The Essence
of Software Engineering

Applying the Semat Kernel

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

Contributors (alphabetically):

Jakob Axelsson, Stefan Bylund, Bob Corrick, Michael Goedicke, Shihong Huang, He Lulu, Bruce Macissac, Mira Kajko-Mattsson, Winifred Menezes, Richard Murphy, Hanna Oktaba, Roland Racko, Ed Seidewitz, Michael Streive, Carlos Mario Zapata Jaramillo.

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

Foreword by Robert Martin

The pendulum has swung again. This time it has swung towards 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, it therefore can not 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 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, that 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 mechanisms as opposed to an assembly line is driven by the Agile worldview. Agile projects do not progress through phases. Rather they operate like that pump, continuously transforming 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 action 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 someone 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; 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 I could look through them and play with them.

Robert Martin

(unclebob)

Foreword by Richard Soley

Once upon a time, a trade was a trade, and settlement was settlement. When J. P. Morgan & Co. sold 1,000 shares of General Electric at 8½ to Merrill Lynch & Co., the requisite $8,500 was eventually carried down Wall Street from Mr. Lynch's company to Mr. Morgan's company, and men and boys eventually carried the GE stock certificates the other way up the street to be deposited in Mr. Morgan's vaults. Pieces of paper representing gold bars, safely stored at the Federal Reserve Bank (or perhaps Fort Knox) went one way while pieces of paper giving voting rights in Thomas Edison's great enterprise went the other, carried by hand and often protected by men with guns. No longer; these days, of course, computers at JPMorganChase have a little chat with computers at Bank of America Merrill Lynch and settle their scores with a few bits flying back & forth cyberspace.

Cloud computing and replicated, worldwide data centers mean that those computers might not even be (physically) at either company, and nobody needs guns to protect those bits that represent the old pieces of paper (that in turn represent voting shares and some understanding of purchasing power, though no longer gold bars). Software runs our world; software-intensive systems, as Grady Booch calls them, are the core structure that not only drives equity & derivative trading, but communications, logistics, government services, management of great national & international military organizations, medical systems, and even allow elementary-school

teacher Mr. Smith to send homework assignments to little Susy (and receive her hard work in return, and perhaps even check it automatically). 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.

Once upon a time, building was building. Did you ever wonder, visiting the great ancient temples of Egypt or Greece, or even the far more modern (but still old) churches of Europe, how those architects & builders of yore knew how to build grand structures that would stand the test of time, would stand hundreds or even thousands of years, through earthquakes, wars & weather? The Egyptians had amazing mathematical abilities for their time, but triangulation was just about the top of their technical acumen, the ability to do stress analysis was a little off the charts at the time. 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 modern 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. Great structures are carefully planned, and those plans use standardized design languages to capture and share structural, esthetic, process and cost models between architects, developers, builders, electricians, masons, steel-workers, glaziers, plumbers and

myriad other trades that have to do with selling, buying, building and inspecting edifices. Those abstract designs, those blueprints, are studied thousands of different ways to understand the structure in detail long before a hole is dug or a brick is laid. Blueprints are analyzed to understand cost, to understand time (to finish the construction, for example), to understand structural integrity (and correct it for local conditions of soil, wind & climate), and even to understand how to support future maintenance and integration of the building into the local environment and future potential uses. Buildings, especially large buildings, are large and complex devices that are costly to maintain, and the more that can be understood from the design (and tuned to lower cost or increase usability), the better. The building trades have been using standardized, well-understood and long-taught design methodologies for several hundred years, and 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 anything like the rate of 500 years ago.

What an odd dichotomy, then, that in the design of some large, complex systems we depend on a clear engineering methodology, taught for centuries in universities and with expectations that can be directly measured (and whose risks can therefore be calculated and covered by liability insurance); but in the development of certain other large, complex systems we are quite content to depend on an the ad hoc, handmade work of artisans to build our systems. To be sure, that's not always the case; quite often, more strict processes & analytics are used to build software for software-

intensive systems that "cannot" fail, where more time & money is available for their construction; aircraft avionics and other embedded systems design is often far more rigorous (and costly) than desktop computing software. When it comes to our everyday word processing needs, however, we don't mind buggy software, as long as it's cheap. Well, perhaps we mind it, but we are quite willing to put up with it.

Really, this is more of a measure of the youth of the computing field more 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, consistent taught at universities worldwide, consistently supported by software development organizations, consistently delivered by software development teams, when we don't have a globally shared abstract language for the blueprints that define the software development process? The end of the 20th Century did deliver to the software world the first global languages for capturing the design of software itself; the Unified Modeling Language (UML) is quite broadly used today to specify the bricks & mortar that make up software edifices. What we need next is a consistent, well-defined and broadly-adopted language to specify the software development process itself.

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 & atoms instead of molecules, atomic and subatomic parts that we can mix

& 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 development vs. non-agile, and traditional team-member programming vs. pair programming, for years to come; but we should demand and expect that the process building blocks we choose can be consistently applied, matched as necessary, and measured for efficacy. That core process design language we call SEMAT, for Software Engineering Methods & Tools.

In late 2009, Ivar Jacobson, Bertrand Meyer and I came together to clarify the need for a process design language and 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.

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 afterwards near Washington, near St. Petersburg and in Milan. That team has struggled to bring together diverse experiences and worldviews into a core language composed of atomic parts that can be mixed & matched, connected as needed, drawn on a blueprint, analyzed and put into practice to define, hire, direct & measure real development teams. As I write this, the

Object Management Group 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 & demands a core language for defining software development practices, practices that can be mixed & matched, brought on board from other organizations, practices that can be measured, practices that can be integrated, practices that can be compared & contrasted for speed, quality & price. Soon we'll stop delivering software by hand (just as the great trading companies no longer deliver shares by hand); soon our great software edifices will stop falling down. SEMAT may not be the end of that journey to developing an engineering culture for software, and it certainly isn't the first attempt to do so; but it stands 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, and every software engineer should read it.

Richard Mark Soley, Ph.D.

38,000 feet over the Pacific Ocean

March 2012

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 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 the 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 the present 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 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

Preface

What this book is about

This book is about how software development fundamentally is done. It is about finding the common ground for all software development endeavors – or as we call it, the kernel. The book deals with fundamentals, but its value reaches beyond fundamentals as the kernel helps you to reason about real-life situations in a practical and lightweight way. To show how, the book includes walkthroughs of how the kernel helps developers to overcome everyday challenges and thereby makes a difference. By developers we mean anyone who plays a part in a software development endeavor, including but not limited to programmers, testers, analysts, architects and leaders.

There are many things to think of when engaged in an anything but a trivial software development endeavor, including selecting technologies, being proficient in these technologies, recruiting and developing good people, knowing what to work on any one time, how to collaborate, how to assess progress etc. The list could go on for a long time. This is what software engineering is supposed to help you with. However, due to decades of heavy-weight processes describing static workflows, mandating heavy-weight documentation, and enforcing detailed work instructions, the reputation of software engineering is doubtful today. We hope to change this with this book by describing a new foundation for software engineering that is lightweight, practical and directly applicable across many domains. If asked for a definition of what software engineering is, we think it suffices to say that software engineering is the discipline of making software systems deliver the

required value to all stakeholders.[1]

The essentials for software engineering described in this book, the kernel, has been developed by Semat a community of volunteers with a broad range of backgrounds and experiences from around the world.

Why you should read this book

This isn't another methodology book, improvement framework, or software standard. Neither is it a book about architecting, programming, requirements or testing. However, the book will allow you to improve communication, and accelerate and amplify learning across these and all other software development disciplines by providing the common essentials for reasoning, sharing experiences and learning about software development.

While the kernel may seem obvious to some readers, many software endeavors fail and more often than not the cause comes back to insufficient focus on essentials. We need a better way to bring the essentials to light when they are needed most. We hope that the kernel will provide that better way for you.

What inspired us to write this book: Semat

The world today runs on software and software has advanced our world greatly. Given the importance of software, it is not surprising that today in 2011[2], there are more than 16 million

[1] Definition of software engineering from Tom Gilb, 2003
[2] http://www.prweb.com/releases/2011/9/prweb8834228.htm

developers in the world and there will probably be more than 20 million by 2015. That is a lot of people and teams building software! It is no surprise that there are lots of research projects, innovations and conferences dedicated to getting teams to become better in how they develop software. So, we talk about things like object orientation, service oriented architecture, agile development, lean development to name a few. Yet, despite all these innovations, the software industry is slow at learning how to avoid common mistakes and how to address common challenges faced by developers. What is needed, we believe, is what this book is about – a broad agreement on the essentials of software engineering. We hear many developers say things like, "Ok, it [referring to innovations and practices] works for you but our situation is different!"

- You are building a small application, but we are building a large scale system.
- Your team members are very experienced, but we are just starting up.
- You are building a new system, but we are dealing with a legacy system.

Sometimes the differences are excuses. Sometimes, they are exaggerated. Other times, they are real. Whatever the case, it points to an underlying problem; a lack of a common ground for communication and for learning from one another – or put another way – a lack of a sound foundation for developing software that is applicable across different software development endeavors. Our industry needs a renaissance for the *way it develops*

software – a renaissance for methods[3].

Your method may be described or it may be kept tacit but if nothing else it will be manifested in the conversations in your team. A method is what you do every day as a team, not what is written down somewhere. Therefore, when your team improves how to work together, it is reflected as changes to your team's method.

At the end of 2009, Ivar Jacobson, Bertrand Meyer and Richard Soley started an initiative called Semat with the aim of refounding software engineering. In their Call for Action (www.semat.org), they gave a broad definition of the problem that the Semat initiative is poised to address. Essentially, they called for a refounding of software engineering based on a solid theory, proven principles and best practices.

As we go to print, this call for action is signed by around 35 leaders in the software field – including leaders in agile and lean in both the industry and the academic world. It has also been signed by representatives of 16 large corporations and universities, and by close to 1700 supporters from all around the world. Two chapters of Semat have been set up – one in China and another one in Latin America – and more are on the way.

The key idea of Semat is that underlying all methods is a kernel of things that you always have, always do and always produce, regardless of the technique or practices you use. The support given to the Semat call for action comes from an understanding that the way we develop software can be improved significantly if we can find a widely agreed upon kernel. The problem is not lack of methods and techniques. In fact, for every problem in software

[3] In this book we use the term "method" as a synonym for "way of working." Please, don't think of methods as a thick book; methods can be tacit – undocumented

development, there are solutions out there. The problem is that the industry lacks a common frame to capture these solutions, their assumptions, and their context. Therefore it is only with great difficulty many methods and techniques are applied anywhere but in their original context. The kernel provides that common frame and the software industry, represented by developers and their teams, and the academics, represented by teachers, researchers, and students, will all benefit greatly. We hope to show this with this book and that you will agree with us that the kernel captures *the essence of software engineering*.

To be clear, Semat is not developing a new method. On the contrary it is an agile way of *working with methods* coming from any camp and for any kind of software. By that we mean that you can be agile working with your current method, with CMMI, with cleanroom development, with embedded software development, with legacy enhancement and migration, and so on. Semat is inclusive of all ways-of-working – modern as well as legacy techniques.

Why we wrote this book

Our primary goal is to explain how the kernel can help software developers. To this effect the book includes small case studies[4] and stories drawn from our diverse experiences. Through these case studies we demonstrate how the kernel can help developers do their job. The real value will come when such kernel is widely used and taught at universities around the world.

Our secondary goal is to present how far Semat has come. Let's be up front and clear about the fact that this book could never have

[4] When we use the phrase case study in this book we mean an example or experience that serves to demonstrate useful lessons to aid developers.

been written without the contribution of all the people working on Semat. They have through careful analysis and extensive discussions identified a candidate kernel and described it in a balanced way that can be widely agreed upon. They have developed a simple but powerful language with which we can describe our ways-of-working in an easy to understand way.

Though we attempt to give an as objective description as possible of where Semat is right now, the Semat initiative is still at its infancy meaning that we may have to update this text as Semat evolves. This does not mean that Semat is something new and immature. The key ideas of this work go back as far as 2004. An initial version of the kernel was developed in 2006 and has been used by several organizations to deliver software successfully. These organizations have not only improved the initial kernel but also broadened the use of it – to business development and embedded systems. In parallel many of those involved in Semat have tried similar ideas and seen how these ideas help software developers far more than traditional approaches to software methods. It is on top of this practical experience that the work on Semat is based. The fact that a diverse group of people from different parts of the world building software for different domains with differing constraints were reaching similar conclusions gives us confidence that the kernel approach is on the right track.

The work with the Semat kernel is still ongoing. Still, we want you to try it. We want you to give us feedback. Like it, or hate it, please tell us. We need your help to improve what we have. This is why we have written this book.

Who this book is for

The book is intended for anyone that wants to have a clear frame of reference for how software development fundamentally is done. We think this is useful for both academia and professional developers.

For developers the goal of this book is to show how the kernel can help solve the challenges you face every day when doing your job. We demonstrate how the kernel is used in different situations from small scale development to large scale development. We discuss the different challenges that may occur, how to think about them and how to come to overcome them.

Today many software developers want to use modern agile practices, but they may face obstacles in their specific situation such as policy constraints or compliance regulations. While this book is not about a specific methodology it can help developers find the "right level of agility" regardless of their specific situation.

For students we want to put you in the shoes of professional developers so you can learn what you otherwise only learn through experience.

Now, to really have a long lasting impact on the software industry, educators have a key role to play. You, educators, also need to stand on the common ground: the kernel. If you do not stand on common ground, neither will your students, and future practitioners. The book is intended to be used as a supplement to an existing software development course, or as a textbook for a second course in software development. The book is organized to allow gradual learning and concepts are introduced incrementally. We explain a little; we demonstrate a little; we discuss and retrospect a little. Then, we proceed with the next set of concepts and so on. Educators, give us your feedback too. We hope that

you will apply the content in your classes.

What this book assumes about the reader

The book assumes the reader has some understanding of software development either by taking a software development course in the university, or through practical software development experience. For example, we assume that you already are familiar with practices like user stories, test driven development and Scrum.

How this book is structured

The book is structured into these parts.

Part 1 – The Kernel Idea Explained

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

Part 2 – Using the kernel to run an iteration

A walk through of how the kernel can be used to run a complete iteration

Part 3 – 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 product

Part 4 – Scaling development with the kernel

Shows you how the kernel is flexible in supporting different

practices, different organizations and different domains

Part 5 – Principles and Values

Reviews the principles and values behind the kernel. A discussion of highlights and key differentiators of Semat and this book

Part 6 – This is not the end

A forward looking discussion of the ways we think we can get even more value from the kernel now when we have it

The book also includes a number of appendices for subjects not for every reader, but that still will interest many of you. Appendix B may be particularly useful to anyone looking for a summary of content in this book that is inside the kernel, content in this book that is outside the kernel, and what is in the kernel but which is not covered in the book.

Contents

Part 1 – The Kernel Idea Explained

Software development is a continuous learning process, you don't just learn it once in school, but throughout your career you will gain more knowledge – you will become experienced.

Experience cannot only be gained by experimentation but it can be shared. The best way of getting new developers[5] up to speed is to sit them next to someone more experienced that continuously share bits of experience and support them as they grow. At the same time, our universities are crying out for experienced developers to share with students what it is really like to be a software development professional. But our best and most experienced developers have limited time and we must be cautious of how much we ask from them.

There are many kinds of challenges in software development, writing good code, finding good requirements, ensuring quality, using an appropriate architecture, releasing your solution to the customer, and so on. How can we make it easier to share experience across all aspects of software development?

The answer all boils down to having a deep appreciation of the essence of software engineering, the foundations, what is always true and present. This essence is captured in the kernel.

[5] By developers we mean anyone who plays a part in a software development endeavor, including but not limited to programmers, testers, analysts, architects and leaders.

1 Common developer challenges and kernel introduction

Being a developer on a tight software development team is very engaging. You are creative, solve hard problems, and collaborate with customers and colleagues. The sense of pride shared in the team when you deliver a high quality piece of software and get praise by end users is truly rewarding. But why can't it be like that more often?

In our work, we meet many developers around the world and even if they all aspire to be on that tight-knit, creative, and focused team they are often hindered by different kinds of uncertainty. Let's hear what some of them have to say about their challenges:

Fred: "The requirements keep changing so our iterations aren't converging on anything"

Eric: "The proposed software architecture doesn't seem to solve the right problems, but I don't have any input to change it"

Susan: "Management is panicking and throwing more developers on the team, but we already have difficulty coordinating the existing team members"

Steve: "I have to spend a lot of time pretending I am following the mandated development process and producing documentation that isn't really useful for anyone"

These are some of the challenges and frustrations, which Fred, Eric, Susan and Steve are facing. You may have your own version

of what they say, and you may also have other challenges.

We all hate challenges like these; they get in our way when trying to do the creative work we like. What can we do to help Fred, Eric, Susan and Steve?

1.1 Why is software development so challenging?

Let's take a step back. Let's not just jump to conclusions and suggest to our developers what we think is best. Instead let's think about what is best for them! One thing for sure is that software development is complex. But *why* is it so complex?

Software development is multi-dimensional – First, challenges can come in any dimension: requirements, architecture, teamwork, competency, and so on. For example, Fred is feeling pains with changing requirements, Susan is facing problems with team organization, and of course with the way management is acting. But, to be successful, you need to deal with all the dimensions simultaneously. Fred's project might not be successful if his requirements never converge. But there might also be problems in other dimensions. Further, solving a problem in one dimension may help solve problems in another.

Software development requires competent individuals – Fred needs to help the customer understand what is needed. The customer representative, Angela, likes to have someone to bounce ideas with. She would appreciate not somebody who simply takes down what she says, but to help her clarify her thoughts. She needs someone whom she can trust. She herself wants very much the requirements to converge. This requires communication skills and requirements elicitation competency from Fred. Fred needs to

be competent enough to give Angela a match.

Software development is a team sport – It is not enough that Fred knows the solution to his requirement problem. His team needs to understand things too, and so do his customer, Angela, and his management. They have to do the right things right and they have to be convinced that they are doing the right things. Otherwise, what Steve say would be true. They would just be pretending that they are following a method, and wasting precious time on irrelevant things. And it is not just about requirements. Together, the team needs to have competencies covering all aspects of the endeavor: architecture, test and so on.

So far, we have not come to a solution to the challenges faced by Fred, Eric, Susan and Steve. However, experience tells us that we need to address three main characteristics of software development: (1) multi-dimensional challenges (2) competency, and (3) teamwork.

1.2 Getting to the essence of software development: the kernel

So, what answer shall we give Fred, Eric, Susan and Steve? It is really back to the basics. Get the basics right, use competent people, and collaborate to get the job done. Well, this is still abstract for Fred, Eric, Susan and Steve, and we need to make this more concrete. This is precisely what the kernel does – it makes the essence of software development concrete. The kernel comprises a number of things that together captures the essence of software engineering and provides practical guidance for the likes of Fred, Eric, Susan and Steve:

a. The kernel identifies several aspects of a software endeavor

which a team must be mindful off and assess for progress and health. These includes a software endeavor's requirements, opportunity, stakeholders, team, work, way-of-working and of course the software system to be delivered. The kernel identifies states for each of these aspects as a guide for how to work with them effectively.

b. The kernel raises the visibility of which competencies a team needs to successfully carry out a software endeavor.

c. The kernel identifies areas of work that you need to attend to in order to achieve progress and health in a software endeavor.

These things which the kernel identifies will, as you will see, help you as a developer resolve the typical challenges of software development.

1.3 Using the kernel to address specific challenges: An example

To appreciate the usefulness of the kernel, let's discuss Fred's situation. Recall that Fred and Angela had problems with changing requirements. Now, Smith, a more experienced colleague of Fred and Angela, was asked to help. The discussion below demonstrates how Smith through the kernel worked with Fred and Angela to resolve the challenge. In the following text, the words in **bold** and *italics* come from the kernel. In this chapter we will use them intuitively first and after walking through the conversations between Fred, Angela and Smith, take a step back and discuss what the words in bold and italics really stand for.

1.3.1 Getting to the essence of software engineering

The first thing, which Smith needed to do was to understand why Fred's requirements were changing and the impact to the development. So, Smith started asking questions related to the following:

Opportunity – The system, which Fred was working on must meet some business need or opportunity. What was that need? It must be of value somehow. What was that value? If this value was not targeted, then going down to detailed requirements would not be useful.

Stakeholders – Was Fred getting information from the right stakeholders? Were the stakeholders engaged and really thinking about requirements? Were the stakeholders in agreement? Specifically, was Angela someone who had knowledge of the system (at least from a user's point of view) and the end-users? Was Angela someone who could make decisions on the scope of the system?

Requirements – Does Fred have a clear understanding of the scope of the system and its requirements? Was he focusing on the requirements in the right order? Was he trying to get too many requirements nailed down upfront? Could he have worked on the requirements that are more stable while the stakeholders get a better understanding of the rest?

Software System – Are the requirements changing because the stakeholders do not have a good understanding of the capabilities and limitations of the existing software system? Is the system and its architecture robust to changes? Is this a requirements problem, or a design problem?

Team – Was Fred and his teammates able to collaborate to design a robust system, a system to withstand changing requirements without much impact? Was Fred someone, who could ask the right questions? Or was Fred merely a note-taker who simply wrote down what the stakeholders say without attempting to help them clarify their thoughts.

Through the conversation with Fred, Smith found out that the key problem was that Fred was not working on the right set of requirements. Now, there were areas which Angela, the customer representative, was clear about and there were areas of which Angela was still not quite sure.

Fred was building a mobile device application. Some parts were about enhancing existing capabilities (e.g. better sound and video experience, better connectivity to social networks) and those parts were clear. There were other parts which Angela called "Value Added Services", which Angela hadn't got her head around. Fred and his teammates were working on these "Value Added Services" because they were deemed important by Angela – "top most important requirements" she said. Angela was also a busy person and only available once in a while.

During this time, the team spent more time discussing and demonstrating what the system could do and did rather than what the system must do. In other words, they had trouble staying focused on their immediate goal. In addition, instead of implementing those requirements that were clear they were implementing those requirements that were unclear (i.e. ambiguous). Hence, Fred and his teammates could not converge on anything.

1.3.2 Addressing the challenge

Having recognized why Fred's requirements were changing, Smith, Fred and Angela could start to devise a strategy to deal with their situation.

They agreed that for those requirements that were clear, they could follow their current approach. Fred and his teammates could work iteratively and demonstrate the improvements.

They also agreed that for those requirements that were unclear and hence subject to changes, an alternate approach was needed. They agreed that implementing requirements in executable code would be too wasteful. Instead they planned to have a separate track just to nail down the requirements. This plan comprised the following:

1. Explore Possibilities: The **opportunity** is about the reason for the new, or changed, software system. This had to be made clear to understand the need for the "Value Added Services" to bring the **opportunity** to a *value established* state.
2. Understand Requirements: Once the value of the **opportunity** is established, Fred and Angela need to quickly get the **requirements** *bounded* for the "Value Added Services" and later *sufficiently described* for Fred's team to start building something. Smith recommended walking through some end user scenarios as a means to bound and ensure sufficiency of the requirements.
3. Shape the System: Fred's team, based on such a better understanding of the requirements, could then implement the **software system** to a *demonstrable* state as a confirmation of what Angela needed.

To be successful with the plan, Fred needed someone with the right skills to help Angela explore the **opportunity**. Angela herself,

as a customer representative **stakeholder,** needs to represent her user community, and actively manage scope. Now, Fred and Angela weren't quite sure they had the right skills, so they asked Smith to help them. Smith, being tasked to help the team, gladly agreed.

So, they got together and executed their plan. Of course, it was not a miraculous change. In the beginning, their workshops progressed things slowly as they tried to nail down the **opportunity** and the **requirements**. Smith had to prompt Fred and Angela with what they needed to focus on, i.e. to do the right things right. Gradually, Fred and Angela understood. Angela brought in other people to join the discussion. Fred also involved his teammates. So, the **team** met their challenge by *collaborating*. But one thing that was happening for sure: they were converging!

1.4 Learning how to address development challenges with the kernel

Smith was quick to find the root causes of the changing requirements that Fred was facing. But how did Smith do it? What knowledge and experience did he use? Can it be made tangible?

If you are good at helping floundering software development endeavors, you know how to do all what Smith did. You know the things to look for; you know how to ask the right questions without offending Fred and Angela. You know how to guide Fred and Angela to a solution. You can see what is sound and what is not and you have skills to help others see that too. You have everything you need in your head and don't need anything more.

However, most people they benefit from a little help, a gentle

reminder, and a little prompting to help them think about the challenges and arrive at a solution. This is where the kernel becomes useful.

The kernel captures a set of concepts that are essential to the success of any software endeavor. In our earlier discussion about Fred's challenges, we have used a number of bold words and phrases, which represents concepts from the kernel. The benefits are twofold:

1. By understanding kernel concepts and knowing how to apply them, you can address many of your challenges.
2. If you and your team mates have a common understanding of these concepts and the words used, you can get to an agreed solution even faster.

So, now, the issue is how you and your teammates get this common ground. This is what the kernel is about, which we will discuss in the next chapter.

2 A little more detail about the kernel

The kernel captures a set of concepts, principles and values that are essential to the success of any software endeavor. It is a knowledge framework that helps you communicate effectively with your fellow teammates, overcome challenges, and run your software endeavor better. These concepts, principles and values are well known and described in the vast amount of software engineering literature we already have. What the kernel does is bring them together in a concise manner so you can make effective use of them. In the following two chapters, we will take a closer look at the kernel and what is in it. As a start, in this chapter, we will look at how the kernel helped the team in chapter 1, while chapter 3 will provide a high level view of the entire kernel.

2.1 How to use the kernel to address specific challenge: An example

In the previous chapter, we showed how Smith was able to find the root cause of the team's requirements challenge and come to a solution quickly. How did he do that? He used a number of concepts from the kernel, alphas and states, marked in **bold** and *italics* respectively in the previous chapter. In this chapter, we will go through the words and phrases Smith used and describe what they mean.

Note that Smith only used part of the kernel, so in this chapter we will only discuss this part. There is more to the kernel and it can be

used to also address challenges with architecture, test, managing your work, and so on. Chapter 3 provides an overview of the complete kernel.

2.2 Introducing the alphas

The kernel has a few key concepts. The concept that we will look at first is what is referred to as an alpha. Alphas are subjects in a software endeavor whose evolution we want to understand, monitor, direct, and control. Examples are the ones we already have introduced such as Opportunity, Stakeholders, Requirements, Software System, and Team which Smith all used.

> Alphas represent the things you need to monitor for progress and health to successfully steer you endeavor to a successful conclusion and have states and checklist to that effect. A good way to remember this is the mnemonic "Abstract Level Progress and Health Attribute" which stresses that 1) Alphas are about progress and health, 2) they can represent abstract concepts, and 3) they should be looked at holistically as a set and not individually.

Alphas exist regardless of their concrete manifestation. For example, there will always be Requirements, regardless if you document them or not, or whatever practice you choose to capture them, e.g. as user stories, features, use cases, etc. Requirements always exist. A simple way to think about alphas is to view them as the most important things you monitor and control for software development success. To achieve this, each alpha has a series of states to guide a team towards achieving the software endeavor's objectives.

We will not go through all the alphas now, but look at the ones which Smith used (see Figure 1). Alphas are denoted by a stylized

Greek alpha character.

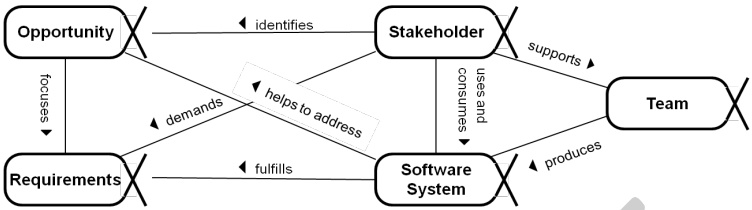


Figure 1 Alphas that Smith used to discuss changing requirements

1. **Opportunity**. The opportunity articulates the reason for the commissioning of the new, or changed, software system. It represents the team's shared understanding of the customer's real needs, and helps shape the requirements for the new software system by providing the business case for its development.
2. **Stakeholders**. The people, groups, or organizations that affect or are affected by a software system.
3. **Requirements**. Requirements are what the software system must do to address the opportunity and satisfy the stakeholder community.
4. **Software System**. A software system is a system made up of software, hardware and digital information that provides its primary value by the execution of the software. A software system can be part of a larger software, hardware or business solution.
5. **Team.** The group of people actively engaged in the development, maintenance, delivery and support of a specific software system.

As illustrated in Figure 1, there are relationships between alphas which show how problems in one dimension affect another. Smith

used these relationships to explore the root causes for changing requirements in Fred's case. One root cause he identified was that Fred's team started implementation work for the least understood requirements as opposed to the ones better understood.

Now, to do well in a software endeavor, your Opportunity must be good, your Requirements must be good, your Stakeholders must be good, your Software System must be good, and your Team must be good. How do you know if they are good? Associated with each alpha is a set of states which guides you to achieve goodness.

Smith used this list to help Fred explore the reason requirements were changing.

2.3 Alphas have states

As we have shown, the kernel alphas can be used to explore the root causes of a challenge. However, the kernel can also help to overcome the challenge. If Fred and Angela had been clear on why their requirements were changing, they would have been able to figure it out for themselves. However, they needed something more. They needed concrete steps to guide them.

Each alpha has a series of states to guide a team to achieve progress and health in a particular dimension. As an example, we show the states for the **Requirements** alpha. In the kernel, **Requirements** represent requirements for a software endeavor. It comprises a number of requirement-items, for example, features, user stories, or use cases.

Figure 2 Requirements alpha states

Side Bar 1 What is the scope of the Requirements Alpha?

So does this assume that we always must assign a single state to *all* of the requirements for the endeavor? No, in real life it will of course be so that different subsets of the total set of requirements will be of different maturity. So therefore, which state is applicable is dependent on the scope you are currently looking at. For example, you could be considering the specific requirement-items that you are going to address in the next iteration, or perhaps the subset of the requirements that are related to one of potentially many new features. We will discuss this when we demonstrate how a small team runs agile iterative development in Part 2.

The alpha states are defined in the kernel as follows:

1. *Conceived* – All development starts with some ideas as to how the eventual software system will address some users' real needs. At this point, it does not make any sense to start

coding, or even scoping an individual requirement-item, because requirements are still in a flux. Nevertheless, participants have a rough goal to work towards to.

2. *Bounded* – The team and stakeholders come to an agreement to the scope of the development – what is in, and what's out, what are the key requirements of the system. Bounding requirements do not necessarily mean having a detailed requirements document for all requirement-items. At this point, the team can start developing the top priority and stable requirement-items.
3. *Coherent* – The team organizes the requirements to reduce conflicts between them and gain a better understanding of their dependencies. This makes it easier for different team members to implement requirement-items in parallel, without waiting for one another or worrying about stepping on one another as they make changes to the software system.
4. *Sufficiently Described* – The team has agreed with the stakeholders on a set of requirements that defines an initially acceptable system. The requirements may only describe a partial solution; however the solution described is of sufficient value that the stakeholders would accept it for operational use.
5. *Satisfactorily Addressed* – The team first gets the highest priority and critical requirement-items implemented and demonstrate to the stakeholders and users that the eventual software system does indeed address their needs.
6. *Fulfilled* – Finally, the team gets all the agreed requirement-items implemented.

With a good understanding of the alpha states, Smith recognized that Fred's team was coding even when the Requirements for the Value Added Services were merely *conceived*. Smith understood that

Fred needed to get them to the *sufficiently described* state. This does not mean that Fred needs to hold all implementation work until the full set of Requirements is sufficiently described. What is important is that implementation work related to the less stable requirements (i.e., the ones that just have reached the *bounded* or *coherent* state) should be focused on driving out uncertainty and risk, as opposed to striving for completeness. Indeed, some demonstration of the software system to Angela and other stakeholders would be useful to gain agreement. However, it is only when he got Requirements to the *sufficient* state that he can be reasonably confident that the risk of requirement changes is mitigated. Of course he also needed to get the other alphas, such as Opportunity, Stakeholders, Requirements and Software System to their respective states. The target states for these alphas are:

Opportunity: *Value Established* – The value of addressing the opportunity has been quantified either in absolute terms or in returns or savings per time period.

Stakeholders: *In Agreement* – The stakeholder representatives have agreed upon minimal expectations of deployment.

Requirements: *Sufficiently Described* – The requirements describe a system that is acceptable to the stakeholders.

Software System: *Demonstrable* – An executable version of the system is available that demonstrates the architecture is fit for purpose and supports functional and non-functional testing.

2.4 Introducing activity spaces

In many cases, the alpha states would be sufficient inputs for teams to work out a plan to address their challenges. But some teams need more. In particular, Fred and Angela needed concrete

advice to overcome their challenge. They needed to know how to work together to achieve the target alpha states. They needed to get people involved. They needed to set up meetings and discussions. They needed some actionable plan.

In addition to alphas, the kernel has another concept called activity spaces to support this. Activity spaces are high-level activities to achieve a set of target alpha states. In particular, the activity spaces that can help to achieve Fred and Angela's target states are depicted in Figure 3. Each activity space is denoted by a dashed arrowed pentagon. The dashed outline indicates that it is high-level and can be refined by more detailed activities.

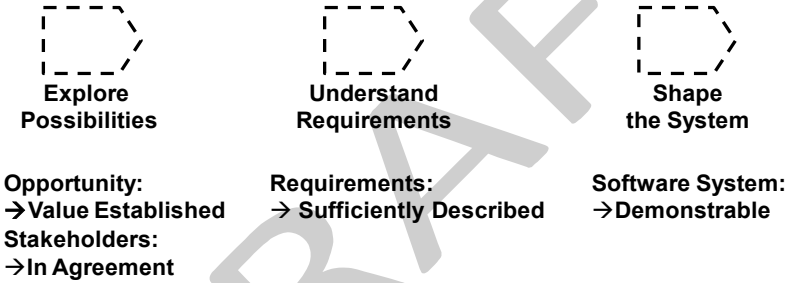


Figure 3 Activity spaces that Smith used to nail down requirements

In Figure 3, the alpha states that can be achieved are listed below each activity space. For example, the goal of Explore Possibilities is to reach the Opportunity: *Value Established* and the Stakeholders: *In Agreement* states. Fred and Angela use the names of the activity spaces as titles in emails they send to stakeholders and teammates when they organize workshops and reviews.

2.5 There is more to the kernel

What we have looked at in this chapter is not the complete kernel. We only showed what Smith, Fred and Angela needed to deal with their particular challenge. You can discuss and address many other challenges with the kernel, through other alphas, and activity spaces.

Glancing Forward: We will later in this book show how to use the kernel, and to scale the usage to different kinds of development, small projects (in Part 2, and Part 3), big projects (in Part 4).

3 A 10,000 foot view of the full kernel

Even if the kernel captures the essence of software engineering, which is well known and understood by many software professionals, there are things with the kernel that are new. In this chapter we will provide a high level overview of the entire kernel, including the key innovations of the kernel approach and the different concepts found in the kernel. In a way, this chapter can be seen as an introduction to the proposal submitted by the Semat community to the OMG. [http://www.semat.org/pub/Main/WebHome/SEMAT_submission_v11.pdf]

We start with, finally, defining what a kernel is and highlight what is new with the kernel approach. We then present the different concepts and elements that are found in the kernel. This is not meant as a full account of the kernel but detailed discussion of how the kernel can be used is saved for later parts of the book. The different kinds of elements in the kernel which we will look at are:

1. The essential things to progress and evolve – the alphas
2. The essential things to do – the activity spaces
3. Competencies

Of course, the first two have already been informally introduced in the previous chapters.

3.1 Innovations in the kernel

So without further due here is the definition of what a kernel is.

> A Software Engineering kernel is a stripped-down, light-weight set of definitions that captures the essence of effective, scalable software engineering in a practice independent way. The kernel forms a common ground for describing and conducting a software engineering endeavor.

However, in addition to providing concepts and definitions the way the kernel is constructed enables it to be used in innovative ways.

So far we have informally seen that the kernel is *actionable* – it prompts you to take action. The kernel is also extensible, which we refer to as it provides *a blueprint for growth*. Finally, it must provide *principles and values* to guide as when our situation is not covered by the kernel being actionable or by its blueprint for growth. Thus, there are three key aspects to the kernel (see Figure 4).

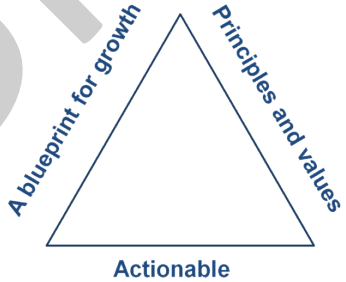


Figure 4 The three key aspects of the kernel

3.1.1 The kernel is actionable

While it is true that the kernel attempts to provide a concise list of words commonly found in software engineering, it is not a static dictionary that you read. The kernel is dynamic and actionable. By that we mean you can make the kernel come alive. You can use it in real life to run an endeavor (e.g. a project or sprint). It includes the essential elements always 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.

3.1.2 A blueprint for growth

The kernel includes the essentials elements that are universal for all software development endeavors. In addition, the kernel provides a simple kind of language that allows you to extend it to support what you need. Together, this means that the kernel contains a blueprint for how to grow specific and more complete methods on top of it. On top of the essential elements you can add practices or guidelines of many different kinds such as user stories, use-cases, component-based development, architecture, pair-programming, daily standup meetings, self-organized teams and so on. You can add these practices on top of the essential elements and create your way-of-working by composing the practices you need. Some larger endeavors may want to extend the kernel with elements specific to their domain. Hence, the blueprint helps us to grow from the kernel to whatever size is needed.

3.1.3 Principles and values

The principles and values of the kernel help us to make decisions

that are not guided by the blueprint. This ensures that developers use and extend the kernel in a coherent way.

They can be categorized as follows:

1. Principles and values to systematically and actively attack risks in software development. These are embedded within the kernel elements (e.g. the alphas) that guide you and your teammates in your software endeavors. For example, keeping stakeholders involved, balancing the breadth and depth of requirements, getting the architecture right first, and so on. All-in-all, it is about systematically and actively attacking risks in software development.
2. Principles and values behind how to use and apply the kernel during. We advocate an agile and lean approach in all that you and your team do. The kernel is lightweight and tangible so that you easily and immediately can put it to use. Based on the kernel, you can evolve your way-of-working incrementally starting from your existing method.
3. Principles and values behind how the kernel is designed. The kernel is based on the principle of separation of concern. For instance, we separate what the practitioners want to work with from what a process engineer needs; in such a way that the practitioner doesn't need to 'see' more than what is of interest to her/him. We also separate the needs that different practitioners have from one another; different people are interested in different things and at different levels of detail.

These principles are not meant to be a replacement for principles like the principles of the agile manifesto, or lean software development, but are merely the principles that the kernel supports and is based on.

Glancing Forward: We have introduced these three aspects of the kernel at this time to present the larger picture of the kernel. One primary focus for the earlier part of the book (Part 2, 3) is about the kernel being actionable – in essence, how to run software endeavors with the kernel. In part 3 we will also talk more about the blueprint for growth when we show you how you can add practices and create a method on top of the kernel. In Part 5 of the book you will learn more about the principles and values of the kernel when we discuss separation of concerns and agile in working with methods.

Now, let's return to the many kinds of challenges you face as a developer. What we have demonstrated to you through Fred's situation is but one of them. There are many more challenges. There will be challenges that are specific to you, but there are so called usual suspects, the common mistakes, which you as a developer ought to be equipped to deal with. In the previous chapter, we have demonstrated how Smith through the knowledge of some concepts in the kernel can help in solving one particular challenge. But to deal with the common challenges in general, you need full knowledge of the kernel.

3.2 Organizing the kernel

The Kernel is organized into three discrete areas of concern, each focusing on a specific aspect of software development. As shown in Figure 5, these are:

- **Customer** – Software development always involves at least one customer for the software that it produces. The customer perspective must be integrated into the day-to-day work of the team to prevent an inappropriate solution from being produced. This area of concern contains everything to

do with the actual use and exploitation of the software system to be produced.

- **Solution -** The goal of software development is to develop a working software system as part of the solution to some problem. This area of concern contains everything to do the specification and development of the software system.
- **Endeavor -** Software development is a significant endeavor that typically takes many weeks to complete, affects many different people (the stakeholders) and involves a development team. This area of concern contains everything to do with the team, and the way that they approach their work.

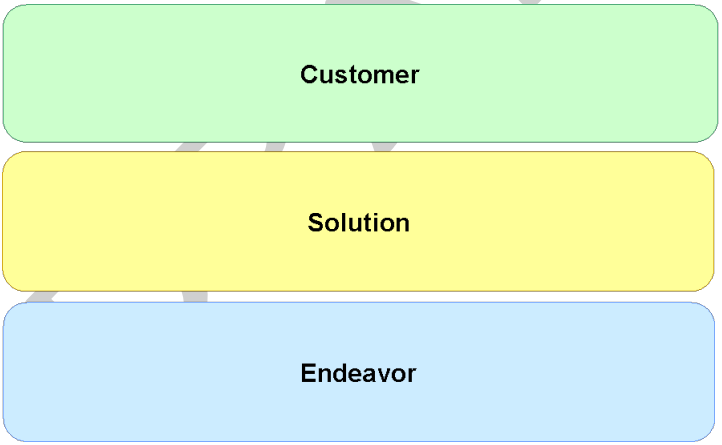


Figure 5 – The Three Areas of Concern

Throughout the diagrams in this book, the three areas of concern are distinguished with different color codes where green stands for customer, yellow for solution, and blue for endeavor. The colors will facilitate the understanding and tracking of which area of

concern owns which kernel elements.

3.3 The essential things to progress and evolve: the alphas

As explained in the previous chapter, in every software development endeavor there are a number of things that we always have, such as a team, and a number of things we always produce, such as a software system. These key elements are represented by the alphas of the kernel. For the sake of completeness we are repeating some material presented earlier. If this material is clear you can skim it quickly. The full set of alphas in the kernel is depicted in Figure 6.

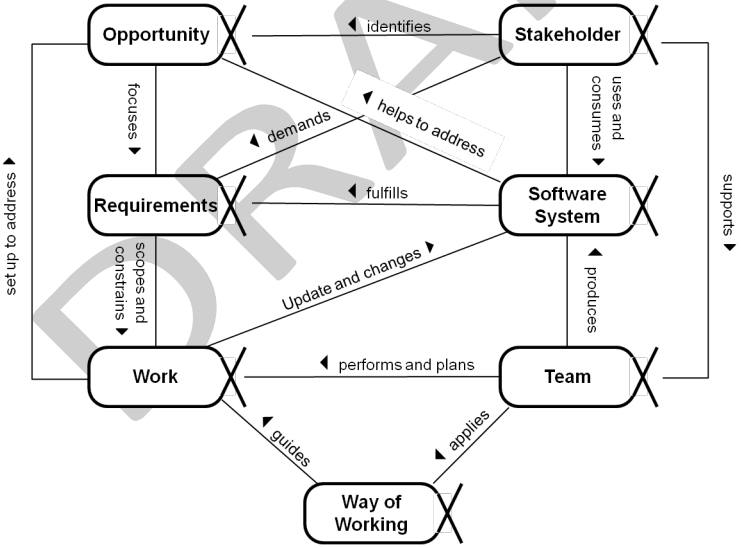


Figure 6 The kernel alphas

1. **Opportunity –** The set of circumstances that makes it

appropriate to develop or change a software system. The opportunity articulates the reason for the commissioning of the new, or changed, software system. It represents the team's shared understanding of the customer's real needs, and helps shape the requirements for the new software system by providing the business case for its development.

2. **Stakeholders –** The people, groups, or organizations that affect or are affected by a software system. The stakeholders are critical to the success of the software system. The feedback provided by the stakeholders or feedback on how the system may influence them shapes the software engineering endeavor and the software system.

3. **Requirements –** What the software system must do to address the opportunity and satisfy the stakeholder community. The requirements express the stakeholder community's intent for the software system. The requirements are captured, shaped, and understood in order to articulate the stakeholder community's need for a software system.

4. **Software System –** A software system is a system made up of software, hardware and digital information that provides its primary value by the execution of the software. A software system can be part of a larger software, hardware or business solution. We use the term software system rather than software because software engineering results in more than just a piece of software. Whilst the value may well come from the software, a working software system depends on the integration of software, hardware and digital information to fulfill the requirements.

5. **Team –** The group of people actively engaged in the development, maintenance, delivery, and support of a specific software system. The effectiveness of a team has a profound

effect on a software endeavors success, i.e., its progress and fulfillment of its mission. To achieve high performance, teams should reflect on how well they work together, and seek to reach their full potential and effectiveness.

6. **Work –** The activity involving mental or physical effort done in order to achieve a result. In the context of software development, work is everything that the team does to produce a software system matching the requirements, and addressing the opportunity presented by the customer. The work is guided by the practices that make up the team's way-of-working.
7. **Way-of-Working –** The practices and tools used by a team to guide and support their work. The way of working a team uses evolves with their understanding of their purpose and their work environment. As the team proceeds they continually reflect on their way of working and adapt it as necessary to their current context.

Each alpha has a number of states that help you as a developer to understand the current status of your development endeavor and determine your next steps (see Figure 7). What is required to reach a state is defined by a set of criteria that serve to guide the team's achievement of each state.

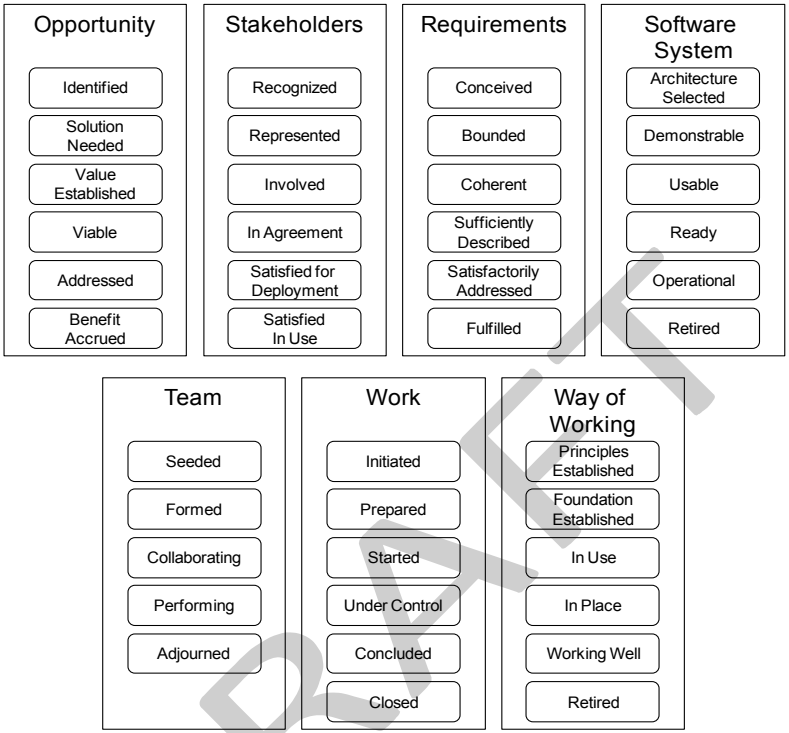


Figure 7 States of the kernel alphas

Side Bar 2 What is the scope of the Alphas?

In the same way as for the Requirements Alpha that was discussed in Side Bar 1 the scope of an Alpha and what state it is in is context dependent. For example when discussing a system that provides a number of more or less independent services, perhaps even to different stakeholders, what state is applicable is dependent on what part of the system you are talking about. Also,

for a system that is developed in increments, potentially in parallel, what state is applicable will naturally depend on the increment you are considering. Similar things could be said for the other Alphas and it is always important to be clear on the current scope when applying the Alpha states.

Glancing Forward: If you want to have a better understanding of the states, you can read more in Part 3 when we go through a story of a development that goes through every alpha and every state.

3.4 The essential things to do: the activities

In every software development endeavor, you do a number of things – which we call activities. Examples of activities include such as: agreeing to a user story with a product owner, demonstrating the system to some customer representative, estimating work, and so on. There are potentially a huge number of activities. The kernel organizes activities into activity spaces, which are depicted in Figure 8, but the kernel as such does not define any activities. You can think of activity spaces as containers to hold specific activities that will be added later. The activity spaces in the kernel are agnostic to your chosen methodology.

Each activity space is denoted by a dashed arrowed pentagon. The dash indicates that it is a space and that the space may be refined by concrete activities.

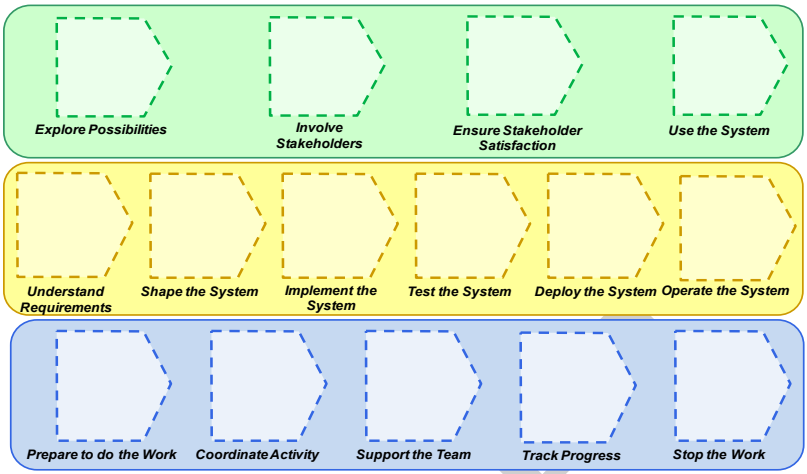


Figure 8 The kernel activity spaces

The activity spaces represent the essential things that have to be done to develop good software. They provide 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 organizes the concrete activities that progress one or more of the kernel Alphas. In Figure 7 the activity spaces are grouped by the Alphas that they are associated with.

In the top row the team has to understand the opportunity, and support and involve the stakeholders. To do this they:

- **Explore Possibilities:** Explore the possibilities presented by the creation of a new or improved software system. This includes the analysis of the opportunity to be addressed and the identification of the stakeholders.
- **Involve the Stakeholders:** Involve the stakeholders in the day-to-day activities of the team to 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 successfully addressed.
- **Use the System:** Use the system in a live environment to benefit the stakeholders.

In the middle row the team has to develop an appropriate solution to exploit the opportunity and satisfy the stakeholders. To do this they:

- **Understand the Requirements:** Establish a shared understanding of what the system to be produced must do.
- **Shape the system:** 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 overall design and architecting of the system to be produced.
- **Implement the System:** Build a system by implementing, testing and integrating one or more system elements.
- **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 7, the team has to be formed and the

work progressed in line with the agreed way-of-working. To do this they:

- **Prepare to do the Work:** Set up the team and its working environment. Understand and sign up for the work.
- **Coordinate Activity:** Co-ordinate and direct the team's work. This includes all on-going planning and re-planning of the work.
- **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 work and handover the team's responsibilities.

To make Figure 7 easily digestible the activity spaces are shown in an illustrative sequence which reads from left to right in each row. The sequence indicates the order in which things are finished and not 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.

3.5 Competencies

To perform well in a software endeavor, you need to have competency in different areas. You need the competencies not just to do the work you are responsible for, but also to understand and appreciate what your teammates are working on.

Competencies can be thought of as containers for key skills. Practitioners will populate their competencies with the skills necessary to do the job. Specific skills, such as Java programming, would not be part of the kernel because this skill is not essential on *all* software engineering endeavors. But there are always competencies required and it will be up to the developers and their teammates to identify their skill needs for their particular software endeavor.

A common problem often observed today on software endeavors is the lack of visibility of the gap that often exists between competency needs and available skills. The kernel approach will raise the visibility of this essential success factor.

At the time of writing (February 2012) the Semat group has not yet decided on a set of specific competencies to be part of the kernel. But to provide additional understanding of the concept we present in Figure 9 some possible competencies that may or may not become part of the kernel as the work in Semat continues.

Figure 9 Example of competencies

The example competencies in Figure 9 are:

1. **Customer Representative**. A person with this competency is skilled at gathering and communicating stakeholder needs, and has a thorough knowledge of the domain in which the solution will be deployed.
2. **Analyst**. A person with this competency is skilled at eliciting

opportunities, needs and stakeholder requests, and turning them into an agreed set of requirements.

3. **Developer**. A person with this competency is skilled at implementing systems based on the requirements.
4. **Tester**. A person with this competency is skilled at verifying that the developed software meets the requirements.
5. **Leadership**. A person with this competency is skilled at leading a team to a successful conclusion, one that satisfies the needs of the stakeholders, within acceptable time and cost.

3.6 Finding more about the kernel

The software community has developed software for more than 50 years. Irrespective of the code being written, the software system being built, the solution being constructed, the methods employed, or the organizations involved, we have recognized that there is a common ground, a set of basics, a foundation, an essence, a kernel of elements that are always prevalent in any software endeavors. With the kernel, you can find answers to your development challenges and questions faster and apply the answers more effectively.

What we have presented in this chapter is a snapshot. The work on identifying a proper set of kernel elements is still ongoing within a working group in Semat (www.semat.org). At the time of writing (Feb 2012), this group has identified a set of alphas, a set of activity spaces and a set of competencies.

Activity spaces and Competencies are not the emphasis of this book, so we will not describe them further. We mention them in this chapter just to let you have an overview of the elements of the full kernel. Subsequent books and papers will discuss the use of

activity spaces in greater detail.

Finding the right elements of the kernel is crucial. They must be universal, significant and relevant elements guided by the notion that, "You have achieved perfection not when there is nothing left to add, but when there is nothing left to take away."[6] Semat is also ensuring that these elements are widely agreed upon.

But remember, the kernel is also proven in practice. Many organizations have since 2006 been applying an earlier version of the kernel and the concepts of alphas, activity spaces, and competencies. Even if the definitions of the kernel elements are adjusted as they go through standardization with the OMG, we are confident that the fundamental proven concepts that we describe in this book will still apply.

[6] Antoine de Saint-Exupéry

4 The kernel alphas made tangible with cards

The kernel captures the essence of software engineering – the essential concepts to run software endeavors, to solve challenges and to communicate and collaborate effectively. So, how do you learn the kernel?

Of course, there are materials which you can read about every concept in the kernel – every alpha, every competency, and every activity space. But how do you remember them all and how do you apply your knowledge in practice? How do you best use the kernel to communicate in a team?

This is where the use of some tangible aids comes in handy. In particular, we want to highlight the use of cards. Cards make the elements of the kernel, and in particular alphas, which is the emphasis of this book, easy to digest and use. For this reason, we present the alphas in two ways, namely alpha definition cards and alpha state cards. We will continue with the discussion involving Smith, Fred and Angela to demonstrate the usefulness of these two kinds of cards.

4.1 Using cards as aids to address specific challenge: An example

Recall in Section 1.3.1 and Section 2.2 that Smith had to help Fred solve his challenge of changing requirements. When talking with Fred, Smith used words like opportunity, requirements,

stakeholders, software system, etc. These words were unfamiliar to Fred and Smith had to explain them. But Smith did something more.

Smith first showed a Requirements alpha definition card (about the size of a 5x3" index card), which lists the essential qualities of good requirements and its state progression (see Figure 10). Smith showed this card because Fred had requirements problems to begin with. Gradually, Smith put all the related alpha definition cards on the table as he explored the challenge with Fred.

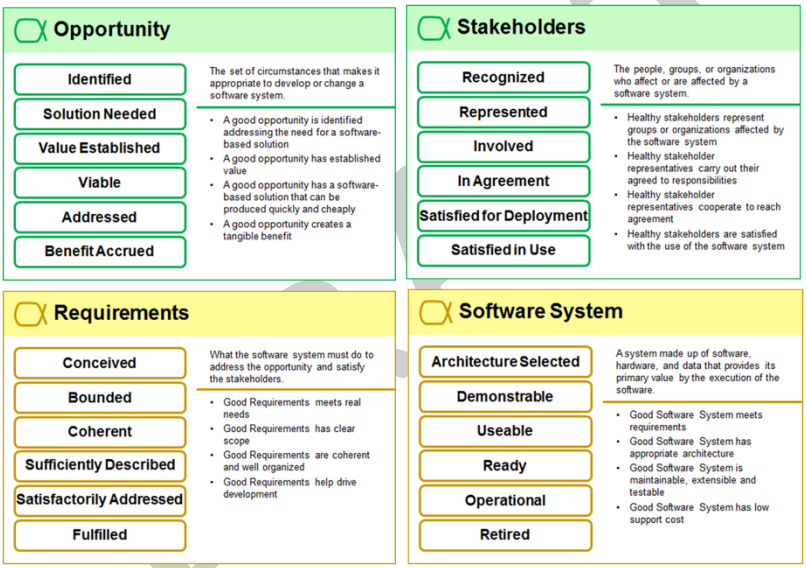


Figure 10 Alpha definition cards that Smith used to explore the problem

Through the alpha definition cards, Fred could visually explore the different aspects to his challenge of changing requirements. Fred was first amused by what he saw. He had never seen anything like it. But it certainly helped Fred focus his thoughts. He was not just hearing what Smith had to say. Fred was also seeing it.

As the discussion proceeded, Smith had to get to the meaning of each alpha state. So, he laid out another set of cards referred to as the alpha state cards. The alpha state cards are a set of cards, each about the size of a business card. Each state card has the name of its alpha, the name of its state and a list of criteria for achieving that state. As before, Smith started with the Requirements alpha and placed the state cards on the table in front of Fred (see Figure 11). The numbers at the bottom of each state card indicates the order of the state and the total number of states for that alpha.

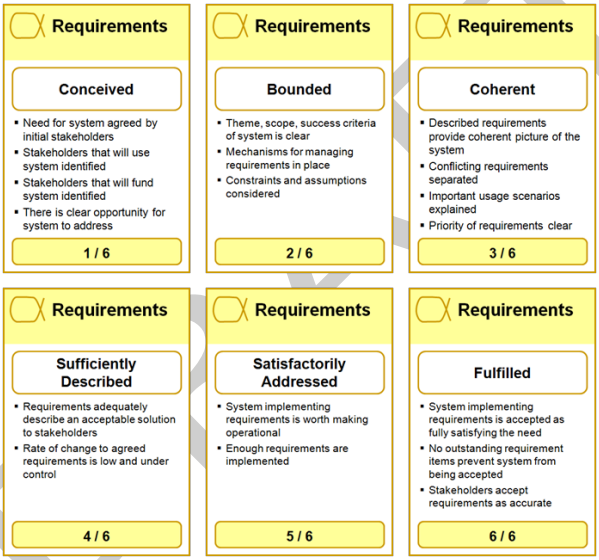


Figure 11 Requirement alpha state cards that Smith used to explore the problem

Smith pointed to the *Conceived* state card and told Fred, "This is where you are, and you are coding. Of course, you are not going to get anywhere". Smith then pointed to the *Sufficiently Described* state and continued, "This is where you want to be to be sure that your requirements are stable." Smith then moved the *Satisfactorily*

Addressed and *Fulfilled* state cards away as they were not relevant to their discussion.

Now, not only was Fred hearing and seeing what Smith had to say, he was also touching it! Fred shifted each state card slightly as an indication that he had finished reading it.

Recall in Section 1.3.2 and Section 2.4 that Smith had to provide concrete steps to Fred and Angela to nail down the requirements. They conducted a series of workshops to discuss different subsets of the requirements and Smith was invited to sit in to observe and facilitate. To kick off the workshops, Smith put the target state cards on the whiteboard. This acted as a reminder of the objectives of the workshop. As an example, during the Understand Requirements workshop for the Value Added Services, Smith put the Requirement alpha definition card and the *Conceived*, *Bounded*, and *Coherent* state cards on the whiteboard.

When discussions strayed, Smith would point towards the cards. At the end of each session, Fred and Angela would review if they had achieved the states based on the listed criteria.

By observing what Smith was pointing at on the cards, both Fred and Angela were able to articulate why the requirements were changing, how they were impacting their work, and how to address the issue. By explaining, using the words from the kernel, and reinforcing with the cards, the whole team was soon speaking with the same vocabulary.

Keep in mind that the cards are just one of the many possible techniques to help developers remember the kernel and apply it. However, our experience is that most teams need some mechanism to keep the relevant states of the alphas and their checklists visible during their daily activities. When we say the kernel is actionable we mean it isn't just a description of what someone would like the team to do. It represents what the team

actually does and includes the team's actual progress and health. As developers go about doing their daily work the results of what they do are reflected in the current and target states of the kernel elements. This helps everyone in the team, regardless of experience, to have a shared understanding of where the team is and where it is going.

4.2 Making the kernel come alive

The alpha definition cards and alpha state cards are very powerful tools. An alpha definition card is about the size of a 5x3" index card and summarizes what an alpha is about. Alpha state cards are about the size of a normal business card. So, what you have is the essence of software engineering captured on something quite small. This is a powerful visual cue, as we can dangle and shake the cards in front of software teams and say, "This is the kernel when it comes to applying it". This is fundamentally different to traditional methods described in thick books, and heavyweight manuals. Having said this, these more elaborate media still have value for team members who want to understand and learn more.

Figure 12 Kernel made tangible with state cards

Cards are tangible. You can easily move cards around during discussions. The left part of Figure 12 shows an example of this, from a Japanese pub, where a person is explaining the kernel to another person using the state cards. The tangible nature of the cards is very important. It makes the kernel concrete and come alive. We have worked with teams where each member carries a stack of alpha state cards with them. Other teams keep the alphas and their states visible as reminder sheets tacked up in their team area.

The right part of Figure 12 shows a team holding a daily meeting in front of a white board. The team have printed enlarged state cards and put them on their white board. This gets team members to view alpha state criteria on a daily basis. They use these criteria to determine if they have reached a particular state and what else they need to do to reach that state. Putting up the criteria for everyone in the team to see helps build a common understanding quickly. This is in contrast to more traditional approaches where such information is kept in large documents, which are seldom referenced, at least during daily meetings. We will discuss in more detail how to use the cards later in the book.

5 What the kernel can do for you

As a developer, you want to create great software quickly. You want to deal flexibly with challenges as they appear. As you advance in your career, you want to take on new, larger challenges, learn new domains and technologies. See Figure 13 how this can be visualized.

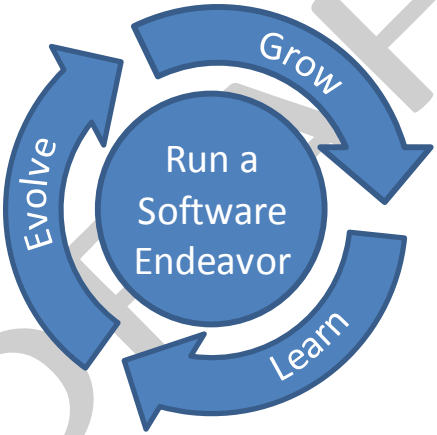


Figure 13 What software professionals do

To do this, you need to always be learning, and the way your team works must always be evolving. Yet, this learning and evolving must be something comes naturally and it should build on what you already know and do.

Glancing Forward: Later we will show you in part 2, 3 and 4 how you can run small and large software endeavors with the kernel.

Let's discuss how the kernel can help:

Run a software endeavor – The kernel helps you address typical challenges as we have demonstrated earlier in the book with the stories with Fred, Angela and Smith. We have seen how the kernel alphas help you to see progress and recognize health in what you do. We have seen how the kernel can improve communication in a team.

Growing – The kernel is not only useful for small software endeavors involving a small team of less than ten people, you can also use it in endeavors involving hundreds of people.

Learning – When you join a new team, you want to learn the team's way-of-working. When you decide to use a new technique in your team you need to get everyone up to speed using it. The kernel improves communication allowing for quick learning.

Evolving – Evolving means you can incrementally improve your way-of-working. It means you can change without throwing away more than what needs to be changed.

We will now look into this in more detail.

5.1 Run a software endeavor

Traditional methods are static; they may help you appreciate new ideas, but they don't help you while you actually do your job. Their descriptions are often heavyweight and do not match what you see or do in practice. There are two things that set the kernel approach apart from the traditional way of describing methods.

The kernel has focus on progress and health – One of the key elements of the kernel, as we have seen, is what we call alphas. Examples include: Requirements, Software System, Work, and

Team. The alphas are not just helpful to describe your method; they also guide you in achieving progress and health while you actually use the method. Each alpha has a series of states and each state has a set of checkpoints, which must be achieved before the state is reached. Hence, as your software endeavor proceeds you will also proceed through the state of the alphas.

Thus, you can, at every moment of a software endeavor assess which states that have been achieved, thereby understanding progress and health. This is a powerful foundation for measuring that you become better, faster and that your customers are getting happier. We will demonstrate this further in Part 2 of the book.

The kernel is actionable – In addition to understanding where you are, the states of the alphas provide a strong indication of where to go next. The checklists for each state provide the first level of guidance for how to achieve a state. In addition to that, activity spaces and activities are tied to the state progressions and can provide additional guidance on how to achieve coming states.

Side Bar 3 What is different with the kernel approach when it comes to measurement?

The software community has tried to measure progress and health for 40 years. So what is different with the kernel approach when it comes to measurement? The kernel is based on a broad experience base and contains the essentials that can help you measure software progress and health. This help is expressed as simple checklists associated with each state of the kernel alphas.

For example, developers who use the kernel know that to determine if they have achieved the *bounded* state for requirements they must ask themselves the following questions:

- Is it clear what success means for this new system?
- Have we identified all the stakeholders and do they have a

shared understanding of the proposed solution?

- Has the way the requirements will be described been agreed upon?
- Is the prioritization scheme clear?
- Are the assumptions and constraints clear?

These questions are really nothing magical, or new. But what is new is the recognition of the importance of reaching wide acceptance on the questions we ask to determine our progress and health, thereby helping us communicate more consistently and accurately. End-of-example.

Common use of the kernel will make it easier to communicate both with teammates and stakeholders of the software endeavor you are engaged in.

5.2 Growing

You might be wondering if the kernel is applicable to your unique situation – development of a totally new product, enhancing a mature product, customizing a product, building a platform, an embedded system, a telecom switch, a financial system, a military defense system, with experienced developers, with developers straight from school, with a co-located team, with a dispersed team (e.g. outsourcing), etc. The broad experience of those involved in developing the kernel gives us high confidence in that the kernel is applicable in each of these situations. But this doesn't mean it solves all possible problems.

This is why the kernel is extensible. You can scale to your unique needs. We have identified three different dimensions of scaling:

Scaling in – This is about going into appropriate levels of details

when your team members have different backgrounds or different levels of competencies.

Scaling out – This is about helping you deal with challenges across your product lifecycle, outside the actual development endeavor – from the fuzzy front end before development begins, through development, and on to the back-end when your software system is in operation.

Scaling up – This is about dealing with situations involving large number of requirements, more than one system, large number of people.

How you scale in these three different dimensions is described in *Part 4 – Scaling development with the kernel.*

5.3 Learning

The usual ways for people to learn include reading papers and books, taking courses and learning on the job from colleagues. It would be nice to attend a class just in time when you need a particular skill, but unfortunately scheduling such classes is not easy.

Learning by doing, learning on the job, is probably the most effective way to learn. But to do that you need support from someone that has the knowledge, time, skills and mindset to coach you in learning the new skill. Unfortunately, people that can do that are often very busy.

Fortunately, even if there many different methods (every team or at the least every organization has one) they are not as different as it may seem. Actually, most of them are made up of smaller commonly used methods which people usually call practices. Examples of practices are user stories, test-driven development,

backlog driven development (essentially what Scrum is).

The kernel provides a common ground independent of anyone particular practice, meaning that any method can be seen as a set of practices on top of the kernel. Some practices may be described while some may be kept tacit.

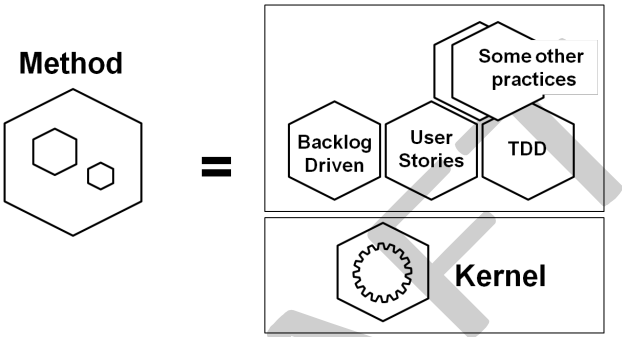


Figure 14 A method is a composition of practices on top of the kernel.

Side Bar 6 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 the way you actually work whether it is written down or kept tacit in your head. What the kernel does is help you structure what you do in a way that supports incremental evolution. In other words, it puts you in charge of the way you work.

Let's discuss how this helps:

Learning and training based on a common ground – When learning a new method, perhaps feature driven development, use cases, or user stories it is sometimes difficult to see how it will fit in to your current way-of-working. It sounds nice but you do not

know where to start. By basing what you learn on the common ground you can more easily apply new knowledge. You learn the kernel once and then you just focus on what is different with each new practice.

Learning and training focused on the essentials – The essentials are usually just a small fraction of what an expert knows about a subject, but if well selected it is a sufficient foundation for you to find out more by yourself. Learning the essentials enables anyone to participate in conversations about your work without having all the details. It helps to grow T-shaped[7] people, who have expertise in a particular field, but also broad knowledge across other fields. Such people are what the industry need as work becomes more multi-disciplinary.

The idea with describing practices on top of the kernel is a key theme of this book. Actually, we have demonstrated this idea in Smith's story in the earlier chapters. In the following sections we look into a bit more detail to see what really went on under the surface. A more full discussion of how practices are formed on top of the kernel is found in *Part 4 – Scaling development with the kernel.*

5.3.1 Making a practice explicit

Earlier in Part 1, Chapter 1, we demonstrated how Smith helped Fred and Angela overcome their problem with changing requirements. This is an example of a practice – a requirements elicitation practice. Smith had what he needed in his head as he coached Fred and Angela. In this case, the requirements elicitation practice is tacit – it is only manifested in the conversations between Smith, Fred and Angela. However, Smith also had to coach several other teams with similar challenges. Quickly, Smith

[7] http://en.wikipedia.org/wiki/T-shaped_skills

can become a bottle-neck for getting teams good at requirements elicitation. To avoid this it is probably a good idea to describe practices explicitly so that teams don't have to depend on Smith being available all the time. The kernel provides a lightweight language and approach to capture practices focusing on what the team produce and do.

Now, in this particular case, what Smith recommended Fred and Angela did not change what they produce, i.e., in what form requirements are captured, for example by replacing a traditional requirements document with user stories. Rather, Smith's recommendation is about what they do to come to an agreement on **Requirements**, through clarifying the **Opportunity** through involving **Stakeholders**. These things come from the kernel, and we color them grey in Figure 15.

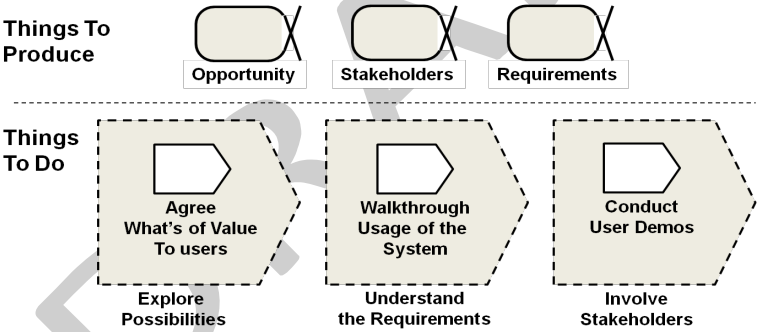


Figure 15 Example of requirements elicitation practice

Let's now look at the things to do. Smith's practice for requirements elicitation involves several explicit activities to help the team Explore Possibilities, Understand the Requirements, and Involve Stakeholders (see the grey activity spaces in Figure 15). Thus, making Smith's approach repeatable to other teams means describing activities as follows:

Agree what's of value to users – Review the success criteria for the system, what its differentiators are, and what the values are that the system needs to deliver to its intended users.

Walk through usage of the system – Walk through the use of the system step-by-step, and discuss the value which the system brings to the user at each step. Discuss and agree on what information users are interested in seeing, and what the user is interested in doing.

Conduct user demos – Conduct demos that walk through usage of the system from the user's point of view as a confirmation of the walkthrough above.

5.3.2 How explicit should practices be?

Practice descriptions provide additional guidance on top of the kernel for doing things. They can also describe the kind of work products to be developed. How explicit a practice should be, i.e., how detailed the descriptions should be, depends on two factors (See Figure 16):

Skill and competency – Skill and competency refers to a person's ability to figure things out for themselves. Team members with high skill and competency need only a few hints and examples to get going. Others may need training, and coaching to learn how to apply a practice effectively.

Background and experience – If the team has worked together using a practice in the past, or have gone through the same training, then they have a shared experience. 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 stories, then they have different backgrounds and experiences. In

this case, practices described to some level of detail will be useful to avoid miscommunication.

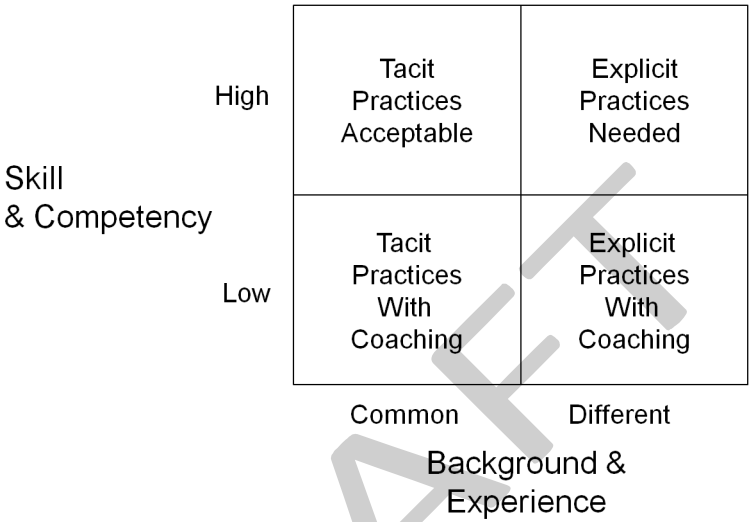


Figure 16 How explicit practices should be depends on background and competency

In the case of Fred's changing requirements, the situation was that Fred and Angela had different background and they didn't know that much about requirements elicitation techniques. Thus, they needed explicit practices and some coaching, which was what Smith provided.

5.4 Evolving

Ideally, changes to a team's way-of-working are evolutionary and not revolutionary. You cannot casually throw away your existing way-of-working when a new method appears. You may have

existing successful products that should live for many years to come. Further, it is unlikely that all is bad; there are surely things in your existing way-of-working that you want to keep.

To make evolutionary changes to your method, you need to recognize the parts of your method to replace and separate them from the parts you want to keep. For example, if a team wants to move from their traditional waterfall development method to an iterative development method, does that mean that everything needs to change? Or can the team keep its way of capturing requirements, planning, designing, testing, its team organization, and so on? Unfortunately most people think of methods as monoliths which makes it hard to determine what to change and what to keep. They lack the vocabulary and the tools to discuss and compare methods, which is another area where the kernel becomes very useful.

We will demonstrate this in both Part 2 and 3 of the book. In Part 2 of the book, we will discuss simple evolutions in an intuitive way, and in Part 3 we will discuss the same ideas in a more structured way.

Side Bar 7 A key difference with the kernel approach

Evolving is a key difference with the kernel approach. While it is true that in some occasions a team may need a more revolutionary reset of their way-of-working, it is far more common that a team's way-of-working is fundamentally working fine and is best improved a step at a time.

Today most methods are described and taught in a way that makes it difficult to change things one at a time. The kernel approach on the other hand supports evolutionary change in a way that allows teams to evolve their way-of-working as they go. Hence, the kernel is well aligned with lean concepts such as

Kaizen and *Teams evolve practices*.

Part 2 – Using the kernel to run an iteration

In Part 1 of the book, we demonstrated how you can use the kernel to address challenges in software development (i.e. Fred's story about changing requirements). Now in this part, we will demonstrate through a story how you can use the kernel to help you run an iteration. An iteration is a single run through the activities in your lifecycle with the goal of producing a stable increment of the software system. It is called a Sprint in Scrum.

In this part, we tell the story very much like a tutorial or a demonstration. So, in the interest of learning this is a somewhat idealistic case. In the next part (i.e. Part 3) we will take a broader view and discuss how you use the kernel from the point when a software endeavor starts to the point when the endeavor is completed. Later in Part 4, we will discuss how our approach scales to large development efforts.

6 Running iterations with the kernel: Plan-Do-Adapt

In this chapter, we will walk you through a story demonstrating how a development team built a software application. The protagonist in our story is Smith. He had to develop a mobile application that permits users to browse their social network (e.g. FaceBook, Google+) offline. The idea for this application came from Angela, Smith's customer representative from the marketing department. Her idea was to develop a mobile application that caches contents from a user's social network in their mobile device. A user can thus show his photos from his social network to his friends even when he does not have a connection.

In this part of the book we will see how Smith and his team perform their development iteratively. In particular we will walk through one of his iterations. In the next part we will walk through the whole development from idea to product.

6.1 Plan-do-adapt

Running a software iteration is similar to taking a journey. Just like a normal journey travelling from one location to another, you always need to know where you are now, where you are heading next, how much fuel you have, and how much further you have to go. As you progress along the journey, you might adapt the way you drive according to the road conditions and the weather. So, you are continually planning, driving and adapting (see Figure 17). This is how Smith and his team ran their iterations.

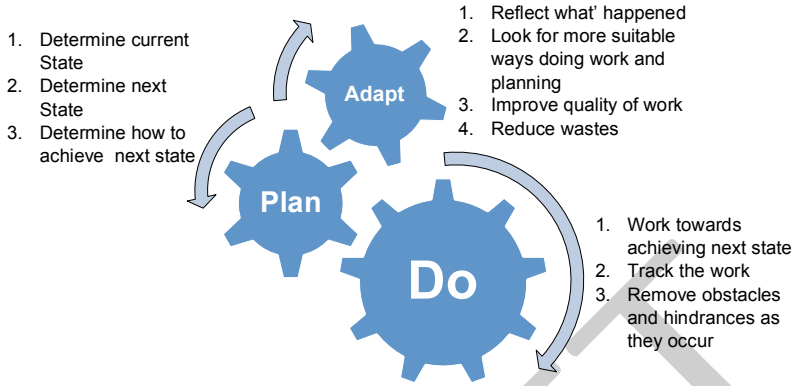


Figure 17 Plan-Do-Adapt Cycle

Plan – Smith's team determines the current state of the whole endeavor by understanding the current state of each of the kernel alphas (potentially different states for different parts of the endeavor). The next step for Smith's team is naturally to try to reach the next set of "target states" for all alphas. Smith's team decides how best to achieve the next set of target states by identifying pieces of work, or work items, to be completed. If the effort to complete these work-items exceeds the iteration capacity, then it will take more than a single iteration to achieve the next set of target states.

Do – Smith's team works towards achieving the next set of target states. This may involve writing code, testing, setting up environments, conducting requirements meetings, writing documentation and so on. His team tracks the work items and solves problems as they occur.

Adapt – Smith's team reviews their way of working, identifies obstacles and finds better or more suitable ways to do things. This often results in small changes to their way of working.

Note that plan, do and adapt may appear in any order. Actually, they might also be conducted together in one sitting. For example, while Smith's team plans how to capture requirements, it might also adapt their way to do this job. That is why in Figure 17, we draw the picture not with a simple cycle, but a set of wheels moving in unison. This is not meant to imply that it is ok to "do" before you "plan." It is rather meant to imply that planning and doing and adapting all happen continuously and often at the same time.

6.2 Understanding the story context

Before going on to tell Smith's story through the iterations, let's tell you a little bit more about Smith's situation. At the beginning of the development, Smith's team had only two developers, Tom and himself. Both had some understanding of the kernel. They were later joined by two more developers, Dick and Harriet, who were fresh and had no idea about the kernel.

Smith's company was one which had very little or no "process baggage". His company really did not believe in "defining methods". They encouraged creativity and agility. This worked well for more experienced teams who could really figure things out on their own. But Smith's company was also growing, and there were new hires joining them. These new hires while being very good with programming languages were less equipped in other aspects of software development, for example working with stakeholders to gain agreement on requirements. Therefore they required support. This was the context under which Smith had to build his application.

Success to Smith was not just to deliver the application on time, and with quality. Success also required that he grew his new

members so they could gain understanding and knowledge of software development. To be successful, Smith, his team members and all those involved with the application needed to be aware of the risks and challenges facing the development. The kernel embodies the different dimensions of risks and challenges through the alphas. Now, if you ask anyone involved, they will highlight different dimensions of risks and challenges. In other words, each has a different view of software development; a different view of the kernel (see Figure 18).

	Angela	Dave	Smith	Tom, Dick, Harriet
Opportunity	✓	✓	✓	
Stakeholder	✓	✓		
Requirements	✓	✓	✓	✓
Software System	✓	✓	✓	✓
Work		✓	✓	
Team			✓	
Way of Working			✓	✓

Figure 18 Views of the kernel from different participants

The people involved in the development of the application (see Figure 18) are:

- Angela is from the marketing department. Her main responsibility is to ensure that the software system addresses the opportunity through continuous involvement of stakeholders (her department colleagues, and department head).

- Dave is the department manager. His primary concern is to ensure something of value is delivered on time within the budgeted costs and human resources.
- Smith is the senior member of the development team. He is the only developer who has full knowledge of the company, its products, and ways of working.
- Tom, Dick and Harriet are new developers. They love to code, and write good software. However, they are not acquainted with the company's way of working.

In addition, depending on when you ask the participants, the dimensions they are most concerned about will change. For example, what concerns them most at the beginning of the development may be different from what concerns them most in the middle of development. Their concerns are likely to change again during acceptance testing. In kernel speak; this means the alphas you want to highlight or emphasize will be different in different stages of the development.

On the one hand, Smith's software endeavor, even though small and not really that complex, would still progress through all the alphas and their states. This is because the alphas are universal and exist in all software endeavors. So, if you ask all of the people involved in the story, you will come into contact with all alphas. However, from the point of view of assessing progress during iterative development, Smith's team (i.e. from Tom, Dick and Harriet's point of view) will not be equally concerned with all of them. All alphas are essential. But each situation is different requiring different degrees of attention on different alphas. For this specific example, to simplify things, we just focus on three alphas (see Figure 19). In the next part of the book, we will walk you through Smith's story from beginning to the end, and across all alphas.

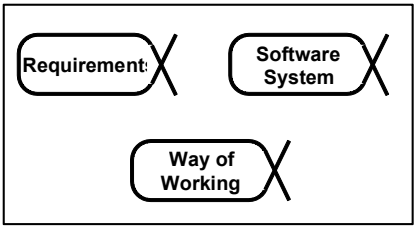


Figure 19 Alphas to emphasize in Smith's team

For this example, these three alphas Figure 19 are Tom, Dick and Harriet's primary progress tracking focus when running iterations in the middle of development. In each iteration, the team would:

1. Resolve some technical challenges.
2. Implement some requirement-items.
3. Check the progress and health of Requirements, Software System and Way of Working.

Side Bar 7 You do not need to make a fuss over the full kernel

The kernel comprises things that go on in every software endeavor. However, you do not need to make a fuss over everything. While all alphas are essential, you focus your attention on those most relevant to the situation you are in.

7 Planning an iteration

Software development can be viewed as a set of moves from some original state (or set of states) to some target state (or set of states). We have seen in the earlier chapters a little bit about how these states progress. .

A software development endeavor can be viewed as state progressions over a series of iterations. Planning is about the following steps: (also refer to Figure 20):

1. Determine the current state of the endeavor, which is about understanding where you are. The current state of the endeavor is expressed as the current state of the alphas.
2. Determine the next desired state of the endeavor, which is about agreeing where you want to go. This is represented by set of target states of the kernel alphas.
3. Determine how to achieve the next set of states, which is about agreeing the work-items the team needs to do specifically to achieve the target alpha states. In our story our team has chosen to use an iterative lifecycle. And they have chosen to use a backlog to manage their work-items. So in our story, the work items are loaded into the team's iteration backlog. When doing the iteration, team members simply work off the work-items from their iteration backlog.

Further on in this chapter and also in the following chapter we will look at how Smith and his team used the Alpha's to support the planning and execution of an iteration. As the subject of this book is to describe how the kernel can help software developers in everyday situations the focus of the story is very much on how the

Alphas are used by the team. Of course this is not the complete story of how Smith and his team went about their iteration but for the sake of brevity we have left out all of the things that already are well known regarding agile planning and execution. Please bear this in mind when you read ahead. Creating good software is of course the end goal for Smith and his team and the Alphas are merely a useful tool that can help to achieve that more elusive goal.

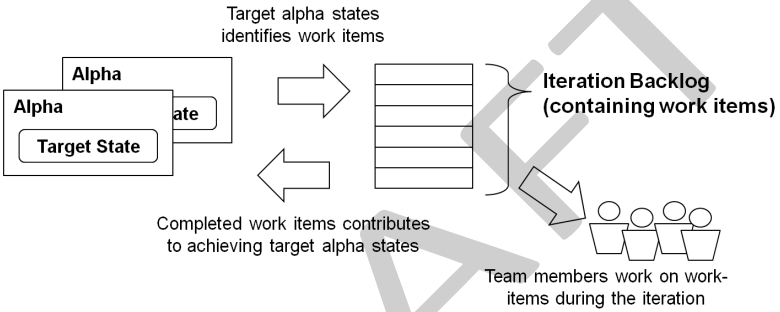


Figure 20 Working from an iteration backlog filled by work-items.

7.1 Planning with alpha states

The kernel encompasses a significant amount of information and we do not expect you to keep it all in your head. So, it is useful to have the key information at your finger tips. One way to do this, as we saw in Part 1 is through alpha state cards. What is important is keeping the alpha states tangible when planning your next iteration.

7.1.1 Determine the current and next states

When planning an iteration, the first step is to understand the

current state of the software development endeavor. The alpha states can help you with this.

Walkthrough – If your team members are new to the kernel, you might need to guide them a little. If you are using cards as discussed in Part 1 you could use the following steps with your team:

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 teammates if your team has achieved that state.
3. If your team has done so, move that state card to the left. Repeat with the next state card and do so until you get to the state, which 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 with teams experienced in using the kernel is poker. In this case use the following steps:

1. Each member is given a deck of state cards.
2. For each alpha, each member selects the state card which he/she thinks best represents the current state.
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, there are several possible reasons. First, there might be different interpretations

of the state criteria (checklist items). In this case, the team can discuss and reach a consensus. This approach also helps team members learn more about the kernel – the essence of software engineering. Second, someone might point out that some checklist items have not yet been met. In this case, you can add a task to achieve the criteria into your backlog. Third, it may also indicate that the team does not have a common understanding of the way of working.

These are just two possible ways to determine your current and next target states. Keep in mind that using state cards is not required to use the kernel. The idea is to get the team members to talk, and discuss what state they are in, and what state they need to focus on next. This gives everyone a chance to learn and contribute.

Once you have determined the current state of development, you also know where you want to go next – to the next state. At this point, you can now start focusing on the set of next states you need to achieve.

Making only the immediate next states visible to the team helps the team focus on what they absolutely need to get done. That being said, it is also useful to keep the future states to be achieved visible so the team can also see what lies ahead.

7.1.2 Determine how to get to the next state

The target alpha states for the iteration sets the objectives for the iteration and this information helps your team plan the iteration. The criteria (checklist) for each of the alpha states provide hints as to what tasks your team needs to do to achieve the objectives of the iteration. Keep in mind that in this part of the book we are just considering a small software endeavor. Reaching a new set of

alpha states may actually require several iterations to achieve. We will discuss later in the book how you identify tasks and measure progress on more complex efforts.

7.2 Determine current and next 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 useable by end users. Only Smith and Tom could use and demonstrate it.

Using the walkthrough approach, Smith led the iteration planning session. Using the above mentioned steps, they had achieved the states shown on the left, and the states not yet achieved can be seen on the right in Figure 21. Recall that in our story we are just focusing on the Requirements, Software System, and Way of Working alphas to help the reader learn how the alphas can help.

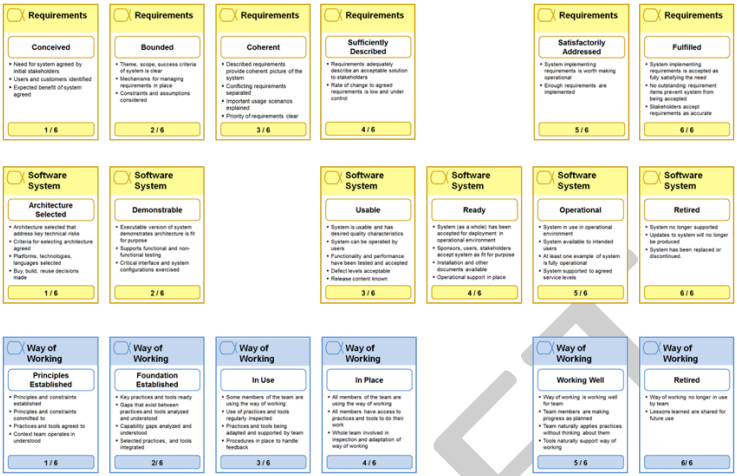


Figure 21 The team uses the alphas to determine the current states

Once the team had agreed on the current state, the team could focus on the immediate next states. The agreed immediate next alpha states became the objectives of the next iteration, but as stated earlier this doesn't mean the team could necessarily achieve these states in a single iteration. States can be achieved in the middle of an iteration, or can take multiple iterations to achieve (refer to Figure 22).

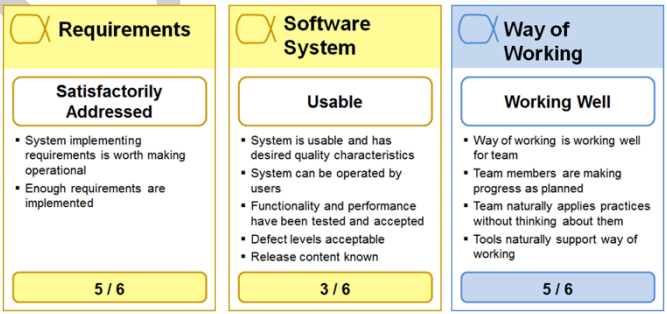
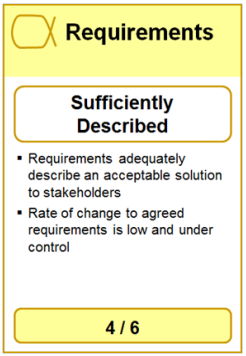
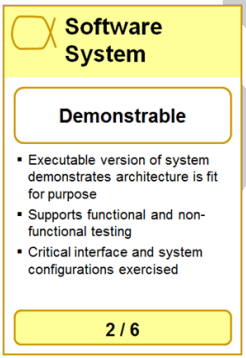
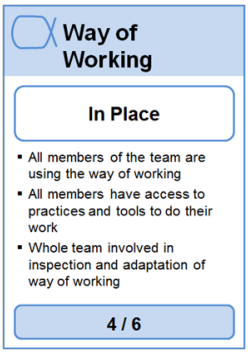


Figure 22 Highlight the next states

The table below shows the current state and describes how the team in our story it.

Current State	How it was achieved
	Smith's team had demonstrated an early version of the application based on a description of the proposed solution. After the demonstration the stakeholders agreed the described requirements were acceptable. The agreed to requirement-items were online and offline browsing of the social network, and making posts offline. However, these requirement-items were only implemented partially at the time of the demonstration. According to the state definition, our team has achieved the Requirements: *Sufficient* state.
	Early during development, Smith's team had identified the critical technical issues for the software system. Moreover, Smith's team had demonstrated an early version of the system to his stakeholders. This means that Smith's team had achieved the Software System: *Demonstrable* state. However, since Smith's team had not yet reached a quality level at which users can use the system on their own Smith's team had not yet achieved the Software System: *Useable* state.

Current State	How it was achieved
	The two new members, Dick and Harriet, who had just come on board were not fully productive yet. In particular, they seem to have trouble applying a test-driven approach to development, which the team agreed was important to maintain high quality during development. They had difficulty trying to identify good test cases, and writing good test code. Still they tried their best. As such, the team agreed that the Way of Working is currently in the *In Place* state because all members of the team were using the agreed to way of working. But they had not yet achieved the *Working Well* state.

Side Bar 4 Learning by doing

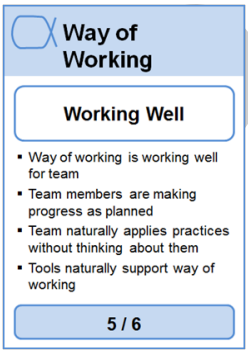
We have worked with many teams using the steps listed above. In most cases, the name of the alpha state itself provides sufficient idea about what classifies the state. If the alpha state name is not clear, team members can find out more by reading the alpha state criteria (checklist). By going through the states one by one for each alpha, a team quickly gets familiar with what is required to achieve each state. In this way, team members 'kill two birds with one stone'. The team learns about the kernel alphas at the same time as they determine their current state of development and their next target states.

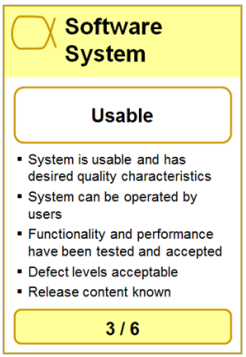
7.3 Determine how to achieve the next states in our story

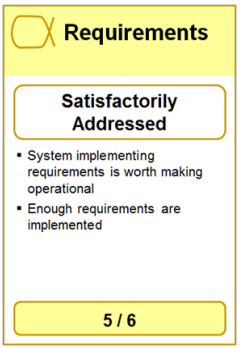
In this section we use the same technique as in Part 1; **bold** calling out Alphas, and *italics* referring to the state of interest within the Alpha.

Smith and his team looked at the next target states and agreed that there was some prioritization needed. In this case, they needed to first get their **Way of Working**: *Working Well*, then the **Software System**: *Useable* and finally **Requirements**: *Satisfactorily Addressed*. The reason was simple; not having the **Way of Working:** *Working Well*, would impede their attempts to get the **Software System**: *Usable*. In addition, they agreed that they needed to get some missing requirement-items useable first before working on others to achieve the **Requirements**: *Satisfactorily Addressed* state.

Smith and his team next discussed what needed to be done to achieve these states.

Target State	How they planned to achieve them
	Both Dick and Harriet agreed that they had difficulties in applying test driven development. They needed help in order to make progress as planned. Tom agreed that he had to invest time teaching them: A task was added to the backlog for Tom to conduct training on test driven development for Dick and Harriet.

Target State	How they planned to achieve them
	This state demands that the implemented requirement-items must reach sufficient quality for end-users to use them. 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", it means that each requirement-item must be fully tested within the test environment. 1. Task 3. Complete Requirement-Item A: "Browse online and offline". 2. Task 4. Complete Requirement-Item B: "Post comment (online and offline)". 3. Task 5. Complete Requirement-Item C: "Browse album".

Target State	How they planned to achieve them
	This state demands that more requirement-items had to be implemented beyond those listed above. Smith had to work with Angela to determine which requirement-items needed to be implemented to get to a system that would be worth making operational. This would result in some additional tasks to get the system worth being operational. Tasks 6: Talk to Angela and agree on additional requirements-items to make the system worth being operational.

So, by going through the target alpha states, Smith was able to work out a set of tasks (work-items using the kernel terminology) for the next iteration. These tasks were broken down so they could be completed within the iteration.

Side Bar 5 Value of individuals and interactions

It is worth pointing out that Task 6 is a different type of task than Tasks 3, 4 and 5. Task 3, 4 and 5 are tasks that will result in implemented and tested code. Task 6, as we will see in the next chapter, will lead to Smith having a conversation with Angela and agreeing on the additional requirements-items necessary before the system will be worth making operational. Such tasks are equally important to those that result in tested code and should be tracked to completion like any other task.

7.4 How the kernel helps you in planning iterations

So how does the kernel help a developer who is using iterative development? It is one thing to come up with a plan. Let's now go over how the kernel can help you come up with a good plan.

First, a good plan must be comprehensive, meaning that you do not miss essential items. Second, it must be concrete, meaning that it is understandable and the team knows what to do. This means that the team must have a way to monitor its progress against the plan. The kernel helps you achieve these characteristics in the following ways.

Comprehensive – It helps you with the different dimensions of software development. The kernel alphas cover these dimensions by serving as a reminder helping you keep your focus on the immediate target of your software endeavor.

Concrete – It helps you and your team through its defined common ground and common vocabulary, making it easier to reach agreement on your iteration plan faster with less misunderstanding. The checkpoints for each alpha state give you hints as to what you need to do in the iteration. The same checkpoints help you determine your progress by helping you to keep visible what you have done relative to what you intended to do.

Compact and Tangible – The kernel alpha states in the physical form as state cards are compact. You can carry them around with you. You can move them around on the table to conduct walkthroughs or use them in a poker fashion. This promotes discussion and makes the kernel come alive.

8 Doing the iteration

By following what we described in the preceding chapter, you would have a plan for the iteration. At this point, your team has a set of tasks to perform to achieve the objectives of the iteration. These tasks form your team's iteration backlog. Each day in the iteration your team will take tasks from the backlog and work on them. This proceeds until the end of the iteration when you once again review what you and your team have done and determine the new state of the development. The plan-do-adapt cycle repeats. This chapter looks at how the kernel helps you in the "do" part of the cycle.

8.1 Doing the iteration with the kernel

Doing the iteration means working off the backlog of work-items. Agile methods encourage teams to conduct daily meetings to discuss what each member has done since the previous meeting, what she or he is going to do that day and potential barriers that hinders their work. This helps each member understand what others are doing, and more importantly, provides a way for them to offer and get assistance from one another. Such daily meetings also offer an effective avenue for the team to adjust their plans in the light of what has happened.

Side Bar 6 The team chooses its own way of working

Keep in mind that in our story the team has chosen their own way of working including working off a backlog, holding daily meetings to discuss their work, and working in short iterations. Your

chosen way of working may differ. The kernel helps you in establishing and assessing your own way of working, but it doesn't require any specific way of working. For example, in situations where high reliability is a premium, your way of working may place much more emphasis on defect prevention, rather than adaption through short iterations. The kernel approach is equally applicable in these situations as it helps you plan where you currently are and what states need to be achieved next.

Such daily meetings are often conducted in front of a task board. A task board in its simplest form categorizes tasks into three columns, namely" "To Do", "Doing" and "Done". Task boards give teams a visual and quick overview of what the team is working on daily. It is also useful to put the immediate target alpha states onto the task board to help the team stay focused.

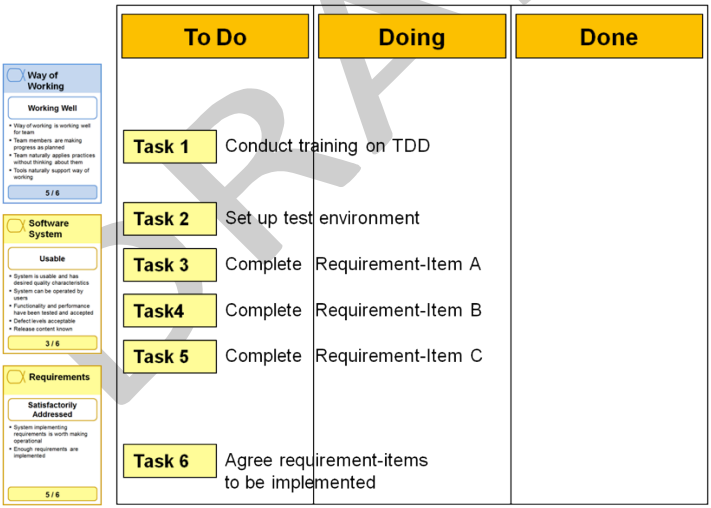


Figure 23 Task board at the beginning of the iteration

Figure 23 provides an example of what a task board may look like with the target alpha states pasted alongside. Specifically, it shows

the target states of the iteration Smith and his team planned as described in the previous chapter. On the left are the target alphas his team needs to achieve. The "To Do" column identifies the list of tasks planned. In practice, these tasks are often written on post-it notes, and the state cards are printed using a bigger size paper (as opposed to business card size) so that members can read and scribble on them easily.

8.2 Doing the iteration in our story

Let's look at how Smith and his team run the iteration.

Doing iteration – Day 1. Figure 24 shows what happened on day 1 of the iteration.

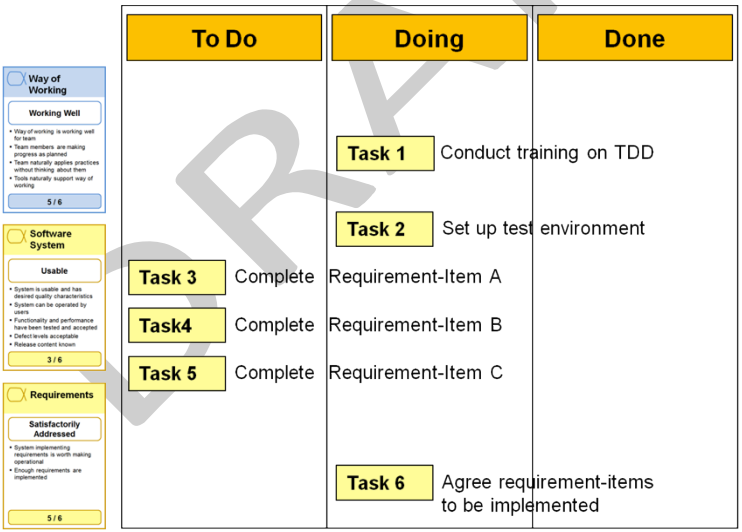


Figure 24 Doing the iteration – Day 1

On day 1 of the iteration, Tom conducted training on TDD for his teammates (see Task 1 in Figure 24), Dick and Harriet. Tom

showed how to identify good test cases, how to write the test code, and he answered the questions Dick and Harriet had. Tom also set up a test environment (see Task 2 in Figure 24). Smith discussed with Angela the requirement-items to be implemented to achieve the **Requirements**: *Satisfactorily Addressed* state.

Doing iteration – Day 2. Figure 25 shows the iteration state on Day 2. Task 1 (conducting TDD training) was completed, but that did not mean that the **Way of Working** was *Working Well*. Both Dick and Harriet needed to be fully up to speed on TDD before the **Way of Working**: *Working Well* state could be considered achieved. Task 6 (Agree requirement-items to be implemented) was completed. This resulted in 3 new tasks added to the "To Do" column. Each of these tasks was about completing another requirement-item.

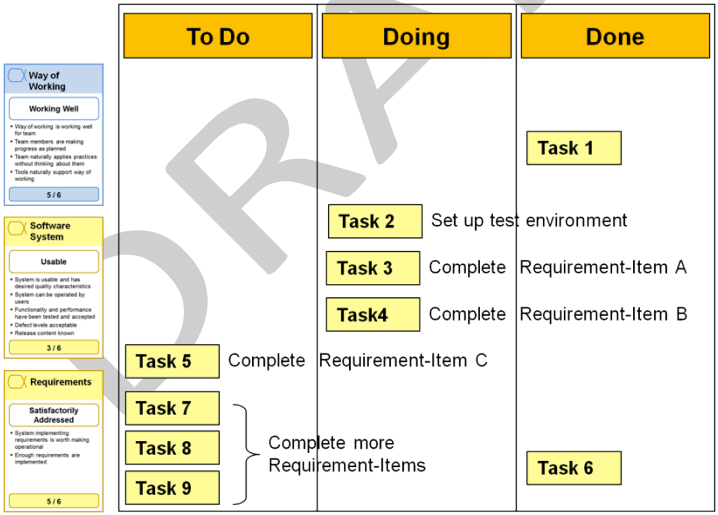


Figure 25 Doing the iteration – Day 2

On day 3 of the iteration, Tom continued to set up the test environment (Task 2). Dick worked on Task 3, and Harriet

worked on Task 4. Smith was assisting the other team members in their respective tasks.

Doing iteration – Day 4. Figure 26 shows the state of the iteration on day 4. Now, Tom had completed setting up the test environment (see Task 2). After working two days on Task 3, and 4, both Dick and Harriet had completed their requirement-items. Angela had reviewed the results from the end-users point of view. Smith had also reviewed it from the technical point of view. Both Dick and Harriet had successfully applied TDD. Their test cases and test code met his standards. As such, the **Way of Working**: *Working Well* was achieved, and its state card was moved to the "Done" column.

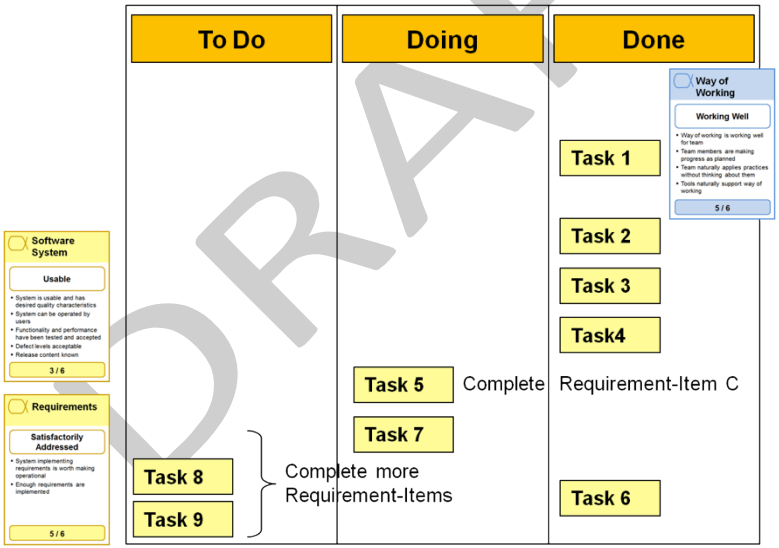


Figure 26 Doing the iteration – Day 4

Side Bar 7 The team chooses its own terminology
The **Work** Alpha uses the work-item term to refer to specific

pieces of work the team does. Many teams use the term task to refer to their work-items similar to the team in our story. It is anticipated that teams will extend the kernel glossary with their own terms, or utilize synonyms that fit their situation. However, teams should be careful not to change the intent of a term that is part of the kernel.

On day 4 of the iteration, Tom and Dick worked on Task 5 (Complete Requirement-Item C). Smith and Harriet worked on Task 7 (Complete a requirement-item, which Smith and Angela agreed to recently on day 1).

Doing iteration – Day 6. Figure 27 shows the iteration state 2 days later on day 6. Task 5 was completed. Now all requirement-items which contributed to the **Software System**: *Useable* state were implemented and reviewed. Having met all the criteria (checklist items), the team had now achieved the **Software System**: *Useable* state. The team moved the associated state card to the "Done" column.

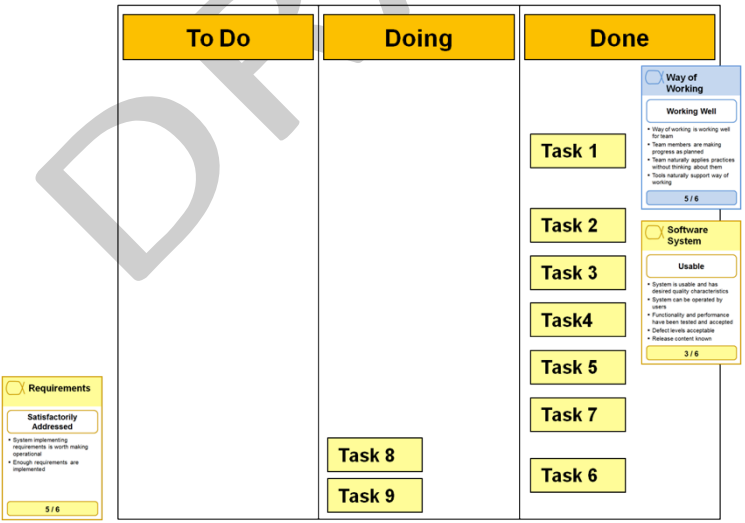


Figure 27 Doing the iteration – Day 6

Now, on day 6 of the iteration, the team was working on the remaining requirement items through Task 8 and 9.

8.3 How the kernel helps you in doing the iteration

Maintaining focus is essential to achieving the agreed iteration objectives. Effective teams know when to say no to unnecessary work, and are always focusing on what it means to achieve "done" in their work. The kernel alpha states with their criteria help the team to maintain focus:

1. **Definition of Done –** The target alpha state criteria help your team to understand and to agree on the definition of done.
2. **Task Prioritization –** If your team is given an additional task, you look at the target alpha state criteria to determine if it is something you have to do now, or something that should be postponed. For example, if it is directly related to achieving a targeted alpha state, you may want to prioritize it. If it is a task related to achieving some later alpha state, you should postpone it.
3. **Reminder for Missing Task –** Every day, your team will look at the criteria for their current target states. Ask yourself, "Do we have a task to achieve this state?" If no such task exists, or the criteria are not met well, there is a missing task. So, you create a task and put it in whatever tool you are using to track your tasks.

9 Adapting the way of working

The kernel captures the essence of software engineering and encompasses sound principles – it encourages customer representatives to work closely with the development team; it reminds team members to always be thinking about risk and focusing on the most important things based on where you are and where you need to go next. When used to run iterations, the kernel serves as a reminder as to what the team needs to do, and helps teams focus. Still, the kernel does not describe everything a team does. There will be gaps. After running a few iterations, a team will recognize better ways of doing things. This chapter looks at how the kernel helps you adapt your way of working to improve or to suit you better.

9.1 Adapting the way of working with the kernel

Modern development approaches suggests retrospectives as a way to improve your way of working. Teams normally do this at the end of each iteration. A typical way to run a retrospective is to have your team ask the following questions:

1. What went well?
2. What did not go well?
3. What can be done better?

The idea is to get team members to talk about their way of working and improve it. The way of working affects all team members, and every team member can contribute to it. This is another area where the kernel becomes useful. The kernel captures the essence of everyone's way of working, but it doesn't require any particular way of working. In earlier chapters in this book we have demonstrated how Smith's team moved through development by progressing through the states. How you progress each alpha and how you achieve each alpha state are subjects for improvement. By talking about the states that a team is progressing through, the team can make the idea of the states more concrete and they can improve how they move forward.

9.2 Adapting the way of working in the story

Let's look at how Smith and his team run a retrospective at the end of the iteration. Smith followed the usual way to conduct retrospectives and asked following questions:

1. What went well?
2. What did not go well?
3. Where can be done better?

At first, both Dick and Harriet were a little lost. Then, Dick said, "Our application does what it needs to, but the user experience is not good. Downloading is a little slow."

Smith realized that Dick needed a little coaching. Dick was talking about the product, not their way of working. Smith tried to make the discussion more focused on the way of working by putting up the target alpha states of the iteration (see Figure 28).

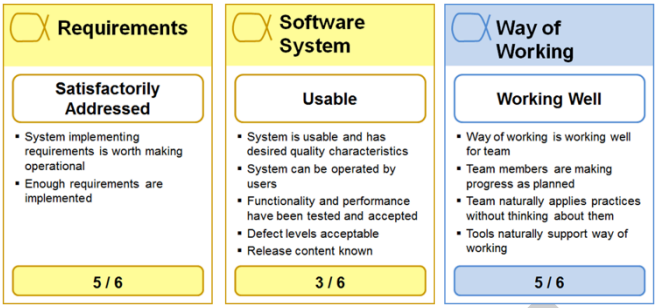


Figure 28 Conducting retrospectives by keeping alpha states visible

Smith then asked:

1. What went well with our planning and doing related to the above alpha states?
2. What did not go well with our planning and doing related to the above alpha states?
3. What can we do better with our planning and doing related to the above alpha states?

Dick said, "Having Tom conduct TDD training for us went well."

Tom said, "The way you achieve the **Requirements:** *Satisfactorily Addressed* state seemed like magic. I learned that I had to ask Angela to agree to the requirement-items to be implemented. Just by looking at the state criteria, I wouldn't have known that I needed to do that."

Harriet said, "Actually, Smith, for me to do my job better I don't need as much help on how to achieve the next state for **Requirements**, **Software System**, or **Way of Working**, as I do with **Work.** For me, it would be better, if I had guidance regarding how to work on a requirement-item."

Smith considered Tom's request. That was easy, all Smith had to

do was to simply supplement the state criteria with some additional guidance. He scribbled two lines of text onto the card as follows (see Figure 29):

- Gain Agreement on requirement-items that fall within scope of *Satisfactorily Addressed*.
- Implement these requirement items.

These added notes were intended as guidance on how to go about achieving the *Satisfactorily Addressed* state checklist items.

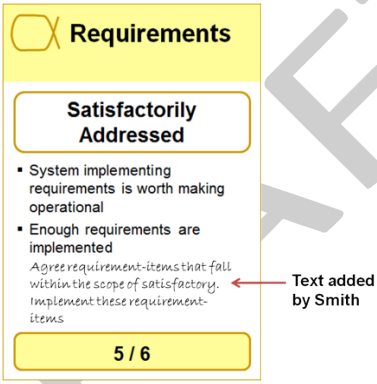


Figure 29 Guidance on achieving a state

Then Smith considered Harriet's request. He liked it, but this request was not as easy. It meant making the way of working on requirement-items more explicit. We will discuss how to extend the kernel with explicit practices in the next chapter.

9.3 How the kernel helps you in adapting the way of working

The kernel helps you adapt your way of working in multiple ways.

Making the way of working explicit – First, it allows you to make your way of working more explicit.

Developers who come straight from the university often know more about programming than software development and more about software development than improving software development. So, their experience is limited and they need more help. For example, it is not uncommon for inexperienced developers to confuse "product" reviews and "process" reviews. Product reviews are about what is good, not so good, and what can be done better with the product. Retrospectives are about what is good, not so good, and what can be done better with the process (way of working). Another problem we observe is that teams often do not know where to start when discussing improvements. They are more familiar with doing work, than thinking about how they work. In general, they have difficulty to move from the "product world" to the "process world". Here the alphas (and the states) become important tools.

When conducting retrospectives, we make the alpha states visible to the team members to help them think about the "process" as opposed to the "product". If you are conducting an iteration retrospective, you only need to make visible those states relevant to the current iteration (i.e. the target states for the iteration). In this way, team members are not overwhelmed.

By visualizing the states, a mental transition takes place. The team is now looking at the process, and thinking about how they are doing at achieving the next set of target states. We then look at each state specifically, and ask the same questions:

1. What went well in our effort to achieve this alpha state?
2. What did not go well in our effort to achieve this alpha state?

3. What can we do better in our effort to achieve this alpha state?

Earlier we discussed how you can clarify and provide guidance on achieving an alpha state. You can also take this opportunity to review these changes. In this way, a retrospective is a process review rather than a product or architecture review.

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 extending the alpha state criteria to meet your team's needs, or modifying previously extended kernel elements. Alternatively, teams can define new alphas. Such additions make the way of working more explicit, and hence provide better guidance to team members, and help team members gain a consensus regarding what they need to do.

Keep in mind that the team should not modify the essential kernel elements or the base kernel checklist items and their definitions. This would undermine the value that we gain through the wide acceptance of a common ground kernel that is taught in the universities and students can count on as they move into industry, or move on from one software endeavor to another.

10 Running an iteration with explicit requirement-item states

The kernel alphas are very useful to drive development across the entire endeavor. The kernel helps development teams see the big picture. We have demonstrated its usefulness in the preceding chapters. However, some developers, like Harriet in our story, have jobs primarily working on requirement-items. In such cases, they may need more guidance on how to work on requirement-items.

10.1 Making requirement-items explicit

As mentioned in the previous chapter, you can adapt your way of working by introducing new alphas. Actually, a Requirement-Item is a kind of alpha just like Requirements and Software System. The difference is that this alpha is not considered universal so it does not belong to the kernel. Instead, it is an alpha introduced outside the kernel to provide guidance on how your team progresses requirement-items, through its states. These states would be defined by a practice outside the kernel. How a team extends the kernel with their specific practices is explained later in the book.

One possible definition for requirement-items state progressions is as shown in Figure 30.

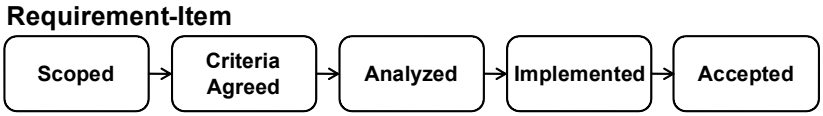


Figure 30 Requirement-item states

1. **Scoped –** Your team (including a customer representative) first scopes each requirement-item to be realized within a time period. For example within an iteration.
2. **Acceptance Agreed –** For each requirement-item, the team agrees on the acceptance criteria for the requirement-item. This helps the customer representative to know what he/she will be expecting at the end of the iteration.
3. **Analyzed –** Your team figures out the best way to realize the requirement-item to reduce any negative impact it will have on the software system.
4. **Implemented –** With a clear understanding of the scope, acceptance criteria, and impact on the software system, you now implement the requirement item (i.e. writing code).
5. **Accepted –** Finally, your team (with the customer representative) reviews and accepts the implementation based on the agreed scope and acceptance criteria. This signifies that you have completed work on the requirement-item.

You can also express the criteria of each Requirement-Item state concisely on a state card (see Figure 31). Each card in Figure 31 describes the key criteria related to what it means to reach a particular state.

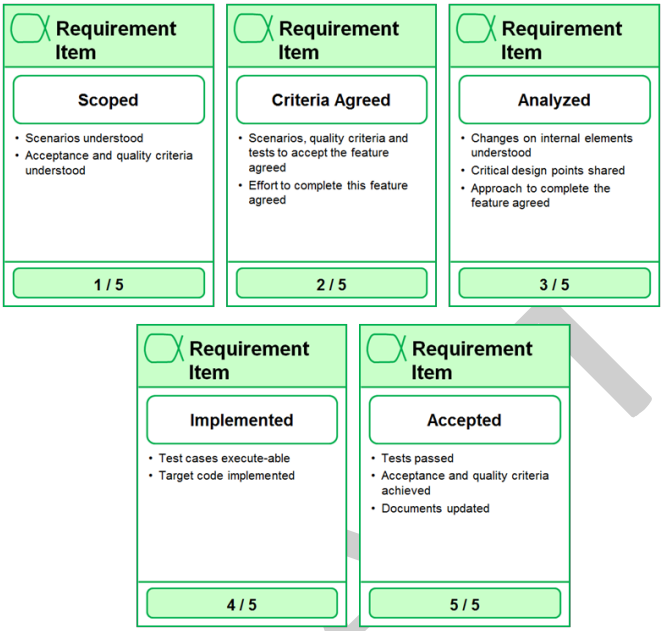


Figure 31 Requirement-Item states

Requirement-Item states are very useful especially when your requirement-items take more than a few days to complete and your team needs more detailed tracking than what we discussed in the previous chapter where tasks were only tracked to the three states of "To Do", "Doing" and "Done".

10.2 Visualizing requirement-item states with Kanban

A Kanban is a technique to visualize the current state of requirement-items. Figure 32 shows one possible way of using a Kanban. It is based on the Requirement-Item alpha states we

discussed earlier (see Figure 30). This Kanban has major columns, one for each Requirement-Item alpha state, and each major column is divided into two sub-columns representing "Doing" and "Done. For example, in Figure 32, the team is working toward getting Requirement-Item A to the *Analyzed* state. The team has gotten Requirement-Item B into the *Criteria Agreed* state and Requirement-Item C into the *Scoped* state.

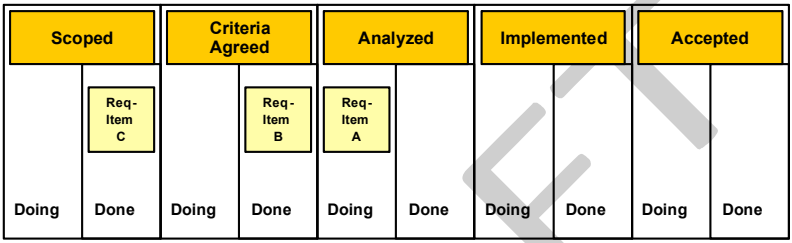


Figure 32 A Kanban showing work-in-progress for Requirement-Items

Working with the Kanban is similar to the task board. Each day, your team comes together and reviews progress and what your team wants to do next. If a requirement-item has achieved a target state, you move it to the "Done" column for that state. If you have started progressing a requirement-item to a certain state, you put it in the "Doing" column for that state.

Side Bar 8 Alphas, individuals and interactions

Recall back in Chapter 7 we explained the value of tracking tasks that may not directly result in a tangible product like tested code. Similarly, notice here that not all of the states within the requirements-item alpha result in a tangible product. For example, *Criteria Agreed* requires conducting a discussion with a key

stakeholder (Angela) to ensure agreement exists on how the requirement-item will be verified.

10.3 Planning an iteration in our story

Let's look at how Smith and his team run their iteration using explicit requirement-item states and Kanban. Our telling of the story here will not be as detailed as in the preceding chapters because the basic idea of plan-do-adapt will still be the same.

Smith and his team had already created a *Useable* **Software System** and were now incrementally adding requirement-items. By now, the members of Smith's team were all familiar with the kernel alpha states. They determine the current state of development and the next states, just as we have described previously. Based on their discussion, they needed to get to the following kernel alpha states:

1. **Requirements**: *Fulfilled*
2. **Software System**: *Ready*
3. **Way of Working**: *Working Well*

They have a number of requirement-items which they need to complete, including:

R1 Browse news feed offline.

R2 Synchronize contents between device and social network.

R3 Set synchronization and download policy.

There are several other requirement items which for brevity we will not discuss (e.g. R4, R5).

The team recognized that once all the requirement-items were

completed (i.e. reached the *Accepted* state), the development would deem to achieve the states **Requirements:** *Fulfilled,* and **Software System**: *Ready*. In addition, if they progressed any of these requirement-item to the final *Accepted* state, they would have validated the fact that they were able to run iterations with the requirement-item and the Kanban technique. For brevity, we will assume that no other tasks were needed.

10.4 Doing an iteration in our story

Smith set up the Kanban but with the state cards made visible to the team (see Figure 33).

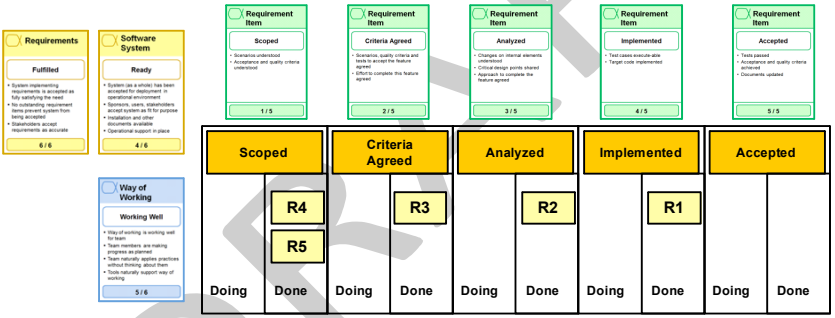


Figure 33 Smith's Kanban with associated state cards

The key differences with Smith's Kanban were twofold:

1. Smith placed the target alpha states on the left of the Kanban to help his team focus on the big picture – the entire development.

2. Smith placed the requirement-item state cards on top of each state in the Kanban. This helped team members familiarize themselves with the definition of done for each

state.

Figure 33 also shows the current state of each requirement-item: R1, R2, etc. at the beginning of this iteration.

After running the iteration, some requirement items were completed. This meant that they have achieved the **Way of Working**: *Working Well* state. But a number of requirement-items had not been completed. They had to be continued in the next iteration. The team agreed that they should get Angela's involvement in the requirement-items that had not been completed.

10.5 Adapting the way of working in our story

At the end of the iteration, Smith facilitated a retrospective. At this time, the team was attempting to achieve the **Requirements**: *Fulfilled* and **Software System**: *Ready* state. Smith highlighted the alpha state used in this iteration (see Figure 34).

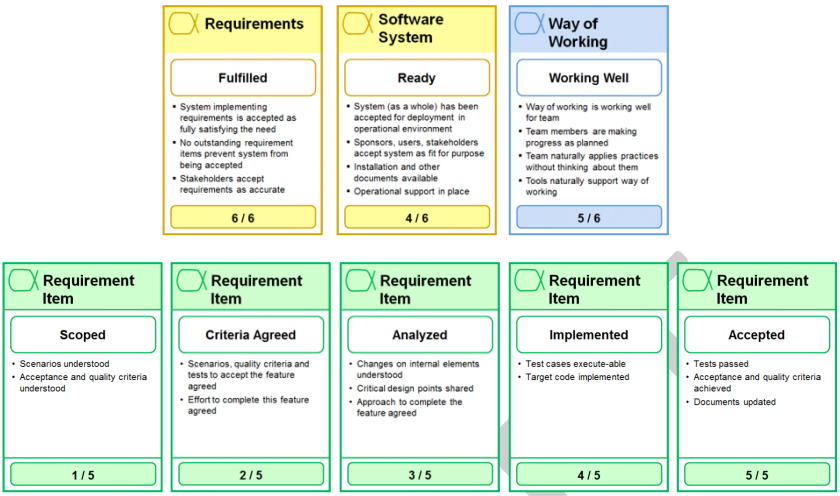


Figure 34 Conducting retrospectives keeping the states visible

As before, Smith asked:

1. What went well with our planning and doing related to the above alpha states?
2. What did not go well with our planning and doing related to the above alpha states?
3. What can we do better with our planning and doing related to the above alpha states?

By now, the team was able to distinguish between the product reviews and process reviews as we discussed in the previous chapter. Dick said, "When getting to the **Requirement-Item**: *Accepted* state, we often find that Angela was looking for something slightly different. It is not easy to get her acceptance."

After some discussion, they agreed that they should get more involvement of Angela when working toward the second Requirement-Item state, *Criteria Agreed*. After the retrospective,

Smith took the feedback to Angela, and she agreed to work closer with the team when working toward this state.

10.6 Discussion

In this chapter you have seen an example demonstrating how the requirement-item checklists can help a developer stay focused on the most important things when working on specific requirements items. You also saw an example demonstrating how keeping the requirement-item states visible to the team can help you track your progress and can help you see where your way of working might be improved.

Part 3 – Using the kernel to run a software endeavor

In the previous part of the book, we discussed how Smith's team ran their development across multiple iterations. In this part of the book we will demonstrate through a story how you can run a software endeavor through its full lifecycle from start to finish. Our goal is to help you gain a better understanding of the kernel, how it helps an individual developer, and how it helps to solve common mistakes.

11 Running a software endeavor: From idea to Product

Smith's specific story began when he was given some initial requirements from Angela, his customer representative. This story is divided into three main phases represented by arrows in Figure 35. The thick vertical lines represent key targets that had to be achieved by the end of each phase.

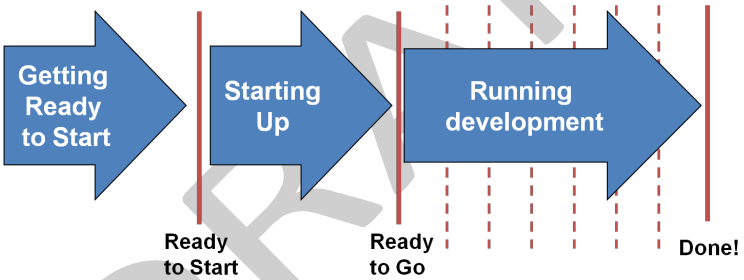


Figure 35 Running a software endeavor: from idea to product

Getting Ready to Start – Smith's first phase starts with Angela, the customer representative in our story, coming up with a product idea. This created an opportunity. With an agreement amongst stakeholders, development could proceed. This was when Smith was brought into the picture. This brought the entire development to a "Ready to Start" state.

Starting up – Smith knew that if development was to be successful, it had to start on the right foot. Thus, Smith in his second phase included:

1. Finding an appropriate way to work together.
2. Drafting an initial plan to set his team off on the right path.

Running development – Now, Smith's team was "Ready to Go". In the third phase, he ran his development iteratively through a series of plan-do-adapt cycles. The iterations are represented by dashed vertical lines in Figure 35. Each cycle achieves some progress, which can be described as a series of kernel alpha state progressions (see Figure 36).

Figure 36 is compact. We will expand each phase in the subsequent chapters.

Side Bar 9 The kernel is life cycle agnostic

This case study assumes your team has chosen to use an iterative process. Keep in mind this is just one example. The kernel approach does not require an iterative life cycle.

This book will discuss how the kernel works with different lifecycles in Section 17.3 when we discuss scaling.

11.1 Dealing with challenges along the way

Smith's job was not all plain sailing. His teammate, Tom, was a hard core coder. He lived and breathed code. Thanks to Tom's keen programming sense, he could solve almost any technical problem, but Tom, could be a little myopic at times by jumping straight into code and losing sight of the big picture.

Figure 36 State progressions across the story

Then there was Angela, their customer representative, who could not quite understand how software is developed. She always thought that software was soft and developers could mold it into any shape she wanted, and at any time she wanted. She didn't understand or care about architecture.

Fortunately, both Tom and Angela did indeed respect Smith and were open to his suggestions and advice, at least to discuss them in an objective manner. Smith's two new teammates were a little more difficult. Dick having come from a traditional "waterfall" development background liked to be told exactly what to do. Harriet was inexperienced having just finished school so she needed a little more hand holding.

Of course, we understand that your situation is likely to be different from Smith's, but we do hope that the way Smith overcame his challenges can shed light on how you can overcome yours.

12Getting ready to start

In this chapter, we will walk through Smith's first development phase: Getting Ready to Start.

Side Bar 10 The kernel supports everyone's story

Keep in mind that the "phases" we are talking about here relate to Smith's story. These are not "phases" that are part of the kernel, or the essentials of software engineering. Your phases could be different. We use this story with its phases just as an example to demonstrate how the kernel can support a typical software endeavor.

It is important to realize that good software is about more than just getting something to work, or even work well. Good software must first of all be useful. So, it is not just about writing good and high quality code, it is about solving problems from the standpoint of many stakeholders (e.g. users, customers). Analyzing what stakeholders need often happens outside what coders and testers see. Stakeholder representatives do this analysis providing inputs to the development team. This is part of what Smith's "Getting Ready to Start" is all about. It also includes getting the team that will develop the software ready to start and preparing the work they will do. The kernel represents this succinctly through a set of kernel alpha states highlighted in Figure 37. It involves achieving specific states of the opportunity, stakeholders, requirements, team and work alphas as highlighted in Figure 33.

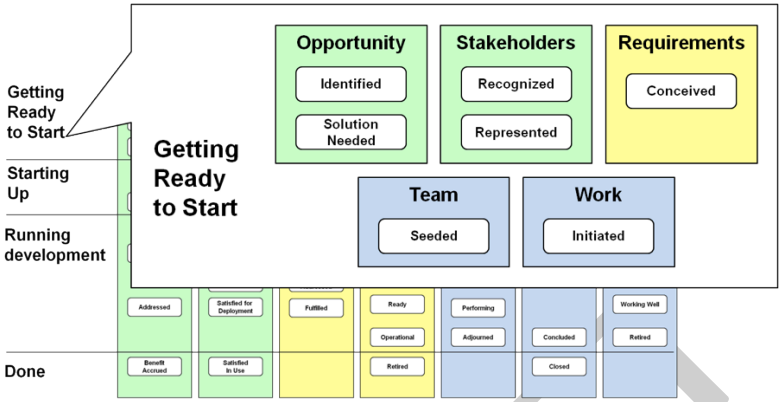
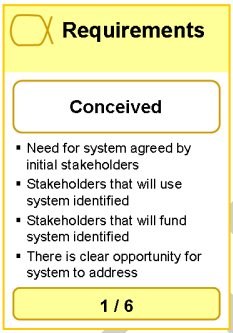


Figure 37 Getting ready to start

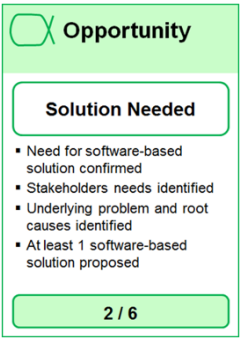
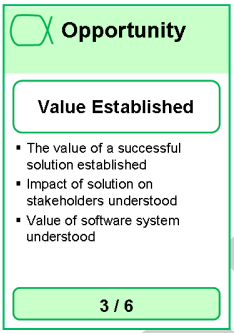
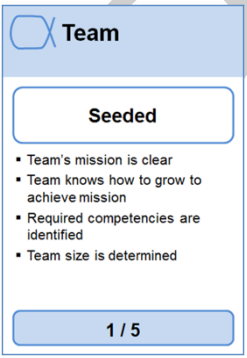
12.1 Getting ready to start in our story

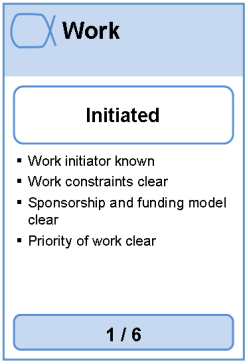
Let's look at how our story achieved "Ready to Start". The text below is formatted into two columns. The left hand side shows an alpha state, and the key criteria (also referred to as checklist items) to achieve that particular state. The right hand side describes what the people in our story did to achieve that particular state. For brevity, we will not talk about all checklist items, but in reality it is useful to consider all checklist items identified in the kernel.

Target State	How the story achieves the target state

Target State	How the story achieves the target state
	**Generate Idea –** Angela's job took her to many parts of the world. While in a restaurant, Angela tried to show her friends some photos from her Facebook (Social Network App), but there was no connection. She was frustrated. Having offline access to your social network is important. Angela believed that if a mobile application could have such a capability, it would be a differentiator to many users. She saw a potential opportunity.
	**Summarize requirements –** Through her research Angela discovered that most social network apps assume a reliable network connection. So, there was a real business opportunity. Angela works in product planning / marketing in a phone company. She wrote a brief overview describing the application including who the expected users and customers would be, and how this application could benefit them.

Target State	How the story achieves the target state
	**Involve marketing department –** Angela could not simply tell developers in her company to work immediately on her idea. She had to get the agreement of her colleagues in her department and her boss (stakeholders). Together, they would consider her idea from the perspective of different kinds of users, and evaluate if her idea was worth a shot. She was working with some of the stakeholders who were responsible to decide if her idea would be given the go-ahead.
	**Run product planning meetings –** Angela's department had regular monthly product planning meetings where they would discuss product ideas. In this way, everyone involved would get their ideas and comments heard and debated. This was how Angela's department ensured that everyone's views were represented. Those invited to the monthly product planning meetings had been given the responsibility and authority to review and make decisions on the direction the product would take.

Target State	How the story achieves the target state
	**Agree solution –** During the following product planning meeting, Angela presented her idea. She explained how often people find themselves in places without internet connections and have a real need to access their social network. Her colleagues and department head discussed it. They agreed that her idea was a good one and a software based solution could solve the root cause of the problem.
	**Agree business case –** Angela's department discussed her idea further. They agreed that it would be a valuable differentiator compared to other similar applications. Moreover, it would be well positioned for their next generation mobile devices, whose target audience was frequent travelers. Her department head, Dave, gave the go-ahead.
	**Assign core members –** Dave then assigned Smith and Tom to work together with Angela who explained the mission to them. Smith and Tom were competent developers who had experience in mobile application development. Dave told Smith if he needed more team members to achieve the mission to let him know.

Target State	How the story achieves the target state
	**Kick off development –** Angela and Dave shared with Smith and Tom, what the opportunity was about, and what the requirements (at a high level) were. With that, development started. Even though Angela would be very much involved, the ball was in Smith and Tom's court. Smith and Tom were now ready to start.

12.2 How the kernel helps you get started

In this chapter, we have demonstrated how the kernel helps you get started. It is important to get the stakeholders, including those that represent the customer, and developers on the same page with shared understanding of the product and the responsibilities each of them will be taking on during the software endeavor. Yes, they should all put their heads together to solve problems together as a team, but they also have their separate responsibilities.

As an example, as a stakeholder who represents the user, you have the responsibility to provide developers with something that they can start working with. As a developer, you have responsibility to produce good software. As a developer, you also need to know what you are expected to do, and to know that you are doing something of value. As a developer, you can go through the states and ask your stakeholder who represents the user:

- What is the opportunity?
- Is the solution needed?
- Are the other stakeholder groups that represent the opportunity recognized?

Understanding the above (and the other checklist items in the kernel) is the first step to get convergence on the requirements for the software system.

The above story shows a happy case. Things proceeded well. There was already a mechanism in place for Angela to discuss and refine her ideas. The stakeholders were very cooperative.

What if you do not have such a mechanism to discuss ideas? Then set one up! And you can do this by getting the right people involved. You help them understand what they need to do through the kernel alpha states mentioned above.

What are the challenges that might occur here? A few examples include:

- Idea not clear?
- Stakeholders not involved?
- Solution not clear?
- Requirements not clear?

The kernel helps you by highlighting what questions to ask and what kinds of actions you could take to keep your effort on track. It also helps you stay focused on the most important things to do now to progress from where you are to where you need to go next with your software endeavor.

13Starting Up

In this chapter, we will walk through Smith's second phase of development: "Starting Up". Through what we discussed in the previous chapter, we have already started well with the opportunity, and stakeholders. This often happens outside the team. In Smith's "Starting Up", it is about starting right from within the development team itself. This means having a good way of working, having a good grasp of both requirements and architecture and also a plan for the overall development.

Now, this does not mean heavyweight documentation. It means that the development team needs to understand where they are heading, and how they will conduct development effectively.

Starting up involves achieving the set of kernel alpha states highlighted in Figure 38. The team needs to understand the requirements scope and the technical complexity of the software system.

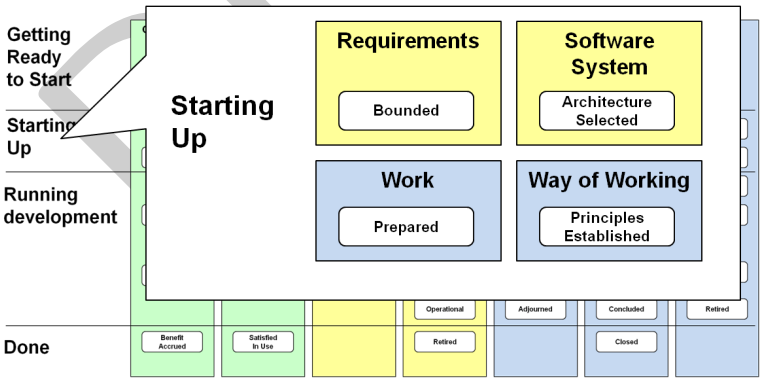
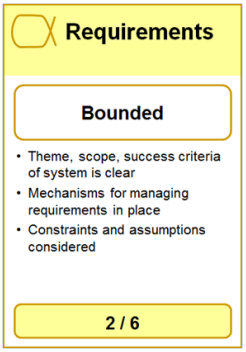
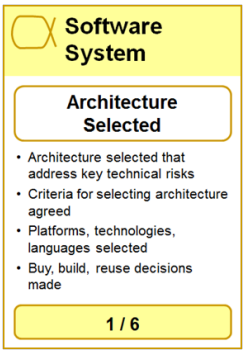
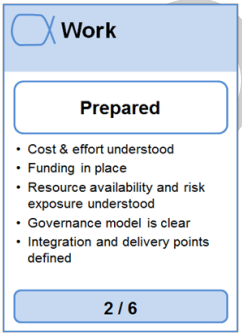


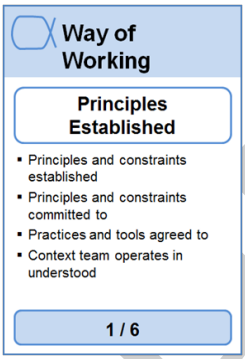
Figure 38 Getting Started

13.1 Starting up in our story

Let's run through how Smith and Tom start up the development.

Target State	How the story achieved the target state
	**Conduct requirements workshop** – Angela gave an overview of what the application was about to Smith and Tom. They brainstormed the requirement-items needed to have a useful application. The agreed list of requirement-items included: 1. Browse profile (online and offline). 2. Browse news feed (online and offline). 3. Browse album (online and offline). 4. Browse videos (online and offline). 5. Post comment (online and offline) 6. Synchronize contents between device and social network. 7. Set synchronization and download policy. This list became the agreed to success criteria bounding the scope of the work.

Target State	How the story achieved the target state
	**Analyze architecture –** Smith and Tom started investigating the architecture for their application. The application was small, but still there were some architecture issues to iron out, namely: 1. The caching mechanism (how much to cache, and how to cache), 2. The caching policy (when to update cache, what to cache, etc.). 3. How to minimize the changes to the existing implementation. They already had an application that could browse social networks in an online mode. Through discussion and investigation, they found some candidate solutions and agreed on which solution they preferred based on key technical risks.
	**Choose an appropriate lifecycle –** Smith and Tom discussed and agreed that their development should progress incrementally as follows: 1. Produce an early demo. 2. Produce a useable system. 3. Produce a shippable system. 4. Hand over. We will discuss this in more detail in Section 13.2.

Target State	How the story achieved the target state
	Prepare for development. Smith and Tom started to plan their development by allocating requirement-items to the lifecycle. This helped them understand the cost and effort that would be required. Because Dave had already given the go-ahead the funding had been approved and the additional resources that would be needed to work on the requirement-items were being made available. We will discuss this in more detail in Section 13.3.
	**Choose an appropriate way of working –** Smith and Tom agreed that they would run the development using the following practices: 1. Iterative development. 2. Test driven development. 3. Continuous integration. They also selected a collaboration and task management tool that they had used on a previous software endeavor.

13.2 Choose an appropriate lifecycle

Smith's plan for their team involved agreeing on an appropriate lifecycle with Angela. They did this by assigning Requirements and Software System alpha states to some milestones (see Figure 39).

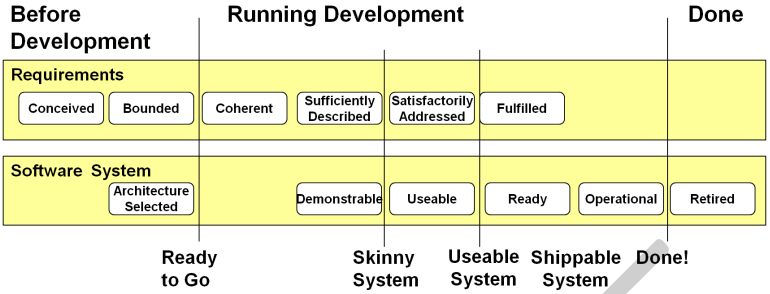


Figure 39 Development lifecycle based on Requirements and Software System alphas

The milestones agreed to were as follows:

1. **Ready to go –** At this time, Smith had already achieved the Ready to go milestone. We discussed this in the preceding chapter.

2. **Skinny system –** The next milestone is one for Smith's team to validate their architecture and get accurate feedback regarding what Angela wanted. This would validate their solution to address the technical challenges identified when they analyzed the architecture (see above when they achieved the Software System: Architecture Selected state). The stakeholders will use this "skinny system" to understand the application better and provide feedback on the application requirements.

3. **Useable system –** The next milestone is a minimal shippable system. This is a system that users can use, but without the full benefits of the opportunity. This system would be of high quality, and hence potentially shippable.

4. **Shippable system –** The next milestone that Smith's team would produce is a shippable system. It would have high quality (i.e. little or no defects) and would have sufficient

requirements implemented to be acceptable by the stakeholders.

5. **Done –** Finally, Smith could declare that his team had completed the work and hand the software system over.

13.3 Prepare for development

Smith's team discussed how they would come up with a first high level plan for the development. The plan would comprise the following:

1. Which requirement-items, on a high level, should be implemented (and to what extent) by which lifecycle milestone?
2. Which architecture issues should be resolved by which lifecycle milestone?

The next immediate milestone was to develop a skinny system. After discussing with Angela, they agreed on the following scope for the skinny system:

- Architecture issues to be resolved:
  1. caching mechanism
  2. caching policy
  3. minimal code changes
- Requirement-items to be demonstrated:
  1. Browse profile offline.
  2. Browse pictures offline.
  3. Synchronize contents between device and social network

The chosen requirement-items were referred to as architecturally significant requirement items by the team in our story because once they ran successfully, there would be sufficient evidence that the architecture issues were indeed resolved. Angela, Smith and Tom agreed that the useable software system would implement the following requirement-items.

1. Browse profile offline.
2. Browse pictures offline.
3. Synchronize contents between device and social network
4. Synchronize contents between device and social network.
5. Set synchronization and download policy.

Angela and Smith agreed that allocating the above to the next two immediate milestones would be sufficient to get the team going. They would refine the plan as development proceeds.

13.4 How the kernel helps you in starting up

In this chapter, we have demonstrated two important uses of the kernel.

Development Lifecycle – There is no single lifecycle that fits all development endeavors. With the kernel, you have the building blocks to define your own lifecycle. The building blocks are the alpha states. You define the lifecycle by aligning the alpha states to the lifecycle milestones. In our story, the development lifecycle chosen by our team is one that attacks architecture risks first. There are other lifecycles – such as a traditional waterfall approach where teams want to be very clear about what they need to do first. This may still be appropriate for mission critical systems

because the cost of prototypes is extremely expensive. Defining your lifecycle using the kernel will be discussed a little more in Section 17.3.

The benefit of using the kernel to define your lifecycle is that you can associate progress in different dimensions of software development to the milestone definitions within your lifecycle. In this way, team members are reminded to take the dimensions that are most important to your situation into consideration.

Planning Ahead – One of the greatest challenges observed on many software endeavors is coming up with a realistic plan that has the right level of detail to enable your team to best meet the opportunity in a timely fashion and to get support from the stakeholders. How does the kernel help? The kernel helps you reason about how far you can plan ahead and to what level of detail. By planning, we mean associating the agreed lifecycle milestones with definite achievements (e.g. which requirement-items to be completed and by when).

So, how far can you plan ahead? It all depends on how much information you have. You can easily evaluate this by considering the states of the other alphas. In our story, the requirements have only reached the *bounded* state and hence Smith's team can only plan until the Software System usable state. If Smith's Requirements have reached a further state, he can plan further, even to Requirements Satisfactory state. Whether it is necessary or even desirable to plan ahead to any significant extent is dependent on the business context that the system is developed in. The kernel is supportive of both evolutionary planning and more frontloaded approaches.

14 Running development

In this chapter, we will walk through Smith's third development phase: "Running Development". From the previous two phases (discussed in the preceding chapters), Smith had gotten a good start. His team was ready, but starting well doesn't ensure all will go well during development.

Software development is extremely challenging because as we said in Chapter 1, software development is multi-dimensional – there are risks and challenges attacking you from many different dimensions: opportunity, requirements, software system, etc. At different points in time and on different software endeavors, the emphasis needed on each dimension will be different. Collectively, a development team must have skills across these dimensions, and need to understand the interactions across these dimensions to ensure each gets appropriate attention.

This is certainly not easy. Because there are many dimensions and they interact, you do not deal with each dimension all at once, or all separately. Instead, you put in place a strategy to deal with a group of them at a time. Smith's strategy to deal with the different dimensions is summarized in Figure 40. Smith's strategy ran development in what we refer to as 7 broad "strokes", each represented by an arrow.

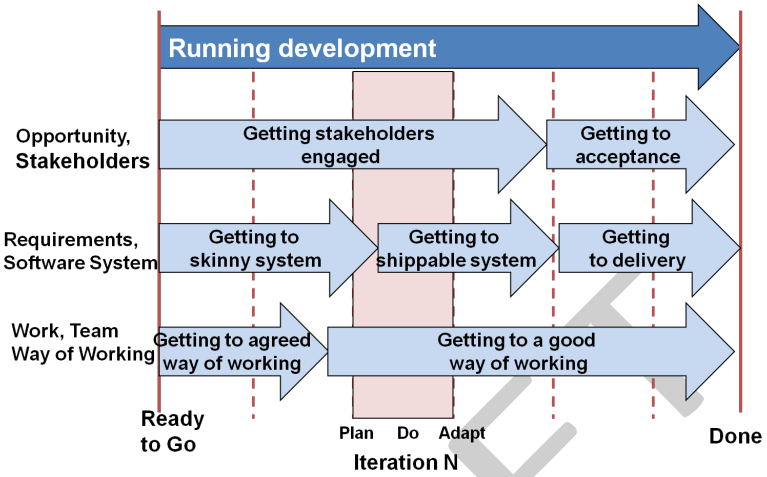


Figure 40 Running development

Each broad stroke is associated with a set of alphas and target states. For example, "Getting to skinny system" will move Requirements to the Coherent state, and Software System to Demonstrable state. It should be understood that getting to the target states in a broad stroke doesn't necessarily coincide with the end of any particular iteration.

Smith's approach to complete the strokes is iterative. The vertical dashed lines in Figure 40 represent the boundary between iterations. A stroke can take more than an iteration to achieve and a stroke can actually be achieved somewhere in the middle of an iteration.

In this chapter, we will look at how Smith's team completes each broad stroke.

1. **Getting to agreed way of working –** At the beginning of development, the team needs to ensure that they have a stable development environment. There is some ramping up to do.

2. **Getting stakeholders engaged –** At the same time, it is important to get stakeholders involved to ensure the team is on the right track.
3. **Getting to good way of working –** The development team needs to get to a steady state, where it runs like a well-oiled machine and continuously improves.
4. **Getting to Skinny System –** With a stable way of working, the development team can focus its attention on producing a skinny system that validates the selected architecture, and has the ability to grow to and become the system to be delivered.
5. **Getting to a shippable system –** Once the architecture is validated, the development team incrementally adds functionality to the system, always ensuring that the system has good quality and is potentially shippable.
6. **Getting to delivery –** The development team has fulfilled a sufficient set of requirements and makes the software system ready for acceptance and deployment.
7. **Getting to acceptance –** In parallel with "Getting to delivery", stakeholders closely monitor development to determine if the software system has reached a point that can be accepted. When the software system is ready, stakeholders that represent the customer and the development team work together to achieve acceptance.
8. **Done! Completing development –** Finally, the software system is handed over for deployment. The team might continue, or the team might adjourn. The development team concludes the development work and runs a final retrospective for the endeavor.

In the following sections, we will walk you through Smith's third development phase: "Running Development", through each of the

broad strokes above that was part of Smith's strategy. These broad strokes are not part of the essence of software engineering. They are the way Smith decided to run his development. The way you decide to run yours will likely be different. It is also likely that everything will not go as smoothly on your development effort since this case is rather idealistic. Keep in mind its purpose is to help you understand how the kernel can help you.

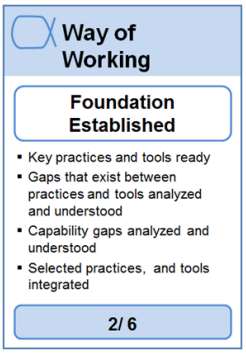
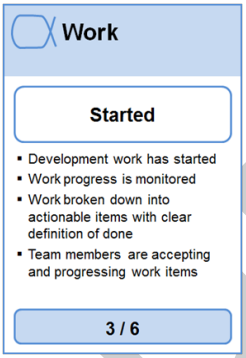
Now, the kernel doesn't help you only when everything goes smoothly on your development effort. In fact, its greatest value can be seen when things don't go so smoothly and you have difficult decisions to make. This is where its greatest benefits can be found.

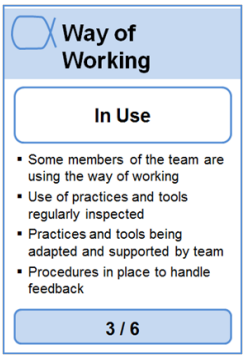
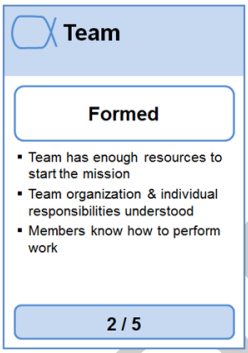
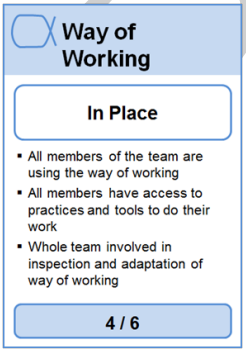
We cannot in this short book possibly explain all the situations you might face and tell you how to use the kernel to help you in each of those situations, but we can provide a few examples to help you learn the idea. These examples are provided in the following sections.

14.1 Getting to agreed way of working

Development has begun! Smith and Tom recognized that they need to ramp up the work and come to a point when they can be productive. This means getting the development environment in place, getting the team members on board and getting them to work well together.

Target State	How the story achieves the target state

Target State	How the story achieves the target state
	**Set up development environment –** Smith and Tom started to set up their development and test environment, which included: 1. A Code Repository 2. An Integrated Development Environment 3. An Application simulation environment 4. Selection of a Continuous integration tool
	In parallel to preparing their way of working, Angela, Smith and Tom also started the development work. Smith and Tom started to analyze the requirement-items, write test cases, and write some code. They started making use of their development environment.

Target State	How the story achieves the target state
	At this point in time Smith and Tom were exercising the work environment by using the tools and working on the agreed to requirement-items through their iterations as they had agreed to.
	The new developers, which Smith had requested, had just joined the team. These two developers were Dick and Harriet. They had different background and experiences. Smith walked through the work principles, practices and key tools to familiarize them with the work environment as well as the requirements for the application.
	Through guidance from both Smith and Tom, both Dick and Harriet got familiar with the team's way of working, the development environment, as well as the practices. And they began using the way of working to do their tasks.

Target State	How the story achieves the target state
	At this point in time, all members are working together, focusing on the team's mission, which is to produce a high quality application to fulfill Angela's initial idea.

What can go wrong in this stroke? How can the kernel help you?

The goal of this stroke is to get the way of working in place, so that each member knows how to collaborate and work together. Coming to agreement on the way of working is not easy. Team members might not be adequately trained, or they may have different backgrounds.

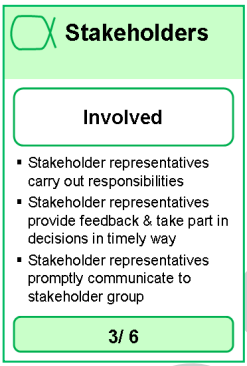
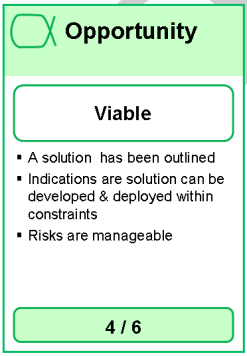
Agreeing on a way of working can be quite philosophical. A good way to resolve this is to walk through the development approach and ask the team if they know what they should do at each step. A good thing about the kernel is that it identifies these steps (through alpha states) for you, and in all key dimensions.

By walking through the development, you can provide an opportunity for the team to act out what will happen (e.g. role-play). Now, if a walkthrough is not enough, then take a simple requirement-item, and do some requirements, design code and test. In this way, you exercise the way of working, highlight issues quickly, and hence have time to fix the issues. Thus, rather than engaging in some methodological debate, the walkthrough can

make your discussions concrete and hence help your team come to an agreement quickly.

14.2 Getting stakeholders engaged

At the same time, when Smith's team is ramping up, it is important to get stakeholders involved to ensure the team is on the right track.

Target State	How the story achieves the target state
	Angela worked with the team rather closely. She made herself available for the team when they had questions to ask. She also made it a point to provide feedback to the team when needed. Angela also communicated the team's progress to the other stakeholders she represented in her department.
	At the beginning of the development, Smith and his team had drafted an initial plan (see Section 13.3). Now, Smith and Angela updated the plan together regularly. Specifically, this resulted in updated priorities of requirement-items. This plan helped to ensure stakeholders that a solution could be produced within the constraints, and that the risks were being managed.

What can go wrong in this stroke? How can the kernel help you?

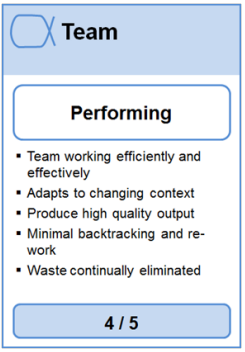
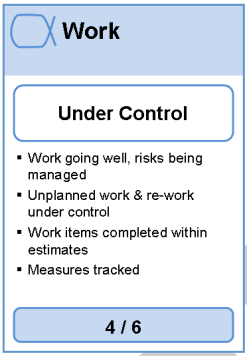
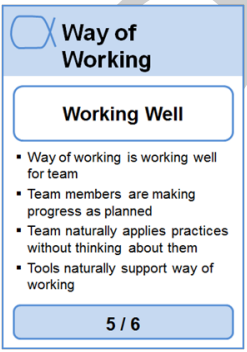
The goal of this broad stroke is to maintain regular contact with the stakeholders as actual development starts. Otherwise, it is very easy for the development team to venture off in the wrong direction.

So, what happens when you go off track? The kernel demands that you have contact with stakeholders when you come to: **Software System:** *Demonstrable* state, and a **Software System:** *Useable* state. Both require stakeholder agreement before proceeding. So, if you do go off-track in some dimension (embodied by their respective alphas), the kernel through another dimension, will help you detect it.

14.3 Getting to a good way of working

With the development environment in place, the team in place, and the stakeholders collaborating, Smith and his team quickly came to a point when they function very well together. Smith and his team were now able to run their iterations relatively smoothly.

Target State	How the story achieves the target state

Target State	How the story achieves the target state
	At this time, the team members are working well. Whenever, there are issues, they would stop their work, come together and resolve the issue efficiently and effectively. They are producing high quality outputs, and they continually eliminate waste when they see it.
	Smith and his team tracked the progress of their requirement-items and the tasks (work items) of each member. They made this visible to all members, including Angela. In this way, everyone on the team had a good idea of how much work they had to do, and how much work was left in each iteration, and for the entire development.
	Smith and his team conducted retrospectives at the end of every iteration to determine what had gone well, what had not gone well and what could be improved. This ensured that their development environment and their agreed practices stayed relevant throughout development.

What can go wrong in this stroke? How can the kernel help you?

This broad stroke attempts to maintain a good way of working throughout development so that the team members can concentrate on their craft – building good software.

What happens if the way of working is not working well? It depends on the degree of difficulties faced. In the best case, it might be just frustration, producing some waste, or accumulating some (small) technical debt, which may not be really delaying the product or resulting in reduced quality in the eyes of the stakeholders. In short, the team could live with this. In the worst case, it might cause fights, delays, and unacceptable quality. The remedy is to work iteratively. Each iteration exercises the way of working and highlights possible areas that are not working well that the development team can fix. In each iteration, the team will assess the state of development with the alphas. For example, by assessing the way of working the team will think about how they are working and if something needs to be fixed. In the previous section, we also highlighted the fact that the kernel alpha states make what they are doing more tangible and hence easier to pinpoint a problem and solve it.

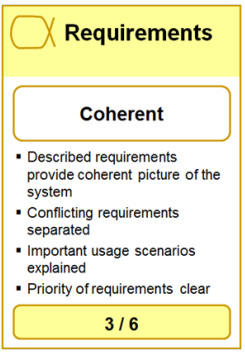
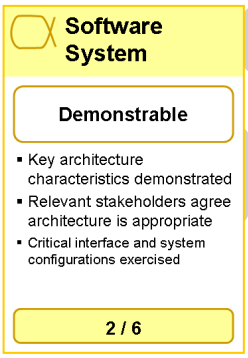
14.4 Getting to skinny system

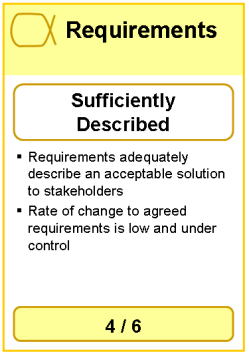
With a stable way of working, Smith and his team members could focus their attention on producing a skinny system. This skinny system would:

1. Implement some key requirement-items
2. Validate the architecture
3. Be able to grow to accommodate other requirement-items

without significant rework

To achieve the above, not only must Smith and his team implement the software system (i.e. write code, write tests, etc.), they had to also work with Angela to nail down the requirements to target for the skinny system.

Target State	How the story achieves the target state
	Smith worked with Angela to nail down the requirements by writing acceptance test cases for them. In doing so, they started to realize that some requirement-items were either duplicated, or similar. So, they re-organized the requirement-items, and prioritized them.
	Meanwhile Tom, Dick and Harriet were implementing the requirements. Tom, a techie-guy by nature was quick to address the technical challenges. Both Smith and Tom made it a point to have all members of the team write test code that tested the software against the key interfaces of their software components. Eventually, they had a version of the system which they could show Angela and the other stakeholder representatives. The stakeholders agreed that the key architecture characteristics were being addressed.

Target State	How the story achieves the target state
	Smith had been fleshing out the requirements and reviewing them with Angela. After a number of changes, Angela agreed the requirements-items were adequate. Both Angela and the team had a clear and agreed idea of the scope of the software system.

What can go wrong in this stroke? How can the kernel help you?

The goal of this stroke is to validate the architecture to ensure that it will address the likely technical challenges, and that it can grow gracefully to incorporate the implementation of the requirements. Smith achieved this through building an initial version of the system called the skinny system that exercised all important characteristics of the system. Since the skinny system was small, it was relatively easy to demonstrate and change. Once the skinny system met its criteria, incorporating the remaining requirement-items was unlikely to cause any major problem.

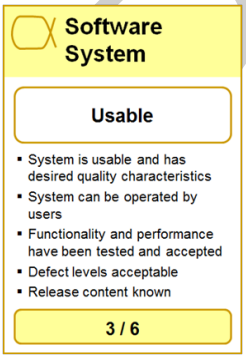
Some teams do not have a good idea of what a skinny system is. They are unable to limit the scope of the skinny system, and thus, they need a fairly large version of the system before they can validate the architecture. In this case, if there are issues with the system (i.e. design flaws), fixing these issues could become costly. This is the advantage of building a skinny system. In the worst case, the team might be validating the architecture for the first

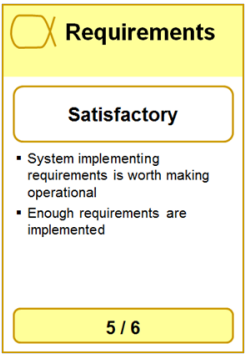
time when it is nearly completed. This is very risky because changes to the architecture this late are more likely to cause delays and dissatisfied stakeholders.

The kernel itself does not provide a concrete way for development teams to build a skinny system, but it reminds the development teams of the importance of doing so. The kernel also indicates that you do not need all requirements to scope the skinny system. You only need to get to the **Requirements**: *Coherent* state to determine what should be in the skinny system.

14.5 Getting to a shippable system

Once the architecture was validated (e.g. skinny system is operating), Smith and his team incrementally added functionality into the system, always ensuring that the system was well tested. This ensured that the system was potentially shippable. The team had decided that with each iteration the quality of the software had to remain high enough so they would always be ready if management decided to ship the product to a customer.

Target State	How the story achieves the target state
	With the architecture validated, Smith and his team worked to get the requirement-items that were demonstrated in the skinny system into a state where they were usable by Angela. After that, they started implementing the remaining requirement-items incrementally getting them tested and accepted. As they did so, they continually paid close attention to quality.

Target State	How the story achieves the target state
	As Smith and his team implement requirement-items into the software system, it grew to a point when Angela agreed that they could start preparing to ship it.

What can go wrong in this stroke? How can the kernel help you?

The goal of this stroke is to produce a software system that can be shipped (i.e. released) to the end-user community. This means that the software system must have sufficient quality and functionality.

There are two primary ways to get here. The traditional approach is to implement all the required functionality first before verifying the quality. The other approach is to continually ensure quality while functionality (i.e. requirement-items) is being incorporated.

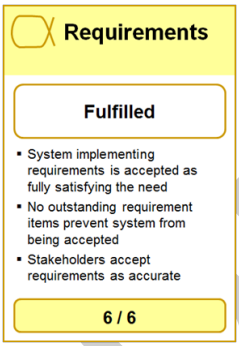
The latter approach allows the development team to accept requirement changes more responsively. In addition, the development team is better prepared to release the system at any time since with this approach we continually test maintaining high quality software throughout development.

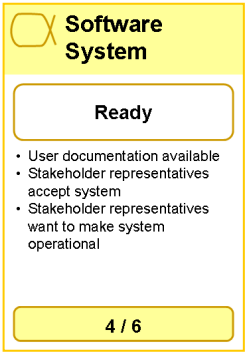
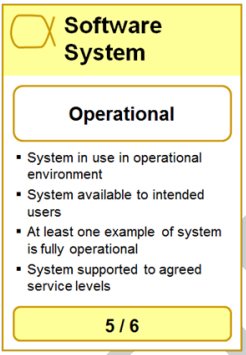
The latter approach also requires the development team and the stakeholder representatives to work together agreeing on the requirements to get the system to where it is worth making operational. Sometimes different stakeholder groups bring

differing perspectives and even possibly conflicting requirements. The kernel can help the team recognize this situation and take appropriate actions to solve it.

14.6 Getting to delivery

Smith and his team incrementally added functionality into the system until a point when Angela deemed that requirements had been fulfilled and the system was ready for acceptance testing. After a successful acceptance test, the system was made available to its users.

Target State	How the story achieves the target state
	Smith and his team implemented the requirement-items gradually into the software system. As they did so, they made it a point to ensure that all defects that would keep the requirements from achieving its fulfilled state were fixed and that Angela was agreeable to what she saw.

Target State	How the story achieves the target state
	After running through acceptance tests, Angela and the other stakeholders agreed that the system was ready. Smith prepared the release notes. Dick and Harriet prepared some help guides for the application
	After the successful acceptance of the system, Smith worked with his colleagues in another department to get their application available on the application market (i.e. the likes of Android Market, and Apple App Store). Angela downloaded the application from the application market and started using it.

What can go wrong in this stroke? How can the kernel help you?

The goal of this stroke is to finish acceptance testing and bring the software system to the end-user community, and in the case of our story, the application market.

Delivering the system can come with a whole range of challenges depending on the kind of development you are doing. For example, if you are building some custom enterprise application, this would mean conducting training, migrating existing data to the

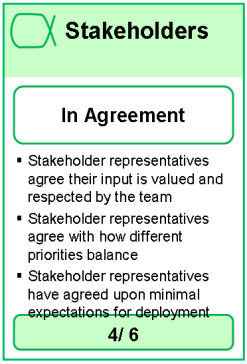
new system, and so on. These would require many other practices, which can be defined on top of the kernel.

Sometimes customer representatives don't let the development team know what is most important to the group they represent. They may want to be collaborative, but may not know how to express a conflicting point of view. The stakeholders alpha and related checklist items can be a good reminder that stakeholder representatives need to provide feedback and take part in decisions in a timely manner. Stakeholder representatives must also communicate changes that are relevant for their stakeholder groups promptly. When stakeholder representatives fail to live up to their responsibilities in a timely way they put the success of the software endeavor at risk. The kernel helps to keep the importance of these responsibilities visible to both the team and its stakeholders.

14.7 Getting to acceptance

In parallel with "getting to delivery" as described in the previous section, Angela watched closely the development to determine if the software system had come to a point where it could be accepted. When the software system was ready, the stakeholders who were representing the customer and the development team worked together to achieve acceptance.

Target State	How the story achieves the target state

Target State	How the story achieves the target state
	Angela was watching the development closely. After a successful demonstration to her and the other stakeholders, she prioritize the requirement-items based on what she heard from all of the stakeholders and worked with Smith to define which requirement-items would make it to the final version of the system. Smith listened to all of Angela's concerns and requested clarification in areas where the team didn't fully understand what was needed.
	Angela worked with the other stakeholders to nail down the criteria for deployment and provided the feedback on the system to the team. Smith and his team made the final finishing touches to the software system. After a successful acceptance test, the stakeholders were satisfied that the system could be deployed.

Target State	How the story achieves the target state
	The initial idea which Angela had was now completely realized. The system had been made available in the application market.

What can go wrong in this stroke? How can the kernel help you?

This final stroke brings the development work to a climax. If you have followed the advice of the kernel, you will have regular contact with the stakeholder representatives, a software system that is demonstrated to be architecturally stable, useable, and potentially shippable as you add requirement-items incrementally. There will be different practices for running acceptance tests. If the system you are developing gets rejected at this point then you have most likely missed something important in your evaluation of the alpha states.

A few questions you could ask yourself to help determine what went wrong, and what needs to be done to get back on course, are:

- Were all the stakeholder representatives really involved, and did they really reach agreement?
- Did the stakeholders accept the requirements as an acceptable solution?
- Did the stakeholders accept that the requirements reflect

what the system actually does?

These are just a few of the questions you could ask to help isolate the root cause. Once the root cause is isolated the next step is to identify the proper course of action to resolve the issue. Just as the kernel helped us identify where we were, and where we needed to go next, it can also be used to troubleshoot a problem and get your software endeavor back on track.

14.8 How the kernel helps you run development

This chapter has shown you how the kernel can help you run development by walking through many of the alphas and their states. Our focus has been the usefulness of the kernel which includes several aspects:

1. It defines intermediate points to help a team focus on their immediate next step.
2. It considers the different dimensions of software development so that the development team may not lose sight of any risk or challenge and get caught unaware.
3. It allows you to choose your way to achieve the states. This gives you flexibility in your situation. It is not rigid.

Keep in mind the strokes are not part of the kernel. You can define your own strokes that make sense for your endeavor. The alphas and the states are reminders to help you achieve your goal without missing anything essential, but they don't require you to follow any specific methodology.

Part 4 – Scaling development with the kernel

The kernel contains the essential elements of software development. The alphas and their states provide guidance and reminders to help a team to plan and do their work and adapt their way of working. We have shown in Part 2 and 3 how a small development teams can use the kernel to drive development. We have assumed that the team members are competent, and that they can determine what to do by referring to the alpha state criteria.

You are probably asking: will this work for all teams? Will this work for all kinds of development, big and small, co-located or distributed teams, in-house, and outsourced development? In short, does the kernel scale?

What does it mean to scale? Can we be systematic about scaling? This part of the book will discuss different dimensions of scaling and how practices on top of the kernel will help you address the challenges of more complex and large development.

15What does it mean to scale?

The kernel contains the essential elements of software engineering, the common ground. As such, by definition it does not contain everything that a team needs to know or do. There are of course gaps between what is in the kernel versus what a team does. Nevertheless, it is a very good starting point to scale to any kind of software endeavor. Different people have different ideas as to what it means to scale. So, we will spend a little time discussing what scaling means. In Figure 41, we show different dimensions of scaling and we will discuss them from the bottom up.

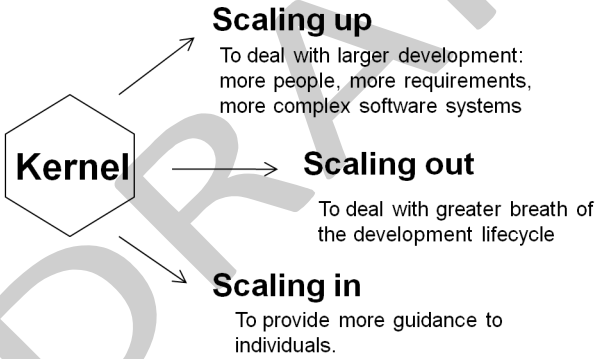


Figure 41 Dimensions of Scaling.

Scaling In – We call the first dimension "scaling in". Scaling "in" is about diving "in" to the details beyond what the kernel provides. If the team members are competent and have common background and experience, they can usually fill in the details by describing the way they would use practices. In the story in Part 3, these practices were left tacit and they only existed in the

conversations between team members. However, this does not work in large development. You wouldn't want to repeat telling different team members how to apply the practices. It would be good to have something handy for which team members, especially new members, to refer to. In addition, it is also important for team members to understand how the selected practices fit together.

Scaling Out – The second dimension is "scaling out". Scaling out is about balancing the emphasis across the software development lifecycle. For example, during the early stages of development, the emphasis might be to determine the right opportunity, followed by having the right requirements and architecture. During development, when requirements and architecture are relatively stable, the team focuses on completing requirement-items. Finally, when the software system is ready for deployment, dealing with, deployment, operation and support challenges become more important. Consequently, teams need to apply and emphasize different practices at different stages of development.

Scaling Up – We call the third dimension "scaling up". Scaling "up" is about moving the use of the kernel from a team with a small number of members and low complexity "up" towards cases involving larger numbers of members and systems with greater complexity. "Scaling up" happens in large and complex development, such as enterprise systems, product lines, etc. These systems are usually complex, developed in a distributed manner at multiple sites, or even outsourced. They have in general many frameworks and technologies. Here we often are no longer dealing with a single set of requirements, a single software system, a single piece of work, and a single team. In these more complex cases, we need to place more focus early on establishing a good organization and clear responsibilities and interfaces between teams. This includes clearly defining what each team should do and produce,

and how teams should collaborate.

16Scaling In – Understand how practices work together

To collaborate effectively the members of a team have to agree on which practices to use and how to use them. They may select the practices that they are familiar with, and that they believe will help them the most. In reality, the following is what happens. A team usually has something it likes from their existing method and something it wants to remove and replace with something better. So, the team starts with the kernel and overlays practices to form a method that consist only of what they want to keep from their existing method and the new practices which they want to adopt (see Figure 42) to form their own method.

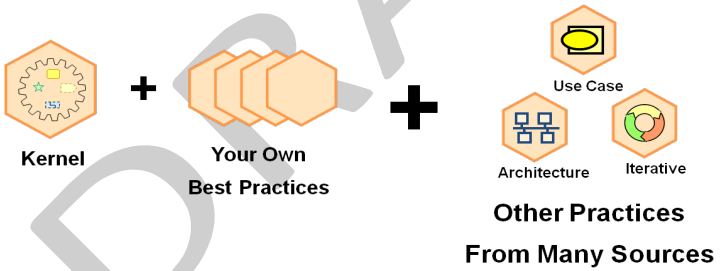


Figure 42 Building your own method by selecting the practices you need

In addition to selecting the individual practices it is also important to understand how the practices should be used together, if there are overlaps or gaps and so on. We will continue with Smith's iterative development story in Part 2 to demonstrate how he made practices to work together, in particular the three practices his

team agreed to use:

- backlog driven development practice,
- user story practice, and
- test-driven-development (TDD) practice

We will provide a brief introduction to these practices and discuss how we design/describe them to be work together well.

Side Bar 11 Do you really need to formalize these well-known practices?

The example practices that we have chosen do of course map onto what can be seen as more or less a vanilla approach to agile development today, i.e., Scrum with some XP practices. Many teams have great success in using these practices and our intent in this chapter is not to "improve" these practices or even prove that they faithfully can be represented as practices on top of the kernel.

What we want to do in this chapter is to show how *any* set of practices that are individually described on top of the kernel can be composed into a coherent way of working. The reason for choosing these well known practices are just to, from a widely-known context, illustrate the kernel approach to describing practices and how this approach supports you in describing your way of working in a more or less detailed way. Clearly, if we had chosen a more realistic situation with special or even proprietary practices the example would have become longer and less useful as an illustration to what we want to show.

16.1 Example of a backlog driven development practice

In Part 2, we showed how Smith's team applied a backlog driven development practice. Smith and his teammates fill the backlog with work-items and requirement-items and work them off iteratively, through a series of plan-do-adapt cycles. To provide concrete guidance to a team, Smith describes this backlog driven development practice through the things to produce and the things to do (see Figure 43). His team's view of the kernel involved progressing Requirements, Software System, and Way of Working. These alphas are grayed in Figure 43 to indicate that they come from the kernel. Smith's team is also interested in Requirement-Items, but it is an alpha introduced outside the kernel and is marked white.

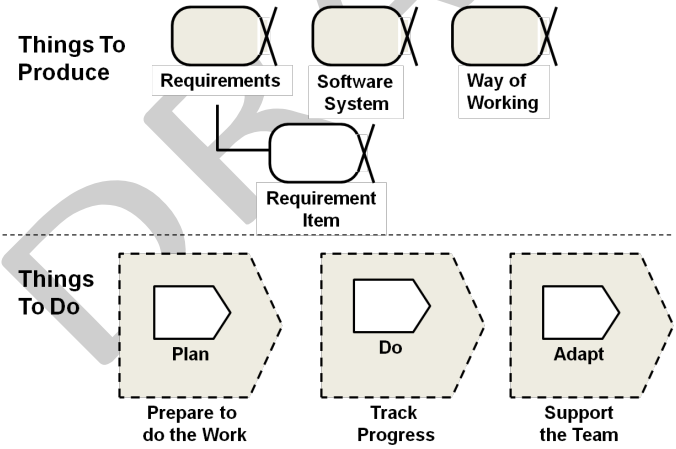


Figure 43 Backlog driven development practice

As part of adapting his way of working to provide greater visibility into his team's progress, Smith defined a set of states for the

Requirement-Item alpha. This Requirement-Item is not part of the kernel, but is something Smith introduced (see Part 2 Chapter 10).

The things to do in Smith's backlog driven development practice description add three activities Plan, Do, and Adapt to the activity spaces in the kernel: Prepare to do the Work, Track Progress and Support the Team respectively.

Now, although Smith's backlog driven development did provide more fine grain tracking of progress (i.e. looking at individual Requirement-Items as opposed to the Requirements as a whole), his team needed something more:

1. They needed a way to describe Requirements and Requirement-Items in a light weight manner.
2. They needed a way to ensure that they deliver good quality all the time.

Smith and his team agreed that they would use user stories and test driven development respectively.

16.2 Example of a user stories practice

A user story describes a usage of the software system through a simple narrative of the form "As a so-and-so user, I want to do-so-and-so". For example, in our mobile application case study, we will have user stories like:

- User Story 1. "As a travelling user, I want to browse my social network offline."
- User Story 2. "As a travelling user, I want to post comments offline."
- User Story 3. "As a travelling user, I want to show my

album to my friends offline."

A user story is a description of a Requirement-Item. In Semat speak it is a work product. Work products are evidence (such as documents} of alphas.

> A work product is an artifact of value and relevance to a software development endeavor.

To make the user story practice explicit for his team members, Smith explained to his team mates how to apply the practice in terms of things to produce and things to do in accordance to the Semat's approach to describing practice (see Figure 44).

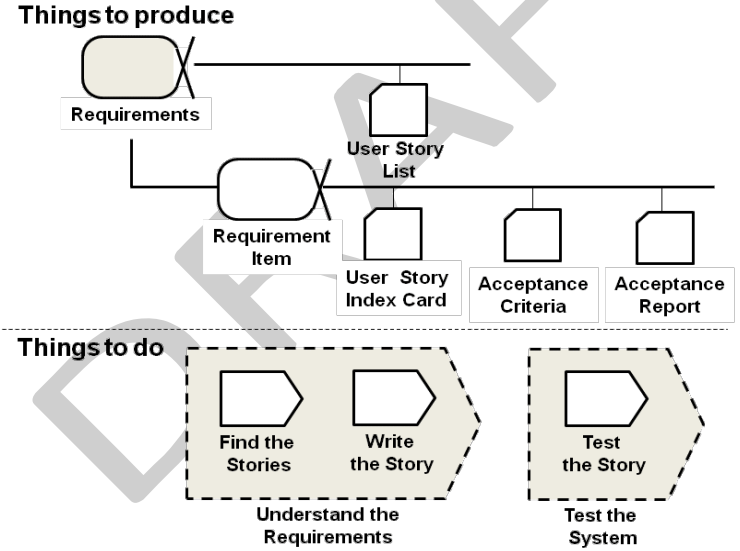


Figure 44 User story practice

Let's consider the things to produce (see top half of Figure 44). The user story practice extends both the kernel and the backlog driven development practice with explicit work products (Figure

44). These work products describes the Requirement-Item from different perspectives, not just from a user perspective but also from its design, implementation and test perspective. If you have been using user stories, you will be familiar with the following work products:

1. **User-Story Index Card –** The primary work product of the user story practice is a user story index card. Physically, this is a 5x4 index card, which contains the user story narrative. The subject of a user story index card is about a specific requirement-item. Hence, in Figure 44, the User-Story Index Card is attached to Requirement-Item alpha.
2. **User-Story List –** A user story list identifies the whole set of user stories. Physically, the user story list is the whole set of index cards. The User Story List is attached to the Requirements alpha.
3. **Acceptance Criteria (for User Story) –** On the reverse side of the user story index card, the team writes the acceptance criteria to meet when testing that the user story has been correctly implemented. The acceptance criteria are about a particular Requirement-Item and are hence attached to the Requirement-Item.
4. **Acceptance Report –** At the end of an iteration, the team demonstrates the software system to customer representatives and takes note of their feedback. This is a simple check on the user story index card to indicate successful acceptance criteria, or some remarks on the user story itself. An acceptance report is yet another description of a specific Requirement-Item.

Let's consider the things to do (see bottom half of Figure 44). Conducting the user story practice involves the following activities:

1. **Find the stories –** As part of Understand the Requirements,

Smith and Angela would identify a set of stories for their mobile application. They found an initial set of stories which describes individual requirement-items) as part of starting up and refined them as they progressed through the iterations.

2. **Write the story –** For each story, Smith and Angela agreed the acceptance criteria as part of achieving the Requirement-Item acceptance criteria state. This helped them to come to an agreement to what Smith and his team must show case to Angela and the stakeholders.
3. **Test the story –** Smith's team criteria for completing each requirement-item was to test each story against the agreed acceptance criteria to produce an acceptance test report, which Angela would verify.

16.3 Example of test driven development practice

Smith and his team agreed that it is their responsibility as developers to ensure quality and adopted the test driven development practice. Test-driven-development (TDD) advocates writing test code before writing or modifying actual (target) code. This is in contrast to traditional development where teams write actual code first. It is only when the actual code is completed that the team writes the test code. With TDD you first begin by identifying the test cases for some given requirement-item and write the test code needed to test that the requirement-item has been correctly implemented. Before the target code is written running the test code will of course result in a failure. Once you have written some code you improve it until the test pass.

Since both Dick and Harriet were new to TDD and needed advice

on how it fits into their way-of-working, Smith explained and wrote down this practice in terms of things to produce and things to do (see Figure 45).

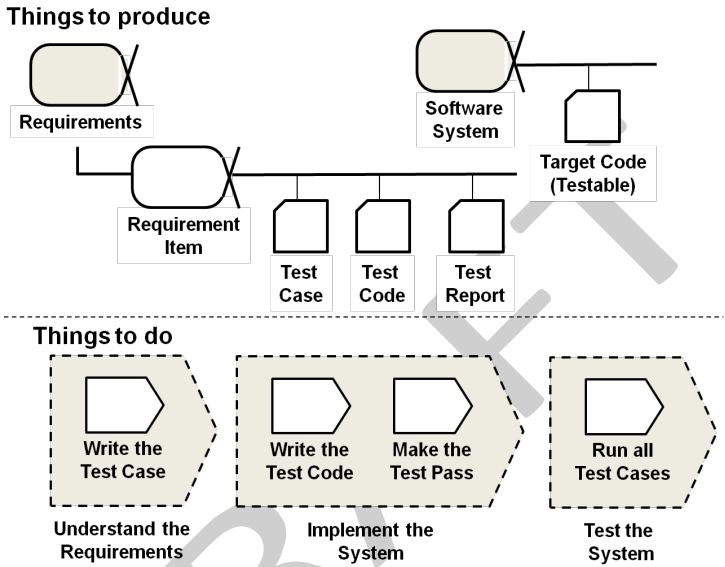


Figure 45 The test driven development practice

Let's consider the things to produce (see top half of Figure 45). Test driven development extends the kernel and the backlog driven development practice through the following work products:

1. **Test Case –** For each requirement-item, Smith's team would write a set of test cases. Thus, each test case is related to a specific requirement-item, and hence Test Cases are attached to a Requirement-Item in Figure 45.
2. **Test Code –** Next Smith's team would write the test code for the test cases using JUnit (A testing framework for Java).
3. **Target Code (Testable) –** Smith's team wrote the code

needed to implement the requirement-item and to pass the test case. Now, the target code is not part of the TDD practice since it is something the team needs to do anyway. However, Smith's team discovered that they had to do something with their software system to make it more testable. This means some small refactoring. The target code is an aspect of the Software System itself, and hence, the Target Code is attached to the Software System.

4. **Test Reports –** Finally, the team uses JUnit to produce test reports as evidence that it has indeed fulfilled all the test cases for the requirement-item. Test reports are obviously attached to the Requirement-Item.

Let's consider the things to do (see bottom half of Figure 45). Conducting the test driven development practice revolves around the states of the Requirement-Item alpha, namely *Scoped*, *Criteria Agreed*, *Analyzed*, *Implemented* and *Accepted*. Smith's team achieves these states through the following activities:

1. **Write the Test Case –** The pre-condition to conducting TDD is to have each requirement-item *Scoped*. Writing test cases achieves the *Criteria Agreed* state.
2. **Write the Test Code –** Smith and his team used the test code as a way to analyze the impact of the requirement-item on the software system, i.e. which parts of the system should be changed, what are the interfaces, and other implementation details.
3. **Make the Test Pass –** Smith and his team mates would achieve the *Implemented* state by making the test cases pass.
4. **Run all Test Cases –** Finally, Smith would run all test cases for all requirement-items as a way to achieve the *Accepted* state.

Note that the test driven development practice would work regardless of how the team chooses to describe its requirement-item, using user stories, use cases, traditional specifications, and so on.

16.4 Fitting practices together

Smith's team did not apply the above practices separately, but together. He needed to explain how these practices fit together, especially for Dick and Harriet, who were new to them. These practices when put together still comprise things to produce and the things to do (as shown in Figure 46 and Figure 47). They

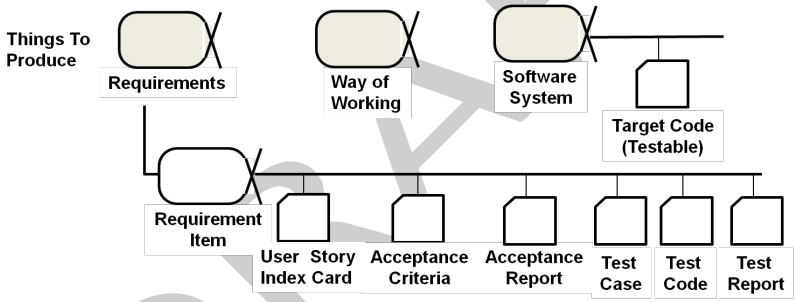


Figure 46 Fitting practices together (Things to produce)

Looking at the things to produce, note that Figure 46 only shows the alphas which Smith's team members emphasized, and thus do not have all the alphas in the kernel. It is just their view of the kernel. It also comprises the Requirement-Item alpha, which was first introduced by the backlog driven development practice and referenced by both the user story and the test driven development practice. The Requirement-Item alpha provides a bridge between these two practices. It also allows you for instance to swap the user-story practice with another practice that can capture or describe requirement-items.

In Figure 46, it is apparent that the subject of many work products such as the User Story Index Card, Acceptance Criteria, Test Case, etc. is the Requirement-Item alpha. As an example, the team writes a User Story Index Card for a particular Requirement-Item, the team agrees the Acceptance Criteria for a particular Requirement-Item, the team writes Test Cases for a particular Requirement-Item, and so on. When discussing these work products in the context of their respective practices, they are relevant, but now when the practices are composed, there is significant overlap between these work products. This is an indication that there is room for improvement, there is room to simplify. As an example, Smith and his team agreed that they could combine Acceptance Criteria and Test Cases together.

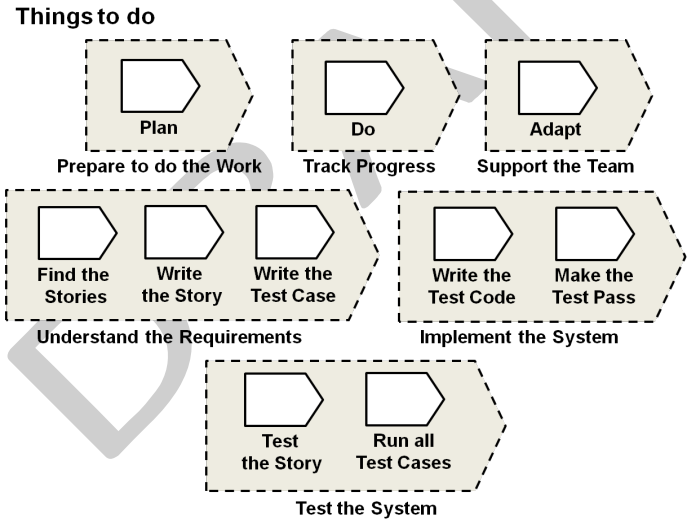


Figure 47 Fitting practices together (Things to do)

Looking at the things to do (see Figure 47), the activity spaces, by grouping activities tells the team to consider these activities together, and perhaps even consider ways to do them together. At

the beginning of development, Smith's team wrote stories and test cases separately. As they gained experience, they wrote stories and outline the test cases together.

16.5 Practices working together

Describing practices or methods explicitly is one thing, but how does the team apply the composed method involving the kernel, the backlog driven development practice, the user story and the test driven development practice? We illustrate this by walking through how Smith achieve the states of the Requirement-Item alpha because it is the central subject of both the user story practice and the test driven development practice work together (see Figure 46).

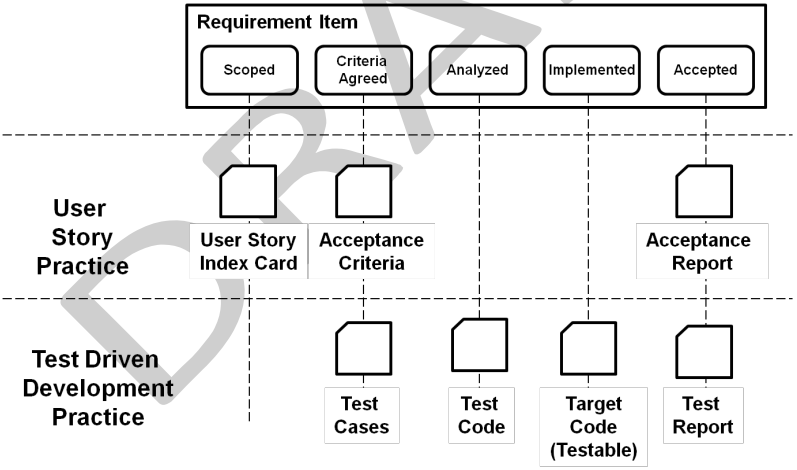


Figure 48 The work products required to reach the Requirement-Item alpha states

Figure 48 shows work products from the user story and TDD practices attached to the Requirement-Item's states. The

horizontal lines separate the boundaries of the user story and the TDD practices. Vertical dashed lines tell which work products have to be produced to achieve each Requirement-Item state. Smith's explained to his team mates how to achieve each state as follows:

1. *Scoped* – Smith and Angela worked out an initial set of user stories to achieve the Requirements bounded state. These set of user stories brought their respective Requirement-Items to the *Scoped* state.
2. *Criteria Agreed* – A team member in Smith's team would be responsible for a Requirement-Item (i.e. Smith himself, Tom, Dick or Harriet) would work with Angela to come to agreement of the Acceptance Criteria and write Test Cases for the Requirement-Item. This achieves the *Criteria Agreed* state.
3. *Analyzed* – A team member would write Test Code to achieve the *Analyzed* state.
4. *Implemented* – The team member would make the test case pass to achieve the Implemented state, which may take several iterations. As part of doing so, he/she had to make the software system more testable.
5. *Accepted* – Finally, the team member would run all test cases to ensure that not only did the requirement-item work, but it did not break other requirement-items.

By attaching the work products according to the Requirement-Item states, Smith's team members had a clearer picture of what they need to do, and when.

Side Bar 12 Do we need explicit activities for each alpha state

change?

Some people think in terms of activities, others in terms of outcomes. Neither is right or wrong. They are just different and such differences are part of why we have many different methodologies. It has been our experience from working with many small to medium sized organizations (25-150 people) that the majority of these organizations like to focus on outcomes giving their people freedom when it comes to how they go about achieving those outcomes. But most of these organizations were building non-life-critical software.

As the initial Semat kernel was being developed we heard stories from others involved with organizations that develop embedded life-critical software systems where strict policies were enforced on the activity side. This is why when building a kernel that must fit a broad range of software domains that participation from people who have a broad range of experiences is critical.

The kernel approach supports software development endeavors where activities can be specified or left tacit. Only the alphas are mandatory when defining and using practices, but you can extend the kernel alphas with your own additional non-kernel alphas and explicit activities to fit your needs.

By themselves, the user-stories practice and the TDD practice do not individually provide guidance for all the states of the Requirement-Item alpha. As such, we call these fragment practices because each covers only part of the Requirement-Item's life cycle, but together they are supposed to cover the whole life cycle. We say "supposed to" because when you combine two practices that are not designed to fit together you may find difficulties making this seamless and you may get waste – not being lean. In contrast a complete practice such as the use-case practice provides guidance end-to-end of the lifecycle. This practice is complete,

because it provides guidance on how to achieve every state of the Requirements alpha and the Requirement-Item alpha from requirements, analysis, design, code and test, and it does so seamlessly.

Side Bar 13 Potential consequences of gaps and waste in your way of working.

The purpose of this discussion is not to try to convince the developer that use cases are the best approach to requirements, but rather to demonstrate the power of the kernel approach in raising the visibility of gaps and waste in a team's chosen way of working.

What a team decides to do with this information depends on their specific situation. However, today, without a common ground reference point, software teams too often are unaware of the gaps and waste in their chosen way of working, and therefore they are also unaware of the risks to the quality of their software product.

Being seamless is important; a team can transition from one state to another without wasting the effort to go from one practice to another, one concept to another. The reason we didn't select the use-case practice as an example is that we wanted to demonstrate how we can work with two different practices instead of just one.

16.6 Empowering teams to build and use their methods

We have just shown how Smith was able to systematically and "easily" compose several practices into a composed method. This is easy because, the practices were designed to work together in the first place:

- By attaching new alphas (e.g. Requirement-Item) to alphas in the kernel.
- By attaching work products on agreed alphas, for example, attaching user story index card (from the user story practice) and test case (from the test driven development practice)
- By adding activities to activity spaces in the kernel.

While designing practices is not something a development team would do, selecting and fitting practices together is something it needs to do to at least clarify what needs to be done and when.

The kernel approach to describing practices is about making them actionable. This means first emphasizing alphas to progress, then emphasizing their states to achieve followed by work products to produce, and in that order. It is something meaningful to the team, unlike practice descriptions in the past, which were less intuitive for developers. Because of this emphasis on being actionable, development teams can quickly learn how to fit practices together and adapt them quickly to suit their needs. With the kernel approach processes are no longer descriptions on a shelf, but they become what developers actually do! This is a major advantage of the kernel approach over traditional process approaches.

Our approach applies the concept of separation of concerns; it allows you to add practices systematically without modifying what exists. This is important because by not changing what originally exists, you do not need to "unlearn" what you already know. Fitting practices one at a time on top of the kernel is what we call method composition: Your team's method is a composition of practices on top of the kernel.

Side Bar 14	Developers taking charge of their own way of working

Some may be concerned that placing developers in charge of their own way of working may lead us back to the days of cowboy programming and chaos. But this is not the case. The reason for past loss of control of software efforts was due to the fact that a common ground reference did not exist to guide the decisions of our developers. Without such a common ground in the past developers didn't have the concrete guidance they needed when they needed it. With the kernel approach and our current target states and their checklists always being kept in the forefront of our developer's minds the risk of lost control is significantly mitigated.

17 Scaling out – practices across the entire development lifecycle

The practices which we have discussed in the previous chapter, i.e. backlog driven development, user stories, and test driven development are centered on the "development" phase of a software development life cycle (SDLC). But software engineering is more than just development (or coding). There are many pre-development things which a team needs to consider such as figuring out what to develop in the first place, what architecture to use and so on. There are also many post development considerations such as planning its release and its operation. For this reason, the team would need different practices across the development lifecycle.

In this chapter we will start with a discussion of how you can build out your way of working to support a broader lifecycle by adding practices on top of the kernel. To some extent this discussion is based on that a rich library of practices is available for you to choose from. Today there are no such libraries that are widely used but the concept is proven on a smaller scale and it is our expectation that such libraries will become available over time as awareness of the kernel grows in the industry.

To round off this chapter we discuss how the kernel can be adapted to different life-cycles, different decision models if you so like. This discussion is, however, immediately useful.

17.1 Practices across the development lifecycle

As a team scales out from "development" to pre-development and post-development it will use more practices than the usual development practices (see Figure 49). For example, the team might use the product planning practice to agree on product ideas worth developing and commercializing. The team might use the architecture tradeoff-analysis practice to agree on a candidate architecture, and the architecture-centric development practice to confirm the architecture early in the software system's lifecycle. If the user interface for the software system is complex, the team may choose to use the user-centered-design practice.

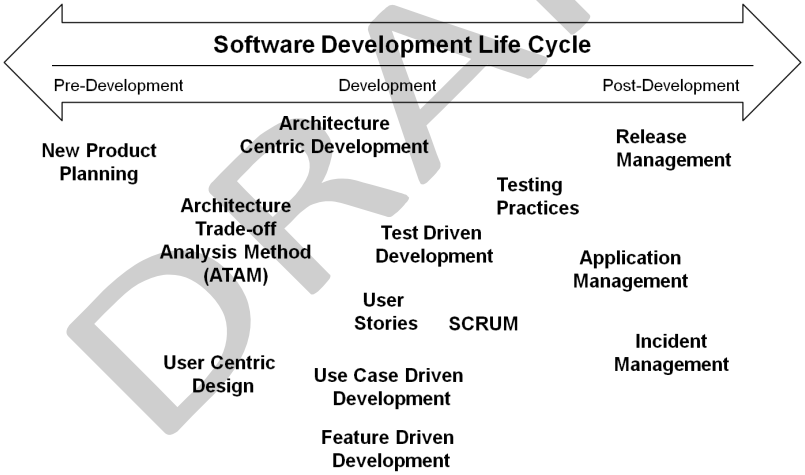


Figure 49 Practices across the software life cycle

During development the team might use user-stories, use-case driven development, feature-driven-development, or test-driven-development. The team may use a Scrum practice to get the team

to focus on implementing requirement-items incrementally. Finally, during post-development, the team might use practices for release-management. During the software system's operation the team might use some application-management practices to ensure that software system runs smoothly, and incident-management practices to handle software system faults, alarms and warnings.

Note that your team does not need to choose all the practices you will use up-front at the beginning of your software endeavor. Rather, your team needs only choose those practices as and when your team members need further guidance. In addition some practices are only relevant during certain phases in the software's lifecycle. Hence the members of a team do not need to grapple with many practices at once.

17.2 Extend the kernel across the entire development lifecycle

Each practice listed above in Figure 49 provides additional guidance to the development team how to solve their challenges at different stages of the software development lifecycle. The practices may do so by introducing additional alphas (see Figure 50). For example, a backlog driven development practice may add the Requirement-Item alpha; an architecture practice may add an Architecture alpha; an iterative development practice like Scrum may add an Iteration or a Sprint alpha; a component based development practice may add a Component alpha; test practices may add a Test alpha. But all of these are outside the kernel – they are added through practices. They are not essential elements of software engineering and it is up to each team to decide how many of these additional practices they need to define.

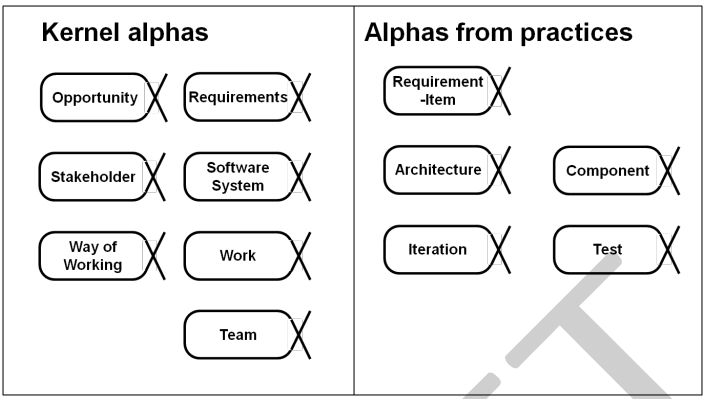


Figure 50 Alphas across the entire engineering lifecycle

Defining practices is not easy, not something which a typical software practitioner will do. This is a job for our most experienced and excellent developers, but when doing that they step away from the actual software endeavor work to work with writing practices – we call them method engineers in this book. A good method engineer not only must have a deep understanding and experience regarding the practice he wants to describe, but must also have a deep understanding of software development methods. Good method engineers are few. However, using practices, and especially if the practices are described as we advocate in this chapter, is easy. A team using the practice merely follows the guidance from the practice to progress the associated alpha states.

17.3 Adapt to different development lifecycles

We have earlier mentioned that "scaling out" is about supporting development with appropriate practices across the entire

development lifecycle. Now there is another aspect of "scaling out", which is about the lifecycle itself. A lifecycle defines a set of milestones that serves as a guide for a team to make its development plans, and for larger organizations, resourcing plans and even funding plans. Each software endeavor has its own set of risks, and hence its own development lifecycle. There is no one size fits all. How can you build a development lifecycle suitable for your endeavor? We will demonstrate how you do so with the alphas.

Let's assume that a development lifecycle has three major phases:

1. Pre-development. This is before any actual development (read requirements, design, coding and testing) starts, and comprises preparatory work.
2. Development. This is when the team development occurs. There is coding and testing. Requirements work occurs here as well.
3. Post development. This is when the software system is available to end-users.

Your organization or your method may have more phases, but let's use these three for discussion's sake. Lifecycles differ from each other by the emphasis in each phase and the duration of each phase. In essence you can build your own development lifecycle by allocating alpha states to the lifecycle phases. Figure 51 and Figure 52 give two examples of development lifecycles. Each figure allocates the Requirements and Software System alpha states to the development lifecycle phases. The labels 1 to 6 indicate the states of the respective alphas.

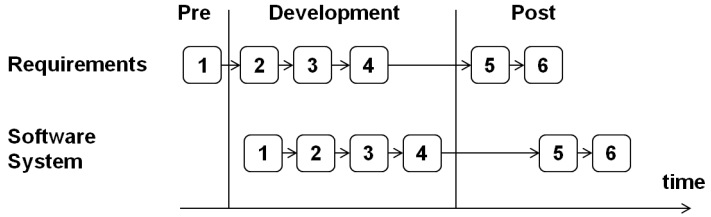


Figure 51 Modern development lifecycle

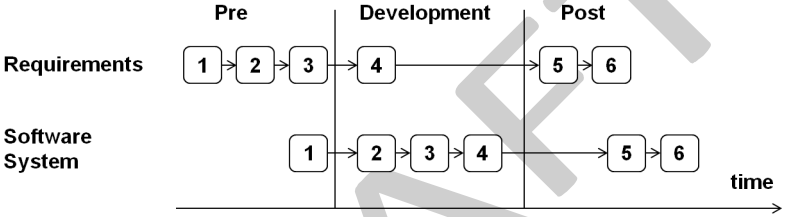


Figure 52 The waterfall development lifecycle

The key differences between Figure 51 (modern development lifecycle) and Figure 52 (the waterfall development lifecycle) are the allocation of states between the pre-development phase and the post development phase. The modern approach starts development earlier. As you can see we have just used the same alpha states to describe both the traditional waterfall and the modern development lifecycles. This is very useful because it means that practices can be described independent of the lifecycle. Practices that have been proven withstand the test of time and withstand the test of the latest methodologies and lifecycles. The kernel acts as a bridge between practices (e.g. user stories and test driven development, etc.) and lifecycles (e.g. traditional waterfall or modern).

18Scaling up – practices for large scale development

The third dimension of scaling is scaling "up" from development involving a small number of people to development involving a large number of people. In such cases there is often not one set of requirements, but many; not one software system, but many, not one set of work, but many; not one person in a team but many; not one single team, but many.

Throughout Part 2 and 3, we have shown how Smith and his team worked together as a small team doing a small mobile application. Actually, this mobile application is actually part of a much larger mobile device development. It has a lot more requirements and they are categorized into several requirement areas:

- Social application requirements, such as that for the social network.
- Entertainment application requirements, such as games, and music players.
- General requirements, such as customization of the device's home screen, hardware interfacing requirements, and so on.

Figure 53 Requirement areas of a large development

In a large development such as this, how you organize work and team responsibilities is very important. This story has a team organization as follows:

- The product planning team comes up with product requirements through market feedback and watching technology and social trends.
- The architecture board has to reach consensus on technical decisions.
- There are several development teams, one responsible for development and acceptance of each requirements subset (i.e. the social application requirements, the entertainment application requirements and the general requirements).
- Even though each development team conducts its own tests, there is another acceptance testing team to perform some final or independent verification.

There is also leadership team to ensure that things progress well and team works well together. In addition, someone is tasked to look at how members can work effectively and consistently and seeking ways to improve their way of working. In certain organizations, this person is known as a process engineer.

18.1 Organize work using the alphas

As mentioned above, our large development story has a team organization that comprises a product planning team, an architecture board, several development teams, and an acceptance test team.

The kernel alpha states provide a simple way to clarify the

responsibilities of each team (see Figure 54). Each box represents a team's responsibilities. Each box contains the 6 states of the Requirements alpha and the 6 states of the Software System alpha.

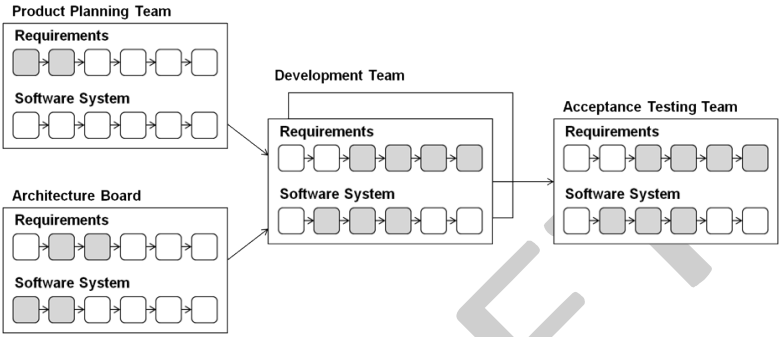


Figure 54 Agree on team responsibilities using alpha states

A shaded state means that the team has something to do with the alpha when it is in that state or when it is about to get to that state. For example, the Product Planning Team is responsible for the first 2 states of the Requirements alpha (namely *Conceived* and *Bounded*); the architecture board is responsible for achieving the second and third state of the Requirements alpha, and the first two states of the Software System alpha.

Because each team has different responsibilities its backlog will also be filled with different kinds of backlog-items (see Figure 55).

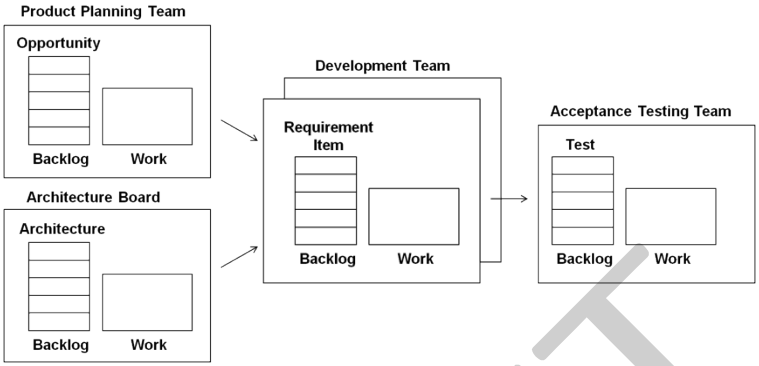


Figure 55 Different kinds of backlog-items for different kinds of teams

To continue our mobile device story, (see Figure 55), every team has their own backlog. The development team's backlog is a list of requirement-items. Other teams have their own backlogs, but different kinds of items.

- The product planning team is responsible for turning opportunity into reality. They work with a number of opportunity-items (i.e. product ideas) to determine their commercial viability. If the opportunity-items are viable, the product planning team will use some practices to translate them into requirement-items and explain these to the development teams.
- The architecture board is responsible to make architecture decisions, such as which technologies to use, the strategies for allocating runtime resources, how processing should be distributed across computing nodes, and so on. Poor decisions will of course lead to poor architecture. So, early during development, the architecture board needs to build a backlog of architecture decisions, rank them according to their impact to the software system and development effort, and solve each

(or a couple) of them at a time.

- As we already have discussed, each development team's backlog comprises a set of requirement-items. We have chosen to let each development team be responsible for one requirement area. So, in our example, there is one development team for the social application requirements, one development team for entertainment application requirements, and so on.
- Ideally, the development teams should be responsible for ensuring the quality of the software system including testing it. However, most organizations have an independent acceptance testing team to conduct some final verification. The acceptance testing team's responsibility is to verify the quality of the software system against a set of test-items. A test-item is basically a set of test cases. For example, a set of functional tests, a set of performance tests, a set of reliability tests, and so on. These test-items form the acceptance testing team's backlog.

Agreeing on team responsibilities using alpha states has several advantages. First, the alpha states provide step by step guidance to the team on how to achieve progress and health. So agreeing on responsibilities through alpha states is certainly more concrete as compared to traditional approaches such as using an organizational chart, which doesn't clearly identify what each teams or member does.

Second, one can quickly determine if there are missing or overlapping assignments. For each alpha state, there must be some team with the primary responsibility for achieving the state. Other teams might be involved or consulted, but they do not have the primary responsibility to get there. For example, the development teams are responsible for getting the Software System to the

Demonstrated state, but the architecture board is also involved in the work to get there.

Side Bar 15 The power of using alpha states to aid visibility of team responsibility

When we have observed large software project failures in the past the root cause has been, more often than not, traceable to a lack of clear ownership of responsibilities within the organization. Aligning responsibilities with the alpha states ensure someone is looking out for all the essentials. It also provides visibility of possible overlap of responsibilities and potential waste.

18.2 Drive development with the alphas

Having agreed on the responsibilities for each team does not necessarily mean that the teams will collaborate well. If each team makes its own plans, and run its own work, collaboration across teams will not be effective. You need to get teams to make their plans and their work status visible and easily understandable to other teams, and provide mechanisms for the teams to discuss progress and remove blockages.

One approach is to have some kind of a leadership team, which comprises representatives from the subordinate teams (i.e. the product planning team, the architecture board, the development teams and the acceptance testing team). There is a plan-do-adapt (as discussed earlier in the book) in the leadership team, as well as the subordinate teams.

- **Plan –** The leadership team determines the current state of development, and plans the iteration. The results of this iteration plan are allocated down to the subordinate teams.

The subordinate teams break down the leadership iteration plan into actionable tasks.

- **Do –** The subordinate teams perform the tasks in their iteration plan and report the results to the leadership team. The leadership team helps coordinate tasks involving more than one team.
- **Adapt –** The leadership team reviews its way of working and how teams collaborate and tries to improve it. Each subordinate team conducts their local retrospectives.

In large organizations, it is very easy for teams to start doing their own thing and lose sight of the overall development priorities. Thus, it is important:

- For the leadership team to ensure development priorities become the subordinate team's priorities.
- For the subordinate teams to understand the contribution of their progress to the overall development progress.

Leadership Team – The leadership team needs a good way to understand the development progress. Here the kernel alphas (in this case the Requirements alphas) become useful. They provide a way to understand the current state of the software endeavor (see Figure 56).

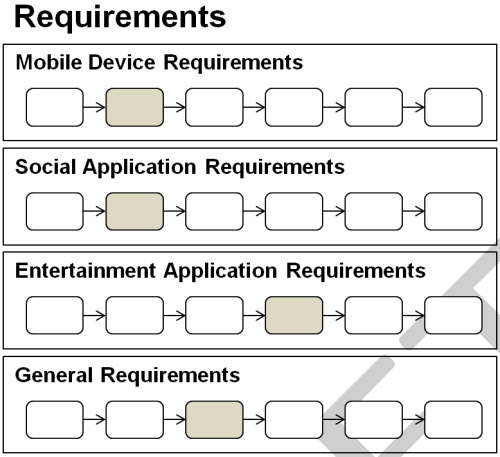


Figure 56 Current states of requirements

Each row in Figure 56 depicts the current state (in grey) of the requirements' alphas. On the top is the overall set of requirements and further down the requirements' subsets. The general rule of thumb is this: the state of the overall requirements is that of the slowest requirements' subset. Here you can see that work on the requirements for the social application requirements is lagging and is therefore holding back the state of the overall mobile device requirements. This means that the development team needs to put in more effort in progressing the social application requirements. The team can do this either by putting greater emphasis here or applying a more appropriate practice for this requirements subset.

The above is very similar to what we have discussed earlier in the book for planning iterations. That is, to determine the current states of development and the next states of development and to determine the way to achieve the next states. While what we discussed earlier in the book was for a small development, now we do this for a large development. Here we apply the kernel alphas recursively for the overall requirements and their subordinate

parts. This demonstrates the scalability of our approach with alphas.

Side Bar 16 Should a developer's practices change due to the size or complexity of a software endeavor?

Some have argued that as software endeavors scale up in size or complexity the degree of explicit and rigorous documented practices should also increase. In other words, your practices need to change based on software endeavor size. This makes sense to a point, but it has been our experience that many practices that are based on the essentials do not need to change and can aid developers on large and complex efforts just like they help on small efforts.

As an example we have observed very large and complex projects, some with over 500 developers physically distributed across multiple organizations each employing their own distinct software methodology.

One particular project was in trouble well behind schedule and over budget. Parts of the project were being managed by traditional functional managers, who were driving their team's effort from a traditional waterfall perspective, while other parts of the project were being driven from a more modern iterative approach.

By first reorganizing all developers into teams of no more than 10-15 people and ensure each of these teams had clear short term goals that aligned with clearly defined near term project milestones the developers on each team better understood what they had to do. By breaking the large complex problem down into small near term chunks of work the project got back on course.

The lesson here is a simple one. The best way to solve a complex problem is to break it down into multiple smaller problems and the best way to solve each small problem is by using the same best

practices that have been proven to work for small endeavors.

It is true that some additional coordination across the teams is needed on large projects, as discussed earlier, to address coordination issues. But best practices are fundamentally the same independent of software endeavor's size.

18.3 Coordinate the subordinate development teams.

Recall earlier in Part 2, we talked about making work visible to team members through task boards and Kanbans. Once work becomes visible, it is easier to align each member's priority and help each member understand his/her contribution. The story in Part 2 was about a small team. Now, we extend the use of task boards and Kanbans for large development teams. We focus our discussion on requirement-items according to their criticality and impact on the software system:

1. Non-critical requirement-items.
2. Critical-requirement-items.
3. Crosscutting requirement items

Figure 57 shows an example of Kanbans used by the leadership team and the subordinate development team.

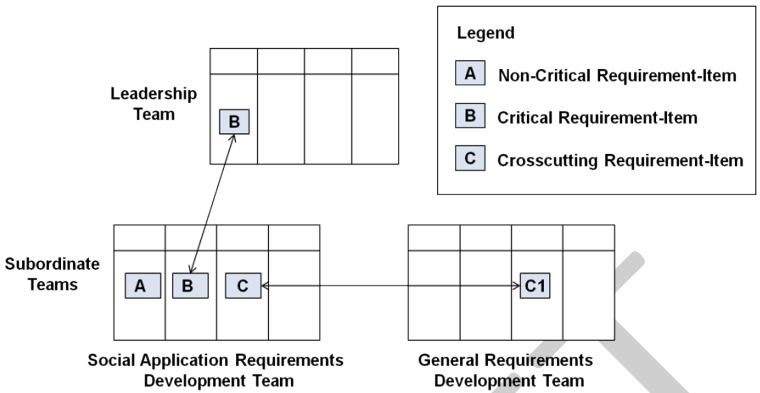


Figure 57 Visualize work across teams using Kanbans in large software endeavors

Non-Critical Requirement-Item – In the simplest case, a subordinate team can manage their own without much collaboration with another team. This applies to non-critical requirement-items (denoted by requirement-item "A"), that are simple and the subordinate team can deliver by itself. It does not affect other teams. In this case, the subordinate team can work very much on its own.

Critical Requirement-Item – Sometimes, a requirement-item, which a subordinate team is responsible for bears great significance to the overall development, which is denoted by "B" in Figure 57. For example, the ability to "Browse photos offline" is very important to the success of the overall development. The leadership team wants to monitor the progress of this requirement-item closely. Here it is useful to not only track this requirement-item's progress within the subordinate team, but to track it within the leadership team. Thus, it appears in both Kanbans.

Cross-Cutting Requirement-Item – Sometimes, requirement-

items are complex and cannot be allocated to a single requirements subset (i.e. they cut across different requirements areas). In this case, they have to be allocated to more than one subordinate development teams. In most cases, one subordinate team is chosen as to have the primary ownership. It is responsible for collaborating with other teams if necessary. This is exemplified in Figure 57 by requirement-item "C", which the social application requirements team is responsible for. This requirement-item depends on another requirement-item "C1", in the general requirements team. You just need a simple way to label each requirement item such that their relationships can be easily inferred. In our example, we use the prefix "C" to highlight the relationships. In practice, you can choose other techniques, such as using colors or some numbering scheme. Regardless, they help the teams understand their contribution of the overall requirements development.

So, you can see that the traceability of requirement-items across the subordinate and leadership teams help them collaborate effectively.

Side Bar 17 Is there really anything new here for the developer?

At this point the developer may be wondering is there really anything new here? Aren't these ideas that are being presented in this chapter on how to scale the same ideas that have been used before? Leadership teams, subordinate teams, and deciding how much attention needs to be placed on critical and non-critical requirements items?

The answer is yes and no. The ideas are not new. They have been proven to work before. What we now know is that when software efforts fail, more often than not, the root cause traces back to fundamental essentials that somehow got missed. What is new is that through the kernel approach we now have a tool to

keep our team focused on the most important things -- those essentials that we all know are critical to success.

We anticipate that the students of the future will learn about the kernel in the universities before they leave school. This is necessary to help them be prepared for what they will face when they go into industry accepting a developer's job, but it is insufficient to ensure their success. It is insufficient because we have found when faced with the pressures and complexities of real world software endeavors, it becomes far too easy to let the essentials fall by the wayside.

The kernel, through its alphas and related checklists, gives us a powerful tool to keep what is most important now visible to each developer helping them practice their own way of working more consistently leading to the higher quality software we seek.

19Practices help scale the kernel

In this part of the book we have introduced the concept of practices. The word "practice" is a common term in software development and the industry uses the term rather loosely. This might not pose any problem when we are just talking about just one practice, but as a team starts to apply different practices it becomes important for them to know which practices are more appropriate for them and how different practices can fit together. If individual members have different backgrounds, they can have very different ideas as to what using a particular practice means. This can lead to miscommunication and confusion and work delays in a development initiative.

Here we give a precise meaning to the term "practices". A practice provides the guidance to deal with some aspect of software development. Specifically, it provides guidance to the team on what to produce, when to produce it and how to produce it, according to alphas and their states explicitly. This does not mean that our form of practices have to be heavyweight in their description. Just like we can compact the kernel into a deck of cards, each practice can be presented as a set of cards. Just like there are cards for alphas, there can be cards for work products.

More importantly, our approach to practices allows you to systematically add practices to form your team's method of working, and in a large development initiative, for different teams to work together. This is very important for a number of reasons:

1. By understanding how practices fit together, teams and individuals can focus on their work without being engaged in debates as to who does what.
2. A team can gradually evolve its method of working by swapping practices with more appropriate ones.
3. Methods engineer can write re-useable practices that can be used by different teams.

It doesn't matter what new practices may come in the future. With the approach we advocate in this book you can easily compare these new practices with what you have. If it is more appropriate, swap them with less appropriate ones. With this ability, your team is prepared to embrace new and better ideas from the software development industry.

Part 5 – Principles and Values

The kernel is not a new unified methodology, nor a new software process meta-model, nor a new body of knowledge, nor a new modeling language, nor is it a trick to get people to build or buy more tools. The kernel is as simple as what we already have (e.g. teams and work), what we already do (e.g. understand requirements, implement, test), and what we already produce (e.g. software systems) when we develop software. This is irrespective of the way we work, whether we document it, or even if the result is good or bad.

On the other hand, what we are doing is remarkably different from previous initiatives. However, before talking about the differentiators, let's first recognize that we are not starting from scratch. Previous attempts have in one way or another taught us something. They have taught us what works and what doesn't. In our search for differentiators we have focused on what really makes a difference to developers that produce the software we rely on today.

This section of the book is intended for readers who want to understand the big picture underlying the kernel approach. It is for those who are interested in understanding how the kernel approach is different from past initiatives and how you can use these differentiators to grow your team, improve the way your team works, and improve the quality of the software your team produces.

In the next chapter we discuss how developers think about their method.

20 Thinking about methods without thinking about methods

As a developer, you are focused on getting your job done, delivering useful high quality software on time on budget for your customers. But in doing so, you meet all kinds of development challenges:

1. Getting the real requirements from your customers
2. Implement the requirements correctly
3. Choosing the most appropriate technology
4. Collaborate with team members
5. Ensuring quality in your system
6. Ensuring quality in your architecture
7. Delivering your system on time

You undoubtedly have other challenges as well.

20.1 Thinking about challenges is thinking about methods

If you were to ask different developers how they each handle these challenges, you are likely to get different answers. Some developers will say it depends on your specific situation and the characteristics

of your development endeavor. For example:

1. Are you developing a brand new system, enhancing an existing system, or integrating two separate systems?
2. Are you exploring the feasibility of some new technology?
3. Are you developing a mobile application, or is it a large military system, or is it a banking system?
4. Are you doing in-house development, off-shoring development, or are you outsourcing?

The implication is this: when someone comes up with a method to solve a particular challenge it must consider the context; the team, the technology, and the environment. Some teams are successful with user stories in a web development environment, but another team building a military system may find user stories insufficient.

Developers most often face multiple challenges. In so doing they think about methods, without trying to think about methods. They spend time thinking and reflecting about the way they develop software considering issues similar to those listed.

To be successful your method must be good helping you develop *good* software *quickly* making your customers *happy*. Some methods are better than others. Some are more lean and more agile than others.

Similarly, not all developers face the same challenges. A developer who just graduated from the university is often given narrowly scoped responsibilities which limit the challenges faced and the need to think about their method.

On the other hand, a developer with many years of experience may spend significantly more time thinking about their method, or the way they work. Figuring out how to best approach current challenges is something that often takes considerable effort. In

fact, thinking about, discussing and debating possible approaches to solving current challenges often take more time and effort than many developers realize.

So, whether you are a young developer or a seasoned developer, if you are serious about being professional in your work and becoming better at what you do, you need to equip yourself with the best tools that can help you think about your method. That is, the way you work.

20.2 In fact, without making a fuss about it, you think about methods all the time

Developers may think they spend more time developing software than thinking about how they develop software. But the fact is they often spend far more time than they realize thinking about how they work. This can be frustrating because it is not just thinking they have to do.

They also have to discuss what they are thinking about with a teammate and this may lead, to prolonged debates, frustration and even project delays.

The time spent thinking about methods is greater if your teammates have different backgrounds, experiences or competence. The time is also greater if the members of your team don't have an effective way to share their knowledge.

Nevertheless, thinking and agreeing about methods is not something you can ignore. It is necessary to build software better, faster and achieve happier customers.

Most developers are not aware of the time spent on this activity.

Time spent thinking about the way you work is usually not measured. You don't make a fuss about it because you know ignoring the challenges we have been discussing puts your development effort at risk. When we begin to realize how much time we devote to this activity we also realize how much potential for improvement exists if we can be more effective in the way we work.

Ideally, you should spend very little time talking and discussing software development methods. You should spend more time doing software development. To do this you need to have a good way to think about the way you work and improve it. This is what the kernel does for you. For example, it helps you:

1. Reach an agreement with your teammates quickly
2. Become more effective in finding the answers to your development challenges

As you become better with methods, you become better at addressing your challenges. As you become better in addressing your challenges, you become a better developer. The kernel helps to make methods a natural part of the way you work so that you use and improve your method as you work without even thinking about it.

21Agile working with methods

To be successful with methods you need to be agile, light and lean in everything you do while working with methods. This doesn't mean you must choose an agile way of working. Rather, it means you must be lean in how you build and adapt your method. You learn a little at a time while developing real software and you must improve how you work as you learn better ways to work.

The agile movement in the early 21st century drew attention to the value of people, and teams. It emphasized the need to continually evaluate and improve the way the team works.

The agile and lean movements have brought greater attention to customer involvement and customer satisfaction. In contrast to many previous initiatives the focus 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. What we have learned by listening to our real customers is that they learn what is best for their endeavor a little more each day of their endeavor. This takes a change in mindset. It requires a change in the way you think about methods, just as moving to agile development from traditional development requires a change in your mindset. What are the mindset changes you need to be aware of? To some extent they are inspired by those behind the agile manifesto[8].

1. The full team owns their method, rather than a select few
2. Focus on method use rather than comprehensive method description

[8] http://agilemanifesto.org/principles.html

3. Evolve your team's method, rather than keeping your method fixed

It is worth delving deeper into each of these mindset changes.

21.1 The full team owns their method, rather than a select few

Traditionally, there is a dichotomy between methods engineers and methods users (i.e. developers).

1. Methods engineers who define methods often do not use the methods themselves.
2. Methods users who acquire real experiences with methods, often are not asked, or do not have time to give feedback and refine the methods.

As a result, methods users find methods 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 work best for their team.

In the preceding chapters we have demonstrated how the kernel can assist this proper balancing through the use of the alphas, states, checklists and describing a team's own method through 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.

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.

21.2 Focus on method use rather than comprehensive method description

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, these method descriptions have often become too heavy-weight. This has only served to make the process description, less, rather than more useful.

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 to help them in their day to day jobs? Consider the following team need with respect to their methods:

1. Teams need a way to determine real development progress.
2. Teams need to plan their projects, their iterations, and they need to discuss and agree upon what it means to be done.
3. Teams need to organize their team members, 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.

These are examples of a team's real needs that the kernel approach can help you address.

21.3 Evolve your team's method, rather than keeping your method fixed

There is no one-size-fits all when it comes to methods. This implies several things:

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. But always consider your current development situation when considering a change

Evolving a method is simple with the kernel approach. You start with the kernel, and consider what practices you already have. Then you replace or refine existing poor or weak practices for better ones. And you should do it gradually so it becomes natural and you don't even need to think about it.

22 Separation of concerns applied to methods

The Semat approach to methods is fundamentally different to what has been done in the past. This can all be summed up with the phrase "separation of concerns". Separation of concerns is a principle that qualified software teams may apply when designing software systems. Separation of concerns helps software systems become extensible. Here, we apply the principle of Separations of Concerns (SoC) to methods. SoC allows us to:

1. Specify a core, and,
2. Make extensions without changing or complicating the core

Not changing or complicating the core as you add extensions ensures that the team, after learning the core, need not unlearn after extending the core. We apply SoC in many areas in this book to separate the core from the extensions, the essentials from the additions. In fact, our work towards what we have presented in this book is a path of applying SoC, in the following steps.

1. Separating the kernel from practices
2. Separating alphas from work products
3. Separating the essence from the details.

22.1 Separating the kernel from practices

The birth of this work stemmed from an observation that our

industry has been zigzagging around for 40 years. We zig to one method, then zag to the next. Often it feels like we are moving from one extreme to another. While there has been evolution we have also thrown out valuable ideas along the way only to later rediscover them. Some developers are tired of the zigzag story of software.

So, we have need of a better approach – one where we do not need to unlearn essentials when there is something new or better – one where we can learn and adopt new ideas readily. Thus, we started to apply the principles of SoC to separate out practices from one another – practices that can be chosen and adopted separately from others. As we attempt to separate practices, we recognize that there is a common ground between methods, which we call the kernel, and the differences between methods can be described through practices. Following the principle of SoC, we employ an approach whereby practices extend the kernel without modifying it. The advantage is twofold.

1. Kernel is a stable core. First, the kernel provides a common ground. It gives teams a common language, and a common solid foundation to collaborate and achieve progress in their software initiatives independent of the kind of software endeavor, independent of the complexity of their requirements, or software system, independent of their team size.

2. Evolve methods practice by practice. Second, practices are modular units that help teams overcome challenges in their software endeavors. Each team can choose the practices which it needs. The team replaces existing practices with more appropriate ones. This provides an evolutionary path for the team to grow their methods.

22.2 Separating alphas from work products

Having recognized that practices can be separated from one another and from the kernel, we next recognized the presence of alphas and how they differed from work products (documents).

Traditional approaches to methods often overemphasized the value of documents and document templates as a measure of progress. Such a mindset is heavyweight. Agile proponents are right in emphasizing working software over documentation. However, working software alone is insufficient as a measure of progress. There might be missing requirements; the software system might not be easy to change or test; there might be excessive unplanned work; the team might not be collaborating effectively, etc. Thus, a team needs to consider progress from different perspectives: requirements, software system, work, team, and so on. This is where the value of alphas can be found. You measure a team's progress based on the alpha states achieved rather than the amount of documentation produced. A team can produce documentation without achieving progress. Our emphasis is on making progress through alpha states, rather than documentation.

This is why we separate alphas from work products. A simple way to recall the alpha name is through the phrase "Abstract Level Progress Health Attributes". Alphas have states that help us evaluate the progress of a software endeavor from a certain perspective. Together, all the alphas can be used to help you provide a comprehensive evaluation of a software endeavor's progress. You can also use them to guide your team to achieve progress. Other uses of alphas include:

1. You use alphas to determine your team's current state, plan

the next state, track your progress.

2. You adapt your way of working through alphas.
3. You agree on team member involvement and team responsibilities through alphas and their states.

22.3 Separating the essence from the details

As we start to describe the kernel, and practices, we also realize that it is important to make our descriptions as practical to teams as possible. We distinguish between what a team needs to understand at a specific point in time from all the other detailed information.

What teams really need when they are doing their work are just some reminders and pointers which are often provided by coaches. Coaches work closely with team members giving them specific advice based on their present situation. During this time, when team members are very focused on their task at hand, they often do not need detailed explanations and excessive training materials.

This is why we separate what your team needs to understand at a specific point in time from the details. The kernel contains the essence the team needs to be reminded of. If they want to "learn" more, they can consult more elaborate materials, such as guidelines. Even then we make strong attempts to have guidelines no longer than a few pages to let them find what they need quickly. Then, if they really need to learn more, they can consult more in-depth materials such as papers, and books.

23 Key differentiators

Why will Semat make a significant difference to software developers, industry and academia?

1. Being agile when working with methods is a principle we apply throughout Semat. This principle is elaborated in Chapter 21.
2. Separation of Concerns (http://www.semat.org/pub/Main/WebHome/SEMAT_submission_v11.pdf) is a fundamental principle that impacts the complete Semat initiative. This subject is addressed further below and in Chapter 22.
3. Finding the essence of software engineering and embodying it in a kernel gives us a foundation to build our knowledge to run software projects better, faster and with happier customers. The kernel is a key differentiator.
4. Defining separate practices on top of the kernel allows you to grow your method (see Figure 58) in an evolutionary way-- one practice at the time -- to meet your specific needs.

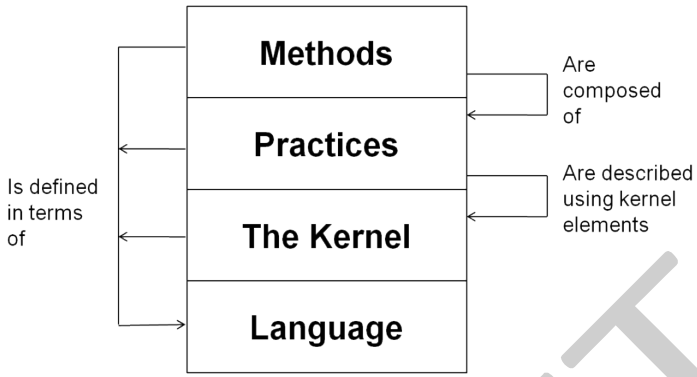


Figure 58 The Method architecture of Semat

5. The whole Semat initiative stands on several key ideas. Here for your reference are the top four ideas listed in priority order:
    1. Alphas are the universals.
    2. Focus on the essentials.
    3. State cards to run development.
    4. Definition cards to capture the essentials.
    5. Making methods scale *up* to large development, *out* to cover the whole lifecycle, and *in* to any level of detail
6. We value more:
    1. *What helps the least experienced developers over what helps the experts* – The least experienced are not encumbered by advanced features. A high proportion of developers in the world will never need advanced features.
    2. *What helps the practitioners over what helps the process engineers* – Process engineers serve practitioners' need, not the other way around. "Practitioners are kings; process engineers are knights serving the kings".

3. *Intuitive, concrete graphical syntax over formal semantics* – Our language is easily understood by developers. While we have formally defined the semantics, our focus is on syntax to aid the developer.
4. *Method use over method definition* – In the past many software initiatives have only paid interest to method definition, namely how to capture methods. They have not focused on how to support the use of a method while actually working in a software endeavor. Thus the methods became shelf-ware, not relevant for the developers running software development. Instead, the kernel approach supports the developers while actually using the method and let they themselves take control of their method and allow the method to evolve as their endeavor progresses.

Part 6 – This is not the end

The grand vision of Semat (as defined by the call for action in 2009 Semat) included a bold statement: "We support a process to refound software engineering … that includes a kernel of widely-agreed elements, extensible for specific uses…". 'Refound' was intentionally a very strong word, a synonym to 'revolutionize', but that was exactly what we felt was needed to get software engineering to where it should be as a discipline. The call for action statement has been signed by many leading experts around the world, by several large corporations and academic institutions and thousands of supporters. That is telling in itself. We must do something fundamentally new. Given this wide support, the expectations on Semat are huge. Have we met them? Let's see where we are.

The grand vision was later interpreted by Ivar Jacobson, Bertrand Meyer, Richard Soley in a Vision statement (http://www.semat.org/pub/Main/WebHome/SEMAT-vision.pdf). The very first step was to achieve a wide agreement on the elements of the kernel and a small language to describe methods, practices and the kernel elements.

With the work now done by Semat and presented in this book, Semat has reached the very first step of the vision. Still the result needs to be presented to the OMG and go through an approval process. Other submissions are also being presented. Ideas will be merged and the outcome will be somewhat modified. Since the Request For Proposal adopted by the OMG (http://www.semat.org/pub/Main/WebHome/SEMAT_submission_v11.pdf) is quite clear about the expected result, the final

result will most likely not be a surprise to anyone. Whatever the result will be, this book will be updated to present the widely agreed upon kernel.

The first mission is soon to be completed

It seems clear that we will get a new standard. The real question is whether this new standard will make a difference to the world. Will we be able to reach the millions of developers which so far have had no or very little interest in working with methods? Why would we succeed this time, when so many previous attempts have just reached at best 100,000 developers? To be successful we need to significantly impact the education in software development. We must get better software faster and with happier customers and this can only happen if what we do can help the practitioners and thus the industry. Moreover, we need to bridge the gap between research and practices, and that can only happen if the framework we suggest helps to better communicate ideas in both directions. We are committed to measure the results of Semat over a three year period[9]. Without reaching a dramatically larger number of developers than what we have done so far, we will not be successful.

As we stated: "This is not the end…"

[9] http://www.semat.org/pub/Main/SematDocuments/Semat_-_Three_Year_Vision13Jan12.pdf

24 …, but perhaps the end of the beginning

"Now this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning"[10].

The work to give concrete value to the community can begin. A team of people within Semat, advised by a larger group of advisors, formulated a three-year vision of Semat, that is a vision up to and including the year 2014. We quote from the three-year vision (http://www.semat.org/pub/Main/SematDocuments/Semat_-_Three_Year_Vision13Jan12.pdf):

"The Semat community through its participants is expected to develop a whole spectrum of products to support its vision. These products are expected to be released separately and at different points in time over the next three years.

The Kernel and the Language – Through the work of the OMG a set of kernel elements and a language based on the agreed common ground for software engineering will be established. Its publication will promote and enable a new ecosystem for methods & practices based on an open standard. It will not be large. These kernel elements will encapsulate what we all agree is essential and used by practitioners on every project when delivering software. It will include terminology in a way that aids understanding and communication with respect to what we all agree about requirements, teams, software system, stakeholder, work,

[10] Winston Churchill.

opportunity, and way of working.

Tools – A collection of tools (including open source) – either as standalone or plug-ins for existing tools – will become available that enable people to author, browse, compose, compare, question, measure and use practices and methods. It is hoped that tools for different areas of concern in software development become interoperable through the vendors' use of the kernel elements and the language. The focus of these tools will be the needs of the developer whose primary concern is developing quality software leading to satisfied customers.

The Practice Market Place – The open standard kernel and the language will enable the publication, cataloging and exchange of practices. The marketplace can in that way advantageously and effectively provide an improved, fluid, and accessible environment where developers are given appropriate freedom to use their preferred ways of working within their specific context.

It will be a tournament where proven practices as well as new innovative ideas are easily accessible.

Curricula – A new and more systematic foundation for teaching software engineering will emerge, which supports learning in academic and professional environments. Curricula based on the kernel, the language, practices and methods will be developed and used both in computer science and software engineering programs in our universities, and in education given by research scientists, and electrical engineers, to name just a few.

Text Books and Papers – New textbooks and reference material to support curricula and personal development based on the kernel and the language will be authored and made publicly available. Many books on practices defined using the kernel that target different level of users will be written to support developers in improving their way of working.

Research – The objective comparable nature and ability to tailor, use, adapt and simulate practices will result in a renaissance of software engineer research. Researchers have a common infrastructure serving as a test-bed and fast deployment of new ideas (extensible practices)."

End-of quote.

All this with the final objective that it will help the industry represented by its practitioners, and the academic world, represented by instructors and researchers, in their missions. As we already have said, the final outcome must lead to a community where we all build significantly better software faster and with more happy customers and users. Moreover, as we also already have said, these results need to be measurable, not just ambitions.

25 When the vision comes true

Assuming our vision comes true, you may ask yourself, "So what?" What value will this work and its products give us as a community? To illustrate we have written four scenarios, representative of our main stakeholders:

1. A team in a small company developing a single application
2. The life of a practitioner
3. Giving the industry a tool for governance
4. Supporting the academic world

The scenarios are basically quotes from the Three Year Vision paper, authored by two of the authors of this book (Ivar Jacobson and Paul E. McMahon) and Shihong Huang, Mira Kajko-Mattsson, Ed Seymour.

25.1 A team in a small company developing a single application

Note: These scenarios are based on looking at least three years into the future.

Imagine a project lead is just about to set up a new project to develop a major release of an existing product.

Her team has a way of working but it is not documented. All have learned about the kernel elements. They want to change their way of working to become more agile, in particular they want to improve the way they work with requirements and test.

The project lead and her team start to work from the kernel reference material. They sketch (e.g. do something similar to a use case model) some of their existing practices, which they want to keep. Since they know their way of working, this is done quickly. Basically they just work through their old terms and synch them up with the names of kernel elements.

Then they go to their companies practice library and select the practices which best meet their needs. They download a tool assisting them to understand the new practices. If needed, they will tailor the practices to fit their specific needs (in this scenario very limited tailoring). The tool also helps them to use the practices in the project, iteration (or sprint) after iteration. They also need (other) tools to support their new practices, but the old tools will still work for their existing practices.

A positive side effect of the practices is that the training material is very effective. The practices in the practice library are actually coming from the Semat marketplace for practices, so its training material has been developed, has been used around the world, and it has been improved over and over again. Some training may also have e-learning facilities.

Because the practice library supplies and references commonly understood practice "states", the team is supportive of the changes since they can grasp more easily the progress and health of the project. From the state checkpoints at any moment they know where they are and where they are going; they know exactly when something is done.

The project lead is satisfied because she sees that the outcome is better, the time to market is shorter and predictable, the costs are lower than estimated, and her customers are happy. And, now other teams want to reuse her experience, so that makes her also happy.

25.2 The life of a practitioner

When a student leaves college today and enters industry, they can directly apply certain skills they have learned (such as JAVA, or C++), but there is a great deal they still must learn. While some of this is unavoidable, such as terminology unique to a given industry, today the use of terms as fundamental as requirements and team can vary widely from one company to the next.

Today we live in a very mobile society where people change employers often throughout their career. When a software engineer moves from one job to another in a different company -- or even in a different part of the same company -- she can take her experience with her, but there is a great deal that must be relearned within that new environment. This can discourage the software practitioner even to the point of making a decision to seek a different career path.

Establishing a kernel is not about creating a standard that excludes certain users. A set of kernel elements having industry wide consensus encourages new approaches appropriate to the job at hand.

So what does this mean for the software practitioner? Software practitioners of the future will have more opportunities for employment. They will experience the freedom of knowing they have greater cross company mobility because their job satisfaction is less dependent on their educational ramp up speed.

They will know that what they learn in the university can be counted on when they move into industry and as they move from one company to another within the industry-- or from one project to another within a company. Knowing they have learned the essentials will also bring greater self-confidence and self-fulfillment. This is because the software developer will be able to

focus more of their limited time on the unique aspects of the job that brings greater value to themselves, their employer and their customer. They will also know that if they inadvertently come into a dysfunctional software endeavor that they have the ability to articulate to managers exactly how the dysfunctional situation can be evaluated, made better, faster and happier. And they can do this using language that the managers already know because it will have been widely accepted across the software industry.

25.3 Giving industry a tool for governance

All organizations care about the health of their projects, and want to know what to do in the face of trouble. Effective measures can help managers find problems early and know when action is needed. Measures are integral to the kernel approach.

One-size-fits-all view of corporate standards cannot provide the answer. Varying factors among teams, projects, products, and organizations must be considered and dealt with by the team.

To the program manager working in the software industry the kernel provides a consistent accepted reference of essentials across all her projects, irrespective of size or situation.

To the team leader establishing a way of working suited to her new project, an understanding of the kernel can provide the guidance to select appropriate ways of working to support her team and meet her corporate goals. It also means less ramp-up time when assigning people to projects because they have a common ground and a common vocabulary to build their work on.

To the process professional the kernel provides a means to communicate how an organization works in a format easily

digestible by established employees and new recruits alike. An easily digestible format improves adoption, facilitates reuse, and provides teams with a means to integrate what we all know works into a team's accepted way of working.

25.4 Supporting the academic world

Looking at today's software engineering education, different universities and professors have different requirements and interpretations related to how software engineering should be taught, and what should be taught. Some accreditation guidelines exist. For examples, the *Accreditation Board for Engineering and Technology* (ABET) in the U.S, and SWEBOK Curriculum Guidelines for Undergraduate Degree Program in Software Engineering. However, these tend to provide guidelines at a very high principle level leaving the implementation details to individual schools and professors. The lack of a core consensual set of kernel elements of software engineering results in a wide range of education approaches without a clear, consistent basis.

This situation leads to students from different education backgrounds having different skill sets and understandings, which fail to meet industry's need. It also leads to research chasing fads rather than following a clear balanced roadmap.

To academia a consensual set of kernel elements can provide a foundation to a) teach software engineering, b) design software engineering curricula, and c) demonstrate to students the pros and cons of different ways of working. A core consensual set of kernel elements ensures the essentials of software engineering are taught in a uniform way across different universities and education programs.

From a research perspective, a consensual set of kernel elements

provides a reference for the conduct of experiments on different software engineering approaches relevant to real world problems, and a solid foundation to aid the separation of hypes from reality.

Appendix A. Concepts and Notation

We provide a summary of the terminology and their notation here for your convenience.

Concept	Icon	Description
Method		A method is a team's way of conducting its work in a software development endeavor.
Kernel		A kernel is a set of elements used to form a common ground for describing a software engineering endeavor.
Alpha		An alpha is an element (an attribute) of a software engineering endeavor, which has a state relevant to assess the progress and health of the endeavor. Alpha is an acronym and means Abstract-Level Progress and health Attribute.
Activity Space		An activity space defines something that should be done in terms of a high abstract-level objectives description.
Work Product		A work product is an artifact of value and relevance for a software engineering endeavor.

Practice		A practice is a repeatable approach to doing something with a specific purpose in mind.

Appendix B. What's does this book cover with respect to the kernel

In this book, we have introduced a number of concepts in the kernel, and how you can use them in your software endeavor. However, as we do so, we inevitably also add our experiences and adaptation. We would also like to say that this book does not describe the entire kernel, but an important part of it, namely: alphas, their states, checklists and the ability to extend the kernel. So, as a reader, you might like to know what parts of the kernel are left outside this book, and what are the things we add on top of the kernel in this book.

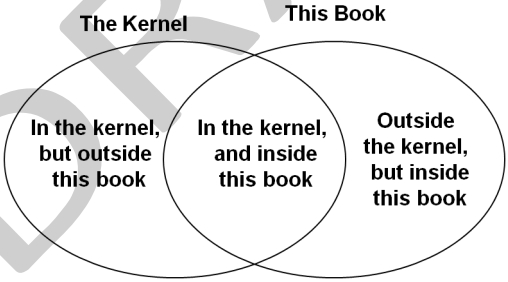


Figure 59 Scope of this book

B.1 Inside the kernel, and inside this book

Our main emphases of the kernel in this book are primarily in two areas:

1. **Kernel usage**. In this book, we introduced the kernel alphas (i.e. opportunity, stakeholders, requirements, etc.) and how you use the alphas, their states, and checklists to help you run a software endeavor.
2. **Kernel extension**. In this book, we also introduced how you extend the kernel by adding practices to yield the method, which you need to help run your endeavor.

B.2 Outside the kernel, but inside this book

In this book, we demonstrate how you can run development with the help of the kernel using stories. The stories include the way each story run's development, which is not part of the kernel.

Only the alpha states and their criteria (checklists) are part of the kernel. These are physically represented by the state cards. The techniques and adaptations used in the stories and examples are outside the kernel. Examples of items discussed in this book that are outside the kernel include:

1. Task boards and Kanbans
2. Iterative development
3. Lifecycles
4. Requirement-Items

5. Specific practices
6. All dimensions of scaling

B.3 Inside the kernel, but outside this book

There are a number of things inside the kernel, which we did not discuss much in this book. The full scope of the kernel can be described using the method architecture of Semat (see Figure 58 in Chapter 22).

The kernel language describes the language constructs. We used a number of constructs in this book, namely, alphas and their states, methods, practices, and the kernel itself. The notation (see Appendix A), the wellformedness rules governing the use of the notation, and the meaning of the notation are part of the kernel language. Thus there is a lot more to the kernel language than just the notation. There are also rules governing their practical use.

The kernel comprises a set of elements (which are known as universals) defined using the kernel language. The specific kernel alphas such as opportunity, stakeholders and work are part of the kernel.

So, what's in the kernel that we did not discuss in this book?

1. Kernel language.
2. Areas of concerns.
3. Activity spaces.
4. Competencies.

Appendix C. Bibliography

This appendix is a work in progress with much more planned to come in next draft of the book.

C.1 Semat Working Documents

http://www.semat.org/bin/view/Main/SematDocuments

[1] Ivar Jacobson, Shihong Huang, Mira Kajko-Mattsson, Paul McMahon, Ed Seymour, Semat: Three Year Vision, in Programming and Computer Software Volume 38, Number 1, 1-12, DOI: 10.1134/S0361768812010021.

http://www.semat.org/pub/Main/SematDocuments/Semat_-_Three_Year_Vision13Jan12.pdf

http://www.springerlink.com/content/n1519665x245u7lg/

[2] Ivar Jacobson. Discover the Essence of Software Engineering. CSI Communications, July 2011.

http://www.semat.org/pub/Main/SematDocuments/Technical_Trends_3.pdf

[3] The approved version of the Request For Proposal (RFP) "A Foundation for the Agile Creation and Enactment of Software Engineering Methods" for a kernel and a language. (This supersedes the full draft previously available).

http://www.semat.org/pub/Main/WebHome/Foundation_for_SE_Methods_RFP_ad-11-06-26.pdf

(Reading tip: Sections 1--5 introduce general rules for OMG

proposals and standards; the actual technical content starts in Section 6).

[4] The highlights of the Request For Proposal (RFP) "A Foundation for the Agile Creation and Enactment of Software Engineering Methods" for a kernel and a language.

http://www.semat.org/pub/Main/SematDocuments/OMG_RFP_Highlights.pdf

[5] A full draft of the Request For Proposal (RFP) "A Foundation for the Agile Creation and Enactment of Software Engineering Methods" for a kernel and a language.

http://www.semat.org/pub/Main/WebHome/ADTF_SEMAT_RFP_Brief_version.pdf

[6] The SEMAT vision statement by Ivar Jacobson, Bertrand Meyer, and Richard Soley.

http://www.semat.org/pub/Main/WebHome/SEMAT-vision.pdf

[7] Ivar Jacobson and Bertrand Meyer. Methods Need Theory. Dr. Dobb's Journal, 6 August 2009.

http://www.ddj.com/architect/219100242

[8] Ivar Jacobson and Ian Spence. Why We Need a Theory for Software Engineering. Dr. Dobb's Journal, 2 October 2009.

http://www.ddj.com/architect/220300840

[9] Ivar Jacobson, Bertrand Meyer, and Richard Soley. The SEMAT Initiative: A Call for Action. Dr. Dobb's Journal, 10 December 2009.

http://www.ddj.com/architect/222001342

C.2 SEMAT: Other Documents and References

[10] Funding opportunity from Microsoft with goals relevant to SEMAT: Microsoft SEIF Awards

http://research.microsoft.com/en-us/collaboration/focus/cs/seif.aspx

[11] Ivar Jacobson, Pan Wei Ng, and Ian Spence. Enough of processes - Lets do practices. Journal of Object Technology, 6(6):41-67, 2007.

http://www.semat.org/pub/Main/PubsandRefs/enough_of_processes.pdf

[12] Philippe Kruchten. A conceptual model of software development. In Software project management with OpenUP, draft of April 2007.

http://www.semat.org/pub/Main/PubsandRefs/Kruchten_conceptual_model.pdf

[13] Watts S. Humphrey. Why teams need operational processes. November 2009.

http://www.semat.org/pub/Main/PubsandRefs/operational_processes.docx

[14] Watts S. Humphrey. The software quality challenge. Journal of Defense Software Engineering, June 2008.

http://www.semat.org/pub/Main/PubsandRefs/Humphrey_SoftwareQuality.pdf

[15] Alan W. Brown and John A. McDermin. The art and science of software architecture. In European Conference on Software Architectures, Lecture Notes in Computer Science

4758:237-256, 2007.

http://www.semat.org/pub/Main/PubsandRefs/the_art_and_science_of_software_architecture.pdf

[16] Barry Boehm and Apurva Jain. An initial theory of value-based software engineering.In Biffl et al. (eds.), Value-based Software Engineering, Springer, 2005.

http://www.semat.org/pub/Main/PubsandRefs/Chapter2_VBSEv510.doc

[17] The EA-MDE project: Evaluating Success/Failure Factors of MDE in Industry. Reference person: Jon Whittle.

http://www.comp.lancs.ac.uk/~eamde

[18] Geert Bellekens. The Modelling method.

http://themodelfactory.org/Modelling_Method

[19] Capers Jones. Errors and omissions in software historical data: separating fact from fiction.

http://www.semat.org/pub/Main/PubsandRefs/MeasurementErrors2009.doc

[20] Tom Gilb. Undergraduate basics for systems engineering.

http://www.semat.org/pub/Main/PubsandRefs/basics_of_systems_engineering.pdf

[21] Tom Gilb. A conceptual glossary for systems engineering.

http://www.semat.org/pub/Main/PubsandRefs/conceptual_glossary.pdf

[22] Capers Jones. Scoring and evaluating software methods, practices, and results.

http://www.semat.org/pub/Main/PubsandRefs/ScoringMe

thod2008.doc

[23] Capers Jones. Analyzing the tools of software engineering.

http://www.semat.org/pub/Main/PubsandRefs/Tools2010.doc

[24] Capers Jones. Software quality in 2010: A survey of the state of the art.

http://www.semat.org/pub/Main/PubsandRefs/software_quality_survey_2010.ppt

[25] Capers Jones. A short history of the cost per defect metric. 5 May 2009.

http://www.semat.org/pub/Main/PubsandRefs/a_short_history_of_the_cost_per_defect_metric.doc

[26] Alistair Cockburn. Keynote presentation at AGILE 2009.

http://www.infoq.com/presentations/cockburn-bury-not-praise-agile

[27] Alistair Cockburn. Foundations for Software Engineering.

http://alistair.cockburn.us/Foundations+for+Software+Engineering

[28] Alistair Cockburn. Software engineering in the 21st century presentation.

http://alistair.cockburn.us/Software+engineering+in+the+21st+century.ppt

[29] Alistair Cockburn. From Agile Development to the New Software Engineering.

http://alistair.cockburn.us/From+Agile+Development+to+the+New+Software+Engineering

[30] Capers Jones. Position paper: Critical problems in software engineering. 22 December 2009.

http://www.semat.org/pub/Main/PubsandRefs/Jones-Position2010.doc

[31] Dines Bjørner. From domains to requirements. In Concurrency, Graphs and Models, pp. 278-300, 2008.

http://www.semat.org/pub/Main/PubsandRefs/from-domains-to-requirements.pdf

[32] Dines Bjørner. The Triptych process model -- Process assessment and improvement. From a seminar at Tokyo University in Honour of Kouichi Kishida's 70th Anniversary, 2006.

http://www.semat.org/pub/Main/PubsandRefs/believable-software-management.pdf

[33] Daniel Jackson. A direct path to dependable software Communications of the ACM, 52(4), 2009.

http://www.semat.org/pub/Main/PubsandRefs/direct-path-to-dependable-software.pdf

C.3 Other References

[34] Pan-Wei Ng, Light Weight Application Lifecycle Management Using State-Cards in Agile Journal, 12 October 2010

http://www.agilejournal.com/articles/columns/column-articles/3325-light-weight-application-lifecycle-management-using-state-cards

[35] Ivar Jacobson, Ian Spence, Kurt Bittner, Use-Case 2.0 ebook

http://www.ivarjacobson.com/use_case2.0_ebook/

[36] Betrand Meyer, Touch of Class: Learning to Program Well Using Object Technology and Design by Contract, 876 + lxiv pages, Springer-Verlag, August 2009. Translations: Russian.

[37] Meyer, Bertrand (1997). Object-Oriented Software Construction, second edition. Prentice Hall. ISBN 0-13-629155-4