Escaping Method Prison – On the Road to Real Software Engineering

Ivar Jacobson and Roly Stimson

Background

The world has developed software for more than 50 years. Software has changed virtually every aspect of our lives so we cannot live without it. Thus, the software industry as a whole has been very successful. We could choose to be happy and continue doing what we are doing. However, under the surface it is not as beautiful as it seems: too many failed endeavors, quality in all areas is generally too low, costs are too high, speed is too low, etc. Obviously, we need to have better ways of working or - which is the same - we need better methods.

In this article a method provides guidance for all the things you need to do when developing software. These things are technical, such as work with requirements, work with code and conduct testing, or people related, such as work setting up a well-collaborating team and an efficient project, as well as improving the capability of the people and collecting metrics. The interesting discovery we made in 2003 was that even if the number of methods in the world is huge it seemed that all these methods were just compositions of a much smaller collection of ‘mini-methods’, maybe a few hundred of such

(continued)

This article had been published in 2017 at [www.infoq.com](https://www.infoq.com/articles/escape-method-prison?utm_source=infoq&utm_campaign=user_page&utm_medium=link)(https://www.infoq.com/articles/escape-method-prison?utm_source=infoq&utm_campaign=user_page&utm_medium=link). Copyright © 2017 The Authors. Republished with permission.

I. Jacobson (✉) · R. Stimson
Ivar Jacobson International, London, UK

‘mini-methods’ in total. These distinct ‘mini-methods’ are what people in general call practices.

In this paper the term method also stands for related terms such as process, methodology, method framework, even if these terms strictly speaking have a different meaning.

This is not a new observation. Over all these 50+ years we have been searching for a better method. In some ways our methods of developing software have dramatically changed over time, in other ways they have stayed much the same. As an industry we have followed a zig-zag path moving from paradigm to paradigm and from method to method, changing very much like the fashion industry inspires wardrobe changes. Every new method adoption is generally a very expensive, demoralizing affair. It is expensive because it means retraining the software developers, the teams and their leaders. In some cases existing software may even have to be rewritten in order to work more efficiently with new software. It is demoralizing because the more experienced developers feel they have to relearn what they already know.

Companies, especially larger ones, realize that having a great method provides a competitive advantage – even if it is not the only thing you need to have. They also realize that their method must be explained and explicit so that it can be applied consistently across the organization. And, they realize that one size doesn’t fit for all they do – they need a multitude of methods.

1 Typical Methods and Their Problems

Let’s take a look at four of the most well-known methods (called method frameworks) for scaling agile: The Scaled Agile Framework (SAFe), Scaled Professional Scrum (SPS), Disciplined Agile Delivery (DAD) and Large Scale Scrum (LeSS).

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. Overlapping means that they include some practices that are the same, specific means they have some special practices that makes the difference. If an organization applies one of these methods its users usually don’t know anything about the other alternatives; the reason usually is that the other alternatives are described in different ways with partly different terminology.

Fig. 1 Big pictures of four well-known scaled agile methods

A method can be tacit – in the heads of people - or explicit – described at different levels of detail. A lot of software in the world is developed using tacit methods. Organizations using a tacit method are generally not aware of the problem with method prisons. This is because they have a hard time to change their methods so they stay with what they have for a very long time. Indeed they are caught in their method prison.

What are then the problems?

1. *They are all monolithic – non-modular.*

Most methods (not just the four ones discussed here) are monolithic meaning they are not designed in a modular way. This means that you can't easily exchange one module with another one and keep the other practices intact.

Instead, what we want is a library of reusable modules, which is being updated as users learn more and more. Since every method is just a composition of practices, we want reusable practices. Teams and teams-of-teams should be able to easily agree on their own method by mixing and matching the practices they want to use from the library and compose them together.

2. *They have their own individual presentation style.*

Every method has its individual specific structure, and uses its own style and terminology to describe its selected practices. The owners of the method have decided about these important aspects for themselves without following any

standard. As a result, its practices are incompatible with practices from other methods.

3. *They have a lot in common – but it is hidden.*

Moreover, though every method has some unique practices, it has a lot more in common with others. Every method “borrows” practices from other methods and “improves” them. So, commonalities are hidden behind new terms and “new” features. We use quotation marks to indicate that it is not really exactly “borrowing” that happens, and it is not always “improving”, but due to misunderstanding or reinterpretation of the original practice, it often becomes a perversion or confusion of the original. Likewise the “new” features are typically not completely new at all, but new name for an evolution or variation of a previously existing practices (“new bottles for old wine”).

4. *Every method is controlled by a warden – the guru*

The guru has decided which practices should be combined into his or her method, and in some cases extended the method with practices “borrowed” and “improved” from other methods. The method reflects the particular perspectives, prejudices and experiences of its guru, and not to what we as a development community have collectively learned. Methods should reuse what the team or organization considers the best practices for their specific challenges and purposes, and not those selected by one single guru independent of these considerations.

5. *Every method is branded and often trademarked and copyrighted.*

Other gurus are now, if its users like practices from other methods, forced to “borrow” these practices and “improve” what could have been re-used. This way of working doesn’t stimulate collaboration with other gurus, on the contrary. Given the investment in time and capital by the gurus of these other methods, they must defend their turf with feverish determination, resulting in method wars.

2 Method Prisons

As a consequence, adopting a method – published or homegrown – means that you are stuck with a monolith, presented with its individual style, using many practices that are common but you don’t know it, guarded by a guru who has branded his method making it difficult to reuse. Your method cannot easily reuse practices from a global practice library. Instead, you are in a *method prison*. You are stuck with how the guru of your method has decided how practices are described and how things are done while working with his/her method. To be clear here, we are not suggesting that gurus consciously try to put you in a method prison; they just continue do what we as an industry have done since our origin, because we didn’t know anything better.

Thus, once you have adopted a method, you are in a method prison controlled by the guru of that method. Ivar Jacobson, one of the authors of this paper, was once one of the gurus governing the Unified Process prison. He realized that this was “the

most foolish thing in the world” (of course the software world) and it was unworthy of any industry and in particular of such a huge industry as the software industry. Recently similar ideas have been expressed by others, e.g. see [0].

We as software professionals need to put a stop to this ridiculous development. We want people with creative practice ideas to collaborate and together provide libraries of reusable practices to the world. We want them to serve the whole industry and not be forced to create branded methods.

As a summary, you are most likely in a method prison if your method (framework) is:

- 1) Branded and controlled by a guru (or several gurus).
- 2) Presented in a way that you cannot mix and match practices from it with practices from other methods.
- 3) Described using a homegrown structure, vocabulary, style instead of using an international standard such as Essence.

Many organizations don't realize they are in a method prison. It is easy to understand why not. They have not identified any problems because they haven't seen how it could be different than today. The problems are too abstract without a solution to them. Once upon a time users didn't know that software should be built using components, e.g. java beans. Similarly, they didn't know they needed use cases or user stories to capture requirements. And so on. However, once they got it, and started to use it, they saw the value. Similarly, once they see that they can have access to a global library of practices, which are continuously improved, and from which they can select their own method, they won't go back to what we have today.

It is easy to understand that branded methods put you in a method prison. However, the situation for in-house developed methods is not different, just not so visible. What about agile methods? Most agile methods are today light in description. However, they also suffer from the same problem of not supporting reuse, mixing and matching practices, building a practice library, etc. We also advocate very light descriptions focusing on the essentials, but with the ability to extend with details when desirable.

3 A History of Methods and Method Prison

Since we started to develop software and adopted published methods we have had method prisons. Moreover, method prisons have some side effects that we also need to eliminate, the three most negative ones are the reliance on gurus, the method war

and the zig-zag path. Our history will focus on how methods have created method prisons and their side effects. We will do that from two perspectives: lifecycles and practices.

3.1 *Gurus, Method Wars and Zig-Zag Paths*

Why is the *reliance on a guru* bad?

1. We all understand that relying on a single method/guru is risky. Big companies cannot accept the risk that individuals outside their domain of control should play such a vital role in their way of satisfying their clients. No single method can possibly effectively contemplate the endless variables that arise from the variety of working environments, industries, individual companies and their employees.
2. You effectively ransom your organization's own future competitiveness and ability to adapt, survive and thrive. In the future the method guru decides if and how the method prison is changed over time. And if you don't like it, or it doesn't match your strategic direction of travel and associated needs, there is nothing you can do, because you are stuck inside this method prison, unless you want to suffer the cost and pain of moving to yet another different method prison.

Of course, many people, executives and developers, but not all, realize that the method they adopt is not the whole truth, so they only adopt part of it. Still, for political reasons, they want to tell the world that they use this particular method, to justify the money they have spent in adoption, and if the method is popular it attracts new people to the company. This was true once for RUP, and it is true for the moment with some other currently popular methods. But history tells us that none of these fashions lasts for very long.

There is a *method war* going on out there. It started 50 years ago and it still goes on – jokingly we can call it the Fifty Year's War, which has been even longer than the Thirty Year's War in Europe early 1600 (which was also a “religious war”, incidentally). There are no signs that this will stop by itself.

It is a war because, as the situation has been and still is today, it is very hard to compare methods. We have not had a common ground to work as shared reference. Methods use different terminology, terms that could be synonyms have been adorned by some small differences and these differences are overemphasized, and terms that are nearly homonyms, but not quite, make any comparison very hard to do. Gurus and their followers talk about their method in religious terms, with a lot of passion and zealotry, which makes reasoned comparison and evaluation much harder. Not standing on a standard platform makes it impossible to compare methods and have a rational discussion on methods.

Once upon a time we had a large number of different notations to describe elements in software engineering. Then we got the Unified Modeling Language (UML) standard in 1997 and all these different notations were replaced by one single standard – the notation war was over. Notations are only one aspect of methods, so we need a similar standard for all other aspects of methods, a standard that allow for all the diversity needed from methods.

The software industry has followed a *zig-zag path* from paradigm to paradigm and from method to method.

1. With every major paradigm shift, such as the shift from Structured Methods to Object Methods in the 1980-90's and from the latter to the Agile Methods in the 2000's-now, basically the industry throw out almost all they know about software development and started all over again, with new terminology with little relation to the old one. Old practices are dismissed as garbage, and new practices hyped. To make this transition from the old to the new is extremely costly to the software industry in the form of training, coaching and tooling.
2. With every major new technical trend, for instance service-oriented architecture, big data, cloud computing, internet of things, the method authors also 'reinvent the wheel'. They create new terminology and new practices even if they could have reused what was already in place. The costs are not as huge as in the previous point, since some of the changes are not fundamental across everything we do and thus the impact is limited to, for instance, cloud development, but there is still significant and foolish waste.

Within every such trend there are many competing methods. For instance, back early 1990 there were about 30 competing object-oriented methods. The issue is that all these methods suffer from the five problems resulting in method prisons. This is of course to the advantage of method authors whose method is selected, even if this was not their conscious intention.

We need to eliminate the need for a continued zig-zag path.

3.2 Lifecycles and Method Prisons

From the ad hoc approach used in the early years of computing, came the waterfall methods. There were hundreds of them published. Some of the most popular were Structured Analysis and Design Technique (SADT), Structured Analysis / Structured Design (SA/SD) and Information Engineering (IE). They had their greatness from 1960 to year 2000.

The waterfall methods were heavily influenced by the practices of construction project management – the mantra was “find ways to build software like civil engineers build bridges”. They described a software development project as going through a number of phases such as requirements, design, implementation (coding), and verification (i.e. testing and bug-fixing).

Around the year 2000 they were more and more replaced by iterative methods originally introduced by Barry Boehm's Spiral Model of Software Development and Enhancement, and methods such as RUP and DSDM, and later simplified and further popularized by agile practices such as XP and Scrum. All the four methods introduced earlier, SAFe, SPS, DAD and LeSS, apply an iterative lifecycle.

Of course, all different methods were accompanied by method prisons, and we relied on gurus and perpetuated the method wars.

3.3 *Practices and Method Prisons*

Since the beginning of software development we have struggled with how to do the right things in our projects. Originally, we struggled with programming because writing code was what we obviously had to do. The other things we needed to do were ad hoc. We had no real guidelines for how to do requirements, testing, configuration management and many of these other important things.

We have had three major eras in software engineering (years are just approximate):

- 1960-1980: The Structured Methods Era,
- 1980-2000: The Object Methods Era, and
- 2000 – now: The Agile Methods Era,

resulting in the zig-zag path from era to era. We don't want any more eras and no zig-zag path in future.

The Structured Methods Era

In this era the most popular methods, such as (e.g. SADT, SA/DT, IE), all separated functional process logic from data design. They did this for what were good reasons at the time - because computers at that time were designed exactly like that - with separate program logic and data storage structures. They were used for all kinds of software development – including both “Data Processing” and “Real-Time” systems, following the common parlance of the time. The value of the function/data approach was of course that what was designed was close to the realization – to the machine – you wrote the program separate from the way you designed your data. The systems were hard to develop and even harder to change safely and that became the “Achilles heel” for this generation of methods.

The Object Methods Era

The next paradigm shift came in the early 1980s, inspired by a new programming metaphor - object-oriented programming, triggered by a new programming language Smalltalk. The key ideas behind Smalltalk were much older, being already supported by Simula in 1967. Around 1990, a complement to the idea of objects came to widespread acceptance. Components with well-defined interfaces, which could be connected to build systems, became a new widely accepted architectural style. Components are still the dominating metaphor behind most modern methods.

With objects and components a completely new family of methods evolved. The old methods and their practices were considered to be out of fashion and thrown out. What came in was in many cases similar practices with some significant differences but with new terminology, so it was almost impossible to track back to their ancestors. A new fashion was born. In the early 1990s, as we already have said, about 30 different object-oriented methods were published. They had a lot in common but it was almost impossible to find the commonalities since each method author created his/her own terminology and iconography.

In the second half of 1990s the Object Management Group (OMG - see omg.org) felt that it was time to at least standardize on how to represent drawings about software – notations used to develop software. This led to a task force being created to drive the development of this new standard. The work resulted in the Unified Modeling Language (UML). This basically killed all other methods than the Unified Process (marketed under the name Rational Unified Process (RUP)); the Unified Process dominated the software development world around year 2000. Again a sad step, because many of the other methods had very interesting and valuable practices that could have been made available in addition to some of the Unified Process practices. However, the Unified Process became in fashion and everything else was considered out of fashion and more or less thrown out. Yes, this is how foolish we were.

The Agile Methods Era

The agile movement – often referred to just as “agile” - is now the most popular trend in software development and embraced by the whole world. The Agile movement changed the emphasis away from the technical practices, placing the team, the work and the people front and center.

As strange as it may sound, the methods employed in the previous eras did not pay much attention to the human factors. Everyone understood of course that software was developed by people, but very few books were written about how to get people motivated and empowered in developing great software. The most successful method books were quite silent on the topic. It was basically assumed that one way or the other this was the task of management. With agile many new people practices came into play, for instance self-organizing teams, pair programming, daily standups.

Given the impact agile has had on the empowerment of the programmers, it is easy to understand that agile has become very popular and the latest trend. Moreover, given the positive impact agile has had on our development of software there is no doubt it has deserved to become the latest trend. And, while some agile practices will be replaced by other, better, practices, agile as a philosophy and attitude is not a fad that will pass away. It will stay with us for the foreseeable future.

To summarize

Though the different eras have contributed knowledge and experience, and a lot of it is specific for each era, they all resulted in a continuation of the method war controlled by a few gurus.

4 What to do to Escape Method Prisons

It took us a while to understand what was wrong with how we have dealt with software development methods (see [1] and [2]). However, once we had seen the “most foolish thing in the world”, it didn’t require a genius to figure out that the key to put an end to it was to find a common language with a common terminology or in one word *a common ground*, which we can use when talking about and using practices and methods. Thus in 2009 the SEMAT community was founded with the mission to “re-found software engineering . . .¹ include a kernel of widely agreed elements that would be extensible for specific uses” [3].

We need to find a common ground

Most methods include (or imply) a lifecycle, technical practices and people practices. Thus there is something we have in common. However this is hidden and not easy to discover, because different gurus describe these things using different vocabulary and language. Thus the common ground we are searching for includes a vocabulary and a language. We called the vocabulary the *kernel* and the language the *kernel language*.

Common Ground = Kernel + Language = Essence

Starting with the kernel

Given that the kernel is intended to help describing methods and practices, it needs to contain “things” that are or should be perceived as always prevalent in any method. In essence, what are the things we always have, always do and always produce when developing software²? We, the team of SEMAT volunteers (about 20 people from around the world), working with the kernel, agreed that these things called the *universals* 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 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.”

As an input to the work on finding the kernel in 2010, the three founders of SEMAT (Ivar Jacobson, Bertrand Meyer and Richard Soley) wrote a vision statement [4]. The three of us understood that finding the kernel needed to be guided by criteria and principles. We first agreed on some *criteria for inclusion* of elements in the kernel (see [4] for more complete description of the criteria).

Elements should be: *universal, significant, relevant, defined precisely, actionable, assessable and comprehensive*. *Relevant* was explained as “available for application by all software engineers, regardless of background, and methodological camp (if any)” and *comprehensive* as “applies to the collection of the kernel elements; together, they must capture the essence of software engineering, providing

¹We use the to denote that we have removed some non-important words from the original quote.

²<https://sematblog.wordpress.com/2009/12/07/establishing-a-kernel/>

a map that supports the crucial practices, patterns and methods of software engineering teams”.

We also identified the following general principles deemed as essential to finding a kernel (also in [4]): *Quality, simplicity, theory, realism and scalability, justification, falsifiability, forward-looking perspective, modularity and self-improvement*. *Theory* meant “the kernel shall rest on a solid, rigorous theoretical basis”, *realism and scalability* “the kernel shall be applicable by practical projects, including large projects, and based where possible on proven techniques”, *self-improvement* “the kernel shall be accompanied by mechanisms enabling its own evolution”.

Moreover, the vision statement [4] also formulated what features the kernel should have: *Practice independence, lifecycle independence, language independence, concise, scalable, extensible and formally specified*. *Scalable* was explained as the kernel must support the very smallest of projects – one person developing one system for one customer – it must also support the largest of projects, in which there may be systems-of-systems, teams-of-teams and projects-of-projects. *Extensible* meant 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 criteria, principles and features the SEMAT team set out to find the kernel.

Followed by the language

To explain the universals in the kernel and also practices and methods we need a language. Using just English is not precise enough so we need to have a formal language with syntax and semantics.

The language must be designed for its principal users who are professional software developers participating in a software development endeavor. The language must also allow competent practitioners to create and improve practices without having to learn an advanced language.

The language should support four principal applications: Describing, simulating, applying and assessing. From [4]: “The concept of state is likely to play an important role in the kernel language, to represent work progress.”

The same vision statement gave rather specific requirements on the language. For example “The language should be designed for the developer community (not just process engineers and academics)”, which is an important requirement asking for a more intuitive and more engaging user experience in working with methods than what has been available today. Another example of a requirement is that the language must provide “validation mechanisms, so that it is possible to assess whether a project that claims to apply a given method element ... actually does, and is not just paying lip service to it.”

We need more than a kernel – we need practices and methods

The role of the kernel and the kernel language is to be used to describe practices and methods with a common ground. To get there, a useful common ground had to be applied in describing a large number of methods. We needed to agree on what a practice and a pattern is [4]. We said for example: “A practice is a separate concern of a method. Examples are ... iterative development, component-based

development”, “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”.

With these principles, values and requirements in the baggage the SEMAT team had got a good idea of WHAT was needed to escape the method prison.

5 How to Escape the Method Prison

From idea to tangible result is a long way. We first had to get a common ground.

5.1 *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 method prison started in 2006 in Ivar Jacobson International (IJI). In 2009 the SEMAT 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 [5]. 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 the vision statement [4].

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 we from all this mass of methods had to find the essence so we paraphrased it: “*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.

5.2 *Using Essence*

Instead of giving the whole theory behind Essence, 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 to describe User Story as an example of an Essence practice – calling it here User Story Essentials. Figure 2 below shows (not to be read in detail) the set of 14 cards that represent the headline essentials of the practice.

Fig. 2 The User Story practice as an example of an Essentialized Practice

It is not our intention to here describe the entire practice but to give you a good understanding of what an essentialized practice look like.

An Essentialized practice/method is described using Essence and it 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 libraries of 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 add it to a practice library 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.

Thus, we have selected a representative set of cards being briefly described next.

User Story Essentials (Overview Card) – gives an overview of the practice in terms of:

- A brief description that gives an 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 as to the scope of application of the practice – in this case we see that the practice is:

- 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/or 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 business / customer area concerns such as the Opportunity and the Stakeholders.
- Zero 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 concerns that User Story Essentials addresses and the Endeavor space, which includes concerns such as the Team and how we manage the Work. The practical impact is that this practice can be used with any number of different management practices that mainly operate in the blue Endeavor space, such as a timeboxed, Scrum-style approach to work management or a continuous flow, Kanban-style approach.

Fig. 3 A selection of five cards form the User Story Essentials practice

Customer Team (Pattern Card) – 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 element.
- 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. The Resources card is one of the 14 cards in Figure 2 describing the practice.

Essentialized practices can be described at different levels of detail. The cards in this practice don't attempt to provide all the information for example that 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 to simply reference the original sources of this guidance.

Essentialized practices such as this one work 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 to adopt, adapt and assess their team practices, or for expert teams to use in the same way.

Finally note that, when presented electronically as browsable HTML images, the association and reference links can all be navigated electronically, as can other link elements on other cards.

Find User Stories (Activity Card) – 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 instance the card requires Stakeholder Representative competency at level 2 and Analysis competency at level 1 (all of which is defined in the Essence kernel, and can be immediately drilled into from the electronic browsable HTML and cards)
- An indication of the space that the Activity operates in – i.e. what “kind of thing it helps us do” (the generic kernel “Activity Space” – 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 produced that expresses 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!

User Story (Alpha) – 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 – you can think of Alphas as the things that you expect to see flowing across Kanban boards, described here in terms of:

- 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 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 that the multiple User Stories all relate to (the Requirements in this case).

Story Card (Work Product Card) – gives guidance on the real physical things that we should produce to make the essential information visible – in this case a key defining (though often forgotten) 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 is defined here on the card in terms of:

- A brief description.
- The Levels of Detail that we progressively elaborate – in this case indicating that initially we ensure that we have captured and communicated the associated value, and that we also need to continue on 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.

Putting it all together

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

Now 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 as to how the elements fit together to provide an end-to-end story – which activities progress and produce which elements, but it is also here useful to tell the joined-up story in terms of end-to-end flow through the different activities.

- 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, which involves ensuring that we have the Acceptance Criteria Listed on the Story Card, and during which we may also get any supporting Conversation Captured. As part of this same activity we also fully 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, for example, iterate around the different activities for different User Stories in different ways. Exactly how 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 the following general rules as a team:

- Do the Find User Stories before we start our First Sprint, but also allow this to happen on an ad hoc basis ongoing.

	Find User Stories	Prepare a User Story	Accept a User Story
User Story			
- Identified			
- Ready for Development			
- Done
Story Card			
- Value Expressed			
- Acceptance Criteria Listed			
- Conversation Captured
Test Case			
- Test Ideas Captured			
- Scripted			
- Automated			

Fig. 4 State Progression Matrix showing end-to-end flow through the Activities

- 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 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 is known as *alphas* while the latter is 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.

(continued)

Essence in defining only the generic standard “common ground” defines no work products, activities or patterns, since these are all practice-dependent. It defines 7 alphas with its states, 15 activity spaces and 6 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

Some of the key features and benefits of essentialized practices as illustrated by this one example are:

- The practice is tightly scoped – it tells us how to do one thing well, and does not constrain or limit any of our other choice when it comes to other practices we want to use in other spaces (Scrum, Kanban, etc.).
- The practice is VERY concisely expressed – it’s a little compressed in the above graphic, but when “life-size” the cards in the practice together represent roughly the equivalent of a side of A4.
- The practice is accessible and can be interacted with – the cards are used in all kinds of ways – including making an annotated team 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 – now you understand these 4 cards from User Essentials, there are no barriers to understanding any other Essence practice from any other source - just because you like this User Story practice, you aren’t now captive in its method prison – you are free to roam the open market to select any other practices from any other sources.
- The practice “plugs into” the Essence standard kernel, thus ensuring it interoperates in well-defined ways with any other essentialized practices.
- This same fact enables 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”, . . .) – so if this is the only practice we adopt, it is clear that we have no agreed or defined way of doing these other things (which may or may not be a problem, but *is* a clearly visible fact . . .).
- It contains all the essentials – you may or may not be doing many other things, but if you are not doing this set of things in this kind of way (or locally modified equivalent things, or possibly explicitly NOT doing one particular aspect for a clearly understood and well-articulated reason) then can you reasonably claim to be doing “User Stories” at all?

5.3 Reflection

In section 5.2 we presented the User Story practice essentialized without first presenting the Essence kernel and language. We presented the practice with “Essence in Stealth Mode”, to coin an expression we have got from Paul McMahon. However, underneath the essentialized practice we rely heavily on Essence. In our example User Story is a sub-alpha related to the *Requirements* kernel alpha. The “Find User Stories” activity is allocated to the “Understand the Requirements” activity space and so is the activity “Prepare a User Story”, while the “Accept a User Story” belongs to the activity space “Test the System”.

We have attempted to show that practices are easily understood even without first giving a long and, too many people, boring introduction to Essence. This has been done in many other papers and books already, see [6] - [10]. Thus, here we will just mention some important things you may need to take away.

When the SEMAT volunteers designed Essence as a response to HOW to escape the method prison, particular attention was paid to the “simplicity clause” that “the kernel shall only include essential concepts”, which the team interpreted as the guidelines for a method or practice should focus on the essentials.

- The experience is that developers rarely have the time or interest to read detailed methods or practices. Starting to learn the essentials gets teams ready to start working significantly earlier than if they first have to learn “all” there is to say about the subject.
- The essentials were defined as a rule of thumb being about 5% of what an expert knows about the subject.
- Some teams and organizations need more than the essentials, so different levels of detail must be made optional.

The SEMAT team also knew we had to come up with a new user experience to teach practices. The current way of doing it through books and web sites didn’t help during actual work – books are dead descriptions and not active guides. We searched for a more engaging way of working and found inspiration in modern work on gamification. We used cards, as you have seen.

We also consistently applied the principle of ‘Separation of Concerns’ in many different contexts (for general discussion see http://en.wikipedia.org/wiki/Separation_of_concerns). Practices are separate concerns, which can be composed into methods through a merge operation, known in Essence as “composition”. The kernel is also a separate, more abstract, concern, on top of which practices can be composed, also merged.

In summary, Essence enables us to escape from method prisons because it sets out a description of what all methods have in common, and a standard language for talking about this common ground and about all our practices. This means we are free to select essentialized practices from any source we choose, including from our own organization as well as external sources, and free to mix-and-match them with

practices from other sources, in order to get the best from all worlds, without being locked in to any of them.

6 Out of the Method Prison

Many companies are now in the process of essentializing their existing methods. For instance, in the words of Tata Consulting Services (TCS): “TCS has engaged with all of its core industry partners like SAP, Oracle, Microsoft and others and also the clients of TCS and is working with the core methodology teams of those companies to help foster the collaborative adoption of the Essence standard and turn this de-jure standard into a de facto standard.”

These companies get reusable practices available in a practice library. Teams and organizations are able to mix and match practices from different methods and create their own ways of working. Today, we believe that there are around hundred practices described on top of Essence. Ivar Jacobson International has developed about 50 practices and made 25 of them available in a practice library at <https://practicelibrary.ivarjacobson.com>.

Those companies are getting out of their method prisons. They don't rely on gurus anymore. They won't follow a zig-zag path, but they expect a sustainable evolution. The method war is over for them. However, getting out of method prisons is not all they are expecting. They have much higher ambitions. They are on a path to industrial-scale agile – moving software development from primarily being a craft to primarily being an engineering discipline, but still being agile in both software development and in working with methods.

To be successful we still will rely on the craftsmanship of our empowered teams, but this will be underpinned with a shared base of codified engineering practices that can be reused in different permutations and combinations across different technical domains and project types. This will enable us to maintain high levels of craftsmanship consistently across all our projects, and to sustain this indefinitely through future challenges and changes.

We also need a supporting organization with a learning culture open to new ideas and comfortable with experimentation. Discussing this is out of scope for this paper, but we refer to papers already published (see [8]-[10]).

Essence is also making inroads in the academic world. Universities around the world are teaching Essence to a varying degree. Here a quote from Professor Pekka Abrahamsson, “At one of the largest technical universities in Scandinavia, Norwegian University of Science and Technology in Trondheim, in the Spring of 2017, 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. ... Data and results convinced me and thus my Software Engineering education in the future will be driven by Essence.”

Maybe this move to Essence is “the smartest thing in the world” to these companies and universities.

References

0. Hastle S., Linders B., McIntosh S., Ferreira R.M., Smith C. “Opinion: What 2017 Has in Store for Culture & Methods”, <https://www.infoq.com/news/2017/01/2017--culture-methods>, InfoQ, 2017.
1. Jacobson I., Meyer B. Methods need theory. Dr. Dobb’s Journal, 2009.
2. Jacobson I, Spence I. 2009. Why we need a theory for software engineering. Dr. Dobb’s Journal, 2009.
3. Jacobson I., Meyer B., Soley R. 2009. Call for Action: The Semat Initiative. Dr. Dobb’s Journal, 2009.
4. Ivar Jacobson, Bertrand Meyer, Richard Soley. “Software Engineering Method and Theory – a Vision Statement”, Feb 2010.
5. Object Management Group, “Essence - Kernel And Language For Software Engineering Methods”, November 2014. (<http://www.omg.org/spec/Essence/>).
6. Ivar Jacobson, Pan-Wei Ng, Paul E. McMahon, Ian Spence and Svante Lidman, “The Essence of Software Engineering: The SEMAT Kernel,” Communications of the ACM, Volume 55, Issue 12, December 2012.
7. Ivar Jacobson, Pan-Wei Ng, Paul E. McMahon, Ian Spence and Svante Lidman, “The Essence of Software Engineering: Applying the SEMAT Kernel”, Addison-Wesley, 2013
8. Ivar Jacobson, Ian Spence and Pan-Wei Ng. “Agile and SEMAT: Perfect Partners”, Communications of the ACM, Volume 11, Issue 9, Oct. 2013
9. Ivar Jacobson and Ed Seidewitz, “A New Software Engineering,” Communications of the ACM, Volume 57, Issue 12, Pages 49-54. December 2014.
10. Ivar Jacobson, Ian Spence, Ed Seidewitz. “[Industrial-scale agile: from craft to engineering](#)”, Communications of the ACM: Volume 59 Issue 12, December 2016.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.