The Seven Deadly Sins of Software Project Management

David Churchville
ExtremePlanner Software, Inc.

http://www.extremeplanner.com

TABLE OF CONTENTS

Introduction

Two out of three software projects were delivered late, over budget or without all the required features, according to a 2004 study by the Standish Group.

Software projects are distinct from most other types of undertakings in that they often involve creating something that has never been done before. Even mundane software solutions are faced with new technologies, increasingly demanding requirements for uptime and scalability, a rapidly changing job market, and changing business needs.

It's no wonder then that it's a major achievement for only one out of three projects to be completed on-time, within budget, and actually meet their stated goals.

Chances are, though, that you'd like to have a higher than 30% success rate on your projects. The last decade has seen the rise of agile software development and project management techniques that encourage frequent releases, close communication with the customer, and less ceremony than more traditional methods. These methods have been shown to increase the likelihood of project success, or at worst, to give your customers enough information to cancel the project early enough to avoid a disaster.

This guide explores some of the common pitfalls in software project management, as well as suggestions on how to steer clear of them and give your project the best chance for not just success, but for truly satisfying your customers.

Read on!

Sin 1: Trying to Define All Requirements Up Front

Overview

Software development is an inherently unpredictable endeavor. If you are starting a project today, even if you have built something very similar in the past, it is still difficult to accurately estimate the effort needed to complete the project.

Software is created by human beings, each of whom has unique perspectives, interpretations, biases, and levels of competence.

Given this, you might try to minimize your risk by at least controlling the input. You attempt to do this by fixing the requirements at the start of the project. Your customers are forced to sign-off on a set of statements that you carefully extracted over a period of months. You file away these documents, confident that you can now focus on managing the development team and build a plan.

This might work well except for one small problem...customers are human beings, too. They have incomplete visions, other things on their minds, and imperfect communication. And if that weren't bad enough, sometimes business circumstances actually change. So even perfectly and completely communicated requirements may become outdated in just a few months.

Can you solve this by trying harder to gather them up front? Is shoving a signed document under your customer's nose going to make her any more pleased about not getting her needs met?

And yet, many project teams ignore this uncertainty and insist on trying to define all of the requirements up front. This isn't just dangerous, it's actually wasteful, since you can almost be assured that a certain percentage of those requirements will be invalidated and the time you spent analyzing them will have been for naught.

So what's a project manager to do?

A More Agile Approach

An agile approach to requirements definition is to defer detailed analysis as late as possible, which is typically just before the work is about to begin. Until then, you capture requirements in the form of "user stories," which are brief descriptions of customer-relevant functionality.

User stories are not requirements; they are placeholders for the more detailed conversations and analysis that will need to happen as those stories are being implemented. As such, they need not be exhaustive descriptions of the functionality of the system, but just enough information for developers and customers to have a common understanding.

User stories can be used in the release planning stages of a project, and are very useful for doing initial estimates and prioritization of a project. When used in conjunction with an iterative development process, they can be a highly effective means of capturing user needs as they arise.

As a project progresses and the customer discovers additional functionality that she needs, you can capture them as new stories to be prioritized for the next release or iteration.

This approach lets customers be human and it minimizes the waste of analyzing and documenting problems that may no longer be valid by the time the team is ready to tackle them. The time saved here can be used to deliver more business value. And that's all you really want anyway.

Sin 2: Ignoring the Iron Triangle

Overview

Software projects, like other types of projects, have to obey the "iron triangle" of project management: **scope**, **time**, and **resources**. You can attempt to control any two of these, but the third will be affected as a result.

Scope refers to the set of features and the level of robustness of the software.

Time refers either to the actual man-hours needed to complete the scope of work defined, or more typically, an externally imposed deadline. Given a deadline, some project teams try to work "harder" to meet the goal, but this inevitably drives down quality, which has a negative impact on productivity. In other words, late projects tend to get later as the team works harder.

Resources refer to the people assigned to the project, as well as hardware, software, and other materials needed to complete the project. Resources can be increased by adding more developers to a project team, but as with the strategy of "working harder'" adding resources to a troubled project can actually make it more troubled.

While many project managers and development teams intellectually understand the iron triangle, in practice, many projects are set up as if it didn't exist.

For example, how many times have you been in a situation where the customer requested a specific set of features that were needed by a specific date? Typically the resources are limited on any project, so constraining both the scope and the time available puts the project in jeopardy even from the start. Any change in your understanding of the requirements, unexpected technology issues, or difficult to find defects will create a crisis.

Even when the project is planned out and it becomes clear that the delivery date cannot be met, some project managers insist on trying to "gut it out," which often backfires in the form of management ire or meeting the delivery date with a product of unacceptable quality.

A More Agile Approach

A more agile approach to managing customer expectations is to work closely with the team to define features, estimate their approximate size, and use this information to prioritize work.

If you already have a limited set of resources, it follows that scope and time cannot both be fixed. In this situation, you'll need to use one of the following approaches:

Deadline-driven approach

For this approach, you start with a fixed deadline. If you have to deliver by a certain date to land a major contract or avoid closing your doors, this approach is for you. Below is a step-by-step process for making this work:

1. Estimate the size of each feature that is on the table. These estimates don't need to be completely accurate, but it's helpful if they are accurate *relative* to each other. In other words, if you estimate that Feature A is 10 hours large and Feature B is 5 hours large, then A should be about twice as hard as B. These initial estimates are just to help your customers to prioritize the work, not a final commitment, and you should clearly communicate this.
2. Determine roughly how much can be done in the time available. Again, this is not a commitment yet, but just a way to put boundaries on the initial feature set scope. A quick way to estimate this is to take the number of full-time team members, multiply by the number of working days on the project, and multiply again by the number of effective hours per working day—this is usually not more than 5 hours, and is less for teams that have other commitments like operational support. *Example: 4 developers * 20 working days (1month) * 5 hrs/day = 400 hours.*
3. Using the list of features that now have rough estimates associated and the rough estimate of the resource budget available, you can sit down with the customer to "shop" for features.
4. The customer can choose whatever features he or she wants, as long as they fit into the timeframe. Again, the point isn't to commit to delivering these yet, but to force a prioritization, so that it's clear to everyone what is most important to work on.
5. Once the set of features has been prioritized, you can then divide the work into *iterations* that are limited development cycles of 2 to 4 weeks. At the end of each iteration you should be able to demonstrate one or more working, tested features. Partially completed features don't count as "done."
6. At the start of each iteration, you need to meet with the customer to make sure that the set of features you'll be working on is still the highest priority, and to see if there are additional requests that might be more important than what you are working on.

The idea is to keep everyone involved throughout the process, so that you can deliver the best possible product in the timeframe allowed. There should be no surprises or missed opportunities, since both the development team and the customer are in constant communication about what's most important.

It also will be painfully clear if the project is not progressing as expected, since at the end of each iteration the team shows only completed functionality.

The key to this approach is complete honesty and frequent communication. The structure of the iteration serves to keep the development team on target by focusing on a small number of critical features. It also serves to allow the customer to steer the project by

being able to change priorities if necessary and to give feedback on the system as it stands, instead of waiting until late in the project.

Scope-driven approach
For a scope-driven approach, the same process as above is used, but instead of choosing a deadline, the team does the following:

1. Prioritize the feature list using the same technique described above.
2. Determine the minimum feature set required for release.
3. Project a range of end dates based on the feature estimates and the number of iterations that will be required. For example, if you have 10 features, each of which will take 20 hours and the iteration capacity is about 100 hours, then it will take roughly 2 iterations to complete the work.
4. Since the numbers you are working with are not exact, it's important to communicate a range based on the uncertainty of the estimates. As each iteration is completed, it should be possible to give more and more accurate estimates of the completion date. For example, a project that is estimated to require 10 two-week iterations may take anywhere from 8 to 12 iterations based on your initial estimates. As time goes on, you can determine whether your estimation is accurate enough or if you need to adjust.

As with the deadline-driven approach, this process requires frequent communication and honesty. It's very important to set expectations correctly by using ranges of estimates initially, then gradually firm up those dates as they become more clear. A warning sign on a project would be hearing "We will finish all of these features by September 17th, 2008" when that date is year or two away.

Sin 3: Trying to Define All Development Tasks Up Front

Overview

Many project managers attempt to estimate completion dates of a project by listing all of the development tasks that will be required. These lists assume that the development team knows exactly how to implement the system and that no surprises will be encountered.

In reality, unless you are building the same software that you built last year, the team is not likely to have a good handle on what will be involved on a six-month or year-long project.

The danger here is that a project plan is created that lists these tasks, which gives a false sense of accuracy and completeness. Many project managers use techniques for buffering the time to account for unexpected events, but this often doesn't get communicated to the customer, who then believes that the detailed task lists are evidence of a firm commitment to a date.

Another problem with a task list is that it focuses on assigning resources well in advance of any work occurring. This places artificial constraints on the project team, since some activities occur in parallel and others are collaborative efforts. Allocating task assignments in advance is often a difficult thing to do when the tasks themselves are uncertain and likely to change.

A More Agile Approach

An agile approach to task management is to defer detailed task definitions until the work is about to begin on a particular feature. The best time to do this is usually during iteration planning, where the set of features to be implemented during the next couple of weeks is being determined.

Task definition is then guided by specific features, which helps to keep your activity very focused on delivering important functionality.

Also, rather than pre-assigning all tasks at the beginning of the iteration, a more effective approach is to let team members sign up for tasks one at a time. This helps to avoid one person becoming a bottleneck for the team. Having team members occasionally work together on tasks is also a good way to spread information around the team and get some insurance against team turnover or unexpected absence.

Sin 4: Setting Unrealistic Customer Expectations

Overview

Many projects start out well, but about halfway through you discover that you're not where you wanted to be. The project will almost certainly be later than you communicated six months ago.

You mention this to your customer, who asks about the new date. You don't really know, but you give an answer designed to minimize his frustration, one which you hope the development team can make.

Several months later, things aren't looking any better, and the new date you gave is now unachievable. This is the point in most projects where things go terribly wrong. Instead of honestly communicating about this, you and the development team start working harder, trying desperately to meet an unrealistic commitment.
The quality of the project slips, the team morale plummets, and even if you pull it off, no one is in shape to take on the next project. Meanwhile, your customer is already mistrustful of your dates and has started to doubt any information you pass on. Grumbling team members are resentful of your incessant demands for overtime and can sense that you're not being completely honest with the management team.

Is this really the only way to run a project? What if you could keep your customer informed without a painful consequence?

A More Agile Approach

Instead of waiting until late into the project, when there is little than can be done about it, you can and should communicate with the customer early and often about the state of the project.

Your customer deserves early warnings of potential delays or project risks. For example, before your customer tells his boss that the new payroll system will go live in March, he needs to know that two key developers left your company last week and therefore it is unlikely you will meet this deadline. Your job is to help keep your customer from looking like an idiot.

Ideally, you should sit down at the end of each development iteration to walk the customer through the completed features. This lets him understand where the system really is, and can help ensure that what he wanted and what you built are well-aligned. Meeting a deadline isn't of much value if the software doesn't meet the customer's needs.

Sin 5: Tracking Activities Instead of Features

Overview

As discussed earlier, many project teams attempt to define all of the development tasks up front, using this information to allocate resources, track dependencies, and otherwise schedule the effort.

As the project evolves, it becomes more difficult to discern what value each task has to the project. Is the task labeled "Build a Widget Framework" really something that you want to track? If it is delayed, does it really mean that you can't deliver the feature that depends on it? Could it be done without the framework at all?

Since task lists are likely to be incomplete, managing exclusively to them can be a recipe for disaster. If your team completes all of the tasks on your list, does that mean that the software is ready to ship? Would your team agree with that?

Also, you may discover that many of the tasks that were defined will turn out to be obsolete or unnecessary. There is a danger that focusing too much on the pre-defined tasks will result in wasted energy as team members attempt to meet artificial task deadlines. To have true ownership of the product, the team needs to see the whole feature as the goal, not the completion of tasks.

A More Agile Approach

Agile projects have one primary status metric – completed features (or user stories). This is a simple, clear way to communicate your status to customers, and it also has the effect of really focusing the development team on what is important.

Feature-level status tracking is a key aspect of agile development when used in conjunction with short iterations. At the end of an iteration, either a feature is complete or it isn't. "Complete" means that the feature functions correctly, meets the customers requirements and is ready to deploy, including whatever documentation is necessary.

By tracking feature progress, you can have a better conversation with your customers by showing them the features completed so far instead of displaying project plans dense with meaningless tasks. Better yet, by using iterative development, you can do a full walkthrough of the features completed and get instant feedback on how you're doing.

Sin 6: Failing to Update the Plan Based on Reality

Overview

"It is a bad plan that admits of no modification."
-- Publilius Syrus

Early estimates for a software project are usually relatively inaccurate, unless the project is very similar to one previously created by the same team. And yet, many project teams stay with these early estimates even as reality intrudes.

Part of this is due to poor setting of customer expectations, where estimates are treated as commitments instead of what they are—educated guesses. Once a customer believes that your estimates are essentially a promise of delivery, you will have a difficult time recovering from this.

Fear is the number one enemy of successful project management. Fear gets in the way of honest communication and it hurts both the project team and the customer by delaying the inevitable. Software does not lie—it is either ready or it isn't, and no amount of trickery will make it otherwise.

A More Agile Approach

Communicating estimates is a delicate task and is best done by using ranges that reflect the uncertainty, rather than specific time units.

For example, if you believe a project will take 9 months, give or take a month or two, saying that it will be completed in 10 months is probably a mistake. A better way to represent this would be to say that you'll know more after the first iteration or so, but for now your best guess is from 9 to 11 months. After each iteration, when you can see what progress you are actually making, you can revise these estimates to be more specific.

Agile projects update the plan at the beginning of each iteration. This is both to get assurance from the customer that the priorities are still correct, as well as to evaluate previous estimates to see if they still hold. Planning is therefore not something that is done once, but continuously throughout the project as more information becomes available.

By the middle of the project, the estimates should be more and more accurate, and the delivery date comes more clearly into focus.

Sin 7: Waiting Months to Get User Feedback

Overview

One critical mistake a project team can make is to assume that the customer doesn't need to interact with the software until it is "ready." On a traditionally managed project, this might mean waiting months or even years before users can review the software. A simple miscommunication can mean months of wasted effort, not to mention precious funds.

In fact, a group of people who claim to have common goals often will have very different ideas of how to best meet those goals with software. An enlightening exercise when first discussing a new software project is to get everyone involved to sit down in a room with a large whiteboard or flipchart and have each person draw a few screens that show how he or she visualizes the application. The results are often surprising in their contrast.

Even when requirements are correctly interpreted, too long of a period before user review can lead to developer resistance to changing areas of the software that aren't well received. While the feature or application may still be acceptable, it is often more complex or less robust than the customer would like, and it's now too late or too expensive to adjust.

A More Agile Approach

By focusing on getting rapid user feedback, agile techniques can mitigate issues such as poor communication, misinterpreted requirements, and poor user interface decisions. The more the customer can touch and feel the evolving software product, the more clear she can be in clarifying requirements and steering the project.

Before any software is even written, lightweight prototyping techniques can help you and your customer communicate more effectively. One approach is to draw potential application screens on paper or a whiteboard and walk the customer through typical scenarios of using the application. For remote customers, you can use interactive HTML prototypes, possibly in conjunction with Web-conferencing software that lets you both see the same computer display in different locations.

During the project, you should make software available for review at the end of each iteration to give the customer the opportunity to review progress and verify your understanding of the system. You will get better feedback if you actually demonstrate the software to your customer, rather than simply making it available to her.

Timely feedback is one of the most critical elements of project success. Demonstrating working software every few weeks is the best way to ensure that you're getting enough of it.

About the Author

David Churchville has over 15 years of experience in software development and project management ranging from Fortune 500 companies to early-stage startups. He is the president and founder of ExtremePlanner Software, Inc.

About ExtremePlanner Software

ExtremePlanner Software makes products that help development teams consistently create remarkable software by communicating and collaborating more effectively with their customers and each other. We believe that the best software is created by inspired teams working cooperatively with the values of honest communication, respect, courage, and integrity.

Our products focus on simplicity and effectiveness in order to help our customers get outstanding results without unnecessary overhead and complexity.

Resources

ExtremePlanner is a web-based agile project management tool that helps distributed software teams collaborate effectively to deliver on time and under budget. It supports Extreme Programming, Scrum and other iterative development approaches.
Learn more at http://www.extremeplanner.com/extremeplanner.html.

EasyPrototype is software for easily creating interactive user interface prototypes and specifications that really resonate with customers to ensure clear communication and save costly rework and client frustration.
Learn more at http://www.easyprototype.com/easyprototype.html.