Tear Down the Method Prisons!

Set Free the Practices!

ESSENCE: A NEW WAY OF THINKING THAT PROMISES TO LIBERATE THE PRACTICES AND ENABLE TRUE LEARNING ORGANIZATIONS

IVAR JACOBSON AND ROLY STIMSON, IJI

The way we develop software struggles to keep pace with changes in technology and business. Even with the rise of agile, people still flip-flop from one branded method to another, throwing away the good with the bad and behaving more like religious cultists than like scientists.

The problem is that the professional practices that have been developed and refined over many years, and that together represent our shared industry knowledge and experience, are all too often imprisoned within proprietary method jails. The only option that development organizations and teams see themselves as having is to adopt this method or that method wholesale, and to reject all others—whereas, in fact, what is needed is for organizations and teams to be free to select the professional practices that they need, from wherever these may be defined, and use them in whatever permutations and combinations are appropriate to meet the exact set of circumstances and challenges they face.

There is a simple way to break out of this cycle of

unhealthy competition among methods—which are more similar than they are different—and that is to free the practices from their method prisons. Free the practices to rise and fall on their own merits. Free the practices so that teams can experiment, innovate, and plug and play with proven practices to create the way of working that they need today and to evolve seamlessly into the one they need tomorrow.

This article explains why we need to break out of this repetitive dysfunctional behavior, and it introduces Essence, a new way of thinking that promises to free the practices from their method prisons and thus enable true learning organizations.

INTRODUCTION

The world has developed software for more than 50 years. The software industry as a whole has been very successful. We could choose to be happy and continue doing what we are doing. Under the surface, however, everything is not as beautiful: too many endeavors have failed, quality in all areas is generally too low, costs are too high, time to market is too long, etc. Obviously, we need better ways of working or, in other words, we need better methods.

Here a *method* provides guidance for all the things you need to do when developing software. In a keynote speech at the 2003 XP conference in New Orleans, Ivar Jacobson suggested a hypothesis that even if the number of methods in the world is huge, it seemed that all of them were just *compositions* of a much smaller collection of potentially reusable "mini-methods," maybe a few hundred in total.

These distinct mini-methods are what many people would call *practices*. In this article, the term *method* stands for related terms such as process, methodology, and method framework, even if these terms, strictly speaking, have different meanings.

As an industry we have searched for better methods, following a zig-zag path moving from paradigm to paradigm and from method to method, changing very much like the fashion industry inspires wardrobe changes. For example, during the 1970s and 1980s the Structured Methods dominated; in the 1990s the Object or Component Methods were favored; and since around 2000 the Agile Methods have ruled. Right now, the top interest is in Scaling Agile Methods. There are many competing methods in this space: for example, SAFe (Scaled Agile Framework), Nexus, and LeSS (Large-Scale Scrum). They are all popular and used by organizations around the world. They deliver value to their user organizations in both overlapping ways and in specific ways.

Now, if they all are good, what are the problems?

1. Methods are essentially monolithic

Maybe the most limiting factor is that most methods are monolithic, meaning they are not designed so that you can easily exchange one practice with another from another method, and keep the other practices intact. A method may be modular, but the modules are unique for the method and not reusable by another method.

2. Methods have a homegrown presentation

Each method has its own unique user experience and its own structure, and uses its own style and terminology to describe its selected practices. The owners of the method have decided on these important aspects for themselves without following any standard. As a result, its practices are incompatible with practices from other methods. Comparing or mixing methods is like comparing or mixing cultures—if not impossible, then very hard, at best.

3. Methods have no common ground

Though every method has some unique practices, it has a lot more in common with others. After all, they all deal with software so they should share a lot. The fact is, what they share is hidden and not explicit, so without deep inspection it seems they share almost nothing, not even the basics such as: What is Software? What is Software Development? What are Requirements, Design, Test? What are Team, Way of Working?

4. Practices are locked in method prisons

Today the practices within a method are locked into that method—*they are in method prisons*—and cannot easily be reused in other methods. In fact, to get a practice incorporated in a method most likely requires that it be rewritten to fit the homegrown style of that method.

5. Method prisons are controlled by method wardens—gurus

The gurus control which practices should be combined into their methods. They have extended their methods

with practices "borrowed" and "improved" from other methods. The quotation marks indicate that it is not exactly "borrowing" that happens, and it is not always "improving," but because of misunderstanding or reinterpretation of the original practice, it can become a perversion or confusion of the original.

The method reflects the particular perspectives, prejudices, and experiences of its guru, and may not be what the development community has collectively learned.

Let's be clear that the gurus are not necessarily striving for the position they placed them in. Since today all practices are described as what we have called homegrown, a guru who wants to "borrow" a practice from somewhere else is forced to rewrite the "borrowed" practice to make it fit within his or her method, and while doing this, "improves" it.

6. We have been in a method war for 50 years

The owners of practices that are "borrowed" and "improved" naturally feel that there has been "misunderstanding" of their practices. This so-called "borrowing" doesn't stimulate collaboration among gurus. Given the investment in time and capital by the owners of these practices, they must defend their turf, resulting in method wars. These wars started 50 years ago and continue with no clear end in sight.

These six problems illustrate how immature our way of working with methods is. We call these problems collectively "the most foolish thing in the world" with "the world" of course referring to software development methods.

WHAT DO WE NEED TO DO TO GET OUT OF THIS FOOLISHNESS?

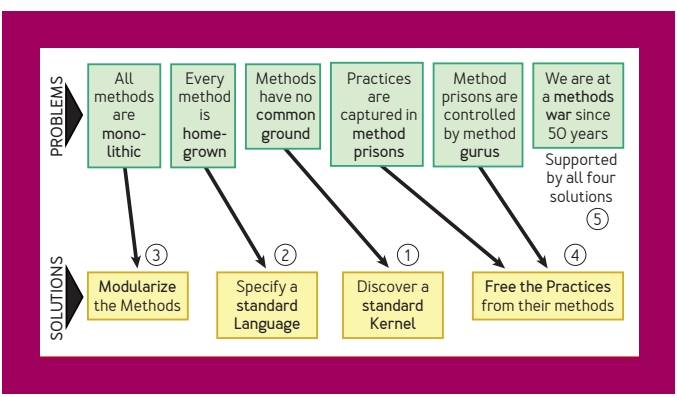
The six problems identified in the previous section have been addressed as indicated in figure 1.

1. Discover a standard kernel

It was obvious that a standard "kernel" would need to contain "things" that are or should be prevalent in any method,[1] such as what are the essential things we always work with and always produce, and what are the essential competencies we always need when developing software? The team that designed the kernel started off specifying the criteria, principles, and features that should guide their work in creating the standard. It is out of the scope of this article to present these in detail, but let us mention some (the quoted material comes from blogs and work documents):

The essential things, the kernel elements, mentioned

FIGURE 1: **PROBLEMS WITH METHODS AND THEIR SOLUTIONS**



above should be "applicable no matter the size or scale of the software under development, nor the size, scale, or style of the team involved in the development."

"In essence it [the kernel] provides a practice-independent framework for thinking and reasoning about the practices we have and the practices we need. The goal of the kernel is to establish a shared understanding of what is at the heart of software development."

The kernel elements should be: universal, significant, relevant, defined precisely, actionable, assessable, and comprehensive. *Relevant* is explained as "available for application by all software engineers, regardless of background, and methodological camp (if any)," and *comprehensive* "applies to the collection of the kernel elements; together they must capture the essence of software engineering, providing a map that supports the crucial practices, patterns, and methods of software engineering teams."

The following general principles are deemed essential to finding a kernel: quality, simplicity, theory, realism and scalability, justification, falsifiability, forward-looking perspective, modularity, and self-improvement. *Theory* means "the kernel shall rest on a solid, rigorous theoretical basis;" *realism and scalability* mean "the kernel shall be applicable by practical projects, including large projects, and based where possible on proven techniques;" and *self-improvement* means "the kernel shall be accompanied by mechanisms enabling its own evolution."

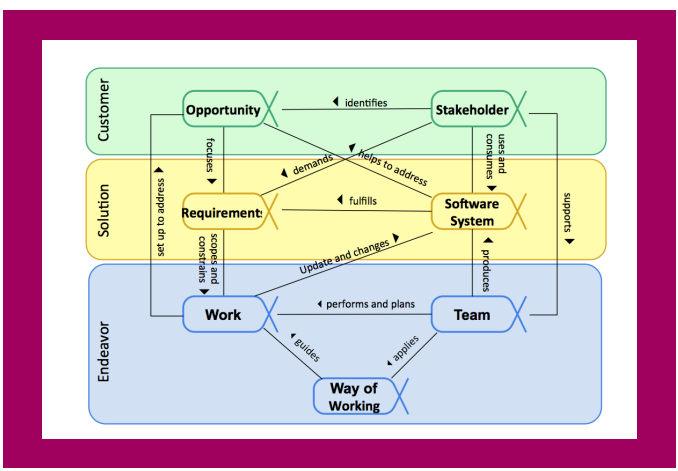
Moreover, the kernel should have these features: practice independent, lifecycle independent, programming language independent, concise, scalable, extensible, and

formally specified. *Scalable* is explained as the kernel supporting the very smallest of projects—one person developing one system for one customer—but it must also support the largest of projects, in which there may be systems-of-systems, teams-of-teams, and projects-of-projects. *Extensible* means the kernel needs to possess the ability to add practices, details, and coverage, and to add lifecycle management and to tailor the kernel itself to be domain specific or to integrate the software development work into a larger endeavor.

With these guidelines, the team set out to find the kernel. Figure 2 shows the essential things to work with—the alphas.[1]

The alphas exist in three different areas of concern: Customer, Solution, and Endeavor. The alphas are not tangible, so they don't represent work products such as

FIGURE 2: **THE ALPHAS AND THEIR RELATIONSHIPS**



documents, but they have states that tell in which state of the lifecycle of the alpha you are. Each state is defined by a checklist that is agnostic to any specific method. The checklist doesn't measure which activities have been performed or which documents have been written, but they measure real outcome. For example, the Team alpha has these checklist elements in state Formed: Enough members recruited, Roles understood, How to work understood, etc. Thus, the alphas are agnostic to any method.

Apart from the alphas, the kernel has other types of elements, but they are not key to following the discussion in this article.

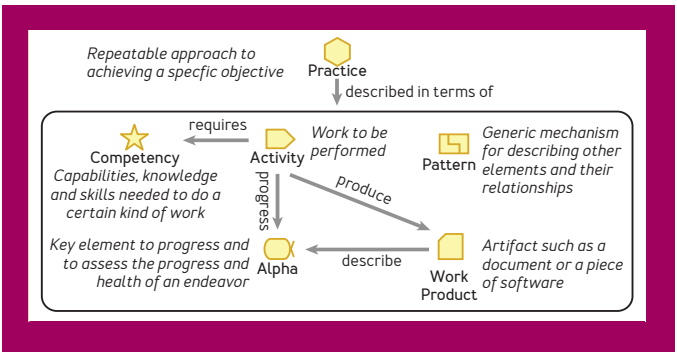
2. Specify a standard language

To be able to reuse existing practices, the practices cannot be described in a homegrown way, specific to the method that uses it. We need a common language—a lingua franca—a formal language with syntax and semantics.

As with the kernel, what was expected was formulated as requirements of the language. For example: "The language should be designed for the developer community (not just process engineers and academics)," which is a requirement asking for a simple, visual, intuitive, and engaging user experience in working with methods and practices. With the language in figure 3, we would be able to describe practices so they are reusable by any method.

The Language and the Kernel together form a Common Ground, something we have been missing for all these years of software engineering. In 2014, OMG (Object Management Group) adopted it as a standard, called Essence.[4]

FIGURE 3: **THE ELEMENTS IN THE LANGUAGE**



3. Modularize the methods

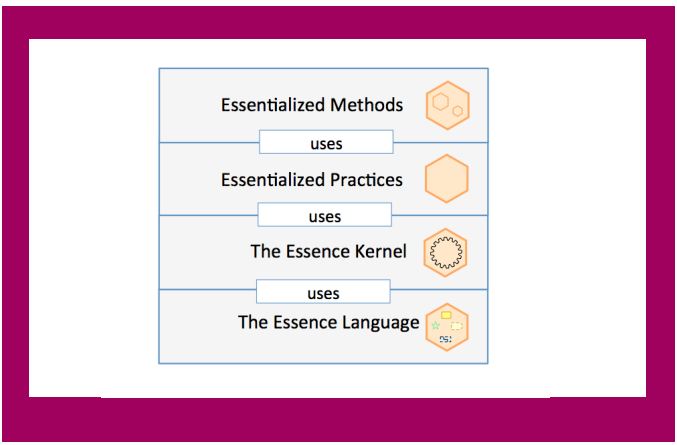
The team needed to agree on what a practice is. They said, for example: "A practice is a separate concern of a method"; "every practice, unless explicitly defined as a continuous activity, has a clear beginning and an end"; and "every practice brings defined value to its stakeholders." The team designed the method architecture as shown in figure 4. Practices became First-Class Citizens.

The two lowest layers are represented by Essence—the language and the kernel. The third layer consists of practices described using Essence, with the kernel being the standard vocabulary.

4. Free the practices from their methods

Essentialization of a method means: (1) Identify the (often hidden) practices of the method; (2) Separate them from one another (even if they are not independent); (3) Describe each practice using Essence (kernel and language); (4) Build

FIGURE 4: **ESSENTIALIZED PRACTICES AND METHODS**



and preserve a sound practice architecture (resolving dependencies among the practices) to facilitate flexible recomposition of the practices; and (5) Ensure that the method owner agrees that the essentialization truly reflects their intentions, or modifies until this condition is fulfilled. The latter is the hardest part of the job for obvious reasons.

Essentialization unlocks the practices from the methods and makes them free to select and create any method needing them.

5. No more method wars

Addressing the problems as suggested in points 1-4 in this section should lead to a significant reduction in method wars. The battle will no longer be about methods. Instead, the debate will move to a discussion of which practices are most suitable in particular situations. This is where the

battle should be fought—among specialists on subjects about which they are real experts. Today the wars are less focused. There is no need to create new cultures with their own values. The discussion should invite everyone with something to say.

HOW TO ESCAPE THE FOOLISH PROBLEMS

Moving from idea to tangible result is a long journey. We first have to find a common ground.

Essence—the common ground of software engineering

As a response to "the most foolish thing in the world," the work on an escape route from the many problems started in 2006 at IJI (Ivar Jacobson International). In 2009 the SEMAT (Software Engineering Method and Theory) community was founded, and in 2011 the work was transferred to OMG, which eventually gave rise to a standard common ground in software engineering called Essence.[4] Essence became an adopted standard in 2014. Thus, Essence didn't come like a flash from the brow of Zeus, but was carefully designed based on a vision statement written by the founders of SEMAT in 2010.

We were also inspired by Michelangelo: "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." We felt that from all this mass of methods we had to find the essence, so we paraphrased Michelangelo: *"We are liberating the essence from the burden of the whole."*

And by Antoine de Saint-Exupéry: "You have achieved

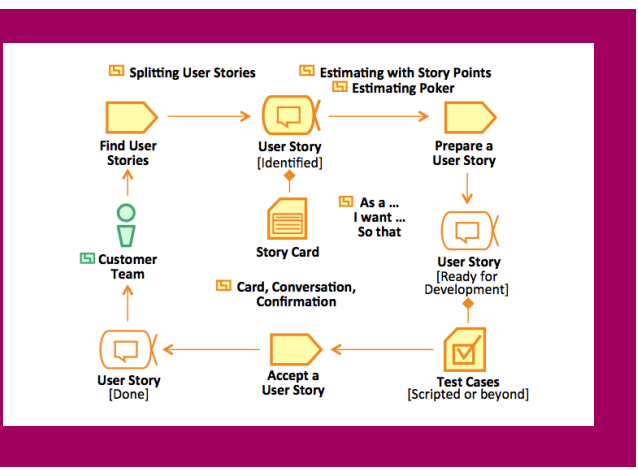
perfection not when there is nothing left to add, but when there is nothing left to take away." We took a very conservative approach in deciding what should be in the kernel and what should be outside the kernel. It is easier to add new elements to the kernel than to take them away.

Using Essence

Instead of giving the whole theory behind Essence, which we have done many times,[1] we will show its usage by presenting a practice described on top of Essence—using Essence as a platform to present the practice. We have selected User Stories as an example of an Essence practice, here calling it User Story Essentials, shown as a Big Picture in figure 5.

The flow of this practice is as follows:

FIGURE 5: **THE USER STORY ESSENTIALS PRACTICE**



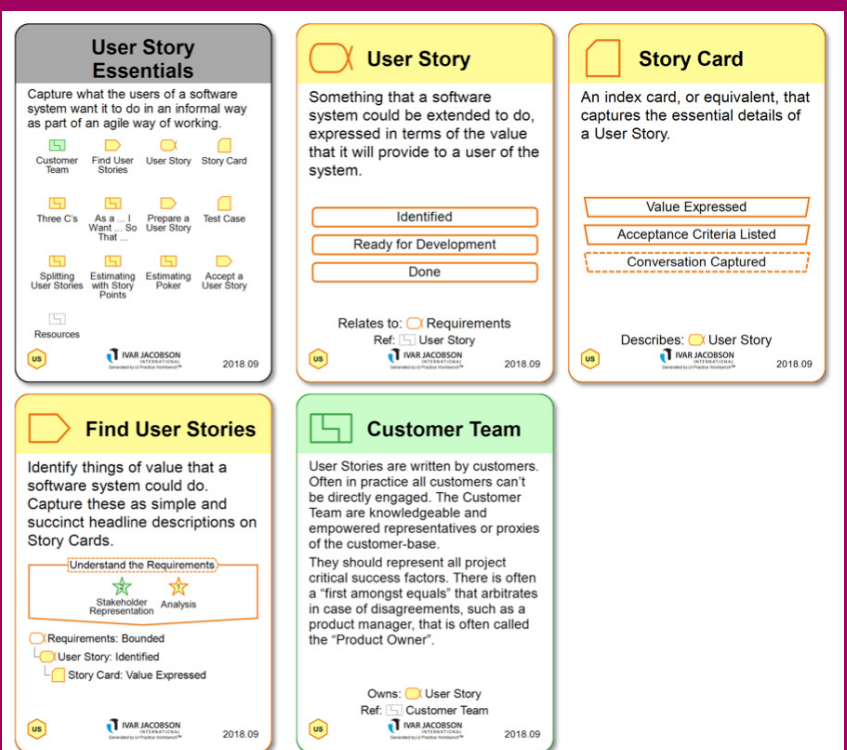
- First we need to Find User Stories. This activity identifies one or more User Stories, each documented by a Story Card with just enough information to ensure that the User Story has its value expressed.
- On a story-by-story basis, we will select a User Story that we wish to get done next, and then we use the Prepare a User Story activity to get it ready for development, which also involves elaborating the associated Test Cases. (Note that we use the convention that the User Story alpha appears in outline form in its later stages to indicate that this is not a new element but the same User Story as before it progressed through its states.)
- The final activity that this practice describes is how we work to Accept a User Story, the successful completion of which gets the User Story done.

It is not our intention to describe the entire User Story practice but to provide a good understanding of what an essentialized practice looks like.

An essentialized practice or method is described using Essence, which focuses the description on what is essential. It doesn't mean changing the intent of the practice or the method. Essentialization provides significant value. We as a community can create many practices coming from many different methods. Teams can mix and match practices from many methods to get a method they want. If you have an idea for a new practice, you can just focus on essentializing that practice and make it available for others to select; you don't need to reinvent the wheel to create your own method. This liberates that practice from monolithic methods, and it will open up the method prisons and let companies and teams get out to an open world.

The User Story practice when essentialized is presented as a set of 14 cards. Figure 6 shows a representative set of five cards, briefly described here.

FIGURE 6: **FIVE CARDS FORM THE USER STORY ESSENTIALS PRACTICE**



User Story Essentials (Index Card)

This provides:

- A brief description that gives insight into why (benefits) and when (applicability) we might use the practice.
- A contents listing showing named practice element icons for all the elements within the practice (each of which is described with its own card).

Note that the color coding gives an immediate visual indication of the scope of application of the practice. In this case we see that the practice consists of:

- Mainly yellow cards, the Essence color coding for the Solution area of concern—telling us that this practice is concerned with the software system we are building and its requirements.
- One green card, the Essence color coding for the Customer area of concern—telling us that the practice also concerns itself with how we interact with customer area concerns such as the Opportunity and the Stakeholders.
- No blue cards. Essence has three areas of concerns, the third color-coded in blue standing for the Endeavor area of concern. The User Story Essentials practice has no cards in this area.

Note also that in this case there is a strong separation of concerns between the Solution and Customer areas of concern that User Story Essentials addresses and the Endeavor area of concern, which includes concerns such as the Team and how we manage the Work. The practical consequence is that this practice can be used with any number of different management practices that mainly

operate in the blue Endeavor area of concern, such as a timeboxed Scrum-style approach to work management or a continuous-flow Kanban-style approach.

Essentialized practices can be described at different levels of detail. The cards in this practice don't attempt to provide all the information that, for example, a novice team would need to successfully apply the practice. If history has taught us anything, it is:

- No amount of written process enables novices to succeed without expert support.
- The more words there are, the less likely that any of them will be read.
- Instead of "borrowing and rewriting" other people's words when it comes to the more voluminous detailed supporting guidance, it is better simply to reference the original sources of this guidance.

Essentialized practices such as this one are based on the principle that novice teams need support from expert coaches to be successful. The cards become a tool for expert coaches to use to help teams adopt, adapt, and assess their team practices, or for expert teams to use in the same way.

User Story (Alpha)

This is a key thing that we work with, that we need to progress, and the progression of which is a key trackable status indicator for the project—think of alphas as what you expect to see flowing across Kanban boards, having:

- A brief description that makes clear what this thing is and what it is used for.

- A sequence of states that the item is progressed through—in this case from being Identified, through to being Ready for Development, through to being Done. (Think of these as candidate columns on a Kanban board—although teams may want to represent other interim states as well depending on their local working practices.)
- The "parent" kernel alpha to which the multiple User Stories all relate (the Requirements in this case).

Story Card (Work Product)

Work product cards give guidance on the real physical things that we should produce to make the essential information visible—in this case a key defining feature of the User Story approach is that we use something of very limited "real estate," an index card or electronic equivalent, as the mechanism for capturing the headline information about what we want to build into the Software System. The work product has:

- A brief description.
- A Level of Detail that we progressively elaborate—in this case indicating that we initially ensure that we have captured and communicated the value of the User Story, and that we also need to continue at some stage to list the acceptance criteria - the dotted outline of the third level of detail indicating that we may or may not capture associated conversations—for example, in an electronic tool if we are a distributed team.
- The alpha that the work product describes—a User Story in this case.

Find User Stories (Activity)

This gives guidance to a team on what they should actually do, in terms of (in this case):

- A description of the activity.
- An indication of the *competencies* and *competency levels* that we need for the activity to be executed successfully. For example, the card requires Stakeholder Representative competency at level 2 and Analysis competency at level 1 (all of which are defined in the Essence kernel).
- An indication of the *activity space* in which the activity operates (i.e., what "kind of thing it helps us do" (in this case "Understand the Requirements").
- An indication of the purpose of the activity expressed as the end state that it achieves—in this case a User Story is Identified and a physical Story Card is produced that describes the value associated with the User Story.

Note that activities are critical because without them nothing actually ever gets done; it is remarkable how many traditional methods inundate readers with posturing and theorizing without actually giving them what they need, which is clear advice on what they should actually do!

Customer Team (Pattern)

Patterns give supporting guidance relating to other elements and/or how these relate to each other, in terms of (in this case):

- Textual description, encapsulating the critical aspects of the guidance that the pattern provides.
- Named associations, showing which other element or

elements the pattern relates to primarily—in this case the User Story.

- A reference link to a named reference on the resources card, which in turn provides one or more pointers to sources of more guidance or information.

Putting it all together

We have now described a representative subset of the different types of cards used in the User Story Essentials practice, so we will not describe the other cards because the story would rapidly become familiar and repetitious. This is part of the value of using a simple, standard language to express all our practice guidance.

Now that we understand what all the cards mean, we also need to understand at a high level how the whole practice works. The cards themselves give us all the clues we need about how the elements fit together to provide an end-to-end story—which activities progress which alphas and produce which work products—but it is also useful to tell the joined-up story in terms of end-to-end flow through the different activities. Figure 5 gives just such an overview of the flow through the practice, which we repeat and summarize here:

- First we need to find User Stories. This brings one or more User Stories into existence in the initial Identified state, each documented by a Story Card with just enough information to ensure that the User Story has its Value Expressed.
- On a story-by-story basis, we will select a User Story that we wish to get done next, and use the Prepare a User Story activity to progress the User Story to be Ready for Development. This involves ensuring that we have the

Acceptance Criteria Listed on the Story Card, during which we may also get any supporting Conversation Captured. As part of this same activity we also elaborate the associated Test Cases.

- The final activity that this practice describes is how we work to Accept a User Story, the successful completion of which moves the User Story to the Done state.

Notice that this chaining of activities, primarily via the state of the things that they progress, does not over-constrain the overall flow. It does not, for example, imply a single-pass, strictly sequential flow. We might iterate the different activities for different User Stories in different ways. Exactly how we do so may be further constrained as part of adopting other practices. For example, if we use the User Story practice in conjunction with Scrum, as is very common, we may agree to the following general rules as a team:

- Do the Find User Stories activity before we start our First Sprint, but also allow this to happen on an *ad hoc* basis ongoing.
- Do the Prepare a User Story activity before the first Sprint and then during each Sprint for the User Stories for the next Sprint, in time for Sprint Planning.
- Aim to Accept a User Story as soon as it is done, to get all the User Stories selected for the Sprint Done before the end of the Sprint Review.

To summarize the general rules and principles illustrated here:

Essence distinguishes between elements of health and progress versus elements of documentation. The former

are known as *alphas,* while the latter are known as *work products*. Each alpha has a lifecycle moving from one *alpha state* to another. *Work products* are the tangible things that describe an alpha and give evidence to its alpha states; they are what practitioners produce when conducting software engineering activities, such as requirement specifications, design models, code, and so on. An *Activity* is required to achieve anything, including progressing Alphas and producing or updating a Work Product. *Activity spaces* organize activities. To conduct an activity requires specific *Competencies*. *Patterns* are solutions to typical problems. An example of a pattern is a role, which is a solution to the problem of outlining work responsibilities.

Essence, in defining only the generic standard "common ground," defines no work products, activities, or patterns, since these are all practice-dependent. It defines seven alphas each with defined states, 15 activity spaces, and six competencies, which are all practice agnostic. Practices defined on top of Essence introduce new elements or subtypes of the standard kernel element types.

Key features and benefits of essentialized practices

Some of the key features and benefits of essentialized practices as illustrated by the User Story Essentials example, are:

- The practice is tightly scoped. It tells us how to do one thing well. In particular, the practice does not constrain or limit any of our other choices regarding other practices we may want to use to handle other aspects of our endeavor (Scrum, Kanban, etc.).
- The practice is concisely expressed. Only a subset of the

cards are shown in the User Story Essentials example, but physically the cards in the practice together represent roughly the equivalent of the size of a sheet of A4 paper.

- The practice is accessible and can be interacted with through the cards, which are used in all kinds of ways. This includes making the team's way of working instantly visible, self-assessing the adequacy of local practices, and prioritizing improvement areas.
- The practice is expressed in a simple, standard way. When you understand these four cards from User Story Essentials, there are no barriers to understanding any other essentialized practice from any other source. Just because you like this User Story practice, you aren't now captive in its method prison. Instead you are free to roam the open market to select any other practices from any other sources.
- The practice is described in relation to the Essence standard kernel, thus ensuring it interoperates in well-defined ways with any other essentialized practices.
- This same fact enables the scope and coverage of any practice to be instantly assessed. Our practice adds activities into the Essence kernel activity spaces "Understand the Requirements" and "Test the System," but adds nothing to the other 13 activity spaces outlined by the Essence kernel ("Implement the System," "Deploy the System," etc.). Thus, if this is the only practice we adopt, it is clear that we have no agreed-upon or defined way of doing these other things. This may or may not be a problem, but it is at least a clearly visible fact.
- The practice contains all the essentials. If you are not working according to these essentials in some form,

then you cannot reasonably claim to be doing User Story Essentials as a practice.

WHAT WE DO

The Essence common ground has been recognized by both industry and academia.

Fujitsu UK and Munich Re have been using Essence for many years and contributed to its development. Several of the largest and most prestigious companies in the world are on a path to essentialization—for example, Tata Consulting Services, Red Hat, and a major telecom vendor in East Asia. In collaboration with Jeff Sutherland, co-creator of Scrum, Scrum has been essentialized. Similarly, with Scott Ambler, key practices of DAD (disciplined agile delivery) are being essentialized.

On the academic side we quote professor Pekka Abrahamsson (NUST): "...we have successfully taught Essence in Software Engineering course to 460 students... Essence empowered students to gain control of their project, work methods and practices. We have finally moved beyond Scrum and Kanban... my Software Engineering education in the future will be driven by Essence."

Universities around the world are already teaching Essence to some extent (e.g., Carnegie Mellon University West, Florida Atlantic University, Copenhagen, Oslo, Stockholm, Vienna, Seoul, Beijing, Johannesburg, Medellin, São Paulo, Mexico City, St. Petersburg, Wellington). A project called Software Engineering Essentialized for first-year students started almost three years ago and has resulted in a new book with the same title, to be published soon. The project has drawn the participation of more than

Related articles

- A Conversation with Steve Bourne, Eric Allman, and Bryan Cantrill
  Three *Queue* editorial board members discuss XP and agile, and the practice of software engineering.
  Part 1: https://queue.acm.org/detail.cfm?id=1413258
  Part 2: https://queue.acm.org/detail.cfm?id=1454460
- Breaking the Major Release Habit
  Can agile development make your team more productive?
  Damon Poole
  https://queue.acm.org/detail.cfm?id=1165768
- This is the Foo Field
  The meaning of bits and avoiding upgrade bog downs
  Kode Vicious
  https://queue.acm.org/detail.cfm?id=2566971

50 university teachers worldwide, of whom more than 25 are active.

REFLECTION

Can we truly enable and empower our teams and become true learning organizations while we behave more like the fashion industry rather than an engineering profession? Can we really see ourselves as an open, diverse, and collaborative community while we continually attack one another and rebrand, reinvent, and rename everything like old hipsters trying to stay in with the in crowd? Are we doomed to be locked in a never-ending method war in the hope that the one true way emerges to rule them all?

The answer is, of course, no. By essentializing the most interesting methods in existence today and freeing the practices, an ecosystem of practices will allow us to create the methods we need—also the good ones now in existence—and to upgrade these methods as new or improved practices become available.

We have reason to have high expectations. There is early evidence that teams will be able to learn and come up to speed significantly faster. Projects can measure progress and health of an endeavor independent of which

method they use. People will get a common language to use across product lines. Most important, organizations that adopt Essence are expected to become forever-learning organizations and will move from primarily relying on software development as a craft to being an engineering discipline.[2,3]

As if that were not enough, based on the great interest from systems engineering experts, in particular from several key leaders in the INCOSE (International Council on Systems Engineering) community such as Bud Lawson, there are already proposals on how to modify the Essence kernel so it can also support systems engineering practices. And why not practices for any human endeavor?

References

1. Jacobson, I., Ng, P.-W., McMahon, P. E., Spence, I., Lidman, S. 2012. The Essence of software engineering: the SEMAT kernel. *acmqueue* 10 (10); https://queue.acm.org/detail.cfm?id=2389616.
2. Jacobson, I., Seidewitz, E. 2014. A new software engineering. *acmqueue* 12 (10); https://dl.acm.org/citation.cfm?id=2693160.
3. Jacobson, I., Spence, I., Seidewitz, E. 2016. Industrial-scale agile: from craft to engineering. *acmqueue* 14 (5); https://queue.acm.org/detail.cfm?id=3012428.
4. Object Management Group. 2014. Essence—kernel and language for software engineering methods; http://www.omg.org/spec/Essence/.

Ivar Jacobson *received his Ph.D. in computer science from KTH Royal Institute of Technology, was rewarded the Gustaf*

Dalén medal from Chalmers in 2003, and made an honorary doctor at San Martin de Porres University, Peru, in 2009. He has both an academic and an industrial career. He has written 10 books, published more than 100 papers, and is a frequent keynote speaker at conferences around the world. He is a father of components and component architecture, work that was adopted by Ericsson and resulted in the greatest commercial success story ever in the history of Sweden, and it still is. He is the father of use cases and Objectory, which, after the acquisition of Rational Software in 1995, resulted in the Rational Unified Process, a widely adopted method. He is also one of the three original developers of UML (Unified Modeling Language). But all this is history. Jacobson founded his current company, Ivar Jacobson International, which since 2004 has been focused on using methods and tools in a smart, superlight, and agile way. This work resulted in Jacobson becoming a founder and a leader of a worldwide network, SEMAT, which has the mission to revolutionize software development based on a kernel of software engineering. The kernel has been realized as a formal standard called Essence, which is the key idea described in this article.

Roly Stimson *is a principal consultant with IJI (Ivar Jacobson International) with more than 30 years' experience in applying software methods to complex development challenges. For the past 15 years he has been involved with iterative, incremental, lean, and agile methods. He has contributed to the development of IJI's kernel-based EssUP practices, SEMAT's OMG Essence standard, and IJI's Essence-based Agile Essentials and Agile at Scale practices.*