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

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.

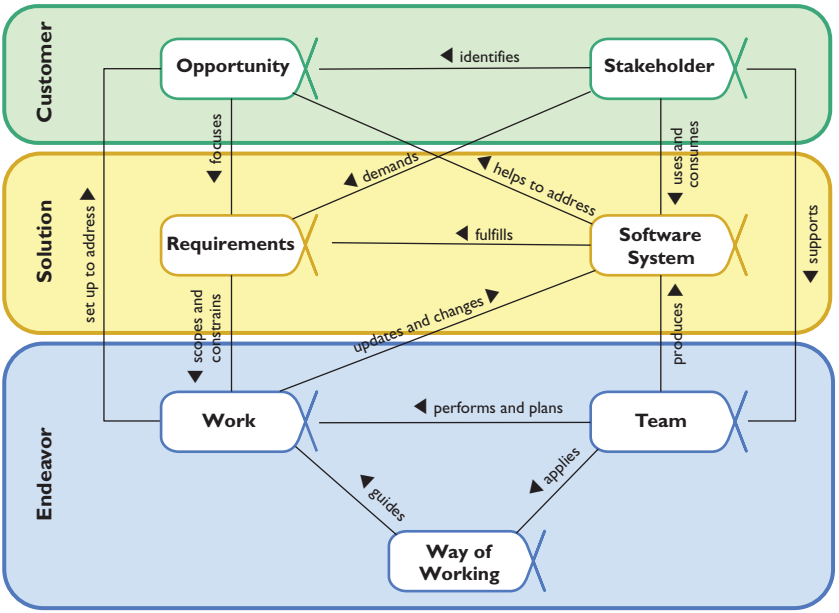

Figure P-2 Things to work with

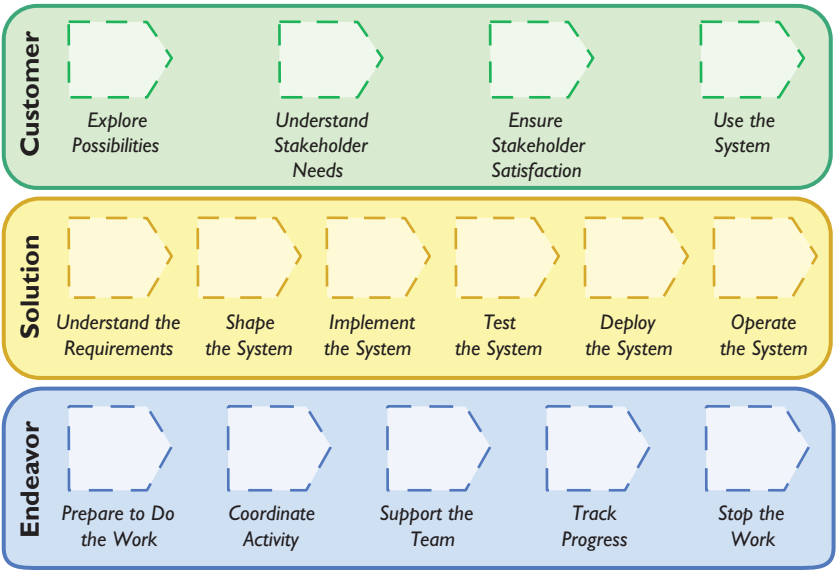

Figure P-3 Things to do

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.

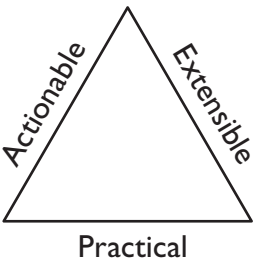

Figure P-4 Guiding principles of the kernel

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

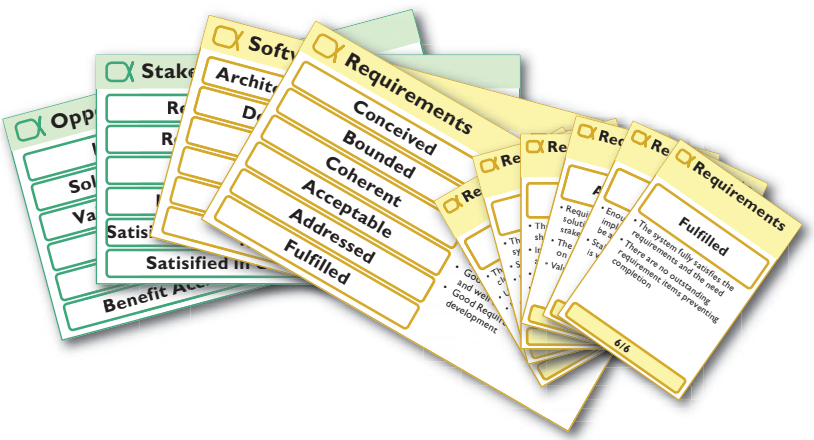

Figure P-5 Cards make the kernel tangible.