The Problem of Prolific Process

Balancing the Quantity and Quality of Documented Process

Phillip Glen Armour, Corvus International Inc., QSM Inc.

Abstract. What is the optimal amount and level of detail for predefined and documented (and enforced) process for systems development? This question has been debated for decades by software practitioners, computer theorists, and those responsible for resourcing the business.

Introduction

Should we have more process quantity, more process detail, more process options (and more rigorous enforcement of process)? Or should we just leave developers to figure out what they need to do as and when they do it? On one side we have the view that if process is good, then more process must be better—such philosophies can generate enormous volumes of paper-based process documents or their electronic equivalent. On the other hand there are advocates of process so lightweight it hardly exists; with this approach developers are pretty much left to their own devices to work out what to do.

"Big process" assumes that developers will (a) read the immense amount of process documents before or during development (b) understand what is written (c) figure out how to apply it to their situation and (d) make any necessary process adjustments while staying true to the original intent if not the letter of the documented process. This approach also assumes that all this adherence to pre-defined process will make for higher quality systems or make the process faster and less costly or provide a better basis for system compatibility, extension or maintenance. The advocates and authors of such process rarely seem to concern themselves with any negative effects on the morale, creativity, or sense of achievement the developers might experience when they work this way.

On the other hand, those who espouse very lightweight (if any) process assume that developers will (a) actually remember all the activities needed to build a system (b) consistently apply all these steps (c) apply their innate creativity (now liberated by freedom from oppressive process) to more than compensate for anything they miss. These advocates also assume that the developers will have the requisite experience and skill to do all this.

The Second Law

It is clear that the answer lies somewhere in the middle. Predefining everything we should do to build a system is just not possible. If it were, we could automate the process and we would not need people at all. However, allowing each project and each developer to make it up how they work each and every time they build something is a recipe for anarchy. We did that 30 years ago and it did not work very well; in fact the move to big process was fueled in part by the erratic results laissez-faire development gave us. And then the move to Agile was driven by the reaction to the stifling overhead of big process.

It seems that developing process documentation at just the right level is hard. I described this difficulty in the Second Law of Software Process: *We can only define software process at two levels: too vague and too confining [1].*

The irony is intentional and it reflects the dilemma we have when writing process:

- **Too Confining:** if the written process attempts to define all activities under all conditions for all projects building any kind of system, or even a reasonable subset of the same, it becomes very large. Simply because it is very large people will be reluctant to read it. It also becomes difficult to dig through the mountain of documents to find the relevant bit of process just when it is needed. Even more problematic is the constraint that overly large process may enforce. While detailed process is helpful in defining what has occurred before, it cannot explicitly define how to build or test something that is new. In fact, defined process tends to force solutions similar to those that have been built before—specifically the solution scenarios that were used to build the process. It is this inhibiting of the creative process that most lightweight process advocates dislike.
- **Too Vague:** if the written process consists of high-level guidelines, a loose *meta-process* framework within which developers operate freely, ignoring it, modifying it and adjusting it as they wish, the process does not add much value. That is, working with the process and without the process is pretty much the same thing. In this case people complain that the process does not provide useful guidance and direction—the process has no "meat."

Balancing Act

Caught between the hard place of too much documented process and the rock of not enough, how can we find the sweet spot? It is a balancing act. But we also need to take a look at what process is, how we get it, what we expect it to do for us, and how we make sure it works. For an example of how balanced process might be built let us go back to October of 1935.

Failing Fortress

On its second evaluation flight Boeing's Model 299 (the prototype of what would become the B-17 Flying Fortress heavy bomber) crashed. It was flown by Major Ployer Peter Hill who, as one of the Army Air Corp's most experienced test pilots, had flown and evaluated nearly 60 of the Air Corp's newest aircraft. The crash was caused by the pilot's failure to disengage the B-17's gust locks (devices designed to lock control surfaces while the plane was parked). In dealing with the novel and complex demands of preparing and flying an experimental four-engine bomber, Hill forgot a very important step. He just forgot and it cost him his life.

The solution to this kind of problem was not more experience or more training; Major Hill and his co-pilot had plenty of both. The solution was simple process. It was from this beginning that the pilot checklist was born: a simple list of things to do to ensure the plane was set up correctly to fly safely.

Floating Flight 1549

At 3:27 p.m. on January 15, 2009, US Airways flight 1549 struck a flock of Canada Geese at 2,800 feet on its climb out from La Guardia airport in New York City. Immediately after impact, Captain Chesley Sullenberger took the controls while First Officer Jeffery Skiles began working the three-page emergency checklist on how to restart the engines. Four minutes later, Captain Sullenberger landed the unpowered 42-ton aircraft in the Hudson River to the west of 50th Street.

The incredible feat of safely landing a huge airplane on water at around 150 mph received widespread publicity and the pilots and crew were accorded well-deserved accolades. The use of the emergency checklist was not so well known.

Essential Process

The story of flight 1549 gives us clues to what constitutes good process and where process has its limits.

- **Value Added:** given the criticality of the situation, the pilots did not have the latitude to make a mistake in attempting the engine restart. Simply forgetting one step, or working steps in the incorrect order, might have had catastrophic consequences. When stress is high the human brain may not function flawlessly and a simple reminder can help avoid a lot of problems. With their passengers and their own lives at stake, the pilots would not have used *any* process that did not add immediate value.
- **Routine, Well-defined:** the restarting of a jet engine is mostly done the same way each time. There is no value to be added by experimenting with novel ways of powering up a jet turbine and, in this situation, there could have been a lot to lose by using an ineffective process. Process works best for things which are precise, repeatable, well-defined and for which there is no point in doing things differently.
- **Not for "New":** Captain Sullenberger did not use a checklist to actually land the plane in the water; no such checklist exists. Even if a set of rules for landing a large commercial jetliner in a river next to a major metropolis did exist, the crew would not have had time to reference it and land the plane. When something is "new" there are intrinsic limits to what process can achieve.
- **Not if Too Many Specific Conditions:** the pilots had to deal with an enormous amount of information on the wind, the behavior of the plane, communicating with the cabin crew, the passengers and the Air Traffic Control. The combination of these conditions was quite specific to this particular situation. Any "process" would necessarily have to abstract the situation to a set of generalized conditions and the pilots, with only four minutes available to them, would have had to decode these generalizations. Even when there is previous experience available and the situation is not entirely "new," if there are specific conditions that apply to a particular situation, attempting to apply a pre-defined process will take more time and will be considerably less valuable.
- **Succinct:** there are many valuable books on flying airplanes in difficult situations. These pilots did not have time to reference and process them. The engine restart checklist contains only and exactly what is needed to restart an airplane engine under emergency situations.

Process works best when it contains only what is *essential.*

Novel Projects

To some extent, software projects are always "new." We are always building something we have not built before—otherwise we should simply use what we built last time. That said, much of what we do in the business of software *is* repetitive. There are many aspects of our work that can and should be done the same way over and over. But there are also things for which previously defined process does not quite apply at the prescriptive level. Perhaps this is where we can define the boundary of process and extemporization.

What We Know, What We Do Not Know

Building systems consists of two kinds of work: the application of what we already know and the discovery of what we do not know (followed, of course, by its application). By "application" I mean the translation of that knowledge into the executable form we call "software." What we already know, we can call "Zero Order Ignorance"—provably correct knowledge (or its inverse, lack of ignorance).

What we do not (yet) know can be divided into several categories: those things we know we do not know or "First Order Ignorance" (where we have a well-formed question, but do not have the answer) and what we do not know we do not know or "Second Order Ignorance" (where we do not know enough to form even a good contextual question) [2].

Well-defined prescriptive process can work well for Zero Order Ignorance (0OI) and some of First Order Ignorance (1OI), but it cannot work well for the more complex 1OI and for Second Order Ignorance (2OI). Since software projects contain all of these, the process must flex.

Well-defined

Prescriptive process can be developed and should be used for those aspects of systems development which are boring and repetitive and for which there is no value in experimenting or learning a new way of working. A good example of this might be the check-out/check-in of code from a configuration management system. Once a good process has been defined, there is little point in doing it in any other way. Indeed, a lot of bad things might happen if people tried to circumvent the process. These processes always deal with 0OI or the simpler 1OI (for which the well-defined questions typically have a menu-driven answer selection). Here there is value in process.

Innovative

For those aspects of system development that are novel, the process must be intentionally sparse. Developers must be allowed to explore options free from restrictions that might constrain the solution. The developers are dealing with the remainder of their 1OI and also what they might be quite unaware of—their 2OI. Here there is value in explicit lack of process.

Process Transition

As systems development progresses, there can be a natural transition between processes. For example: when we start

testing a system, we do not (and cannot) know exactly what to test since to some extent we are looking for things we do not know are not there [3]. Much of the time we are seeking to expose those things we do not know about the system (like what it does do that it should not do). To design tests and test processes, we cannot be highly prescriptive since we do not know what we are looking for. We might have general indications: that tests should focus on predicate boundaries or cover representatives of all (known) input classes, but we cannot say exactly where we will find defects. This process requires opening up the process to the innovative creativity of the testers.

However, once tests have been created, run, and proved, testing can be transitioned to the usually highly prescriptive process we call "regression testing." Setting up an automated regression process before the knowledge is obtained is ineffective and it might force early testing into a high restrictive process mold that constrains testing to the point where it doesn't find what it needs to find.

Write, Test, Measure, Reduce

Good process focuses on the value it delivers. This depends on what has to be done: old or new? Repetitive or innovative? Restart the engines or land in the Hudson? Good process does not over-prescribe where that is not valuable. But there are other aspects of process definition that are often missed:

- **Test the Process:** in many decades of working in software I have rarely seen documented (i.e., on paper) actually tested to see if it works. Paper documented process is often written by people who do not actually use the process they are defining. Even more often these process writers themselves do not use a well-defined, tested and measured process—which is a little ironic. Commercial pilot checklists are written by a team of pilots, aircraft primes, engine manufacturers, and the FAA. They are written by people who use the process. Once the checklists are created they are tested in simulators and in the field to ensure they provide the value that is essential to keeping people safe.
- **Measure the Process:** software process is rarely *measured* to find out if it does, indeed, reduce defects, speed up the process, improve the lot of maintenance staff or any of the other attributes used as rationale for writing, using, and enforcing the process.
- **Reduce:** a further step is necessary and that is to *reduce* the process. As pilot checklists are tested and the effectiveness measured, much effort goes into making them more concise, more pertinent, more valuable, and smaller.

Prolific Process

This intentional and careful reduction of process does not occur in software development—quite the opposite. Once documented process is created, it tends to grow and grow as it attempts to deal with more and more different conditions, to identify more and more different situations, and to cover wider ranges of application. The documented process gets bigger and bigger, more and more complex, requiring more and more effort to read, to understand, and to apply. In doing so it becomes more and more unwieldy and less and less valuable and so less likely to be used at all.

Projects do not crash as spectacularly as the B-17 prototype. But they do crash. To bring them in to a safe landing, we need process that truly supports the business we are in; both the boring repetitive parts and the interesting innovative aspects of what we have to do. The process for each of these aspects should be designed for and support the true nature of the work; such process needs to be more focused and more concise, we should test it and measure it in operation to ensure it is really delivering value.

And we should make it smaller.

ABOUT THE AUTHORS

Phillip Armour is a Senior Consultant at Corvus International and a Principal Consultant at QSM Inc. He has over four decades of experience in software and systems development, was Master Instructor at Motorola University and on the external faculty of two graduate schools. He is the author of The Laws of Software Process (Auerbach 2003) and has penned the column "The Business of Software" at Communications of the ACM since 2000.

Phone: 847-438-1609
E-mail: armour@corvusintl.com

REFERENCES

1. Armour, P.G. The Laws of Software Process CRC Press LLC 2004. p.13
2. Ibid p.8
3. Armour. P.G. "The Unconscious Art of Software Testing" Communications of the ACM. Vol.48 No.1 January 2005