THE ESSENCE of SOFTWARE ENGINEERING

Applying the SEMAT Kernel

IVAR JACOBSON
PAN-WEI NG
PAUL E. MCMAHON
IAN SPENCE
SVANTE LIDMAN

The Essence of Software Engineering

This page intentionally left blank

The Essence of Software Engineering

Applying the SEMAT Kernel

Ivar Jacobson
Pan-Wei Ng
Paul E. McMahon
Ian Spence
Svante Lidman

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Figures P-1, P-2, P-3, 2-1, 3-1, 3-2, 3-4 and 22-2 are provided courtesy of the Software Engineering Method and Theory (SEMAT) community.

Cataloging-in-Publication Data is on file with the Library of Congress.

ISBN-13: 978-0-321-88595-1
ISBN-10: 0-321-88595-3
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, January 2013

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

This page intentionally left blank

Contents

Foreword by Robert Martin

The pendulum has swung again. This time it has swung toward craftsmanship. As one of the leaders of the craftsmanship movement, I think this is a good thing. I think it is important that software developers learn the pride of workmanship that is common in other crafts.

But when the pendulum swings, it often swings away from something else. And in this case it seems to be swinging away from the notion of engineering. The sentiment seems to be that if software is a craft, a kind of artistry, then it cannot be a science or an engineering discipline. I disagree with this rather strenuously.

Software is both a craft and a science, both a work of passion and a work of principle. Writing good software requires both wild flights of imagination and creativity, as well as the hard reality of engineering tradeoffs. Software, like any other worthwhile human endeavor, is a hybrid between the left and right brain.

This book is an attempt at describing that balance. It proposes a software engineering framework or *kernel* that meets the need for engineering discipline, while at the same time leaving the development space open for the creativity and emergent behavior needed for a craft.

Most software process descriptions use an assembly line metaphor. The project moves from position to position along the line until it is complete. The prototypical process of this type is the waterfall, in which the project moves from Analysis to Design to Implementation. In RUP the project moves from Inception to Elaboration to Construction to Transition.

The kernel in this book represents a software development effort as a continuously operating abstract mechanism composed of components and relationships. The project does not move from position to position within this mechanism as in the assembly line metaphor. Rather, there is a continuous flow through the mechanism as opportunities are transformed into requirements, and then into code and tests, and then into deployments.

The state of that mechanism is exposed through a set of critical indicators, called *alphas*, which represent how well the underlying components are functioning. These alphas progress from state to state through a sequence of actions taken by the development team in response to the current states.

As the project progresses, the environment will change, the needs of the customer will shift, the team will evolve, and the mechanism will get out of kilter. The team will have to take further actions to tune the mechanism to get it back into proper operation.

This metaphor of a continuous mechanism, as opposed to an assembly line, is driven by the agile worldview. Agile projects do not progress through phases. Rather, they operate in a manner that continuously transforms customer needs into software solutions. But agile projects can get out of kilter. They might get into a mode where they aren't refactoring enough, or they are pairing too much, or their estimates are unreliable, or their customers aren't engaged.

The kernel in this book describes the critical indicators and actions that allow such malfunctions to be detected and then corrected. Teams can use it to tune their behaviors, communications, workflows, and work products in order to keep the machine running smoothly and predictably.

The central theme of the book is excellent. The notion of the alphas, states, and actions is compelling, simple, and effective. It's just the right kind of idea for a kernel. I think it is an idea that could help the whole software community.

If you are deeply interested in software process and engineering, if you are a manager or team leader who needs to keep the development organization running like a well-oiled machine, or if you are a CTO in search of some science that can help you understand your development organizations, then I think you'll find this book very interesting.

After reading the book, I found myself wanting to get my hands on a deck of cards so that I could look through them and play with them.

—Robert Martin
(unclebob)
February 2012

This page intentionally left blank

Foreword
by Bertrand Meyer

Software projects everywhere look for methodology and are not finding it. They do, fortunately, find individual practices that suit them; but when it comes to identifying a coherent set of practices that can guide a project from start to finish, they are too often confronted with dogmatic compendiums that are too rigid for their needs. A method should be adaptable to every project's special circumstances: it should be backed by strong, objective arguments; and it should make it possible to track the benefits.

The work of Ivar Jacobson and his colleagues, started as part of the SEMAT initiative, has taken a systematic approach to identifying a "kernel" of software engineering principles and practices that have stood the test of time and recognition. Building on this theoretical effort, they describe project development in terms of states and alphas. It is essential for the project leaders and the project members to know, at every point in time, what is the current state of the project. This global state, however, is a combination of the states of many diverse components of the system; the term *alpha* covers such individual components. An alpha can be a software artifact, like the requirements or the code; a human element, like the project team; or a pure abstraction, like the opportunity that led to the idea of a project. Every alpha has, at a particular time, a state; combining all these alpha states defines the state of the project. Proper project management and success requires knowing this state at every stage of development.

The role of the kernel is to identify the key alphas of software development and, for each of them, to identify the standard states through which it can evolve. For example, an opportunity will progress through the states Identified, Solution Needed, Value Established, Viable, Addressed, and Benefits Accrued. Other alphas have similarly standardized sets of states.

The main value of this book is in the identification of these fundamental alphas and their states, enabling an engineering approach in which the project has a clear view of where it stands through a standardized set of controls.

The approach is open, since it does not prescribe any particular practice but instead makes it possible to integrate many different practices, which do not even have to come from the same methodological source—like some agile variant—but can combine good ideas from different sources. A number of case studies illustrate how to apply the ideas in practice.

Software practitioners and teachers of software engineering are in dire need of well-grounded methodological work. This book provides a solid basis for anyone interested in turning software project development into a solid discipline with a sound engineering basis.

—Bertrand Meyer
March 2012

Foreword
by Richard Soley

Software runs our world; *software-intensive systems,* as Grady Booch calls them, are the core structure that drives equity and derivative trading, communications, logistics, government services, management of great national and international military organizations, and medical systems—and even allows elementary school teacher Mr. Smith to send homework assignments to little Susie. Even mechanical systems have given way to software-driven systems (think of fly-by-wire aircraft, for example); the trend is not slowing, but accelerating. We depend on software, and often we depend on it for our very lives. Amazingly, more often than not software development resembles an artist's craft far more than an engineering discipline.

Did you ever wonder how the architects and builders of the great, ancient temples of Egypt or Greece knew how to build grand structures that would stand the test of time, surviving hundreds, or even thousands of years, through earthquakes, wars, and weather? The Egyptians had amazing mathematical abilities for their time, but triangulation was just about the top of their technical acumen. The reality, of course, is that luck has more to do with the survival of the great façade of the Celsus Library of Ephesus, in present-day Selçuk, Turkey, than any tremendous ability to understand construction for the ages.

This, of course, is no longer the case. Construction is now *civil engineering,* and civil engineering is an engineering discipline. No one would ever consider going back to the old hand-designed, hand-built, and far more dangerous structures of the distant past. Buildings still fail in the face of powerful weather phenomena, but not at anywhere near the rate they did 500 years ago.

What an odd dichotomy, then, that in the design of some large, complex systems we depend on a clear engineering methodology, but in the development of certain other large, complex systems we are quite content to depend on the ad hoc, handmade work of artisans. To be sure, that's not always the case; quite often, stricter processes and analytics are used to build software for software-intensive systems that "cannot" fail, where more time and money is available for their construction; aircraft avionics and other *embedded* systems design is often far more rigorous (and costly) than desktop computing software.

Really, this is more of a measure of the youth of the computing field than anything else, and the youth of our field is never more evident than in the lack of a grand unifying theory to underpin the software development process. How can we expect the computing field to have consistent software development processes, consistently taught at universities worldwide, consistently supported by software development organizations, and consistently delivered by software development teams, when we don't have a globally shared language that defines the software development process?

It is worth noting, however, that there is more than one way to build a building and more than one way to construct software. So the language or languages we need should define quarks and atoms instead of molecules—atomic and subatomic parts that we can mix and match to define, carry out, measure, and improve the software development process itself. We can expect the software development world to fight on about agile versus non-agile development, and traditional team-member programming versus pair programming, for years to come; but we should demand and expect that the process building blocks we choose can be consistently applied, matched, and compared as necessary, and measured for efficacy. That core process design language is called *Essence*. Note that, in fact, in this book there is a "kernel" of design primitives that are themselves defined in a common language; I will leave this complication for the authors to explain in detail.

In late 2009, Ivar Jacobson, Bertrand Meyer, and I came together to clarify the need for a widely accepted process design kernel and language and to build an international team to address that need. The three of us came from quite different backgrounds in the software world, but all of us have spent time in the trenches slinging code, all of us have led software development teams, and all of us have tried to address the software complexity problem in various ways. Our analogies have differed (operatic ones being quite noticeably Prof. Meyer's), our team leadership styles have differed, and our starting points have been quite visibly different. These differences, however, led to an outstanding international cooperation called Software Engineering Method and Theory, or SEMAT. The Essence kernel, a major Object Management Group (OMG) standards process, and this book are outputs of this cooperative project.

Around us a superb team of great thinkers formed, meeting for the first time at ETH in Zürich two years ago, with other meetings soon afterward. That team has struggled to bring together diverse experiences and worldviews into a core kernel composed of atomic parts that can be mixed and matched, connected as needed, drawn on a blueprint, analyzed, and put into practice to define, hire, direct, and measure real development teams. As I write this, the OMG is considering how to capture the work of this team as an international software development standard. It's an exciting time to be in the software world, as we transition from groups of artisans sometimes working together effectively, to engineers using well-defined, measured, and consistent construction practices to build software that works.

The software development industry needs and demands a core kernel and language for defining software development practices—practices that can be mixed and matched, brought on board from other organizations, measured, integrated, and compared and contrasted for speed, quality, and price. Soon we'll stop delivering software by hand; soon our great software edifices will stop falling down. SEMAT and Essence may not be the end of that journey to developing an engineering culture for software, and they certainly don't represent the first attempt to do so; but they stand a strong chance of delivering broad acceptance in the software world. This thoughtful book gives a good grounding in ways to think about the problem, and a language to address the need; every software *engineer* should read it.

—Richard Mark Soley, Ph.D.
38,000 feet over the Pacific Ocean
March 2012

Preface

Everyone who develops software knows that it is a complex and risky business, and is always on the lookout for new ideas that will help him or her develop better software. Luckily, software engineering is still a young and growing profession—one that sees new innovations and improvements in best practices every year. These new ideas are essential to the growth of our industry—just look at the improvements and benefits that lean and agile thinking have brought to software development teams.

Successful software development teams need to strike a balance between quickly delivering working software systems, satisfying their stakeholders, addressing their risks, and improving their way of working. For that, they need an effective thinking framework—one that bridges the gap between their current way of working and any new ideas they want to take on board. This book presents such a thinking framework in the form of an actionable kernel—something we believe will benefit any team wishing to balance their risks and improve their way of working.

INSPIRATION

This book was inspired by, and is a direct response to, the SEMAT Call for Action. It is, in its own way, one small step in the process to refound software engineering.

SEMAT (Software Engineering Method and Theory) was founded in September 2009 by Ivar Jacobson, Bertrand Meyer, and Richard Soley, who felt the time had come to fundamentally change the way people work with software development methods. Together they wrote a call for action, which in a few lines

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

- The prevalence of fads more typical of a 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

We support a process to refound 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

Figure P-1 Excerpt from the SEMAT Call for Action

identifies a number of critical problems with current software engineering practice, explains why there is a need to act, and suggests what needs to be done. Figure P-1 is an excerpt from the SEMAT Call for Action.

The call for action received a broad base of support, including a growing list of signatories and supporters.[1] The call for action's assertion that the software industry is prone to fads and fashions has led some people to assume that SEMAT and its supporters are resistant to new ideas. This could not be further from the truth. As you will see in this book, they are very keen on new ideas—in fact, this book is all about some of the new ideas coming from SEMAT itself. What SEMAT and its supporters are against is the non-lean, non-agile behavior that comes from

1. The current list can be found at www.semat.org.

people adopting inappropriate solutions just because they believe these solutions are fashionable, or because of peer pressure or political correctness.

In February 2010 the founders developed the call for action into a vision statement.[2] In accordance with this vision SEMAT then focused on two major goals:

1. Finding a kernel of widely agreed-on elements
2. Defining a solid theoretical basis

To a large extent these two tasks are independent of each other. Finding the kernel and its elements is a pragmatic exercise requiring people with long experience in software development and knowledge of many of the existing methods. Defining the theoretical basis requires academic research and may take many years to reach a successful outcome.

THE POWER OF THE COMMON GROUND

SEMAT's first step was to identify a common ground for software engineering. This common ground is manifested as a kernel of essential elements that are universal to all software development efforts, and a simple language for describing methods and practices. This book provides an introduction to the SEMAT kernel, and how to use it when developing software and communicating between teams and team members. It is a book for software professionals, not methodologists. It will make use of the language but will not dwell on it or describe it in detail.

The kernel was first published in the SEMAT OMG Submission.[3] As shown in Figures P-2 and P-3, the kernel contains a

2. The SEMAT Vision statement can be found at the SEMAT website, www.semat.org.

3. "Essence – Kernel and Language for Software Engineering Methods." Available from www.semat.org.

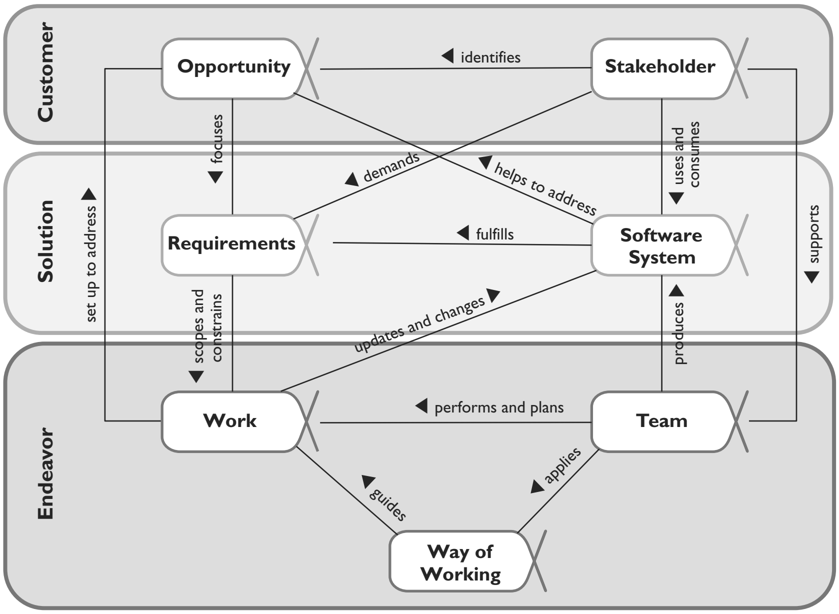


Figure P-2 Things to work with

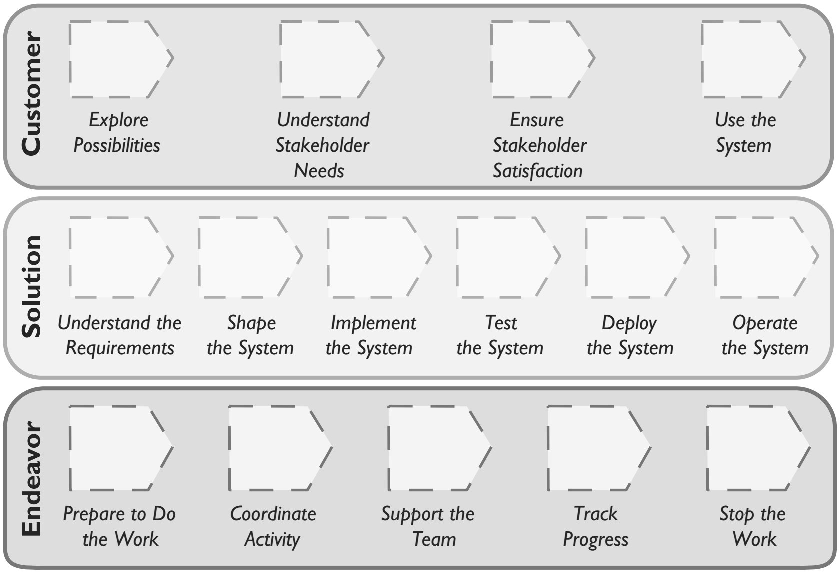


Figure P-3 Things to do

small number of “things we always work with” and “things we always do” when developing software systems. There is also work that is ongoing, with the goal of defining the “skills we always need to have,” but this will have to wait until future versions of the kernel and is outside the scope of this book.[4]

We won’t delve into the details of the kernel here as this is the subject of Part I, but it is worth taking a few moments to think about why it is so important to establish the common ground in this way. 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
- A common ground for improved communication, standardized measurement, and the sharing of best practices
- A foundation for accessible, interoperable method and practice definitions
- And most importantly, a way to help teams understand where they are and what they should do next

THE BIG IDEA

What makes the kernel anything more than just a conceptual model of software engineering? What is really new here? This can be summarized into the three guiding principles shown in Figure P-4.

4. A kernel with similar properties as the SEMAT kernel was first developed at Ivar Jacobson International in 2006 (www.ivarjacobson.com). This kernel has served as an inspiration and an experience base for the work on the SEMAT kernel.

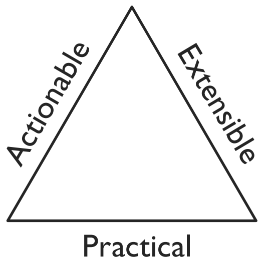


Figure P-4 Guiding principles of the kernel

The Kernel Is Actionable

A unique feature of the kernel is how 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 software engineering endeavor, one that is relevant to an assessment of its progress and health. As shown in Figure P-2, SEMAT has identified seven alphas: Opportunity, Stakeholders, Requirements, Software System, Work, Way of Working, and Team. The alphas are characterized by a simple set of states that represent their progress and health. As an example, the Software System moves through the states of Architecture Selected, Demonstrable, Usable, Ready, Operational, and Retired. Each state has a checklist that 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 software development teams.

The kernel presents software development 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 software development efforts, regardless of the practices applied or the software development 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 state 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.

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, offshore, software product lines, etc.). The kernel allows you to add practices, such as user stories, use cases, component-based development, architecture, pair programming, daily stand-up meetings, self-organizing teams, and so on, 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 development as a soup of indistinguishable practices and lead teams to dump the good with the bad when they move from one method to another.

The Kernel Is Practical

Perhaps the most important feature of the kernel is the way it is used in practice. Traditional approaches to software development methods tend to focus on supporting 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 can be touched and used through the use of cards (see Figure P-5). 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 uses on a daily basis. This is a fundamental difference from

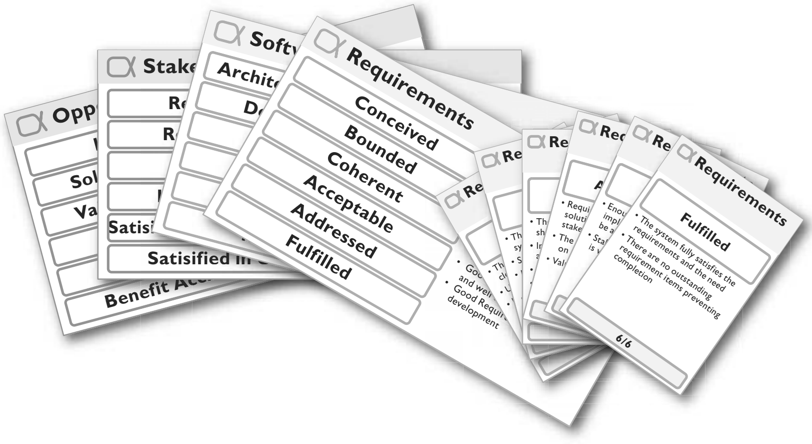

Requirements
Conceived
Bounded
Coherent
Acceptable
Addressed
Fulfilled
Requirements
Fulfilled
The system fully satisfies the requirements and the need
There are no outstanding requirement items preventing completion
6/6
Satisified in Use
Benefit Accrued

Figure P-5 Cards make the kernel tangible.

traditional approaches, which tend to overemphasize method description as opposed to method use and tend to only be consulted by people new to the team.

THE KERNEL IN ACTION

Although the ideas in this book will be new to many of you, they have already been successfully applied in both industry and academia.

Early adopters of the kernel idea[5] include the following.

- MunichRe, the world's leading reinsurance company, where a family of "collaboration models" have been assembled to cover the whole spectrum of software and application work. Four collaboration models have been built on the same kernel from the same set of 12 practices. The models are Exploratory, Standard, Maintenance, and Support.
- Fujitsu Services, where the Apt Toolkit has been built on top of an early version of the software engineering kernel, including both agile and waterfall ways of working.
- A major Japanese consumer electronics company, whose software processes have been defined on top of an early version of the kernel, allowing the company to help teams apply new practices and manage their offshore development vendor.
- KPN, where a kernel-based process was adopted by more than 300 projects across 13 programs as part of a move to iterative development. The kernel also provided the basis for a new result-focused QA process, which could be applied to all projects regardless of the method or practices used.

5. In all cases they used the kernel and practices developed by Ivar Jacobson International.

- A major UK government department, where a kernel-based agile toolset was introduced to enable disciplined agility and the tracking of project progress and health in a practice-independent fashion.

The kernel is already being used in first- and second-year software engineering courses at KTH Royal Institute of Technology in Sweden.

- The first-year courses were run by Anders Sjögren. After the students conducted their projects, Anders and the students went through the SEMAT alphas and matched them to their project results. Here, the students had the opportunity to acquaint themselves with and evaluate the alphas as well as gain insight into the project's progress and health.
- The second-year courses were run by Mira Kajko-Mattsson. Here, the students were requested to actively use the SEMAT kernel when running their projects along with the development method they followed. Mira created an example software development scenario and evaluated the scenario for each alpha, its states, and the state checklist items. The students were then requested to do the same when conducting and evaluating their projects.

The courses taught the students the following lessons.

- The kernel assures that all the essential aspects of software engineering are considered in a project. By matching the project results against the kernel alphas, the students can easily identify the good and bad sides of their development methods.

- The kernel prepares students for future software engineering endeavors with minimal teaching effort. Because they had to follow all the kernel alphas, the students could learn the total scope of the software engineering endeavor and thereby know what will be required of them in their professional careers.

HOW DOES THE KERNEL RELATE TO AGILE AND OTHER EXISTING APPROACHES?

The kernel can be used with all the currently popular management and technical practices, including Scrum, Kanban, risk-driven iterative, waterfall, use-case-driven development, acceptance-test-driven development, continuous integration, test-driven development, and so on. It will help teams embarking on the development of new and innovative software products and teams involved in enhancing and maintaining mature and established software products. It will help teams of all sizes from one-man bands to thousand-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 software development teams, it values teams working and fulfilling team responsibilities over the following 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 this book, 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 to 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 normal in the past).

The kernel doesn't just support modern best practices. It also recognizes that a vast amount of software is already developed and needs to be maintained; it 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 while maintaining and improving the team's ability to deliver.

HOW THE KERNEL WILL HELP YOU

Use of the kernel has many benefits for you as an experienced or aspiring software professional, and for the teams you work in. For example, it provides guidance to help you assess the progress and health of your software 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. And it will help the industry as a whole by improving interoperability between teams, suppliers, and development organizations.

By providing a practice-independent foundation for the definition of software methods, the kernel also has the power to completely transform the way 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.

1. Teams are no longer trapped by their methods. They can continuously improve their way of working by adding or removing practices as and when their situation demands.
2. Methodologists no longer need to waste their time describing complete methods. They can easily describe their new ideas in a concise and reusable way.

Finally, there are also benefits for academia, particularly in the areas of education and research. The kernel will provide a basis for the creation of foundation courses in software engineering, ones that can then be complemented with additional courses in specific practices—either as part of the initial educational curriculum or later during the student's further professional development. Equally as important is the kernel's ability to act as a shared reference model and enabler for further research and experimentation

HOW TO READ THIS BOOK

This book is intended for anyone who wants to have a clear frame of reference when developing software, researching software development, or sharing software development experiences.

For software professionals the goal of this book is to show how the kernel can help solve challenges you face every day when doing your job. It demonstrates how the kernel is used in different situations from small-scale development to large-scale development.

For students and other aspiring software professionals, the goal of the book is to illustrate some of the challenges software professionals face and how to deal with them. It will provide you with a firm foundation for further study and help you learn what you otherwise only learn through experience.

The book is organized to allow gradual learning, and concepts are introduced and illustrated incrementally. We hope this book will be useful to software professionals, educators, and students, and we look forward to your feedback.

The book is structured into seven short parts.

Part I: The Kernel Idea Explained

An overview of the kernel with examples of how it can be used in practice.

Part II: Using the Kernel to Run an Iteration

A walkthrough of how the kernel can be used to run an iteration.

Part III: Using the Kernel to Run a Software Endeavor

A description of how you can use the kernel to run a complete software endeavor—for example, a project of some size—from idea to production.

Part IV: Scaling Development with the Kernel

A demonstration of how the kernel is flexible in supporting different practices, organizations, and domains.

Part V: How the Kernel Changes the Way You Work with Methods

Takes a step back and discusses the principles for you to apply the kernel effectively and successfully to your specific situation.

Part VI: What's Really New Here?

A summary of the highlights and key differentiators of SEMAT and this book.

Part VII: Epilogue

A forward-looking discussion of how we can get even more value from the kernel in the future.

FURTHER READING

Jacobson, I., and B. Meyer. 2009. Methods need theory. *Dr. Dobb's Journal.*

Jacobson I., and I. Spence. 2009. Why we need a theory for software engineering. Dr. Dobb's Journal.

Jacobson I., B. Meyer, and R. Soley. 2009. Call for Action: The Semat Initiative. Dr. Dobb's Journal.

Jacobson I., B. Meyer, and R. Soley. 2009. The Semat Vision Statement.

Fujitsu, Ivar Jacobson International AB, Model Driven Solutions. 2012. Essence – Kernel and Language for Software Engineering Methods. Initial Submission – Version 1.0.

OMG. 2012. Request for Proposal (RFP). A Foundation for the Agile Creation and Enactment of Software Engineering Methods. OMG.

Jacobson I., P.W. Ng, and I. Spence. 2007. Enough of Processes: Let's Do Practices. *Journal of Object Technology* 6(6):41–67.

Ng P.W., and M. Magee. Light Weight Application Lifecycle Management Using State-Cards. *Agile Journal*, October 10, 2010.

Azoff, M. EssWork 3.0 and Essential Practices 4.0. Ivar Jacobson International. OVUM Technology Report, Reference Code TA001906ADT. April 2012.

Azoff, M. Apt Methods and Tools. Fujitsu. OVUM Technology Report, Reference Code O100032-002. January 2011.

This page intentionally left blank

Acknowledgments

We wish to acknowledge and thank the contributors to SEMAT, in particular (alphabetically): Jakob Axelsson, Stefan Bylund, Bob Corrick, Michael Goedicke, Lulu He, Shihong Huang, Carlos Mario Zapata Jaramillo, Mira Kajko-Mattsson, Philippe Kruchten, Bruce Macisaac, Winifred Menezes, Richard Murphy, Hanna Oktaba, Roland Racko, Ed Seidewitz, and Michael Strieve.

This page intentionally left blank

8 Planning an Iteration

The art of planning an iteration is in deciding which of the many things the team has to do should be done in this iteration—the next two to four weeks. Every iteration will produce working software, but there are other things the team needs to think about. They need to make sure they develop the right software in the best way they can. The kernel helps the team reason about the current development context, and what to emphasize next, to make sure a good balance is achieved across the different dimensions of software development.

You can think of planning an iteration as follows.

1. *Determine where you are.* Work out the current state of the endeavor.
2. *Determine where to go.* Decide what to emphasize next, and what the objectives of the next iteration will be.
3. *Determine how to get there.* Agree on the tasks the team needs to do to achieve the objectives.

In our story, because of the way the team chose to run their iterations, the iteration objectives were put into the team's iteration backlog and broken down into more detailed tasks. In this way the iteration backlog served as the team's to-do list. We will now look at how Smith and his team used the alphas to guide the planning and execution of an iteration.

8.1 PLANNING GUIDED BY ALPHA STATES

When you plan an iteration the alphas can help you understand where you are and where to go next. By aligning the objectives they set for each iteration, Smith's team made sure they progressed in a balanced and cohesive way. This relationship between the alphas, and the objectives and tasks in the iteration backlog, is illustrated in Figure 8-1.

8.1.1 Determine Where You Are

When preparing for an iteration, the first step is to understand where you are. This involves, among other things, understanding details relating to technology, risks, quality, and stakeholder needs. But it is also important to have a shared understanding of where you are with the software endeavor as a whole, and this is where the kernel can help.

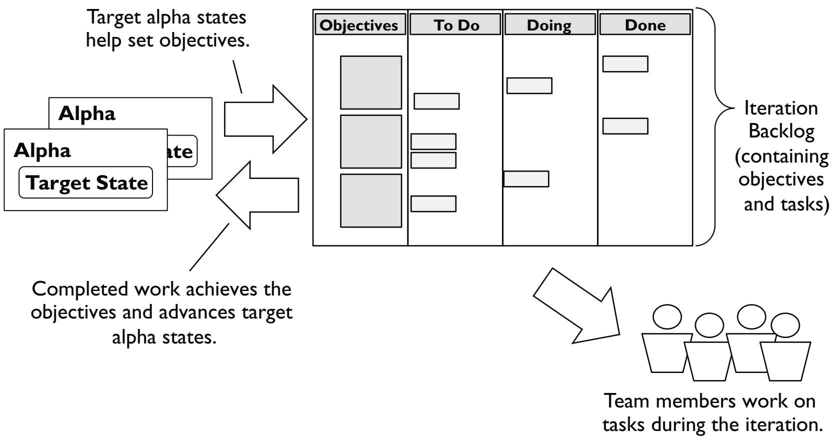

Target alpha states help set objectives.
Alpha
Alpha
ate
Target State
Objectives
To Do
Doing
Done
Iteration Backlog (containing objectives and tasks)
Completed work achieves the objectives and advances target alpha states.
Team members work on tasks during the iteration.

Figure 8-1 Working from the tasks and objectives in an iteration backlog

There are a number of ways you can use the kernel to do this. If you are using alpha state cards, as discussed in Part I, you can do this as follows.

- **Walkthrough:** This is a simple approach using one set of cards.
  1. Lay out the cards for each alpha in a row on a table with the first state on the left and the final state on the right.
  2. Walk through each state and ask your team if you have achieved that state.
  3. If the state is achieved, move that state card to the left. Continue with the next state card until you get to the state that your team has not yet achieved.
  4. Move this state card and the rest of the pending state cards to the right.
- **Poker:** Another approach that sometimes works better is poker.
  1. Each member is given a deck of state cards.
  2. For each alpha, each member selects the state card that he or she thinks best represents the current state of the software development endeavor.
  3. All members put their selected state card face down on the table.
  4. When all are ready, they turn the state card face up.
  5. If all members have selected the same state card, then there is consensus.
  6. If the selected state cards are different, it is likely there are different interpretations of the checklists for the states. The team can then discuss the checklists for the state to reach an agreement.

Using state cards is not required to use the kernel, but they are a useful tool to get the team members to talk, and to discuss what state the endeavor is in and what state they need to focus on next.

Once you have determined the current state of the endeavor, you can start discussing what the next set of states to be achieved should be.

8.1.2 Determine Where to Go

Identifying a set of desired alpha states guides the team in determining what to emphasize in an iteration. In fact, the iteration objective can be described as reaching a set of target alpha states.

Once the team has determined the current state of their alphas ,it is fairly easy to select which of the next states they should target in their next iteration. The target states make well-formed objectives as their checklists provide clearly defined completion criteria.

Selecting the target states can easily be done as an extension to the walkthrough and poker techniques described in the preceding section.

8.1.3 Determine How to Get There

After identifying a candidate set of objectives for the iteration, the team has to decide how they will address them and whether or not they can achieve them in the iteration timebox. Typically this is done by identifying one or more tasks to be completed to achieve the objective.

Again the alpha states help the team with the checklist for each state providing hints as to what tasks they will need to do to achieve the objective. In this part of the book we are just considering a small software endeavor. Later in the book we will discuss how you identify tasks and measure progress on more complex efforts.

8.2 DETERMINING THE CURRENT STATE IN OUR STORY

Smith and his team were six weeks into development. They had provided an early demonstration of the system to their stakeholders. Angela and the other stakeholders were pleased with what they saw, and they gave valuable feedback. However, the system was not yet usable by end users.

Smith started the iteration planning session with a walkthrough to determine the current state. Figure 8-2 shows the states they had achieved on the left, and the states not yet achieved on the right.

Table 8-1 shows the current states for the alphas and describes how the team in our story achieved them.

8.3 DETERMINING THE NEXT STATE IN OUR STORY

Once the team had agreed on the current alpha states, the team discussed what the next desired "target" states were to guide its planning. The team agreed to use the immediate next alpha states to help establish the objectives of the next iteration. These are shown in Figure 8-3.

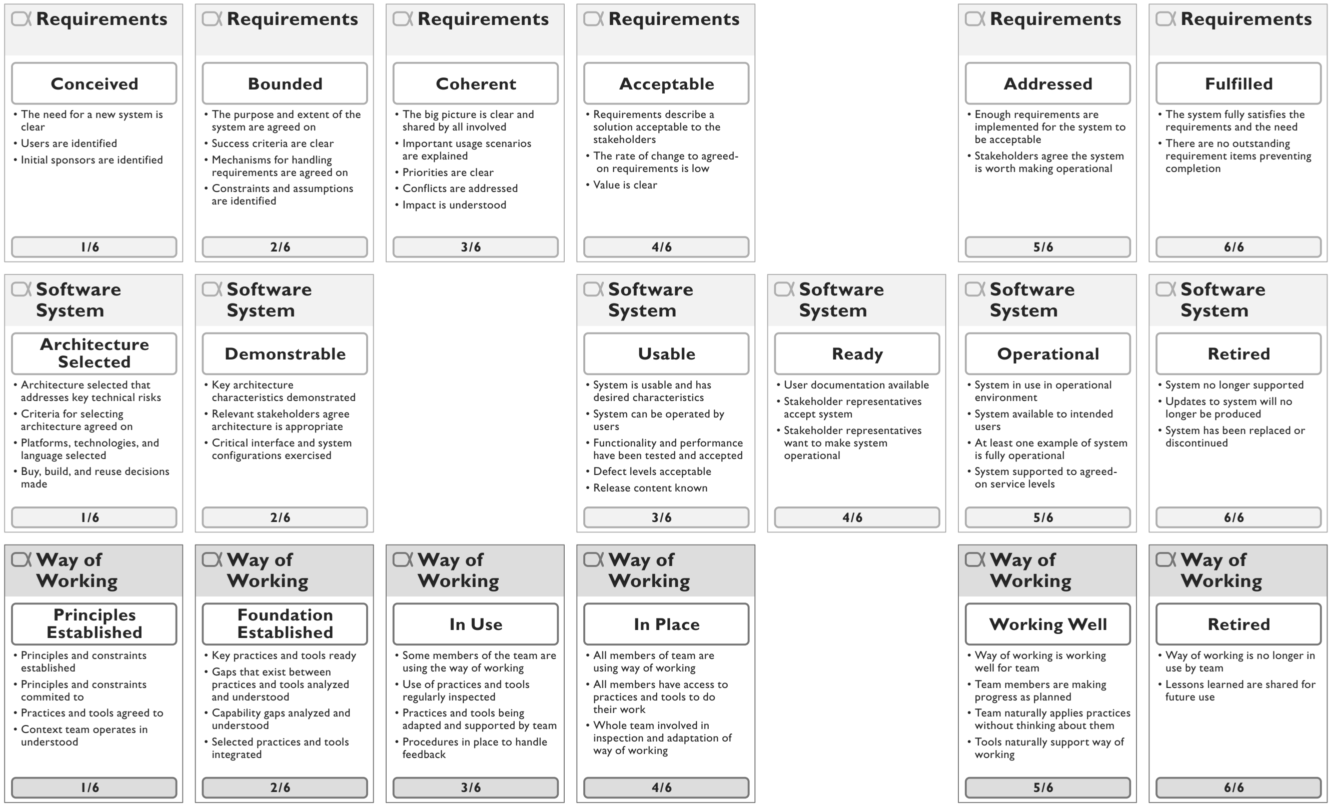
In most cases, the name of the alpha state itself provides sufficient information to understand the state. But if needed, team members can find out more by reading the alpha state checklist. By going through the states one by one for each alpha, a team quickly gets familiar with what is required to achieve each state. In this way the team learns about the kernel alphas at the same time as they determine their current state of development and their next target states.

8.4 DETERMINING HOW TO ACHIEVE THE NEXT STATES IN OUR STORY

Smith and his team looked at the next target states and agreed that some prioritization was needed. In this case, they needed

Table 8-1 How the Team Achieved the Current State of Development

Current State	How It Was Achieved
Requirements **Acceptable** • Requirements describe a solution acceptable to the stakeholders • The rate of change to agreed-on requirements is low • Value is clear **4/6**	Smith's team had demonstrated an early version of the application based on an initial set of requirements. After the demonstration, the stakeholders agreed that the understanding of the requirements was acceptable. The agreed-on requirement items were online and offline browsing of the social network, and making posts offline. However, these requirement items were only partially implemented at the time of the demonstration. According to the state definition, our team has achieved the Requirements: Acceptable state.
Software System **Demonstrable** • Key architecture characteristics demonstrated • Relevant stakeholders agree architecture is appropriate • Critical interface and system configurations exercised **2/6**	Early during development, Smith's team had identified the critical technical issues for the software system and outlined the architecture. This had allowed them to achieve the Software System: Architecture Selected state. Moreover, Smith's team had demonstrated an early version of the system to their stakeholders. This means that Smith's team had achieved the Software System: Demonstrable state. However, since Smith's team had not completed enough functionality to allow users to employ the system on their own, Smith's team had not yet achieved the Software System: Usable state.
Way of Working **In Place** • All members of team are using way of working • All members have access to practices and tools to do their work • Whole team involved in inspection and adaptation of way of working **4/6**	The two new members, Dick and Harriet, who had just come on board were not fully productive yet. In particular, they seemed to have trouble with the approach to automated testing, which the team agreed was important to maintain high quality during development. They had difficulty identifying good test cases and writing good test code. As such, the team agreed that the Way of Working is currently in the In Place state. But they had not yet achieved the Working Well state.



Requirements — Conceived
- The need for a new system is clear
- Users are identified
- Initial sponsors are identified

1/6

Requirements — Bounded
- The purpose and extent of the system are agreed on
- Success criteria are clear
- Mechanisms for handling requirements are agreed on
- Constraints and assumptions are identified

2/6

Requirements — Coherent
- The big picture is clear and shared by all involved
- Important usage scenarios are explained
- Priorities are clear
- Conflicts are addressed
- Impact is understood

3/6

Requirements — Acceptable
- Requirements describe a solution acceptable to the stakeholders
- The rate of change to agreed-on requirements is low
- Value is clear

4/6

Requirements — Addressed
- Enough requirements are implemented for the system to be acceptable
- Stakeholders agree the system is worth making operational

5/6

Requirements — Fulfilled
- The system fully satisfies the requirements and the need
- There are no outstanding requirement items preventing completion

6/6

Software System — Architecture Selected
- Architecture selected that addresses key technical risks
- Criteria for selecting architecture agreed on
- Platforms, technologies, and language selected
- Buy, build, and reuse decisions made

1/6

Software System — Demonstrable
- Key architecture characteristics demonstrated
- Relevant stakeholders agree architecture is appropriate
- Critical interface and system configurations exercised

2/6

Software System — Usable
- System is usable and has desired characteristics
- System can be operated by users
- Functionality and performance have been tested and accepted
- Defect levels acceptable
- Release content known

3/6

Software System — Ready
- User documentation available
- Stakeholder representatives accept system
- Stakeholder representatives want to make system operational

4/6

Software System — Operational
- System in use in operational environment
- System available to intended users
- At least one example of system is fully operational
- System supported to agreed-on service levels

5/6

Software System — Retired
- System no longer supported
- Updates to system will no longer be produced
- System has been replaced or discontinued

6/6

Way of Working — Principles Established
- Principles and constraints established
- Principles and constraints commited to
- Practices and tools agreed to
- Context team operates in understood

1/6

Way of Working — Foundation Established
- Key practices and tools ready
- Gaps that exist between practices and tools analyzed and understood
- Capability gaps analyzed and understood
- Selected practices and tools integrated

2/6

Way of Working — In Use
- Some members of the team are using the way of working
- Use of practices and tools regularly inspected
- Practices and tools being adapted and supported by team
- Procedures in place to handle feedback

3/6

Way of Working — In Place
- All members of team are using way of working
- All members have access to practices and tools to do their work
- Whole team involved in inspection and adaptation of way of working

4/6

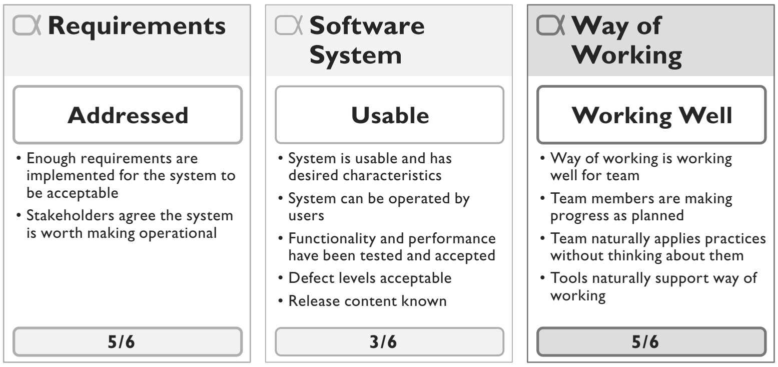
Way of Working — Working Well
- Way of working is working well for team
- Team members are making progress as planned
- Team naturally applies practices without thinking about them
- Tools naturally support way of working

5/6

Way of Working — Retired
- Way of working is no longer in use by team
- Lessons learned are shared for future use

6/6

Figure 8-2 The team uses the alphas to determine the current states.



Requirements — **Addressed**
- Enough requirements are implemented for the system to be acceptable
- Stakeholders agree the system is worth making operational

5/6

Software System — **Usable**
- System is usable and has desired characteristics
- System can be operated by users
- Functionality and performance have been tested and accepted
- Defect levels acceptable
- Release content known

3/6

Way of Working — **Working Well**
- Way of working is working well for team
- Team members are making progress as planned
- Team naturally applies practices without thinking about them
- Tools naturally support way of working

5/6

Figure 8-3 The selected next states

to first get to the Way of Working: Working Well state, then the Software System: Usable state, and finally the Requirements: Addressed state. The reason was simple: If their way of working did not work well, this would impede their attempts to get the software system usable. In addition, they agreed on the priority for the missing requirement items necessary to achieve the Requirements: Addressed state.

Smith and his team next discussed what needed to be done to achieve these states (see Table 8-2).

Table 8-2 How the Team Planned to Achieve the Selected Target States

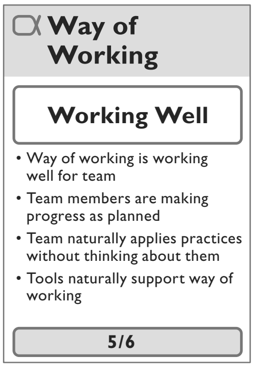
Target State	How They Planned to Achieve Them
 **Way of Working** — **Working Well** • Way of working is working well for team • Team members are making progress as planned • Team naturally applies practices without thinking about them • Tools naturally support way of working 5/6	Both Dick and Harriet agreed that they had difficulties in applying automated testing. They needed help in order to make progress. Tom agreed that he had to spend time teaching them. A task was added to the iteration backlog for Tom to conduct training on automated testing for Dick and Harriet.

Table 8-2 How the Team Planned to Achieve the Selected Target States (*continued*)

Software System **Usable** • System is usable and has desired characteristics • System can be operated by users • Functionality and performance have been tested and accepted • Defect levels acceptable • Release content known **3/6**	This state reminds us that the software system must be shown to be of sufficient quality and functionality to be useful to the users. So far, Smith's team had been testing within its development environment. Now it had to conduct tests within an acceptance test environment, which they had yet to prepare. This resulted in the following task: Task 2. Prepare acceptance test environment. Smith's team had to bring all requirement items currently demonstrable in the system to completion. By "complete" they meant that each requirement item must be fully tested within the acceptance test environment. Task 3. Complete requirement item A: "Browse online and offline". Task 4. Complete requirement item B: "Post comment (online and offline)". Task 5. Complete requirement item C: "Browse album".
Requirements **Addressed** • Enough requirements are implemented for the system to be acceptable • Stakeholders agree the system is worth making operational **5/6**	This state reminds us of the need to work with stakeholders to ensure that they are happy with the system produced. In our story Smith had to work with Angela to determine which additional requirement items needed to be implemented. This resulted in the following additional task: Task 6: Talk to Angela and agree on additional requirement items, fitting in the iteration, to make the system worth being operational.

By going through the target alpha states, Smith was able to determine a set of objectives and tasks for the next iteration.

8.5 HOW THE KERNEL HELPS YOU IN PLANNING ITERATIONS

A good plan must be inclusive, meaning that it includes all essential items and covers the whole team. It must also be concrete, so it is actionable for the team. The team must also have a way to monitor its progress against the plan. The kernel helps you achieve this as follows.

- *Inclusive:* The kernel alphas serve as reminders across the different dimensions of software development, helping you to create a plan that addresses all dimensions in a balanced way.
- *Concrete:* The checklists for each alpha state give you hints as to what you need to do in the iteration. The same checklists help you determine your progress by making clear what you have done and comparing this to what you intended to do.

Index