Extracted from:

Software Estimation Without Guessing

Effective Planning in an Imperfect World

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Software Estimation Without Guessing

Effective Planning in an Imperfect World

George Dinwiddie
edited by Adaobi Obi Tulton

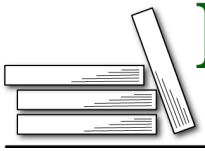
Software Estimation Without Guessing

Effective Planning in an Imperfect World

George Dinwiddie

The Pragmatic Bookshelf

Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-698-3
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—September 11, 2019

Aspects to Compare

There are many aspects you will want to consider when estimating a software development project. Too narrow a view is likely to lead you into trouble. A narrow view increases the chances that significant contributors to the schedule lie in the blind spots you've failed to consider.

When comparing future work to past experience, if you neglect to think about some aspect that is roughly the same in both cases, no harm is done. But if you neglect some aspect that differs in a way that affects the pace of accomplishment, it can have a major impact on the accuracy of your estimate. Let's look at what aspects might differ. This checklist is, of course, not exhaustive, but it will help you consider the matter more thoroughly.

Aspects of the System to Consider

The most obvious consideration when estimating system development is what's known about the intended system, itself. That is, after all, what what we're building. And it's the building of it that we're estimating. Therefore, looking at those desires, or "requirements" as some call them, is a natural starting point.

Quantitative Aspects of the System

Of course, the scope of the development is a primary concern. Ask yourself how big is the code being developed, and how much functionality is being added. Notice the things that you can count, such as the following examples:

- How many user workflows are needed?
- How many screens will that take?
- How many "logical items" will need to be stored?
- With how many other systems does this one communicate?
- How many interfaces does thie system provide to others, and how many functions per interface?

These, of course, might be approximate counts based on a naive implementation model. That's OK. This is an estimate we're making, not a prediction. Such quantitative measures can alert you to differences in size compared to your reference.

Qualitative Aspects of the System

Dig into the qualitative aspects of the system:

- How complex is the functionality being developed?
- Has something like this been implemented before?

- Are there significant interactions between the parts, or are they relatively independent of each other?

These, also, can alert you to differences in size compared to your reference, even though they don't have numbers attached.

Quality Aspects of the System

Consider the quality of implementation:

- How much emphasis should be given to maintainability of the system and to future extensibility?
- For that matter, if you're building on an existing codebase, how much emphasis was given for it?
- And how good is the development team at writing maintainable, extensible code?

People often take such quality issues for granted, but there can be a wide range of interpretations. If the attention to quality in the reference system or of an existing codebase which you'll modify differs from the current expectations for the future work, then that's a significant difference which must be accommodated in your estimate.

Internal code and architectural quality can be quite cheap if you've learned the knack. As Philip Crosby said, "Quality is free." Attention to detail quickly pays for itself, but some people don't recognize how to work that way. Having to come back and try to put in the quality after the fact can be very expensive. And building on a system made without concern for quality will certainly have a lot of unexpected work.

Aspects of the System Context to Consider

The way the system relates to the systems around it can also have a major impact on how much time and effort development takes. The relationship of the system being developed and the people and organization that interact with or otherwise depend on the system has an effect, too.

Constraints of the System Context

Consider the constraints placed on the development:

- Are there decisions that are assumed and can't be changed, such as aspects of the architecture or deployment configuration?

Implicit expectations can easily blow your estimate out of the water. Better to ask these questions now, rather than be surprised by them later.

Non-functional Expectations of the System Context

Consider the "-ilities," the characteristics that cut across the functional requirements:

- What is the need for scalability, or the immediate and long-term needs in terms of users and data?
- What is the expected throughput and service level agreements?
- How much safety factor should be included beyond expected needs?
- What level of system availability is needed?
- How responsive does it need to be?
- How reliable does the system need to be?
- Is there a need to degrade gracefully in the face of problems outside of its scope, such as other systems being down or communication bottlenecks?
- When something goes wrong, how will people know what went wrong? Can you give them clear and relevant information? Can you store information to be examined later? Can Customer Service deduce what happened when talking with the user over the phone?

The expectations surrounding implementation quality can be widely varied and are often implicitly assumed rather than explicitly discussed. Mismatches in expectations can have major impact on the suitability of your estimates.

Security Expectations of the System Context

Consider the expectations of the system security in its intended environment. Sometimes these dimensions haven't been fully explored when you're asked for an estimate. That's OK; they can probably be deferred. If you're practicing lean product discovery, you don't want to expend energy on bulletproofing a feature until you've validated that customers will use it. In other situations, you may be surprised by people saying "of course it needs to be bulletproof."

- How secure does the system need to be, and against what threat models?
- Does the system need to be auditable? To what level of detail?
- Does traceability data need to be stored and, if so, at what detail and for how long?
- Is there personally identifiable information that needs to be protected from disclosure to others?
- Are there privacy laws that govern the system?

Such requirements are often overlooked prior to the approval of a project. It could be a significant amount of functionality that's invisible to the nominal user, and will take development time.

Usability Expectations of the System Context

Consider the expectations of user factors. This is another category of expectations that are often not mentioned explicitly until later, when someone outside the project complains.

- How stringent are the usability requirements?
- What are the accessibility requirements?
- Is the system required to conform with Section 508 or other regulations protecting the disabled?
- Does the system need to be internationalized to support multiple languages and cultures?

Such concerns can add a lot to the effort, especially if the development team isn't experienced at meeting such demands. There is a broad range of potential expectations.

Priority of Expectations of the System Context

Consider when these contextual considerations become important. Early releases may not have the same needs and expectations as others. Perhaps you can validate the core functionality with a limited audience for earlier feedback.

- Do you need to include these at the start?
- Will it be sufficient to patch any issues raised?
- Can you iteratively add these after each function is developed?

Consider how much support needs to be implemented for operations and customer service to detect, identify, and analyze problems during operation. This is another category of often invisible requirements. Neglecting these functions can save a lot of development time, but greatly reduce the long-term satisfaction with the system.

All of these contextual demands are generally under-discussed at the beginning of a project. They are hard to bring up, also, as asking "do we need such-and-such" will often trigger the "kitchen sink" response. "Well, of course we need it. If we can think of it, put it in." Bloating the project with expectations that have not been thought through thoroughly can blow more than your estimate.

Aspects of the Development Context to Consider

The details of the development process have a huge impact on development speed. In my experience, rushing into development without proper preparation is a major cause of systems development taking much longer than anticipated. Setting things up for success is, of course, the prudent plan. If that's not

within your power, then being aware of the potential issues is necessary for an understanding of what might slow down the development process. Perhaps that awareness will also aid in improving some of these aspects.

Familiarity of the Development Context

Consider the familiarity of the functionality and proposed implementation:

- How much of the scope is well understood and how much is vague or new?
- Do you have a solid background in the business domain?
- Are you fluent in the implementation technology?
- Do you even know yet who will be doing the implementation?

All forms of novelty impose a learning tax on the development process. Giving the process of learning short shrift will undoubtedly lengthen the amount of time required.

Relationships Surrounding the Development Context

Consider the relationship with the customer or manager requesting the system:

- Are they congenial and easy to please, or nit-picky and opinionated?
- Are they willing to engage throughout the project to clarify the requirements as they are addressed or new questions come up?
- Are they likely to want "the kitchen sink" when presented with options?

You can easily spend significant time convincing a stakeholder of some essential fact. Or, you may be constrained to doing something the hard way because you can't convince them. On the other hand, a good working relationship with the customer can save a lot of unneccesary work and avoid needless rework.

Building the Customer Relationship

Sidney called Ryan to talk about the Empire Enterprises call center situation. Sidney's first question was "When can I have the new call center?"

"We've got a few people exploring some new-to-us technology to support it. We've got some people examining the current system to figure out what it currently supports. Our rough order of magnitude estimate for a replacement system based on the time it took to build the old system seems way out of whack to me."

"When will the whole thing be ready?"

"We calculated it might take about three years. I think that's too long, but we need a better sense of what the 'whole thing' entails. Reverse-engineering the current system is a slow way to determine the requirements. It's also likely to pull along current errors in implementation, plus create some new ones. We can surely do better than that, but we'll need your help."

Ryan looked at him suspiciously. "What sort of help? Why can't you just build what we need?"

"Building custom software isn't like assembling a known product. You've seen how sometimes you don't get what you expected. Neither one of us likes it when that happens. But if you'll work with us, we can order the work to give you some value earlier, make sure we're on the right track, and take care of any problems as we go, when they're still small problems."

"But I've got a Customer Service department to run. Developing software isn't my expertise or responsibility."

"I know. But handling customer service calls isn't our expertise, either. We don't know your operation like you do. Why don't we meet next week some time. You can bring one or two of your most experienced people and I'll bring a couple good analysts. Let's spend an hour or two and see what we can come up with. It's a small price to pay that might pay off big."

"Let me check our schedules, and I'll get back to you."

Rewrites don't always have to have all the features of the system they're replacing. Often there are features that are little used, or will be obsolete when other new features are added. Resisting the temptation to provide feature parity offers a potential solution to such rewrites.

As this story shows, business people easily assume that the software development organization will do whatever it is that the business people will later find they want. If you're working from that assumption, then it makes sense to concentrate on your own work and wait for the solution to be delivered to you. As anyone who's been in software development awhile probably realizes, this is a recipe for repeated cycles of building something to have it rejected and rebuilt.

Business: *Bring me a rock.*

Development: *Here is your rock.*

Business: *No, not that rock. Bring me a different rock.*

Even when the relationship isn't as unhelpful as this, the nature of the relationship between those asking for the software and those building it can have a huge impact on the time it takes to successfully complete it. Consider the following aspects:

- Is there one customer voice to be satisfied, or multiple constituencies?
- What is the procedure when there's a difference of opinion on the requirements?
- How clear and unambiguous is "done" for each requirement?

Any fuzziness in understanding the requirements will surely slow things down. In the worst case, gaining an understanding with one constituency may result in work that must be redone when another constituency disagrees.

If you find yourself in the middle of a battle between two powers who want different systems, you may never complete it.

Also consider issues that might arise as you try to untangle uncertainties and miscommunication.

- Do you have easy access to determine the answers to questions that will arise in the future?
- Are you dealing directly with the decision maker, or with a proxy?
- When you ask a question, how quickly can you expect a response?

Duration depends on effort plus waiting. Proceeding without waiting can waste even more effort, and it leaves a lot of work in progress. The open questions will slow you down more than you might imagine.

Effort also depends on duration. It takes effort to get back up to speed after an interruption or delay.

Consider the organizational components to the rate at which work can be done.

- Will there be interruptions in the work?
- What else will be going on at the same time?
- Will there be task switching between projects vying for attention?

Consider all of these aspects, and any others that come to mind when comparing the future work to past experience. They are all ways in which that experience could differ and have a significant impact on the time and effort. Are they multiplicative or additive effects? How big? Handle these the same way we saw in Multiplicative and Additive Adjustments, on page ?.

As you consider these aspects, particularly Aspects of the System to Consider, on page 3, you have a choice to think of the system as a single whole thing, or as composed of smaller parts.