Other Considerations

The biggest reason that estimates go over is that they didn't consider everything that went into the project. We look at the features requested, and the tasks to be done. What we don't consider is the facilitating tasks, and some of the most abstract time sinks that projects can run into along the way. This section will cover some common stumbling blocks you should be aware.

The “Mythical Man Month” Effect

If it would take one developer three months to finish a project, that does not mean you can assign it to three developers and get the same project completed in one month. In Fred Brooks book talks all about this phenomenon in *The Mythical Man Month*. A “Man Month” is a mythical unit of measurement that non-technical managers like to believe exists. They think that work is divisible among many people to reduce the timeline, but it never works this way.

To put it another way, nine women can’t make a baby in a month.

In fact, in what is a counterintuitive fact to anyone who hasn’t lived through it multiple times, adding more developers to a project that is behind schedule **increases** the delay in schedule.

If it takes one developer three months to finish a software project, it will likely take three developers three months to complete the software project. There are two primary reasons this happens:

Reason #1: Sequencing

The completion of certain tasks needs to occur in a particular order. If the designer needs the copy, they start working on their design; then you can't have the designer and the copywriter working in parallel.

Reason #2: Communication Overhead

When there is one developer working with the project owner, you can keep overhead low. The developer can use whatever task tracking system and client communication strategy that works best for them. Once you bring in a second developer, now you have a new channel of communication.

The two developers will need to organize who is doing what, and have some way of knowing what is done when. If one developer takes on the job of being the point man for the project owner, then that dev will need a way to know the progress of the other dev so he can update the client accordingly.

When you add a third developer, the real issue starts to arise. Communication channels grow exponentially. Here are our open channels now:

- Developer A <-> Client
- Developer A <-> Developer B
- Developer A <-> Developer C
- **Developer B <-> Developer C**

We've increased the amount of production time by 33%, but we've increased overhead by 50%. This increase means that when you add a new developer to a project, *you make every other developer worse*.

Teams aren't always a bad idea, just to be clear. More developers can help a project get delivered faster, even if it means more work hours involved. Let's say two developers are working 40 hour work weeks can get a job done in 160 hours of work a task that would take a single developer 120 hours. In this case, you are trading an additional 40 hours of effort for a one-week reduction in delivery.

Apply This Knowledge:

- Know the size of your team before getting started on the project.
- If the team size changes, understand that that means the estimate changes.
- Have a project management system in place, and allocate additional time accordingly. How much time does your team spend in meetings every week? How much time to they spend on other communication such as email, phone, or Slack? How much time to they spend on documentation?

The Network Effect

The issue with developer communication in the previous chapter is a negative example of the *network effect*.[2] *or congestion. Each additional developer is like adding another car to the freeway.*

Something else to consider is that adding features and complexity to a software project has **The same effect.**

Every time you add a feature to a project, you affect the cost of other features, in addition to other feature development down the road. If you built a particular kind of feature in a previous project, it's not always safe to assume to that it will take an equal amount of time in another project.

Apply This Knowledge:

- If you are using time tracking data from a previous estimate, consider how the implementation might be different this time.
- For each feature, consider each other feature that you've listed. Do they connect or interact in any way? What kind of development effort will it take to implement those interactions?

---

[2] https://en.wikipedia.org/wiki/Network_effect

There's No Such Thing as a 5-Minute Fix

Small fixes are something that tend to come up later in a project cycle. You've shown your client your current progress, some wireframes, or a functioning beta. They want "one small tweak" or "just one more thing."

I once spent three months going back and forth on revisions with a client who wanted about sixty "one more things".

It's easy to forget about all of the facilitating work that goes into making small changes in a software project. Even if I wanted to add one line of code to a project, I'd need to do the following:

- Turn on my computer.
- Startup all my text editor.
- Check out the latest version of the code from the source code repository.
- **Write the one line of code.**
- Test the one line of code.
- Fix anything that it happened to break.

- Check the code back it to the source code repository.
- Deploy the code to staging or production, possibly both.
- Update my progress in any project management, bug tracking, customer support, or any other communication software I'm using.

All this work more than five minutes. You can overcome this time sink and be more productive by batching. Instead of deploying to production every time you have a bug fix, you could roll them out in daily or weekly updates instead. Designers push for rounds of revisions instead of getting piecemeal feedback from a client.

Time-Boxed Fixes

While I'm not an advocate for Agile development, here is one key takeaway: Sprints aren't just for additional feature development. They can also be used to improve code quality or reduce defects.
In Agile development, a "sprint" is a set amount of time, usually one or two weeks, in which issues are taken from a project roadmap ("backlog") and worked on. In addition to this usual iteration cycle, there are some other types of sprints that can happen at the end of the lifecycle of a project:

Hardening sprints: The most common term. Hardening sprints are dedicated towards testing, finding, and fixing defects. Also known as a

stabilization sprint.

Release readiness sprints: A release readiness sprint is a time spent getting an application ready for deployment. For example, setting up the production server, getting live API keys, and making any last- minute changes to the codebase that are needed to get ready to be released out in the wild.

Spring cleaning sprints: These are sprints that refactor and cleanup code. This time is where you pay back your technical debt.

The Power of Time Boxing

When you ask a client for revisions, they may not feel like they are doing their job if they don't come back with any feedback. The same goes for tweaking and improving internally. There is *always* a way a piece of software can be enhanced. If there weren't, then the only full-time development jobs would be working for agencies that work on one-off projects.

These gaseous clouds of little extra things can quickly make an estimate balloon to a much larger size than anticipated. Since these tasks are so nebulous, they often get left out of estimates entirely, since we believe

there is no way to predict how long they will take. Since it can't be predicted, it has to be **proactively defined.**

Apply This Knowledge:

- If you are offering revisions as part of your project, set a fixed number of revisions.
- Include time-boxes periods to handle testing, quality assurance, deployment, and code cleanup. These issues **always** come up, regardless of technology stack or project management process used. Budget for them now.

Leave Yourself Margin

No matter how much time you put into planning and researching your project, there is always the risk of hitting a roadblock. You could get sick or have emergency show up. There will always be little things that come up. The project could be going swimmingly, and then you a phone call that upsets or distracts you. Maybe your neighbor was loud, and you didn't get as much sleep as you would like. Now something you thought would take at most two hours ends up taking three. These kinds of distractions and bumps in the road are never a big deal individually, so they get ignored, and you didn't account for it. These can end up making a project end up weeks behind schedule, and no one is sure why.

All Hours Are Not Created Equal

A common mistake is to assume that there are 8 hours of a work day. If you bill by the hour, you are likely familiar with this. But consider all the other things that eat into your day:

- An hour at lunch.
- 30 - 45 minutes of the day on a few short breaks. - An hour on email and phone.
- Set up time in the morning.

- Wrap-up tasks at the end of the day.

After taking all of that into consideration getting 4 - 5 productive hours in a day is pretty good.

Factoring in a Margin

The template sheet provided offers you a few different ways to handle this. Firstly, there is an "hours per day" variable, it is not assumed to be 8. This margin can also be used to figure out deadlines if you have a project where you may only be allocating 1 - 2 hours per day towards it.

Also, there is a field for adding margin to the project as a whole. Adding some margin is a common practice, especially when fixed bids are being made. Your margin depends on your experience. People have suggested anywhere from 30%-300%.

Death Marches Don’t Work

A *Death March* is a period when deadlines are looming, and developers are putting in lots of extra hours. They are exactly as fun as they sound. If you work at a video game or start-up company, you may know this process as “all of the time”.

The problems with taking this crunch time are well documented: Working extra hours doesn’t mean more productivity. In fact, unfocused work can lead to additional bugs and cut corners, which will take more time to fix. *Negative productivity* is entirely possible in software development.

Extra hours are not always bad. There are cases when a few extra hours or a Saturday shift can be productive. They should used and in extreme cases. The is- sue is that death marches can be habit-forming within the company culture. Teams in a death march tend to stay on a death march.

But death marches aren’t the real problem. They are a symptom of a disease. The cure for a death march is simple:

- Change the deadline.
- Alter the scope of the project.

Death marches are a symptom of **communication issues**.

How Death Marches Start

You first have to look at how the death march started in the first place. One possibility is that the estimate and project plan were not reasonable from the outset. If a project manager or sales team member promises six weeks of work and a deadline two weeks away, then the project schedule was doomed before a single line of code was written.

Another possibility is that the schedule was started to slip, but no one communicated about these slips. Maybe someone on the development team didn't want to admit that they were taking longer than they thought and feared looking incompetent. Maybe someone thought they could 'make up' the time later. So instead of talking to the owner of the project they gave a progress report of 'A- OK', and put their nose back to the grindstone.

Or once it became apparent that the original project could not be completed as promised, no one felt as though they could talk to the project owner about an increase in the timeline or a decrease in scope.

Apply This Knowledge:

The answer to all of these situations is **clear communication**. Developers need to be clear to clients and project managers about reasonable expectations. They also need to be clear about changes in scope, and when things aren't going well. If a change in regards to deadline or scope is required, then that needs to be communicated as early and often as possible. In the next section, we'll dive deep into communicating your estimates before, during, and after the project.