

How to Keep Your Boss from Sinking Your Project

By Andrew Stellman and Jennifer Greene

Like it or not, your project needs management. Yet few good software projects can survive bad management. If you're a programmer on a high-visibility project, this PDF offers five principle guidelines for managing upward that will help you help your boss make the right decisions about setting expectations, working with users and stakeholders, putting the project on the right track, and keeping it there.

Even if you have the best boss in the world, chances are he or she does not come from an IT or software engineering background, and may lack the time or technical focus to manage your project well. And if your boss sinks your project, you'll share the blame—even if you did everything right. The guidance in this PDF can help you understand what problems cause projects to fail and how to fix them, and what you can do to keep your software project from running into trouble.

Contents

Why Do Projects Fail?.....2

Get familiar with the telltale warning signs of project problems

Set Expectations Early and Keep Them Up to Date.....4

Set the scope and scale of the project

Make an Informed Selection of a Software Process9

Get involved in how the software is built

Keep Your Manager Involved with the Project.....14

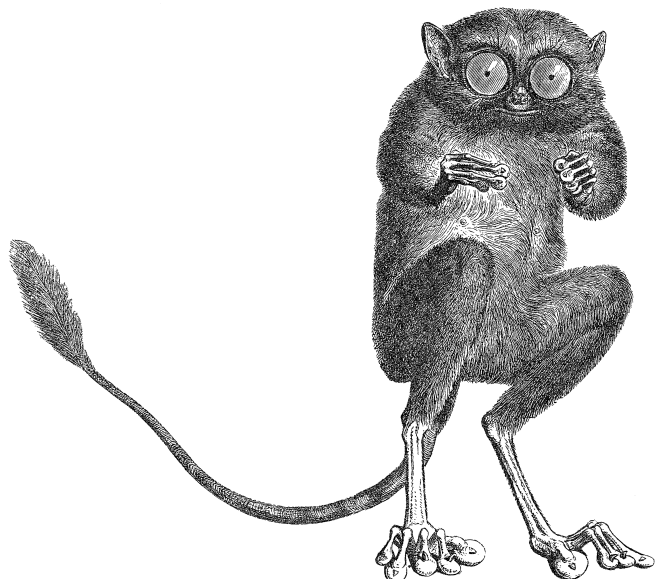
Work with your manager as you and your team build the software

Make Intelligent Course Corrections.....17

Don't compromise quality when things go wrong

Alter Your Tactics, but Stay True to Your Strategy20

Keep a big-picture approach



How to Keep Your Boss from Sinking Your Project

Like it or not, your project needs management. It's not a stretch to say that many good software developers have a healthy distrust of management. It's difficult to work in the software industry for a long time without having encountered at least one or two incompetent managers. Yet few good software projects can survive bad management. If you're a programmer on a high-visibility project, this PDF offers five principle guidelines for managing upward that will help you help your boss make the right decisions about setting project expectations, working with users and stakeholders, putting the project on the right track, and keeping it there.

Even if you have the best boss in the world, chances are he or she does not come from an IT or software engineering background, and may lack the time or technical focus to manage your project well. And if your boss sinks your project, you'll share the blame—even if you did everything right. The guidance in this document can help you understand what problems cause projects to fail and how to fix them, and what you can do to keep your software project from running into trouble.

Why Do Projects Fail?

By and large, good programmers have one universal response to bad management: "It's not my problem." Unfortunately, it *is* your problem. You may not be able to control it directly, but if you put your head in the sand and keep coding, your project can fail no matter good your work is. And in the end, you may be seen as a poor performer because of that failure.

You may be skeptical of this, but the facts are there. Many large corporate and enterprise development projects are troubled—as many as two-thirds of them, according to a recent study by the Standish Group (http://www.standishgroup.com/sample_research/PDFpages/q3-spotlight.pdf). It's a well-established fact that as software projects get larger or more complex, they become much more likely to fail. And while some of the failures are due to poor development talent, the vast majority of them are due to bad decision-making, bad prioritization, and other stuff that your manager does before the project is even assigned to you.

It may seem unfair that you are going to be held responsible for the outcome of a project whose problems were injected before you even got your hands on it. But it happens all the time. And when it does, you end up wasting time, working many late nights and weekends, and delivering software that doesn't make your users happy, if you manage to deliver anything at all.

Luckily, you do have some options. If you understand why projects fail, and the consequences of the decisions being made above you, you can sway those decisions in favor of your project's success. You have one big advantage here: you are the expert on software development, and many people in the company will defer to you on technical matters. If you have a good understanding of the sources of your project's problems, and you voice your opinions, you can use your expertise to make real and positive change in your project's direction (and its success).

The most serious project problems originate in management, but they can be easily confused with technical difficulties. When that happens, you'll be asked to come up with a technical solution to a management problem, and that's unfair to you. (After all, it didn't matter how well tuned the engine was on the Titanic.)

Spot Project Problems Early

It's easy to miss the telltale signs of project failure until it's too late. Think back to your last gut-wrenching project where everything seemed to fall apart. (If you've been around long enough, you have almost certainly seen your share of these. And if you're like most competent developers, you've been very hard on yourself about it.) At the beginning of the project, everyone seemed to have a good grasp on what needed to be done. They were confident when the project started, and every time your boss asked about the project's status, you and the rest of the team reported that things were moving forward.

But at some point, it became increasingly clear to everyone that the project was facing serious problems. It was well past the software release deadline, but major quality problems kept coming up. It might have been that the team was having trouble with a particular third-party tool, or that the database was running far too slowly, or that the system couldn't handle the amount of traffic that the users needed. Or, maybe when the users got their hands on a preliminary version, they kept saying that critical features were missing—features that you had never heard of before then. No matter what the project problem was, the business outcome always seemed to be the same: the project has run late, and now someone has to answer for the problems. And, unfortunately, that usually ends up being the team. (If you're lucky, the manager will share the responsibility.)

One frustrating aspect of this typical failing project is that nobody seems to be able to do anything about it. It always seems like the solution to these nagging problems is right around the corner. But it's been around that corner for weeks or months, and often the manager has to make a very tough decision: to either release the faulty software prematurely with the promise of (possibly interminable) patches and fixes, or go back to the users and tell them they have to wait. If you can influence that decision, your manager will make the right one.

How did the project end up in this sorry state? And, more importantly, how can he prevent the next project from ending up the same way? By understanding what causes project problems and how to correct them, you can use your influence to make all the difference in your project's success.

When projects fail, they most often fail from the top down. Even the most competent and talented software engineering team can fail to produce software if they are not given the right direction and oversight.

There are five important guidelines to help your boss follow that will make it much more likely for even a large-scale project to succeed:

- Keep expectations and scope up to date
- Make an informed selection of a software process
- Track the progress of the project
- Make course corrections that don't compromise the quality of the software
- Keep a big-picture perspective throughout the course of your project

Each one of these principles involves definite actions and addresses specific problems that can cause projects to fail. And by working with your manager, you can make sure that they all happen.

Set Expectations Early and Keep Them Up to Date

Misunderstanding the scope of the project is the single most common cause of project failure. The people who will use the software have one idea of their needs, while the people building the software have a different idea of what those needs might be. This can be very subtle, like a misunderstanding about how a certain number is calculated. Or it can be dramatic, like missing an entire expected feature.

It's important that the team is following a single working vision. One of the most important things that a good manager is supposed to do is keep the software true to that vision. Unfortunately, few bosses really understand that, which is why so many project teams end up building software that doesn't quite meet the users' needs. If the project is going to go well, everyone in the organization needs to agree on exactly what that vision is. If your boss can't make that happen, then you need to do it yourself.

Understand the Vision of the Project

Many people—from both technical and business backgrounds—have trouble with the concept of the project's vision. The word “vision” is often abused as a business buzzword, and many people assume it's simply a vacuous concept. It's not. To

understand the vision of a project you must have a clear understanding of these four elements:

- The business background of the project
- The stakeholders who need the software built
- The users who will work with the software
- A high-level understanding of the stakeholders' and users' needs

Most managers live in the world that defines the business background. By the time a project has started and a team has been assembled, your boss should have a good understanding of who the stakeholders are, because these are the people who need the software built. And he should also know who the users are—the people who will be doing the day-to-day work of the business, or the customers that the business is trying to reach. He's likely to know these things because it's very hard to get a software project funded and agreed to without them.

Your project is likely to go wrong in one of two ways at this point. The first is when your manager doesn't share the vision with you and the rest of the team. This is a very common problem, because often the vision seems self-evident to managers and business people who have spent weeks talking about it. It's easy for them to forget that you don't come to the project with the same perspective.

This is compounded by the fact that most software engineers—be honest—are very optimistic, and are often happy to agree to the project without fully understanding it. What's worse, we're generally willing to begin building the code before we gain this understanding. If the developers don't understand why the project is needed and who needs it, it is likely that the path of least resistance will be taken. The team will build a simpler project that's easier to develop but which doesn't fully meet the needs of the users and stakeholders.

The second way the project might go wrong is that you and your team will do exactly what you're told. This is a case of, "You did what I said, not what I meant." The typical organizational structure has the engineering team reporting up to a manager. If that manager asks the team to begin building the software, they'll often simply start doing it, without asking why. Any miscommunication at this point will get "baked" into the software. If your team isn't anchored by an overall vision of the product, there's no way you could possibly have the perspective to recognize that you've been asked to go down the wrong path. Of course, mistakes will eventually get caught, but they'll be caught very late in the project, when it will be expensive (or impossible) to fix them.

Of course, it's also possible that your boss doesn't fully understand the vision. Or, even worse, there may be no vision at all. Many projects are undertaken without a

clear goal. It's very important for you to recognize when someone is pushing for a project without really understanding why. Successful projects fill a clear need. If these needs are unclear to you or your boss, then it's impossible for the team to meet expectations, and the project will fail. *You need to be very loud at this point.* Luckily, as a respected developer, you have a voice that will be heard, as long as you're saying something that makes sense. If you are asked to build something but nobody can explain exactly why they need it or what it should do, you have every right to tell them that they're asking you to read their minds, and that nobody will be happy with the result.

Make Sure the Vision Is Clearly Stated

The way to avoid this problem is to make sure that the vision and goals of the project are clearly documented. This may take the form of a vision and scope document, a project charter, a business requirements document, a system requirements document, a concept of operations document, and so on. There are many ways to write down the vision.

One minor trap that the software engineering team may fall into is to think of the "vision" as corresponding to the solution: the features that will be developed, the scope of the release, and how it will be deployed. These aspects certainly need to be determined before the project starts. But it's likely that your boss thinks about the project differently. He thinks about the needs that are being served. A good project manager will connect these two ideas, and make sure that the solution being developed meets the needs of the users and stakeholders. But if you don't have access to a good project manager, you'll have to do that yourself. Luckily, most good programmers have the skills for this task. But everyone needs to be clear on both ideas before a single line of code is written.

No matter whether you, your boss, or a project manager is gathering the vision and scope, the same things must happen. Gather all of the relevant information needed to build the document. Speak with all of the stakeholders, and with as many business users as possible. You will generally need to research the project background, and will also need to gain a clear understanding of why the project is needed. Most importantly, you must write this information down and share it with the project team. This document will be relatively short (usually just a few pages), and will always be written in language that team members, users, stakeholders, and other senior managers use in their day-to-day work. If you do good research, the document should be relatively easy to write.

Using a Vision and Scope Document

One way to make sure that the vision is clearly stated is to use a “vision and scope” document. This is your first and best opportunity to make sure that the project starts right. The outline for a typical vision and scope document looks like this:

1. Problem Statement
 - a) Project background
 - b) Stakeholders
 - c) Users
 - d) Risks
 - e) Assumptions
2. Vision of the Solution
 - a) Vision statement
 - b) List of features
 - c) Scope of phased release
 - d) Features that will not be developed

This document is one of the most important tools that you have; it is also one of the easiest to implement. A good vision and scope document will help you avoid some of the costliest problems that a project can face. By writing the document and circulating it among everyone involved in the project, you can ensure that each of the stakeholders and engineers shares a common understanding of the needs being addressed—and that each agrees that the software must address them.

For more information on the vision and scope document, see http://www.stellman-greene.com/Vision_and_Scope.

Once the vision is written down and publicized, you must read it, understand it, and feel that it is an accurate statement of the problem. Make sure that every stakeholder and as many users as possible read the document, understand it, and agree with it. And, most importantly, make sure that any problems that were found with the vision and scope are corrected. Your boss will need to be a part of that, because he’s the one with enough clout in the company to drive the effort to get everyone on board.

Use Estimation to Understand the Scope and Scale of the Project

In order to get started with the project, there must be a consensus between you, the team, your boss, and the project stakeholders about its scale. Unfortunately, it’s easy to make big projects sound like they’re not much work. Sometimes business

problems that are simple to describe on paper are technically challenging to solve in code. It's often easy to ignore problematic details when looking at the big picture.

An unfortunate truth is that many technically competent developers tend to put off thinking about a part of the project that is potentially tricky by simply deciding to “deal with it later.” It's very common for technical people to oversimplify projects. This is partly due to software engineers' optimistic nature. It's also partly due to poor planning practices. Either way, if your team does not understand the scale of the project from the beginning, you will be in for an unfortunate surprise later on when your project goes off the rails.

One effective way to anchor the scale of the project is to lead an “off-the-cuff” estimation session. To do this, call a meeting with your team, your boss, and one or more stakeholders. The goal of the meeting is to come up with the number of days, weeks, or months (depending on the size of the project) that it will take to build and release the software.

However, the purpose of this meeting is not to set a time frame or deadline. And, in fact, you need to make it 100 percent clear to everyone at that meeting that *this information is not be used in that way*. If you can, make sure this caveat is written down in big letters on anything that comes out of the meeting. The estimate is used as a tool to help everyone learn more about the project, and to help both the technical and business sides gain the same understanding about what's involved in completing it. It will also help your boss work with the stakeholders to trim down the scope of the project if it looks like it will take too long.

Estimation is the right tool for this stage because it requires that the team members deliberate on every aspect of the project. The only way to put a number on a project is to sit down as a team and think through many of the details that would otherwise be left to address later.

Since the project has not even begun yet, you and your team do not have all of the information you need in order to come up with a number. To deal with incomplete information, you must make assumptions about the work to be done. Assumptions are an important key to estimation. If two people disagree on how long the project will take, it's likely that the source of their disagreement is that each made different assumptions about details of the work product or about the strategy for producing it.

For example, if one person thinks that a particular feature will take nine weeks and another thinks it will take only four weeks, their disagreement is probably not about the work itself. It's more likely that they disagree on the scope of the project. The person with the longer estimate may think that they're planning on building a complex user interface for that feature, while the other thinks that there will be a

simple command-line interface. The way to resolve this discrepancy is to make an assumption one way or the other, just for the sake of the estimate. By writing down that assumption, the team makes a promise to revisit the issue later. The written assumption also gives the users and stakeholders important information about decisions they will need to make if they want to change the scope of the project.

By documenting these assumptions, everyone is forced to think through all of those issues that they would otherwise gloss over. During the estimation meeting, write down these assumptions as they are brought up, and then compile them into a brief, easy-to-digest document after the meeting is over.

The estimation session should produce this list of assumptions, as well as a good discussion of how you and the others arrived at them. During the meeting, everyone should talk about the assumptions they need to make in order to put a number on the project. Your boss and the stakeholders should be made to feel comfortable participating in this discussion as well; many of those assumptions will often turn out to be nontechnical, or about the expectations of upper management. Try to keep in mind that while your boss and the stakeholders are not technical, they do have a perspective that you need to understand. The things that these people say at this meeting can make the difference between success and failure for your project, so the more comfortable you can make them, the better off you will be in the long run. This meeting will also help everyone gain a much clearer understanding of the complexity of the project. The list of assumptions will help a great deal later on when you run into problems and have to guide the team through difficult project changes.

Make an Informed Selection of a Software Process

Before the team begins building the software, everyone needs to agree on exactly how it will be built. There are many ways to approach a software project. The only way that a manager can really understand what's going on in a software project after it gets underway is to have a general understanding from the beginning of how it will be built. He needs to be satisfied that everyone involved in building the software is in agreement on this, and that they have all of the resources they need to follow that approach.

Most companies that build software already have an approach to building it: it's called a software process or software development lifecycle (SDLC). A software process is a set of activities that, if performed, will result in working software that meets the needs of the stakeholders and users. A typical process includes activities from many different areas of software engineering: requirements management, project planning, quality assurance, risk planning, project monitoring, configuration management, subcontract management, process improvement, and

other areas that the project team needs to concentrate on in order to build enterprise software.

Programmers often wince at talk of a software process—and rightly so. Most experienced programmers have lived through at least one process improvement initiative that has amounted to little more than paying a consulting company to fill binders full of documents that nobody reads and to get certifications that nobody cares about. But while this abuse of process is disturbingly common, that doesn't make it any less important to get it right. You don't get a pass on process just because a lot of people get it wrong. If you can turn this trend around and get it right, you can increase the odds of your project succeeding. It will make your life much easier along the way, too, because everyone will know what to do, and you can spend more time programming and less time babysitting.

This is where your manager will have the most impact on the project. If he makes the wrong decisions, he can cause the team to do unnecessary work or miss important activities. It takes some effort and learning to really understand why all of the activities have been included in the process. So, while your manager doesn't necessarily need to know the technical details of how the software will be constructed, he *does* need to know who will be reviewing each intermediate work product, and why team members are performing certain activities. And face it, you and your team members have technical expertise, but you do not have the organizational clout that your boss has. The same way your manager needs to trust your technical knowledge, you need to trust him to use his authority to give the informed approval necessary for determining how the team should work. But that means making sure that his approval is truly informed.

Your boss can do a lot of damage to a project by making uninformed decisions about software process. One mistake many managers make is to avoid this area entirely, and simply leave this up to you and your team. This “hands-off” approach leaves projects unchecked until the very end. Many developers are perfectly happy with a “hands-off” approach, because it means less work in managing upward. However, it also means that your project could spiral out of control before you're even aware of it, and that your boss will feel comfortable blaming you for the failure down the road because he left all of the decisions along the way up to you. Don't fall into that trap—keep your boss informed, regardless of whether he's asking for details.

Should the Team “Go Agile”?

A timely example of a process question that many developers are encountering is whether to adopt an *Agile* software process. Agile is a philosophy of building software that has gained support within the development community. There are many developers who have reported that Agile practices have made an enormous

difference in their projects. If your past projects have not gone well, you may already be thinking about asking your manager to give it a shot on your next project.

One important aspect of Agile methodologies is that they depend on technical teams breaking the project down into small, independently delivered chunks. Sometimes this is done with a phased or iterative release. Other times it's accomplished by breaking the software itself into smaller modules or even separate programs that work as a suite. Some projects can be broken down efficiently in this way. However, not all projects can be divided this way, and attempting to do so can potentially lead to serious problems later on.

Many teams that “go Agile” do so as a result of a grassroots campaign by the developers to change the way they work. This is often in response to unrealistic expectations, constantly shifting scope, and generally bad management practices. But that's not necessarily the case; a well-managed team looking for ways to improve will also often look toward Agile methodologies, because those techniques have a focus on constant quality improvement and allow developers to influence the quality of the product in ways that are normally out of their reach. In either case, you may find yourself going to your boss with a request to go Agile, and it's important to realize that this will change the entire way you and your team do your work.

What Is Agile?

Agile methodologies started to gain popularity in the late 1990s. Agile is a loose collection of principles and practices that project teams can use to build software. Extreme Programming, SCRUM, and Crystal are three of the most popular processes used by Agile followers. Agile development focuses on very tight collaboration between developers and business people; breaking projects down into small, frequently delivered releases with lots of user feedback; and keeping the team small and flexible in order to react to changes from the users.

The overall philosophy of the Agile processes is stated concisely in the Agile Manifesto: <http://www.agilemanifesto.org/>.

Programmers often forget that there are many correct answers to the question of how to save failing projects, and Agile is only one of them. It can be an extremely effective one, but only for the right projects and in the right situations. There are costs and benefits of any change, and both are magnified when the change is sweeping. And there are few changes as sweeping as throwing out the entire software process and replacing it wholesale with something that you haven't necessarily tried before. This means that you and your manager need to accurately and honestly assess the pros and cons of going Agile.

One easily overlooked cost of going Agile is the extra effort required to break the software down into small, phased releases (or iterations). Each iteration must produce high-quality, release-ready software. This means that each iteration must be tested and thoroughly evaluated. Instead of one vision and scope effort at the start of the project, you'll need to do a new one for each iteration. There are other hidden costs that are easy to overlook as well. And, most importantly, each new iteration that's sent to the users and stakeholders will come back with feedback and comments, and you'll have to change direction to implement them. This is good because it means your project stays flexible in the face of changing needs. But it's not free—it takes a whole lot of effort. And teams that adopt this approach can often fall into the trap of putting off some features that the users need, only to be faced with nasty surprises (and maybe even software they can't release) in later iterations, along with other problems that could have been avoided had more planning been done early on. To make matters worse, sometimes the decision on what features should be included in a given iteration is difficult to make without fully exploring the users' requirements.

In other words, there are projects for which iteration or phased releases make perfect sense, and others where they don't. Luckily, there are a few rules of thumb that can help you figure out when to pressure your boss to go Agile, and when it might be best to stick with your "stodgy" traditional development process. Projects that can't be iterated easily often include:

- Automation of large-scale end-to-end business processes that must be replaced all at once and without errors (such as enterprise resource planning systems).
- Systems that have a costly hardware engineering component (such as computer-aided manufacturing or engineering systems).
- Projects where the scope cannot be broken down into smaller pieces (such as an antilock brake system in a car or the security system for a bank).
- Systems where the costs of defects or misunderstood scope are too high to allow users to discover them on their own (such as air traffic control or medical systems).
- Systems where miscalculations could carry a legal consequence (such as billing, accounting, or tax systems).
- Software that is difficult to roll out to the users, or for which there are no human users (such as embedded systems).
- Large internal projects that could disrupt the business, where users depend on the software working exactly as they need it in order to do the company's day-to-day work.

Another often-overlooked aspect of Agile is that it requires a very high commitment from the business users. The Agile Manifesto states this explicitly: “Business people and developers must work together daily throughout the project.” Sometimes an organization can accommodate full-time involvement from users and stakeholders. Indeed, the more that the users and stakeholders are involved, the higher the probability of success. However, few organizations can afford to dedicate even one user or stakeholder to the project full-time, and it’s rare for a business team to hire a technical liaison as a full-time role. In some organizations, it’s not even clear that such a role is possible: can this person maintain their business knowledge if they are working full-time for the technical team and not out in the field?

So before you push for using an Agile approach, make sure you understand the business climate and whether you’re really asking for something that your company can deliver. It may be the case that your project and your business users are perfectly suited to Agile. If your boss has the manpower (and clout) to assign business users and subject matter experts to work full-time with the project team, and if your project lends itself to iterative development, then it may well be worth it. There’s one way to find this out: ask him, and lay out all the facts. This is his domain, and if he can’t even answer those basic questions, it’s unlikely that you’ll make much movement in this area anyway. And the last thing that you want is to select a doomed project as your Agile pilot. If the project is set up to fail from the beginning, it won’t matter how good the methodology is. But a visible failure is a sure-fire way to convince people in your organization that this Agile stuff doesn’t really work.

The best way to think about this is to try to see things from your boss’s perspective. Is there enough benefit in going Agile to make it worth the cost? The only way for him to assess this is to have a good understanding of the way the team builds software now, and how much it will cost to make this change. And that’s not just a cost in man-hours or calendar time; it’s also measuring the cultural cost (in terms of whether your company’s work culture can support it) versus the cultural benefits (because the added collaboration and the feeling of control that the team gets can provide a big morale boost). There are quality costs and benefits as well. Iterative development gives everyone on the team and potential users a better intuitive grasp of the quality and health of the software; however, very complex projects can careen out of control if more time is spent evaluating small, consecutive builds than on planning and developing them.

There’s no blanket answer. The only way to figure this out is to work closely with your manager, your users, and your stakeholders to look at the particular circumstances of your project. It is only through understanding of the scope and

impact of each change proposed by your team that you can all reach the right decision.

Keep Your Manager Involved with the Project

If you've followed our advice so far, then by the time the project is underway, your boss will be involved at a level where he can help head off problems as they arise. You'll be able to alert him early, so that he won't be surprised by problems when it's far too late to take action. You will have come up with a real set of estimates—all of which are much more detailed than the “off-the-cuff” ones discussed earlier—and built a project schedule from them. By using this schedule and creating a transparent environment, you can help your boss stay involved with your project without micromanaging it or becoming too involved in its day-to-day work.

Hold Periodic Schedule Walkthroughs

Set up a regular meeting with your manager to walk through the progress that has been made on the project. The frequency of this walkthrough meeting will depend on the duration and scale of the project. A three-month project will probably have more frequent meetings than a two-year project; a highly complex and risky project should have a higher level of management involvement than a simple and straightforward one.

Build a Project Schedule

When the software team puts together an initial schedule, this should be set aside and used as the *baseline*. A baseline is a fixed schedule that represents the standard that is used to measure the performance of the project. This means that there are two versions of the schedule: one is a fixed baseline version that is kept as a reference, and the other is the real schedule, which is updated to reflect what actually happened over the course of the project. Project managers will often use metrics to demonstrate how well the project is progressing, and how accurately the project was planned.

One metric used to evaluate the accuracy of the schedule is the *variance*. This involves two numbers: the *budgeted cost for work scheduled* (BCWS), the estimated effort of the actual tasks that appear on the baseline schedule; and the *actual cost of work performed* (ACWP), the effort spent on the tasks in the schedule that have actually been completed by the development team members. The variance is the difference between these two numbers: BCWS and ACWP. (BCWS and ACWP are standard acronyms used in most earned value calculations.) If the variance is positive, then the project cost fewer man-hours than were budgeted; if it is negative, the project overran the budget.

Another way earned value can be calculated is by generating the *cost performance index* (CPI) for the project. CPI is calculated by dividing BCWS by ACWP ($BCWS / ACWP$) and then multiplying by 100 to express it as a percentage. A CPI of 100 percent means that the estimated cost was exactly right and the project came in exactly on budget. If it is under 100 percent, the work cost less effort than planned; a CPI greater than 100 percent means that the estimate was not adequate for the work involved. CPI can be used to compare either projects or phases within a project.

For more information on building a project schedule, see <http://www.stellman-greene.com/Schedule>.

During the schedule walkthrough, you should explain what has happened on the project since the last meeting. Discuss any tasks that have been added or changed since the last meeting, and alert your boss to any problems that may affect the schedule in the future. You should come prepared with a list of everything that has happened on the project since the last meeting, and an overview of what you expect to happen between now and the next one.

You should also present some numbers to show schedule progress. This may seem like overkill, but it's the most effective way to show a real track record. It's very easy for a boss to dismiss the team's progress so far when confronted with a minor delay or problem; having numbers to back up your progress is one way to keep

everyone on the same page, and to make sure that you and your team get the credit they deserve. One good way to show this is to use schedule variance (see Project Schedule Tips sidebar). Be prepared to explain not only how all of the numbers were calculated, but also what they mean in terms of project success and how accurate your planning has been.

If the project is going well, you should *not* let your boss start canceling these meetings. He'll often be tempted to focus only on problems, and assume that a project that's going well today will continue to go well for the next two months. But even if your project is going well and stays in good shape, you should continue to meet with your boss. It's an important motivator for him to feel like he continues to understand and appreciate your work. (Yes, sometimes you need to think about motivating your boss.)

Insist on Transparency

Your team creates many work products during the course of a typical software project: schedules, scope documents, requirements, issue logs, technical documentation, test plans, and, of course, code. All of this should be shared, not just among the project team, but also with the stakeholders—and you should give your boss open access to all of it. Help him familiarize himself with as much of this documentation as he can (depending on his level of technical expertise).

The reason for this is that you need him to feel comfortable browsing the work products for the project. You should be happy if he occasionally checks them against the project schedule to see if you and your team are producing everything you said you would. Many developers instinctively feel that this is somehow intrusive or “Big-Brother-ish,” but that's not really so. The more involved your manager is with the day-to-day goings-on of the project, the more likely your project is to be successful.

The only danger is micromanagement: your boss should be aware of your output, but he should still trust you to do a good job. He should be following a “trust, but verify” approach, checking that you are doing all the work you promised but not inserting himself as a bottleneck and insisting on personally approving everything you and your team produce. Your boss should feel that the team members are accurately reporting information.

The best way to avoid micromanagement is for your manager to stay on top of the project by keeping abreast of any developments on his own, to review publicly available work products, and to ask questions about how you and your team members are doing your jobs. That way, he can be a part of the overall project dialog, but without making the team members feel like he's redoing their work or questioning their expertise. This will help him make informed decisions and reinforce his trust of the team overall, which is the goal.

Make Intelligent Course Corrections

An “unfortunate” side effect of staying on top of the project is that your boss will be aware of many problems that happen along the way. Every project runs into problems: bugs, quality problems, scope changes, personnel problems, even petty arguments. Even though every programmer knows that these things are a day-to-day reality on a typical project, many managers are shocked when they see these problems for the first time, and do not like dealing with them. Sometimes the problem stems from individuals who require extra guidance or corrective action. Other times, there are technical challenges or external problems, and the team needs to come up with creative solutions. Users, stakeholders, and managers often cause the most serious problems by changing their needs and requirements late in the project. But in all cases, some sort of corrective action is necessary.

If you’ve kept your manager in the loop throughout the project, then you’ll be in a very good position to help him lead the team through a course correction—before the project fails. There are two basic rules that, if followed, will help you work with him to make the right decision about the direction of the project without compromising the quality of the final product. To many programmers, these rules are counterintuitive. But they are time-tested, and really do work. First, you should stick to the process that the team agreed upon from the beginning. And second, you should not accept a low-quality final product. If you do these things, your project will stand a much better chance of surviving a crisis.

Don’t Abandon the Process...

If a project is running late, many important activities start to seem frivolous to some managers: that three-hour meeting that brings the programmers together to review the architecture or new patches; the important code review or refactoring that improves the design of an important object; the building of automated unit tests to catch bugs before the software is shipped; and so on. To someone not directly involved in these activities, it may seem that they are too expensive to waste those programmers’ hours on.

The reason that most teams agree at the beginning of their project to build the software in a way that involves communication, collaboration, and review is that those practices work. Any time they’re circumvented, treated as frivolous, or cast aside as over-engineering, the project suffers time delays. Every mailing-list post, every team meeting to discuss a design decision, every knowledge transfer session between a senior and junior team member, and every single review can potentially save time and effort by catching costly defects.

Yet when your boss feels that your project is coming under time pressure, you may find yourself being asked to cut these practices. If he makes the wrong decision, it

could lead to a great deal of rework that might otherwise have been avoided if defects had been caught much earlier. By dropping practices from the process, your boss is showing the team that he doesn't value the planning you did up front, and that he would rather the team spend time writing bad code than spend time avoiding it.

...Unless the Process Is the Problem

When managers start cutting practices out of the process, it's almost always because of the schedule. A deadline is looming, and you're telling him that the project is not going to make it. His first instinct will often be to "cut the fat"—find any practice that seems "extraneous" and remove it.

The fact is many practices pay for themselves. Code reviews have a cost in terms of the effort it takes to perform them, but they save the team more effort later because they don't have to spend time tracking down and fixing the bugs. The same goes for testing with respect to support costs. A boss whose projects come under schedule pressure will routinely cut code reviews and testing practices. But there certainly may be other practices that no longer make sense in the current project.

So how do you know which practices are cost-effective and which can be cut? In general, you and your project team will not have agreed to a practice at the outset of the project if that practice costs more effort than it saves. So if it's in your process, it's probably there for a reason, and the fastest way to deliver the code is usually to insist that each practice is performed. But there certainly *could* be some room for improvement in your process. It's possible that the team is over-architecting the solution: gold-plating, or solving technical problems that don't necessarily address business needs. Or they may be taking a good practice and abusing it, like reviewing every single line of code in the entire project rather than focusing only on high-risk areas.

This is where making sure that your boss has a good handle on the team's process and the project progress so far will come in handy. If you've kept him on top of the project, then you will both know exactly why a certain practice is being performed, and whether or not it is cost-effective. You won't just cut practices because they take time: you'll make educated decisions. And sometimes you'll recognize that there's simply no way to meet the deadline, no matter what you cut. But even when this happens, you'll catch it earlier. Rather than ineffectually cutting down the process, you'll be able to work with your manager, so that he can spend his time managing expectations in your organization. After all, that is his job.

Don't Compromise on Quality

There's an old saying: "Faster, better, cheaper—choose two." Many people take this to mean that the team can deliver software faster and cheaper simply by somehow sacrificing quality. They mistakenly believe quality is expensive, and it's possible to cut costs and effort by compromising on it.

Unfortunately, the picture isn't that simple. A project manager, for example, can say with certainty how much time, effort, and money have been expended, but he can't say how much quality is there. And that's why it's tempting to compromise on it. When the quality is unknown, it doesn't feel like a sacrifice to give it up. This is why it's sacrificed so often.

Understand Software Testing

Throughout the life of the software project, your team does many things to find and prevent defects. Once the software has been built, it's time to look back and make sure that it meets the user and stakeholder needs that were laid out in the vision. The goal of software testing is to make sure that the product does what the users and stakeholders need it to do. Software testers review the final product to make sure that the initial requirements have been met.

In software testing, quality is defined as "conformance to requirements." Software is built to meet requirements that define the specific behaviors that the software must exhibit. When the software does not behave the way that the requirements say it must behave, that is a defect. Consequently, software testers are responsible for figuring out whether the software that was produced by the team behaves in the way that the requirements it was built from say that it should.

Every engineering discipline defines quality in exactly this way. When an auto manufacturer provides a specification to a subcontractor to create a part for a car, that specification contains tolerances for various measurements for that part. When the subcontractor ships a box of those parts back to the manufacturer to be built into the car, any part that does not meet the requirements in the specification is considered defective. And it's exactly the same with software—any feature that does not meet the requirements in the specification has a defect.

For more information on software testing, see <http://www.stellman-greene.com/Testing>.

If your software projects follow a typical testing process, they will have several rounds of testing. During each test cycle, the QA team will uncover defects. Some of those defects might cause a disaster if users find them. To determine this, after each round of tests your manager will review the list of defects that were discovered. He then can work with the team, stakeholders, and users to decide

whether to release the software as is or have the team repair it and then rerun the tests.

When a project runs late or encounters technical problems that will lead to schedule delays, managers will often suggest that the team should simply ship the product on time, regardless of the state of the software. This usually means cutting out testing and defect repair tasks, because those are the last tasks on the project schedule. They are also, probably not coincidentally, tasks that many developers do not like performing. Tracking down and fixing bugs seems like tedious work to you and your team, which is why programmers will often support this decision.

This kind of pressure from the programming team will put any manager in a difficult position, making it tough to come to the right decision. It seems so simple: here's a program that runs and a team that seems confident that it's ready to ship. Sure, the product might be "beta quality," or the users may have to put up with a few little glitches here and there. But why not just take a chance?

The fact is that by doing this, you and your manager have put your project at risk. There is no "quality continuum." Since the quality of the product is directly related to its ability to perform the requirements that have been defined, it's important that all of the behaviors in those requirements work. There's a difference between making an informed decision about defects that have been uncovered and cutting the testing tasks because the team ran up against a deadline. In the end, you need to make sure that software testing activities—and not the schedule—determine whether the software is ready for release. If you throw your weight behind good QA practices, your project will succeed. If you take the easy route and let your manager undercut the testers, you will be taking a huge risk. Many programmers' reputations have been scarred by this practice.

There's a simple rule of thumb: if it's worth the time to make a change, then it's worth the time to test the fix. After all, if your boss had decided not to have the QA team perform the last round of tests, your team would have happily shipped the software with this current defect that's giving everyone so much grief. As Murphy's Law clearly tells us, it's the one defect that the team doesn't yet know about that can sink the project, because that's the one that the CEO will find the first time he or she runs the software.

Alter Your Tactics, but Stay True to Your Strategy

Experts on military strategy often say that the war plan lasts until first contact with the enemy. The same often seems true of software project plans: they last until the work begins. But that doesn't mean it makes sense to abandon all of your planning at the first sign of project trouble. Problems will happen on even the best-planned projects: estimates turn out to be incorrect, people leave the organization, and

technology and business needs constantly shift. When these problems occur, you need to help your manager make good decisions. And the only way to do this is to keep your perspective and help your boss keep his.

In the end, you're the only one who can really keep the project grounded, by ensuring that everyone involved has a clear understanding of the overall strategy and vision. You'll almost certainly have to make changes to the way that you and your team operate. There may be additional practices that must be implemented; there may be others that are costing more time than they're worth.

So stay informed, keep everyone in the loop, and always keep the needs of your users and stakeholders in mind. And, more importantly, make sure that your boss is doing that as well. Managing upward can be a difficult balancing act. But you know what to do: manage expectations, keep your software process in order, keep everyone in the loop, and stay committed to the quality of the software. If you do all of that, you'll find that your projects (and your life) will improve.

About the Authors

Jennifer Greene and Andrew Stellman are the authors of *Applied Software Project Management* (<http://www.oreilly.com/catalog/appliedprojectmgmt/>), and have been building software together since 1998. Andrew comes from a programming background, and Jennifer has a testing background. They formed Stellman & Greene Consulting (<http://www.stellman-greene.com/Consulting>) in 2003, with a focus on project management, software development, management consulting, and software process improvement. They have focused on projects in a wide range of industries, as well as science and academia.

Copyright

How to Keep Your Boss from Sinking Your Project, by Andrew Stellman and Jennifer Greene

Copyright © 2006 O'Reilly Media, Inc. All rights reserved.

Not for redistribution without permission from O'Reilly Media, Inc.

ISBN: 0596528027