

Industrial-Scale Agile *from* CRAFT *to* ENGINEERING

IVAR JACOBSON, IAN SPENCE, AND ED SEIDEWITZ

ESSENCE IS
INSTRUMENTAL
IN MOVING
SOFTWARE
DEVELOPMENT
TOWARD A TRUE
ENGINEERING
DISCIPLINE

In his paper titled “Industrial Scale Agile,”⁹ Roly Stimson characterizes industrial-scale agile as:

- “Agile at any scale.”
- “Agile as the rule, not the exception.”
- “Agile sustainably, forever,” not just as an unrepeatable “one-off.”

This means being able to sustainably apply agile strategies appropriately to anything and everything that can benefit from them. This includes:

- Being able to do “agile at scale” as and when appropriate.
- Doing small-scale agile as and when possible/ appropriate.
- Evolving the entire application landscape and not just individual applications.

Although it is important, and a necessary precursor to industrial-scale agile, scaling agile is not the challenge here. Rather, it’s about how to achieve sustainability of the following:

- The way of working in the face of ever-changing teams.

- The systems in the face of rapid change.
- The application landscape as a whole.
- Individuals and their careers, and the development organization as a whole.
- Long-term investment in IT.

There are many, many ways to illustrate how fragile IT investments can be. You just have to look at the way that, even after huge investments in education and coaching, many organizations are struggling to broaden their agile adoption to the whole of their organization—or at the way other organizations are struggling to maintain the momentum of their agile adoptions as their teams change and their systems mature.

Another frequent example of unsustainability is in the way that many companies are facing an uncontrolled explosion in the number of applications they have to support and the overall cost of ownership of IT as a whole.

So industrial-scale agile requires much more than just being able to scale agile. It also means taking a disciplined approach to ensuring that IT investments result in sustainable benefits for both the producing organization and its customers.

This involves adopting a different approach to many aspects of agility. We need to look beyond small-scale agile, beyond independent competitive islands of agile excellence, beyond individual craftsmanship and heroic teams, and beyond the short-term instant gratification that seems to be the focus of many well-intentioned but self-centered agile teams. It is this adoption of a more holistic approach that we call moving from craft to engineering. [We have tried to keep this article short,

but for readers new to this space, we recommend “A New Software Engineering”³ for more background.]

FROM CRAFT TO ENGINEERING

The move toward agility has led to many benefits for the software industry. It has broken the tyranny of the prescriptive waterfall approach to software engineering, an approach that was causing more and more large project failures, and it has allowed software developers to keep up with the ever-increasing demand for more innovative IT solutions.

It has enabled many companies to do great things but in many cases has led to a culture of entitlement, heroic programming, and short-term thinking that threatens the sustainability of the parent companies and the IT solutions on which they depend. Little or no thought is put into maintainability, the heroes become potential single points of failure, and the cost of keeping the lights on just keeps growing and growing.

What is needed is a way to maintain the values of agility while making software development more an engineering discipline than a craft—a new form of agile software engineering fit for the Internet Age.

What are craft and engineering?

The term *craft* is usually applied to people occupied in small-scale production of bespoke goods and trades where skills are passed in person from master to apprentice. *Engineering*, on the other hand, is defined by Wikipedia (<https://en.wikipedia.org/wiki/Engineering>) as “the application of mathematics, empirical evidence and

scientific, economic, social, and practical knowledge in order to invent, innovate, design, build, maintain, research, and improve structures, machines, tools, systems, components, materials, and processes.”

There have been many discussions about whether or not the term *engineering* should be applied to software development and whether or not software engineers are actually engineers. With the rise of cloud computing, big data, and the Internet of things, however, it is clear that there are many types of software and many aspects of software development that would benefit from an engineering approach.

In her seminal paper, “Prospects for an Engineering Discipline of Software,”⁷ in 1990, Mary Shaw suggested that a definition of software engineering would include these clauses: “Creating cost-effective solutions...to practical problems...by applying scientific knowledge...building things...in the service of mankind.” She also said about software work that “most tasks are routine and not innovative,” but it “is treated more often as original than routine,” implying that there is a lot of potential for improving quality and shortening time to market “if we captured and organized what we already know” by codifying our knowledge, possibly even automating it.

Her observations are still highly relevant; at the GoTo Amsterdam 2015 conference on software development, she talked about the progress that has been made toward establishing a software engineering discipline. According to Shaw, the characteristics of engineering are as follows:

- Limited time, knowledge, and resources force decisions on tradeoffs.

- The best-codified knowledge, preferentially science, shapes design decisions.
 - Reference materials make knowledge and experience available.
 - Analysis of design predicts properties of implementation.
- Although software development shares many of the characteristics of an engineering discipline, we are not there yet. The rise of agile is not a problem unless this is where we stop.

Why more engineering?

Why is it important to move from craft to engineering?

Doing so will help us cope with the ever-increasing challenges of a more automated, more interconnected world—where small improvements in software performance can make the difference between profit and loss; where a reputation for robustness, scalability, and security can add millions to the share price; and where software is more and more the public face of the business.

The codified knowledge and professionalism of an engineering discipline are necessary for:

- Sustaining and growing delivery capability through changes in technologies, teams, and suppliers.
- Predictably scaling operations from early prototypes to global rollouts.
- Taking control of investments and knowing when to pivot to solutions more likely to deliver favorable returns.
- Systematically growing the levels of reuse and interoperability of solution components and systems.
- Producing long-lived solutions with affordable costs of ownership.

Perhaps not everybody needs to move from craft to engineering. As Mary Shaw says, “The greatest need for engineering discipline exists for software systems that are fully automated and are operating unattended and where the consequences of failure are catastrophic. Examples are telecom equipment, nuclear safety devices, medical implants, self-driving cars, and stock-trading programs. The need for engineering in software development depends upon how serious the consequences are when things go wrong and whether human beings can take action in time to minimize the consequences.” There is also a strong need for engineering systems used for e-commerce, finance, electronic medical records, and even human resources. The consequences of failures in such systems may not include the immediate loss of life, but they can still be “catastrophic” to either the businesses or the individuals affected.

Thus, for many organizations and software systems craft is not enough.

The good news is that there is a way forward that maintains the values of agile while making software development more of an engineering discipline than a craft. It involves:

Engineering of software. This means the holistic engineering of all software to improve the application landscape as a whole, as well as the individual point solutions. Practices are needed that help teams engineer their software for capturing requirements and for developing software designed for engineering great products. It also means encouraging innovation in the large as well as the small—innovation of new business and new product

opportunities as well as innovation that addresses the total cost of ownership impacting the whole organization, rather than just individual users and applications.

Engineering of methods. Methods should be engineered to support the full range of development challenges faced today and in the future. The emerging best practice should be captured and codified in a way that makes it easy to communicate and share among teams, and enables each team to compose the method they need from this growing set of reusable, proven practices.

Furthermore, moving from craft to engineering provides a robust platform for encouraging, establishing, and sustaining true organizational agility.

Engineering of software

How would software be developed if the craft were already a real engineering discipline? As in other engineering disciplines, it would be engineered by using commonly accepted, consistent practices that would be supported by models and analysis based on a common ground of foundational knowledge.

In the past, such an engineering mindset has been misinterpreted as meaning “big upfront design,” with everything downstream of this being akin to manufacturing rather than engineering. Upfront blueprinting is, indeed, often necessary for the engineering of physical artifacts such as buildings, bridges, and cars. This is done so that proper analysis can be carried out on the upfront models and blueprints, because of the capital cost required to build those things and the difficulty of changing them once built. Software, however, is a different kind of artifact—one

that does not require manufacturing in the physical sense. Agility in software development takes advantage of

Craftsmanship and Engineering



Related to the idea of *craft* is *craftsmanship*, performed by a person who practices or is highly skilled in a craft. Software development will always need craftsmanship that can stand on more or less science, more or less engineering, and more or less structured knowledge. We would, for example, describe an engineer as a craftsman using engineering practices in developing software. The new software craftsmanship movement is supportive of many engineering practices—for example, they are strong supporters of design and architecture patterns, domain-driven design, etc. They take pride in using the right tools, techniques, and design methods to achieve high-quality software. They do not believe in heroics, but in quality of work and in tools. They believe in sustainability, and in keeping the system “clean” and able to absorb change and rework. The craftsmanship movement, however, doesn’t fully address the whole engineering space and, in particular, how to systematically grow knowledge about the discipline.

this characteristic, allowing software to be developed in a rapid and incremental, but still reliable, way; however, there is a place for disciplined design within an agile development approach. It is just that, with software, developers can also carry out analysis and evolve designs incrementally, as they build the software system itself.

What is needed is, in fact, a merger of the agile mindset with the engineering mindset, combining incremental development with the disciplined application of foundational knowledge. In such an approach, not everyone will necessarily be an engineer, but developers will continue to be treated as skilled craftsmen, not factory workers. (See sidebar, “Craftsmanship and Engineering.”)

It is common in agile approaches to talk of the *emergence* of the design of a software system as that system is iteratively developed. This is the very embodiment of

evolutionary design as opposed to big upfront design. It can be very effective in allowing a team to explore alternatives creatively, while still converging on a good solution with a clear overall design.

Such emergent design, however, tends to produce point solutions for specific teams. Serious software development organizations, though, are almost always dealing with multiple teams working on multiple projects within an overall enterprise-level application landscape. Various project-level solutions need to fit into this evolving landscape. Indeed, the development of a large software system often requires multiple teams whose products are components that must fit together to create the overall system.

Dealing with design at this level is the province of software architecture, which, at both the system and enterprise levels, can and should still be evolutionary. Rather than being entirely emergent, however, key architectural decisions, presented in a development roadmap, often need to be made in advance of the corresponding development work in order to provide common guidance across projects and teams. This is where engineering practices can be particularly important, allowing for innovations that benefit the organization as a whole, based on careful analysis of business benefit versus engineering cost.

Engineering of methods

Moving from craft to engineering relies on the codification and sharing of knowledge. What is needed is for organizations to *engineer their methods* in order to be

more effective at engineering their software.

Most methods in use today are at the extremes, either monolithic or tacit. The agile space is experiencing the rise of a number of competing, monolithic scaled agile methods, such as DAD (disciplined agile delivery), SAFe (Scaled Agile Framework), LeSS (Large-scale Scrum), and SPS (Scaled Professional Scrum). All these methods have their special strengths and weaknesses. They have their own camps of supporters, but their monolithic nature doesn't make it practical to borrow ideas from one another, even less to borrow complete codified practices. This situation is very similar to what we had in the past with methods such as RUP (Rational Unified Process), Open, Structured Analysis and Design, etc. We have also observed that, at many large organizations, the success of tacitly applied agile practices has led to a situation where the previously used and codified (documented) methods have been replaced by undocumented agile folklore.

The paradox here is that the discomfort caused by not having a documented method causes many organizations to seek to replace their tacit agile methods with one of the new monolithic methods. What they don't realize is that it will end up being rejected in the same way as the original method as teams seek to innovate and meet the day-to-day challenges inherent in their systems and circumstances. This often leads to a constant churn as method replaces method with little or no rhyme or reason. The industry's habit of constantly switching between no methods and the latest "one true way" (an affliction that is sadly affecting even the agile community) is not the way forward.

Instead, organizations need an effective way of using

what they learn from effort to effort, applying and adapting it to new projects. Moving blindly from one fad method to another provides no consistent basis for building common knowledge. Mandating a one-size-fits-all process for all projects does not support the need for continual learning and adaptation, however, and suppresses craftsmanship and creativity.

The move from craft to engineering requires first freeing the practices, presenting them in an accessible, reusable way that allows engineers confidently and predictably to select the right engineering practices for their context and the problems they are trying to solve.

ESSENCE: A HOPE FOR A BETTER FUTURE

Software development is a multidimensional endeavor—where human ingenuity meets human need meets collective endeavor meets codified knowledge—that would benefit from the judicious application of engineering practices. The path from craft to engineering progresses—from ad hoc practice to codified professional engineering practices—through scientific learning.

The key to this transformation is the ability to readily capture, share and improve the practices.

What is Essence?

Essence is a simple intuitive language and kernel of foundational elements for the capture, description, and assembly of practices and methods. Work on Essence has been going on for more than 10 years—for the last six years within the SEMAT (Software Engineering Method and Theory) community—resulting in a new international

standard, adopted by OMG (Object Management Group) in 2014.⁵ It goes beyond just providing syntax and notation for describing practices to establishing a solid common ground—a kernel—that enables teams to:

- Describe their practices on top of a universal, shared kernel.
- Easily share, adapt, and plug and play with their practices to create the innovative ways of working they need to excel and continuously improve.
- Understand and visualize the progress and health of their endeavors, regardless of their way of working. (For more information on Essence, additional resources are listed at the end of this article.)

Essence has several roles to play in the move from craft to engineering:

- Helping to achieve the right balance in software engineering endeavors.
- Helping to codify and capture engineering practices.
- Acting as the basis for a new kind of engineering community.

The use of Essence alone won't turn craftspeople into engineers, but its adoption will help an organization make this important transition and, moreover, help the industry prepare for the future.

Balancing progress and health

Essence provides a kernel of elements that establishes a common ground for carrying out software engineering endeavors. This can be used in a number of ways to increase the effectiveness of software engineering teams, including the following.

Actively monitoring the health of an endeavor.

The kernel defines seven aspects of concern for any software engineering endeavor: opportunity, stakeholders, requirements, team, work, way of working, and the software system itself. For each of these elements Essence defines a series of states, with checklists, representing healthy progress. As shown in figure 1, these can be used to create practice-independent health monitors that can be used to check that the endeavor is on course and proceeding in a healthy manner. On the top, the radar chart (an interactive, online version complete with checklists is available¹) represents progress as growth from the center; and on the bottom, on the milestone map (available from the App Store as the Alpha State Explorer app by Ivar Jacobson International), all the states are laid out in order from top to bottom, with achieved states shown filled in. The second example also shows the checklist used to confirm the achievement of the Software System Demonstrable state.

The kernel can also be used to create lightweight governance and compliance practices to help ensure the team achieves the required level of engineering rigor. By basing the governance and compliance on the kernel itself, this can be achieved in a practice-independent fashion, allowing the teams to safely innovate and own their own ways of working. Figure 2 shows the four different governance life cycles that were at the heart of Munich Re Essentials, the modern practice-based software development method created by Munich Reinsurance.⁴ The four life cycles are Exploratory, Feature Growth, Maintenance, and Support, and the checkpoints



FIGURE 1: SIMPLE ESSENCE-BASED HEALTH MONITORS

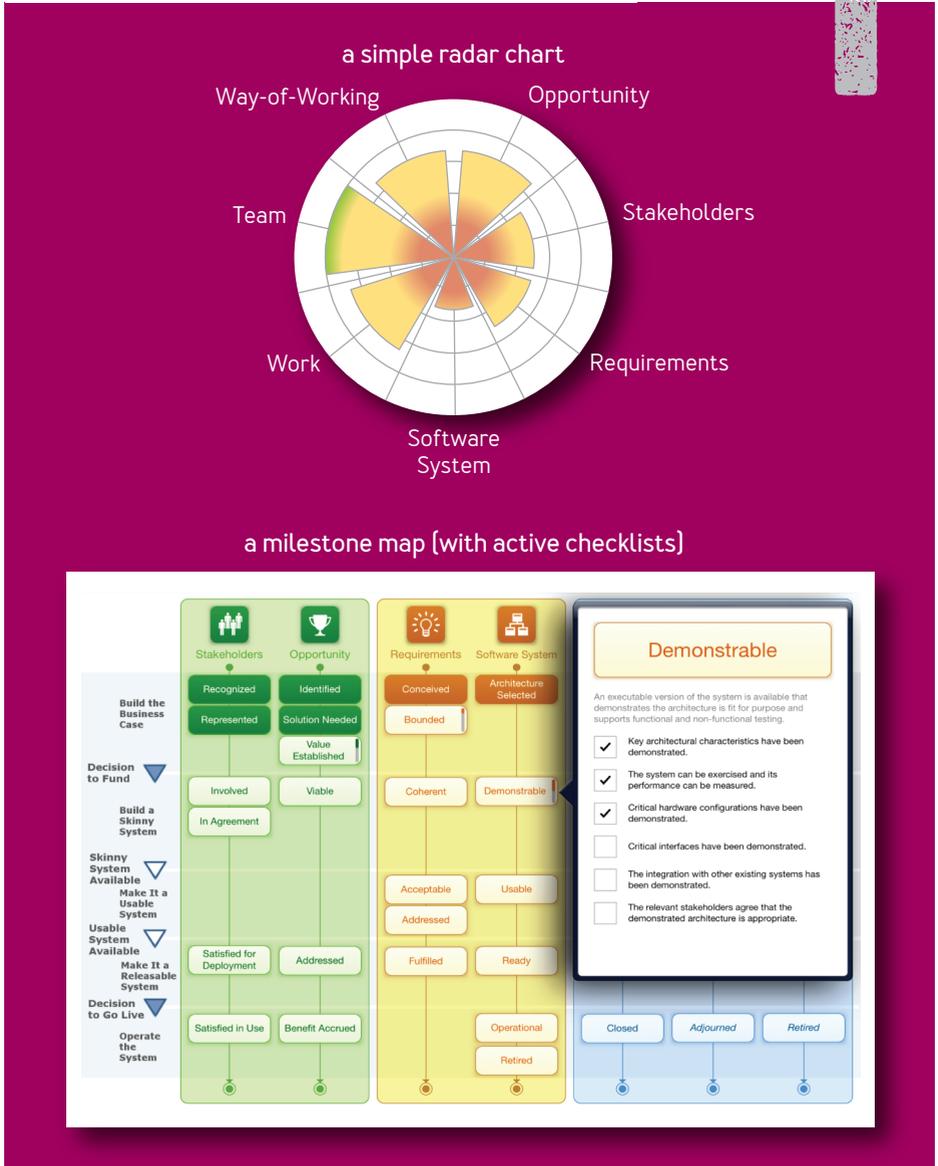
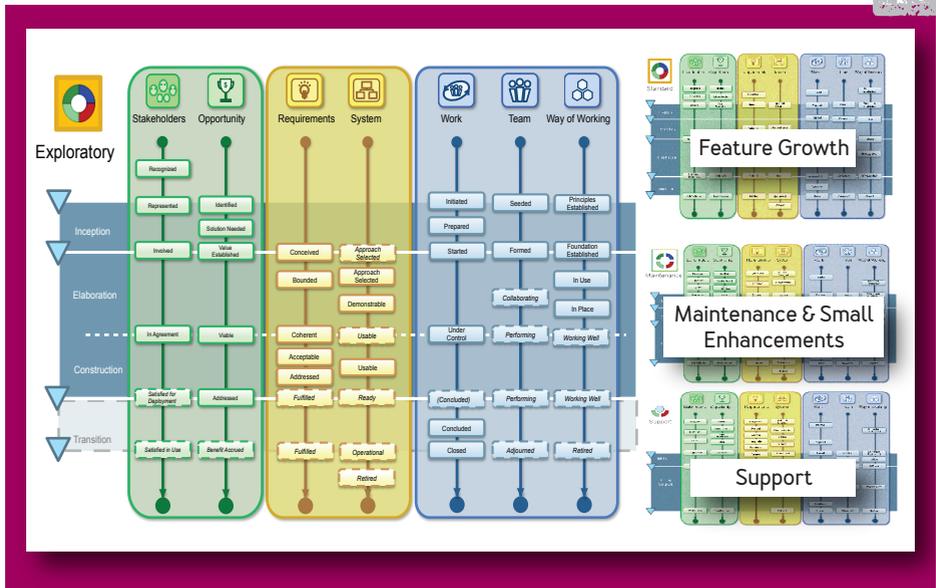




FIGURE 2: **FOUR COMPATIBLE GOVERNANCE LIFECYCLES DEFINED USING THE ESSENCE KERNEL**



[milestones] of each life cycle are defined by the states to be achieved for each alpha.

Assessing the effectiveness of methods.

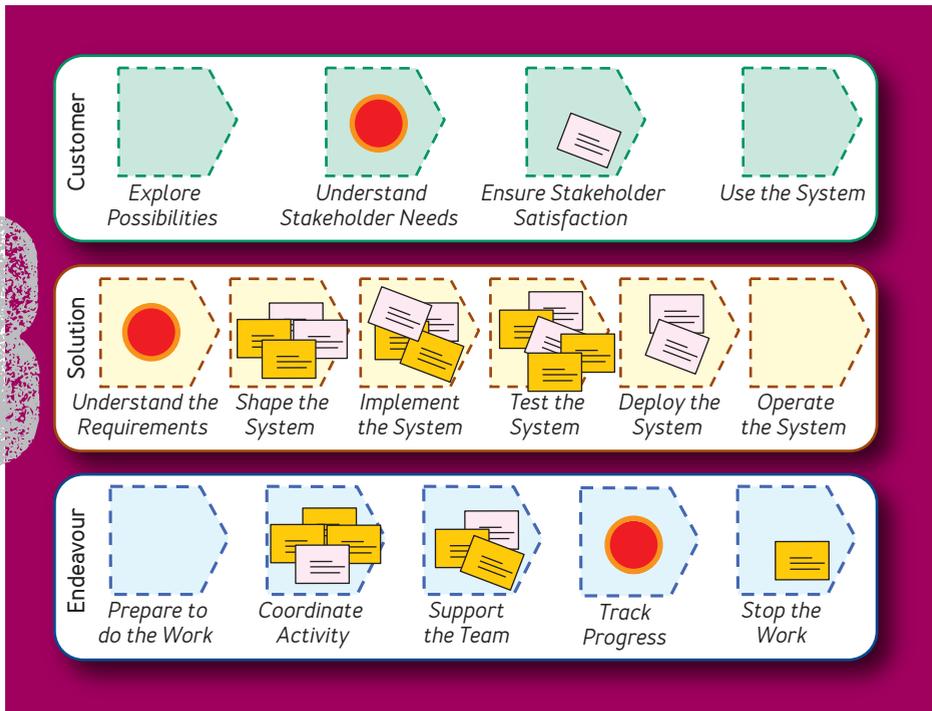
Essence at its roots gives a detailed definition of software engineering. In the search for a GTSE [general theory of software engineering],⁶ several researchers use Essence as such a definition, and more is expected to come out of this work. A key aspect provided by such a theory is the capability to be *predictive*.

A construction engineer can use material science and the theory of structures to understand at an early stage whether a proposed building is likely to stand or fall. Similarly, using Essence, one can understand whether

a proposed method is well constructed, whether or not there are any gaps or overlaps in its practices, and if there are gaps or overlaps, how to resolve them.

The kernel has many mechanisms for method analysis, the simplest of which is provided by its high-level activity map. This is a set of 12 *activity spaces* organized into three areas of concern. An activity space is a generic placeholder for method-specific activities. These activity spaces, as shown in figure 3, can be used to assess the spread of the team's activities. In this example, the team has added notes

FIGURE 3: SIMPLE ESSENCE-BASED ACTIVITY MAP



to the map to indicate their activities and red circular markers to highlight the danger areas.

Note: Without understanding the meaning of the Essence language, the symbol of a pointed arrow to represent an activity space can make them appear sequential, which is not the intended meaning. The activity spaces and the activities that they contain can of course be applied iteratively, concurrently, or in any order the practices require.

These are just a few simple examples of the kernel's capabilities, but they illustrate the many ways it can help teams and organizations assess the effectiveness of their methods.

Codifying and capturing engineering practices

In addition to the kernel, Essence provides a language for creating practices on top of the kernel and then composing methods from those practices. This is extremely important for moving from craft to engineering.

As discussed previously, most current practices are embedded in monolithic methods that aggressively compete with one another. Rather than admit that they share practices and encourage reuse and cross-pollination, they willfully slander and steal from one another. Even worse, from an engineering perspective, they are all concerned with the design and sustainability of the development organization rather than the design and sustainability of the systems produced. This is not to say that the former isn't important. Indeed, it is crucial to the success of any development organization. As the integration of software into the fabric of our daily lives

grows, however, the need for proven, reusable engineering practices grows as well.

Practices are needed that help teams engineer their software: practices for working with requirements, such as use cases, features, and stories; for developing components and services; for applying an appropriate pattern or framework; for testing complex, distributed systems; that encourage reuse; and that help engineers code with confidence. In particular, practices are needed for dealing with architectural concerns such as concurrency, security, user experience, microservices, and data protection, as well as for addressing broader architectural concerns such as enterprise architecture, product-line architecture, service-oriented architecture, and the architecture of systems of systems. Many of these practices already exist codified in the Essence language [see the section on Sharing Practice: Methods and Practice Libraries].

These engineering practices need to be streamlined (lean), agile, and, most importantly, composable into complete methods to provide guidelines for teams working with a multitude of practices for complex systems. They are needed to help deal with the complexities of modern software engineering. They need to be available to all engineers whether they are working alone, in small teams, or in larger teams of teams, regardless of the style of team working or work-management practices adopted.

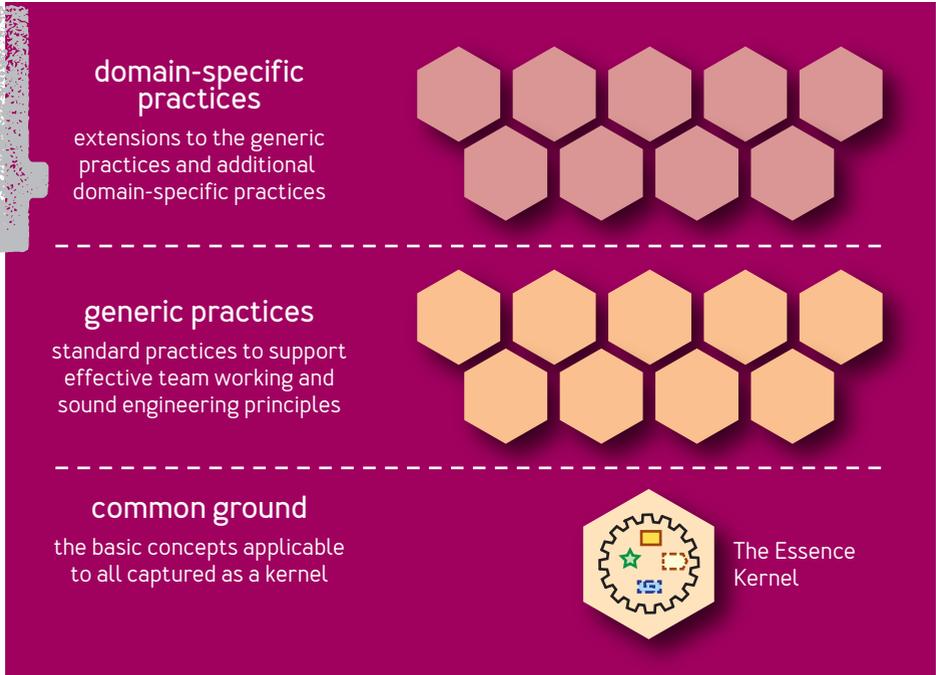
Globally, we want a robust and flexible library of codified professional engineering practices that reflect the multidimensional nature of software development and that can be used to support the many different types of software being developed today and in the future.

These practices can only come from engineering teams working on the cutting edge of technology, and these teams need a better way to capture, communicate, and share their practices.

A new method architecture

With the Essence kernel as common ground, you can use the Essence language to describe any practices, including engineering practices, in a way that allows them to be composed seamlessly together to form methods. Figure 4 illustrates a three-layer method architecture with the

FIGURE 4: THE ESSENCE METHOD ARCHITECTURE



kernel as the foundation, generic practices in the middle, and domain-specific practices at the top.

Starting from the bottom of the stack, the three layers are:

The Essence kernel. This provides the common ground for all practices and methods and the underlying foundation for the definition and composition of the practices.

Generic practices. These are practices that are applicable across many software engineering domains. Examples of generic practices include Scrum, use cases, user stories, test-driven development, and acceptance-test-driven development. Many engineering practices will be generic, but many of the most valuable will be domain-specific.

Domain-specific practices. These practices are explicitly targeted to a specific domain such as business intelligence, data warehousing, or telecommunications. Domain-specific practices are equally as important as generic practices, if not more so. For example, many domain-specific practices are needed to develop solutions for the Internet of things; these practices cater to things such as asset integration architecture and different technology profiles. Just as generic practices extend the kernel to provide specific guidance, domain-specific practices are often extensions/specializations of the generic practices. For example, an asset integration architecture practice could be presented as an extension to a generic agile architecture practice.

The separation of generic practices from domain-specific practices helps teams find the practices that they need and helps organizations establish common ways of organizing and tracking their work. It is not uncommon for an organization to standardize on a small set of generic

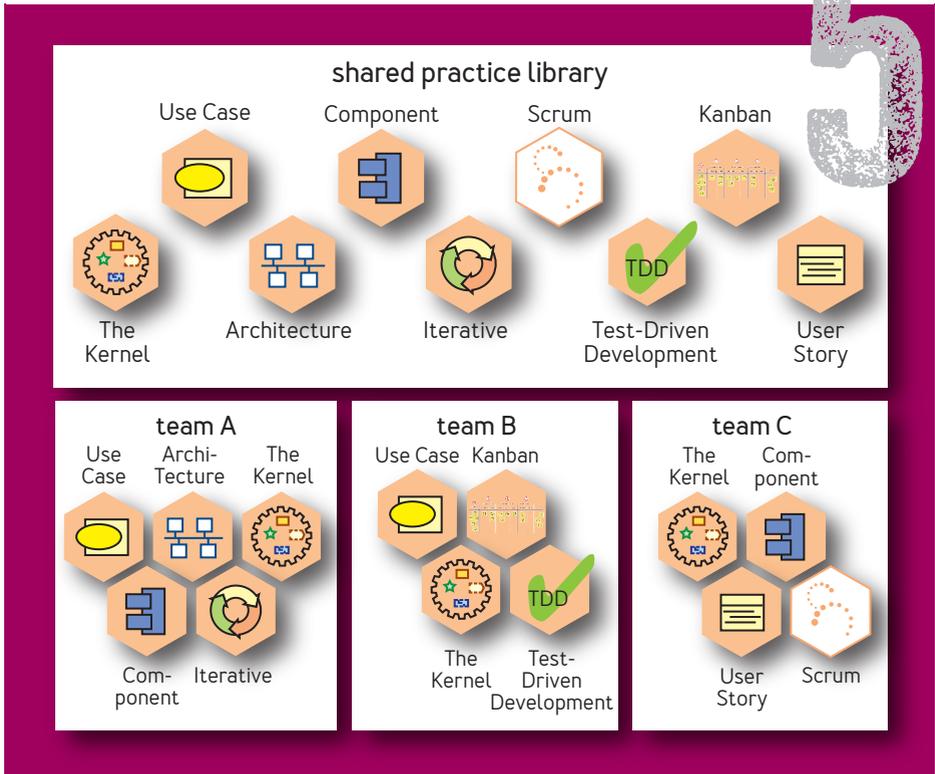
practices as the foundation for all of its teams' methods.

Liberating practices in this way is very powerful. Once practices are codified in Essence, teams can take ownership of their ways of working and start to assemble their own methods. This can start with even a simple library of practices, as shown in figure 5.

This capturing and sharing of engineering practices, both generic and domain-specific, in a way that lets them

FIGURE 5: **THREE TEAMS SHARING A SIMPLE PRACTICE LIBRARY**

5



be applied alongside popular management practices (agile or otherwise), provides the codified knowledge needed to support a true software engineering discipline. It is also the key to moving away from monolithic management methods and isolated engineering practices.

Sharing practice: methods and practice libraries

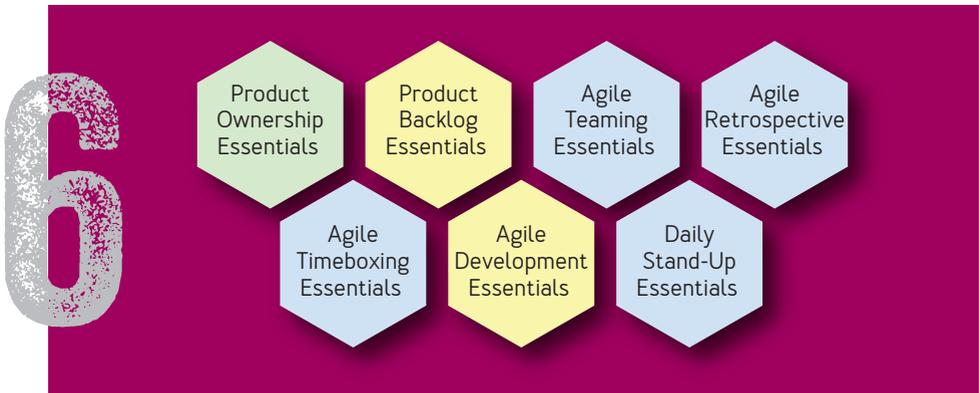
It's easy to say that teams will be able to plug and play with sets of practices to build their own methods and take ownership of their way of working. But where are the practices going to come from?

Let's take a look at two concrete examples.

Agile methods

The industry has seen an explosion in the number of generic agile practices being published and promoted. Unfortunately, most of these “belong” to one method or another and, even though they share the same values, are

FIGURE 6: **AGILE ESSENTIALS WITH ITS SEVEN PRACTICES**



rarely presented in a way that lets them play well together. This is particularly true in the area of scaled agile methods, where each method contains many of the same practices tangled up with a few new, unique, and innovative practices in such a way that the safe separation of the new practices for use with another method is nearly impossible.

In contrast, figure 6 outlines a starter pack of agile practices based on Essence, called Agile Essentials.² It includes practices from Scrum, Kanban, and XP (extreme programming).

This is a small library of seven practices, which, when composed together, form a starting point for a team's agile method. Scrum, user stories, and use cases have also been "essentialized" and can be used alongside the Agile Essential practices.

Thus, with Essence, a library of generic, reusable practices can be created, from which a team can select the ones they want to use and that they can compose together to kick-start their own method.

The Ignite Internet of Things methodology

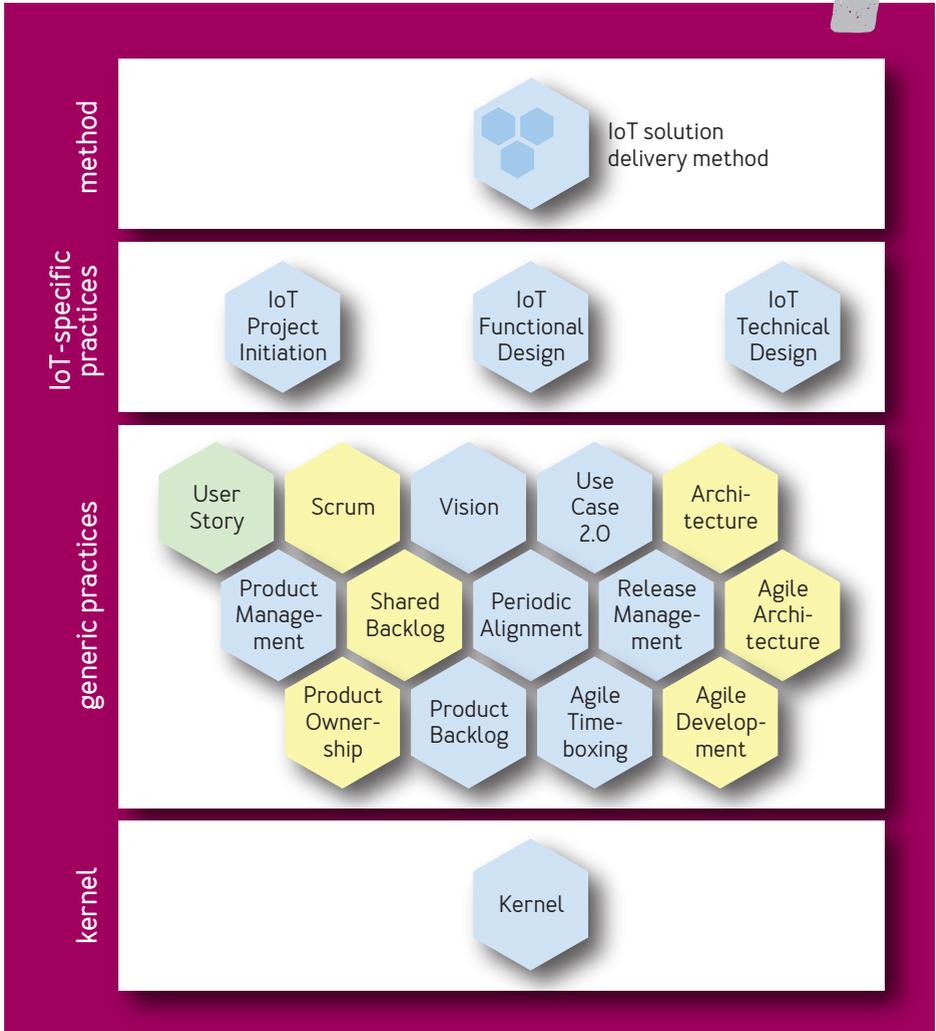
Ignite is a methodology developed for the Internet of things.⁸ It supports a number of different approaches and attempts to bridge the gaps between "machine guys" and "Internet guys," and between "five-year thinking" and "continuous beta." Ignite can easily be described as a set of practices on top of the Essence kernel. Figure 7 demonstrates what Ignite looks like when presented using Essence.

This picture readily illustrates a number of key points:

- Ignite clearly contains and reuses a number of generic

7

FIGURE 7: **IGNITE EXPRESSED AS A SET OF ESSENCE PRACTICES**



practices that are applicable in many more domains than the Internet of Things, including those already available

as part of the Agile Essentials practice library.

- Successful development for the Internet of Things requires many domain-specific engineering practices.
- Whenever anyone wants to create a new method, they currently have to rewrite, re-present, and, in many cases, rebrand already established generic practices.
- The more comprehensive the approach, the less likely it is that anyone will use all of it. For example, no one is ever going to use all of these practices at the same time. Clearly, there are many methods that could be built from the practices contained within the Ignite methodology, but without the use of Essence this will be very difficult, if not impossible, for teams to do.

Many other practices could be useful for teams developing for the Internet of Things. Some of these will be innovations unknown at the time of writing this paper or the creation of Ignite. This can easily lead to the approach becoming out of date and unfashionable. The presentation of Ignite as a practice library allows the practice set to respond to the needs of the users, who may regularly add new practices and retire those that are no longer needed.

The process of extracting the practices from an existing method or methodology is called *essentialization*. Essence is designed to allow people to extract the essence of any method or practice, so essentialization of a method means identifying the method's practices and practice architecture. Moreover, each practice is described/codified in terms of the elements in Essence and the Essence language with new practice-specific elements added as needed. As of this writing, the Unified Process has been essentialized, and DSDM (dynamic systems development

method] is in process. Several other methods are in the planning stages to become essentialized. Many companies around the world are now using the Essence standard to essentialize their methods.

The value of essentialization is that people can easily learn what really matters about a practice, compare it with other practices, compose it into a method (with many other proven practices), and easily modify/change the method as new knowledge becomes available. Applying Essence also

makes it easier to govern the methods you have in your organization, so you create an effective learning organization. Moreover, an essentialized method is not just a static description, but helps the team while they actually use the method, allowing them to measure progress and health at any moment during their endeavor.

Less work has been done to capture the domain-specific practices needed to bridge the gap between craft and engineering. As seen earlier, the concepts can be illustrated using Ignite[®] and other popular methodologies, but a vibrant and committed engineering community must flesh out and complete the necessary set of engineering practices.

There are two ways to accelerate the transition:

- Slice the popular methods into practices and design these practices so that they can be composed in any reasonable way teams

Some Definitions



The terms used in the method space are often ill-defined or confused. For example, what is the difference between a method and a methodology? A practice and a process?

Essence provides the following simple definitions used throughout this article, but particularly relevant in the section on Essence.

Practice: A repeatable approach to doing something with a specific objective in mind.

Method: The documentation of a team's way of working. A method may or may not be documented using Essence. If it is documented in Essence,



then it is the composition of the Essence kernel and a set of practices to fulfill a specific purpose. A method could belong to a single team or be shared among teams.

Essence kernel: An actionable reference model of software engineering that provides a framework for the definition of practices and the assembly of methods.

Way of working: This is what a team actually does. It may or may not match their method.

Methodology: A collection of practices known to share a common set of values and work well together. It's a form of practice library.

Practice library: A collection of potentially competing practices. For example, a requirements management practice library could contain many different competing practices such as declarative requirements, use cases, and user stories.

Starter pack: A partially built, often incomplete method that a team can use as a framework to seed their own method.

want, maybe resulting in a method with practices from several already-existing methods such as DAD, SAFe, LESS, and SPS.

➤ Codify existing or new practices so that they can be composed with other practices to form complete methods. There are already hundreds of practices in the world, but they are not described in a way that allows them to be easily composed. Now this can be done without having to describe a complete method.

In both cases Essence is key as it provides the foundation for this work and for the industry to transition successfully from craft to engineering.

CONCLUSION

As software becomes more and more essential to the world's day-to-day activities, it is time for software development to move beyond a craft-based approach to become a true engineering discipline.

This will require a shared base of codified engineering practices that can be reused across various technical domains and various types of software; this set of practices will grow and adapt as better ways of developing software come along.

This is not going to happen overnight, but it is a challenge to which our industry



Composition: The process of merging practices into practices and methods. It is important to understand that practices are separate concerns composed through a merge operation and not components interacting through messages.

Essentialization: The process of rendering a method or practice down to its essence and capturing it using the Essence language.

needs to rise as it matures and evolves into something beyond agile and other current practices.

We still need the dedication, innovation, and invention of craft, embodied in:

- Skilled professionals, passionate about their subjects and committed to mastering new, complex, fast-moving technologies.
- Local experts who understand complex problems in depth and respond rapidly to changing needs, perceptions, and challenges.

However, we also need the codified knowledge and professionalism of an engineering discipline to be able to:

- Sustain and grow delivery capability through changes in technologies, teams, and suppliers.
- Predictably scale operations from early prototypes to global rollouts.
- Take control of investments and know when to pivot to solutions more likely to deliver favourable returns.
- Systematically grow the levels of reuse and interoperability of solution components and systems.
- Produce long-lived solutions with affordable costs of ownership.

This is what we mean by moving from craft to engineering—a journey that needs to be made practice by practice, domain by domain. Thanks to Essence, that journey can start today for all of us.

References

1. Graziotin, D., Abrahamsson, P. 2013. A web-based modeling tool for the SEMAT Essence theory of software engineering. *Journal of Open Research Software* 1(1); e4; <http://dx.doi.org/10.5334/jors.ad>.
2. Ivar Jacobson International. 2015. Agile Essentials; https://www.ivarjacobson.com/sites/default/files/field_iji_file/article/agile_essentials_paper.pdf.
3. Jacobson, I., Seidewitz, E. 2014. A new software engineering. *acmqueue* 12(10); <http://queue.acm.org/detail.cfm?id=2693160>.
4. McDonough, A. 2014. Munich Re and ESSENCE – Kernel and language for software engineering methods: a case study. Object Management Group; http://www.omg.org/news/whitepapers/Munich_Re_Essence_Case_Study-2014-12-01_JP.pdf.
5. Object Management Group. 2014. ESSENCE – Kernel and language for software engineering methods (Essence); <http://www.omg.org/spec/Essence/>.
6. Ralph, P., Johnson, P., Jordan, H. 2012. Report on the first SEMAT workshop on general theory of software engineering (GTSE 2012). *ACM SIGSOFT Software Engineering Notes* 38(2): 26-28.
7. Shaw, M. 1990. Prospects for an engineering discipline of software. *IEEE Software* 7(6): 15-24.
8. Slama, D., Puhlmann, F., Morrish, J., Bhatnagar, R. 2015. *Enterprise Internet of Things: Strategies and Best Practices for Connected Products and Services*. O'Reilly.
9. Stimson, R. 2015. Industrial scale agile—challenges and solution strategies. Ivar Jacobson International; <https://>

www.ivarjacobson.com/publications/white-papers/industrial-scale-agile-challenges-and-solution-strategies.

Additional Essence resources

For help in really understanding what Essence is, how it can be used to build practices and methods, and the value it gives when used, we recommend you read:

Jacobson, I., Ng, P.-W., McMahon, P. E., Spence, I., Lidman, S. 2012. The Essence of software engineering: the SEMAT kernel. *acmqueue* 10 (10); <http://queue.acm.org/detail.cfm?id=2389616>.

You can complement this by also reading:

Jacobson, I., Ng, P.-W., McMahon, P. E., Spence, I., Lidman, S. 2013. *The Essence of Software Engineering: Applying the SEMAT Kernel*. Addison-Wesley.

Jacobson, I., Ng, P.-W., Spence, I., McMahon, P. E. 2014. Major-league SEMAT: why should an executive care? *acmqueue* 12(2); <http://queue.acm.org/detail.cfm?id=2590809>.

Jacobson, I., Spence, I., Ng, P.-W. 2013. Agile and SEMAT: perfect partners. *acmqueue* 11(9); <http://queue.acm.org/detail.cfm?id=2541674>.

Ivar Jacobson, Ph.D, is a father of components and component architecture, use cases, aspect-oriented software development, modern business engineering, the Unified Modeling Language, and the Rational Unified Process. His latest contribution to the software industry is a formal practice concept that promotes practices as the “first-class citizens” of software development and views method (or

process} simply as a composition of practices. Jacobson is also one of the founders of the SEMAT (Software Engineering Method and Theory) community, the mission of which is to refund software engineering. He is the principal author of seven influential and best-selling books and a large number of papers. He was awarded the Gustaf Dalén medal (“the little Nobel Prize”), and he is an honorary doctor at San Martin de Porres University, Peru.

Ian Spence *is CTO at Ivar Jacobson International and the team leader for the development of the SEMAT (Software Engineering Method and Theory) kernel. An experienced coach, he has introduced hundreds of projects to iterative and agile practices. He has also led numerous successful large-scale transformation projects working with development organizations of up to 5,000 people. His current interests are agile for large projects, agile outsourcing, and driving sustainable change with agile measurements.*

Ed Seidewitz *is the former CTO Americas at Ivar Jacobson International. He is experienced in agile system architecture and development in both the commercial and government sectors. His work ranges from business process analysis to system architecture to full implementation of enterprise-class information systems, deployed in the data center or in the cloud. He has leading expertise in UML (Unified Modeling Language), including involvement in the continued evolution of the standard, as well as a background in state-of-the-art information system technologies.*

Copyright © 2016 held by owners/authors. Publication rights licensed to ACM.