virtual.blue

How (not) to run a software development project

The top 10 most common noob mistakes - so you can avoid them.

Tom Gracey,
Director at Virtual Blue

Q Who is this list for?

A *Anyone setting out to run a software project, including:*

- *individuals taking the plunge with their first app*
- *bootstrapped and/or investor-funded startups*
- *new project managers taking the reins in an established company*
- *anyone else looking for more insight and tips for managing a software project*

Q Why do I need this? I know what I'm doing!

A *Are you sure? Evidence suggests you probably *think* you know what you are doing - while the most likely reality is that you actually don't!*

Ouch! Did that hurt? But I'm not trying to offend. It's just a simple statement of fact, based on the following 2 observations:

1. The vast majority of software development projects don't even make it to market, let alone generate any revenue. You can verify this fact for yourself by searching for something like "percentage of software development projects that fail". And when you hit on a Forbes article claiming that number is 85%, just remember the business media generally refers to well-funded projects built by established companies. In my opinion if we include shoestring-budget projects and bootstrapped startups that figure shoots into the upper nineties. We really are talking about almost all projects. Based on statistics alone, your project is almost certain to fail. The first step in defying these odds is accepting this is the position you are starting from.

2. As an independent contractor often invited onto stalled or partially finished projects, I have personally witnessed these failures firsthand. Time and time again. Over many years. I can tell you that the vast majority of failures are for *exactly* the same reasons. The same mistakes repeated in an endless loop. Software development Groundhog Day. And in every single one of those cases, the project leader thought they knew what they were doing. (If they didn't think they knew what they were doing, they might have done some research - and perhaps not made those exact same errors.)

Q You got me! So, what's on the list?

A *Read on to find out!*

Complete guide

The top 10 most common noob mistakes - so you can avoid them.

I have carefully arranged my list in the approximate time order the mistake is likely to occur (and not by seriousness). If you are making a mistake toward the bottom of the list you probably have already avoided the majority of the ones near the top. To rank them in terms of seriousness would be more difficult, since the truth is any one of these errors has the potential to derail the whole project.

The mistakes towards the top of the list are therefore likely more *common*, since you can't make a further mistake if your project has already collapsed. Indeed, I would say that mistake no. 1 is by far the most prevalent, and usually stops the project dead on day one. However, do remember this is all just a rough rule of thumb; for example, it is perfectly possible for an outfit to have put together a clearly defined spec (no. 4) while still vastly underestimating development costs (no. 1) - and I have witnessed this exact scenario.

Note that careful deliberation of what items to include meant also deciding what items to leave off. Yes, there are pitfalls not listed. This is the "top 10"! Don't think for a moment that if you avoid all the items on this list, your project will automatically succeed. (If you want to know the rest of them, you can always hire us!).

1 | Vastly underestimating development costs

Software development seems to occupy a special position on the engineering spectrum. For a start, unlike disciplines such as mechanical or electrical engineering, it is frequently forgotten that software development is engineering at all. One reason for this may be that it is perhaps the most mysterious of the engineering disciplines; very physical projects such building cars or bridges are easy for the layman to visualise, but the guts of the software systems are almost always completely hidden from the user.

It's common for the naive software system user to mistake ease of use ("You just click the button!") for ease of development - when in truth the two are polar opposites. The easier a system is to use, the more it is doing on the user's behalf, and the more effort needed to go into creating it. It is common to hear the word "just" from newcomers describing their expected feature list: "It's just two-factor authentication", "It's just geofencing" etc. When I hear this kind of talk, I think to myself, "If you think it's that easy, you build it!".

The consequence of this misunderstanding is an overall under-appreciation of the work that is going to be involved - often translating into a proposed budget that is underestimated by a factor of 10 or more. I hate to be the bearer of bad news, but the mature apps that you are installing for a few dollars - or quite possibly free - cost tens if not hundreds of thousands to create. That cost is covered through advertising or volume of subscriptions. Not by magically dividing development costs by wishfully dreamed up factors of ten.

I refer to projects advertised with vastly underestimated budgets as "$100 space rockets" - because that's the physical engineering equivalent. Normally these end with the $100 being spent, but unfortunately no space rocket appears. Instead, there's just a very red-faced and angry project owner cursing about broken promises (and often moving right on to the next developer who is going to promise to do the job for $100).

The lesson: You need to cost up your project realistically based on estimated developer-hours, and at a reasonable rate for a developer. You can't simply rely on quotes. See point 4 for a discussion of this, and some tips on costing.

2 Aiming to develop a feature rich application rather than an MVP

This error arises because most people use a variety of software applications every day but generally never come into contact with code that is actually being executed. Imagine being surrounded by cars your whole life - except the cars have invisible engines. If you look under the hood, you only see some extra space to place your luggage. You'd be forgiven for thinking cars run by magic - and this often seems to be the misconception with software.

And since software works by magic, it seems tempting to rattle off all the magical things you expect your imagined software is going to do. The more magic that can be packed into the description the better. After all, those tech savvy developers are not going to think much of you if you don't wow them with your vision. Because of course this is your overall objective: impressing the developer. Yes, I am joking! But putting humour to one side, it seems there are plenty of people out there who actually adopt this approach.

When coming up with your first feature list, it is easy to be influenced by those apps you use every day. The problem is most apps in common usage are already pretty mature and have collected features and been refined over many release cycles. The incremental improvements have been made post-initial release, using real-world feedback from the market. This has involved long hours of repetitive testing and bug fixing. It doesn't make sense to try and create something like this from the outset; it's just going to be expensive and you're probably going to waste time and money adding features users don't actually want.

What you need to remember is that every functional item you add, down to the buttons, the form fields and the "automatic" processes that happen in the background - each and every one of these needs coding up and therefore adds to the cost. So, what you want to do is ask yourself, ***"what is the bare minimum set of features I need to include to make my app perform the function it is intended for?"*** That's going to slim your costs to something you might actually be able to afford. And by the way, if your core features don't catch on with your intended audience, then no amount of additional bells and whistles is going to rectify that.

3 No clear specifications ("spec") document

You're on the brink of commissioning a job that's going to cost several thousand dollars, but you're only going to bother writing a few vague sentences to explain it. Can you see how this might go wrong? Perhaps you think, "I don't need to describe this task completely, because I'm talking to experts. I only need to give them a few rough instructions, and they will be able to figure out what I want." Wrong! The developer has expertise in *how* to build software applications; this doesn't mean they are magically going to know *what* you want to build. You're expecting the developer to read your mind; but unfortunately, clairvoyance is not a skill most engineers are equipped with.

Another possibility is that you think, "Well I don't actually really know *exactly* what I want. I'm hoping the developer can ask me questions and we can arrive at a plan together." Yes, it's reasonable to seek expert assistance in creating technical specifications; but in this case you really have two separate tasks to complete:

TASK 1:	The creation of the specification document; and
TASK 2:	The implementation of the instructions within that document.

In this case, consider hiring a developer to produce a spec (task1) and start by advertising this job on its own. This is a great way to test the waters and see if you're going to get along with the developer of your choice. If all goes well and you're happy with the relationship, you can hire them again for the implementation (task 2). By then they'll be familiar with the project and should be perfectly positioned to carry out the instructions, since they helped decide them in the first place.

However, whether or not you decide to commission help with the planning of your project, remember one thing: the specifications are *your* responsibility, because it's *your* project. There is no way around the fact you need to communicate what you want with the developer. Yes, you can be lazy and say, "We'll figure it out along the way through Q&A", but what this really means is that you'll constantly go down blind alleys as the developer repeatedly misinterprets your vague instructions. That's going to equate to wasted time and effort, escalating costs and a strain on your relationship.

The equation is simple: the more effort you put into creating clear instructions, the more time, money and headaches you will save - and the more successful your project will be.

You can't expect:

- to go straight to implementation when there are no instructions to follow;
- to get a reliable quote for implementation without specifications;
- to have developers essentially work for free - pre-contract - asking you questions to try and figure out what you actually want.

4 | Estimating costs based on developer quotes

This point may seem surprising and counter-intuitive: you can't automatically trust quotes you receive on contracting websites - and you certainly can't hire on the basis of the quote alone. This is because costing a software project can be complex and a skilled task in itself. However at any one time there is a demographic of developers who may have a solid coding background but are quite inexperienced with the costing process.

Costing a software project can be difficult even for those who have been doing it for many years! The problem with fixed-price bids for software work is that developers are pressured to quote low to win contracts. This normally means that for any given contract there are going to be a percentage of bids which are too unrealistically low for those developers to actually honour.Now you might think, "if they quote too low, that's their own problem. A promise is a promise!" In fact, this is normally the exact next step that happens after going with the "$100 space rocket" bid. You insist the work must be done for the stated price - because you tell yourself that's what you agreed. But try doing a per-hour earnings calculation for the developer. How many man-hours does it take to create a space rocket? If you insist on paying the developer $100 because "that's what you agreed" then you are insisting on an hourly rate that might be measured in fractions of a cent. The developer is going to starve to death before completing your project! Obviously, this kind of practicality is going to put a limit on how successful you are at holding someone to a price "because that's what you agreed". The developer is going to eventually consider their own survival to be more important than your agreement.

And now we arrive at the next phase: there are arguments, the developer jumps ship, and you are left with a stalled project. Maybe you are able to get some funds refunded through the contracting platform, or from the developer directly (because they now just desperately want out of the contract so that they can find a way to start actually earning enough money to live on) - or maybe not. Either way you have a half-finished project which is going to be difficult for a new developer to pick up and which might cost more to rescue than simply to start again. The "cheapest" option turned out to be the most expensive.

The lesson is that you have a responsibility to check quotes are realistic yourself. In fact, it is better if you can obtain a rough costing calculation independently of any quotes. In the absence of prior experience with costing software features, here's a neat trick: post your app feature list on a Q&A website (like Quora, Reddit, StackOverflow etc) - targeting professional programmers with your question - and ask, "How many developer-hours do you estimate these features should take to build?" Note that it's better to word the question in terms of developer-hours, because professional coders usually get paid salaries and so don't think in terms of costs. Answer any questions they fire back at you in as much detail as you can. The great thing about this exercise is you'll get honest answers; there will be no competing to present the lowest price. You might encounter some disagreement - but that will likely turn out to be a good thing, because then there will be an argument and an eventual consensus. If you can keep the thread alive with suitable prompting, you'll learn a lot about the challenges of your own project and where the costs will lie. Then in the final step you can simply multiply the developer-hours estimate by a reasonable rate for a developer. Now cross reference this figure with any future quotes and use it to reject the $100 space-rocket bids (and future stalled project).

5 | Giving the developer too many responsibilities

This problem can stem from trying to stretch a thin budget far beyond its elastic limit. You hire a developer at a cut-throat rate and then expect them to perform non-development tasks that are far beyond their remit.

The “classic” example of this - being the most obvious and common - is to expect your developer to also be a designer. Just pass them a vague description of an interface and expect it to magically appear with slick aesthetics. This problem is often exacerbated by the fact developers tend to like playing around with design - even though they are generally not very good at it. So, they probably won’t give you a heads up, “Hey, by the way it’s probably better if you get an actual designer to come up with some screens”.

It’s a good idea to plan screens and layouts early on, because architecture and interactivity can often be tied to the design. If instead you simply leave this up to the developer and then go along with whatever is produced, you may well end up with an ugly looking product which is difficult to back out of.

Other inappropriate responsibilities that can be handed to the developer (often inadvertently) include expecting them to:

- create the feature list
- purchase server hosting, email accounts and other infrastructure services
- make business or marketing decisions
- inject some kind of magic ingredient into your plan to make it “even better”

This last point seems to frequently recur among outfits launching “trading bot” projects. Directly in the job post demands are made like, Iinclude any innovative features you would recommend integrating into the bot”. So, the developer needs to come up with a killer trading strategy, on top of actually building the system? And what are you going to pay them - $100? If they actually had a killer trading strategy together with the skills to build the bot, why wouldn’t they just create it themself? (And the owner’s input is going to be...?)

The theme among these examples is lazy project management. Expecting you can just take your hands off the wheel and glide smoothly to your destination. That’s not going to end well!

The solution is simple: take responsibility! With every new task that comes along ask yourself, “Who is the most appropriate person for this task?” If the task is not development, then maybe the answer is not “the developer”. And if you can’t find someone suitable to delegate it to, maybe you should consider doing it yourself.

6 No allowance made for extensions, revisions, maintenance and marketing

If you think that fixed development fee you negotiated back at the start is going to be the sum total cost to build your project, think again. This assumption is not just wrong, it's likely to be wrong by at least a factor of two - and possibly over the course of the software's lifetime, a factor of 10 or more (assuming the software is successful).

What you need to remember is that any software project is more like a marriage than a one-night stand. Unfortunately, it's an ongoing commitment! One reason is because pretty much all software (there are some exceptions, but these are unlikely to be applicable in your case) rides upon a constantly changing environment. If you've ever heard of the "Red Queen" theory from evolutionary biology, then the same basic principle applies: "It takes all the running you can do to stay in the same place" (originally a quote from the character "the Red Queen" in Lewis Carroll's "Alice in Wonderland").

In evolutionary biology the "Red Queen Theory" refers to a constantly changing natural environment. In software development the changing environment is man-made. The operating system(s), open-source libraries, APIs, websites and any other third-party software your project relies on are all in a constant evolutionary cycle; updates are going in, new versions are being released, old versions are being discontinued etc. It takes constant effort to stay on top of this. In fact, it is advisable to have some kind of maintenance arrangement already in place for the moment your software goes live.

Maintenance is an important reason why you need to have additional funding at the ready from the outset, but there are also business considerations. For one, as soon as your app gets out into the real world and people actually start using it, you're going to find you need to make changes.

What you imagine users are going to do and what they actually do are often two very different things. This is one reason it is better to get the app to market with minimal features first, and then evolve it based on real-world feedback - rather than building what you *imagine* users are going to want.

Indeed, the market itself is a constantly changing environment. New tools are making old tools obsolete every day. Competition appears and disappears. Fashions come and go. A particular tool that might create a buzz at a certain time may receive a lukewarm reception at another. Most successful software producers engage in a constant update and release cycle to stay on top of changing market conditions.

Finally, it's easy to underestimate how much money you'll need for marketing. It's very unlikely that simply going live with a new website, releasing your app on Google Play etc. will be enough to hook in a substantial user base. The amount of marketing you'll need depends on the type of software application you release, how much competition is around, whether you have existing customers and so on - but bear in mind a realistic figure for marketing can often be many multiples of the development cost.

7 | Paying no attention to architecture or code quality

This one bit you some distance down the road. Everything seemed great at the beginning - development was roaring along. You just told the developer, "I need a widget x that does X" and lo, that widget appeared - and sure enough it did X. So, you said, "Well done, developer! Now I need a widget y that integrates with x to do Y" - and once again the widget sprung out of the ether, and just as requested, it integrated with x to do Y. But by the time you needed widget z which was supposed to integrate with x and y to do Z, progress had slowed significantly. You had started to notice areas that seemed to work perfectly previously - like widget x - were now breaking regularly whenever there was a new release.

Sometimes you'd be browsing your own system and discover a feature was broken and must have been broken for some time. It was starting to get difficult to even know what was working and what wasn't - and which areas would break on the next update. You would identify a bug, and a fix would go in for it, only for a new bug to be introduced elsewhere - and discovered weeks later. Your frustration with the lack of progress grew steadily alongside the growth of your not-quite-functional platform. You were burning through funds rapidly, but that polished interface you were hoping for always seemed just out of reach. Your relationship with the developer was fraying at the edges; you'd find yourself losing your temper with them and demanding to know why there were so many problems. They suddenly seemed a lot less available, and one day they just disappeared.

"Good riddance!" you cried. "Now I'll find a real developer, who actually knows what their doing!" So, you put a new job post up and invited a fresh set of developers to look over your codebase. One by one they all told you the same thing: your code was spaghetti, and it would probably cost as much to rescue as it would do to start again. Once again that "cheap" developer rate you were paying had turned out to be the most expensive.

Unfortunately, this scenario is far from fantasy. I see it on a regular basis, most often in the context of being invited to do the rescuing. In fact, I believe it is so common that for every tidy, functional codebase there are hundreds of abandoned, half-finished spaghetti heaps sitting around on dusty hard drives because their delusional owners can't bring themselves to delete them completely.

And there is a common theme which pervades all these cases: during development the owner did not look at what was happening under the hood. Now at this point maybe you throw up your arms and say, ***"But how am I supposed to be able to tell good code from bad code? I don't know how to code!"*** To that I say, imagine you are inspecting the engine of a car. Would you not be able to distinguish good design from bad design - at least at some level - even if you weren't a mechanic? If the engine consisted of perfectly congruent and smoothly polished chrome parts, with precisely aligned rows of bolts and tidy wiring, you'd probably say, "Well this engine looks in good shape".

But not if you looked under the hood and saw loose, ill-fitting parts dripping oil, tangled wires and missing bolts.

7 | Paying no attention to architecture or code quality

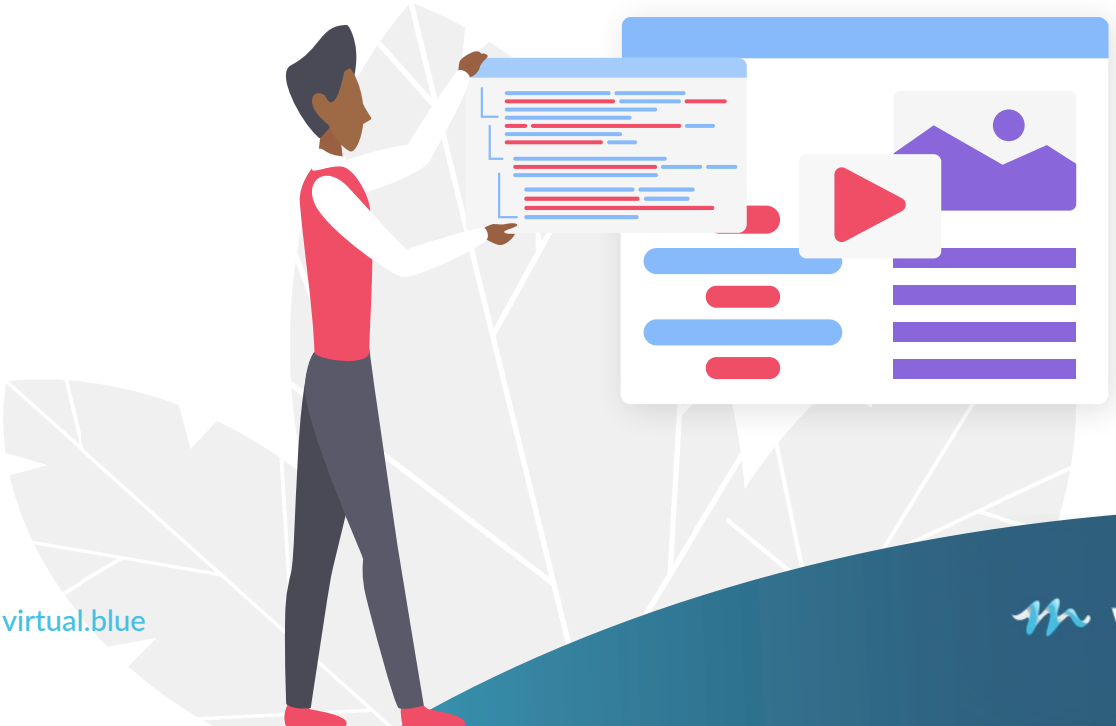
All you need to do is the equivalent when you look under the hood of your software project. Your aim is just to check the project is organised sensibly and tidily. Now admittedly you're definitely going to do a better job of this if you learn some development best practices - such as how to identify hard coding and the "DRY" ("don't repeat yourself") principle. But these only take an hour or so to read up on and understand! It would also help to research the component technologies that are being used on your project, so you have some understanding of how they fit together. For example, the choice of database system, the languages and the frameworks.

Look under the hood on a regular basis. Poke around. Put code lines into search engines and find out what they do. The more often you inspect the codebase, the more familiar and confident you will become with it - and the less daunting it will seem.

Now here's the secret ingredient: let the coder know you are interested in the codebase, and that you are including code quality as a measure of success. If you only test functionality from the perspective of a user, you encourage the developer to abandon code quality. Over time they get used to the fact no-one is checking and take to cutting more and more corners - particularly if they feel pressure to produce features quickly. Don't put pressure on the developer to produce features as quickly as possible! On the contrary, be suspicious of rapidly developed features.

The coder will adapt to the manner of your oversight. If they know you are making an effort to measure code quality, they will take more care with it. Development may seem slower at the start, but as the project gets larger, you'll keep on making progress at the same steady rate - and everything will fit together smoothly and work as it is supposed to. Keep inspecting the code. Don't be afraid to challenge design choices and ask questions.

> *Remember:* in software development nothing is more expensive than technical debt - it can write off an entire project without it earning a penny. You need to take it seriously and take appropriate precautions.



8 "False beginners" hiring chain

Congratulations - you made it to point number 8! That means you managed to avoid mistakes 1 to 7 and have successfully released your app onto the market. The promising early subscriptions have impressed an investor, and you've been able to secure a new round of funding. Great work! Now it's time to expand your development team.

You think to yourself, "I'm going to need to conduct technical interviews. But I can't really do this myself, because my background is not technical. What should I do?" Of course, the solution seems obvious: you already have a technical member of staff - the developer you hired in the first place. They must know their stuff - after all they've got you this far, right? So, you decide to handle "team fit" interviews yourself - but assign "technical" interviews to your current developer. "Go off and conduct some interviews," you say to your developer, "And then come back and tell me - in your opinion - which applicant has the best technical skills".

There are a number of problems with this approach. Firstly, interviews and hiring are a completely different ballgame to hands-on development work. Your developer has been coding solidly day-in, day-out for years - but conducting interviews is an entirely new activity, which they don't necessarily have experience with. Just because they are technically accomplished doesn't mean they are automatically well-placed to recognise technical skills in another individual. For example, they may ask questions based on their own specific knowledge - and if the applicant does not share that specific knowledge, conclude the applicant is not suitably skilled. In reality, no two individuals share exactly the same knowledge base, and so those specific questions are not going to reveal the applicant's actual expertise. Furthermore, the developer may not ask questions in a manner that elicits desirable qualities such as problem-solving skills or communication.

But the biggest issue with relying on your existing developer to grade applicants' technical skills may come down to motivation. What is the developer going to do if they encounter an individual more able or accomplished than themself? Are they going to say, "Wow, I'm blown away. I'm recommending we hire you as my supervisor!" Unlikely, don't you think? Given the developer receives a fixed payment rate, which is not directly tied to the success of the business, their primary motivation is always going to be preserving and advancing their own position. They may not be inclined to hire someone who they perceive might threaten their comfortable perch - and instead find excuses why that particular applicant would not be a suitable hire.

8 "False beginners" hiring chain continued...

The fact is, you didn't hire the best developer in the world in the first place; you just tried to get a good deal. The only reason developer A is interviewing developer B is because you hired developer A first. If you had hired developer B first, they would be interviewing developer A. If A is better than B and you hire A first, then you might get both A and B. But if you hire B first then you'll never get A - or anyone else better than B. Your strategy of enlisting the first developer you hired to hire subsequent developers may result in ensuring the first developer is the best one you ever get.

"If each of us hires people who are smaller than we are, we shall become a company of dwarfs." - David Ogilvy (the "father of advertising"). Can you really trust an inexperienced interviewer with a vested interest *not* to do that?

Once again, the solution lies with you. You are ultimately responsible for who is hired, so you should take steps to ensure you have complete oversight of the hiring process. But how do you conduct those technical interviews without a technical background? Short answer: you don't.

In fact, recent research has shown that evaluations from unstructured interviews are not particularly good predictors of future job success, whether conducted by a "technical" person or otherwise. Without a formal scoring system, evaluators tend to fall back on their own personal biases - and pick the candidate they like the most, rather than the one most suitable for the position. The answer is to ditch the technical interview in favour of a more formal technical test with a rigorous pre-defined scoring system. That scoring system is all-important; "rigorous" means that every point should have a specific, checkable reason for being awarded (or not). There should be no subjective grading (e.g. "give the candidate a mark out of 10 for how good his subroutine is").

You can feel free to involve your existing developer in the creation and even the grading of the test - the formal scoring system will make it difficult for biases and personal motivations to influence the results. Just make sure to double check that, during grading, the scoring system is being properly applied.

9 Assuming "Scrum" (or other "development methodology") will save you

Now you're really getting into the bigtime! Your operation has expanded and you've got yourself a whole team. But how to manage them? You have to admit, you're feeling a little bit out of your depth. You have the responsibility of managing a group of people who all possess technical skills - while you don't have these yourself. They know more about the project than you do! So what value is your input going to have? The answer, you reason, is to implement a "development methodology", like "Scrum". After all, that's what big businesses are always raving about, isn't it? If you implement Scrum then your responsibility automatically becomes clear - you are the one who directs the Scrum-like activities: stand-up meetings, writing "user stories" on post-it notes and collecting them onto "story-boards", organising the voting on the point assignment of tickets, holding "retrospective" meetings where you ask developers to write their feelings on more post-it notes. Sounds productive, right? (And not at all childish or patronising...)

Whilst you are indeed going to need some kind of system to organise your team - and this is likely to include conventions for tracking issues, communication and task-assignment - it is a mistake to think that implementation and maintenance of this system is all you require to succeed. This is like being in charge of a fleet of vehicles, and thinking all you need to do is arrange them in a nice, neat convoy and you'll get to your destination. They're going to drive round in circles in that nice, neat convoy because no-one is telling them where to go.

To know where you need to go means fundamentally understanding the project you are working on, and coordinating efforts so they are all in the direction of the overall objective. You can do this using an organisational system like Scrum - but you could equally do it with another. Thinking the set of conventions you choose is the important part is like thinking the important part of communication is whether it is done using post-it notes or not.

Actually, if you have made it to this point organically as a founder of your own operation, and you have essentially been adhering to the principles I have outlined in previous points, I think it is unlikely you'll jump to this kind of practice - at least not intentionally. This is because by now you probably already understand your own role and the value it adds. You've been taking a regular interest in what is happening under the hood and have a reasonable comprehension of it. You know you have a responsibility to control code quality. You've realised that developers are great at carrying out well-defined assignments - but need your input in choosing the assignments. You've been steering the ship successfully up to this point, because you are the one with the vision and an eye on the objectives.

Most often this mistake is made not by founders, but by middle managers coming on to a project that is already under development. As a founder, you are more likely to meet it in the context of manager(s) (or perhaps consultants) trying to persuade you of the benefits. Be wary of development methodology aficionados; these people are often falling back on management bureaucracy as a band-aid for their own imposter syndrome - to mask the fact they don't actually know what they are doing. The tell-tale sign is that they try to operate purely via endless meetings and they make their decisions by "consensus" to hide the fact they don't feel equipped to make decisions by themselves. They don't look under the hood, and don't find the time to gain insight into the actual project.

The solution is simple: avoid being charmed by development methodology salesmen and their promises of greater productivity. If your system is not working for some reason, you need to take practical steps to fix it.But this probably doesn't mean totally abandoning the one you have in favour of adopting a completely different one. Be suspicious of middle managers pushing for the adoption of new development practices; always ask yourself whether what is being advocated is really going to add value. Above all, don't take your eye off your projects' real objectives.

10 Getting sucked into fads

I decided to make this the last on the list, because although it can occur at any point during development, it has the capacity to do the most damage when a system is already reasonably mature.

Software development has always had a lot of fads. New methods or technologies which suddenly become all the rage, and simply everyone is suddenly feverishly implementing them, and raving about the wonders. Only in a small percentage of cases does the hysteria turn out to have foundation; more often than not the enthusiasm fades leaving behind mountains of semi-functional code, in many cases created using tools that developers have already stopped learning. You only need to look at the evolution in popularity of front-end frameworks to confirm this is the case.

As a project director you'll meet this in practice when developers start complaining your existing system is "out of date" - and that it "needs to be rewritten" in shiny new language X (or even worse, have actually started the rewrite without your consent). The mistake is to get swallowed up in the hysteria and go along with the recommendations. Remember the golden rule: if it isn't broken, it doesn't need fixing!

Always be suspicious of calls for rewrites. Calls for rewrites have literally bankrupted businesses. The classic is Netscape Navigator. Once upon a time NN was the world's most popular browser - but where is it today? Well, they tried to rewrite it!

Ask the developer why they think a rewrite is necessary. If they say it's because the feature has bad architecture and is too much of a mess to be rescued, then the call might be valid. However, if the answer is, "because no-one is using this language/tool anymore" then it's time to put on the brakes. Remember: users don't care what technologies are being used so long as the app works and is useful to them. This is your priority - not whether it is built according to the latest development trend.

CONCLUSION

If you've been paying attention, you'll notice the above points all have solutions that share a common theme which can be summarised in a single word: effort. The success of your project depends on how much effort you personally are prepared to put in. Abandon your duties and abandon your project. Hands off, and you crash. But if you maximise your involvement and embrace your responsibilities, you'll naturally avoid the pitfalls - and reap the rewards.