CHAPTER 2

Recipes

Software recipes appear in different shapes and forms, but they all share a common property. They advocate a subdivision of the software development process into separate predefined activities, to be performed by separate roles if not necessarily by separate people, at separate times. They assert that such predefined divisions of labor will yield tangible benefits in terms of quality and productivity.

> *For full disclosure, just in case it was not blatant enough yet, it is fair to say that I believe that recipes bring no value whatsoever. They allow people to feel important, they look nice on a resumé, but the overwhelming evidence shows that their application to software development is counter-productive. Recipes make projects late and more expensive.*
>
> *Among the recent projects failing because (or despite) of strong processes, Obamacare is a telling example. It involves 50 contractors, has cost fortunes, was delivered late and crippled with bugs. It was developed using a*

> *typical waterfall process, and if only because of that, the Agile community started howling, claiming that they would have made the project a success [Healthcare.gov failure].*
>
> *And when an Agile project fails like Universal Credit in Great-Britain [UniversalCredit] [NAO2013], even when the full report states that the lack of detailed blueprint – typical of Agile methodologies – was one of the factors that caused the failure, common Agile wisdom says it is because it was not applied properly, or should I say, not Agile enough.*
>
> *Right.*

Formalizing reproducible sequences of activities is a natural human trait, with a proven track record in many domains. Cooking is one. Knitting is another. A dramatic crash in a test flight for the Boeing 299 (a.k.a. The Flying Fortress) in 1935 led to the widespread adoption of checklists to be processed systematically before takeoff [Schamel2012]. In 2001, Peter Provonost reduced mortality when using catheters in hospitals by promoting a checklist made of five simple actions to take to avoid infection [Brody2008] [Pronovost-YouTube]. Atul Gawande [Gawande2011] comes from the medical world as well, and advocates the extension of checklists to other domains.

> *But checklists and recipes working for infections, airplanes and chocolate chip cookies does not imply that they will be just as efficient when applied to software development. Extrapolation does not work by wishful thinking alone.*

When applied to software, some recipes are more constraining than others. Some come with extensive provisions for adjustments and tweaks, and allow for some do-it-yourself on the job fitting. Such flexibilities are presented as addressing the flaws of more rigid methodologies. As explained later in this section, I take these circumvolutions as admissions of failure.

> *One's success is someone else's failure. It is just a matter of perspective.*

People misunderstand my point, and take my aversion to recipes for a plea to forego any effort at quality, just letting the code be written and hoping for the best.

Nothing could be farther from the truth. Software quality is one of major challenges of modern world, and requires all the bandwidth and attention we can devote to it. My disagreement with software recipes lies in the means to achieve quality. The overwhelming evidence shows that the process has only superficial effect on the quality of the resulting software. Quality must be built into the product, not into the process used to build the product.

> *The programming languages matter. The system's architecture matters. The testing strategy matters. Third party components such as databases and middleware matter. How you guarantee stability, flexibility, performance, and robustness matters. How you keep the number of moving parts under control matters. How you define your quality criteria matters (aligned on quality characteristics as defined by ISO 25010-2010 [SQuaRE], for instance).*
>
> *But how you force your development teams to fit into formal activities as defined by recipes does not matter. Or to add insult to injury, if and when it does matter, it is in a negative way, by being in the way rather than contributing positively to the project's outcome.*

Cost vs. value

> *Of the simplistic assumptions that there is such a thing as a free lunch*

Whether a software recipe contributes anything isn't the point. Any activity advocated by recipes can be useful in some adequately chosen context. Any formalized separation in phases can be shown to bring value in some cases. My claim is that when applied systematically as recipes should, they cost incommensurably more than whatever they deliver.

Part of this cost is tangible, with training, tools, time spent in meetings, dedicated head count and more. The intangible part is more pervasive. It comes from the loss of flexibility induced by any formal division of a task into sub-activities, and from the burden of checking for things just because they are on a checklist, without always realizing that they don't raise a flag more than once out of hundreds or thousands of cases.

> *To keep its staff alert, the TSA sends undercover agents with fake weapons and explosives through airport security to ensure they are detected; not the most reassuring of tests, given the abysmal results [TSA2015].*

The need for such frustrating activities must be measured by a rational cost/benefit analysis. If one considers their cost as nil as a matter of principle, such processes are always justified, and there is a perpetual case for adding more layers, more checks, more meetings, more reports. Who knows, they may actually bring something, which is always welcome if their cost is estimated at zero.

> *When setting up his catheter checklist [PronovostYouTube], Peter Provonost reduced an original set of dozens of issues to a mere five by focusing on the most effective ones, recognizing that such recipes' effectiveness goes down with size. A sobering filter that all process formalizations in the software field need badly.*

One may then wonder, what is so special about software development that makes it different from other industrial domains, where such formalizations of the process has been introduced fruitfully, areas where the cost has been measured to be offset by the gains?

The non-constant nature of software

Recipes make sense in a kitchen, where one tries to produce and reproduce the same dish as consistently as possible. A systematically mediocre restaurant is always preferred to a less predictable one with huge variations in quality, even if it allows for the occasional sparkle of genius. It is a domain where consistency of the user experience has a value for its own sake.

Setting up and improving the process used to build cars improves quality and reduces costs. The whole point of mass production is to produce identical items, in the most predictable and standardized way.

> *Arthur Hailey is known for having gathered extensive documentation for his novels about specific industries (airline, cars, energy, etc.). In "Wheels" [Hailey1971], he mentioned a fact that sounded just right, the kind of plausible thing that would come from such thorough preliminary research. He claimed that cars built on Mondays and Fridays (just after or in anticipation of a festive weekend) had more flaws than the ones built on the other days of the week. Knowing this, executives in the car industry made sure that the cars they bought for themselves were not built on Mondays or Fridays.*
>
> *So much for mass-production determinism.*
>
> *"Wheels" was written almost 45 years go. Things may have changed a lot in the time since. It may even not have been true at the time, it is a work of fiction after all. True or false, this fact made a lasting impression on many readers, as shown by the number of posts asking for confirmation or rebuttal for this piece of trivia on fact checking web sites.*

The same goes for weapons, airplanes, shoes, trampolines, condoms, light bulbs and more. Anything that is built in quantity, and where reproducibility, consistency and predictability are the rules of the game.

But software is different. Even when people talk about software factories, industrialized development processes and other similarly impressive metaphors, software is not mass produced in the way chocolate chip cookies and Toyotas are. Because intrinsically, organically, every piece of software we build is different from the one that was built before.

It differs functionally. It also differs technically: the environment, the hardware power, the software infrastructure we build on, all these aspects evolve continuously. The software we build today differs from that of the team next door, or from what the same team developed last year.

> *Some redevelopment projects reproduce the functionality of an existing system, but are then motivated by radical changes to the technical environment.*
>
> *In mission-critical systems, diversification is a feature in its own right. The same software is developed multiple times by independent teams using different techniques. The results delivered by the various implementations are compared for consistency, and a voting system is used to select the preferred answer in case of conflicting results. If two or more systems, as different as can be, do agree on a result, the odds of it being correct are much higher. In such cases, while the functionality is the same (the test cases may be common as well), everything else will be made as different as possible: architecture, programming language, endianness for numeric values, sometimes even the hardware platform.*

This intrinsic non-reproducibility weakens the case for formalized processes, which are now reduced to vague guidelines that must apply to large number of different projects, with different focuses, different concerns, different quality requirements, different technical environments, etc.

In fact, when considering the variation factors among software projects, it is hard to believe that anyone in her right mind would ever consider formalizing its development process.

Iterative recipes

The final blow to software methodologies of all shapes comes from iteration.

In the early days, the recipe mantra was all about dividing the software development process into steps that had to be followed sequentially. This simplistic division of labor was nicknamed "waterfall"[1], reflecting on the mathematically monotonous aspect of the process: one did not go back to a phase after it was closed and labeled as completed.

But reality is stubborn. This division of labor into phases made some sense from a theoretical point of view, but it soon appeared not to work in practice. Even when all the checks and validations had ensured that a phase was to be considered as completed, its deliverables had to be revisited, questioned, altered and reworked.

The first reaction was to add validation layers to really ensure that intermediate deliverables and activities were final, that one had not overlooked anything of importance and that it was time to move on. It only made the process even heavier than it already was, without making much of a dent in the need to review past deliverables.

> *The teams that were committed to waterfall methodologies ended up developing the software anarchically under the radar, and faked the process by providing the expected intermediate deliverables without following the corresponding process, which was, for all practical purposes, a fiction.*

The waterfall model just does not work.

Even the most stubborn of the methodologists can only face damning evidence for so long. At some point, the purely sequential model got replaced by iterations, where sequences of phases are restarted periodically as the project advances. Phases are thus never final. They

[1] Also referred to as V-shaped process, to express that one first goes down from abstract specification to the code, then goes up ensure that the deliverables match the requirement at increasing levels of abstraction.

can always be questioned, and amended to reflect evolving specifications, more maturity in the understanding of the domain, ungracious performance figures or any other change in the surrounding environment.

Iterations in the development process are the acceptance that one cannot develop half-decent software without reconsidering, refactoring, redesigning as the project advances. The waterfall is a delusion. Completion is uncertain. The past must be questioned, over and over again.

But iterations defeat the purpose of methodologies. How much of a methodology is it still? How much of a measure of completion does it allow for? How much of a structure does it put on software development? How more manageable and more measurable does the resulting process become?

Iterations are not an advance in the state of the art. They are a concession of defeat. They are an admission that formalizing the software development process is pointless.

> *Would we imagine building a car iteratively, taking it through the mounting chain over and over again until it is considered adequately completed? What about cooking? Or even building a house? Would it be acceptable to reconsider the foundations as part of an iteration?*
>
> *Oddly enough, the only elaborative − as opposed to reactive − process I can envision where iteration is accepted (and even then, with limits) is the legislative design of new laws, where projects go back and forth between chambers until an agreement is reached.*
>
> *There is probably something juicy in this analogy. Let me think it over and get back to you.*

Claiming success by lowering expectations

Foregoing V-shaped processes for iterations is not the only way methodologists lower expectations. They also do so by asking for what methods and recipes are not about, namely nuance.

None of the arguments I am making in this section is entirely original. These points have been made repeatedly over the years, and the answer of the methodologists has always been the same: recipes are not to be used dogmatically. They should be applied with caution and adapted to local circumstances.

Ironically, even when the methodology aficionados recant their most bombastic claims, I beg to differ. The whole point of a recipe is to be used dogmatically. Follow it blindly, and you'll reap the benefits. It should be that simple. If you require the practitioner to exercise nuanced judgment, you essentially give up on the methodology. You admit that software development it is not to be industrialized. You are admitting defeat. You are deliberately lowering the expectations of what you are advocating to be able to claim success.

But as victories go, this one is Pyrrhic.