Too Much

Martin Hunter, May 2019
https://www.continuity.kiwi.nz

In working with "agile" projects I find that sensible discussion, refinement and agreement on the vagaries implicit in defined agile frameworks are not often prioritised, or even had at all, which leads to impaired and sometimes contradictory understandings between team members. One such discussion surrounds what constitutes "too much" analysis. Often the term "too much" is wheeled out to counter concerns that "not enough" analysis is being performed by a team ahead of entangling themselves in the complex and costly activity of programming software. "Too much" is given credence by a fear-driven cultural response to troubled IT projects of the past that rejects analysis as a signal that you are engaged in a "waterfall" project; as if waterfall is implicitly bad and guaranteed to fail. "Too much" is rationalised by a secondary principle from agile frameworks regarding flexibility to change, which is enshrined for programming and software design in the concept of "refactoring". The ability to refactor, it is argued, means that you don't need to do "too much" analysis, and indeed design, before you start coding. In fact, the opposite is argued: Refactoring gives license to the ability to learn from the real-world of programming such that it is ideal **not** to do "too much" analysis or design before hand. The argument about "too much" has religious or evangelical qualities; challenges to those that argue against a statement of "too much" are sometimes treated as blasphemers, as if more belief or faith in the "agile way" is the trouble here.

Of course, whether "too much" analysis is undertaken is not the problem for teams, rather when and who the analysis is done by, the quality of that analysis, the quality of its results, how its results are communicated, actions agreed and how you respond to unforeseen issues that arise at any point in a software delivery lifecycle (which is virtually inevitable). Whether analysis is "too much" is a **qualitative** judgement, not a quantitive one, despite its feel as referencing some sort of knowable metric. It is therefore the subject of professional opinion, not mathematics. To avoid engaging in fruitless religious warfare (or much worse, avoidance of discussion) it is important that the team acknowledge this and together construct an "independent framework" upon which they can reliably judge whether analysis is indeed "too much" or not. This requires the sometimes personally uncomfortable activities of robust discussion, thinking, conflict resolution, compromise and agreement. In many teams, there are by accident many such independent frameworks based on the individual experiences of practitioners, which is of course where judgement conflicts originate. Teams need to embrace these and nut-out a clear analytical framework that they can together rely on. Of course, whilst it is important to invest in this at the beginning of any endeavour, if the team finds it to be unhelpful or inoperative in anyway during the course of that endeavour, the "independent framework" must of course be tweaked and changed as they progress.

Many principles from the agile manifesto directly address the issue of responding to change, embracing change and actively working with change rather than resisting it. This is enshrined in the value "responding to change over following a plan" but is implicit in other values and principles too. The principles acknowledge that requirements evolve, especially as you find out more about a given problem domain and solution, and that finding out more is predicated on actually getting into programming software and not just thinking about it. This is partly because feedback from customers and end-users is of utmost importance in the software development cycle and by its very nature software, certainly before it exists and even afterwards, is highly intangible and difficult to understand and conceptualise. If projects of history have taught us anything, it is to seek early feedback from customers and to incorporate it. However, there is nothing here that says you shouldn't think deeply about a problem and solution before you embark on coding a software program, just that you need to expect that your thoughts about the problem and optimal solution will change as you progress through coding. In fact the word "change" implies that you have at least some known state that you are changing *from* and quite possibly that you know the intended state that you are moving *towards* as well. Analysis is simply the act of collectively or individually thinking about a problem and/or solution, with the best analysis being undertaken in a structured way that allows hidden avenues and concerns to be identified, explored and addressed methodically. Coding software involves a natural tension because coding is expensive to create and very expensive to change afterwards (regardless of what the refactoring evangelicals tell you) whereas thinking is not that expensive and much easier to change in comparison. However thinking alone does not subject you to technical realities nor address the early acquisition of quality feedback from customers very well. Successful software development is therefore about striking a balance between these two countervailing forces. This is the message that is implicit in the agile manifesto.

Quite sensibly, the agile manifesto and the frameworks that have both fed and stem from it decline to comment on exactly what the optimal balance is. This is because they recognise that there is no one-size-fits-all answer to every endeavour. Endeavours in software development have multiple dimensions that all factor into the balance; people (personality, soft and hard skills, power relationships, experience, history), technology (constraints, opportunities, limitations), architectural context (standards, stakeholders, strategies, regulations, legislations), finances (budget, resources, procurement rules) and politics (internal, external, explicit, implicit) to name a few. However, there is one standard that does transcend all of these factors in terms of striking a balance for success; a standard that the agile manifesto somewhat takes for granted: The software development value chain. This is identified Ivar Jacobson's "Essence Kernel" and in the TOG's "IT for IT" framework that describes this as one of the value chains of IT. TOG call it the "Requirements to Deployment" value chain. This value chain is not however a new concept and in fact is core to the vast majority of software development methodologies that have emerged in the last 50 years. The chain, in terms of activities, is simply: Analysis -> Design -> Programming -> Testing -> Deploying -> Operating. History has taught us that this fundamental chain cannot be circumvented by any methodology or framework, no matter how an endeavour is organised or managed nor the wishful thinking of management or consultants.

Analysis, then, is a fundamental element of the software development lifecycle. As postulated earlier, the problem teams encounter is not what is "too much" or "too little" analysis, but who does it, when, where and how. Clearly, it is important that the team has the skills — soft and hard — to perform analysis. Agile frameworks don't say anything about not doing analysis and equally they don't say anything about not having analysis skills on the team (quite the opposite, in fact). The same goes for the team's performance of the other activities of the value chain. Hypothetically there are three ways that a team can organise the skills required for each value-chain activity: 1) Each team member can be equally skilled in all activities and able to perform each equally (a "vertical" alignment of the skills matrix to people); or 2) different team members can specialise in different activities in the value chain (a "horizontal" alignment of the skills matrix to people); or 3) the team could organise some mixture of some people aligned horizontally and some vertically (e.g. a team of generic "doers" with specialist leads in each value-chain activity). All three are logically plausible team structures, however those at the extremes (i.e. 1 and 2) are unlikely to be able to exist in reality: Teams are made up of people who have varied soft and hard skills. Some people are better at analysis than others. Some people communicate better than others. Some people focus on detail better than others. Some people enjoy technology and others don't. Some like finding problems and others solutions. These variances in personal traits form a real-life skills profile that favours some individuals for some activities in the value chain over others. The history of IT in the last 50 years has been a history of gradual specialisation into roles aligned with the value-chain activities precisely because of these variations in personal attributes and, by virtue of efficiency in the division of labour, a positive reinforcement from economic imperative. It is arguable that IT projects that have rigidly enforced type 2) team organisation have suffered as a result and the same is true for type 1). Due to the reality of varied people having varied skills, the subsequent growth of specialisation and, following this, training and enshrinement in job descriptions, the IT industry today finds itself in a position where the industry's actual skills profile is more aligned with type 2) than type 1). It wasn't always so, but it is today. Religiosity — adherence to a philosophical faith — in agile teams has the dangerous potential to attempt to force a team to one extreme or the other rather than working towards a pragmatic balance that takes account of actual people and their actual skills. By enshrining certain roles with authority to decide (e.g. Product Owner) agile frameworks have, in unintended ways, created an expectation that the people filling those roles "know best" regardless of their personal attributes and what analysis performed by people in the team (who may be more highly skilled at analysis) have established. In the face of perceived or real push-back and decision-override by those who "know best", adequate analysis can wither and die in a team which then defaults to slavishly delivering what those that "know best" want, without fulfilling the team's obligation to ensure that the value-chain is completed in the best way it can. This is the polar opposite of historical project failures, but is nonetheless also a failure. The danger of the “too much” statement is that it gives legs to primacy of the “knows best” no-questions-asked decision authority by disempowering analytically-minded team members to challenge and refute on the basis of analysis (i.e. simply asking questions and thinking things through) and fomenting blanket distrust in the usefulness of analysis output relative to issues that arise in coding, even though those issues may be easily foreseen and avoided by adequate analysis. The danger for the agile movement as a whole is that it is being misrepresented to senior decision-makers by this and other such dynamics that have no basis in the agile manifesto or frameworks but are the mere product of people being people. In recent years the number of CIOs reporting that they think agile is a “fad” has increased, for example, to now more than half. As reported in CIO magazine this sentiment has come also with a desire to “bring back” analysts and architects into IT teams (https://www.computerweekly.com/news/450418205/Agile-development-an-IT-fad-that-risks-iterative-failure). The irony is that “agile” has never advocated that adequate analysis and architecture skills and activities should ever have left in the first place.