Case Lite Practice Explicit Using Essence

We have just introduced the concepts and the purpose of use cases and use-case slices, which are essential parts of our Use Case Lite practice. We will now look at how to describe this practice using Essence. The first questions we always ask when essentializing a practice are as follows:

- What are the things you need to work with?
- What are the activities you do?

Figure 16.5 expresses the elements of our Use Case Lite practice using the Essence language.

The Use Case Lite practice decomposes Requirements into Use Cases, which in turn are broken down into Use-Case Slices. Requirements, Use Cases, and Use-Case Slices are all important things we need to work with and progress. They are alphas in our essentialized representation of the Use Case Lite practice.

The Requirements are described using a Use-Case Model. A Use-Case Model is a tangible description of the Requirements, and therefore it is a work product. Each Use Case within our Use-Case Lite practice has one related work product, a Use-Case Narrative, and each Use-Case Slice has one related work product, a Use-Case Slice Test Case.

The bottom of Figure 16.5 shows four activities in our Use Case Lite practice, namely:

1. Find Actors and Use Cases to gain an overall understanding of what the system is about;
2. Slice the Use Cases to break them up into a number of intelligently selected Use-Case Slices that each fit within a single sprint;
3. Prepare a Use-Case Slice by enhancing the Narrative and Test Cases to clearly define what it means to successfully implement the slice; and
4. Test a Use-Case Slice to verify it is done and ready for inclusion in a release.

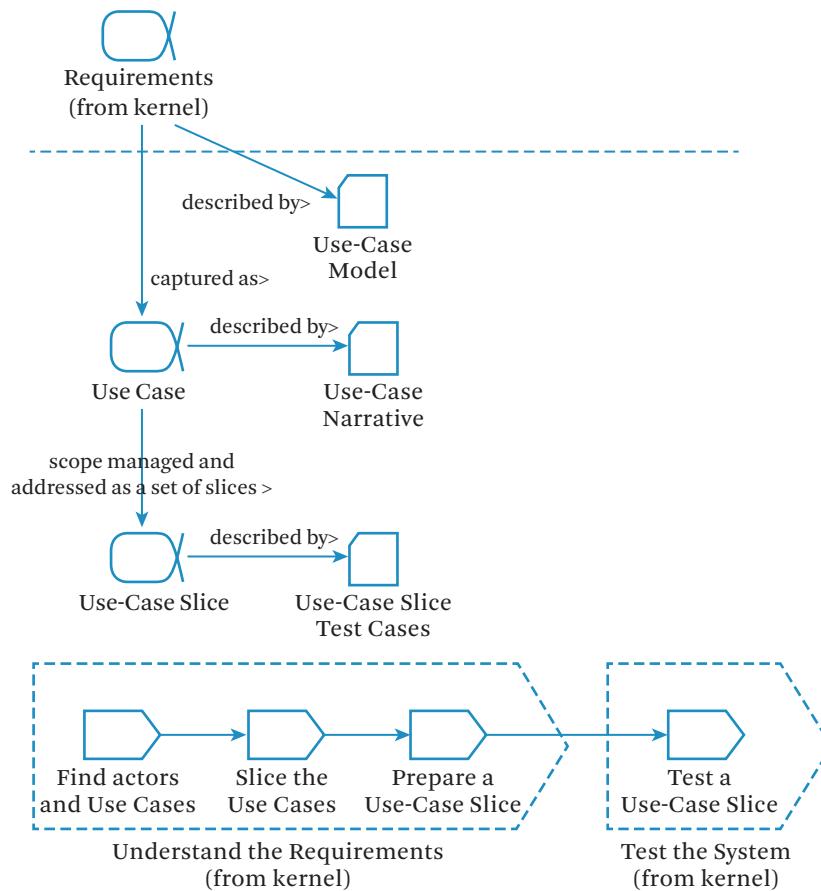


Figure 16.5 Use Case Lite practice expressed in the Essence language.

The first three activities reside in the Understand the Requirements activity space and the fourth resides in the Test the System activity space.

Use-Case Slices are identified by working through the use case to identify paths, scenarios, or—as we say—the stories that build up the use case. Typically, a *story* is any path that you may want to follow going through the use case: its basic flow or its alternative flows. Concrete examples of stories in the use case “Make a travel plan” are

1. the basic flow;
2. the basic flow complemented with alternative flow 1; and
3. the basic flow complemented with alternative flow 2.

The story idea is similar to that in the User Story Lite practice and is a very important step to find good slices. A use-case slice typically includes one or more stories.

Note that this practice does not provide any patterns. This illustrates that it is up to the author of a practice to dictate its scope. Normally, the more there is in a practice, the more specific it is. The less complexity it has, the more generic it is, and others can supply more specific information, such as patterns, when necessary.

The elements of the Use Case Lite practice are summarized in Table 16.1.

Table 16.1 Elements of the Use Case Lite practice

Element	Type	Description
Use Case	Alpha	All the ways of using a system to achieve a particular goal for a particular user.
Use-Case Narrative	Work Product	The story of how the system and its actors work together to achieve a particular goal.
Use-Case Slice	Alpha	One or more stories selected from a use case to form a work item that is of clear value to the customer.
Use-Case Model	Work Product	A model that captures and visualizes all of the practical ways to use a system.
Use-Case Slice Test Case	Work Product	Defined inputs and expected results to help evaluate whether a system works correctly. There can be one or more test cases to verify each use-case slice.
Find Actors and Use Cases	Activity	Agree on the goals and value of the system by identifying ways of using and testing it.
Slice the Use Cases	Activity	Break each use case up into a number of intelligently selected smaller parts for development.
Prepare a Use-Case Slice	Activity	Enhance the narrative and test cases to clearly define what it means to successfully implement the slice.
Test a Use-Case Slice	Activity	Verify the slice is done and ready for inclusion in a release.

This practice is primarily in the solution area of concern because it focuses on the Requirements and the Software System kernel alphas. Like the User Story Lite practice, it provides no explicit guidance in the endeavor area of concern, and is therefore well complemented by the Scrum Lite practice.

16.3 Use Case Lite Alphas

The Use Case Lite practice involves the following alphas (shown in Figure 16.6): Requirements (from the kernel), Use Case, and Use-Case Slice. The cards show the states of each alpha.

16.3.1 Progressing Use Cases

The Use Case Lite practice provides an effective way to progress requirements. Again, *a use case is all the ways of using a system to achieve a particular goal for a particular user*. So, the use case first starts off with a goal, and along with it go all the ways—or what we call stories—for achieving that goal. Through the use-case narrative, the stories are organized with a structure that is understandable by both the customer representative and the development team. A team then fulfills the simplest story and proceeds to incrementally progress through other stories until all of them are fulfilled.

Thus, the Use Case alpha has the following states of progression and health (see Figure 16.7).

Goal Established. The scope of a use case is defined by the goal of the use case: what the actor wants to achieve.

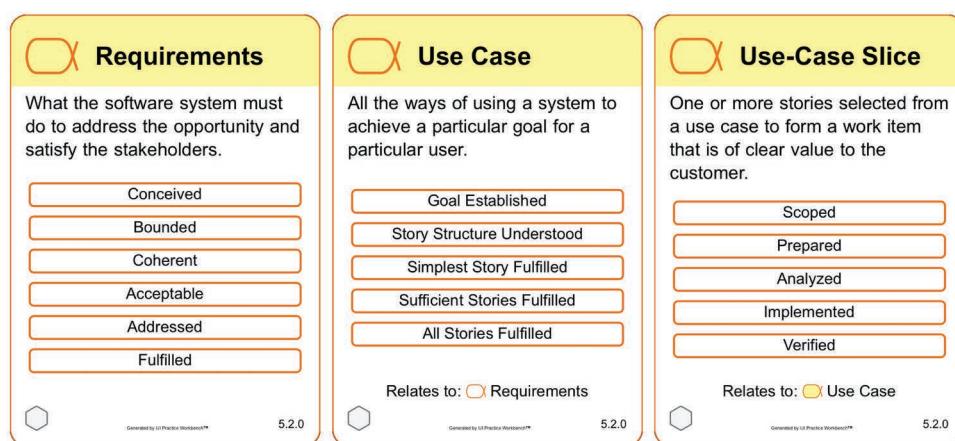


Figure 16.6 Things to work with (alphas).

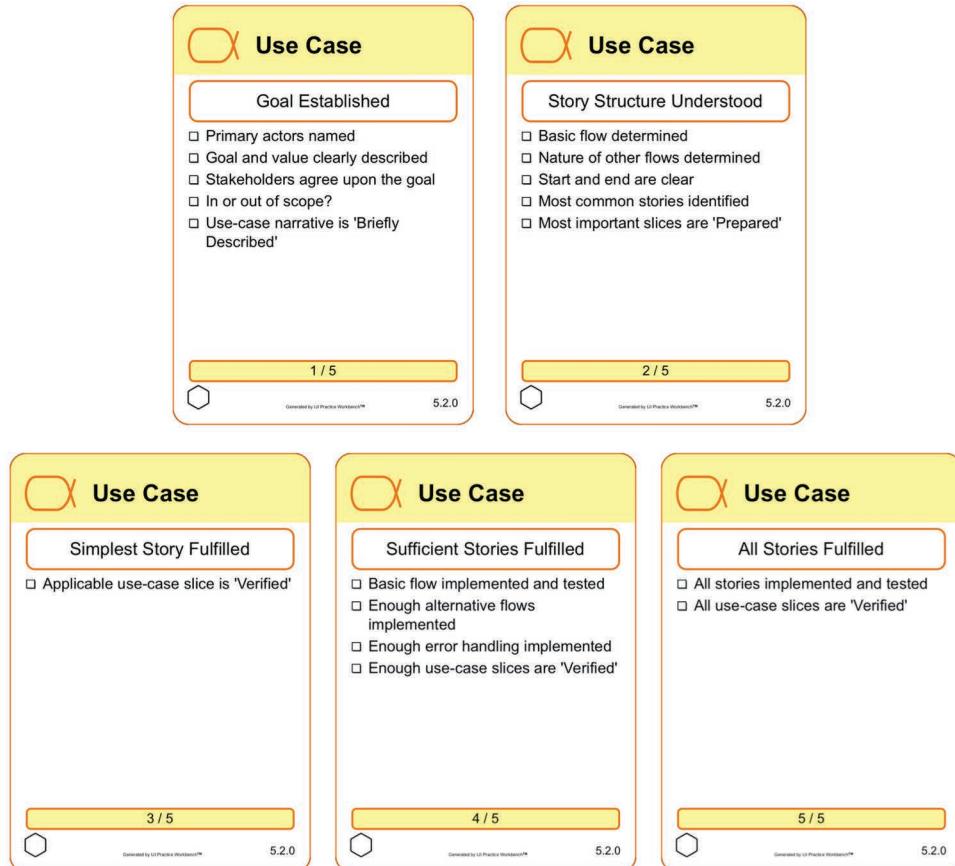


Figure 16.7 Use Case alpha state cards.

Story Structure Understood. One of the key benefits of use cases is that it provides a structure. Rather than a heap of requirement items or user stories, use cases provide a structure. In this state, that structure is defined and understood.

Simplest Story Fulfilled. The simplest or the most basic flow through the use case drives the code skeleton. Once this code skeleton is formed and stabilized, it becomes easy to implement the rest of the stories.

Sufficient Stories Fulfilled. Once sufficient stories are fulfilled, the use case can be evaluated as to whether it achieves its goal well.

All Stories Fulfilled. Finally, the entire use case is completed.

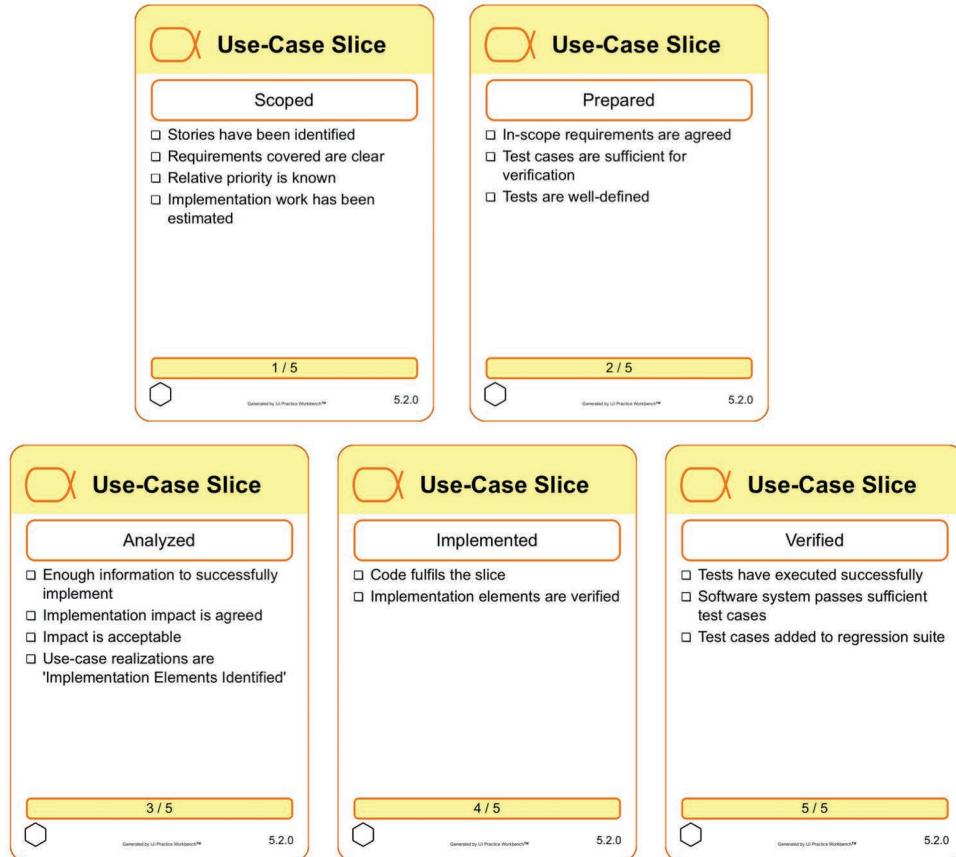


Figure 16.8 Use-Case Slice alpha state cards.

16.3.2 Progressing Use-Case Slices

It will take several sprints, or even releases, to fulfill all the stories in a use case. In each sprint, teams fulfill a portion of a use case, which (as you know) we call a use-case slice.

Working with a use-case slice is quite similar to working with a user story. The use-case slice alpha has the following states (see Figure 16.8).

Scoped. When this state is complete, the use-case slice has been identified and its scope clarified.

Prepared. At this state, the information the development team needs to implement the use-case slice is available, including priorities relative to other

slices, estimates of cost to implement, dependencies on other use-case slices, etc.

Analyzed. At this state, the development team has a common agreement on how the use-case slice will be implemented. This includes agreeing on aspects such as user interfaces (i.e., how information should be presented on the screen, how the user would interact with the system), persistence (e.g., updating the database), and so on.

Implemented. At this state, the use-case slice is implemented. This involves writing actual code.

Verified. At this state, product owners verify the use-case slice does what is expected.

16.4 Use Case Lite Work Products

The work products in this practice are depicted in Figure 16.9. There are cards whose content provides guidance on the levels of detail for each of the work products (Use-Case Model, Use-Case Narrative, Use-Case Slice Test Case). The following quote is taken from the Use Case 2.0 ebook [[Jacobson et al. 2011](#)]:

All of the work products are defined with a number of levels of detail. The first level of detail defines the bare essentials, the minimal amount of information that is required for the practice to work. Further levels of detail help the team cope with any special circumstances they might encounter. For example, this allows small, collaborative teams to have very lightweight use-case narratives defined on simple index cards and large distributed teams to have more detailed use-case narratives presented as documents. The teams can then grow the narratives as needed to help with communication, or thoroughly define the important or safety critical requirements. It is up to the team to decide whether or not they need to go beyond the essentials, adding detail in a natural fashion as they encounter problems that the bare essentials cannot cope with.

The lightest level of detail is shown at the top of each work product card. The amount of detail increases as you go down the card. For example, in Figure 16.9, Value Established is the lightest level of detail for the Use-Case Model, while Structured provides the greatest detail. For the Use-Case Narrative, Briefly Described is the lightest level of detail, whereas Fully Described is the deepest level. A team always starts with the lightest level, and can then decide how much more detail they need, based on their own situation. These levels of details will be described in the next section, when we elaborate each work product.

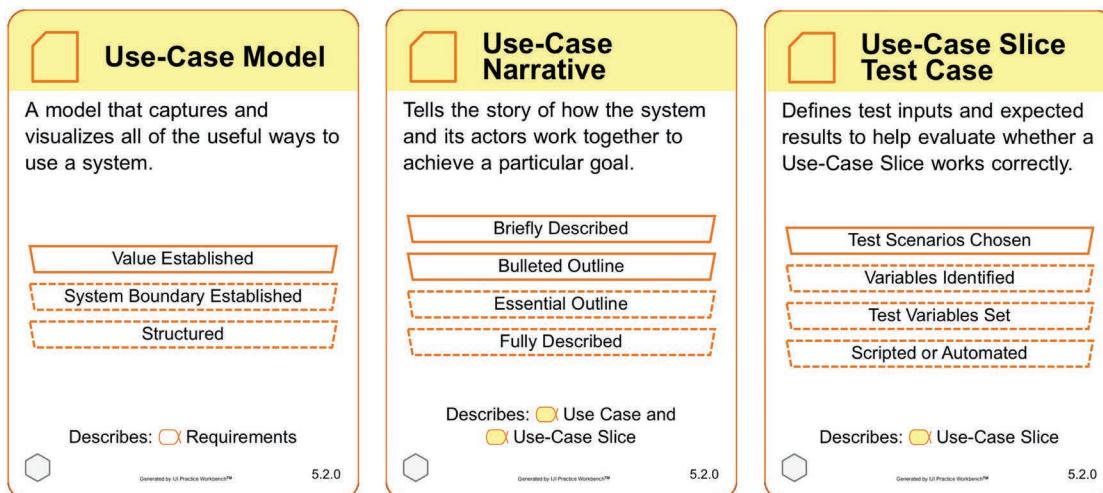


Figure 16.9 Things to work with (work products).

It is to be expected that each use case will evolve over several iterations/sprints by coming back to the activity slice the use cases. More slices may be added to the use case, and each use case is improved as we learn more. As such, the alpha states that show progress of a use case could be based on the evolution of the slices. This is a similar idea to what we saw earlier in Part II, where the progress of the Requirement alpha was based on the progress of Requirements Items. This will be more apparent later when we provide a concrete example.

16.4.1 Use-Case Model

A use-case model captures and visually presents all the useful ways to work with a system (see Figure 16.10). Figure 16.2 gave an example of a simple use-case model.

A use-case model describes not just one but several use cases and how together they provide value to its users (i.e., actors). These use cases need to have clearly defined scope. The use-case model has the following levels of detail.

Value Established. At this level of detail, the value of the use cases and hence the use-case model is established. Readers of the use-case model have a good understanding of what the use cases are about, what they do, and how actors benefit from them.

System Boundary Established. At this level, the scope and boundaries of the requirements are described. The team and stakeholders have a clear understanding of what is within its scope and what is not.

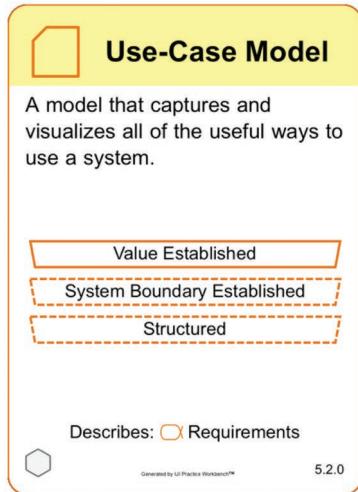


Figure 16.10 Use-Case Model work product.

Structured. At this level of detail, the Use-Case Model is well structured. There is little or no overlap between Use Cases. The dependencies and relationships between Use Cases are described clearly.

16.4.2 Use-Case Narrative

A Use-Case Narrative describes the story (i.e., sequence of steps) of how the system and the actors work together to achieve a particular goal. (The term “story” here is not the “user story” we presented in the previous chapter. Rather, the term “story” is just the normal English word.)

A use-case narrative can be described at several levels of detail (see also Figure 16.11).

Briefly Described. At this level of detail, the use-case narrative only has a brief description of the use-case goal and what it is about.

Bulleted Outline. At this level of detail, the story of how the system and actors will work together is available. The examples provided in Figures 16.3 and 16.4 are at this level of detail. Of course, the lists can also be numbered.

Essential Outline. At this level of detail, the story is full blown. In the context of requirements of the software system, the various alternative usages and exceptions to be handled are clearly described.

Fully Described. This is a very detailed description of the use case. All conversations are clearly spelled out.

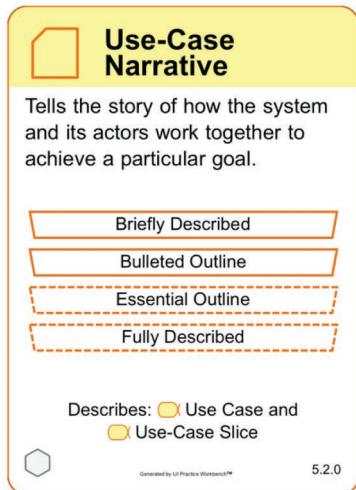


Figure 16.11 Use-Case Narrative work product.

16.4.3 Use-Case Slice Test Case

The use-case slice test case work product defines the inputs and expected outputs to help evaluate whether a use-case slice is implemented correctly.

Figure 16.12 depicts the level of details in a use-case slice test case.

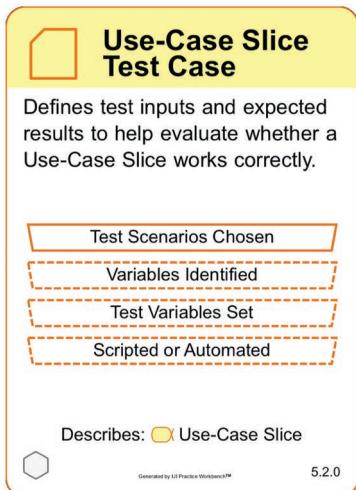


Figure 16.12 Use-Case Slice Test Case work product.

Scenario Chosen. At this level of detail, the different scenarios required to test the use-case slice are described. This includes the normal way of using the use-case slice and other variations (alternative usages and exception cases). The example in Section 16.6.2, Table 16.3, is at this level of detail.

Variables Identified. At this level, the different variables are listed. For example, the variables for testing the “Make a travel plan” use case include traveler identification and destinations (see Figure 16.3).

Variables Set. At this level, the actual variables are defined. To continue the same example, the traveler might be Sam, whose identification is 12345678. The destination is Singapore. The popularity rating of the Singapore Zoological Gardens and Shangri-La Hotel are set.

Scripted or Automated. At this level of detail, the test cases are clearly described such that a person can run the test case by following a step-by-step procedure without misinterpretation, or a software tool can execute it repeatedly with pass/fail results clearly defined.

16.5 Kick-Starting Use Cases Lite to Solve a Problem Our Team Is Facing

One day at TravelEssence, Tom raised a question: “Our endeavor is getting more complex and it is difficult to see the big picture when looking at our product backlog. I have heard that the Use Case Lite practice could help us with this problem. But if we migrate to use cases, do we need to rewrite all our old product backlog items into use cases?”

Smith replied, “Kick-starting Use Case Lite for an endeavor starts with identifying users and use cases to produce a skeletal use-case model, and skeletal use-case narratives. By skeletal, I just mean it is not necessary to have detailed descriptions of all use cases of the system. So, no, we don’t need to rewrite all our old product backlog items. We just need to make a use-case model and map the backlog items that are already done to use-case slices of the use cases in that model. The new backlog items will of course be the use-case slices that are not yet implemented.”

The activities to apply Use Case Lite are Find Actors and Use Cases, Slice the Use Cases, Prepare a Use-Case Slice, and Test a Use-Case Slice, which we will discuss in subsequent subsections.

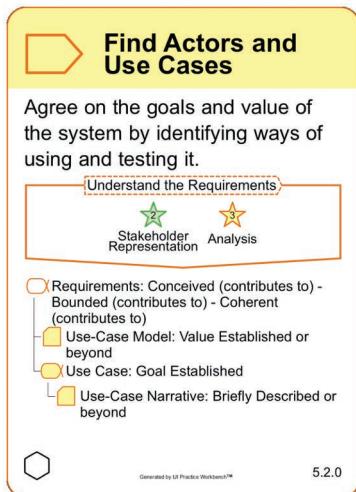


Figure 16.13 Find Actors and Use Cases work product.

16.5.1 Find Actors and Use Cases

The Find Actors and Use Cases activity is about agreeing on the goals and value of the software system by identifying the different ways of using it. As a corollary to this, we also find the different ways of testing it (see Figure 16.13).

On the Find Actors and Use Cases card, we see that the Stakeholder Representation and Analysis competencies are needed. We also see that this activity contributes to achieving the Requirements alpha's Conceived, Bounded, and Coherent states.

The card also indicates that the use-case model needs at a minimum to achieve the Value Established level of detail and that the use-case narrative at a minimum must be Briefly Described. The use case alpha needs to achieve the Goal Established state.

At TravelEssence, Smith next worked with his team to create a skeletal use-case model for the system as developed so far (see Figure 16.2). This can be said to represent release 1 of the TravelEssence system.

Smith then went on to create a new use-case model for the new release of the system—release 2—that they were now working on (see Figure 16.14). He added a new actor, Digital Officer, responsible for overseeing all new product releases. He also added two new use cases: “Manage release” and “Analyze recommendation results.” The “Manage release” use case provided features to review upcoming planned releases, and the “Analyze recommendation results” use case contributed

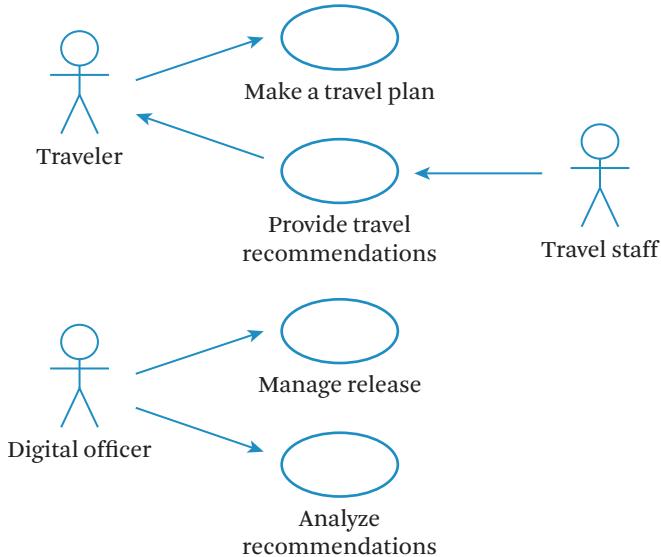


Figure 16.14 Use-Case Model for the next release of TravelEssence.

features to examine recommendations generated by the system and provide feedback.

Moreover, after a successful rollout to internal users, Angela had collected a number of feedback comments, which Smith also wanted to act upon for the next release of TravelEssence. Some of the feedback comments related to usability enhancements, while others related to new functionality requests, such as the following:

Recommendations by Advertisements. “We have revenue from advertisers. If these advertisers are within the vicinity of the traveler’s destination, they should be in the recommendations. However, we need to come up with a fair approach for prioritizing recommendations.”

Sorting by Vicinity. “The list of recommendations is rather long; it should be sorted according to how close the advertisers are to the traveler.”

Handling Favorites. “Sometimes, the traveler might want to remember the recommendation for future trips through some kind of ‘favorites.’ Favorites should appear in future recommendations.”

From the above list of suggested improvements, Smith made the following updates to the “Provide travel recommendation” use-case narrative shown in Figure 16.15. Updated and new lines are labeled in the figure.

Basic Flow:

1. Traveler provides travel details (travel dates and destination).
2. Traveler requests recommendations.
3. System provides list of recommendations.
4. Traveler browses recommendations.
5. Traveler selects and views a recommendation.

Alternate Flows:

- A.1 Recommendations of different entities
 - a. Hotel
 - b. Places of Interest
- A.2 Recommendation computation
 - a. Recommendations based on popularity rating
 - b. Recommendations based on pricing
 - c. #New Recommendations based on advertisements
 - d. #New Recommendations based on favorites
 - e. #Updated Weighting function for the above parameters (popularity, pricing, etc.)
- A.3 Recommendation request trigger
 - a. User initiated
 - b. System triggered
- A.4 Sorting of recommendations
 - a. Sorting based on prices
 - b. #New Sorting based on vicinity
- A.5 #New Recommendation actions
 - a. #New Add selected recommendations to favorites.

Figure 16.15 Use-Case Narrative: Provide Travel Recommendations
(Updated in the next release).

16.6

Working with Use Cases and Use-Case Slices

Working with Use Cases Lite is about iteratively updating the use cases and use-case slices. It provides a high-level guide to everyone on the team, not only to developers and testers, but also to product owners like Angela. As a software system evolves, the use-case model, with the use-case narratives, continuously provides an easy-to-understand overview of what the system does.

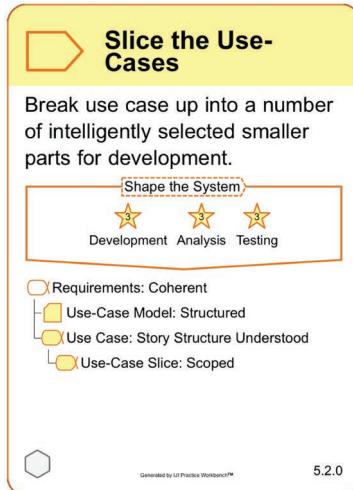


Figure 16.16 Slice the Use Cases activity card.

16.6.1 Slice the Use Cases

In the Slice the Use Cases activity, each use case gets broken up into smaller parts to facilitate development (see Figure 16.16).

Identifying the use-case slices for simple situations is extremely easy, because we don't need to think about all possible conditions. Of course, we eventually have to include alternative paths to cover each possible situation, but by handling each separately we keep things simple. The use-case narrative itself is structured in a form amenable for this purpose.

The use-case slices for the subsequent iterations of the TravelEssence team's endeavor are shown in Table 16.2. The three use-case slices in the table correspond to the three requests listed earlier. The key difference is that the use-case slices now have explicit reference to modular and testable changes to the use-case narrative.

After Smith and Angela had discussed and gained agreement on the modifications to the use-case narratives, they started conversing with the team about the changes for the next iteration.

After reviewing the narratives Smith had developed, Tom exclaimed, “Wow! Now I see how everything fits. This organization of the narratives gives a very good structure to help me understand how new requirement items will impact our system, the requirements, and the tests. After we implement our chosen slices for each sprint, we will need to verify that each one is done and is ready for inclusion in our next release.”

Table 16.2 New Use-Case Slices for the Use Case “Provide Travel Recommendations”

Use-Case Slice Name	Use-Case Slice Description
Recommendation by advertisements	#New Recommendations based on advertisements #Updated Weighting function for the above parameters
Sorting by vicinity	#New Sorting based on vicinity
Handle favorites	#New Add selected recommendations to favorites #New Recommendations based on favorites #Updated Weighting function for the above parameters

As you can see from Figure 16.15, the use-case narrative is organized such that each of the alternate flows provides a way to group related requirements. For example, recommendations on places of interest to visit or hotels to stay at (alternate flow 1) are grouped together, and recommendations based on attributes such as pricing and popularity are grouped together (alternate flow 2). This kind of organization can help developers structure their code and test cases in a way that eases the long-term maintainability of the system.

16.6.2 Prepare a Use-Case Slice

The Prepare a Use-Case Slice activity enhances the use-case narrative and the use-case slice test cases to clearly define what it means to successfully implement the use-case slice.

For brevity, we will only focus on one use-case slice: “Handle favorites.” Favorites are just a list, which the application stores for the user. If a user determines that a recommendation is useful, she might want to store this recommendation in her/his favorites list. This favorite list also acts as an input to the recommendation engine. So, the recommendation will behave differently for a new user without any favorites than for an old user that has some favorites.

It is clear from Table 16.2 that there are three distinct and separate slices:

1. #New Add selected recommendations to favorites
2. #New Recommendations based on favorites
3. #Updated Weighting function for the above parameters

Another important work product output of this activity is the use-case slice test cases. Smith and his team brainstormed the use-case slice test cases for each of these slices. The resulting outlines are summarized in Table 16.3.

The level of detail for the use-case slice test cases was Scenario Chosen.

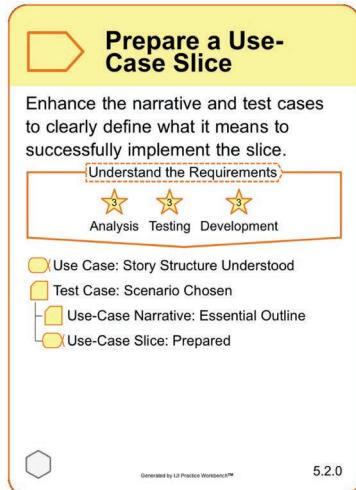


Figure 16.17 Prepare a Use-Case Slice activity card.

Table 16.3 Test Cases for “Handle Favorites”

Use-Case Slice	Use-Case Test Cases
#New Add selected recommendations to favorites	1. New favorite 2. Favorite already exists 3. Maximum number of favorites
#New Recommendations based on favorites	1. No favorites 2. One favorite within vicinity of traveler destination 3. One favorite outside vicinity of traveler destination
#Updated Weighting function for the above parameters	1. Weightage of favorites set to 0 2. Weightage of favorites set to 0.5

16.6.3 Test a Use-Case Slice

The goal of the Test a Use-Case Slice activity is to verify that the slice is done and ready for inclusion in a release (see Figure 16.18).

The use-case slice test cases chosen in Section 16.6.2 are of course an important input to this activity. During testing, these test cases are refined further with additional details to make sure that they are repeatable.



Figure 16.18 Test a Use-Case Slice activity card.

16.7

Visualizing the Impact of Using Use Cases for the Team

In our story, as the complexity of the endeavor grew, team members recognized that their approach to capturing requirements as a collection of user stories was insufficient. By migrating to use cases, the team found they could see the big picture of the system better through their use-case model and use-case narratives.

This is made visible by looking at Use Case Lite's coverage of the solution activity spaces in Figure 16.19. Compare this with Figure 15.13 in Section 15.7.1, and you will see that the activity space Shape the System is covered by the Find Actors and Use Cases activity. This activity has the dual purpose of understanding requirements and shaping the system. Both the use-case model and use-case narratives (basic and alternate flows) are tools that help teams organize/structure/shape requirements. They help teams see the big picture.

As new requirements came to the team, they could easily see where these would fit in the overall structure of use cases, use-case slices, and associated work products such as use-case narratives and use-case slice test cases. The team members agreed that the overall maintainability of the system had been much improved by their move to use cases. This was for multiple reasons: first, because of the structure provided by the organization of the narrative separating basic and multiple alternative flows; second, because of the big picture provided by the use-case model.

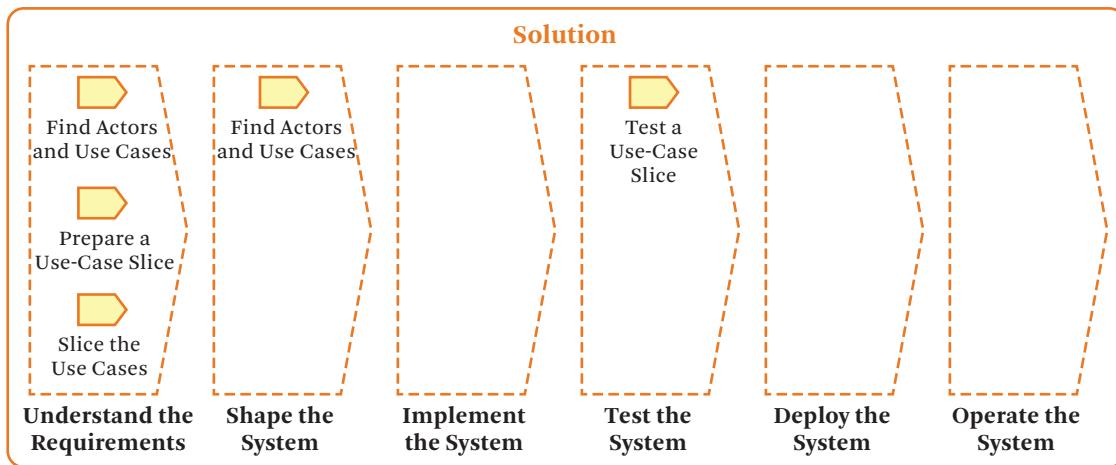


Figure 16.19 Use Case Lite practice coverage of kernel solution activity spaces.

16.8

Progress and Health of Use-Case Slices

You have just learned about how to apply the Use-Case Lite practice for several use-case slices from a single use case, “Provide travel recommendations.” In general, teams work with multiple use-case slices from multiple use cases at any given point in time. They complete use-case slices within each sprint (i.e., drive them to the Use-Case Slice: Verified state). While individual use-case slices are completed in each sprint, often it requires multiple sprints to complete a full use case.

Since on most software endeavors there are a number of use cases and use-case slices progressing in parallel, it is important that the team has agreed to a way to monitor their progress and health. One approach that is popular at the time of writing this book is for team members to have the competency to self-monitor their progress and health. The alpha state cards for Use Case and Use-Case Slices (see Figures 16.7 and 16.8) provide a tool for this purpose.

Now, recalling that our TravelEssence team has chosen multiple practices, this means that they have a number of alphas to juggle.

- From the Scrum Lite practice:
 - Sprint—focusing on the sprint goals.
 - Product Backlog Item—a change to be made to the product in a future release.

- From the Use Case Lite practice:
 - Use-Case Slices—certain ones need to be Verified by the end of the sprint.
 - Use Cases—they need not be completed for each sprint, but they are useful for determining which Use-Case Slices should be implemented first. As a group, they will take several sprints to be completed (i.e., All Stories Fulfilled). Thus, different use cases will be at different states at the end of each sprint.
- From the Essence kernel:
 - Work—the team needs to maintain the Under Control state as development progresses.
 - Requirements—this alpha progresses towards Addressed or Fulfilled, depending on the goals of the sprint.

Smith's team found that agreeing on these target states helped them achieve focus at the beginning of each sprint. They were also useful for reviewing progress during their Daily Scrum (see Chapter 14).

During one of their sprint retrospectives, Tom stated, “I find the checklists for the Use Case alpha states and Use-Case Slice alpha states more useful to me than the Requirements alpha states.”

Smith replied, “Yes, that is reasonable, as you are focusing on progressing specific use-case slices for specific use cases. However, from my perspective, I still need to view the big picture of development progress for the whole endeavor. Thus, I would need to understand the progress and health of it all. That would include not just the Requirements or Work alphas, but also the Opportunity, Stakeholders, Team, Way of Working, and Software System alphas.”

16.9

User Stories and Use Cases—What Is the Difference?

In the previous chapter, Smith's team applied user stories and in this chapter they applied use cases. This change resulted in much discussion and debate among the team members. In particular, Tom had been reluctant to make the transition. However, Tom agreed that the point should not be about his own preference, but about what was best for the team. The team decided to do their homework, research both approaches (i.e., user stories and use cases), and determine which was most appropriate for them.

The best way to answer the question “What is the difference between User Stories and Use Cases?” is to start by looking at their common properties, the things

that make them both work well as backlog items and enable them both to support popular agile approaches such as Scrum.

Use-case slices and user stories [Cohn 2004] share many common characteristics, such as the following:

- They both define slices of the functionality that teams can accomplish in a Sprint.
- They can both be sliced up if they are too large, resulting in numerous smaller items.
- They can both be written on index cards.
- They both result in test cases that represent the acceptance criteria.
- They are both placeholders for a conversation.
- They can both be estimated in similar ways.

So, given that they share so many things in common, what is it that makes them different?

Use cases and use-case slices provide added value. This is because they:

- provide a big picture to help people understand the extent of the system and the value it provides;
- offer support for simple systems, complex systems, and systems-of-systems; and
- result in easier identification of missing and redundant functionality, thanks to the big picture.

The sweet spot for user stories is when you have easy access to an expert on the subject of the requirements and when the severity of errors is low. Use cases and use-case slices are more suitable when there is no easy access to an expert or when error consequences are high. However, because the use case approach can scale down to the same scope as for user stories, you may still want to apply them. If you are confident that the subject system will always be in the best range for user stories, they may be a good choice. If you expect the system to grow outside that area, though, you might consider use cases and use-case slices.

Even though Smith's team had Angela close by, they found that when it came to explaining details of the requirements, she had difficulty expressing herself. That was when the team found that using use-case narratives in the lightweight manner described above became extremely useful. They found that when the requirements

became explicit, communications were simplified. Each person clearly understood what the others meant.

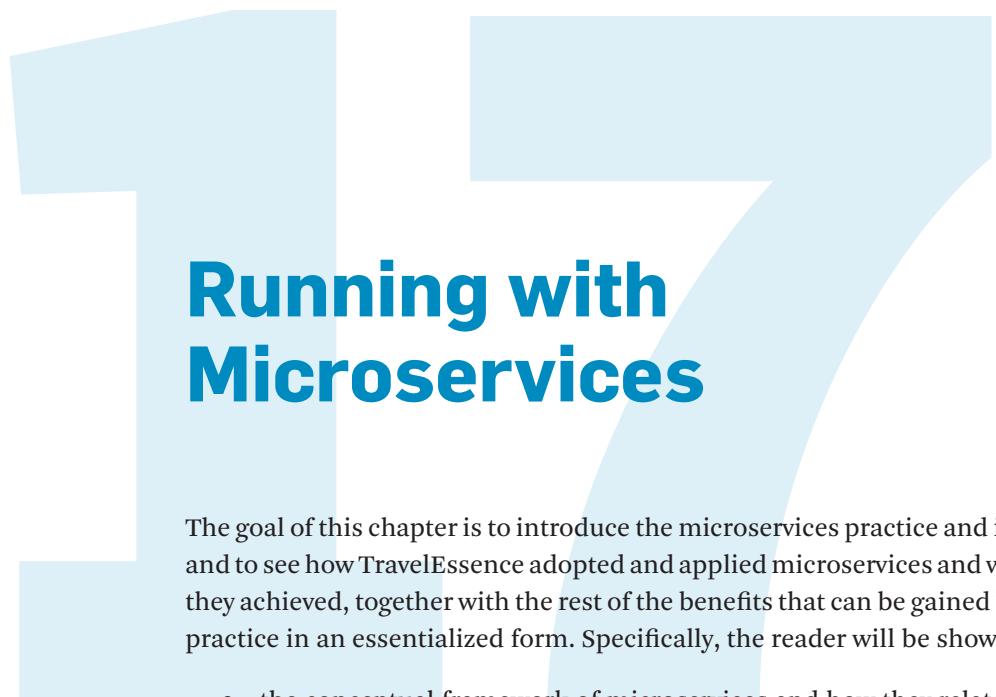
As we mentioned in this chapter's introduction, the term "use case" has been used as a normal English expression since around 2010. Consequently, at the introductory and executive level, many things are presented in terms of their use cases (for instance, within both Dropbox and Industrial Internet). Having this term in common encourages seamless traceability from introductory presentations to their realizations, and simplifies communication between people working with early visioning and people working with development.

Despite his early reluctance to apply use cases, Tom had one meeting with Dave that changed his stance. Dave was talking about the new digital "use cases" for the hotel system where the recommendation engine was part of the digital transformation. Dave used the term "use cases" like a normal English term. Moreover, when Tom was demonstrating the "Provide travel recommendations" use case, he found that he could explain his intent very well by simply walking through its narrative as he stepped through the demo. It was then that Tom was finally convinced about this term and the ideas behind it, from stakeholder conversations early on to realizations by the team (both developers and testers). From then on, the team as a whole had no more qualms living with use cases.

What Should You Now Be Able to Accomplish?

After studying this chapter, you should be able to

- explain the purpose of the Use Case Lite practice and the problem it solves;
- explain the difference between the basic and alternate flow of a use Case;
- explain use-case slices and their benefits compared to pure use cases;
- list and explain the alphas, work products, and activities of Use Case Lite;
- explain how TravelEssence adopted and applied Use Case Lite and the benefits they achieved, together with the benefits implied by using the Use Case Lite practice in an essentialized form; and
- compare use cases to user stories and describe scenarios in which each of them is more beneficial than the other.



Running with Microservices

The goal of this chapter is to introduce the microservices practice and its elements, and to see how TravelEssence adopted and applied microservices and what benefits they achieved, together with the rest of the benefits that can be gained by using this practice in an essentialized form. Specifically, the reader will be shown

- the conceptual framework of microservices and how they relate to software components;
- the elements of the microservices practice and how they interrelate, in addition to activities to progress the alphas and work products;
- how to use the microservices practice in a real endeavor, and specify potential obstacles and challenges;
- how to identify which kernel solution activity spaces are covered by this practice; and
- the visual representation of microservices using the Unified Modeling Language (UML).

Developing software with microservices is an evolution of designing software with components, which facilitates a modular approach to building the software system.

A component can be described as an element containing code and data. Only the code “inside” the component can modify the data inside the component, and only when another component sends a message with a request to do that. This idea is known as “data hiding” (data is hidden to other components) and is an accepted best practice in developing maintainable software.

Components were created to help solve the problems many developers face with large complex software systems that require long change cycles. Like components,

microservices are interconnected via interfaces over which messages are sent. Components as well as microservices manage their own databases, only accessible from other components via the interfaces. Such a “decentralized” approach allows each component and microservice to be developed and managed independently. Like components, then, each microservice can evolve separately from other microservices, thus making it easy to introduce new functionality.

In a software system built this way, each microservice runs a unique process, which is an execution of a computer program. There may be several such executions or instances of the same program running in parallel. What microservices bring to software beyond what components already did in this aspect is the ability to also deploy them independently without stopping the execution of the entire software system. This is very attractive to businesses because they can now update their software more quickly and hence adapt to market changes more rapidly.

17.1 Microservices Explained

To understand the value of microservices [Newman 2015], one needs to know how most software systems were traditionally developed before components became the standard. Traditional approaches resulted in monolithic software; everything was one big chunk of code and data with no modularity of real value (see Figure 17.1).

Before we continue our discussion, we would like to introduce some terms and concepts used in Figure 17.1 that a student new to software engineering may not be familiar with. The following are the four primary parts of a software system.

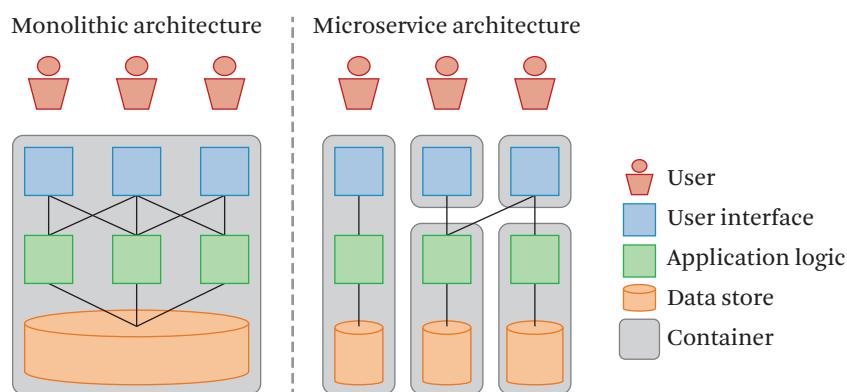


Figure 17.1 Microservices big picture.

User Interface. A user interface is the part of a software system that users interact with (screens, buttons, and the like).

Application Logic. The application logic is the code behind the user interface that performs computation, moves data around, etc.

Data Store. The stored data, retrievable by the application logic, lives in a data store.

Containers. Containers are components of a software system that can be managed separately (i.e., started, stopped, upgraded, and so on).

The left-hand side of Figure 17.1 depicts a monolithic software system. Even though it serves many users with different user interfaces, there is no clear separation of application logic or the data store. These appear as a single complex piece, wherein changes to one part affect other parts in a possibly uncontrolled manner. Such systems are not easy to maintain and extend.

The right-hand side of Figure 17.1 shows a microservice-based software system. Each microservice runs as a separate process, possibly in its own container or virtual machine. It has its own programming language, user interface, application logic, and data store. This allows developers to upgrade a particular microservice—say, from Java 8 to Java 9 to take advantage of new language features in Java 9, or to upgrade the data store without impacting other microservices. If, however, code for a different logical software element were to run in the same process or virtual machine, an upgrade of one element may inadvertently impact another element. Thus, enhancing the functionality of an existing microservice is easier than improving an entire monolithic software system. As an analogy, you can treat each column on the left side of Figure 17.1 as a village of people eating out of a single pot—i.e., the data store. With the monolithic approach, every time the one pot is being cleaned, everyone is affected. With the microservice approach, each family has their own pot. While one family is cleaning their pot, other families are not affected. Each family can clean and use their pots, independent of other families. This is similar to the modularity of microservices.

However, designing microservices is not without its challenges. If microservices are not designed properly, there might be too much coupling between microservices—for example, too much inter-microservice communication—leading to performance degradation. You can resolve this in one of two ways.

1. There must be appropriate identification of microservices. In addition to considering modularity, one needs to consider performance, specifically the interactions between microservices. There are a number of trade-offs that

require design practices and patterns, which are mostly out of the scope of this book.

2. One can utilize the fact that we are in the era wherein computing resources are inexpensive through the use of the cloud. This is because with cloud computing, when you need more computational computer time, you don't need to upgrade your own hardware; rather, you just request it from the cloud and the additional resources are provided via hardware that is owned and operated by the cloud service providers. For example, when Smith's recommendation engine was first introduced to a small group of users, a single instance of the engine was sufficient. When it was introduced to more users, more instances were needed, each requiring computing resources. With cloud services, adapting to changing requirements can be done easily and usually with reduced infrastructure costs because you can just purchase the amount of resources you need from the cloud service provider. You don't need to worry about the impact to modifying your own hardware setup inside your organization. Analogously, not so long ago, we needed to buy external hard drives to back up our data, but today we can put our data in the cloud for a yearly subscription fee. Through economies of scale the cloud provider can optimize the allocation of cloud storage for all its subscribers.

The second point above is the view taken today. Adaptability is key with the advent of inexpensive computing power. Upgrading finer-grained microservices is less disruptive to end users compared to upgrading monolithic systems, because you can upgrade module by module. You have probably encountered service providers saying, "We will not be providing any services during the maintenance period." This can be a thing of the past with microservices.

17.2

Making the Microservice Practice Explicit Using Essence

Microservices has become a very popular practice for software design. It was in the 1990s that it became standard to develop software by utilizing components with well-defined interfaces. Although this approach had already been used for more than 20 years (information hiding, object-oriented programming), it became popular through new platforms such as .COM and new languages such as Java. Part of what microservices has added to this evolution is support for components all the way down from design, to code, to deployment. Although this practice is popular, it is not a new idea; similar support has been provided with components since their introduction in the 1970s.

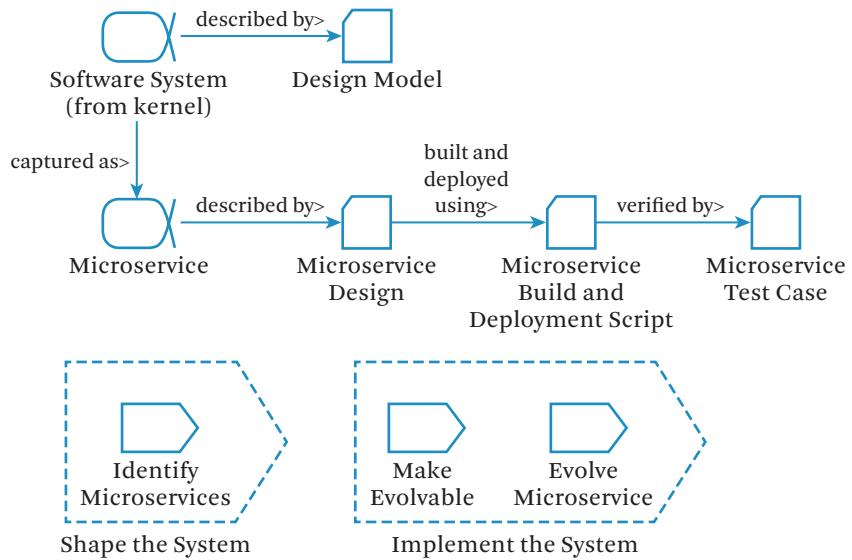


Figure 17.2 Microservices Lite practice expressed in the Essence language.

Moving from a monolithic approach to a microservice-based approach is not easy. It requires a new way of thinking. In particular, there are two important challenges:

- finding an effective way to decompose a monolithic software system into cooperating microservices; and
- finding an effective way to make changes to microservices and upgrade them once they are deployed.

A simple practice that takes advantage of microservices and addresses these challenges is depicted in Figure 17.2. This is a “Lite” practice because we have selected what we deem as a minimal core of the practice: we can make it minimal for demonstration, but in the real world it would require much more technical detail. For instance, we do not go into the depths of object-oriented analysis and design, or the technology needed to implement microservices, which would each require a set of books by themselves. The Microservices Lite practice highlights what we deem as important differences from traditional component thinking.

In this chapter, to demonstrate the use of the Microservices Lite practice, we will continue to use the example of the recommendation engine that Smith and his team were building. They implemented it as a set of microservices outside of

the existing legacy hotel reservation system. This allowed Smith to take advantage of new data processing technologies to perform the recommendation computation.

This practice extends the kernel with guidance regarding the following.

Microservice. This is an alpha, which the team will progress from its identification to its deployment (i.e., when it is made available to users). Since a software system is comprised of microservice sub-alphas, the progress and health of the software system is dependent on each and every microservice. In our TravelEssence example, the recommendation engine's set of microservices included one for general recommendations (based on user preferences and history) and one for location-based recommendations (based on the traveler's current location).

Design Model. This is a work product that describes the software system at a higher abstraction level than code—usually in some graphical notation such as UML. A good design model must simplify the understanding of the code, which normally means that elements in it are traceable to elements in the code. Here, we are looking at the software system as a whole to identify appropriate ways of decomposing it into microservices: a delicate activity called Identify Microservices, which relies not only on experience by the developers, but also on well-known criteria such as keeping low coupling between different microservices and high cohesion within a microservice.¹ The work on the design model also focuses on creating well-defined interfaces between microservices. Through the Design Model work product, we can also understand how individual microservices collaborate with one another. In our TravelEssence example, the work product describes how the recommendation microservices interact with the legacy hotel reservation system.

Microservice Design. This is a work product that clearly describes the design of a microservice, from its interfaces to its behavior and internal design. In our Microservices Lite version, we do not go into further detail beyond saying that a good microservice design exhibits the characteristics of clear interfaces, internal cohesion, and low coupling to other microservices.

Microservice Build and Deployment Script. This is a work product made concrete as an automated script that supports rapid production and deployment of each microservice independent of other microservices.

1. “Coupling is the degree of interdependence between software modules; a measure of how closely connected two routines or modules are; i.e., the strength of the relationships between modules. Coupling is usually contrasted with cohesion.” From [https://en.wikipedia.org/wiki/Coupling_\(computer_programming\)](https://en.wikipedia.org/wiki/Coupling_(computer_programming))

Microservice Test Case. This is a work product for measuring the behavior of a microservice.

The description of how the TravelEssence team applied the Microservices Lite practice is provided later in this chapter.

There are a number of activities providing explicit guidance on how to progress the above alphas and work products.

Identify Microservices. This first activity identifies and clarifies the purpose and scope of the set of microservices that will best fulfill requirements, still focusing on high cohesion within each microservice and low coupling between microservices. In our story, from earlier requirements background studies, Angela had found that recommendations fall into two categories: those based on customer traits, and those based on geographical localities. These two categories of recommendations are subject to independent requirement changes and enhancements. Thus, Smith identified them as two candidates for microservices.

Make Evolvable. This second activity looks at all the ways the development can support the evolution of a microservice. This includes making the design extensible, analogous to adding/upgrading an app on your phone without upgrading or buying a new phone. It also includes automation to test and deploy the microservice quickly without impacting other microservices. This is usually done by developing scripts that allow each microservice to be produced and deployed (made ready for use), independent of other microservices. By keeping each microservice small and reducing its impact on others, rapid deployment is supported.

Evolve Microservice. This is an ongoing activity to add new functionality to the microservice. Evolving a microservice includes modifying and testing the code to ensure each part meets its intent. Because microservices can be changed, tested, and deployed independently, evolving one can be accomplished rapidly. This includes making the microservice available to users.

Most of Smith's team found the idea of microservices very attractive. Joel, the ever-so eager-to-try-new-technology geek on the team, was elated when the suggestion was proposed. Joel was tasked to investigate the technologies needed to support microservice development. But Grace was not as convinced as Joel on the value microservices would provide to the team. She said, "I don't see how microservices will help us any more than using use cases and components." Joel replied,

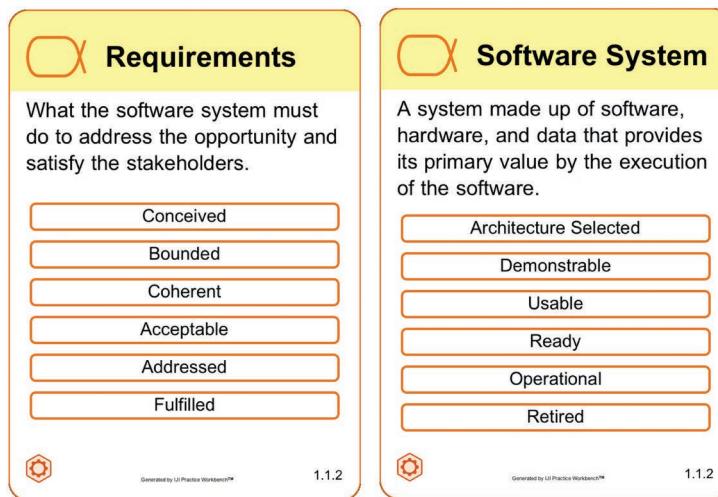


Figure 17.3 Requirements and Software System alpha cards.

“Grace, let me show you by using the Essence kernel.” He then pulled out the Requirements and Software System alpha cards (see Figure 17.3) and said, “Use cases help us progress the Requirements alpha, and components help us with the Software System alpha. Microservices also help us to progress the Software System alpha, but more on the deployment side.”

17.3

Microservices Lite

Microservices Lite is a practice that begins first by having a team identify the requirements, which can be done by applying use cases or user stories or using any other requirement practice. Next, they must find a proper set of microservices to implement the requirements; this is basically the same activity as finding appropriate components for a software system. The next task is to make each microservice evolvable by developing the needed support environment, such as deployment scripts that support the rapid deployment of each microservice, independently of other ones. Last, each microservice is evolved independently, which involves rapid coding and testing of each microservice as well as its rapid deployment into the production environment (which is the “live” environment in which real users use the software system).

The elements of the Microservices Lite practice are summarized in Table 17.1.

Table 17.1 Elements of Microservices Lite

Element	Type	Description
Microservice	Alpha	A separately replaceable piece of software that exhibits the properties of high internal cohesion, low coupling to external microservices, and well-defined interfaces.
Microservice Design	Work Product	A description of the responsibilities of the microservice and how it fulfills them.
Design Model	Work Product	A description of the overall software system and how the microservices relate to and interact with one another.
Microservice Build and Deployment Script	Work Product	An automated script that supports rapid production and deployment of each microservice, independent of other ones.
Microservice Test Case	Work Product	The set of tests to verify the behavior of the microservice.
Identify Microservices	Activity	Finding an appropriate set of microservices needed in the software system to implement the requirements, then outlining the responsibilities for each one.
Make Evolvable	Activity	Developing deployment scripts that allow each microservice to be produced and deployed (made ready for use), independent of other ones.
Evolve Microservice	Activity	This incremental activity includes rapid coding and testing of each microservice as well as rapid deployment of the microservice into the production environment.

17.4

Microservices Lite Alphas

The primary alpha in the Microservices Lite practice is of course the Microservice alpha. All work products and activities in this practice revolve around this alpha. The Microservice alpha relates to the Software System alpha as a sub-alpha; see Figure 17.4. As shown in the figure, the Microservice alpha represents “a resilient and elastic piece of the software” (i.e., part of the Software System) “that delivers a well-defined capability.” By resilient, we mean that it will be robust to changes made in other microservices. By elastic, we mean that the microservice can adapt to changes in workload (e.g., from a thousand users to one million users).

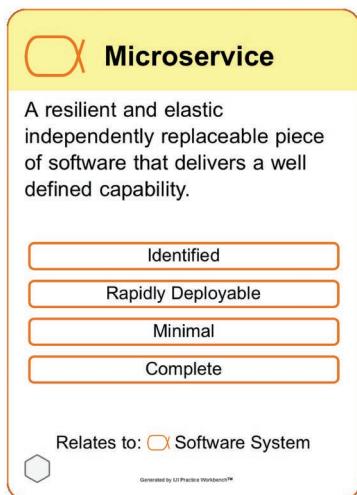


Figure 17.4 Microservice alpha card.

Building and delivering microservices progress through the following states.

Identified. First, the scope of the microservice must be clear. Team members must understand what functionality they are responsible for.

Rapidly Deployable. The major point about using microservices is the ability to quickly change each one and re-deploy it to the production environment. In this way, team members can quickly evolve the functionality of a microservice within the production environment. This rapid deployment capability is what gives microservices its advantage over traditional approaches. This is very much unlike monolithic systems, which—due to their complexity and tight coupling—are difficult to test and upgrade. Achieving rapidly deployable microservices, in contrast, requires an architecture with, again, low coupling between microservices and high cohesion within the microservice. It also requires setting up build and deployment scripts to automate the test and deployment steps.

Minimal. Once the microservice can be updated and deployed rapidly, the team can then work iteratively to realize its required functionality. This starts with fulfilling a minimal set of requirements so that it can be integrated and tested in collaboration with other microservices.

Complete. The team then evolves the microservice to fulfill all its required interfaces. At the same time, the team would likely refine the structure of the

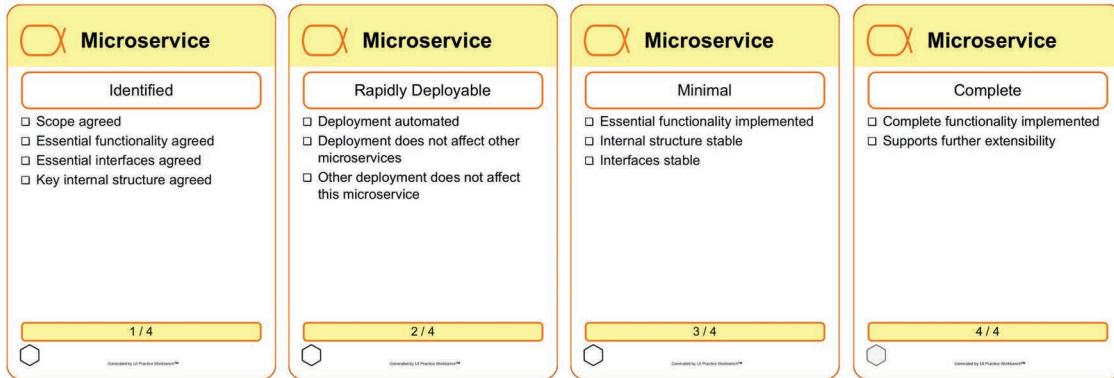


Figure 17.5 Microservice alpha state cards.

microservice so that it is extensible (i.e., new functionality to the software system can be added without big changes to the microservice).

The checklists for the Microservice alpha states are depicted in Figure 17.5.

17.5

Microservices Lite Work Products

The Microservices Lite practice introduces several work products: the Design Model, Microservice Design, the Microservice Build and Deployment Script, and the Microservice Test Case.

The Design Model work product describes how a microservice fits within the overall Software System alpha, and the Microservice Design describes the scope, interfaces, and the design of a particular microservice. The Build and Deployment Script is the work product that makes the microservice rapidly deployable. The Microservice Test Case is a work product to evaluate the quality of the microservice, ensuring that its implementation indeed meets the requirements.

Working with microservices is not easy. It is not just about understanding engineering concepts, but also the implementation technology. This book is not meant to be a comprehensive source on microservices. What we provide is a glimpse of how to work with microservices in a healthy manner, using the Microservices Lite practice.

There are several prerequisites when working with microservices, such as being able to clearly articulate their design, implementation, and deployment. A good form of visual notation can help. After the next subsection, in which we give an

overview of one standard form of such notation (UML), we will describe how to use each Microservices Lite work product in detail.

17.5.1 A Brief UML Primer

Before we go into greater detail about the Microservice Lite work products, we will provide a very brief and concise introduction to the Unified Modelling Language (UML). The UML is a visual language to describe various systems, particularly ones that are software intensive. Whereas Essence provides a visual language to describe software engineering practices and methods through constructs such as alphas, work products, etc., UML provides a visual language to describe software-intensive systems. By this we mean systems that may involve hardware, but are driven largely by the software. The main focus of UML is describing the software part. As we use UML to describe how Smith's team makes use of microservices, you will see how the languages of Essence and UML work together to build even more complete understanding and communication within Smith's team.

A microservice, again, is a special kind of subsystem that has its own data store. A legacy system is an existing system or subsystem that was likely built using some traditional approach that has resulted in it being monolithic. However, both microservices and legacy systems are denoted by the same subsystem notation.

Note that UML involves a lot more than what we discuss above. It also provides guidance to describe the structure and behavior of software elements. We only focus on a small portion of UML's structure description, using its notation for interfaces and subsystems, as shown in Table 17.2. We do, however, recommend that you learn more about UML.

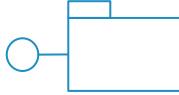
17.5.2 Design Model

The design model (see Figure 17.6) is a description of the Software System alpha. It depicts the elements in the Software System and how they interact with one another. Using Smith's recommendation engine as an example, we will just provide a brief overview of the design model, bearing in mind that comprehensive design is outside the scope of this book.

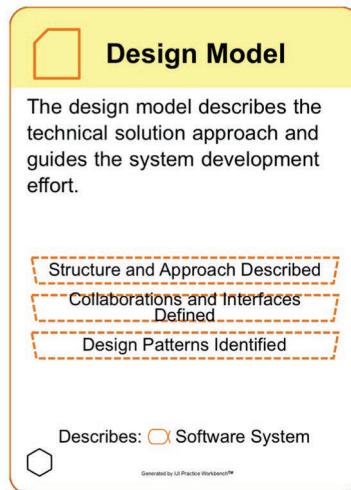
The Design Model work product has the following levels of detail.

Structure and Approach Described. At this level of detail, the design model clearly describes the elements of the software system. It delineates how the different parts are organized and the purpose of each part.

Table 17.2 UML for Microservices Design

Element Type	Notation	Description
Subsystem		This notation is like a folder, or what we like to call a package. A package contains a group of code or classes to offer some functionalities that clients might want to invoke.
Provided Interface		This notation looks like a lollipop. It is the outward functionality a subsystem offers. When using the functionality, the client does not need to understand what goes on within the subsystem.
Required Interface		This notation is like a socket. You can plug stuff into the socket to provide more functionality for a subsystem. For example, on your computer, you can plug hard disks, cameras, etc. into your USB port.

Collaborations and Interfaces Defined. At this level of detail, the roles and responsibilities of each part are more detailed. (The example in this section progresses to this level.)

**Figure 17.6** Design Model work product card.

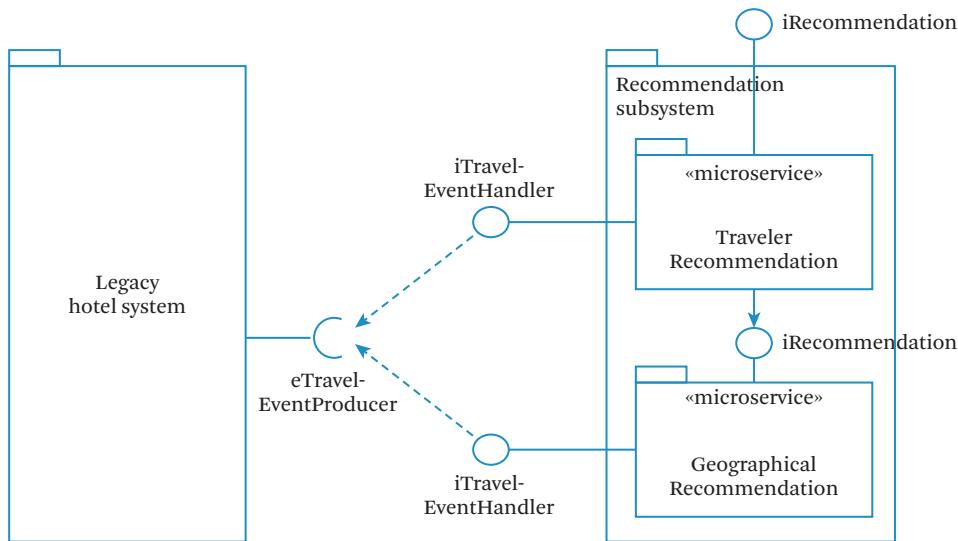


Figure 17.7 Design Model for recommendation functionality.

Design Patterns Identified. At this level of detail, common design patterns that can be shared across elements are identified and described. A design pattern is a common design solution to a common design problem.

At TravelEssence, re-engineering the legacy hotel system from a monolith to one entirely based on microservices would have been too big an effort. However, its new recommendation functionality was quite separate and distinct. The company's decision was to leave the legacy hotel system alone with minimal changes, and so the team addressed the task of developing the required new recommendation functionality using a separate microservices approach, as shown in Figure 17.7.

The left-hand side of Figure 17.7 shows the legacy hotel system. It would be modified slightly to provide an interface (referred to as an eTravelEventProducer) used to push out traveler events. Once the software in the legacy hotel system executed and got to a point where something of interest happened, such as when a traveler booked a hotel, that fact would be communicated to the iTravelEventHandler by an event. Such information would then be used by the Recommendation subsystem to analyze the popularity of hotels, travelers' preferences, etc.

The right-hand side of Figure 17.7 shows the recommendation subsystem designed using two microservices: (1) the "Traveler Recommendation" microservice dealing with specific travelers or groups of travelers and (2) the "Geographical

Recommendation” microservice dealing with specific recommendations related to the traveler’s current or upcoming geographical locations.

- As the name implies, each traveler would collect events through the iTravel-EventHandler interface, which would analyze them either geographically or according to specific travelers or traveler groups.
- These microservices would each provide an iRecommendation interface to present recommendations to travelers. The “Traveler Recommendation” microservice would provide some basic recommendations, and delegate the determination of further recommendations based on geographical details to the “Geographical Recommendation” microservice.

Once the system was deployed, there would be multiple microservice processes running (one for each traveler group and for each geographical area), each with its own data store for traveler recommendations. This was in line with TravelEssence’s goal to gradually roll out the recommendation functionality to increase the number of travelers and geographical regions.

17.5.3 Microservice Design

The microservice design (see Figure 17.8) is a work product describing the design of a microservice, from its interfaces to its behavior and internal design. It provides more detail than the design model for that specific microservice.

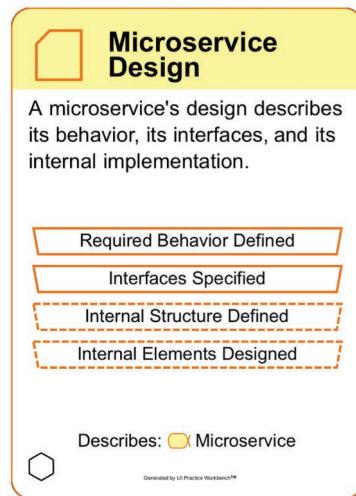


Figure 17.8 Microservice Design work product card.

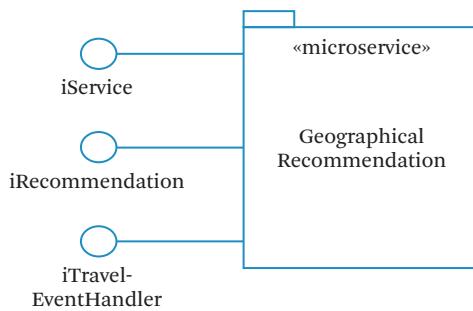


Figure 17.9 Microservice Design of “Geographical Recommendation.”

The levels of detail for this work product are as follows.

Required Behavior Defined. At this level of detail, the scope of a microservice is described in words.

Interfaces Specified. At this level of detail, the scope of the microservice is described using interfaces.

Internal Structure Defined. Once the external behavior is agreed on, this level of detail describes the elements within the microservice. Developers can then start to write code with a good understanding of this structure.

Internal Elements Designed. For complex microservices and elements therein, more details are needed; they are provided at this level.

We will continue to use TravelEssence’s recommendation engine as an example, and in particular the “Geographical Recommendation” microservice.

In addition to the interfaces highlighted in the design model, a microservice also needs interfaces to manage its execution, such as setting configuration parameters and controlling its execution (starting, pausing, resuming, stopping, reset). Figure 17.9 shows all the interfaces to the “Geographical Recommendation” microservice: those that manage its execution (*iService*), manage its recommendations (*iRecommendation*), and handle traveler events (*iTravelEventHandler*).

17.5.4 Build and Deployment Script

The Build and Deployment Script (see Figure 17.10) is an automated script that supports rapid production and deployment of each microservice, independent of other ones. The primary purpose of this work product is to make the build and deployment process as repeatable as possible, which is critical when working with

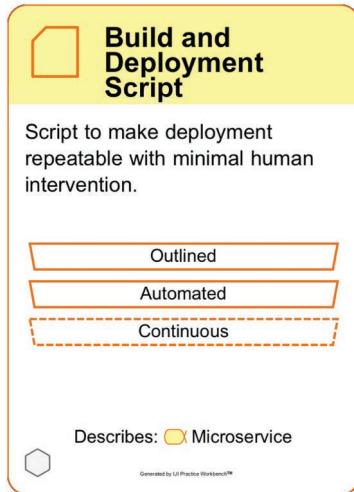


Figure 17.10 Microservice Build and Deployment Script work product card.

microservices. Without any rapid deployment and upgrade capability, there is no real advantage to using microservices when compared to using more traditional component approaches.

Working with build and deployment scripts involves the following levels of detail.

Outlined. At this level of detail there is an agreement as to what “rapidly evolvable” entails, and steps to achieve it are agreed on and described. There is not an actual runnable script available yet.

Automated. This is the level at which the real work has been done. The team has written the actual build and deployment script and it has made sure that it works within the development and deployment environment.

Continuous. This is a higher level of detail that ensures that the script runs in continuous support of microservice upgrades, without disruption to other microservices.

17.5.5 Microservice Test Case

The testing of a microservice follows a similar approach to testing user stories and use-case slices: You start by first identifying scenarios.

The execution of a microservice will likely depend on other microservices. So, you might have to “mock out” the surrounding dependencies (i.e., substitute real

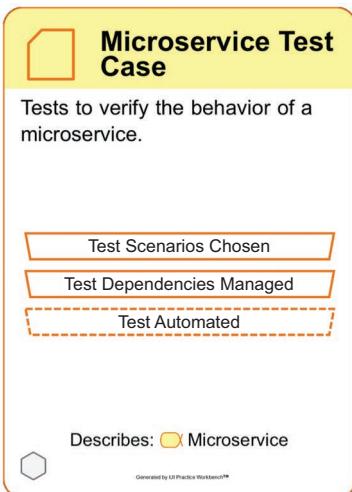


Figure 17.11 Microservice Test Case work product card.

parts of the software system that developers have little control over with parts that developers do have control over during testing). For example, at TravelEssence, the recommendation engine has a dependency on the legacy hotel system. So, instead of testing the recommendation engine microservices with the real legacy hotel system, Smith's team chose to create a mock (substitute) hotel system whose behavior they could change quickly. This might sound like a big task, but in reality only the interfaces needed by the recommendation engine need to be mocked out.

Testing microservices is not much different from how you think about test cases for use cases and user stories. Of course, the tools you use to test microservices (which normally do not have user interfaces) and use cases (which quite possibly have user interfaces) will be different. The levels of detail of the Microservice Test Case work product (see Figure 17.11) are as follows.

Test Scenarios Chosen. At this level of detail, different scenarios in which the microservice is used are enumerated systematically and prioritized.

Test Dependencies Managed. At this level of detail, the scope for each test case is agreed on, including dependencies, which will be mocked or stubbed.

“Mocked” in this case means to create extra test code that simulates the other side of the interface. “Stubbed” means that, instead of simulating the interface, the testers just ensure that the test code will execute without the program crashing.

Test Automated. At this level of detail, the test cases are scripted and automated. They usually run as part of the build and deployment process.

17.6 Microservices Lite Activities

Applying the Microservices Lite practice involves several activities: Identify Microservices, Make Evolvable, and Evolve Microservice. Each of these is discussed in the following subsections.

17.6.1 Identify Microservices

Microservice development begins with the identification of microservices within a software system. Identifying microservices requires both the Development and Analysis competencies. These competencies are needed to identify microservices that exhibit high cohesion, low coupling, and well-defined interfaces. Identifying microservices with these characteristics helps teams achieve the Software System alpha's Architecture Selected state (see Figure 17.12).

Depending on the specific endeavor, teams may make different decisions on the level of detail needed for work products during the identification of microservices. For example, some teams may decide that their design model work product only needs to describe the structure and approach (lowest level of detail; see Figure 17.6). Other teams may decide that the collaborations and interfaces among

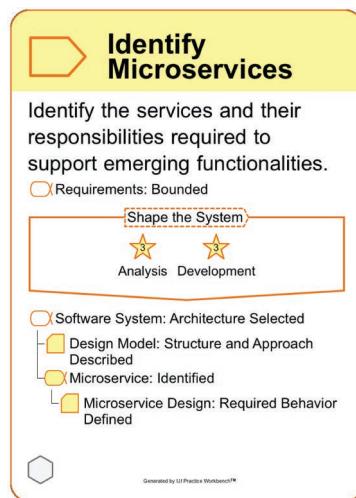


Figure 17.12 Identify Microservices activity card.

microservices need to be documented explicitly (second level of detail in Design Model). Still, other teams may decide that the design patterns employed need to be explicitly captured in the model (highest level of detail in Design Model).

Similarly, different teams may reach different conclusions on the level of detail needed in their Microservice Design work product with respect to the level of detail needed for defining required behavior, specifying interfaces, defining the internal structure, and internal element design (see Figure 17.8).

The Design Model work product (see Figure 17.6) gave Smith's team an overall perspective on how to design and implement the recommendation functionality. Working with microservices involves making them evolvable and then evolving each microservice and adding levels of detail to its design, build and deployment scripts, and test cases. Progressing the Microservice alpha through its states will be discussed later in this chapter (see Section 17.8).

17.6.2 Make Evolvable

The key and compelling characteristic about microservice development is the ability to make rapid changes to a software system in the production environment. The key goal is to be able to replace a single microservice quickly without affecting other parts of the software system (i.e., other microservices). By “rapidly” or “quickly,” we mean that changes that used to take months are now reduced to hours, and even to minutes. This does not come free, and it is what the Make Evolvable activity is about. It requires at least two areas of investment.

1. Ensuring the modularity and extensibility of each microservice so that requirements changes are localized to individual microservices, and that changes to a microservice do not severely impact other microservices. This is about great design, which we unfortunately cannot cover in this book. But suffice it to say that a team will capture its design approach in the Design Model, Microservice Design, and Microservice Test Case work products (see Sections 17.5.2, 17.5.3, and 17.5.5, respectively).
2. Improving the development and production environment so that changes to microservices are repeatable, reliable, and fast. This requires the streamlining of the deployment pipeline with plenty of automation. The idea is to reduce as much mundane and manual work as possible. This is embodied in the Build and Deployment Script work product (see Section 17.5.4).



Figure 17.13 Make Evolvable activity card.

The first point above is covered largely by the activity Identify Microservices (see Section 17.6.1), whereas the activity Make Evolvable focuses more on the second point (see Figure 17.13). It requires the scope and boundaries of microservices to be stable.

In our story, Smith had earlier identified several microservices, and they were now in the Identified state. The next state to achieve was to make each microservice Rapidly Deployable, which means that it should be possible to deploy the microservice to a target environment quickly, be it the test environment or the deployment environment. This deployment always has to be automated, which necessitates the use of build and deployment scripts.

17.6.3 Evolve Microservice

Once a microservice has been made rapidly deployable and its interfaces identified, it becomes straightforward to evolve each microservice, and thereby introduce new functionality to the entire software system (see also Figure 17.14).

Having gotten to this point, Smith's team could now safely develop and test the code required for each microservice. The team achieved this by first agreeing on the test cases before adding/changing microservice code required to successfully pass the test cases. In effect, Smith was applying another practice known as Test Driven Development (TDD) (i.e., first agree on the test cases and then use the test cases to drive the development). The test cases would initially fail because there was no

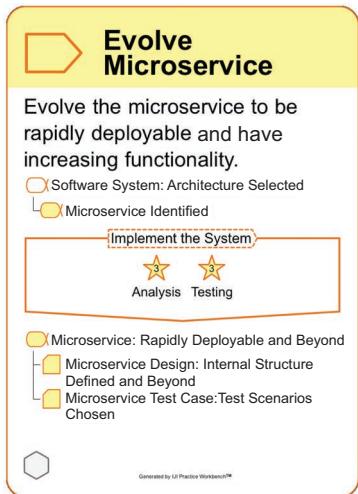


Figure 17.14 Evolve Microservice activity card.

implementation. Smith's team might identify more test cases along the way. But as the team completed the implementation, more test cases would pass. Thus, the number of test cases that passed would act as a progress indicator.

17.7

Visualizing the Impact of the Microservices Lite Practice on the Team

Recall that neither User Story Lite nor Use Case Lite provided any guidance on how to implement the software system. These two practices focused on requirements and tests of the software system. The Microservices Lite practice addresses implementation guidance, as shown in Figure 17.15. The Microservices Lite activities, for example, provided guidance to Smith's team on how to design and implement solutions.

Note that Microservices Lite does not deal with the requirements and test of the solution itself. It provides guidance only for the testing of individual microservices, not the entire software system's functionality. That is covered by the Use Case Lite practice discussed in the previous chapter.

At the time of writing this book, microservices are gaining widespread attention. Although the idea of having components as a separate and independent deployable unit of modularity has been in existence for a long while, the implementation of this idea more often than not had not been ideal. Monolithic designs often inevitably

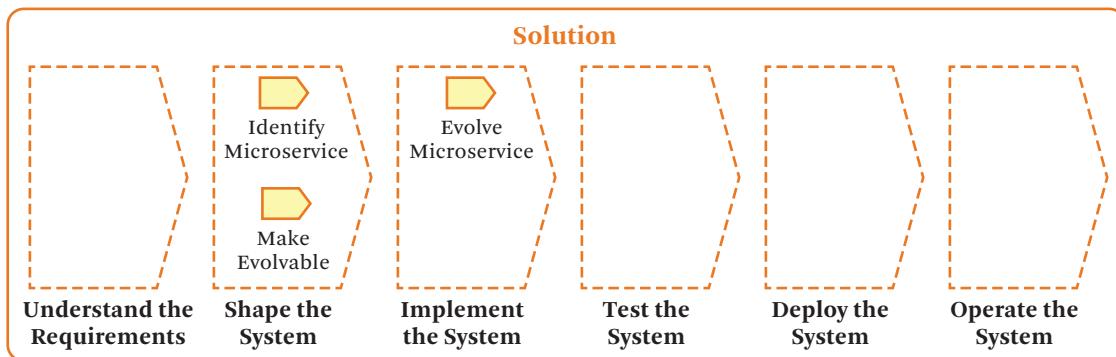


Figure 17.15 Microservices Lite coverage of the kernel solution activity spaces.

resulted in components tightly coupled with either language, infrastructure, or data store. The idea of microservices each being independently deployable seems to be an attractive answer.

Of course, having many microservices each running separately raises other problems such as

- how to manage and coordinate their execution;
- how to propagate the change of data from one microservice to others;
- how to manage the security of each microservice; and so on.

Fortunately, there are a number of approaches to deal with the above problems, and cloud providers such as Amazon Web Services provide standard mechanisms to solve them. This allows developers to concentrate on the application rather than on the low-level infrastructure plumbing that now happens behind the scenes. As a result, developers can focus on realizing user requirements, push out functionality to the users quickly, get user feedback, and innovate. Such rapid cycles are ultimately the highest value of microservices.

17.8

Progress and Health of Microservice Development

What we have demonstrated in the above sections is a single pass on how a microservice is developed from its identification to its evolution. The progress on this development is captured through the Microservice alpha state cards, as shown in Figure 17.5.

As their endeavor progressed, Smith's team found that not all work was directly related to implementing requirements, as we discussed in Chapter 16. There was

additional work including identifying microservices, building deployment scripts, coding, and testing. All of this work required time. What the team realized was that the Microservice alpha state provided a way to help them think about and plan all the tasks they would have to do to get the Software System to a state where the customer would be happy with the result.

Thus, by the time they explicitly applied the Scrum Lite, Use Case Lite, and Microservices Lite practices, they had quite a number of alphas in addition to the kernel alphas. To recap, they had

- Sprint;
- Product Backlog Item;
- Use-Case Slice;
- Use Case; and
- Microservice.

Someone outside Smith's team was looking at how they were running development with Essence and this latest practice. He asked, "For a small endeavor like ours, isn't this too many things to check? It seems that you have many cards!"

The usually quiet Grace was quick to reply, "When we are doing the actual work, each of us will only focus on achieving a few alpha states, for example Use-Case Slice: Verified, and Microservice: Rapidly Deployable. The states are like micro-checklists for the small chunks of work we do that can be completed each day. They help us split the work into small chunks, and give us a sense of progress during the day."

That is indeed true; each practice deals with a specific set of challenges, and would normally be useful for persons playing specific roles. Smith as the leader of the team focused most on the Essence kernel alphas and the sprint and use case alphas. The developers were mostly focusing on the use-case slice and microservice alphas. Joel, the techie guy, was especially delighted with the microservice alpha states. He always said, "The essence of Microservices is getting to Rapidly Deployable first, before adding functionality! This is a 180-degree change from old ways that implement first, before even considering how to deploy to the production environment."

What Should You Now Be Able to Accomplish?

After studying this chapter, you should be able to

- explain the purpose of microservice architecture and the problem it solves in contrast to a monolithic system architecture;

- explain the challenges connected to the microservice architecture;
- explain the concept of “data hiding”;
- explain the effect of microservice communication on performance degradation and how to prevent it;
- name and explain the alphas, work products, and activities of the Microservices Lite practice;
- name the activities used when applying the Microservices Lite practice with examples; and
- explain how to ensure that the system is ready for quick replacement of one microservice with another without affecting other parts of the software system.

Putting the Practices Together: Composition

The goal of this chapter is to introduce the concept of the composition of practices as a means to arrive at the best fitting software engineering method. In this chapter, the reader will understand

- the concepts of composition and the description of how essentialized practices facilitate the composition process and thereby help teams to learn and apply practices better;
- that while practices are separate, they are not independent, and thus composition has certain rules that need to be respected;
- the ways in which the composition of practices treats its things to work with, things to do, competencies, and patterns;
- that during practice composition the kernel forms the core of the resulting method that is then decorated with elements of individual practices; and
- that software engineering is knowledge centric and its mastery is about being able to apply appropriate practices effectively, discovering improvements as you apply them.

In this part of the book, we have demonstrated the need for explicit practices and how small development teams use the essentialized forms of these practices (along with the kernel) to help solve common challenges. We used Scrum, User Stories, Use Cases, and Microservices as just a few examples of the many kinds of practices that teams can use explicitly.

Our purpose in this, again, is to show you how to go about representing practices using the Essence language so you can do comparisons yourself and make the best decisions on which practices are best for your team given your own endeavor

situation. To demonstrate the use of the Essence language, we only essentialized what we deem to be the real core of these practices. This is why we referred to the practices we described with the “Lite” adjective. It is, again, not our intent in any part of this book to present arguments for or against any particular practice, nor to present arguments as to why one practice may be better than another. Moreover, it is not our position to show that these lite versions of the practices convey everything you need to know about them. Rather, we hope they will act as a solid foundation from which to learn and go on to master these practices.

When beginning this process, teams do not need to adopt all practices at the same time. Instead, they usually try one at a time. In fact, this was how Smith’s team learned and grew. They started with the kernel. Then they added Scrum Lite to improve their collaboration. They tried User Story Lite, but then decided to go for Use Case Lite because it gave them a better requirements structure. They then chose Microservices Lite to help make their software system rapidly deployable. Each time they added a practice, they gained explicit know-how. Their knowledge about software engineering grew.

Although teams *adopt* one new practice at a time, in reality they *apply* several practices at the same time. In short, teams apply a composition of practices. In this chapter, we will explain what composition is and how essentialized practices facilitate the composition process, and thereby help teams to learn and apply practices better.

18.1

What Is Composition?

As mentioned in Chapter 3, composition of practices is an operation merging two or more practices to form a method. The composition operation has been defined formally in the Essence Standard to achieve precision. The precision is achieved through the essentialization of practices, i.e., describing practices using alphas, work products, activity spaces, activities, competencies, and patterns as you have observed in earlier chapters.

The merging operation can be understood intuitively like overlaying overhead projector transparencies on top of one another to form a new image, as shown in Figure 18.1. In the figure, we have a set of Christmas decoration transparencies, which comprise the following images:

- (a) an empty corner of a room,
- (b) an undecorated Christmas tree,
- (c) a Christmas star,

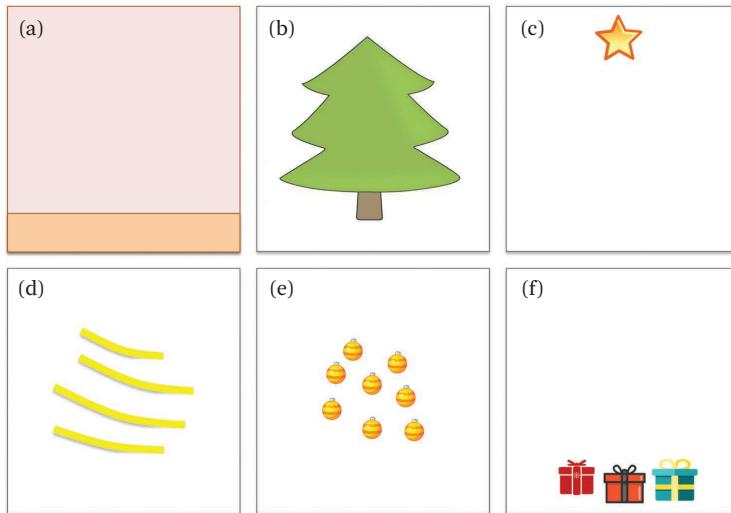


Figure 18.1 A set of transparency overlays.

- (d) streamers,
- (e) decorative balls, and
- (f) three presents.

If you overlay the above transparencies in sequence, one on top of the other, you will gradually form, in sequence, the pictures shown in Figure 18.2.

Now consider what will happen if you replace transparencies (c) and (e) with transparencies (c2) and (e2) shown in Figure 18.3. What will happen now if you overlay or compose them in sequence?

If you do this composition, you find the Christmas star on the floor, and the decorative balls become red instead of yellow. Herein lie some important concepts behind valid composition.

- Although the transparencies are physically separate, they must be drawn such that their coordinates match; otherwise you will have the kind of situation as with (c2), where the Christmas star appears on the floor. The Christmas tree in (b) anchors the position of all other images. The corollary to this in composition of practices is that, while practices are separate, they are not independent. They are dependent on the namespace within which they are composed. The Essence kernel defines such a namespace by its alphas and activity spaces. It acts as the unifying anchor.

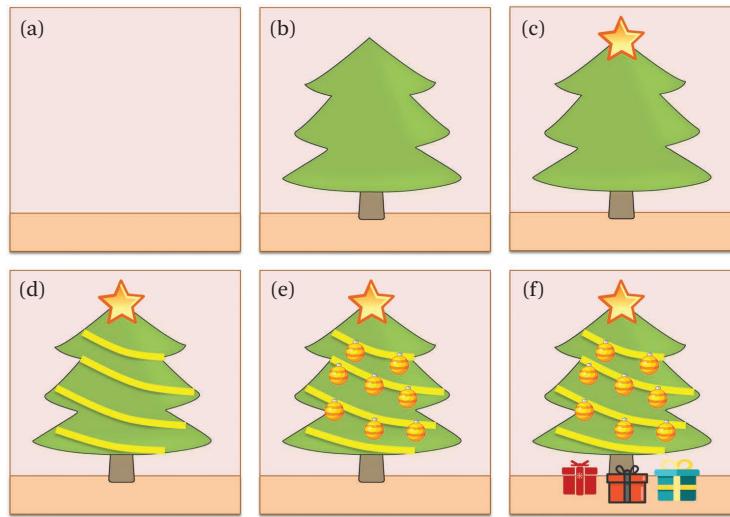


Figure 18.2 Composition as overlaying the transparencies in sequence.

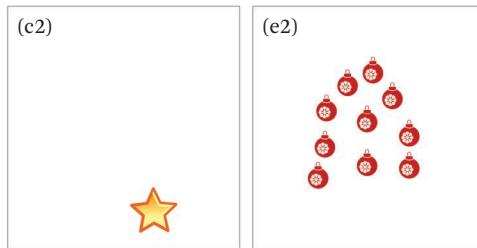


Figure 18.3 A different set of transparencies.

- As long as the namespace is adhered to, practices for the same purpose can substitute for one another. In the transparency example above, we can swap (e) with (e2) and get red balls instead of yellow balls as Christmas decorations. Theoretically, you can substitute one sub-alpha with another—for example, substituting the User Story alpha with the Use Case alpha under the Requirements alpha. In this case, the Requirements alpha acts as an anchor. However, when you replace an alpha, you will likely need to replace activities as well. So, in general, the substitution happens by replacing practices. In Smith's story, you see his team replaced the User Story Lite practice with the Use Case Lite practice. We call these "alternative practices." With alternative practices, you can usually replace one with the other.

There is another kind of practice, which we refer to as an “extension practice.” For example, Smith found that there was value in having greater stakeholder participation in their sprint review activity to help gather better feedback. (Initially, the team only had the product owner present, but in some cases Smith decided more people should be involved.) For this reason, Smith created a “Stakeholder Sprint Review” practice that provided further guidelines. This included additional activities to prepare the stakeholders for the review, preparing the team for the review, and acting on review feedback. It comprised more than just adding simple guidelines to the one single sprint review activity in that it included these many new activities. These progressed the stakeholder alpha, as well as the sprint alpha. Such changes did not, however, add any alphas. This is an example of an extension practice to the Scrum Lite practice.

Alternative practices and extension practices are only two of the many kinds of practices you can overlay on top of the kernel. In fact, there are many kinds and combinations that we will not discuss in this book. Regardless, all practices must conform to the namespace structure of the kernel, including the other practices that have already been composed.

Just as you can overlay multiple transparencies in sequence to form a final image, you can compose multiple practices in sequence to form a final method. When composing two practices together, we compose their things to work with, things to do, competencies, and patterns separately. The composition of things to do best illustrates the composition operation. Activity spaces (see Figure 18.4) provide a useful visualization tool with regard to how the two practices being composed helped Smith’s team and where potential gaps could still occur in the future. The figure shows all activity spaces in the kernel, and the activities that cover them.

It is quite clear that a number of activity spaces are not populated, in particular all activity spaces in the customer area of concerns. Not all activity spaces in the solution and endeavor activity spaces are covered. In subsequent releases, when Smith’s team dealt with larger user groups, they would need practices to fill the gaps (i.e., those in the Deploy the System and Operate the System activity spaces). Examples of such practices included Continuous Integration, Continuous Delivery, and so on, about which we will not go into any further detail.

The composition of things to work with is a little more complicated. The complication lies in the relationships between alphas. Figure 18.5 shows the composition of the alphas from the Scrum Lite, Use Case Lite, and Microservices practices. The kernel alphas Work, Requirements, and Software System provide the top-level

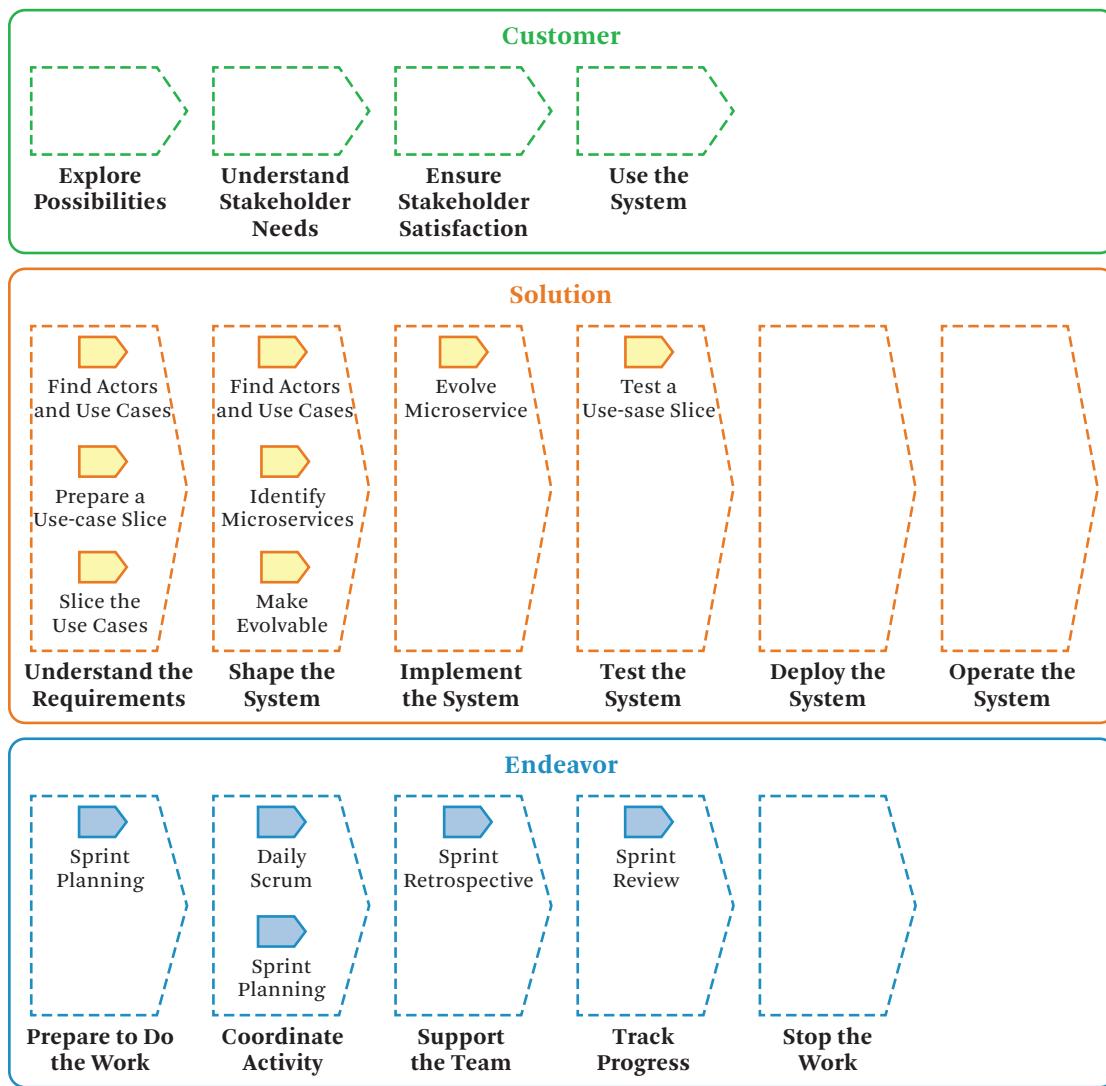


Figure 18.4 Activity spaces populated with activities from Smith's practices.

structure under which sub-alphas are grouped. The composition operation is like hanging decorations on a Christmas tree. The kernel acts like the Christmas tree on which you hang the decorations, which in this case are the sub-alphas Sprint, Use Case, and Microservice, respectively.

There is additional work needed to relate sub-alphas from one practice with sub-alphas from another practice. For example, there is a relationship between the

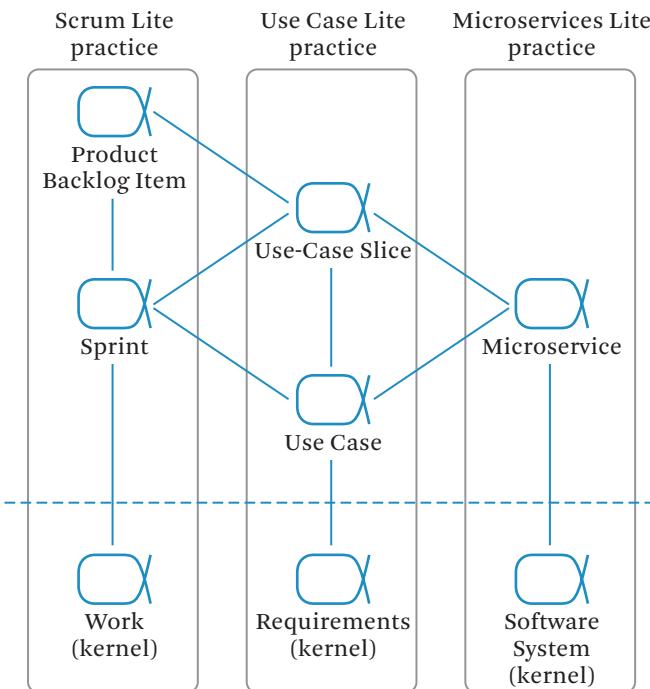


Figure 18.5 Composition of alphas in Smith’s Endeavor.

Microservice and Use Case sub-alphas. Likewise, the realization of a use case may cut across several sprints; each sprint implements some use-case slices; a use case or a use-case slice may be realized by more than one microservice. The composed set of alphas generates a simple map of all the critical things that Smith’s team must keep track of. It is also a map of checklists for the team.

At TravelEssence, in terms of progress and health, Smith relied on ensuring his team members knew all the alphas, their states, and their checklists. He was happy that these alphas had entered the team’s daily conversational vocabulary. For example, during their sprint planning and daily scrums, team members would say that they had moved a particular use-case slice to, say, the “Implemented” state, and everyone would understand what was meant. When someone wasn’t that sure, he/she could easily refer to the state cards.

When Smith’s team was first adopting the practices, team members had to learn about how to conduct activities and which work products to produce. However, gradually, as they got more familiar with the practices, what the team members still

relied on were the alphas and their checklists. These were gentle self-reminders as the team progressed through each state.

18.2

Reflecting on the Use of Essentialized Practices

As we have shown from the preceding chapters, Essence helps teams apply practices more effectively in multiple ways. First, practices described on top of Essence give practical guidance visible to development teams. Second, essentialization helps teams translate agreed-on principles into practices. Third, essentialization supports regular team feedback and adaptation through its simple graphical language. Fourth, practices described on top of Essence keep the relationship to kernel alphas and states visible across all essential dimensions of software engineering (including Opportunity, Stakeholders, Requirements, Software System, Team, Work, and Way of Working). This helps teams stay focused on the most important things as they go about their jobs, making daily decisions. This is because the essentialized practices have been organized and streamlined through the use of the kernel. Fifth, the kernel serves as a reminder, and substitute, for practices that may fall outside the current scope of interest. For example, a team applying the Microservices Lite practice may focus only on the Software System alpha and may become blind toward other important software engineering dimensions such as Stakeholders and Team. So, even though practices provide more direct guidance for teams, it is always useful to refer back to the kernel to evaluate the endeavor's overall progress and health, such as we saw with the radar diagram (Figure 11.2), back in Part II. In this sense, the Essence kernel serves as a practice-agnostic lens that the team uses to continuously be mindful of the progress and health of their overall endeavor. For instance, the alpha state checklists provide vital reminders of necessary outcomes to keep an endeavor healthy, such as the need to keep stakeholders involved and the need to listen to their feedback, regardless of which practices are in use.

Overall, one of the goals of Essence is to enable team members to become masters of their craft practice-by-practice as they learn more and more. By explicitly essentializing practices, novice practitioners have a guided ongoing path supporting their growth as professional software practitioners.

Now that you know what a practice comprises, you will know what to look for when being introduced to a practice new to you. As an example, suppose you are introduced to a new practice named Behavior Driven Development (BDD), which we will not explain. By now, you know that a practice addresses a specific challenge in software engineering. You also know that a practice comprises alphas (and their states), work products (and their levels of detail), activities, and patterns.

Thus, when trying to learn about BDD, you will be thinking about these same factors. Once you understand the elements of the practice, you can understand how it fits into and contributes to your development endeavor. It is from this discovery that you will get a sound understanding of the practice and thereby achieve mastery.

There are certainly many more practices than what we have touched upon. In your software engineering career, you will encounter many practices, some of which haven't been invented yet. After all, software engineering is knowledge centric and mastery is really about being able to apply appropriate practices effectively, discovering improvements as you apply them—all of this leading to continuous adaptation to the needs of your own environment. Essence helps teams achieve this mastery by providing a common ground reference for them to use as they continuously reflect on their way of working and improve. In Part IV, you will learn how practices can help large teams collaborate better, and also how they can help them work with a much larger set of practices.

18.3 Powering Practices through Essentialization

The value of Essence to teams is twofold: serving as a lens from which they can evaluate the progress and health of their development endeavor, and making practices explicit through essentialization. We have seen the latter in action in Part II, when Smith's team did not use any practices explicitly, but found guidance through the kernel. The kernel alphas help a team determine its overall progress and health. When teams apply practices through the lens of Essence, both the strengths and weaknesses of their approach become more visible to them all.

Essence uses the term “practices” in a way unlike how it has been used in general. Rather than keeping this term a vague concept, Essence make practices concrete and actionable by calling out each one’s alphas and work products. For example, the essentialized Scrum Lite practice identifies alphas such as Sprint and Product Backlog Item and work products like Product Backlog, Sprint Backlog, and Increment. These come with alpha state checklists and level of detail checklists that can help team members clarify the appropriate definition of done. These checklists go above and beyond the checklists provided by the standard Scrum literature. They also go above and beyond the kernel, as they are specific to a particular practice. Moreover, we have seen in this part of the book how the Scrum Lite practice, User Story Lite practice, Use Case Lite practice, and Microservices Lite practice can be defined explicitly on top of Essence to give greater guidance to a team and to help them see where they still may have weaknesses and need of improvement.

The explicit structure of each practice supported by the use of the Essence language provides the skeleton onto which additional team members can contribute their experiences. For example, they can update the checklists in the cards, and add other hints and tips to any practice elements. This, of course, depends to a large degree on the policies and constraints within each organization. However, the Essence language has been intentionally developed with the goal of being easily used by practitioners, and not just method development professionals. By reusing a consistent and widely accepted structure as we now have with the Essence kernel language, team members can more easily evolve and share their knowledge not only with other team members but also with other teams. We will elaborate more on this in the next part of the book.

This finishes Part III of the book concluding our explanation of the use and value of Essence for a single team as well as a single software system. We now look at the situation when multiple teams develop more than one system at a time. This is called “scaling up,” and it will be the topic and view we address in Part IV.

What Should You Now Be Able to Accomplish?

After studying this chapter, you should be able to

- give an example of practice composition and explain when it can and cannot be performed;
- explain what the composition could look like for two specific practices;
- explain the concept of an anchor in the composition;
- explain the meaning of “alternative practices” and “extension practices”; and
- apply composition to alphas from different practices.

Recommended Additional Reading

- K. Schwaber and J. Sutherland, “The Scrum Guide. The Definitive Guide to Scrum: The Rules of the Game” [[Schwaber and Sutherland 2016](#)]. This is the latest version of the Scrum Guide (at the time of the initial publication of this book) and should be read by anyone interested in learning more specific details on the “official” rules of Scrum.
- E. Derby and E. Larsen, *Agile Retrospectives: Making Good Teams Great* [[Derby and Larsen 2006](#)].

- K. Beck, *Extreme Programming Explained: Embrace Change* [Beck 1999]. This book should be read by those interested in learning more about Extreme Programming.
- M. Cohn, *User Stories Applied: For Agile Software Development* [Cohn 2004]. This book should be read by those interested in learning more about the User Story practice.
- I. Jacobson, Object-oriented software development in an industrial environment [Jacobson 1987]. This paper introduces the use-case construct to software engineering.
- K. Bittner and I. Spence, *Use Case Modeling* [Bittner and Spence 2003]. This book provides a comprehensive treatment of use case modeling. Though the book was written before agile became the trend, it is still a valuable treatment of the subject.
- G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide* [Booch et al. 2005]. This book should be read by anyone interested in learning more about the Unified Modeling Language.
- I. Jacobson, I. Spence, and K. Bittner, *Use-Case 2.0: The Guide to Succeeding with Use Cases* [Jacobson et al. 2011]. This ebook provides detailed guidance for applying Use Case 2.0 and should be read by anyone interested in learning more about Use Case 2.0.
- I. Jacobson, I. Spence, B. Kerr, Use-Case 2.0: The hub of software development [Jacobson et al. 2016]. This article is recommended for those interested in learning more about Use-Case 2.0, and comparison of the user story and use case practices.
- S. Newman, *Building Microservices* [Newman 2015]. This book should be read by anyone interested in gaining a more in-depth understanding of the microservices practice.

The graphic features a large, stylized, light blue letter 'V' shape. Inside the left vertical stroke of the 'V', the word 'PART' is written in a bold, blue, sans-serif font. Inside the right vertical stroke, the word 'V' is also written in a bold, black, sans-serif font.

PART V

LARGE-SCALE COMPLEX DEVELOPMENT

Our journey in this part of the book proceeds to large-scale complex development. In previous parts of this book, we looked at how a team can improve the way they work through the help of explicit practices, such as Scrum, User Stories, Use Cases, etc. Adopting these practices is relatively easy; the team learns the practice, tries it out, and then adapts it to work better for them. The adoption of these practices normally does not change how the team is organized (e.g., team size, team member roles, team communication mechanisms).

In this part, we will continue the journey with TravelEssence and understand how Smith's development team operated as part of a much larger organization. The objective here is to convey what it means to work with large-scale development and how that is different from working with small-scale development. We wrote this part of the book because many of you will work as part of large and complex endeavors. It is not the objective here to equip you with the competency needed to actually participate in large-scale development, but rather to let you have a glimpse of what it is like. We also want to highlight the fact that Essence is scalable to large-scale development, and that therefore what you have learned so far can carry you many years into your profession. You will find in your work that large-scale complex development encompasses many challenges, both collaboratively and architecturally; we have chosen to focus here, though, only on the collaborative aspects. The more advanced architectural aspects are out of the scope of this book.

Large-scale complex development involves more people. The way teams are organized, how they collaborate with one another toward a common goal, and the way they support one another in achieving their mission all differ from small-scale

development. Large-scale complex development involves not just one team, but many teams working in parallel and collaborating with each other. Although they will all use the kernel, each team may have different practices for their way of working, as well as following their organization's practices to support the collaboration between teams. In large-scale development, there is not just one group of stakeholders, but multiple groups of stakeholders, and there may be not just one software system but multiple software systems.

10 What It Means to Scale

The term “scale” can mean different things in different situations. In this chapter, we clarify what we mean by scaling and how the kernel approach guides scaling in different situations. Specifically, the reader will be shown

- that software engineering goes beyond just programming and what else it involves;
- the various dimensions of scaling (for instance, increase of the software complexity and the relevance of additional requirements);
- the challenges to scaling and dimensions to scaling (i.e., zooming in versus scaling up or reaching out); and
- the role and importance of continuous learning within development teams.

19.1 The Journey Continued

Before we proceed further, let’s dive a little deeper into Smith’s development organization. Having gone this far into the book, you should now be familiar with Smith’s development team. Let’s now see how that team fits within their larger development context. Figure 19.1 shows TravelEssence’s development organization chart.

Right at the top of the IT organization we find Cheryl the CIO (Chief Information Officer). Cheryl reported to the CEO (Chief Executive Officer) and is responsible for digitalization efforts within TravelEssence. Assisting her is Lim, the development vice president (VP), who is responsible for developing IT systems, and Tan, the operations VP, responsible for running the IT systems in their data centers. These IT systems included the recommendation engine that Smith’s team is responsible for. Jones is the development program manager responsible for the legacy Core Hotel Management System (CHMS).

A program is a larger endeavor, which usually comprises several smaller, but still potentially large, related endeavors. Different teams may work on different

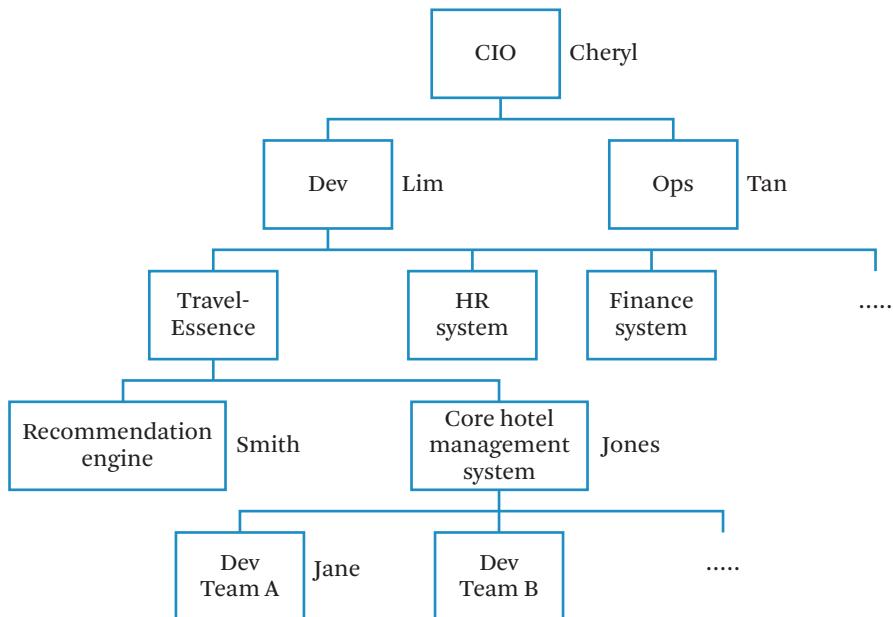


Figure 19.1 TravelEssence's organization chart.

endeavors within the program. Sometimes, “the program” also refers to the teams working on the program. This program comprises several teams, one of which is led by Jane. In such a large organization, with many systems being developed and operated in parallel, many challenges arise. Software engineering does not just mean programming; it also involves analysis and design, collaboration between teams and stakeholders, etc. Figure 19.1 shows some names for individuals that will appear in our story in subsequent chapters.

Before we continue, we want to stress that large-scale complex development is not just about having more people. The software system itself usually has more complex and more stringent requirements for performance, reliability, security, compliances and so on. For example, during peak season, TravelEssence’s CHMS needed to handle many user requests at the same time. Users do not like having anything broken, lost, or tampered with. Just imagine if someone were to hack into the CHMS and steal user data. (Indeed, we do hear of such things happening in the news.) Exactly how to satisfy such architecture-related qualities of performance, reliability, security, etc. are advanced topics that we will not cover in this book. But there definitely can be practices (built upon Essence) to address these qualities. As an example, the microservices practice is an example of a technical practice

that enables teams to evolve software systems faster and handle performance (i.e., handling more requests at the same time) and reliability (i.e., reducing downtime) requirements.

19.2

The Three Dimensions of Scaling

Looking at the large development organization exemplified by Figure 19.1, you should be able to easily recognize several major challenges.

1. Members will often have diverse background and experiences. They may or may not come into the team already possessing the needed competencies. Some may need more explicit guidance than others to do their job, and to collaborate with other team members.
2. With large-scale development, coordination of the work between members becomes a challenge. This was particularly the case for Jones's program. Smith's team did not experience this issue because it was small. But when team size grows beyond about seven people as was the case for Jones, coordination and communication often start to break down. One of the greatest challenges with large-scale development endeavors is how to go about ensuring that members discuss, share, and agree on their plans, progress, and the results of their work.
3. Different kinds of software endeavors have different risks and constraints, and therefore face different challenges. For example, Smith's team and Jones's Core Hotel Management System will definitely have different challenges. Smith's team's major challenge was how to deliver quickly, whereas Jones's challenge was about maintaining quality while functionality was being added to the system.

The above are three distinct challenges to scaling. Accordingly, when considering what it means to scale, we need to address three corresponding dimensions of scaling (see Figure 19.2), all of which may happen at the same time in specific situations.

Zooming In. We call the first dimension “zooming in,” which involves providing guidance beyond what the kernel provides. This is necessary in several cases. First, inexperienced team members need more guidance. Second, larger teams whose members have different backgrounds need more specific details in order to ensure that they have a common understanding of how to conduct development. This additional guidance comes in the form

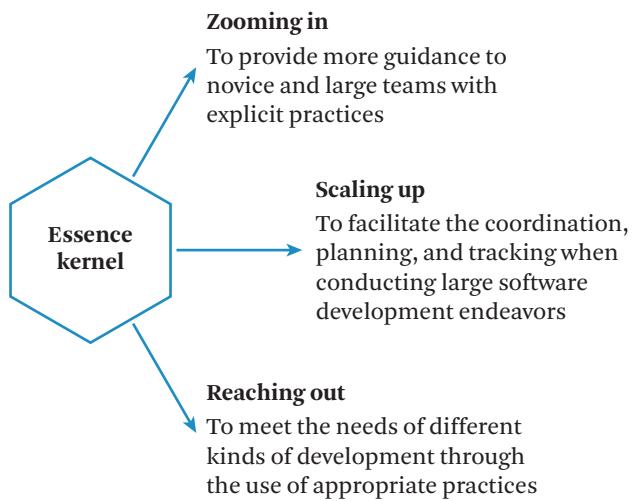


Figure 19.2 Important dimensions of scaling.

of what we refer to as practices, which are extensions to the kernel. As we have demonstrated, Essence provides a way to describe and assemble practices together in a systematic manner. For team members it is a practical way to understand how to apply practices, and how different practices work together. Essence allows teams to zoom in through *essentialization*, which is not only about making practices explicit but also about making them available through a *practice architecture*. We will elaborate on these two concepts in Chapter 20. In Part III, we gave quite a number of examples of how Smith's team received explicit guidance through practices like Scrum, User Stories, Use Cases, and Microservices. By making the alphas, alpha states, work products, activities, and patterns explicit, Smith's team generated clear explicit guidance on how to proceed and continuously improve. Chapter 20 will continue this story, but taking a larger view in the context of TravelEssence's organization chart.

Scaling Up. We call the second dimension “scaling up.” Scaling up is about expanding the use of the kernel from a team with a small number of members and low complexity to endeavors involving large numbers of people and systems with high complexity. “Scaling up” happens in large development. This is the case with Jones's Core Hotel Management System (CHMS) development program. Jones had to manage his development teams and also interact with other development teams, such as Smith's. This kind of large-scale development is known as enterprise systems development. There are

other kinds of large-scale development such as building product lines (e.g., a smartphone series with various screen sizes and configurations) and large-scale systems and software engineering (e.g., building an aircraft). These are all extremely challenging development scenarios and are beyond the scope of this book. However, the Essence kernel can be extended with practices for such complex development. What we focus on here is providing a glimpse into large-scale development, and how explicit practices can help, by diving into Jones's development program in Chapter 21 and comparing it with Smith's development team (which we saw earlier in Part III).

Reaching Out. The third dimension is "reaching out," which means extending the kernel with appropriate practices to meet the specific needs of different kinds of development. At TravelEssence, Lim's development organization not only developed the hotel management system, but also other IT systems: some small, some large. Small development included applications such as automating spreadsheets to produce management reports. Large development included applications such as the hotel management system, human resource systems, and finance systems. Some of this development was conducted in-house, while other development was outsourced to another company. (Outsourcing refers to contracting work to an external organization.) Cheryl was responsible for leading the digitalization efforts, which included applying facial and emotion recognition to authenticate users, capturing user feedback and the results of data analytics, and artificial intelligence. These required new technologies involving much research and exploration as well as collaboration with expert organizations (i.e., organizations and service providers who specialized in these technologies). Each development endeavor had its own set of challenges, and often required its own set of practices.

Reaching out is also about selecting appropriate practices and composing an appropriate method to address the risks and challenges for a particular software endeavor. A simple approach is to empower the team to build their own method. For example, in Part III, we saw how Smith's team selected practices to address their challenges. Once it has been shown that a particular set of practices is useful for a particular kind of endeavor, organizations may choose to make known to their teams a selection of pre-composed practices. Thus, instead of composing their own methods, teams can choose from pre-composed methods. For example, Smith's development organization had evolved a number of such pre-composed methods, e.g., for small development, large enterprise enhancements, or exploratory development. In most organizations, each pre-composed method comprises mandatory

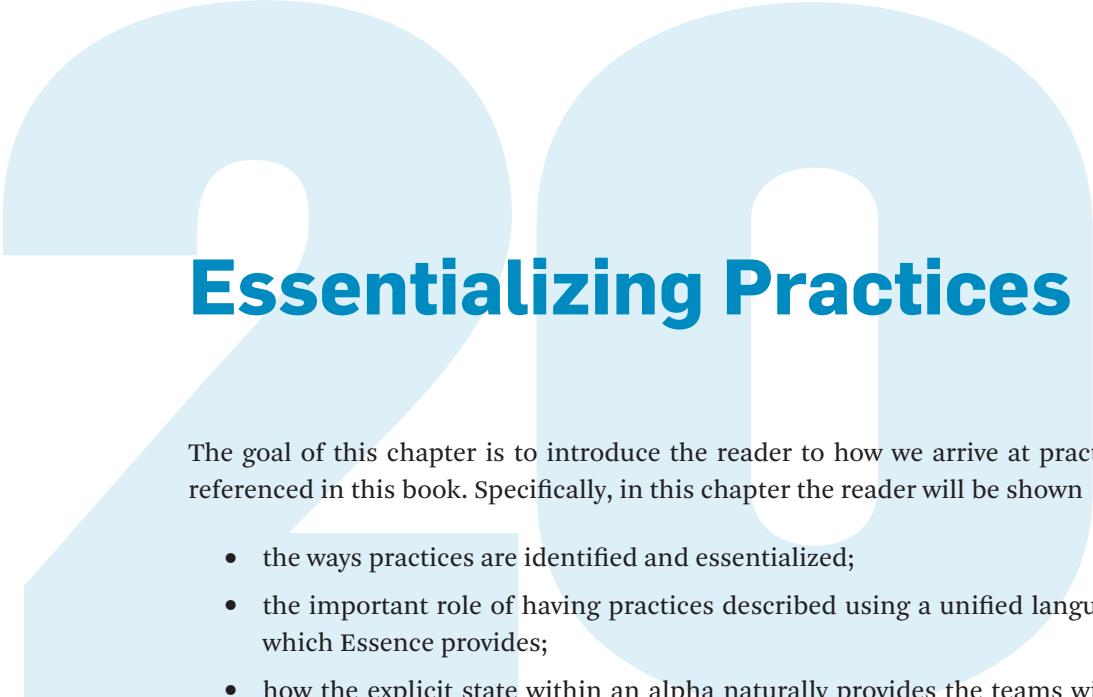
practices on top of the kernel to jump-start teams through explicit guidance. These practices can be about collaboration, or else about architecture and technical aspects of development. Teams are still permitted to add additional practices as needed and they are encouraged to improve their practices as they continuously learn through their development effort. Reaching out to different kinds of development does not just mean making practices available, but also training and coaching teams about those practices and transforming them into a learning organization. This is the focus of Section 23.1.

In these more complex cases, it is important to provide tools and mechanisms for teams to self-manage. Self-management means taking responsibility for one's own behavior. Self-management has been shown to be an essential management approach for successful large-scale development. The kernel, with its alphas and structured approach to explicit practices, provides the tools for team members to design the practices needed. Examples of such practices include the way teams collaborate with other teams (sometimes referred to as Team of Teams) and the way teams ensure their work is aligned with that of other teams (sometimes referred to as Periodic Alignment). Traditional approaches to scaling often rely on strengthening the command hierarchy (e.g., creating more managers that direct team members rather than encouraging increased self-management) and adding more checks, which can lead to bottlenecks and unnecessary conflicts. Modern approaches emphasize self-management, self-organization, and even self-governance. Essence supports whatever approach your case requires. All these aspects are elaborated further in Chapter 21.

What Should You Now Be Able to Accomplish?

After studying this chapter, you should be able to

- list and explain the key challenges to scaling;
- explain with examples the meanings of zooming in, scaling up, and reaching out as the three key dimensions of scaling;
- explain the importance to learn, modify, and enhance the chosen practices; and
- compare the modern approach to scaling with the traditional approach to scaling.



Essentializing Practices

The goal of this chapter is to introduce the reader to how we arrive at practices referenced in this book. Specifically, in this chapter the reader will be shown

- the ways practices are identified and essentialized;
- the important role of having practices described using a unified language, which Essence provides;
- how the explicit state within an alpha naturally provides the teams with a view of the whole process and keeps energy focused on full completion; and
- the concept of practice architecture as a layered composition of practices (with more generic practices at the bottom and more specialized practices at the top).

The first dimension of scaling, as presented in Chapter 19, is about zooming in on details and providing the necessary explicit guidance to practitioners. Back in Part III, we demonstrated how teams in TravelEssence took reusable practices and adapted and applied them to the specific situation they faced in their own endeavor. These practices were presented in their essential form, in terms of alphas, work products, and activities (i.e., using the Essence language to facilitate the understanding and use of the practices by the team). So, where do these practices come from, and how do these practices end up in this essentialized form? Moreover, how do we arrive at the practices like the ones we illustrated in Part III? We answer these questions in this chapter.

20.1 Practice Sources

Let's first discuss the variety of sources where practices come from. These practices are just good ways to deal with specific problems in certain contexts, which practitioners in the industry have either discovered or invented.

Some of these practices are given names, while others are not. The popular and more successful ones spread across the community through numerous software engineering conferences and events. These practices have been presented, debated, tried, and tested before ending up in papers and books and adopted by fellow practitioners as practices within their methods. Examples include Scrum, Use Cases, Extreme Programming (XP), and recently, at the time of our writing this book, large-scale development frameworks—such as the Scaled Agile Framework (SAFe), Disciplined Agile Delivery (DAD), Large-Scale Scrum (LeSS), and Scaled Professional Scrum. These large-scale agile development methods have been developed based on original small-scale agile development methods such as Scrum and XP. They have added higher layers of coordination between teams to meet business objectives.

Practice sources are not limited to agile methods and include traditional ones (often adapted to agile). Some of these address how to deal with systems engineering, object-oriented analysis and design, and use cases. Moreover, each large organization, due to its unique growth path, evolves its own unique set of practices. Since these practices are very domain specific, they are not as widespread as the more generic and more popular ones.

Organizations typically evolve their own practices as they find better ways of working. Lim, the development VP from our TravelEssence case, emphasized this strongly. He would hold regular meetings with program managers and team leaders to stimulate discussions leading to continuous improvement in their way of working. Through this effort, Lim drove the evolution of a number of practices related to improving quality of TravelEssence's products, delivering those products, and improving collaboration between individual teams within his organization and operations (which was led by Tan, the operations VP). These homegrown practices initially appeared only in TravelEssence's conversations and meetings. They were not labeled with brand names, and were not made explicit through documentation. (This often happens in many companies.) However, once his organization adopted Essence, Lim understood that he had a solid architectural structure he could rely on to coach his colleagues and sustain these improvements well into the future as his enterprise continued to grow.

20.2

Monolithic Methods and Fragmented Practices

Because of the large number of methods, in the past people used different terminology, or used the same word, but with different meanings, and organized their

practices in very different ways. This caused these methods to become either too monolithic or too fragmented.

A method is “monolithic” when it is non-modular, making it difficult to add, modify, or replace (substitute or exchange) a practice from outside (e.g., another author’s practices or another group’s improvement ideas), or to replace an existing practice with another one.

Methods like Rational Unified Process (RUP), Scalable Agile Framework (SAFe) [[Knaster and Leffingwell 2017](#)], Disciplined Agile Delivery (DAD) [[Ambler and Lines 2012](#)], and Large Scale Scrum (LeSS) [[Larman and Vodde 2016](#)] are monolithic by nature. RUP is an iterative software engineering method. SAFe is a lean software and systems method. DAD is a process decision method. LeSS is Scrum applied to many teams working together on one product. Since this book is for students new to software engineering, we will not dive into these examples of monolithic methods. Instead, we will consider the essence of how these methods organize practices.

Monolithic methods are an amalgam of practices that are not clearly distinguishable from one another. Practitioners rarely, if ever, apply any of these methods in their entirety, but only parts of them. The question then is which part? How is a part extracted and clearly defined so team members understand what is expected of them? Another question you might be asking yourself is: How can we make it easier for teams to reuse parts of a method that make sense just for their team? A good answer to these questions is essentialization, as we will soon describe.

A method is “fragmented” when it is difficult to see or understand how its constituent pieces fit into the whole. Fragmented practices are loose bits of guidance with different concepts and jargon. They usually comprise some kind of summary to address a certain challenge, such as product design or team collaboration, with no regard for how they relate to other practices. Professionals who encounter these practices need to spend the effort to link concepts and terminology.

Many times, different terms end up being used to denote the same thing. For example, “iterations” and “sprints” have largely similar meanings. Some use the term “squads” for teams, “tribes” for large teams, and “chapters” for communities. Although it may be at times useful to use different descriptors to highlight certain qualities, using different terms especially when there are so many practices confuses software professionals who already have much to learn. For example, Scrum’s use of the term “sprint” metaphorically emphasizes that a team should have everything ready to achieve the iteration goals. A sprinter definitely doesn’t adjust his shoes or detour during the 100 m race. No, he has absolute focus and will not be distracted. Nevertheless, a sprint is still an iteration, a time-box when things get

done. Thus, there are two terms, “sprint” and “iteration,” to refer to essentially the same thing. With fragmented practices, though, there are a lot more terms referring to the same thing. Some terms may have overlapping meanings. They are similar but somewhat different, creating much confusion to professionals. It is hence difficult if not impossible to understand how multiple practices fit together. You cannot easily fit a practice from one source (method) with another practice from another source. If you ever want to merge practices, the two practices must be described using the same language. This is what Essence provides.

This, then, brings us to essentialization. From a methodology point of view, it means identifying and scoping a practice from a practitioner’s perspective, describing it using the Essence language such that it is compatible with other practices. In the larger picture, it is also about introducing such a practice to teams, teaching and training them such that they can learn, apply, and adapt the practice to their context.

20.3

Essentializing Practices

You have seen that essentialized practices can come from many sources. Essentialized practices can come from explicitly calling out practices from well-known practice sources (like SAFe, RUP, XP, etc.). By doing this, we make the practice more precise, easier to teach and apply, even within different contexts. A natural consequence of essentialization is that practices become easier for teams to compare and choose.

The essentialization of a practice is the activity of making a practice explicit by describing it using the Essence kernel and language as a platform. The practices that we presented in Part III exemplify this.

In the past, practices have been implicitly described or described in a non-standard way, and teams have found them difficult to understand and apply. Teams also didn’t have an answer to the question of “How should we start describing a practice?” However, with Essence, we now have a language—a well-defined, widely accepted way to explicitly define practices.

The Essence standard tells us what a practice is and what a practice is intended to be used for. By definition from this standard, “a practice is a repeatable approach to doing something with a specific purpose in mind.” A practice expressed in the Essence language provides explicit guidance to teams by clarifying the things to work with, the things to do, the required competencies and patterns. In particular, alpha state progression emphasizes a view from beginning to end—for example,

from Requirements: Conceived to Requirements: Fulfilled, or from Use Cases: Goal Established to Use Cases: Stories Fulfilled. This end-to-end view channels teams' energy toward the goal of full completion. The other elements of an essentialized practice (e.g., activities and patterns) provide guidance and approaches to make alpha state progression more effective and hence faster. Appropriate competencies are of course necessary and specified; without them, the team cannot achieve good progress.

The size of an essentialized practice depends on the number of elements in the practice (i.e., total number of alphas, work products, activities, competencies, and patterns). In general, the smaller the number of elements in a practice, the easier it is to understand and apply. There is no lower limit for how small a practice can be, but there is a practical limit. This is because smaller practices result in a larger number of practices, which places a cognitive load on teams who need to select and compose them.

Given the nature of both the kernel and the language of Essence, we can describe many practices using it as a vocabulary. Every such practice should be designed to be reusable, meaning that it could be applied in different contexts and scope. This will become more apparent as you read this chapter and the next.

20.4 Establishing a Reusable Practice Architecture

Having extracted practices this way, it is important to have a way to organize and present them so that they can be reused by teams that select their own method. Making a practice reusable means ensuring that each practice selected is a proven practice and that the elements described for it are essential elements only. This is one reason why essentialization is so important. Moreover, often teams need many practices that are designed to be composed together in what we call a practice architecture. Even if practices are separate elements, they are not independent.

Let us take an example composing two practices: use cases and microservices. When talking about use cases here, we talk about the more complete Use Case 2.0 practice (not the Use Case Lite practice we talked about in Part III); it also has a part describing how a use case is realized by collaborating system elements, in this case microservices. The microservice practice guides people to create small rapidly evolvable and deployable microservices. These two practices have overlapping elements—namely, use case, use-case slice, and microservice—the effect of which has to be described when composing the two practices. A simple way to think

about practice composition is to think about a merge operation. For example, one practice may require an alpha to have a certain group of checklist items and another practice may require the same alpha to have a different group of checklist items. The composition or merge operation will merge these two groups of checklist items together into that single alpha. This same idea can be applied to the composition of work products, activities, and so on.

One way to construct a practice architecture is in a layered form, with more generic practices at the bottom and more specialized practices at the top.

In our TravelEssence story, Lim, the development VP, understood the need to learn and share practices across teams. It was because of this continuous sharing and evolving that he was able to build up a systematic knowledge base, allowing the company to become a learning organization with a strong team of developers.

However, it wasn't like that a few years previous, before Essence was introduced to the company. Back then, TravelEssence was smaller, and team members worked fervently to deliver the software systems. Teams used whatever practices they deemed to be appropriate. Now as TravelEssence grew, with new applications, and more members, their old way of working needed to be adapted to fit their current size and organizational structure. It had to change.

Fortunately, Lim was immediately attracted to the work and the contributions of Essence, including the essentialization of practices. He was able to reuse practices and the language and terminology across teams, following an accepted standard. Some members of the teams were skeptical at first, but became gradually convinced when they saw how speaking the same language of practices reduced unnecessary misunderstanding and helped them focus on what they needed to do (i.e., deliver great quality software).

Smith supported what Lim was trying to achieve, which was why he strongly backed the use of a practice-based approach to run his development team.

With some help from Smith, Lim developed a practice architecture to be used for TravelEssence (shown in Figure 20.1). Lim needed practices not only for development teams, but also for teams all across the IT organization. He organized the practice architecture into two layers, separated in the figure by a dashed line.

20.4.1 Development Practices

At the bottom layer in the figure, there are practices for development teams. This includes practices like User Story, Scrum, Use Case, and Microservices, the Lite versions of which were presented and discussed in Part III. In addition, the figure shows several other practices that Lim found important.

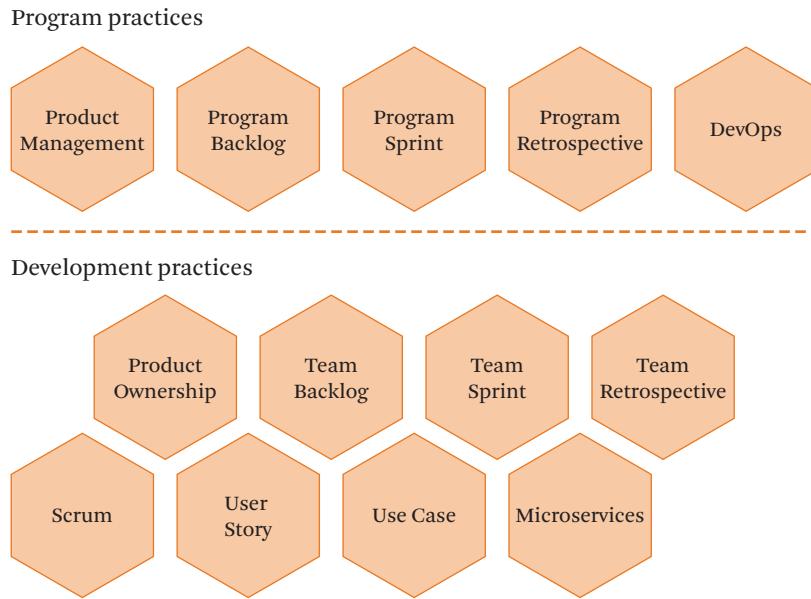


Figure 20.1 Practice architecture for TravelEssence.

Product Ownership. This is a practice for deciding and prioritizing the kind of requirements that a team would deliver. In Part III, we saw that Angela played the role of a Product Owner. This practice provides more explicit guidance to let those in her position do their jobs better.

Team Backlog. This practice provides guidance on how to manage and track items in a team's backlog.

Team Sprint. This practice provides guidance for how teams can work iteratively, from sprint planning to sprint delivery.

Team Retrospective. This practice provides guidance for how teams can continuously reflect on how they are working and making necessary improvements.

If you think about it, Scrum, which we presented in Part III, does include product ownership, team backlog, team sprint, and retrospective, so why list these four practices instead of just Scrum? As we have just shown, Scrum can be represented as a composition of four practices, so that teams can choose to use only some practices in Scrum and not be forced to use all of Scrum. This makes the practice architecture more reusable.

20.4.2 Program Practices

You can think of a “program” as a large “project” involving smaller “projects.” Organizing the work within a program can be complicated because you need to balance priorities from different stakeholders, and manage the dependencies across the teams within the program. Moreover, the work may cut across different departments.

In our story, Lim recognized the importance of having practices that go right from business departments to development to operations. He called these “program level practices.”

Product Management. This practice is about managing the source or requirements and analyzing them before they are placed in the program backlog for development.

Program Backlog. Once the requirements are analyzed and agreed on for development, their progress is tracked using the program backlog practice. Like Team Backlog, this practice provides guidance on the prioritization and acceptance of the program backlog items.

Program Sprint. Programs under Lim applied an iterative cycle to deliver Program Backlog Items. These are allocated to development teams. In the next chapter, you will see how Jones, a program manager for the Core Hotel Management System (CHMS), used this practice with his subordinate teams.

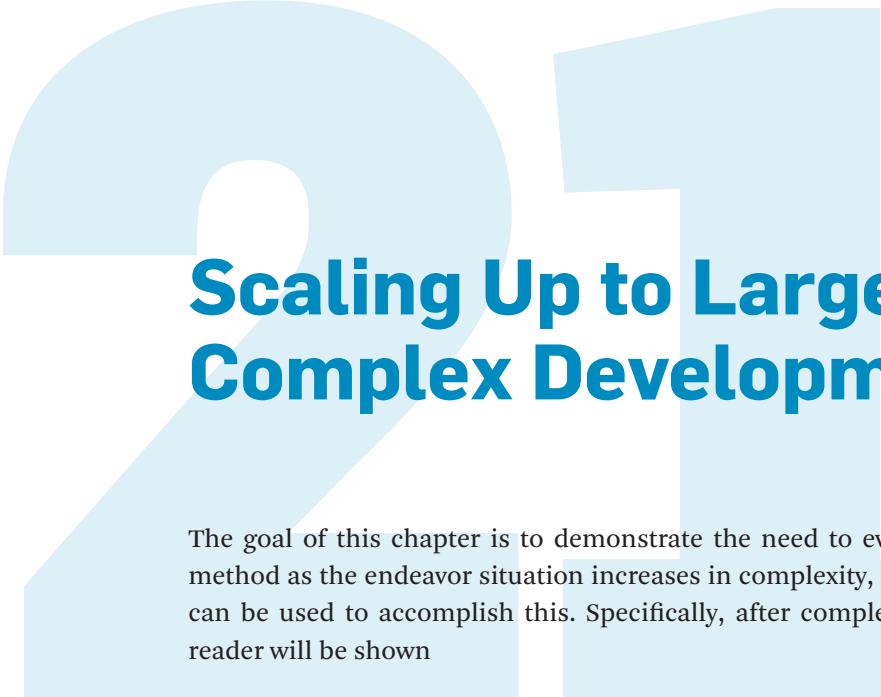
Program Retrospective. Lim’s programs applied regular retrospective to analyze their way of working and seek improvements.

DevOps. After new requirements such as use-case slices have been implemented and integrated, production-like test environments have to be updated too. At TravelEssence, Tan managed the automated deployment pipeline that maintained these systems, and was responsible for making them run smoothly, taking care of processor and disk capacities. Basically, an automated deployment pipeline is a set of automated scripts and mechanisms that performs checks such as compilation checks, and testing to ensure that the deployed software works not just on the developer’s machine, but everywhere the deployed software needs to run, even in the production environment. DevOps is the practiced set of technologies used to achieve this.

What Should You Now Be Able to Accomplish?

After studying this chapter, you should be able to

- list and explain the sources from which practices might emerge;
- explain why organizations typically define their own practices;
- explain the term “monolithic method,” providing some examples;
- give examples of different terms that are used in software engineering to denote the same thing, and explain why that is problematic;
- explain the term “alpha state progression” together with an example;
- give an example of a practice architecture; and
- describe the difference between development practices and program practices.



Scaling Up to Large and Complex Development

The goal of this chapter is to demonstrate the need to evolve the development method as the endeavor situation increases in complexity, and the strategies that can be used to accomplish this. Specifically, after completing this chapter, the reader will be shown

- the sources of complexity in a software endeavor;
- the bottlenecks that might appear in the endeavor if the management dependencies among subsystems and teams are not reduced;
- how scaling was addressed in the past (when technology was the bottleneck) compared to today (when human interaction is the bottleneck);
- the extended view of practice architecture, including the team layer;
- the important role of team alignment toward a common vision, supported by clear responsibilities and division of work; and
- the important role of synchronization for team leads, to align their deadlines and priorities.

When we talk here about “large-scale development,” we mean scaling to development that involves 30 people or more. The number 30 is not a magic number. Some consider 50 or more to indicate large-scale, some 100 or even more.

In large-scale development, the team members are often distributed across geographical locations. They may be in the same building, but we frequently encounter large development teams being spread across different cities and even different time zones. These teams will likely have multiple requirement sources coming from different customers at any point in time. They may also be serving different opportunities from different stakeholder groups. Stakeholder groups might be related or

independent of one another. Moreover, each team might have different skill sets and competency levels.

One of this book's authors has applied Essence practices to a large development endeavor with several hundred people. The trick is to reduce and manage dependencies among subsystems and teams. If this is not addressed, bottlenecks appear quickly and collaboration becomes difficult.

21.1

Large-Scale Methods

There are many approaches to large-scale development. In the past, computers were very expensive, and development endeavors took a long time. It wasn't the right environment for rapid prototyping. The Capability Maturity Model Integration (CMMI), which helps teams improve processes, and the Rational Unified Process (RUP), an iterative software engineering method, were both used mostly for large-scale development. In today's modern world, though, computing is becoming cheaper, and technology is less of a bottleneck compared with human interactions. Thus, we see the rise of the large-scale agile development methods we already mentioned, like DAD, LeSS, SAFe, and so on. (It is outside the scope of this book to further describe these methods.)

In our TravelEssence example, Lim, the development VP, organized his practices into layers according to their scope of influence, as shown in Figure 21.1.

- The team layer at the top of the figure comprises practices useful for teams to work effectively together to deliver great quality software. You have already seen how Smith's team was able to work effectively throughout Parts II and III. Jones heads the team for the core hotel management IT support. Thus he faces the challenges we describe here.
- The program layer at the bottom of the figure comprises practices to help coordinate the work across teams and across departments or IT systems. At TravelEssence, for example, this might involve the collaboration between Smith, Jones, and Jane to evolve the Core Hotel Management System (CHMS).

Note that all these practices are similar to those shown back in Figure 20.1. The practice architecture had since been customized by Lim and his colleagues.

Before continuing our discussion, we would like to highlight the fact that the team layer practices in Figure 21.1 are a decomposition of Scrum. Scrum is what we call a composite practice (see Figure 21.2). Although we presented Scrum Lite as a single practice in Part III, here we want to show how Scrum can be composed.

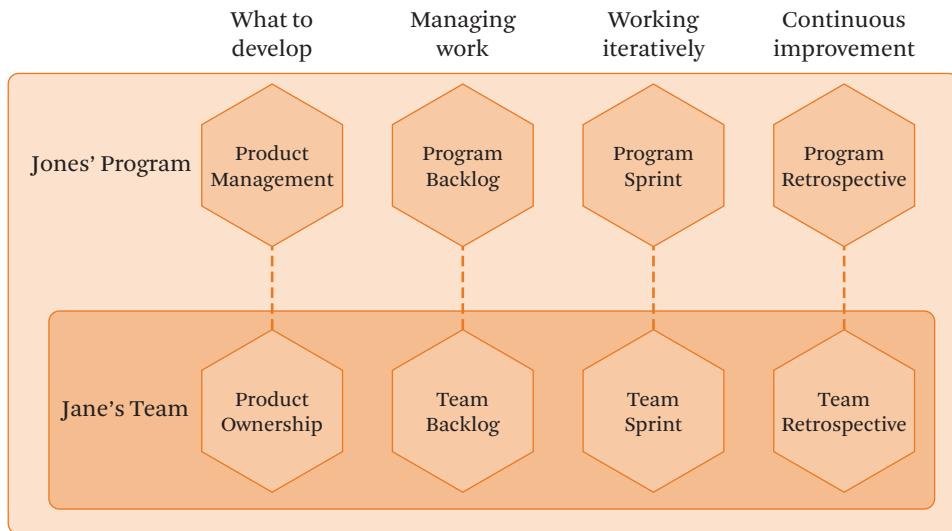


Figure 21.1 Practice architecture organized into layers.

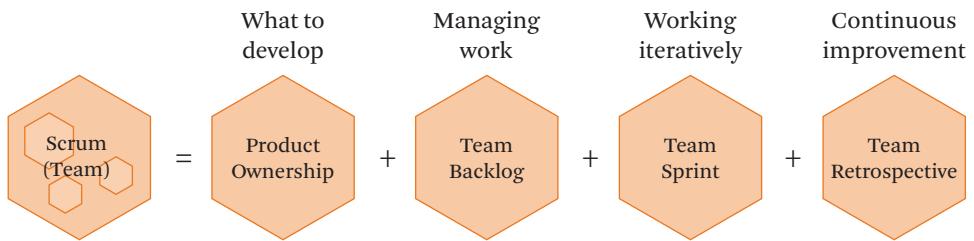


Figure 21.2 Scrum as a composite practice.

It is in fact a composition of product ownership (what to develop), backlog management (managing the work), team sprint (working iteratively), and retrospectives (continuous development), all at the team level.

The upper layer, the program practices, uses the same notions, but applies them at a larger scale. At the program layer, Jones was not working directly with individual team members, but through representatives such as Jane. Other than that, the concepts at the program layer are very similar to those at the team level, except that they operate at a larger granularity, with a longer planning horizon.

The practice architecture shows how each practice contributes to helping the organization work effectively at each level, as seen in Figure 20.1. Even though

organizations have different problems at different organizational levels, they can reuse the same principles to solve them.

In a similar way, Figure 21.1's lines linking practices show a simple relationship to indicate that these practices should be applied together to achieve synergy. For example, the team sprint and program sprint share the same notion of having teams working in a cyclic manner, with clear and periodic schedules for when to conduct planning, reviews, and retrospectives. This periodic alignment worked well for Smith's team and Jane's development team, who both worked on weekly sprints. In particular, it helped coordinate dependencies between them, such as when Smith's use-case slices depended on Jane's use-case slices.

Jones, the development manager for the Core Hotel Management System (CHMS), was responsible for several development teams working collaboratively to evolve and enhance the CHMS. The periodic cycles at the program level were longer, due to cross-team dependencies. Jones and his colleagues concluded that a monthly cycle would work well for his program. Lim, the development VP, would review the progress of CHMS on a quarterly cycle to determine if budgets should be extended or changed.

These practices all apply the same principle of repeating certain practices iteratively. It is natural, therefore, that their cycles should somehow match one another. For example, it makes sense for team sprints to fit within program sprints. This also means that these practices are best applied together to give the greatest benefit.

The same goes for each column in Figure 21.1, which will be described in the next section. However, to keep within the scope of this book, we will restrict our discussion to the program and team layers.

21.2

Large-Scale Development

TravelEssence was in fact operating in a large-scale manner. In Part III, we saw how Smith's team applied several practices to effectively run its endeavor. As the team was part of a larger development organization, its recommendation engine drew data from TravelEssence's Core Hotel Management System (CHMS), discussed in the previous section.

Recall that CHMS is an existing legacy system that provides critical functionality for many customers within the hotel management business. The users of this software were constantly requesting enhancements and it was Jones's primary responsibility to assess those requests and manage the agreed-upon CHMS enhancements. Because of the number of enhancements that were constantly being approved, the work required multiple teams to be making changes to the CHMS core

product at the same time. This obviously added a level of complexity to the management of the work. We shall use the experiences of one of those teams, led by Jane, to explain how the multiple development teams worked together as part of Jones's overall program. Because of the need for these teams to be working in parallel, some additional practices were necessary to help ensure the work was coordinated properly. Figure 21.1 depicts some of the practices that Jones and Jane's team used.

Let's start by discussing the two layers of practices identified in Figure 21.1, referred to as Jones's program layer and Jane's team layer. The columns in each layer highlight practices that are similar, but applied to a different target audience. Each column represents a distinct concern of development, including agreeing on what to develop, managing work (through backlogs), working iteratively, and making continuous improvement (through retrospectives). It should be noted that these four concerns are perspectives that Lim came up with, and not general concerns from the Essence perspective. We will elaborate on each column, the practices therein, and how Jones and Jane benefited from the practices they applied to help them in their large-scale development challenge.

Large-scale development practices are not simple practices. They span beyond the realm of the small team, and much that we encountered in Part III. It is not an easy task in a classroom setting to replicate what happens in the real world of large-scale development. It is also not easy for new students who work mostly alone to visualize how large teams operate. Hence, for the rest of this chapter, we will intersperse the explanation of the above practices with the TravelEssence story. This will help you appreciate how large-scale teams actually collaborate, and the dynamics involved.

21.3

Kick-Starting Large-Scale Development

Running large-scale development is not just about selecting and applying practices as we have seen so far. There are also steps to kick-start the process, similar to those we explained in Chapter 13 for small-scale development, except that we have more practices (as mentioned above). The steps are as follows.

1. Understand the context through the lens of Essence.
2. Agree on development scope (where it begins and where it ends) and checkpoints.
3. Agree on practices to apply.
4. Agree on the important things to watch.

We will review each of these steps in the following subsections.

21.3.1 Understand the Context through the Lens of Essence

Understanding who is involved in the development, as well as the current health and progress of the endeavor, can easily be found by walking through each kernel alpha and assessing its state, along with providing rationale for each assessment; see Table 21.1. In the interest of simplicity, we will just look at the alpha states achieved for Jones's program team.

21.3.2 Agree on Development Scope and Checkpoints

The scope of development context can be defined easily by agreeing on what should be achieved before the beginning of each cycle (or sprint), so that the team or program can start effectively, and by agreeing on what the team or program should achieve by the end of each cycle. Again, these can be defined readily through alpha states (see Table 21.2).

At TravelEssence, the criteria for the sprint to start served as a gentle reminder regarding the health of Jones's program. In particular, the team had been lacking Stakeholder involvement. They also lacked the understanding of the value of the Opportunity in preceding sprints (compare Tables 21.1 and 21.2). Jones pointed out these challenges to Cheryl and the business departments. It wasn't easy, but they knew that these issues had dragged on for too long, and had to be fixed. To solve them, some roles were assigned. Jones would act as the Scrum Master at the program level, while Jane accepted the role as team-level Scrum Master. Working alongside them, respectively, were Samantha and Seet, who would function as Product Manager and team Product Owner, respectively. This more explicit agreement of roles overcame the chaotic nature of previous interactions. Now it became clearer to the busy stakeholders who should attend which meetings. As a result, stakeholder involvement improved.

21.3.3 Agree on Practices to Apply

The clearer indication of roles had led to expectations becoming clearer, and Jones's program was soon able to reach the Stakeholders: Involved state. But this was just the beginning. The newly appointed Samantha and Seet still needed some explicit guidance regarding how to do their jobs. This was where practices became helpful.

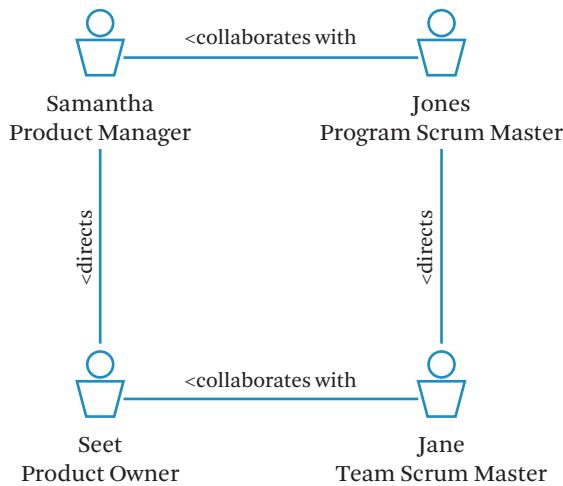
This third step in kick-starting large-scale development is about agreeing on which practices to apply at both the program level and the team level. Here, though, we summarize just the practices that Samantha, Jones, Seet, and Jane applied (see Table 21.3). The first column highlights the purpose of the practice selected

Table 21.1 Development context through the lens of Essence

Alpha	State Achieved by Jones's Program	Rationale for Achieving the State
Stakeholders	Recognized	In Part III, Smith's small TravelEssence team identified just Cheryl, Dave, and Angela as key <i>stakeholders</i> that needed to be involved. But now that the endeavor had grown, Jones's team had to be concerned about involving many more stakeholders, including the representatives of the users who had enhancement requests for CHMS. At the start of the endeavor, Jones knew he had not yet received the full commitment and involvement of all of the needed stakeholder representatives.
Opportunity	Value Established	The value of the system had been established because the users of the CHMS were already gaining great value, which is why they kept asking for more enhancements, and the successful demo of the recommendation engine would just increase the value more.
Requirements	Conceived	Jones recognized that many of the user groups had been making requests for enhancements beyond the original scope of the CHMS.
Software System	Architecture Selected	The legacy CHMS system had an existing architecture and the mobile app, through the use of microservices, had been proven to support the needs of the recommendation engine.
Work	Started	Work was underway, as parts of Jones's teams were already successfully working on enhancements to the CHMS system.
Way of Working	Foundation Established	Jones realized that the collaboration between teams was sometimes more important than collaboration within teams. Teams would need to understand what other teams were doing and give the necessary support. They would need help to do this.
Team	Formed	Part of Jones's team was already working on enhancements to the CHMS systems.

Table 21.2 Development scope for Jones's program

Alpha	State Before Sprint Starts	State Before Sprint Ends
Stakeholders	Involved	Satisfied (for deployment)
Opportunity	Value Established	Addressed
Requirements	Coherent	Fulfilled
Software System	Architecture Selected	Ready
Work	Prepared	Concluded
Way of Working	Principles Established	Working Well
Team	Formed	Collaborating

**Figure 21.3** Participants in Jones's program.

(agreeing what to develop, managing the work, working iteratively, and continuous improvement). The practices in the next two columns have already been presented. The last column indicates the kind of guidance that practices provide. For example, both the product management and product ownership practices provide guidance on how to advance Opportunity and Requirements alpha states at the program and team level, respectively.

You might wonder why there are just four rows, one for each purpose. Is that all we need? This question too can be easily answered by walking through the kernel alphas. For example, agreeing what to develop is guided by Opportunity

Table 21.3 Practices selected

Purpose	Program level Practices	Team level Practices	Kernel Alpha Progression Guidance
Agreeing what to develop	Product management	Product ownership	Opportunity, Requirements
Managing the work	Program backlog	Team backlog	Requirements, Work
Working iteratively	Program sprint	Team sprint	Work
Continuous improvement	Program retrospective	Team retrospective	Way of Working

and Requirements alphas. Similarly, managing the work is guided by Requirements and Work.

From Table 21.3, it is clear that there are no explicit practices to address challenges with Stakeholders, Software System, and Team. Of course, we can add practices to deal with these alphas, but in the TravelEssence situation they did not present significant problems and challenges. Thus, in their case they dealt with these practices without explicit guidance in the form of a practice description; they dealt with these practices, then, with their tacit knowledge. However, in other organizations, these might be major challenges and in that case defining explicit practices to monitor and progress Stakeholders, Software System, and Team may be appropriate.

Note that beyond the practices listed in Table 21.3, development teams also apply more technical practices like use cases and microservices. We do not discuss these practices here because we have done so in Part III. Here, we place more emphasis on collaboration between teams and team members and other challenges that large-scale complex development needs to address.

21.3.4 Agree on the Important Things to Watch

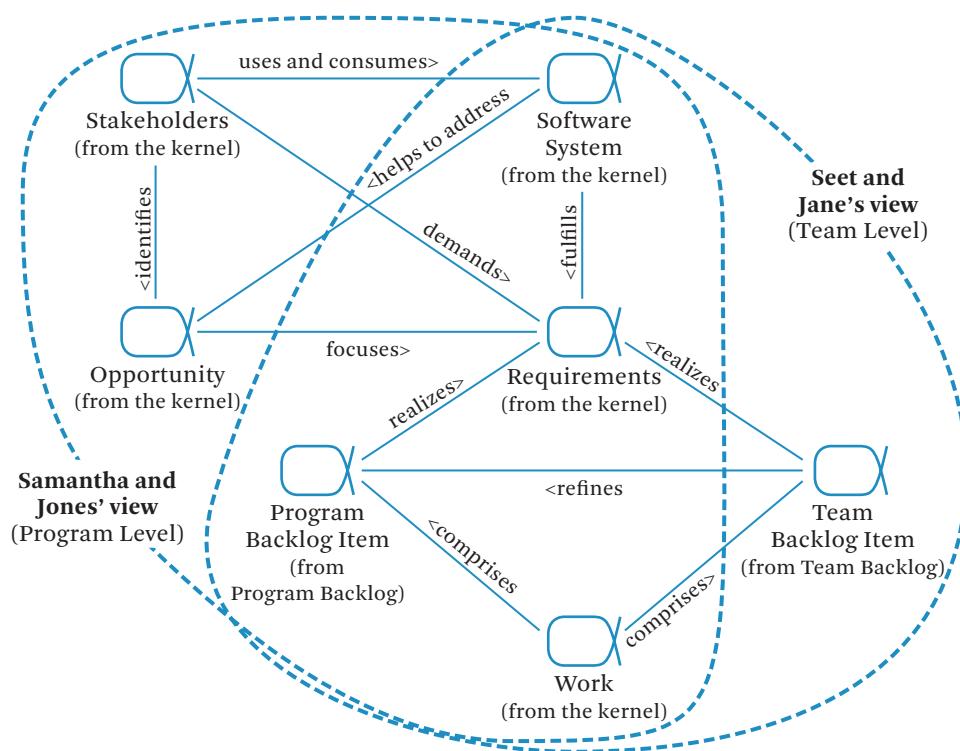
The fourth and final step to kick-start development is to agree on the important things to watch. The point of Table 21.4 is that in each endeavor, regardless of size, there are alphas from the kernel and alphas from the practice level that teams agree need to be watched to assess the progress of the endeavor accurately.

It is clear that running a large-scale development is not easy. There are just so many things happening, and it is important for everyone to focus, and filter out the noise. Everyone needs to do her/his job well and overall make sure that nothing is

Table 21.4 Things to watch

Layer	Program	Team
Alphas from the kernel	Opportunity, Stakeholders, Requirements, Software System, Work	Requirements, Software System, Work
Alphas from the practices	Program Backlog Item	Team Backlog Item, Program Backlog Item

missed. Jones would have his responsibilities and so would Jane. Figure 21.4 shows the things that Jones and Jane would be watchful of. There would doubtless be some overlap, albeit from different perspectives. Jones would have to look at the bigger picture focusing on the Opportunity, whereas Jane would have focus on progressing the Team Backlog Items, making sure the work her team was doing (Team Backlog Items) stayed consistent with overall program requirements and the agreed-on

**Figure 21.4** Things to watch in a particularly large-scale endeavor.

program work (Program Backlog Items). They both would have to be concerned with the Software System, Requirements, Work, and Program Backlog Items.

21.4

Running Large-Scale Development

To help the reader further understand what it means to run a large-scale development, we will next organize the story around the rows in Table 21.3, beginning with agreeing what to develop, then managing the work, working iteratively, and making continuous improvement.

21.4.1 Agreeing What to Develop

The first set of practices is about building the right product, and getting the teams to focus on a common vision, while being able to respond to customer change requests. This is about product ownership and product management.

As we have seen in Part III, Angela was responsible for setting the priorities of development for TravelEssence. She was playing the role of a Product Owner. Her team was small and the scope of development equally small. Hence, a single person playing the role of a Product Owner sufficed.

However, this was not the case for Jones's program. Recall that Jones had many requirements sources, and many enhancement requests coming from the many customers of TravelEssence within the hotel management business. In this case, due to the increased complexity of his requirements sources, he needed to make sure that team members were aligned toward a common vision at the program level. This would be best achieved through a product vision work product that clarified the value of what they were developing, and the way requirements were to be prioritized. As mentioned, Jones had also established a new product manager role that was filled by Samantha. Her responsibilities now included evolving the Product Vision and achieving acceptance at the program level. Second, Jones had established Seet's team product owner role, which included the corresponding responsibilities at the team level (see Figure 21.5).

With these new responsibilities, Samantha and Seet, along with other Product Owners, started to systematically agree on what to develop, and they started to evolve the Product Vision, both at the program and team level.

They all agreed that the overall theme should be to provide great customer experience over a robust software system. At the program level, Samantha and Jones, then, updated the vision.

To achieve great customer and user experience, they first agreed to reduce the number of user actions to complete a use case. Some CHMS functionality required

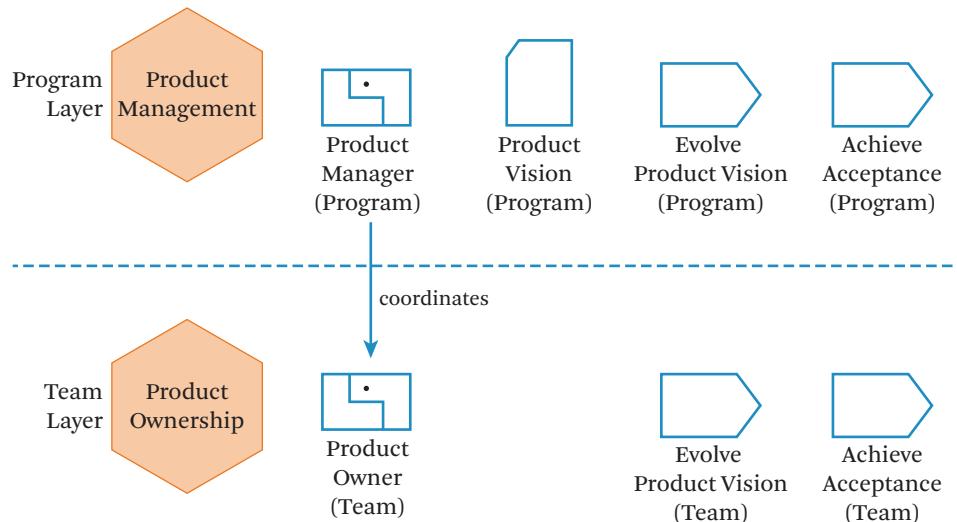


Figure 21.5 Product ownership and product management.

users to traverse through complicated form-filling steps. Because of that, the CHMS lost some customers. Thus, an important goal for the next release was to explore options for reducing the number of steps in all customer-facing use cases.

They also agreed to provide a waiting list functionality to deal with situations when hotel rooms were fully booked. This provided a way for TravelEssence to keep in contact with potential customers even when the rooms for a given destination were full. In this way, they could follow up and notify customers when rooms became available.

To achieve a robust software system, they agreed to split the monolithic database. Over several years of operation, the CHMS database had grown large and difficult to maintain, and slower. It was time to streamline it. Jones suggested that it would be useful to separate the database into smaller parts, and archive historical or unused data outside of the database.

Seet and Jane were responsible for the reservations-related part of the CHMS. There were other parts of the CHMS that dealt with customer check-in and check-out, customer relationship, sales, etc., but we will not discuss them in detail because they are not within the scope of this book.

Seet and Jane's team updated their vision to include distinct goals:

1. Reduce the number of steps needed to complete a reservation, through the use of default data or past user data entries.

2. Provide waiting functionality as required by program level.
3. Move old and unused reservation records out of the existing database to improve database performance.

21.4.2 Managing the Work

With a good understanding of what needed to be done, the next step was to work toward their goals effectively and efficiently. The teams agreed on how they would achieve the vision by identifying pieces of work they needed to achieve (i.e., specifying Product Backlog Items and prioritizing them). They made sure that these Product Backlog Items were work that was open and transparent to everyone, so that teams could understand each other's priorities and workload. This also helped in managing dependencies across teams. For this reason, Jones made sure that his teams had a shared and transparent backlog, as depicted at the program level in Figure 21.6.

When scaling to large and complex endeavors, individual smaller teams within the overall program often have their own more detailed product backlogs. However,

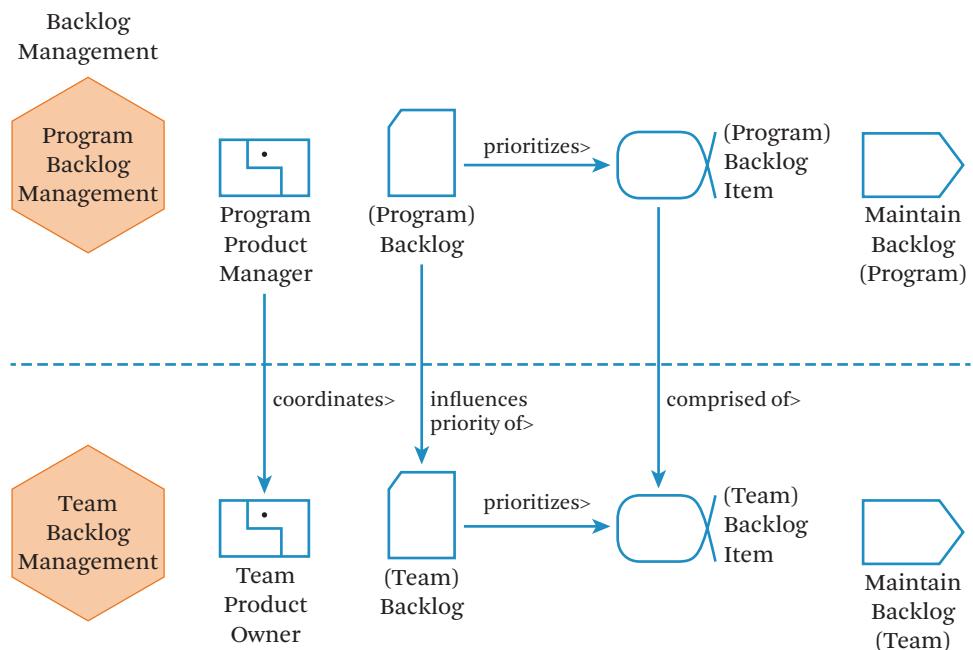


Figure 21.6 Program and Team Backlog management.

Table 21.5 Program and Team Backlog Items

Item	Program Backlog Item	Team Backlog Item (Seet's and Jane's team only)
1	Improve user experience by reducing number of steps	Reduce the number of steps during reservation by using cached data, and prediction based on user's location and other information.
2	Provide waiting list functionality	<ol style="list-style-type: none"> 1. Provide waiting list functionality when making reservation. 2. Notify customers when rooms are available. 3. Handle the scenario when room becomes available when customer checks out early. 4. Handle the case for VIP customers.
3	Improve database performance	<ol style="list-style-type: none"> 1. Move old reservation records out of existing database (i.e., archive). 2. Provide administrative functionality to archive records. 3. Update report generation functionality to use archived records.

these need to align properly and be coordinated with the overall program backlog. This is an example of why we need additional practices when software endeavors scale up. In this case, a practice to help align and coordinate multiple detailed team backlogs with the overall program backlog may be needed.

At TravelEssence, this program backlog management practice had a Maintain Backlog periodic activity that Jones conducted. It was attended by all of the team Product Owners with the purpose of ensuring that each individual team backlog had the proper priority, given the overall program backlog priority.

Based on their agreed-on Program Vision of providing great customer experience over a robust software system, Samantha, Seet, Jones, Jane, and the rest of the CHMS members collaborated to translate this vision into specific program backlog items (see Table 21.5). These were then translated to team backlog items that Jane's team could work on.

There was, for example, a program backlog item for reducing the number of steps to carry out a use case. Seet studied the reservation use case and found that it indeed had too many steps. So he found ways to simplify it, and put the improvements into Jane's team product backlog.

Through discussions, Seet and Jane agreed on all the team backlog items for Jane's team, as shown in Table 21.5.

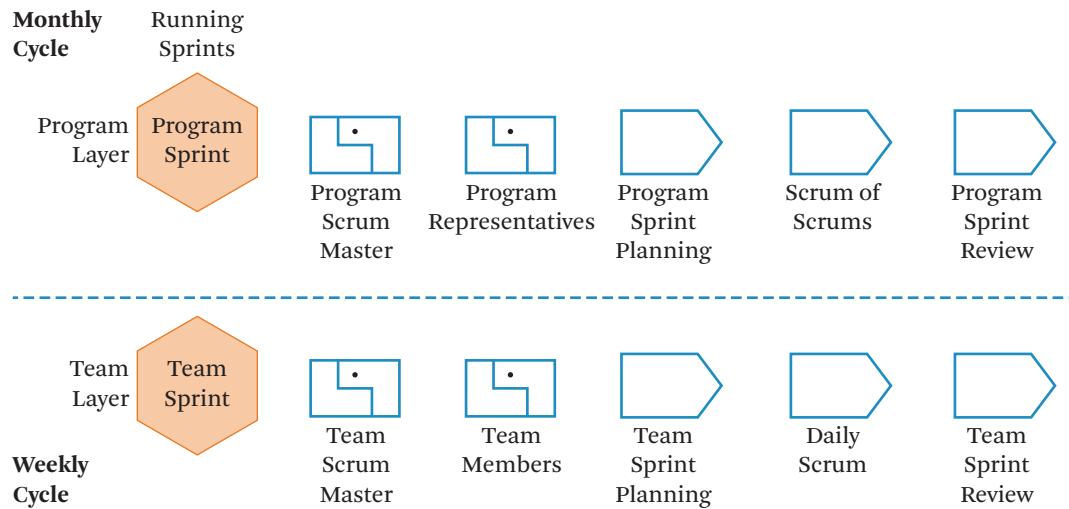


Figure 21.7 Running Program and Team Sprints.

21.4.3 Working Iteratively

Having a good understanding of what needed to be done, as prioritized in the overall program backlog, Jones and Jane had to break that work down and determine what they could complete in each sprint. To achieve this, they worked iteratively in a synchronized way, as shown in Figure 21.7. This is really the application of Scrum Lite (discussed in Part III) at both the team level and at the program level.

Jane's team, as well as other teams in Jones's program, agreed to run two-week sprints. At the program level, Jones decided to run monthly sprints. It is not uncommon in large-scale development that you might have multiple development teams running shorter sprints than an overall program-level team. Usually, the program sprints are multiples of team sprints. For example, if a team sprint is two weeks, the program sprint can be 4 weeks, 6 weeks, or 8 weeks. This allows the development team to reach internal checkpoints and verify their work prior to integrating that work with other teams' work at critical points agreed on, at the overall program level.

Often in these cases, on large complex efforts, individual smaller teams find that they cannot complete all the work that is part of the overall vision within the proposed time frame. This is one reason why, in complex efforts, additional practices are needed to communicate related issues and coordinate any changes needed across development teams working in parallel.

Table 21.6 Jones's program activity

Activity	Schedule	Duration
Program Sprint Planning	Last Friday afternoon of each month	3 h
Team Sprint Planning	Each Monday morning	2 h
Daily Scrum	Every morning at 9 am	15 min
Scrum of Scrums	Every Tuesday and Thursday after Daily Scrum at 9:15 am	15 min
Team Sprint Review	Every Friday morning	2 h
Program Sprint Review	Last Wednesday afternoon of each month	3 h

A common practice to help coordinate such issues across multiple Scrum teams is referred to as a Scrum of Scrums (see Figure 21.7). A Scrum of Scrums can be understood as a form of Daily Scrum (see Section 14.8.2) that is applied across teams rather than among members within a single team. At a typical Scrum of Scrums, representatives from the individual teams highlight issues that are hindering their progress and creatively look for solutions. Just like the Daily Scrum, the Scrum of Scrums meeting should be short to make efficient use of time. The idea is to get teams moving as smoothly as possible. If there are any severe issues, affected representatives can organize a separate problem-solving session.

Jones and his fellow colleagues agreed that they would run sprints according to the rhythm and duration summarized in Table 21.6. With this settled, they were able to mark their calendars, book meeting facilities, and make themselves available for the activities.

Some team members did not see the value of having these meeting rituals at first, but that changed soon. In the past the team had always resorted to ad hoc meetings, which often disrupted the team members' work. Having meetings at regular times allowed the team members to maintain their work momentum. Moreover, over a period of time, the meeting agenda items evolved and became more focused and productive.

We now dive deeper to see how Jones and Jane benefitted from this way of working.

Jane's team was responsible for providing the waiting list functionality. This functionality was related to room availability. Room availability in turn was affected by guest check-in/out, which was the responsibility of another team. Moreover,

since Jane was additionally responsible for room reservations, she was also responsible for moving old reservation records to an archive as part of improving database performance.

In the past, Jane had to persuade other teams, whose actions her team depended upon, to conform to her deadlines and priorities. But of course, other teams had their own deadlines and priorities, which were quite different from hers. This was resolved through the Program Sprint Planning activity, which got all team representatives together to align their deadlines and priorities, and reach a healthy compromise together with the program Product Manager and team Product Owners.

Both Jones and Jane, then, sorted out issues with dependencies across teams through the Scrum of Scrums meetings. In particular, through the Scrum of Scrums meetings, Jane could get other teams to collaborate with her team, which allowed her team to complete integration earlier.

Finally, the review meetings were a highlight for everyone. Samantha participated as the program Product Manager and saw the value of having other stakeholders involved. She gradually invited others from sales, marketing, and customer service to attend the review meetings. This allowed people like Jane to understand business needs. It also helped these business people understand what was required to communicate requirements clearly to development teams.

21.4.4 Continuous Improvement

Continuous improvement is essential in all development endeavors, regardless of scale. Jones knew that there would always be ideas his teams would come up with that could lead to better ways of working. This would be true regardless of the size and complexity of the development effort. Therefore, Jones made sure that retrospectives were held at both the team level and the program level, as depicted in Figure 21.8.

At the team level, the Scrum Master facilitates the Team Retrospective, in a similar manner to that discussed in Section 14.8.4. The main difference when you scale in size and complexity is that there will be issues and impediments that the individual teams cannot fix on their own. This, then, is an example of why we need additional practices when we scale up, in order to make it clear to individual developers how they can raise to the program level the issues they see for discussion and proper attention. At the program level, there needs to be a program Scrum Master, just as at the individual development team level there needs to be a team Scrum Master. The program Scrum Master is responsible to conduct a retrospective that is appropriate for the overall program. Experience has shown that program-level

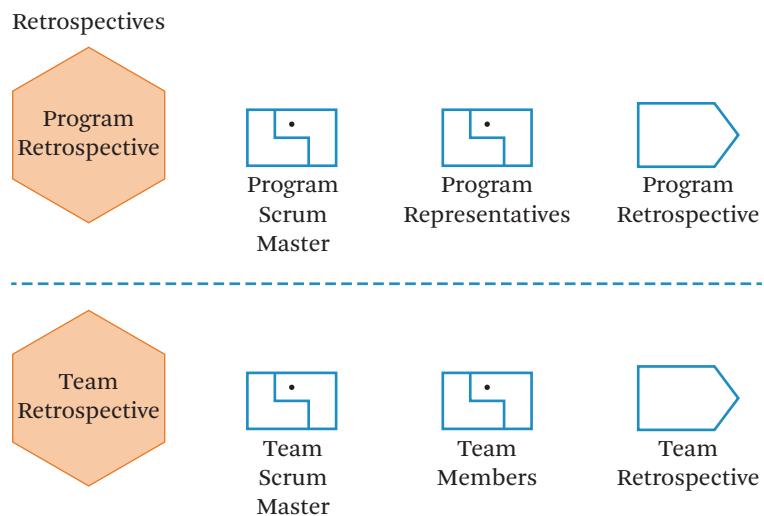


Figure 21.8 Conducting program and team retrospectives.

retrospectives need to have representatives from each development team to properly express each team's concerns.

In Jones's program, these retrospective meetings were held in conjunction with the review meetings both at the team level and at the program level, at weekly and monthly cycles, respectively. Smaller improvements that teams could quickly put into action got implemented quickly, while larger improvements that would impact several teams, or even those beyond the CHMS, were deliberated in the Program Retrospective. Retrospectives were an important part of Jones's program. It was the avenue through which team members saw that management took an active part in understanding the problems they faced and cared enough to fix the problems. These problems ranged from having the right computing resources, to having a good working environment, to policies. They could be as mundane as moving a printer to a more convenient location.

21.5

Value of Essence to Large-Scale Development

What we have just described is the way of working for Jones's program and Jane's team. If you compare what Jane's team did in this chapter and what Smith's team did in Parts II and III, you will notice that there were significant differences. Jane's team was using a decomposed Scrum in collaboration with the program practices. Moreover, she was working on part of the legacy CHMS, whereas Smith's team was

working on the new recommendation engine and using relatively new approach (at least at the time when this book was written) such as microservices. This means that the practices that are needed by one team are often different from another team, even on the same endeavor, when you are working in large-scale development. This is because the issues the teams have to face can be very different. For example, when you are dealing with a legacy system, often the issues are related to understanding the underlying design and legacy requirements. In contrast, when you are working on new functionality, often the issues are related to the real need of the customer that is driving the new functionality.

Even Jane's peers, and other teams in Jones's CHMS program, used somewhat different practices. They had practices for testing legacy software, code reviews, and so on, which we will not discuss in this book. Of course, Jones's program management team had other (program-layer) practices, which were different from the development teams' practices.

Regardless of whether they were working on the team level or the program level, or what specific practices they applied, their practices were all defined on top of the kernel (see Figure 21.9), and the progress and health of both programs and teams could be based on kernel alphas and the alphas introduced via their chosen practices.

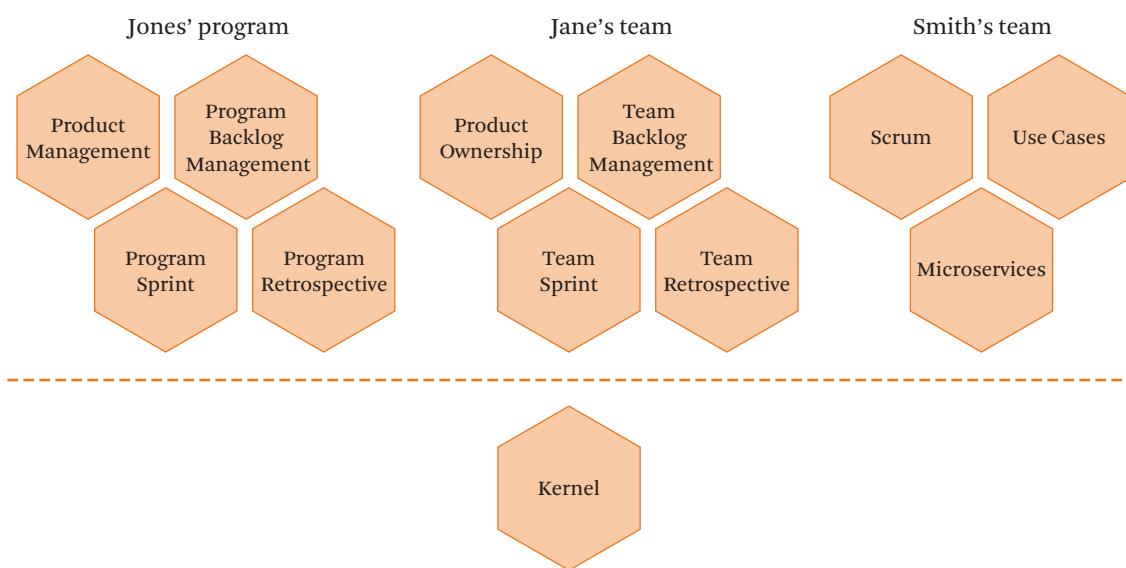


Figure 21.9 Different practices for different teams.

From what we have discussed thus far in this book, there are several important benefits of using Essence for both small teams and large teams (i.e., a program with subordinate teams):

1. Essence provides a lens to evaluate the way of working, what practices to use, how to highlight risks, and so on.
2. It provides a means to make practices explicit.
3. It provides a way to evaluate the progress and health of each endeavor.

We have seen these benefits in action throughout the book with Smith's story and now in this part with Jones's story.

What Should You Now Be Able to Accomplish?

After studying this chapter, you should be able to

- explain possible elements of endeavor complexity;
- explain the difference between the practices at the team layer and the program layer;
- give an example of a practice architecture involving the team-layer practices;
- describe the team perspective on Scrum as a composition of smaller team practices as applied to product ownership, team backlog, team sprint, and team retrospectives;
- list additional steps needed when kick-starting large-scale development (comparing to small-scale development);
- list roles that are employed in a large-scale development endeavor, together with their descriptions; and
- explain the Scrum of Scrums.



Reaching Out to Different Kinds of Development

The goal of this chapter is to emphasize that one can often encounter different kinds of development within each organization, as there is no one-size-fits-all method for developing software. In this chapter, the reader will be shown

- the role of Essence in the context of the organization to promote communication regarding practice selection and composition;
- the role of a practice library, where practices can be added and evolved to meet all required company needs;
- the concept and roles of coaches to mediate the discussion and agreements to improve and update the practices in the practice library; and
- the important task of keeping track of different versions of practices.

In a large organization it is common to see different types of development cases: new development, legacy migration, business process re-engineering, exploratory development, enhancements to the core, mobile development; the list goes on. Will a single method work for all these development cases? Definitely not! Will an agreed-on set of practices work for an organization forever? Definitely not! The industry evolves and new knowledge and technologies emerge daily. We do not live in a static world, but a very dynamic one.

Our goal has been to make the life of practitioners easier so that discussions about methods and practices become second nature, something that they do not need to worry about and spend too much time considering. We want them to focus on doing what they are best at doing, to produce high-quality software.

22.1

From a Practice Architecture to a Method Architecture

Let's do a recap. So far in this book, we have introduced two different development scenarios: a small-scale development led by Smith and a large-scale development led by Jones. Each applies a different method. Instead of having each program and team identify practices they need, it is oftentimes useful to provide pre-composed methods. This implies the need for a method architecture.

Figure 22.1 shows TravelEssence's method architecture. It is very similar to the practice architecture shown back in Figure 20.1, with the addition of a pre-composed method layer at the top. Each item in the method layer is a composition of a set of practices to fulfill a specific purpose. For brevity, we only show two: one

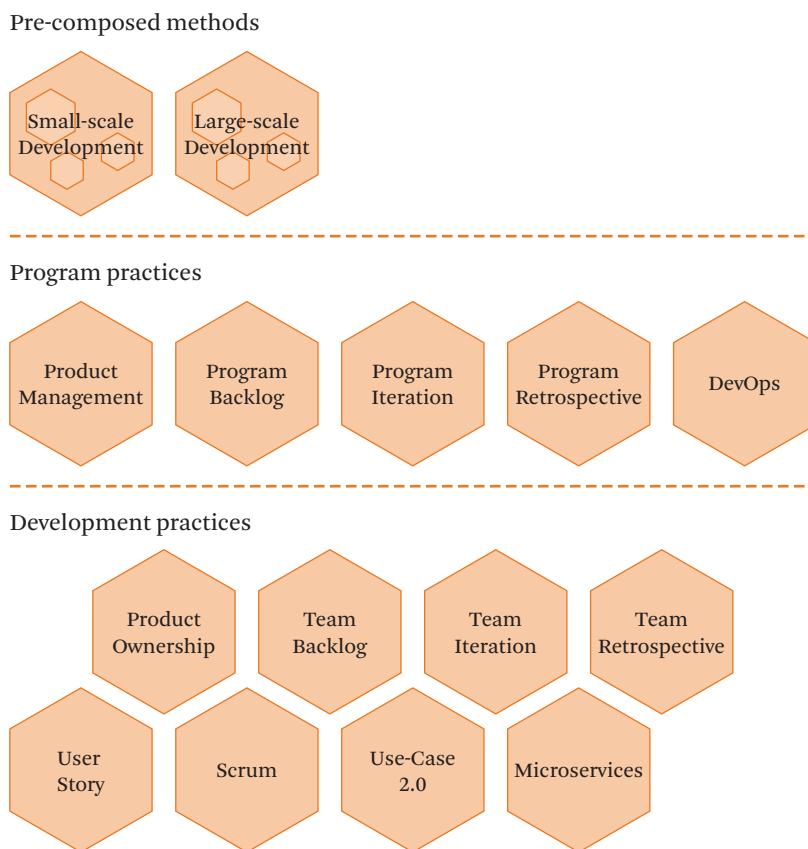


Figure 22.1 Method architecture with practice architecture and pre-composed methods.

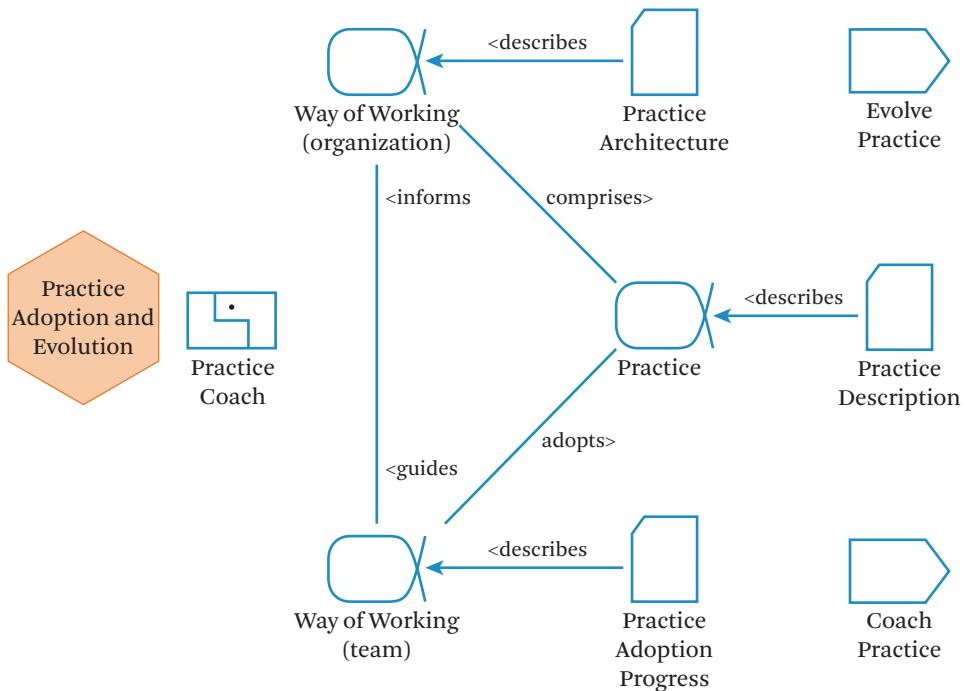


Figure 22.2 Practice to adopt and evolve practices.

for small-scale development (exemplified in Parts II and III), and one for large-scale development (exemplified in Chapter 21).

Having the method and practice architecture made explicit is not all there is to software engineering, though it is an important step. It is also important, in fact a lot more important, that members of an organization actually know how to find and apply the practices in a deliberate and disciplined manner.

At TravelEssence, Jones recognized that in addition to capturing practices, the practices must be made available to team members. Practice guidelines should not be on the shelves collecting dust or hidden in some folder that nobody accesses. Consequently, with assistance from Smith, Jones wrote a practice to help team members adopt and evolve practices.

Using the language and notation of Essence, Figure 22.2 shows that an organization's Way of Working alpha can be described by a work product called the Practice Architecture (such as the one for TravelEssence, shown in Figure 22.1). Each team's Way of Working, in turn, comprises a set of practices (each of which will be an alpha)

that the team has chosen to adopt. The Practice Description work product (such as those we have provided throughout Parts III and IV) defines each such practice. As an organization's practices evolve, they in turn influence the future selection and application of practices, resulting in activities such as Evolve Practice. An experienced person in the role of a Practice Coach can guide teams (using activities such as Coach Practice) to select practices in keeping with both the team's and the organization's needs. Essence provides a language and notation to describe these practices in a composable and actionable way.

The adoption of each practice takes time, as team members learn. At Travel-Essence, Jones used a Practice Adoption Progress work product to track how well a team adopted a practice, and to record the lessons learned through it. Smith played the role of a Practice Coach to facilitate this process. He helped teams select practices to adopt and advised them to instill a continuous improvement cycle to continuously evolve their way of working, using activities such as Coach Practice and Evolve Practice.

22.2

Establishing a Practice Library within an Organization

In order to apply this, an organization will need to set up a practice library whereby practices can be added, and adapted and evolved to address many different kinds of endeavors. To continue learning, such an organization should be constantly renewing its way of working and its practice library. Figure 22.3 provides a simplified view of this.

The practice library is made available to programs and teams. The coaching network is a pool of coaches who help programs and teams select and apply practices effectively. Programs and teams will, while using the practices, likely improvise

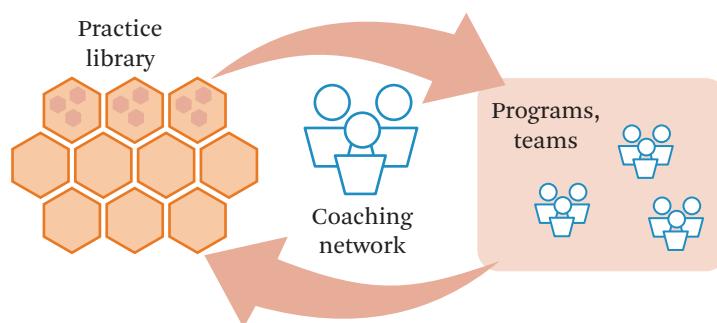


Figure 22.3 An organization with a practice library to draw upon.

and adapt the practices to meet challenges in their specific contexts. Coaches will then bring these experiences to discussions that in turn can lead to agreements to improve and update the practices in the practice library. This completes the continuous cycle in such a learning organization.

To initialize this cycle and allow teams to easily select the practices they need, organizations need to first essentialize practice candidates to set up the practice library. Once several are available, teams can assemble the practices found in the library into usable methods.

Anyone can essentialize a practice, but the best result is obviously achieved if an experienced developer does it; in particular, someone experienced in the specific practice concerned. Once an organization has essentialized their practices, they will need to be maintained and improved. Improving essentialized practices can be done by anyone in the organization. Since a changed practice may only be used in some teams and the original practice may still be used by other teams, an organization needs to keep track of different versions of practices and which version has been used in which endeavor.

Whether first essentializing a practice or improving it, one must realize that the objective of essentialization is to help an organization learn. Second, there is a cost associated with essentialization decisions such as deciding on alphas vs. work products, and deciding on exactly what should be an explicit practice versus a pattern or a resource, or just a principle, such as simple design. In some organizations, games or “mini-practices” are treated as resources that provide references to previously published books on these subjects. Other organizations may decide there is value in describing these smaller practices more explicitly to aid their less experienced practitioners. Third, the level of detail for each practice has to be worked out. In particular, the following steps help to establish these details:

1. Identify alphas, states, work products, checklists, and activities in practices.
2. Provide supporting material to supplement the practices, such as references to published books about them.

The greater the level of detail, the more effort is required. An organization that is growing with many new employees would probably need well-written practices. This organization could essentialize their own practices, or they could draw from the growing community of practice authors around the world. This book itself represents the authors' contribution of practices to the community (e.g., Use Case Lite and Microservices Lite in Chapters 16 and 17). We encourage experts in the industry to essentialize more practices. As students, when you learn something

new in software engineering and technologies, we encourage you to essentialize it as a way to summarize and capture your understanding, for your own benefit and for that of others.

22.3

Do Not Ignore Culture When Reaching Out

So far in this book we have not discussed cultural and social aspects of software engineering. Yet culture is something not to be ignored. The adage “Culture eats strategy for breakfast” (meaning that culture is critically important when it comes to people’s behavior) has been attributed to management guru Peter Drucker. Indeed, an organization’s culture has great impact on the success of introducing and adopting new software engineering practices and methods. An organization’s culture affects individuals’ and teams’ capacity and motivation to learn and grow.

Briefly, culture is the unspoken but acceptable norms and behaviors in an organization. It influences the way people act in an organization. It also influences what people readily accept or reject. Some people at TravelEssence for instance, welcomed new ideas, but in general, most were not so ready. They liked the way things were, as it was comfortable.

Smith did not know the impact culture had before accepting the role to reach out to different parts of the organization that had different kinds of development (new development, legacy development, experimental development, etc.).

Smith already understood that establishing a practice architecture would uplift competency across his company. He was convinced and passionate, but persuading his peers seemed to take forever. The successes that occurred in Jones’s program as a result of adopting Essence were clearly visible, but people in other programs that had not seen these successes were not so enthusiastic. Not wanting to try this new approach, they gave all kinds of reasons, which Smith saw as excuses: they were busy, they had more important things to do, and so on. Smith was gradually losing his cool until he had a serious chat with Jones. Jones’s program had been applying the practice shown in Figure 22.2, and had learned a trick or two.

Jones understood that the adoption of practices could not be achieved via a top-down, command-and-control approach, but instead could be socialized across the organization. Jones had earlier created opportunities and events for people playing different roles or from different departments (e.g., from the business side and the development side, or from the development side and the operations side) to meet and discuss how to improve their way of working. After holding such events several times, people participating in his program began opening up. They

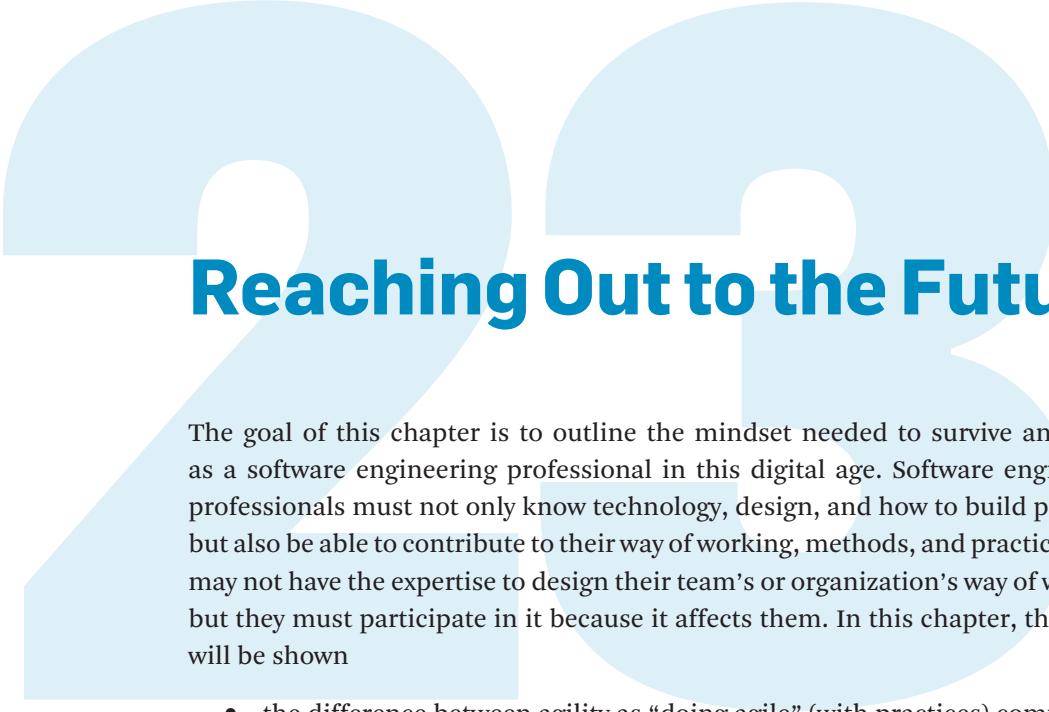
gradually stopped insisting on traditional practices and instead opted for more appropriate ones. These events that Jones held had the blessing of his department head, and were always in an enjoyable and facilitated setting. Drinks and food were provided, and there was a facilitator to prompt discussions and drive actions. Jones suggested that Smith could do something similar.

After speaking with Cheryl, the CIO, Smith received the go-ahead to conduct similar events. Slowly, gradually, Smith started to observe some movement. Programs and teams across the company started to become more open and requested Smith and his coaching network to help them out. And the rest was history.

What Should You Now Be Able to Accomplish?

After studying this chapter, you should be able to

- explain the benefits and challenges of combining multiple methods or practices within a single organization;
- explain the role of a practice library;
- explain the role of coaches when mediating the discussion on updating practices in the practice library; and
- explain the role of culture when reaching out to different parts of the organization.



Reaching Out to the Future

The goal of this chapter is to outline the mindset needed to survive and thrive as a software engineering professional in this digital age. Software engineering professionals must not only know technology, design, and how to build products, but also be able to contribute to their way of working, methods, and practices. They may not have the expertise to design their team's or organization's way of working, but they must participate in it because it affects them. In this chapter, the reader will be shown

- the difference between agility as “doing agile” (with practices) compared to “being agile” (with mindset) and explain why the latter is more important than the former;
- important mindset changes to be aware of, including team ownership (instead of selected-few ownership), a focus on method use rather than description, and embracing the evolution of the methods (instead of keeping them fixed);
- important shortcomings in method descriptions, e.g., failing to communicate what the team really needs—which might be a way to determine real development progress, a means to plan, a system to organize their team members, etc.; and
- the important role of continuous improvement and intentional creation of an environment, in which continuous improvement of the employed method can naturally happen.

The intent is to encourage readers to research and evolve a method that helps teams and organizations deliver software much better.

This is an exciting day and age in which to be software professionals. We are at the forefront of every transformation in every business and every nation. Businesses are becoming digital. Nations are becoming digital. There are rapid technological changes every day, with lowering costs of computing and bandwidth. We see

augmented and virtual reality applications becoming commonplace. Instead of traveling to a new place, you can do it through your headset. Instead of merely looking at a building design on paper, you can walk through and experience it using virtual or augmented reality before confirming your wishes to the architect. Electric cars are common now, and we are at the advent of large-scale proliferation of autonomous vehicles and a revolution in transportation. Artificial intelligence applications are also becoming popular, and they can do things that before we could only have imagined in science fiction. We could go on forever talking about these exciting advances, but we want to pause and ponder their implications for software engineering.

In the midst of all this, software is at the forefront and at the core of this change. It is not presumptuous to say that the world runs on software. With all these changes, some very natural questions software professionals can ask are: What will software engineering be like in the future? Will it be something revolutionary and brand new, requiring complete overhaul? Or will it be an evolution of what we already have established?

These were the same questions that plagued Smith, who had recently been promoted to become the head of software engineering for TravelEssence. He needed to build the competency of his teams and establish an effective way of working to face rapid advancement and strong competition. His teams were adopting new tools like advanced analytics, blockchain, AI, and many more, and they needed to collaborate effectively. These advances entailed different kinds of development, and Smith discovered that with all of them there would be something “old” and something “new.”

There would be something “old” in the sense that there are timeless principles and practices that were useful even with the newest advances. In particular, the kernel was always helpful in evaluating the progress and health of teams’ endeavors. Practices like Scrum Lite, Use-Case Lite, and Microservices Lite, presented in Chapters 15–17, and the program and team practices discussed earlier in this part of the book, were still applicable even for these new technologies.

Nevertheless, there would be something new, as expected. New applications, such as advanced analytics, required new competencies and new practices. For example, they required an Analytics Translator competency: someone able to take vague goals like “I want to increase sales by 50%” and translate them into actionable data requirements that could then be implemented by development teams. Even the recommendation engine that Smith and his team had implemented (see Part III) was already, in fact, an instance of an AI application, as that system could intelligently recommend and guide users in their bookings and travels.

Because Smith and TravelEssence had their methods and practices essentialized, they were able to add new practices easily. Team members knew how the new practices fit. This wouldn't have been the case if TravelEssence had been using monolithic methods. Their early investment into essentialization had paid off.

Breaking from our fictional TravelEssence story, in the larger scheme of things, the authors of this book and the SEMAT community have been reaching out to different kinds of development across the industry. An early result is the application of Essence to enterprise IoT. In simplistic terms, enterprise IoT systems comprise many smart devices (e.g., smart watches, wearables, light bulbs, sensors, etc.) connected to and sending information to powerful enterprise backends to perform sophisticated computations. Enterprise IoT powers smart buildings and even smart cities. What are software engineering practices like for these applications? Interested readers may refer to [Jacobson et al. 2017] for details. At the same time, the SEMAT community is also building a library of practices with participation from industry leaders. Methods such as Scrum and Disciplined Agile have already been essentialized and made available. All these efforts are designed to help software teams and organizations deliver better software better, faster, cheaper, and happier.

23.1

Be Agile with Practices and Methods

In this digital age, being agile with practices and methods is crucial. Even back in the late 1990s, software engineering experts had recognized this fact: in our fast-changing Darwinian world, survival is about the ability to adapt quickly. In 2001, a group of software engineering pragmatists came together to discuss how to overcome challenges faced by our industry. At that time, most development endeavors were using a so-called “waterfall approach,” which took years to complete, and they were not really delivering what customers or users needed. These experts had been successful with their (at that time) new approaches and were constantly improvising and innovating. They summarized what they believed in as a manifesto:

“We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.¹

Perhaps the most agile statement in the manifesto is that “We are uncovering better ways of developing software by doing it and helping others do it.” It is this continuous striving to be better that spurs learning and advances in software engineering, not just in the industry, but in every team. Essence has taken the additional step to make the capturing, and communication—and hence the learning and application—of appropriate practices better. But this is not enough.

We recognize that every team has its own unique development context; they are engaged in different kinds of development, their members have different levels of experience and backgrounds. They can be working on small- or large-scale development. But in all these situations, a team can always collaborate and grow together to become better. It is this continuous adaptation to the development context and continuous learning that being agile with practices and methods is fundamentally about.

The authors of the Essence book [Jacobson et al. 2013a] take the values and principles of the agile movement to another level by providing tools for teams to explicitly evolve their way of working.

In contrast to many previous method initiatives, the focus of SEMAT is on those who know best what works and what doesn’t work when building software systems: that is, architects, analysts, designers, programmers, testers, and software managers. This takes a change in mindset—changes in the ways you think about methods—just as moving to agile development from traditional development (e.g., the waterfall lifecycle) requires a change in mindset on how to develop software.

What are the specific mindset changes you need to be aware of? To some extent they differ from the Agile Manifesto, referenced above, but they are inspired by it:

1. The full team owns their method, rather than a select few.
2. The focus is on method use rather than on comprehensive method description.
3. The team’s method evolves, rather than staying fixed.

These mindsets are crucial in your journey as a software professional. Not only must you understand technology, but you must also collaborate effectively with your teammates and lead your teams effectively. To help you do this well, we delve deeper into each of these mindset changes in the following sections.

1. Agile Manifesto, <http://agilemanifesto.org/principles.html>

23.2

The Full Team Owns Their Method

Traditionally, there is a dichotomy between method engineers and method users (i.e., developers):

1. Method engineers who define methods often do not use the methods themselves.
2. Method users who acquire real experiences with methods often are not asked, or do not have time, to give feedback and refine the methods they use.

As a result, method users often find method discussions to be out of touch with their real experiences and therefore a waste of time.

The kernel approach seeks to bridge this divide by placing a proper balance on all stakeholder perspectives. It recognizes the value of every team member in determining what ways of working are best for their team.

The Essence kernel can support this proper balancing through the use of the alphas, states, checklists, and by describing a team's own method as a composition of practices. Examples include:

1. conducting retrospectives guided by the alphas and their states;
2. defining practices on top of alphas and their states;
3. using alphas and states to agree on team member involvement and team responsibilities; and
4. using alphas and states to define lifecycles.

The alphas and their states and checklists provide a very simple but powerful tool, which team members can employ to take ownership of their own method.

23.3

Focus on Method Use

Traditionally, when most people talk about methods they are thinking in terms of method descriptions. Having a method description is a good thing in that it allows new team members or even existing team members to familiarize themselves with the team's method. Too often, however, these method descriptions fall short in communicating what team members really do in their day-to-day work. Unfortunately, the method descriptions have often become too heavyweight. This has only served to make the process description less useful, rather than more.

This dilemma is not best solved by more words, but rather by fewer words and more use. How do teams and team members actually use methods, then, to help

them in their day-to-day jobs? Consider the following needs with respect to a team's methods.

1. Teams need a way to determine real development progress.
2. Teams need to plan their endeavors and their sprints, and they need to discuss and agree upon what it means to be done.
3. Teams need to organize their team members, and agree on team member involvement and responsibilities.
4. Teams need to do their work and adapt their way of working.
5. Teams need to scale to varying size endeavors to handle varying challenges and complexity.

23.4

Evolve Your Team's Method

There is no one-size-fits-all when it comes to methods. This implies the following:

1. You cannot simply take any method and follow it blindly. All methods must be adapted to fit your situation.
2. Once adapted to your current situation, you are certain to learn more as your endeavor proceeds, requiring more adaptations.

A team's method is never fixed. Teams must constantly evolve their method as long as there is work to do on the product. This implies two fundamentals:

1. Always be ready to embrace new and better ways of working.
2. Always consider your current development situation when considering a change.

Evolving a method is straightforward with the kernel approach. You start with the kernel, and evaluate the practices you already have. Practices that are inadequate are then refined or replaced with better ones. This is best done gradually so continuous improvement becomes natural and not something you need to think about at great length.

Making continuous improvement a natural habit is easier said than done. It takes effort and proactive leadership and culture. So, on top of having a practice architecture, great organizations invest in creating a conducive environment to promote continuous improvement. Our approach of decomposing complex methods into much smaller practices helps teams take ownership of their methods and the outcomes of using them.

You have chosen a profession that is at the forefront of technological advances. Not many other professions are changing at a similar pace. To face such challenges, you need to be well prepared; what you have learned in this book, especially about Essence, will help you go a long way. As a final and parting note, we urge you to always be learning and growing, uncovering better ways of working all the time.

What Should You Now Be Able to Accomplish?

After studying this chapter, you should be able to

- explain the intent of the Agile Manifesto and its key principles;
- describe mindset changes that need to happen when becoming agile;
- explain what a team really needs from a method, by giving examples; and
- explain what it means to say there is no one-size-fits-all when it comes to methods, and what this implies.

Recommended Additional Reading

- R. Knaster and D. Leffingwell, *SAFe 4.0 Distilled: Applying the Scaled Agile Framework for Lean Software and Systems Engineering* [[Knaster and Leffingwell 2017](#)]. This is a great resource for understanding the Scaled Agile Framework (SAFe). It covers SAFe 4.0.
- S. W. Ambler and M. Lines, *Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise* [[Ambler and Lines 2012](#)]. This is an excellent resource for understanding Disciplined Agile Delivery (DAD).
- C. Larman and B. Vodde, *Large-Scale Scrum: More with LeSS* [[Larman and Vodde 2016](#)]. A great overview of LeSS, a method to apply Scrum in large-scale development.
- I. Jacobson, I. Spence, and P.-W. Ng. Is there a single method for the Internet of Things? [[Jacobson et al. 2017](#)].

APPENDIX

A Brief History of Software and Software Engineering

You are probably curious about why and how the profession of software engineering arose. Thus, it is useful to briefly examine the evolution of software provisioning and especially the awareness of the need for the structured approach provided by software engineering.

While the forerunner of computing is often cited as the work of Charles Babbage and Ada Lovelace in the 19th century, as well as the development of Hollerith punched card equipment in the early part of the 20th century, it was the developments starting in the 1940s that provided the starting point for modern computing. These early computers (some electrical-mechanical) were programmed in a very primitive manner by switches, by punched paper tape, or by some form of plug board wiring that directed the computers' operation. The application of these early machines focused upon numerical computation of mathematical functions, and the programs were quite small.

A major breakthrough occurred when the notion of stored program computers was proposed by John von Neumann in the latter part of the 1940s and was implemented, among other applications, for the first commercial computer, the Univac I. At this stage, while numerical computation was still important, the focus began to shift to data processing (being able to store and process large quantities of data stored on magnetic tapes). In fact, the first Univac I delivery in 1951 was to the United States Bureau of the Census, where processing of population and production data as well as provisioning of statistics was the focus.

During the 1950s, a variety of stored program computers were produced and marketed by several companies in the United States, Europe, the Soviet Union, and Japan. Programming of these early computers was still a challenge since machines were programmed at the machine instruction level. A major breakthrough occurred in 1951 when Grace Murray Hopper introduced a means of compiling program code

for numerical computations and called the program that did this a “compiler.” This provided for easy re-use of program code in an effective manner, in particular the code for mathematical functions.

During the 1950s, a number of programming languages and compilers evolved for numerical computation—for example, Fortran—as well as the increasingly important area of data processing resulting in COBOL (which even today remains as the most pervasive programming language for data processing). Further, in the early 1960s, the utilization of computers for physical process control started to evolve, and eventually languages that facilitated a higher-level means of programming these devices were provided.

As these higher-level languages furnished a means to construct significantly larger applications and the use of computers became more pervasive, it rapidly became clear that developing and sustaining large suites of programs presented an enormous intellectual and management challenge as described by Fred Brooks [Brooks 1975]. During the 1960s it was recognized that there was a “software crisis” as indicated in the following quotation.

“The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.”

[Dijkstra 1972]

Thus, as computing technology advanced and as larger and larger software suites evolved during the 1950s and 1960s, it became evident that a more structured approach to software was required. While the term “software engineering” had been used by some authors in the mid-1960s, it was at a NATO-sponsored meeting in Garmisch, Germany, in 1968 that the need for a software engineering profession was clearly addressed [NATO 1968]. The conference was attended by international software experts who agreed upon the need to define best practices for developing and sustaining software systems that are grounded in the application of an engineering approach.

There are a variety of brief definitions of software engineering; for example, the Association for Computing Machinery (ACM) and the Institute of Electrical and Electronic Engineering (IEEE), respectively, each still define the profession directly in line with the 1968 Garmisch intentions. According to them, software engineering entails: “systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation

of software to optimize its production, support, and quality [[ISO/IEC 2382 2015.](#)]” and “application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software [[ISO/IEC/IEEE 24765 2017.](#)]”

As the need for software engineering evolved, advances in hardware technology in the 1960s and 1970s provided both more powerful large computers and a range of smaller so-called mini-computers. A major hardware advance occurred in the mid-1970s with the development of large-scale integrated circuits. Certainly, the availability of inexpensive microprocessors, large primary and secondary memories, graphic processors, and communication via networks, including the Internet, have been game changers leading eventually to desktop computers and today’s laptops, mobile phones, and tablets as well as the pervasive use of embedded systems in a variety of products.

So, in addition to the early application areas of numerical computation, data processing, and process control, the software business has expanded to provide a wide variety of new application products such as e-mail, chatting, games, voice recognition, advanced graphics, smart phones, robotics, mobile communication, intelligent embedded devices, and so on. Furthermore, more recent developments have led to new infrastructure facilities in the form of cloud computing, the Internet of things (IoT), and cyber-physical systems. All of these developments have had a radical effect upon our society. While providing many new possibilities, the “software crisis” that Dijkstra pointed to has definitely intensified and we live in an era that raises important socio-technical issues as well as fundamental concerns related to safety, security, and integrity. Thus, there is a definite need to improve our capability to develop and sustain high-quality software.

Based upon this brief history of software evolution and the advances in hardware leading to new software challenges, you now have an idea of why the software engineering profession evolved, but there is much yet to do. It has become clear that programming and coding are only one aspect of software engineering. As you will have discovered in reading this book, there are many problems (technical, organizational, managerial, and social) that need to be dealt with in the provisioning and sustainment of high-quality software. The presentation of software engineering through the lens of Essence in this book provides a generic yet substantive definition of the profession far beyond the typical single-sentence definitions.

In the challenging environment that has evolved, there have been many discussions of the best practice approach to deal with the multiplicity of factors involved in providing sustainable and high-quality software systems. Is it a prescriptive engineering approach or the more agile practice approach that is to be preferred? Or is

it some combination of both approaches? In continuing this brief software history story, then, focus is placed upon how various approaches to software engineering evolved.

Some of the most important software engineering historical developments are as follows.

- During the late 1960s and 1970s, most popular approaches were based on the structure of programs emphasizing a function-data paradigm, so that basically a software system had two parts—the program part and the data part where the program part was organized to process a flow of data. One of the most popular was Jackson Structured Programming (JSP) introduced by Michael Jackson [Jackson 1975]. In another important development, Douglas Ross developed SADT (Structured Analysis and Design Technique), which also emphasized the flow of data (for program structures) as well as the flow information and processing activities [Ross 1977].
- During this period, an approach to organizing work now called the “waterfall method” evolved, in which activities were related to a sequential design process where progress is seen as flowing steadily downward (like a waterfall) through phases such as conception, initiation, analysis, design, construction, testing, production/implementation, and maintenance.
- During the same time period in the telecommunication business, another program structure approach based on the component paradigm was applied—a system here is a set of components interacting by sending messages to one another. Ivar Jacobson (an author of this book) was the original developer of this new approach. His colleague Göran Hemdal refined it and implemented it with programming language support and firmware to provide components “all the way down” to executable code. In 1976, Ericsson AB used this approach and produced the AXE telecommunication switching system that became the world’s leading system and resulted in the largest commercial success story ever in Sweden. At the same time, the visual language of the approach inspired the development of an international standard in telecommunications called Specification and Description Language (SDL). Sequence diagrams showing interactions among components, one of Ivar’s ideas, were adopted in SDL.
- During the 1980s, object-oriented programming (with languages such as Simula—which had appeared already in the late 1960s—plus Smalltalk, Eiffel, Objective-C, C++, Ada, Java and so on) became mainstream and a

large number of approaches to building object-oriented systems became popular. Object-orientation took the idea of components to a level of much finer granularity (everything was an object), so that the system was viewed as a huge set of objects where program behavior was realized by the objects interacting with one another.

- Eventually (in the late 1980s and 1990s) the component idea and the object idea were merged and components became exchangeable packages; the objects were then executable elements creating the functional behavior.
- During this period, an important abstraction called Use Cases, used to model the requirements of a system, was introduced by Ivar Jacobson [[Jacobson et al. 1992](#)].
- A more comprehensive set of abstractions were provided in the Unified Modeling Language (UML) standard for design of software intensive systems, which was adopted by Object Management Group in 1997. The three original developers of UML were Grady Booch, Ivar Jacobson, and James Rumbaugh [[Booch et al. 2005](#)].
- An international standard for Software Life Cycle Processes—namely, ISO/IEC 12207—was developed in the early 1990s [[ISO/IEC/IEEE 12207](#)] and was followed by the development of the ISO/IEC 15288 standard for System Life Cycle Processes. This was in recognition that a software system always exists in a wider system context. One of the authors of this book, Harold “Bud” Lawson, was the architect of this system standard. More recently, Ivar Jacobson and Bud Lawson contributed to and edited a book that provides various perspectives on software engineering in the systems context [[Jacobson and Lawson 2015](#)]. The 12207 and 15288 standards are being harmonized in recognition of the strong relationship between systems engineering and software engineering [[ISO/IEC/IEEE 15288](#)].
- The most popular method for software engineering became the Unified Process (UP), which was provided as Rational UP (RUP) in 1996. Ivar Jacobson was a father of RUP, but many other people contributed to its development—in particular, Philippe Kruchten and Walker Royce [[Kruchten 2003](#)]. RUP was based upon some proven best practices still in use today, such as use cases, components, and architecture. However, RUP was too large, too prescriptive, and very difficult to adopt widely.
- Around 2000, as a counter-reaction to RUP and other document-heavy approaches, a new movement arose that promoted light and flexible methods

combined with modern ideas for teamwork and fast delivery with feedback. Agility became the word of the day, and agile methods became the way forward. The most influential contributors to this movement were the following:

- Extreme Programming with User Story and Test-Driven Development (TDD), introduced by Kent Beck [[Beck 1999, 2003](#)]; and
 - Scrum by Ken Schwaber and Jeff Sutherland [[Schwaber and Sutherland 2016](#)].
- Now in more recent years, new methods for scaling agile have arisen, such as:
 - Scaled Agile Framework (SAFe), introduced by Dean Leffingwell [[Leffingwell 2007](#)];
 - Disciplined Agile Delivery (DAD), introduced by Scott Ambler and Mark Lines [[Ambler and Lines 2012](#)];
 - Large Scale Scrum (LeSS), introduced by Craig Larman and Bas Vodde [[Larman and Vodde 2008](#)];
 - Scaled Professional Scrum (SPS), introduced by Ken Schwaber (see <https://www.scrum.org/index.php/resources/scaling-scrum>).

While various approaches have imparted structure and discipline in providing software products, the number of these methods and practices has exploded. And, while there are successes, there are far too many failed and very expensive software endeavors. Many methods have become “religions” among enthusiastic creators and their followers. In fact, the popularity of the methods seems to be more like in the fashion industry, where differences are not well understood and artificially magnified. Further, there is a lack of objective evaluation, comparison, and validation of the methods and their composed practices.

So, given the multiplicity of methods that have arisen, a vital question is WHAT method should YOU (and the team of which you are a member) learn to utilize? Further, how well do the method and its practices that you select provide the multiple capabilities required to perform effective team-based software engineering?

In the series of best practice and method churms, a new community woke up. This community, called Software Engineering Method And Theory (SEMAT) and founded by Ivar Jacobson, Bertrand Meyer, and Richard Soley, wanted to stop the madness of throwing out all of what organizations and their teams had established and constantly starting over with new methods. As a result of the diligent efforts of a group of software engineering professionals, the Essence of Software Engineering evolved as a method- and practice-independent approach that has become an Object Management Group standard [[OMG Essence Specification 2014](#)]. This new

approach will certainly provide a basis for re-founding the profession of software engineering [Jacobson and Seidewitz 2014].

So now you should have an appreciation of how software and in particular software engineering have evolved. Essentializing software engineering as presented in this book has provided, for the first time, a means of unifying the multiple perspectives of software engineering.

References

- Alpha State Card Games. 2018. <https://www.ivarjacobson.com/publications/brochure/alpha-state-card-games>. 99, 153
- S. Ambler and M. Lines. 2012. *Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise*. IBM Press. 297, 339, 346
- K. Beck. 1999. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman. 203, 285, 346
- K. Beck. 2003. *Test-Driven Development by Example*. Addison Wesley. 346
- K. Bittner and I. Spence. 2003. *Use Case Modeling*. Addison-Wesley Professional, 2003. 222, 285
- G. Booch, J. Rumbaugh, and I. Jacobson. 2005. *The Unified Modeling Language User Guide*. 2nd edition. Addison-Wesley. 222, 285, 345
- F. Brooks. 1975. *The Mythical Man-Month*. Addison Wesley. 342
- M. Cohn. 2004. *User Stories Applied: For Agile Software Development*. Addison-Wesley Professional. 204, 247, 285
- E. Derby and D. Larsen. 2006. *Agile Retrospectives: Making Good Teams Great*. Pragmatic Bookshelf, Dallas, TX, and Raleigh, NC. 196, 198, 284
- E. W. Dijkstra. 1972. "The Humble Programmer." Turing Award Lecture, *CACM* 15 (10): 859–866. DOI: [10.1145/355604.361591](https://doi.org/10.1145/355604.361591). 342
- D. Graziotin and P. Abrahamsson. 2013. A web-based modeling tool for the SEMAT Essence theory of software engineering. *Journal of Open Research Software*, 1,1(e4); DOI: [10.5334/jors.ad](https://doi.org/10.5334/jors.ad). 147, 153
- ISO/IEC/IEEE 2382. 2015. Information technology—Vocabulary. International Organization/International Electrotechnical Commission, Geneva, Switzerland. <https://www.iso.org/obp/ui/#iso:std:iso-iec:2382:ed-1:v1:en>. 343
- ISO/IEC/IEEE 12207. 2017. https://en.wikipedia.org/wiki/ISO/IEC_12207 345
- ISO/IEC/IEEE 15288. 2002, 2008, 2015. Systems and software engineering—System life cycle processes. International Standardization Organization/International Electrotechnical Commission, 1 Rue de Varembe, CH-1211 Geneve 20, Switzerland. 345

- ISO/IEC/IEEE 24765. 2017. Systems and software engineering—Vocabulary. International Organization/International Electrotechnical Commission, Geneva, Switzerland. <https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:24765:ed-2:v1:en>. 343
- M. Jackson. 1975. *Principles of Program Design*. Academic Press. 344
- I. Jacobson. 1987. Object-oriented software development in an industrial environment. *Conference Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 87)*. DOI: [10.1145/38807.38824](https://doi.org/10.1145/38807.38824). 221, 285
- I. Jacobson and H. Lawson, editors. 2015. *Software Engineering in the Systems Context*, Systems Series, Volume 7. College Publications, London. 345
- I. Jacobson and E. Seidewitz. 2014. A new software engineering. *Communications of the ACM*, 12(10). DOI: [10.1145/2685690.2693160](https://doi.org/10.1145/2685690.2693160). 347
- I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. 1992. *Object-Oriented Software Engineering: A Use Case Driven Approach*. ACM Press Addison-Wesley. 345
- I. Jacobson, I. Spence, and K. Bittner. 2011. Use-Case 2.0: The Guide to Succeeding with Use Cases. <https://www.ivarjacobson.com/publications/whitepapers/use-case-ebook>. 169, 222, 226, 233, 285
- I. Jacobson, P.-W. Ng, P. E. McMahon, I. Spence, and S. Lidman. December 2012. The essence of software engineering: The SEMAT kernel. *Communications of the ACM*, 55(12). <http://queue.acm.org/detail.cfm?id=2389616>. DOI: [10.1145/2380656.2380670](https://doi.org/10.1145/2380656.2380670). 30, 95
- I. Jacobson, P.-W. Ng, P. E. McMahon, I. Spence, and S. Lidman. 2013a. *The Essence of Software Engineering: Applying the SEMAT Kernel*. Addison-Wesley. xxvi, 30, 90, 95, 336
- I. Jacobson, I. Spence, and P.-W. Ng. (October) 2013b. Agile and SEMAT: Perfect partners. *Communications of the ACM*, 11(9). <http://queue.acm.org/detail.cfm?id=2541674>. DOI: [10.1145/2524713.2524723](https://doi.org/10.1145/2524713.2524723). 30, 96
- I. Jacobson, I. Spence, and B. Kerr. 2016. Use-Case 2.0: The hub of software development. *Communications of the ACM*, 59(5): 61–69. DOI: [10.1145/2890778](https://doi.org/10.1145/2890778). 169, 222, 226, 285
- I. Jacobson, I. Spence, and P.-W. Ng. 2017. Is there a single method for the Internet of Things? *Queue*, 15.3: 20. DOI: [10.1145/3106637](https://doi.org/10.1145/3106637). 335, 339
- P. Johnson and M. Ekstedt. 2016. The Tarpit—A general theory of software engineering. *Information and Software Technology* 70: 181–203. https://www.researchgate.net/profile/Pontus_Johnson/publication/278743539_The_Tarpit_-_A_General_Theory_of_Software_Engineering/links/55b4490008aed621de0114f5/The-Tarpit-A-General-Theory-of-Software-Engineering.pdf. DOI: [10.1016/j.infsof.2015.06.001](https://doi.org/10.1016/j.infsof.2015.06.001). 91, 92, 96
- P. Johnson, M. Ekstedt, and I. Jacobson. September 2012. Where's the theory for software engineering? *IEEE Software*, 29(5). DOI: [10.1109/MS.2012.127](https://doi.org/10.1109/MS.2012.127). 84, 87, 96
- R. Knaster and D. Leffingwell. 2017. *SAFe 4.0 Distilled: Applying the Scaled Agile Framework for Lean Software and Systems Engineering*. Addison-Wesley Professional. 297, 339
- P. Kruchten. 2003. *The Rational Unified Process: An Introduction*. 3rd edition. Addison-Wesley. 345

- C. Larman and B. Vodde. 2008. *Scaling Lean & Agile Development: Thinking and Organizational Tools for Large-Scale Scrum*. Pearson Education, Inc. [346](#)
- C. Larman and B. Vodde. 2016. *Large-Scale Scrum: More with LeSS*. Addison-Wesley Professional. [297](#), [339](#)
- D. Leffingwell. 2007. *Scaling Software Agility: Best Practices for Large Enterprises*. Addison-Wesley. [346](#)
- P. E. McMahon. January/February 2015. A thinking framework to power software development team performance. *Crosstalk, The Journal of Defense Software Engineering*. <http://www.crosstalkonline.org/>. [96](#), [154](#)
- NATO. 1968. "Software Engineering: Report on a conference sponsored by the NATO Science Committee." P. Naur and B. Randell, editors. Garmisch, Germany, October 7–11. [342](#)
- S. Newman. 2015. *Building Microservices*. O'Reilly Media, Inc. [250](#), [285](#)
- P.-W. Ng. 2013. Making software engineering education structured, relevant and engaging through gaming and simulation. *Journal of Communication and Computer* 10: 1365–1373. [99](#), [153](#)
- P.-W. Ng. 2014. Theory based software engineering with the SEMAT kernel: Preliminary investigation and experiences. *Proceedings of the 3rd SEMAT Workshop on General Theories of Software Engineering*. ACM. DOI: [10.1145/2593752.2593756](https://doi.org/10.1145/2593752.2593756). [30](#), [96](#)
- P.-W. Ng. 2015. Integrating software engineering theory and practice using Essence: A case study. *Science of Computer Programming*, 101: 66–78. DOI: [10.1016/j.scico.2014.11.009](https://doi.org/10.1016/j.scico.2014.11.009). [96](#), [152](#), [154](#)
- Object Management Group. Essence—Kernel and Language for Software Engineering Methods (Essence). <http://www.omg.org/spec/Essence/1.1>. [63](#)
- OMG Essence Specification. 2014. <http://www.omg.org/spec/Essence/Current>. [95](#), [346](#)
- D. Ross. 1977. Structured Analysis (SA): A language for communicating ideas. In *IEEE Transactions on Software Engineering*, SE-3(1): 16–34. DOI: [10.1109/TSE.1977.229900](https://doi.org/10.1109/TSE.1977.229900). [344](#)
- K. Schwaber and J. Sutherland. 2016. "The Scrum Guide. The Definitive Guide to Scrum: The Rules of the Game." Scrum.org.

Index

- 201 Principles of Software Development*, 84–85
- Accept a User Story activity, 207, 214–215
- Acceptable state in Requirements alpha, 57, 215–217
- Acceptance Criteria Captured detail level for Test Case, 209
- Acceptance Criteria Listed detail level for Story Card, 209
- Achieving the Work alpha, 215
- ACM (Association for Computing Machinery), 342
- Actionability in Essence kernel, 89
- Activities
- Essence, 55
 - Essence kernel, 72–75
 - Microservices Lite, 255–257, 267–270
 - Scrum, 175–176, 178
 - Scrum Lite, 188–197
 - thing to do, 62–63
 - Use Case Lite, 229, 238–244
 - User Story Lite, 207, 211–215, 217–218
- Activity spaces
- Essence kernel, 68, 72–73
 - essentializing practices, 63–65
- Adapt in Plan-Do-Check-Adapt cycle, 132
- Adaptability in microservices, 252
- Adapts achievement level in Development competency, 61–62
- Addressed state in Requirements alpha, 57, 80
- Agile Manifesto, 335–336
- Agility and Agile methods
- Agile methods era, 25–26
 - Essence kernel relationship to, 90–91
 - introduction, 346
 - practices and methods, 335–336
- All Stories Fulfilled state in Use Case alpha, 231
- Alphas
- Chasing the State game, 105–108
 - Checkpoint Construction game, 112–113
 - composition of practices, 279–281
 - customer area of concern, 160–163
 - development, 128–132
 - development journey, 146–148
 - Essence, 54–55
 - Essence kernel, 68–72, 89, 151–152
 - kick-starting development, 122–123
 - large and complex development, 311
 - Microservices Lite, 257–259
 - Objective Go game, 108–111
 - overview, 56
 - Progress Poker game, 100–104
 - Scrum, 175–177
 - Scrum Lite, 179–182
 - states, 55–59
 - sub-alphas, 124
 - Use Case Lite, 229–233
 - User Story Lite, 207–209, 215–216
- Alternative practices, 278–279
- Ambler, Scott, 346
- Analysis competency, 76

- Analyzed state in Use-Case Slice alpha, 233
- Anomalies in development journey, 148
- Application logic, definition, 251
- Applies achievement level in Development competency, 61–62
- Architecture Selected state in Software Systems, 59, 80, 311
- Areas of concern in Essence kernel, 67–68
- Assists achievement level in Development competency, 61–62
- Association for Computing Machinery (ACM), 342
- Attainable attribute in SMART criteria, 196–197
- Automated detail level
 - Build and Deployment Script, 265
 - Test Case, 210
- AXE telecommunication switching system, 344
- Babbage, Charles, 341
- Background in practices, 34
- Backlog-Driven Development practice, 33
- Beck, Kent, 86, 346
- Booch, Grady, 345
- Bounded state in Requirements alpha, 56, 80, 215–216
- Briefly Described detail level in Use-Case
 - Narrative work product, 235
- Brooks, Frederick P., 84, 342
- Build and Deployment Script, 264–265
- Building blocks, 10
- Bulleted Outline detail level in Use-Case
 - Narrative work product, 235
- Bureau of the Census, 341
- Capabilities in practices, 33–34
- Capability Maturity Model Integration (CMMI), 306
- Capacity Described work product, 183
- Card games
 - Chasing the State, 105–108
 - Checkpoint Construction, 111–113
 - Objective Go, 108–111
- overview, 97–99
- Progress Poker, 99–105
- reflection, 113
- Cards
 - alphas, 57–58
 - Essence, 38
 - user stories, 204
- Census, 341
- Chasing the State game, 105–108
- Check in Plan-Do-Check-Adapt cycle, 131–132
- Checking in Essence, 138–139
- Checkpoint Construction game, 111–113
- Checkpoint pattern, 78–80
- Checkpoints
 - kick-starting development, 122–123
 - kick-starting development with practices, 159–165
 - large and complex development, 310
- Cloud computing for microservices, 252
- CMMI (Capability Maturity Model Integration), 306
- COBOL programming language, 342
- Code
 - thing to work with, 54, 56
 - work product cards, 60
- Coder role pattern card, 79
- Coherent state in Requirements alpha, 57, 215–216
- Collaboration
 - importance, 11–12
 - Scrum, 165–166, 174
- Collaborations and Interfaces Defined
 - detail level in Design Model work product, 261
- Common ground in Essence, 34–37
- Competencies
 - Essence, 55
 - Essence kernel, 68, 75–77
 - programming, 61–62
 - testing in, 10
- Compilers, 341–342
- Complete state in Microservices alpha, 258–259

- Completed PBIs Listed work product, 184
- Complex development. *See* Large and complex development
- Component methods, 22–25
- Component paradigm, 344–345
- Composition of practices
 - description, 276–282
 - Essence, 282–284
 - overview, 275–276
 - reflection, 282–283
- Conceived state in Requirements alpha, 56, 311
- Confirmation in user stories, 205
- Consensus-based games
 - Chasing the State, 105–108
 - Checkpoint Construction, 111–113
 - Objective Go, 108–111
 - Progress Poker, 99–105
 - reflection, 113
- Containers definition, 251
- Context in kick-starting development, 118–121
- Continual improvement in large and complex development, 321–323
- Continuous detail level in Build and Deployment Script, 265
- Conversation Captured detail level in Story Card, 209
- Conversations in user stories, 205
- Coordination in activity space, 74
- Culture issues, 330–331
- Customer area of concern
 - alphas, 160–163
 - competencies, 76
 - development perspective, 119–120
 - development process, 139
 - Essence kernel, 68–70
- Customer-related practices, 19
- Customers
 - description, 42–43
 - value for, 43–44
- DAD (Disciplined Agile Delivery)
 - agile scaling, 27
- introduction, 346
- monolithic methods, 297
- practices from, 296
- Daily Scrum activity
 - description, 173, 178
 - diagram, 175–176
 - overview, 192–193
- Daily Standup practice in Scrum, 26, 33, 173
- Data in structured methods era, 21–22
- Data processing focus, 341
- Data stores, definition, 251
- Davis, Alan, 84–85
- Definition of Done (DoD) in Scrum, 176–177
- Demonstrable alpha state, 59, 100
- Deployment in activity space, 74
- Descriptive theory of software engineering, 87–88
- Design Model work product, 254, 257, 260–263
- Design overview, 14
- Design Patterns Identified detail level in Design Model work product, 262
- Design phase
 - iterative method, 21
 - waterfall method, 19–20
- Detail levels
 - Build and Deployment Script, 265
 - Microservice Design work product, 264
 - Story Card, 209
 - Test Case, 210
 - Use Case Lite work products, 233–236
 - Use-Case Narrative work product, 235
 - Use-Case Slice Test Case work product, 237
 - work products, 60–61
- Developers
 - Scrum, 173
 - Tarpit theory, 92
- Development
 - doing and checking, 138–139
 - kick-starting. *See* Kick-starting development; Kick-starting development with practices
 - overview, 127–132

- Development (*continued*)
 - Plan-Do-Check-Adapt cycle, 128–132
 - plans, 132–138
 - way of working, 140–142
- Development competency, 61–62, 77
- Development Complete checkpoint, 80
- Development endeavor, 79–80
- Development journey
 - anomalies, 148
 - overview, 145
 - progress and health, 146–148
 - visualizing, 145–146
- Development types
 - culture issues, 330–331
 - overview, 325
 - practice and method architectures, 326–328
 - practice libraries, 328–330
- DevOps practice, 302
- Dijkstra, E. W., 86, 342–343
- Disciplined Agile Delivery (DAD)
 - agile scaling, 27
 - introduction, 346
 - monolithic methods, 297
 - practices from, 296
- Disciplined approach in software engineering, 14–15
- Do in Plan-Do-Check-Adapt cycle, 131
- Document elements in Essence, 54
- DOD (Definition of Done) in Scrum, 176–177
- Doing alpha in PBIs, 181
- Doing in development, 138–139
- Done alpha in PBIs, 181
- Done term, definition, 99–100
- EA (enterprise architecture), 24
- Endeavor area of concern
 - competencies, 77
 - development perspective, 120–121
 - development process, 136–139
 - Essence kernel, 68, 70–71
 - kick-starting development with practices, 163–165
 - practices, 19
- Scrum, 199
- Endeavors
 - description, 42–43
 - teams, 48–49
 - ways of working in, 49–50
 - work in, 49
- Engaging user experiences, 37–38
- Enterprise architecture (EA), 24
- Ericsson AB, 344
- Essence
 - common ground, 34–37
 - composition of practices, 282–284
 - development. *See* Development
 - development journey. *See* Development journey
 - engaging user experiences, 37–38
 - essentializing practices, 63–65
 - essentials focus, 37
 - evolution, 346
 - insights, 32
 - kick-starting development. *See* Kick-starting development
 - language, 54–61
 - large and complex development, 310–311, 322–324
 - methods and practices, 32–34
 - microservices, 252–256
 - OMG standard, 29–30
 - overview, 31
 - practices, 298–299
 - purpose, 42
 - Scrum with, 174–179
 - serious games. *See* Serious games
 - theory of software engineering, 87–91
 - Use Case Lite practice, 227–230
 - User Story Lite practice, 207–208
 - work products, 60
- Essence kernel
 - actionability, 89
 - activities, 72–75
 - alphas, 68–72
 - applying, 151–152
 - competencies, 75–77
 - extensibility, 90

- growth from, 93–94
- observations, 151
- organizing with, 67–69
- overview, 67
- patterns, 77–80
- practicality, 88–89
- relationship to other approaches, 90–91
- User Story Lite practice, 215–218
- validity, 151
- Essential Outline detail level in Use-Case
 - Narrative work product, 235
- Essentialized practices, 35–36
- Essentializing practices
 - composition of practices, 283–284
 - description, 35–36
 - Essence, 298–299
 - for libraries, 329
 - monolithic methods and fragmented practices, 296–298
 - overview, 63–65
 - reusable, 299–302
 - sources, 295–296
- Estimatable criteria in user stories, 205–206
- Evolve Microservice activity, 255, 257, 269–270
- Exchangeable packages, 345
- Explicit approaches in Scrum, 173–174
- Extensibility
 - Essence kernel, 90
 - software systems, 47
- Extension practices, 279
- Extreme Programming Explained*, 86
- Extreme Programming (XP)
 - introduction, 346
 - practices from, 296
 - user stories, 203
- Feedback in Use Case Lite practice, 239
- Find Actors and Use Cases activity, 229, 238–239
- Find User Stories activity, 207, 212
- Formed state in Teams, 311
- Fortran programming language, 342
- Foundation Established state in Way of working, 311
- Fragmented practices, 296–298
- Fulfilled alpha state, 57
- Fully Described detail level in Use-Case
 - Narrative work product, 235
- Function-data paradigm, 344
- Functionality in software systems, 46
- Functions in structured methods era, 21–22
- Future, dealing with
 - agility, 335–336
 - methods evolution, 338–339
 - methods use, 337–338
 - overview, 333–335
 - teams and methods, 337
- Games
 - Chasing the State, 105–108
 - Checkpoint Construction, 111–113
 - Objective Go, 108–111
 - overview, 97–99
 - Progress Poker, 99–105
 - reflection, 113
- General predictive theory of software engineering, 91–92
- “Go to statement considered harmful” article, 86
- Goal Established state in Use Case alpha, 230
- Goals Specified work product, 183
- Gregor, Shirley, 84–85
- Hacking vs. programming, 6
- Handle favorites use-case slice, 242–243
- Happy day scenarios, 224
- Health and progress
 - development journey, 146–148
 - Essence, 54
 - Microservices Lite, 271–272
 - use-case slices, 245–246
- Hemdal, Göran, 344
- Higher-level languages, 342
- History of software and software engineering, 341–347

- Hollerith punched card equipment, 341
- Hopper, Grace Murray, 341–342
- Identification of microservices, 251–252
- Identified state
 - Microservices Lite, 258
 - user stories, 209
- Identify Microservices activity, 255, 257, 267–268
- IEEE (Institute of Electrical and Electronic Engineering), 342
- Implementation phase
 - activity space, 74
 - iterative method, 21
 - waterfall method, 19–20
- Implemented state in Use-Case Slice alpha, 233
- In Progress state in user stories, 209
- Increment elements
 - description, 177
 - work products, 183–184
- Increment Notes Described work product, 184
- Incremental development in use cases slices, 226–227
- Independent criteria in user stories, 205
- Innovates achievement level in Development competency, 61–62
- Institute of Electrical and Electronic Engineering (IEEE), 342
- Interfaces Specified detail level in Microservice Design work product, 264
- Internal Elements Designed detail level in Microservice Design work product, 264
- Internal Structure Defined detail level in Microservice Design work product, 264
- INVEST criteria for user stories, 205–206
- ISO/IEC 12207 standard, 345
- Items Ordered work product, 182
- Iterative operations
 - development, 127
- development journey, 147
- large and complex development, 319–321
- lifecycle methods, 20–21
- Jackson, Michael, 344
- Jackson Structured Programming (JSP), 344
- Jacobson, Ivar
 - component paradigm, 344
 - method prison governing, 27
 - OMG, 345
 - RUP, 345
 - SEMAT, 28, 346
 - Use-Case Driven Development practice, 221
- JSP (Jackson Structured Programming), 344
- Kernel. *See* Essence kernel
- Key elements of software engineering
 - basics, 41–43
 - endeavors, 48–50
 - overview, 41
 - value for customers, 43–45
 - value through solutions, 45–48
- Kick-starting development
 - context, 118–121
 - overview, 117–118
 - scope and checkpoints, 122–123
 - things to watch, 124–126
- Kick-starting development with practices
 - context, 158–159
 - overview, 157–158
 - practices to apply, 165–167
 - scope and checkpoints, 159–165
 - things to watch, 167–169
- Kruchten, Philippe
 - method prison governing, 27
 - RUP, 345
- Language of software engineering
 - competencies, 61–62
 - essentializing practices, 63–65
 - overview, 53
 - practice example, 53–54
 - things to do, 62–63

- things to work with, 54–61
- Large and complex development**
 - alphas, 311
 - common vision, 315–317
 - continual improvement, 321–323
 - Essence, 310–311, 322–324
 - iterative operations, 319–321
 - kick-starting, 309–315
 - large-scale development, 308–309
 - large-scale methods, 306–308
 - managing, 317–319
 - overview, 305–306
 - practices, 310–313
 - running, 315–322
 - scope and checkpoints, 310
 - things to watch, 313–315
- Large-scale integrated circuits, 343
- Large-Scale Scrum (LeSS)**
 - agile scaling, 27
 - introduction, 346
 - monolithic methods, 297
 - practices from, 296
- Larman, Craig, 346
- Lawson, Harold “Bud,” 345
- Leadership competency**, 77
- Leffingwell, Dean, 346
- LeSS (Large-Scale Scrum)**
 - agile scaling, 27
 - introduction, 346
 - monolithic methods, 297
 - practices from, 296
- Levels of detail**
 - Build and Deployment Script, 265
 - Microservice Design work product, 264
 - Story Card, 209
 - Test Case, 210
 - Use Case Lite work products, 233–236
 - Use-Case Narrative work product, 235
 - Use-Case Slice Test Case work product, 237
 - work products, 60–61
- Libraries for practices, 328–330
- Lifecycles, 19–21
- Lines, Mark, 346
- Lovelace, Ada, 341
- Machine instruction level**, 341
- Make Evolvable activity**, 255, 257, 268–269
- Management competency**, 77
- Martin, Robert, 90
- Masters achievement level in Development competency**, 61–62
- Mayer, Bertrand, 28
- Measurable attribute in SMART criteria**, 196–197
- Method prison**, 27
- Methods**
 - agile methods era, 25
 - component methods era, 22–25
 - consequences, 26–28
 - definition, 19
 - Essence, 32–34
 - evolution, 338–339
 - large-scale, 306–308
 - lifecycles, 19–21
 - people practices, 25–26
 - rise of, 18–19
 - structured methods era, 21–22
 - team ownership, 337
 - technical practices, 21–25
 - use focus, 337–338
- Methods war**, 22, 26–27
- Meyer, Bertrand, 346
- Microprocessors**, 343
- Microservice alpha**, 254, 257
- Microservice Build and Deployment work product**, 254, 257
- Microservice Design work product**, 254, 257, 263–264
- Microservice Test Case work product**, 255, 257, 265–267
- Microservices**, 166–169
 - description, 250–252
 - Essence, 252–256
 - overview, 249–250
- Microservices Lite practice**
 - activities, 255–256, 267–270
 - alphas, 257–259

- Microservices Lite practice (*continued*)
 - Build and Deployment Script, 264–265
 - description, 256–257
 - design model, 260–263
 - impact, 270–271
 - Microservice Design work product, 263–264
 - Microservice Test Case work product, 265–267
 - overview, 253–256
 - progress and health, 271–272
 - reusable practices, 299–300
 - work products, 259–267
- Mini-computers, 343
- Mini-methods, 19
- Minimal state in Microservices Lite, 258
- Modular approaches in Scrum, 173–174
- Monolithic methods, 296–298
- Mythical Man-Month*, 84
- NATO-sponsored conference, 342
- Negotiable criteria in user stories, 205
- NZ Transport Agency, 18
- Object Management Group (OMG) standard
 - Essence, 29–30, 346
 - Essence kernel, 71
 - notation, 23–24
 - UML standard, 345
- Object-oriented programming
 - acceptance, 344–345
 - components in, 23
- Objective Go game, 108–111
- On the Criteria to Be Used in Decomposing Systems into Modules*, 86
- Operational alpha state, 59
- Opportunity
 - alpha state card, 72
 - customer area of concern, 69, 71
 - development context, 158
 - development endeavors, 42–43
 - development perspective, 119–120
 - development plans, 133–134
 - large and complex development, 311–312
- scope and checkpoints, 161–162
- value for customers, 43–44
- Outlined detail level in Build and Deployment Script, 265
- Pair programming teams, 26
- Paradigm shifts, 22–23
- Paradigmatic theories, 85
- Paths in use cases slices, 228
- Patterns
 - Essence kernel, 68, 77–80
 - essentializing practices, 63–65
 - Scrum, 178–179, 184–186
- PBIs. *See* Product Backlog Items (PBIs)
- People practices, 25–26
- Performance in software systems, 47
- Perlis, Alan, 92
- PLA (product-line architecture), 24
- Plan-Do-Check-Adapt cycle, 128–132
- Planned alpha in sprints, 179
- Plans
 - development, 132–138
 - Plan-Do-Check-Adapt cycle, 128–131
 - Scrum Lite, 188–192
- POs (product owners)
 - description, 178
 - pattern cards, 184–185
 - Scrum, 172–173, 175
- Possibilities in activity space, 73
- Post-development phase in development
 - endeavor, 79–80
- Practicality in Essence kernel, 88–89
- Practice separation in Essence kernel, 90
- Practices
 - agile methods era, 25
 - background, 34
 - capabilities, 33–34
 - common ground, 34–37
 - component methods era, 22–25
 - composition of. *See* Composition of practices
 - consequences, 26–28
 - definition, 174
 - Essence, 32–34, 298–299

- fragmented practices, 296–298
- kick-starting development with. *See* Kick-starting development with practices
- large and complex development, 310–313
- libraries, 328–330
- lifecycles, 19–21
- people, 25–26
- reusable, 299–302
- rise of, 18–19
- Scrum, 173–174, 177, 198–199
- sources, 295–296
- structured methods era, 21–22
- technical, 21–25
- types, 19
- Pre-development phase in development endeavor, 79–80
- Precision in Scrum, 200–202
- Preparation in activity space, 74
- Prepare a Use-Case Slice activity, 229, 242–243
- Prepare a user story activity, 207, 212–213
- Prepared state
 - Use-Case Slice alpha, 232–233
 - Work alpha, 215
- Priorities in Scrum, 172
- Problems in kick-starting development, 118
- Product Backlog Items (PBIs)
 - alphas, 181
 - description, 172, 177
 - example, 176
 - identifying, 173
 - Scrum, 168
- Product Backlog practice, 302
- Product Backlog work product
 - activity cards, 190–192
 - description, 177
 - Scrum Lite, 182–184
- Product-line architecture (PLA), 24
- Product Management practice, 302
- Product owners (POs)
 - description, 178
 - pattern cards, 184–185
 - Scrum, 172–173, 175
- Product Ownership practice, 301
- Product Retrospective practice, 302
- Product Sprint practice, 302
- Program backlog management, 318
- Program practices, 302–303
- Programming, defined, 4
- Programming and software engineering differences, 6–8
 - intern view, 8–10
 - overview, 3–4
 - professional view, 10–12
 - programming, 4–6
 - software engineering, 12–15
- Progress and health
 - activity space, 74–75
 - development journey, 146–148
 - Essence, 54
 - Microservices Lite practice, 271–272
 - use-case slices, 245–246
- Progress Poker game
 - benefits, 102
 - example, 103–105
 - overview, 99–102
- Progressing
 - use-case slices, 232–233
 - use cases, 230–232
- Provided interface, UML notation for, 261
- Quality in software systems, 47–48
- Quantifiable approach in software engineering, 14–15
- Rapidly Deployable state in Microservices Lite practice, 258
- Rational Unified Process (RUP)
 - development of, 24, 345
 - large-scale development, 306
 - monolithic methods, 297
- Reaching out in scaling, 293
- Ready for Development checkpoint, 80
- Ready for Development state in User Story, 209
- Ready requirement, 80

- Ready state
 - PBIs, 181
 - Software Systems, 59
- Recognized state for Stakeholders, 311
- Relevant attribute in SMART criteria, 196–197
- Reliability in software systems, 47
- Required Behavior Defined detail level in Microservice Design work product, 264
- Required interface, UML notation for, 261
- Requirements
 - activity space, 74
 - alpha state card, 72
 - alphas, 56–58
 - development context, 158
 - development perspective, 120
 - development plans, 134–135
 - large and complex development, 311–312
 - Ready for Development checkpoint, 80
 - scope and checkpoints, 161–162
 - solution area of concern, 70
 - in solutions, 42–43, 45–46
 - thing to work with, 54–56
 - User Story Lite practice, 225, 227, 230
- Requirements alpha
 - Progress Poker game, 100–101
 - User Story Lite practice, 215–217
- Requirements engineering, 13–14
- Requirements phase
 - iterative method, 21
 - waterfall method, 19–20
- Retired alpha state, 59
- Retrospective practice in Scrum, 33
- Reusable practices, 19, 299–302
- Reviewed alpha in sprints, 179–180
- Roles in Scrum Lite, 184–186
- Roles pattern, 77–78
- Ross, Douglas, 344
- Royce, Walker, 345
- Rumbaugh, James, 345
- RUP (Rational Unified Process)
 - development of, 24, 345
 - large-scale development, 306
- monolithic methods, 297
- SA/SD (Structured Analysis/Structured Design), 21
- SaaS (Software as a Service), 8
- SADT (Structured Analysis and Design Technique)
 - description, 21–22
 - development of, 344
- Scaled Agile Framework (SAFe)
 - agile scaling, 27
 - introduction, 346
 - monolithic methods, 297
 - practices from, 296
- Scaled Professional Scrum (SPS)
 - agile scaling, 27
 - introduction, 346
 - practices from, 296
- Scaling
 - challenges, 289–291
 - dimensions of, 291–294
 - large and complex development. *See* Large and complex development
 - overview, 289
 - reaching out, 293
 - scaling up, 292–293
 - zooming in, 291–292
- Scenario Chosen detail level in Use-Case Slice Test Case work product, 237
- Scenarios in use cases slices, 228
- Scheduled alpha in sprints, 179
- Schwaber, Ken, 346
- Scope
 - kick-starting development, 122–123
 - kick-starting development with practices, 159–165
 - large and complex development, 310
- Scoped state in Use-Case Slice alpha, 232
- Scripted detail level in Test Case, 210
- Scripted or Automated detail level in Use-Case Slice Test Case work product, 237
- Scrum
 - collaboration, 165–166, 174

- components, 33
- composite practices, 306–307
- description, 168
- with Essence, 174–179
- fragmented practices, 297
- introduction, 346
- overview, 171–173
- practices, 173–174, 198–199, 296
- precision, 200–202
- reflections, 198–202
- Scrum Lite**
 - activities, 188–197
 - alphas, 179–182
 - overview, 174–177
 - planning, 188–192
 - roles, 184–186
 - usage, 187–188
 - work products, 182–184
- Scrum Masters**
 - description, 173, 178–179
 - large and complex development, 321–322
 - pattern cards, 184–186
 - patterns, 175
- Scrum of Scrums meetings, 320
- Scrum Teams**
 - description, 179
 - Essence, 175
 - pattern cards, 185–186
- SDL (Specification and Description Language)**, 344
- Self-organizing teams, 26
- SEMAT (Software Engineering Method And Theory)**
 - description, 28–29
 - founding, 346
- Serious games
 - Chasing the State, 105–108
 - Checkpoint Construction, 111–113
 - Objective Go, 108–111
 - overview, 97–99
 - Progress Poker, 99–105
 - reflection, 113
- Service-oriented architecture (SOA), 24
- Simplest Story Fulfilled state in Use Case**
 - alpha, 231
- Simula 67 language, 23
- Slice the Use Cases activity**
 - description, 229
 - working with, 241–242
- Slicing use cases, 226–227
- Small attribute**
 - SMART criteria, 196–197
 - user stories, 206
- Smalltalk language, 23
- SMART criteria, 196–197
- “So that” clauses in user stories, 206
- SOA (service-oriented architecture), 24
- Social issues, 330–331
- Software as a Service (SaaS), 8
- Software crisis, 18, 343
- Software development, defined, 4
- Software Engineering Method And Theory (SEMAT)**
 - description, 28–29
 - founding, 346
- Software engineering overview
 - challenges, 17–18
 - defined, 4–5, 14–15
 - history, 341–347
 - key elements. *See* Key elements of software engineering
 - language. *See* Language of software engineering
 - methods and practices, 18–28
 - OMG standard, 29–30
 - and programming. *See* Programming and software engineering
 - SEMAT initiative, 28–29
 - Tarpit theory, 92
 - theory, 84–87
- Software Life Cycle Processes, 345
- Software Systems**
 - alpha cards, 58, 72
 - Demonstrable alpha state card, 100
 - development context, 158
 - development perspective, 120
 - development plans, 135–136

- Software Systems (*continued*)
 - large and complex development, 311–312
 - Objective Go game, 109–111
 - scope and checkpoints, 161, 162
 - solutions, 42–43, 45–48, 70
 - thing to work with, 54–56
- Soley, Richard, 28, 346
- Solution area of concern
 - competencies, 76–77
 - development perspective, 120
 - development process, 139
 - Essence kernel, 68–70
 - kick-starting development with practices, 161
- Solution-related practices, 19
- Solutions
 - description, 42–43
 - value through, 45–48
- Specification and Description Language (SDL), 344
- Splitting User Stories activity, 207, 213–214
- Sprint Backlog
 - activity cards, 190–191
 - description, 177
 - PBIs, 172
 - work products, 183
- Sprint Planning activity
 - activity cards, 188–192
 - description, 178
- Sprint Retrospective activity
 - activity cards, 195–196
 - Scrum, 178
- Sprint Review activity
 - activity cards, 193–195
 - description, 172, 178
- Sprints
 - alphas, 179–181
 - description, 177
 - Scrum, 172–173
- SPS (Scaled Professional Scrum)
 - agile scaling, 27
 - introduction, 346
 - practices from, 296
- Stakeholder alpha in Chasing the State game, 105–107
- Stakeholder Representation competency, 76
- Stakeholders
 - activity space, 74
 - alpha state card, 72
 - customer area of concern, 69, 71
 - as customers, 42–43
 - development context, 158
 - development perspective, 119
 - development plans, 133
 - large and complex development, 311–312
 - Objective Go game, 108–111
 - scope and checkpoints, 159–160
 - value for, 44–45
- Started state in Work alpha, 311
- States in alphas, 55–59
- Stored program computers, 341
- Story Card work product, 207, 209–210
- Story practice, 166
- Story Structure Understood state in Use Case alpha, 231
- Structure and Approach Described detail level in Design Model work product, 260
- Structured Analysis and Design Technique (SADT)
 - description, 21–22
 - development of, 344
- Structured Analysis/Structured Design (SA/SD), 21
- Structured detail level in Use-Case Model work product, 235
- Structured methods era, 21–22
- Student Pairs pattern card, 78
- Sub-alphas, 124
- Subsystems in UML notation, 261
- Sufficient Stories Fulfilled state in Use Case alpha, 231
- Support in activity space, 74–75
- Sutherland, Jeff, 346
- SWEBOK, 84–85
- System Boundary Established detail level in Use-Case Model work product, 234
- Systematic approach in software engineering, 14–15

- Tarpit theory, 91–92
- TD (test-driven development) in Essence, 36
- TDD (Test-Driven Development) in Extreme Programming, 346
- Team Backlog practice, 301
- Team Retrospective practice
 - description, 301
 - large and complex development, 321–322
- Team Sprint practice, 301
- Teams
 - activity space, 74–75
 - agile, 26
 - alpha state card, 72
 - development perspective, 120
 - development plans, 136–137
 - endeavor area of concern, 42–43, 48–49, 70–71
 - Essence, 36
 - large and complex development, 311–312
 - methods ownership, 337
 - need for, 12–13
 - scope and checkpoints, 163–164
- Technical practices, 21–25
- Technology stacks, 10
- Test a Use-Case Slice activity
 - description, 229
 - working with, 243–244
- Test Automated detail level in Microservice
 - Test Case work product, 267
- Test Case work product, 207, 209–210
- Test Dependencies Managed detail level in Microservice Test Case work product, 266
- Test-driven development (TD) in Essence, 36
- Test-Driven Development (TDD) in Extreme Programming, 346
- Test Scenarios Chosen detail level in Microservice Test Case work product, 266
- Testable attribute
 - SMART criteria, 196–197
 - user stories, 206
- Testing
 - activity space, 74
 - waterfall method phase, 19–20
- Testing competency, 10, 77
- Theory
 - arguments, 85–87
 - Essence, 87–91
 - general predictive theory, 91–92
 - growth from, 93–94
 - overview, 83–84
 - software engineering, 84–87
 - uses, 87
- Things to do
 - activities, 62–63
 - backlogs, 49
 - composition, 279
 - Essence kernel, 72–75
- Things to watch
 - kick-starting development, 124–126
 - kick-starting development with practices, 167–169
 - large and complex development, 313–315
- Things to work with
 - alpha states, 56–59
 - alphas, 56
 - Essence kernel, 69–72, 89
 - overview, 54–56
 - Use Case Lite practice, 230–234
 - work products, 59–61
- To Do alpha in PBIs, 181
- Turing tar-pit, 92
- UCDD (Use-Case Driven Development) practice, 221–222
- Unified Modeling Language (UML) standard
 - development of, 24
 - introduction, 345
 - Microservices Lite practice, 260–261
 - primer, 260
 - use cases, 222–223
- Unified Process prison, 27
- Unified Process (UP), 24, 345
- Univac I computer, 341
- University of Wisconsin, 18
- UP (Unified Process), 24, 345
- Usable alpha state, 59
- Use Case alpha, 229–231

- Use-Case diagrams, 24
- Use-Case Driven Development (UCDD)
 - practice, 221–222
- Use Case Lite practice
 - activities, 238–244
 - alphas, 229–233
 - Essence, 227–230
 - impact, 244–245
 - kick-starting, 237–240
 - overview, 221–222
 - reusable practices, 299–300
 - use-case slices progress and health, 245–246
 - use cases description, 222–226
 - use cases slicing, 226–227
 - user stories vs. use cases, 246–248
 - work products, 233–236
 - working with, 240–244
- Use-Case Model work product, 227, 229, 234–235
- Use-Case Narrative work product, 227, 229, 235–236
- Use-case narratives, 224–225
- Use case practices, 166, 168–169
- Use-Case Slice alpha, 229, 232–233
- Use-Case Slice Test Case work product, 227, 229, 236–237
- Use-case slices
 - process, 226–227
 - progress and health, 245–246
- Use Cases
 - introduction, 345
 - practices from, 296
- User experiences in Essence, 37–38
- User interface, definition, 251
- User stories
 - description, 204–207
 - Scrum teams, 166
- User Stories practice
 - description, 168–169
 - vs. use cases, 246–248
- User Story alpha in User Story Lite practice, 207–208
- User Story for Extreme Programming, 346
- User Story Lite practice
 - activities, 211–215
 - alphas, 207–209
 - Essence, 207–208
 - Essence kernel, 215–218
 - impact, 216–218
 - overview, 203
 - usage, 211
 - user story description, 204–207
 - work products, 209–210
- Validity in Essence kernel, 151
- Valuable criteria in user stories, 205
- Value
 - for customers, 43–45
 - through solutions, 45–48
- Value Established detail level in Use-Case Model work product, 234
- Value Established state in Opportunity, 311
- Value Expressed detail level in Story Card, 209
- Variables Identified detail level in Use-Case Slice Test Case work product, 237
- Variables Set detail level in Use-Case Slice Test Case work product, 237
- Verification phase
 - iterative method, 21
 - waterfall method, 19–20
- Verified state
 - Use-Case Slice alpha, 233
 - user stories, 209
- Vodde, Bas, 346
- von Neumann, John, 341
- Waterfall method
 - description, 19–20
 - development of, 344
- Way of working
 - adapting, 140–141
 - alpha state card, 72
 - development context, 158
 - development perspective, 120–121
 - development plans, 138

- endeavor area of concern, 42–43, 49–50, 71
- Essence kernel, 141–142
- large and complex development, 311–312
 - scope and checkpoints, 163, 165
- “Where’s the Theory for Software Engineering?” paper, 84
- Work activity
 - alpha state card, 72
 - development context, 158
 - development perspective, 120
 - development plans, 136–137
 - endeavor area of concern, 42–43, 49, 71
 - large and complex development, 311–312
 - scope and checkpoints, 163–164
- Work alpha, 215–216
- Work Forecast Described work product, 183
- Work products
 - Essence, 54–55
 - Microservices Lite practice, 259–267
 - overview, 59–61
 - Scrum, 175, 177
 - Scrum Lite, 182–184
 - Use Case Lite practice, 229, 233–236
 - User Story Lite practice, 207, 209–210
- Write Code activity cards, 62–63
- XP (Extreme Programming)
 - introduction, 346
 - practices from, 296
 - user stories, 203
- Zooming in in scaling, 291–292

Author Biographies

Ivar Jacobson



Dr. Ivar Jacobson received his Ph.D. in computer science from KTH Royal Institute of Technology, was awarded the Gustaf Dalén medal from Chalmers in 2003, and was made an honorary doctor at San Martin de Porres University, Peru, in 2009. Ivar has both an academic and an industry career. He has authored ten books, published more than a hundred papers, and is a frequent keynote speaker at conferences around the world.

Ivar Jacobson is a key founder of components and component architecture, work that was adopted by Ericsson and resulted in the greatest commercial success story ever in the history of Sweden (and it still is). He is the creator of use cases and Objectory—which, after the acquisition of Rational Software around 2000, resulted in the Rational Unified Process, a popular method. He is also one of the three original developers of the Unified Modeling Language. But all this is history. His most recently founded company, Ivar Jacobson International, has been focused since 2004 on using methods and tools in a smart, superlight, and agile way. Ivar is also a founder and leader of a worldwide network, SEMAT, whose mission is to revolutionize software development based on a kernel of software engineering. This kernel has been realized as a formal standard called Essence, which is the key idea described in this book.

Harold “Bud” Lawson



Professor Emeritus Dr. Harold “Bud” Lawson (The Institute of Technology at Linköping University) has been active in the computing and systems arena since 1958 and has broad international experience in private and public organizations as well as academic environments. Bud contributed to several pioneering efforts in hardware and software technologies. He has held professorial appointments at several universities in the USA, Europe, and the Far East. A Fellow of the ACM, IEEE, and INCOSE, he was also head of the Swedish delegation to ISO/IEC JTC1 SC7 WG7 from 1996 to 2004 and the elected architect of the ISO/IEC 15288 standard. In 2000, he received the prestigious IEEE Computer Pioneer Charles Babbage medal award for his 1964 invention of the pointer variable concept for programming languages. He has also been a leader in systems engineering. In 2016, he was recognized as a Systems Engineering Pioneer by INCOSE. He has published several books and was the coordinating editor of the “Systems Series” published by College Publications, UK.

Tragically, Harold Lawson passed away after battling an illness for almost a year, just weeks before the publication of this book.

Pan-Wei Ng



Dr. Pan-Wei Ng has been helping software teams and organizations such as Samsung, Sony, and Huawei since 2000, coaching them in the areas of software development, architecture, agile, lean, DevOps, innovation, digital, Beyond Budgetings, and Agile People. Pan-Wei firmly believes that there is no one-size-fits-all, and helps organizations find a way of working that suits them best. This is why he is so excited about Essence and has been working with it through SEMAT since their inception in 2006, back when Essence was a mere idea. He has contributed several key concepts to the development of Essence.

Pan-Wei coauthored two books with Dr. Ivar Jacobson and frequently shares his views in conferences. He currently works for DBS Singapore, and is also an adjunct lecturer in the National University of Singapore.

Paul E. McMahon

Paul E. McMahon has been active in the software engineering field since 1973 after receiving his master's degree in mathematics from the State University of New York at Binghamton (now Binghamton University). Paul began his career as a software developer, spending the first twenty-five years working in the US Department of Defense modeling and simulation domain. Since 1997, as an independent consultant/coach (<http://pemsystems.com>), Paul helps organizations and teams using a hands-on practical approach focusing on agility and performance.

Paul has taught software engineering at Binghamton University, conducted workshops on software engineering and management, and has published more than 50 articles and 5 books. Paul is a frequent speaker at industry conferences. He is also a Senior Consulting Partner at Software Quality Center. Paul has been a leader in the SEMAT initiative since its initial meeting in Zurich.

Michael Goedicke

Prof. Dr. Michael Goedicke is head of the working group Specification of Software Systems at the University of Duisburg-Essen. He is vice president of the GI (German National Association for Computer Science), chair of the Technical Assembly of the IFIP (International Federation for Information Processing), and longtime member and steering committee chair of the IEEE/ACM conference series Automated Software Engineering. His research interests include, among others, software engineering methods, technical specification and realization of software systems, and software architecture and modeling. He is also known for his work in views and viewpoints in software engineering and has quite a track record in software architecture. He has been involved in SEMAT activities nearly from the start, and assisted in the standardization process of Essence—especially the language track.

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

The first course in software engineering is the most critical. Education must start from an understanding of the heart of software development, from familiar ground that is common to all software development endeavors. This book is an in-depth introduction to software engineering that uses a systematic, universal kernel to teach the essential elements of **all** software engineering methods.

This kernel, Essence, is a vocabulary for defining methods and practices. Essence was envisioned and originally created by Ivar Jacobson and his colleagues, developed by Software Engineering Method and Theory (SEMAT) and approved by The Object Management Group (OMG) as a standard in 2014. Essence is a practice-independent framework for thinking and reasoning about the practices we have and the practices we need. Essence establishes a shared and standard understanding of what is at the heart of software development. **Essence is agnostic to any particular method, lifecycle independent, programming language independent, concise, scalable, extensible, and formally specified.** Essence frees the practices from their method prisons.

The first part of the book describes Essence, the essential elements to work with, the essential things to do and the essential competencies you need when developing software. The other three parts describe more and more advanced use cases of Essence. Using real but manageable examples, it covers the fundamentals of Essence and the innovative use of serious games to support software engineering. It also explains how current practices such as user stories, use cases, Scrum, and micro-services can be described using Essence, and illustrates how their activities can be represented using the Essence notions of cards and checklists. The fourth part of the book offers a vision how Essence can be scaled to support large, complex systems engineering.

Essence is supported by an ecosystem developed and maintained by a community of experienced people worldwide. From this ecosystem, professors and students can select what they need and create their own way of working, thus learning how to create ONE way of working that matches the particular situation and needs.

ABOUT ACM BOOKS



ACM Books is a new series of high quality books for the computer science community, published by ACM in collaboration with Morgan & Claypool Publishers. ACM Books publications are widely distributed in both print and digital formats through booksellers and to libraries (and library consortia) and individual ACM members via the ACM Digital Library platform.

ISBN 978-1-947487-24-6

90000



9 781947 487246