Management Pitfalls and common solutions final split

Part 1

The goal of this article is to analyze common management pitfalls that one can find themselves in, and to offer possible solutions or approaches on how to best avoid them while providing the highest return of investment and value. First there is a short description of the current state and the problem it is facing or trying to solve, after which follows a short management insight on how to approach it, managing the risks and providing action points and best practices that can be followed.

The Slipping Deadline (what causes it and how to mitigate it)

Sometimes a deadline can start slipping through our hands due to not managing the risks properly. The highest probability and impact risk is always… people. Even if we put all the right processes and procedures in places, people are unpredictable and not managing this properly can wreak havoc to a time sensitive product. Usually, if the team is small, the developers are not easily replaced or filled in in their absence, or there are simply not enough resources to cover for them. This risk is called “overlapping time off”. Another risk in managing a slipping deadline is how to balance between the time we have, the work that needs to be done, and the resources that we have, without compromising the product quality. If we have a fixed deadline and we have planned our work accordingly, we put some buffer in place, yet we are falling behind, we need to balance this out.

Solution: Overlapping time off can be easily solved. This problem has happened so many times that many proposed solutions have been proven effective. The easiest one is better communication between the team members and knowing things slightly in advance. Maybe during the Sprint Planning session, discussions can be opened for the current iteration or the next two, for if anyone is planning any time off in this period. Also, if some roles are covering each other, those roles should be encouraged to plan their time off together, so they can cover their scope of work without any impact on the delivery plan. Some examples: the Scrum Master role can be briefly covered by the Team Lead; two senior developers working in pairs on a same feature should not leave at the same time without having someone who can cover for them.

We can always play with the triangle1 (perhaps better depicted as a pyramid) of project management. The PM can try to move the deadline; the scope can be reduced (usually the preferred option); the quality should not be reduced because long-term this will cause many problems and will leave a lot of technical debt; some teams tend to skip writing unit-tests perhaps, but again, this will just extend the fragility and rigidity of the application, as Robert Martin would argue in his book Clean Architecture2. Following the Project Pyramid, we can play with Budget, Scope, and Time, some argue a Quality as the fourth corner or the central area between the tension of these three. So, at the end I believe we are left with deploying an MVP solution that can satisfy the market needs at the beginning and continue developing the product by further prioritizing the needs of the end users.

We also need to know before we start a project what we are going to measure, and we need to agree and commit as a team that this is our main metric. Teams usually measure velocity, but that is a high-level metric. What is causing that metric and what does it mean? Each team individually, the velocity can have a relative steady number that the team aims for for each iteration. But what does this mean to a client? We introduce something called “signals”. Each metric has signals that cause it. For the velocity metric we can use several signals, but best practices and experience dictates that there are typically three that are of high interest:

- Work In Progress (WIP). This is also a limitation, a constraint that the team needs to agree upon. Having too many items in the “WIP” state tends to defocus the sprint goal and disorient the team. Putting a hard limit can be done using the issue tracking system, but any mature team can know not to begin working on multiple items that have been committed to the sprint simultaneously.
- **Time In Progress (TIP).** This is an indicator of a slow or stuck ticket. These tickets should be our highest concern since they pose the highest risk. A ticket signaling this should be escalated and elevated to a blocker

for the certain sprint, and sometimes more developers or even the whole team should stop what they are doing and address this issue.

- **Throughput.** This is our completion rate, how much it goes in against how much it comes out, i.e., how much has been processed. This signal can detect and alarm us early on when we have a scope creep, or we deliver more than we committed. Sometimes this is not an indication of a creep, rather a lack of proper planning at the beginning of the sprint. Scrum allows us to start a sprint without having a complete picture of what it contains, granted that it always follows our sprint goal. Adding items that do follow the sprint goal, do not belong to the sprint, therefore are creeping our scope.

The Guesstimate (story points)

noun

/ˈgɛstɪmət/

an estimate based on a mixture of guesswork and calculation.

Is estimating necessary and if so, why do we estimate in story points? Why don't we just go along with the requirements, designs, implementation, testing and deployment and see how long it takes us, after all, rushing will not provide a high-quality solution. But management always insists on estimating, and since man-days was the antipattern of the 20th century, don't we now introduce the antipattern of the 21st, the trending story points. Is this really the case, and can we go by without estimations, or as the developers like to call them – guesstimations, since nobody knows what it really needs to be done and how long will it take at the end. The endless war between the implementers and the managers. Why don't we just translate man-days into now more flexible Fibonacci scaled numbering and be done with it. The managers will be happy, the client will be happy, and the developers will be relieved.

What about estimating niche cases that were previously unknown or we don't have experience estimating, ex. Documentation. How can we estimate how long it will take us to document something that we are not very familiar with, and we don't know what type of documentation it will be needed? In that case, isn't it better if we just timebox the task and say "you will get 5 story points of documentation" if that is what the time or budget allows?

Solution: The benefits story points offer as pure measure of size are compelling. That story points help promote cross-functional team behavior is a huge advantage. Shifting a team's thinking from "my part will take three ideal days, and your part will take two ideal days, so the total is five ideal days" is very different from "overall, this story seems about the same size as that one, so let us call it five story points also." That a story point to me can be the same as a story point to you, while the same may not be true of an ideal day, is another big benefit. Two developers of different skill or experience can agree on the size of something while disagreeing about how long it will take to do. The shortcomings of story points are indeed short. Yes, it is easier to get started with ideal days. However, the discomfort of working with nebulous story points is short-lived. Ideal days are easier to explain to those outside the team, but we probably should not choose based on how hard it will be to explain to outsiders.

Of course, that we guess at some point or another, we must start with guesses, and as time goes on and we get a better product knowledge, the guessing game is moving into an actual estimation. We define a "baseline" that everyone is comfortable with, a task or a story that everyone agrees immediately that falls into the category of 2 story points. Of course, no two tasks are the same and even compared to this one, the one we are now trying to estimate is perhaps 2.2 relative. Okay, so what about if it is 2.5 or above, do we round it up or round it down? The main goal of working agile is to speed up, but naturally not forcefully. If we try to bloat up story points, we are only making a smokescreen and we are not actually going faster. So, the desirable approach is to round it down to free up capacity to take more work. Maybe at the start the team will be skeptical and scared to do so, but it takes some courage to be continuously better. That's why we are doing agile, it promotes this thinking and encourages it, so let's exploit it. So now, not only are we trying to get a higher velocity each sprint, we are also maximizing the available capacity, as Jeff Sutherlands argues, twice the value for half the time3, but now even more daring, 4 times the value for half the time.

Estimating fringe cases should follow the same rules. Even if we are working on an issue rarely, having the experience and knowledge for such cases would benefit the team, and those can be obtained only through

practice. One should never resort to estimation regression, or back to man-days estimations. Also, if the time budget allows, we should avoid “n story points worth of feature”. Once down this road it’s hard to get off from. Next thing you know, we start to put “n story worth of effort” on every complex thing that we would like to avoid in any way possible. I tried my best. It is not a way that we write down our sprint goals, rather we reach milestones, finish epics, and deploy features that are an “usable increment” each sprint as defined by the scrum guide.

As a finishing thought, estimations are a bridge between the development team and the product team. This is the language for communicating or translating requirements and designs to tasks, features, and epics, creating roadmaps and delivery plans, within a reasonable allotted time and budget (again the project pyramid creeps in). This is the way that both sides can commit to a joint effort, through transparency and trust, forming strong bonds of collaboration that can result in lengthy and worthy partnerships, for the endgame of a continuous raise of technological, ethical and humanitarian standards.

[1] https://en.wikipedia.org/wiki/Project_management_triangle

[2] https://www.goodreads.com/en/book/show/18043011-clean-architecture

[3] https://www.goodreads.com/en/book/show/19288230-scrum

Part 2

In the second part of the project management pitfall series we continue with some practical examples of how we can manage certain sensitive situations using some of the tried and tested methods. The approaches or best practices suggested here are a funnel of several great minds in IT and project management, such as Frederick Brooks, James Coplien, Robert C. Martin, Jeff Sutherland and the likes. This in no way is an exhaustive list of advice or the best possible approaches for any management predicament. As Mr. Brooks says in his later addition to the Mythical Man-month book,

"There is no single development, in either technology or management technique, which by itself promises even one order of magnitude [tenfold] improvement within a decade in productivity, in reliability, in simplicity."

The Joint Effort (how to structure client-vendor collaboration)

A company is developing a product for which they know the business rules that are governing it. This product is the focus of the company, and they don’t have other ventures that bring back any financial return. They are looking to improve the overall user’s experience and client satisfaction by providing a “fresh” look and feels to their flagship application. The new approach is a joint venture between a development team from a contracting vendor implementing the solution following the Scrum framework and a product team from the client’s side with an idea to be consisted of a Scrum Master, Product Owner, and QA Engineer for analyzing and defining product requirement, designs, and solution direction.

Solution: Putting the roles of Scrum Master, Product Owner and QA engineer on the client’s side gives an insight that the client wants a better control over the development effort, but the lack of an engineering team puts the pendulum towards Business Requirements and less toward technical feasibility. A strong consultation between the client’s team and the vendor’s team should be put in place to reduce this discrepancy. We can also argue that the SM needs to be from the vendor's side because the SM is dedicated to the team’s effort, thus a better chance to bond and perform is when they share the same “predicament”, and they can safely and openly communicate about impediments and work progress. We can also praise the approach here for the SM and PO role to be given to two people. The SM role is focused on protecting the team and working for the team’s benefit, while the PO role is in a way in a constant attack stance towards the team, demanding more work, faster delivery, and better quality for a lower price. These two roles are always in tension, and placing them both from a single reference point, i.e., the clients, can lead to lower satisfaction and performance by the team.

The Inspection (communicating unsatisfactionary code)

There are many code inspecting platforms that run through the codebase of the application and return a rating. That rating is usually measured as the quality done on a static analysis of the code. However, there are even

programs that go deeper and inspect the code complexity, fragility, rigidity, and immobility, defined as the package principles by Robert C. Martin in his book “Agile Software Development, Principles, Patterns, and Practices”1. Why this is bad is because it shows an antipattern called “code rot”. It is bad if it is an inhouse project, but it is a price to pay if it belongs to the client. Having a low rating code means that we are approaching a decline in productivity as we continue to increase the codebase, because it slowly becomes unmanageable even for its original creators. And what about if it needs to move to another vendor? There is a limit to how tangled a code anybody will accept, and the inspection platforms warn us of the risk of taking ownership of such a spaghetti codebase. And experience shows that asking the Software Architect responsible for this mess to create a documentation for the code will result in a continuous livelock situation – when (or even if!) the documentation is done, the code has grown again that it became even more entangled, or in a different way than the first time, so the documentation is useless.

Solution: Communicating a low-quality code to a development team is always bad news. This means that the team structure or the team members' experience is not at the expected level. Sometimes this review can break the spirit of the team but done properly it might lift it and motivate the team to try harder. The team needs to understand the specifics of the inspection to provide a better-quality product. A PM who has a technical background can better translate this into actions. It is not only to communicate and leave it at that, but a plan must be prepared on how to tackle the feedback.

Communication is a delicate thing. First a deeper understanding and analysis of the results needs to be done. This can be made in collaboration with the Team Lead to begin with, sharing this information with a limited audience until we can find the root cause of the problem. After a firm ground for the feedback has been established, meaning that the feedback is properly analyzed and all aspects of the low-quality code tackled, the feedback can be shared with the team. This way, it is not only demoralizing, but provides encouragement that we as a team accept the problem and want to address it – collective ownership. We need to motivate everyone to work harder and better, but only by example. We can explore several options on how to increase the quality. We can try the XP methodology. We can attend classes and certification paths; we can even start a book club studying together and unearthing better approaches to higher standards and quality. Everyone can work towards the common goal of the team, specializing in their area of expertise.

There is always one approach to hire consultants, or even better, senior staff members for the team to work with and learn from if the budget allows. We need to communicate this to the client and explain the reasons behind the low-quality code. Perhaps, providing a larger buffer for the project can increase the quality. Communicating transparently the needs of the team with the client, the results from the analysis and a plan of action can result in a joint effort to achieve higher quality and improved trust – after all we are together in this one. However, the best practice dictates that sticking with the current team and raising their knowledge and standards is a better way then starting from scratch. Starting a green field project does not guarantee the results in the first try. It is a pattern called “The Second System Effect” stating that the first product has a lot of issues, that you try to solve each and single one of them in the second product, making it so heavy and sluggish that it fails also. Only reaching the third attempt you understand how to build the most streamlined version of the product – as argued by Frederick Brooks in his timeless classic “The Mythical Man-Month”2. But it remains to be seen such a thing, a client who can approve, finance, and wait for the third version. The sad truth is that if you put the original Tiger Team from the first product in any subsequent attempts, they will keep rebuilding the same thing repeatedly. That does not mean that the team should be completely disbanded, but a fresh set of eyes with a higher experience can motivate the team and bring them to higher satisfaction of the result.

The Handing Over

A project is being handed over by a Project Manager who is swamped with work and has accepted a temporary replacement role of a delivery manager for an ongoing effort; he can provide only limited information, that currently everything is ok with the project, except some minor bumps, such as someone from the previous management team has left the company few weeks back and currently the scrum ceremonies and release plannings are done by the development lead. Some people are complaining that the Scrum framework is not followed as it was before, and the meetings are leaking over the timebox due to incomplete state of the requirements in the backlog.

Solution: The handing over by the PM who was responsible at the start was not done properly. A good advice is for the new Project Manager to have a chance to see how the previous, even the temporary, manager, works with the team, to “shadow” him or her for a while, to learn the spirit and the delivery cadence of the team and to better analyze their strong and weak points as a bystander. Knowledge transfer sessions should be separately scheduled between the two of them to transfer any business knowledge and detailed state of the project or product, that can be further explored in detail, on a need to base.

Since the previous manager, or we can deduce since it was the scrum that suffered the most, the Scrum Master left the company and was replaced by a technical person, having in mind that the technical team is from the vendor side, we can assume that at least a step in the right direction was held, i.e., to the common pressure for the client to take over the PM role was not yielded. However, working both as an implementer and a manager, one role will suffer, most often the less valued one by subjective point of view, often the management role in this specific case, given the tendency for technical people to find satisfaction within the development effort. However, if the technical person is solely dedicated to the management role, this can be one of the best-case scenarios for the team, since the manager can also understand the predicament of the development team better, escalate in right direction to monitor, and manage the risk of any arising impediments having an insight into the implementation work.

Unfortunately, we don’t have that luxury here. As expected, the management role is suffering mostly by a lack of knowledge in agile methodologies, frameworks, and best practices. This should be communicated with the new delivery manager taking over and be clear that if the technical person wishes to continue working as a Scrum Master, a better education on the role must happen. A first step would be to better equip the person to handle the Scrum theory by passing the PSM1 or even PSM2 certificate, thus obtaining proper base knowledge for the role. However, most often this is not enough, since covering the bases brings you only halfway there. A further interest in Scrum and the Agile world needs to be expressed by the person proactively, followed by engagement in self-studying in such areas as i.e., release management, delivery management, roadmap elicitation and prioritization, etc.

To provide some ideas on how to attack the quality of the work done by the PO or the BA, is to better specify a ticket following an industry standard for defining the stories, tasks, and bugs. A lot has been said on this topic; my personal choice is the BDD3 (Behavioral Driven Development) using a short User Story description, followed by Gherkins scenarios or Use Cases, Acceptance Criteria or Tests and a mockup or design proposal. To keep it complete, we should stick to satisfying the “Definition of Ready” defined by the team for when a ticket is considered finished in the specification phase and can be moved in “To Do” state beginning with implementation and moving on following the agreed “Definition of Done” by the team.

The Complaint (communicating with people, FE and DevOps practices)

A team can be found in many operational states; the usual ones are forming, storming, norming and performing. Each state has its challenges, but most of them can be attributed to the storming phase. Here the team shows the “Five Dysfunctions of a Team”4: Absence of trust, Fear of conflict, Lack of commitment, Avoidance of accountability, Inattention to results. Usually, the finger pointing goes through the manager at this state and usually they are addressed to the weakest link of the chain. Most of the cases involve engineers working in isolation from the rest of the team, such as QA, DevOps, or to best blame a functionality on, the Front-End engineer. The complaints can come in many forms, but they usually find their way as a written communique; it is easier to hide the true emotions and to break through the defense.

Solution: The team conclusion from the low quality of the product is focused on the work of the DevOps engineer. From personal experience, I have worked with great DevOps practices in place, such as provided by the “Accelerate”5 study. I’ve also worked with a great team that had no DevOps practices in place, they did everything manually, since that was the nature of the application and placing DevOps would open unnecessary burdens to the team and prolong the deployment cycle even further. The first team, even though they had, I can argue, a perfect CI/CD in place, had a terrible throughput, while the second team yielded much greater results much faster. DevOps is not a silver bullet, as argued in the book. Good research has to be done whether DevOps including CloudOps is a way to go in a certain case, ex. Do we plan to develop and maintain this product for a longer period (is it an

engineering effort?), or do we plan to try out an idea as an MVP that we can perhaps later include a CI/CD/CD pipeline (a programming effort?).

Sometimes the client can provide their feedback about what they see as a possible root cause of a problem. Usually, they give this feedback through an email without any prior investigation and it shouldn't be acted on immediately. First, we need to further address this, perhaps on a call as a better approach in this situation if that is possible, to get a better insight and more information why this is their conclusion. Maybe they lack transparancy into the product development progress or they are lacking information radiators (see how to write reports and emails) or we are not having regular reviews and demos and status meetings between managers).

Let's say that they usually see the result of an effort and they jump the gun that it has not been implemented as expected, so the logical conclusion is that the front-end has not been according to specification. This conclusion that the FE developer is the chief problem can be challenged. How did they reach this conclusion? Who reviewed the FE developer's work? Are they perhaps unsatisfied with the UX design instead of the FE work? Who provided this information and why? We are missing crucial information that can dictate the course of action. We need to find the root cause of the problem taking every branch to its origin; perhaps this will lead back to a conclusion that it was wrongly specified or there were many ambiguities at the start that were not addressed so the team did the best they could, given what they knew.

Assuming that the client is correct, and we should always investigate this option no matter how we feel about it, we should schedule a call with the FE developer and see if this is really the case, inspect the things where they are pointed at. We should also talk with the Team Lead before the call and perhaps one or two of the FE developers' peers, to collect more background information from several sources, so that they are not biased. We should also consult the QA engineer who has been testing out his work in the past and collect their insight as well.

Finding that the accusation is correct, we need to plan our next course of action – communicate this with the FE developer face to face, carefully, and try to find out what is the cause behind this. Perhaps there is some lack of knowledge; or perhaps there are some problems of personal nature. Maybe they did not find their right place in the team, or their work is not appreciated. Anyhow, finding that we can't solve the problem immediately, i.e., the quality of the FE can't be improved, we can try to look for another FE developer with better experience as an addition to the team or as a mentor. This should be communicated with the current FE developer also, for openness and fairness and transparency are the chief pillars of Agile, so they will know their options. Perhaps having a second FE developer in the team can motivate the current one to provide higher quality, thus keeping both developers, if not in the same team, perhaps in the same company.

Similar approach can be taken for the DevOps work. Also, throwing away work should never be our first option. However, if the work of the DevOps engineer proves to be really causing more harm than good, building on the existing work can slow things down even further. If no knowledge of how to improve the work of the DevOps engineer is present, there are again several options on how to proceed. Maybe external consultancy can be used in addition to better educate the engineer on proper techniques and practices, or to "patch things up" for the time being. However, a short-term solution to work without DevOps practices, will reduce the deliverability of the team greatly. Hiring a more experienced engineer in this position, or a DevOps team lead can increase the quality of these roles for all teams and can put higher standards and solutions into practice for a long-term benefit. But we should be mindful of Brooks's law: "Adding manpower to a late software project makes it later".

[1]https://www.goodreads.com/book/show/84985.Agile_Software_Development_Principles_Patterns_and_Practices

[2] https://en.wikipedia.org/wiki/The_Mythical_Man-Month#The_second-system_effect

[3] https://en.wikipedia.org/wiki/Behavior-driven_development

[4] https://en.wikipedia.org/wiki/The_Five_Dysfunctions_of_a_Team

[5] https://en.wikipedia.org/wiki/Accelerate_(book)

[6] https://abseil.io/resources/swe-book/html/ch01.html