THE KERNEL JOURNALS

Roly Stimson

Table of Contents

This e-journal discusses how development organizations can make step changes and sustainable improvements in their software delivery capability by using a kernel. A software process kernel is a model of the essentials of what all software development teams have to do to be successful, irrespective of precisely how they choose to do it. This journal focuses on the practical applications and benefits of a kernel, which includes de-risking process change and improvement, and minimizing process constraints and overheads while maximizing visibility of progress and predictability of outcome.

The Kernel Journals 1:
The hegelian dialectic of software engineering

We in the software development industry face a seemingly intractable problem. We have learnt the lesson that prescriptive process is a bad thing. Process bureaucrats sitting in ivory method towers, telling highly-skilled professionals how to do their job and setting the process police on them if they don't follow their instructions to the letter, can (unsurprisingly) be quite damaging. It disempowers the development team and engrains apathetic attitudes along the lines of "When we inevitably under-deliver, it will not be our fault, but the fault of these ludicrous process hoops that we are forced to jump through, instead of being able to focus on writing great software". The agile revolution was software engineering's way of learning this lesson, and the agile manifesto pledge to value "people over process" and "software over documentation" has got to be right. But (… there was always a "but" coming …), we are already finding that the opposite extreme of little or no explicit process isn't going to cut it either, because it leaves too many problems unsolved, such as:

1. **Avoiding confusion** – there is a big and noisy world of competing processes and methods out there. How do software development teams find the right practices for their specific needs?

2. **Team efficiency** – do we really want our development teams to constantly have to reinvent new solutions to old process problems when they should be focusing on writing great software?

3. **Supporting diversity** – some software engineers need / want more explicit practice guidance than others. How do we provide just enough guidance without being overly prescriptive?

4. **Consistency** – if every team works differently, how do we get resource mobility / flexibility?

5. **Governance** – how does the sponsoring organization manage the risks associated with its investments in software and ensure visibility of progress towards a successful outcome?

We face the following Hegelian Dialectic (I graduated in philosophy – you have to humor me!):

1. **Thesis:** Prescriptive process kills the golden goose (development team creativity and motivation)

2. **Antithesis:** Removing the prescriptions can equally distract, confuse and frustrate the team

3. **Synthesis:** We need some kind of smart practice framework that enables us to provide clear direction on *what* needs to be done and *just enough* default standard guidance on *how* to do it, but enables teams to easily "swap out" and replace practices that are not exactly right for them.

This smart practice framework is known as a process kernel. In this short series, I will describe in a little more detail what a kernel is and report on my experiences of using process kernels in practice to enable software teams to find answers to some of these seemingly intractable process problems.

The Kernel Journals 2:
An executable model of software development

I first learnt about the power of domain models more than 25 years ago when I first applied the Jackson System Development (JSD) process. This approach involves modeling the key conceptual entities in the problem domain and the business rules that define how value is delivered by advancing the value states of these key entities. Add a few key business attributes and you now have an executable model of your problem domain / business. You can then simulate the execution of your business merely by slapping some rough-and-ready user-interface screens onto these entities.

For example, if we are building an accounting system a very few domain entities, such as "Account", "Account Entry" and perhaps "Account Holder", quickly become central and critical to the whole endeavor. They feature in all our conversations, models and statements about the business domain and the business rules that in turn form a key part of the solution requirements. For the Account entity, we will find ourselves discussing what actions we can do to it (Request, Create, Activate, Suspend, Close etc.), what the outcomes of these actions should be and how these will depend on the current state of the Account (Open, Suspended, Closed etc.).

When object-orientation proper arrived I was fortunate that one of my first points of reference was Ivar Jacobson's book on Object-Oriented Software Engineering. I was particularly struck by a statement on Page 340: "to start identifying the use cases it is often appropriate to have a first picture of the system in terms of some problem domain objects". And so it has proved to be on the numerous software development projects I have been involved with ever since.

So, why have we as software process engineers not got around to eating this most excellent and nutritious software process dog-food ourselves?

As part of Ivar Jacobson International I was lucky enough to be around when Ivar and his team first asked and answered this question some four years ago by developing just such an executable process model, which we call a process kernel. The beating heart of this process kernel is (… wait for it …) a domain model of the key entities that we need to progress (known as "Alphas") and the states that we progress them through as we execute a software development project.

For example, concepts like "software system", "requirements", "team" and "project" crop up in all our conversations about where we are now and what we need to do next to progress a software project in a controlled way to a successful conclusion. A kernel simply codifies these key concepts and progressions in a standard way so that we can frame these conversations and formalize their outcomes in a way that is clear, concise, unambiguous and measurable.

In this series of Kernel Journals, I will share some of the experiences I have had over the past few years in applying this simple but powerful concept to solve a number of process problems for software teams, including:

- How can we get rid of our process documentation overheads without losing process maturity?
- How do project teams get the help they need, when they need it, without prescriptive process?
- How can we enable teams to own and adapt their working practices efficiently and effectively?
- How can we judge whether a team's way of working is sufficient / fit for purpose?
- How can we share successful team practices and the lessons learned in applying them?

The Kernel Journals 3:
Process spaces and bases

Software development processes have long advocated structuring a software solution around a domain model of the problem space being automated. A domain model shows how our business processes add value by progressing the states of our key business entities. These entities and their life histories tend to be much more stable over time than the processes that surround them. Modeling the entities and their states enables us to experiment with different ways of achieving the same outcomes (state progressions) as we seek to rationalize and automate these processes.

So, why have we never thought to build a good domain model of the software projects at the heart of our software development processes? At IJI, we started building such a model some four years ago and this model now forms the heart of our process kernel around which we built the Essential Unified Process. One key motivator was to model the value that different development practices and processes can / do / should provide so that we could enable our customers to evaluate and select between different ways of achieving the same outcomes.

Some of the key entities in the IJI software development kernel include:

- Project – the software development project itself
- Requirements – how the software system we are developing should behave
- System – the software system we are developing
- Opportunity – the opportunity / problem space into which the system delivers value
- Team – the team that undertakes the project to develop the system

These are the "spaces" in which we can expect development practices and processes to add value, for example by helping us to:

- Steer the *project* to a timely and successful conclusion
- Understand, share and confirm the *requirements*
- Build the right team for the job and enable it to collaborate and operate effectively

Even at this very high level we can start to put this simple domain model to effective use. For example, we can get various stakeholders to rate our current capabilities within each space as a precursor to any process improvement or practice rollout initiatives – how critical is each space to our success, and how good or bad a job do we do in each space currently? If we find that we do pretty well in the project and team spaces, but we suffer significant pain in the requirements space, then looking at adopting a project management practice like Scrum should perhaps be a lower priority than looking at requirements practices such as Use Cases or User Stories.

Now, if we start to think specifically about the kind of advances or progressions that we need to achieve early on in any project, we can say for example that we need to:

- Agree the key project milestones, such as the next planned release point (i.e. progress the *Project* to state *Milestones Agreed*)
- Share the goals and scope of the release that we will produce at that point (i.e. progress the *Requirements* to state *Shared*)

Having established WHAT we need to achieve, we can now look at the different options that we have in terms of HOW we might go about achieving it. Scrum, for example, has the concept of release planning, which tells us that we need to do these things, but gives us little specific guidance on how to achieve them. User Stories are great for prioritizing and sizing the requirements and driving and tracking requirements progress, but we really need a scope-level view before we can go about getting the right kinds of users into the right meetings and give them the right scope steers to get the right kind of user stories coming our way. Actually, a Use Case diagram is a great way of very rapidly sketching, agreeing and sharing the scope, context and user types for a release in support of Scrum release planning and in preparation for a user story workshop. Suddenly the "Use Cases vs. User Stories", "either or" "method war" is shown to be an unhelpful and overly simplistic view of the world that prevents us discovering and leveraging some powerful practice combinations.

So, even with a few, very simple and obvious codified "spaces" (Alphas) and "bases" (state progressions) we can start to achieve some very powerful things that no other process construct can help us with, namely:

- Establish and communicate exactly WHAT a given technique, process or practice helps us with (and what it does to *not* help us with)
- Evaluate potential practices and processes against identified process gaps, needs and priorities
- Establish which processes, practices and techniques are complementary and which are competing for like-for-like alternatives.

The Kernel Journals 4:
A cure for document and template addiction

Many organizations have achieved a degree of process maturity (reliability, discipline, consistency) only by paying a very heavy price – they have become addicted to documents and document templates.

Unfortunately, it can happen all too easily. Most processes end up being document-centric even though they never set out to be so. They start by offering useful process guidance on how to progress the project in a controlled way, in the form of a set of activities, each of which is defined in terms of the artifacts it produces. Most artifacts are documents of some kind and the process helpfully comes with templates for each document – a template is better than having to start with a blank sheet of paper, after all. The project milestones we need to pass through are evidenced using the documents, and the whole thing hangs together nicely.

The problem is that in practice the documents rapidly become ends in themselves. The way to progress, after all, is to fill out the templates and get someone to agree to the results. To make sure the documents are in good shape, they are reviewed for completeness (i.e. no sections of the template left empty) and correctness (no typos or spelling errors), and are reworked and re-reviewed until they are 100% complete and typo-free. Even attempts at declaring that we are "iterative and incremental" by adding statements to the documents declaring them to be "living and evolving" doesn't seem to help our case - our world now revolves around documents and it takes us weeks or months to get enough agreement about the vision and the plan to even start the real job of writing some software.

But going back to "coding and hoping" that what we're building might be what was really needed won't hack it either. We know there are certain pre-requisites to "sprinting" – like enough handle on the vision (problem, stakeholders, needs, solution scope), technical approach and platform, and enough of a credible release plan for it to be right to make the investment in letting a fully tooled-up development team loose on the job. But how to make sure we are "OK to go" without getting into the document review and approval time trap?

What we need is a simple way of characterizing what state we need to progress the different aspects of our project to (e.g the problem is understood, the release is scoped and the release milestones are agreed) without prescribing exactly how we should evidence it. Frankly, any old evidence will do as long as the people that need to agree on what needs to be agreed are happy to agree to it on the basis of whatever it is they have seen.

So, once again our kernel domain model comes to the rescue, with its simple characterization of the key things that all projects need to manage (such as the project, the requirements and the team) and the states that we need to progress them through (such as "project milestones agreed" and "release scope agreed"). By adding a few simple assessment criteria to the states (such as "the needs of the key stakeholders have been captured ", "the next release point and release cycle milestones have agreed target dates", "the scoping requirements have been prioritized for the release"), we can rapidly agree what needs to be agreed without worrying too much about precisely how it is documented, or whether it is totally typo-free.

And this is more than empty theory – I have seen software projects in mature development shops cut their lead time (from project start to starting the first "sprint" to develop and demonstrate increments of production quality code) from weeks or months to just a few days, without losing the ability to progress in a controlled way through the quality gates of the governance framework within which they operate.

The Kernel Journals 5:
Making the invisible, visible

The kernel Alphas are the core, key, critical, central, essential conceptual entities that we need to manage and progress in a controlled way in order to ensure a successful outcome for our project. But, like all concepts, they have the distinct disadvantage of being invisible. A project manager is convinced that his project is important and that managing its progression to a successful conclusion is critical. But if, as an outside assessor, I were to say "bring me this project of which you speak so that I may gaze upon it and assess its status" he would be stumped. He can show me plans, people, documents, but he can't show me "the project". These key concepts are the elephants in the room that no one can see because … well , because they are invisible. To be useful, they need to be made visible and real, so normal people can interact with them.

In the first Kernel Journal, I talked about how domain models can be used to simulate the execution of a business by adding attributes to them and slapping a few crude user-interface screens over them. And we can do exactly the same with the Alphas that are the domain entities in our software development process kernel. This has become known in my sad, process-geeky world as "skinning the kernel".

A great way to get a quick-and dirty but fit-for-purpose "skin" in place is to use a wiki: one wiki page for each required view onto the alpha states and attributes. Wikis are useful because they solve four process problems in one:

- Where do we put our key information as we gather it?
- How do we communicate the information to all our stakeholders?
- How do we get agreement and consensus?
- How do we record key agreements and decisions?

We capture the key project information where people can see it, contribute to it, refine it and agree to it.

To help projects focus on what the key project information is and to help them to capture it in a simple, structured, concise and consistent way we can provide a simple set of wiki templates. (After all it's better to have a template than to make each project start from scratch with a blank wiki each time).

(But wait, oh no, not templates!! We're not supposed to use templates, because we are recovering template addicts. How are we going to stop ourselves becoming addicted to this (admittedly small, lightweight and relatively harmless looking) set of templates?)

Fortunately, we already have the antidote in place. Our templates can't become ends in themselves, because we know that they are only a means to the end of progressing the states of our key Alphas. We can easily define who needs to assess which states and when (e.g. "the product owner needs to agree to the target date and scope of the next release point before we start incrementally developing the product release"). We then close the evidential loop by stating where the evidence is to be found: "based on the information in the wiki Project and Requirements pages". If the assessing party is happy that the progression has been achieved, based on the evidence they see, then the progression has been achieved (even if there is the odd spelling mistake on the wiki!).

So now we have a simple way of executing our development project by capturing the information we need to progress our project in a controlled way to a successful conclusion. And, once again, this is more than just process theory. I have seen many projects use this approach to reduce their documentation overheads from dozens of documents totaling many hundreds of pages (which of course virtually no one ever read) to a dozen or so wiki pages which told everyone exactly what they needed to know about the project and its status, when they needed to know it.

The Kernel Journals 6:
Where to (first / next)?

We all know that we want to "cut to the chase" as soon as we can and start incrementally developing the software product through which we deliver value back to the business. But we also know that there are certain essential pre-requisites to "sprinting", such as some kind of vision of where we are supposed to be going and the right team and tools to get us there. If we start motoring before we are ready we may head off in the wrong direction or we may find that the wheels come off as we accelerate through the gears.

It's confession time. I'm a huge fan of Barry Boehm's standard project milestones, as originally set out in "Anchoring the Software Process" [Barry Boehm, November 1995]. Honestly, you can run your life with these simple but powerful check-points. Personally, if I'm given any significant job to do, I make sure that:

- Firstly, I confirm my understanding of what I need to achieve, and when by
- Secondly, I make a list my biggest worries – what might cause me to fail and anything that I just don't know how I am going to do or whether I can do it in time, and then I do the bits of the job that can resolve my uncertainties and relieve my worries (or rapidly prove them well-founded) and play back the resulting, decisions, outcomes and implications
- Thirdly, I get my head down and get the rest of the job done

My three steps are, in essence, Barry Boehm's common milestones. The problem with great practices like this, though, is that in the past they have tended to be glued together with bits of other great practice to form monolithic, prescriptive processes. So, we find Boehm's milestones inside the Rational Unified Process, but mixed up with a lot of other guidance about the project phases to achieve the milestones, sub-dividing phases into iterations (normally at least two-to-six weeks long), the activities we perform in each iteration and the documents they produce that we then use to assess the milestone. But, of course, it's hard to produce documents and get them reviewed, reworked and approved in one iteration. So, before we know it, all our projects are spending one-to-three months messing around with Vision documents and the like before any code is cut (irrespective of how "nearly ready" they were to get motoring on day one of the project). Whatever happened to "cutting to the chase"?

It just so happens that the original motivation for Boehm's common milestones paper exactly matches the challenges we face today as we seek to empower software projects to use the practices that best suit them: "The current proliferation of software process models provides flexibility for organizations to deal with the unavoidably wide variety of software project situations, cultures, and environments. But it weakens their defenses against some common sources of project failure, and leaves them with no common anchor points around which to plan and control." To get back to Boehm's original intent, we need a way of characterizing the state that our project needs to be in, for example before it is right to start sprinting, that is independent of whatever the appropriate means of getting there might be.

This is where we come back to our software development kernel and the domain model at its heart. This characterizes the key aspects of a project that we need to progress (known as Alphas) and the states that we need to progress them through to advance the project in a controlled way to a successful conclusion (without the wheels coming off). To define a major project milestone such as Barry Boehm's first "life cycle objectives milestone", we simply need to list the Alpha states that we need to achieve for this milestone. For example, some key state progressions required for the first (lifecycle objectives) milestone are:

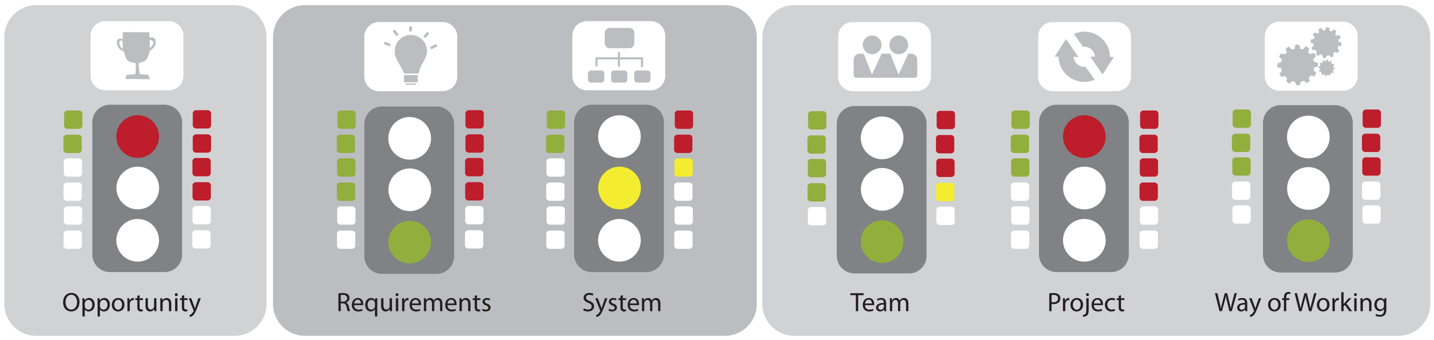
- The *Requirements* are *Shared* (scope and value agreed)
- The *Team* has its *Mission Defined* and (ideally) a team has been Formed to achieve the mission
- The *Implementation* has an *Approach Selected* (e.g. the language / platform must be agreed)
- The *Project* has its *Milestones Agreed* (roughly what needs to be done and when by)

Depending on how far off we are from being where we need to be, we can set ourselves a target date for getting there and select the appropriate practices to help us get there on time. On a simple project, with a clear steer, and a lightweight approach to confirming what we need to confirm (as described in the previous Kernel Journal), we can get there in hours or days, not weeks or months. And so cut to the chase!

The Kernel Journals 7:
SatNav for software development projects

In the last Kernel Journal we looked at the problem that Barry Boehm was aiming to solve back in 1995 when he first proposed his standard process milestones (which later gained industry prominence as the milestones in the Unified Software Development Process and the Rational Unified Process), namely that "the proliferation of software process models provides flexibility", but leaves us "with no common anchor points around which to plan and control." [Barry Boehm, November 1995]. We looked at how a small set of domain entities with simple state progression models (which we call "Alphas") can make these common anchor points much more practical and useful while ensuring that they remain process-neutral and do not become "document-driven".

The alphas, when used with common milestones such as the Unified Process Milestones, can actually give us much more than this – they can provide a project status and health dashboard that can be used by the customer and supplier organizations to assess the current status and health of any / all projects, irrespective of which processes or practices they are following. The graphic below shows just such a dashboard, with a set of kernel alphas and a traffic-light status for each alpha, which is derived by comparing where the project is now (the state machine to the left of each traffic light) with where it needs to be to achieve the next project milestone (the state machine to the right of each traffic light).



In Kernel Journal 5: "Making the invisible visible" I described how we can easily "skin" a process kernel, by providing a portal for projects to capture, share and agree to the essential project information that is needed to achieve each state progression (for example, using a set of templated wiki pages). Once we have done this, we can make the alpha dashboard much more useful to the project teams themselves, by flagging which sections of the project portal need to be updated and agreed upon to get the project to where it needs to go next.

This gives us the equivalent of a Satellite Navigation System for our software projects project that enables us to:

- Set our journey destination and waypoints (milestones)
- Track where we are now, compared to where we want to be
- Get guidance on what to do next in order to progress towards our destination.

The Kernel Journals 8:
A little help here!

In the previous kernel journals we have looked at the many practical applications of a process kernel that is based on the key software development entities (known as Alphas) that software projects add value to by progressing through their value states. We have seen that there are many advantages to separating these "whats" (progressions) that we are trying to achieve, from the many possible "hows" (software processes and practices) that describe how we might achieve and evidence these progressions, including the ability to set objectives and track progress in a common and consistent way irrespective of what specific practices a project happens to adopt.

So, our kernel gives projects advice on what to achieve (first / next) without providing any specific guidance on how to achieve it. This is both good (because we are not telling software development teams exactly how to do their job) and bad (because we aren't giving them any help at all on how to do the things that they have to do).

Clearly there is no shortage of advice out there in the form of the myriad of competing software development processes. But we have seen that they have traditionally suffered from being somewhat monolithic ("all or nothing at all"), and while many contain much useful practice guidance, it is difficult to find and use the specific piece of guidance that we need without getting tangled up in a huge, prescriptive process web of activities to follow and artifacts to produce (and templates to complete).

What we need is a way of accessing process guidance that enables us to find our way straight from what we are trying to achieve here and now (such as establish a shared understanding of the value and scope of a software system) to specific guidance on how we can achieve it (e.g. with use-cases and use-case scenarios).

This is what IJI has done with it's Essential Unified Process – provide access to guidance on how to achieve something, that is accessible directly from the progressions that we know our project has to achieve first / next. And we have provided this guidance as a set of modular practices, so that if an organization or project does not like a particular practice (such as the use-case practice), they can "swap it out" and "swap in" a different practice into the same process space instead (such as a traditional declarative-requirements practice or a user-story based practice).

Organizations are now free to choose the balance that is right for them between process consistency and process freedom. At one end of the scale an organization can provide a single set of cohesive practices that provide just one single standard way of achieving all the things that projects need to achieve. At the other end of the scale, they can allow projects to select any "hows" they wish from any practice guidance that is "out there", and to freely refine and evolve these working practices over time. A project, for example, might choose to scope its system with a use case diagram and then drive the development and test of the system with user stories. Furthermore, as we will see in the next kernel journal, we always know whether the process guidance available to a team "covers all the bases" or not (i.e. whether there are any spaces left in which people are expected to "just wing it").

The Kernel Journals 9:
Is our way of working fit-for-purpose?

We have seen in the previous kernel journals that a process kernel, defined in terms of the key domain entities, or Alphas, and their value state progressions, can give us a simple, shared view of what each and every project has to achieve (which is common across all projects), that is distinct and separate from how they might go about doing it (which may vary from project to project).

If we provide projects with at least one "how" (practice guidance) for each and every "what" (alpha state progression), then no team ever has to discover or invent new ways of working. But, if we allow teams to adapt or substitute any specific piece of practice guidance with an alternative means of achieving the same end, then we can empower teams to adopt a way of working that best suites the evolving needs of their project and team circumstances.

This approach has significant benefits for process quality auditing and assessment too. In the past, quality assessors within an organization have often had an up-hill battle to determine which process a given team thinks it should be following, before they can even start to assess whether or not it is an appropriate process and finally whether or not the team is actually following it.

Being pointed to a "tailor down" process like the Rational Unified Process has never been popular with quality managers in my experience. The RUP may document everything that any project of any size might ever conceivably need to do, but everyone knows that no project ever should or could attempt to do all of it. But determining and documenting which bits of process do or don't need to be done is challenging and onerous for projects (and therefore is either not done well or not done at all). Likewise, assessing whether the bits of process that the project team is supposedly doing, together constitute a complete and coherent process that is "necessary and sufficient" is almost impossible.

A process kernel on the other hand provides a process quality assessor with a simple process-coverage checklist. We know that each and every project needs to achieve each and every alpha state progression somehow. If the project team knows how they will go about trying to achieve each and every progression, then they have a fully defined process that is both necessary and sufficient (in terms of coverage at least). A process quality assessor can therefore be presented with a project team's defined process in the form of the standard set of alpha state progressions, with each one pointing to the practice guidance that the project team is following in order to achieve the progression.

The current state of the project is easily assessed as it is expressed in terms of the progressions that the project has made, each of which points to the evidence that shows it has been achieved, expressed in the way the practice guidance suggests it should be. The process assessor also knows which practices should be currently "in play" and guiding the planning and execution of the current project activities and tasks – these are the practices that relate to the state progressions that the team hasn't achieved yet, but is striving to achieve next.

The kernel thus enables process quality assessors to effectively and non-invasively assess what process is theoretically being followed, whether it is fit for purpose and whether it has been and is being successfully followed.

The Kernel Journals 10:
The Essential does not change

In the previous kernel journals we have explored the value that development organizations can get from a process kernel that codifies what each and every project always has to do, in a way that is distinct and separate from how different project teams might choose to go about doing it, some benefits being:

- The status and progress of diverse projects can be made visible and tracked in a consistent way
- The overheads associated with evidencing (documenting) project progress can be minimized and does *not* become an end in itself
- No teams need unnecessarily reinvent process wheels – teams can be provided with demonstrably "necessary and complete, but minimal" practice guidance on how they *could* achieve everything they do actually always have to achieve
- Teams can be pointed to one or more alternative tried and trusted ways of achieving what they are trying to achieve on their project here and now
- Process need not be prescriptive no more - teams can be empowered to adapt any specific aspect of their way of working in the light of their project circumstances and team experiences
- Teams can try out new ways of achieving specific things and add them to the corporate memory banks for others to find and use if they work well
- The fitness-for-purpose of a team's adopted and adapted way of working can be easily assessed.

The key to achieving any and all of these benefits is the fact that the kernel is a shared, useful and generally accepted description of the essential aspects of all each and every project within an organization, where essential means:

1. Critically important and
2. Invarient

IJI has provided its customers with just such a standard process kernel and supporting practices over the past four years and have enabled them to reap many of these benefits as a result. Other process suppliers are also converging on a similar "kernel plus practice" approach to structuring, presenting and deploying their process offerings. Currently, the exact nature, shape (and even the names) of the underlying process kernels varies from process supplier to process supplier. If practices from different sources are to be made available to teams as interchangeable pieces within the picture of a team's preferred ways of working, it is clear that we as an industry need to define and agree some standards for these process kernels.

Standards, standards, standards. Frankly, the heart sinks at the very sound of the word. However, it really, really, really has got to be a lot easier to arrive at a shared consensus on what all project teams have to do, than it has ever been to achieve any kind of consensus on how they should go about doing it. All software development projects really do have a stack of stuff in common. All we have to do is codify it well.

Just to pluck a few examples of where supposedly warring parties more than agree:

- We all talk about software development *projects* and agree about importance of key project milestone dates such as software release dates
- We all talk about software being a team sport and emphasizing how important team and stakeholder communication and collaboration is to the success of this endeavor
- Many of us talk about the need to control project progression by iteratively and incrementally preparing and verifying sub-sets of the product release (although some of us tend to call these "iterations" and others "sprints")
- Most of us talk about the expedience of achieving early agreement on a vision or scope-level view of what needs to be achieved in a software release, without attempting to predetermine and cast in stone everything about the functionality and behavior of the release

This list of things we all agree on is easy to write and could very easily be made a lot, lot longer. And this should not surprise us, because the various processes and practices out there all share the same problem domain after all. And just beneath the surface we so-called methodologists are all experienced campaigners with similar war stories to share.

If we can't come together to codify this problem domain in a common way to help our mutual customers find, compare, contrast, select and use the right practices for the right jobs then I for one think it is a pretty bad job.

This is the job that the SEMAT consortium is now embarking upon. I hope you will join us.

1 For those of us for whom "the obscurer the better", "The essential does not change" is a quote from Samuel Beckett, which has certainly those of us well-versed in the Lockean and Berkleyan metephysics rolling in the aisles (I told you'd have to humor me!)

References

1. Barry Boehm, Anchoring the Software Process, USC November 1995
2. Philippe Kruchten, The Rational Unified Process: An Introduction, Addison Wesley (December 2003)
3. The Essential Unified Process -
   http://www.ivarjacobson.com/process_improvement_technology/essential_ unified_process_software/
4. Software Engineering Method and Theory (SEMAT) – www.semat.org
5. Samuel Beckett, Waiting for Godot, Faber and Faber (1956)

About the Author

Roly Stimson graduated in philosophy from Southampton University and trained in software engineering at Aberystwyth University. His career in IT spans some 26 years, always in and around software development, usually directly involved with development methods and tools. Roly has experience of many diverse disciplines and roles including business analysis, requirements, architecture, design, development, testing, team leadership, process authoring, training and consultancy. He has developed in environments that range from punched cards, machine code and assembler, through 3GLs and CASE tools to enterprise object-oriented development platforms. He has worked on many diverse applications, from embedded real-time to large-scale commercial, using methods and tools from traditional large-scale government and commercial ("waterfall") methods, through object-oriented analysis and design approaches, to iterative, incremental and agile processes. Outside software engineering his passions include his family, jazz music, modern literature and philosophy (still).

United Kingdom
+44 (0)20 7025 8070
info-eur@ivarjacobson.com

Americas
978-649-2856
info-usa@ivarjacobson.com

The Netherlands
+31(0) 20 654 1878
info-nl@ivarjacobson.com

Sweden
+46 8 515 10 174
info-se@ivarjacobson.com

Asia
+8610 82486030
info-asia@ivarjacobson.com

Ivar Jacobson International

Ivar Jacobson International is a global services company that helps software organizations transform and improve the way in which they develop software solutions as well as guide them in meeting the expectations of the business. Our consultants provide an environment of experiential learning to develop the right competency levels amongst all roles and functions by becoming an intricate coach and mentor to the entire team. We have a framework that we adapt to effectively define and communicate business and technical expectations across the organization as well as create collective responsibility by teams and individuals for project outcomes. We introduce a proven practice driven approach that is goal oriented, incremental and measureable and is highly successful with either an existing software project or the implementation of new systems. We support our customer engagements with a rich set of technology assets inclusive of training materials, practice guides, and tooling.

Find us on the Web:
www.ivarjacobson.com

IVAR JACOBSON
INTERNATIONAL