Estimation

Intelligent Forecasting, Project Control and Client Relationships Management

Dimitre Dimitrov

Toronto, Canada

First Edition, 2017

Editors: Roger Hardnock, Robert Acton
Illustrations: Dimitre Dimitrov

ISBN 0-0000000-0-0

To a friend

Estimation, Software Project Estimation

Intelligent Forecasting, Project Control and Client Relationship Management

Table of Contents

Why This Book?

Have you been on projects where halfway down the road it seems increasingly unlikely that you will finish as desired, but you can't put your finger on it and simply push through with a growing resentment? Have you been in meetings where scope discussions get increasingly difficult and depressing, ultimately sucking the energy out of everyone instead of enabling them to move forward with certainty and determination? Wouldn't it be good if a confident assertion for the project's ability to deliver is made as early as one or two months into the effort? What additional value and quality of working relationships can you generate if the time for estimation, tracking, and change management is slashed by a factor of 10, while simultaneously providing clients and team members with true peace of mind so they can focus on other important business activities? This book provides a practical tool that will help with all of this - it is a how-to book. But ultimately it is about the relationships we develop during projects and the appreciation of our colleagues and business partners.

It is 2017! Yet estimation and forecasting in the software development industry is still considered mystifying at best, and an archaic and obsolete concept at worst. Reliable forecasting continues to present a challenge for many teams and software development organizations. The practices associated with the two predominant software development methodologies are inadequate. Methods related to waterfall development are notoriously bad for long-term forecasting because they encourage too much information processing too early, and have a tendency to skew reality into a Gantt chart. And methods that relate well to agile software development are not as notoriously bad, however, mostly because long-term forecasting is avoided altogether. This is problematic in many cases because it pushes important decisions too late in the project, adds unnecessary stress on people's relationships, and ultimately diminishes the chances for successful projects.

Where forecasting on projects utilizing agile software development is done, it is with short-term commitments - an iteration or a few at best. Such small-span commitments have minimal value for business people who want to look a year or two ahead. When teams practicing agile software development commit to a longer term delivery, they suffer many of the same issues as people working on waterfall projects do - the desired "project targets" get missed and there is lack of confidence throughout the project about what will be delivered or when. Senior businessmen and businesswomen lose the capacity to trust the delivery teams and organizations. People who work together for years remain at a distance and never build true partnerships. Many business teams and development teams are openly adversary.

One reason behind much of these issues is the inability for meaningful long-term commitment. And while reliably providing accurate and precise software estimates is close to impossible, reliably providing accurate and precise forecasts is not. This book shows you how to do this and commit to a purpose early. A satisfactory solution can be reached with a small investment in understanding people's problems, and through the adoption of simple statistical principles. The security and reliability that you will be able to contribute to projects will improve your team's performance, morale and motivation, and will have a positive impact on long and healthy client relationships.

Who Is The Reader of This Book and What Can You Find In It?

"Estimation, Software Estimation" is for anyone who wants to have a fresh and adequate outlook on the process of software estimation and forecasting, and how these activities facilitate the conversations and relationships among people. It is directly relevant to the roles of scrum masters and project managers, and provides practical tools for intelligent project control. The book is also valuable for business people who want insight into the type of problems that delivery teams face, and for programmers and other delivery team members who want to gain an understanding of the project manager's day-to-day challenges. While life experience as a member of a software project is useful for quickly recognizing some of the situational nuances, this book will appeal to curious people who are early in their careers.

The described forecasting method relies on the ability of a technical team to deliver working software with consistency. Many of the technical practices enabling such delivery have been perfected and popularized by what came to be known as agile software development. Thus, in practice, the forecasting and control methods align well with the operational mode of many teams delivering software with an agile development methodology. However, this is not a book about agile software development. If you and your team are practicing solid software development and do not label yourself as agile or extreme, then this book will be valuable for you too.

Estimation by itself is of limited use. We do it to forecast and plan. Forecasting and planning are in turn done to improve the chances of making good decisions and taking appropriate actions. A forecast, does not improve a project's performance. It is only a tool for visualizing that performance. This book shows us how to use an intelligent forecast for making timely decisions and applying measured project control for steering towards a valuable goal.

side note

Before going any deeper in this book, we need to establish that estimation and forecasting are two different things. This book is more about forecasting and control. However, the term "estimation" has been misused for so long, that it has become the normal term for describing what amounts to forecasting. Estimation is only the initial guess about the size of something. Forecasting is the activity of processing input data, including the estimates, and formulating an intelligent prediction.

Where this book ventures into a longer treatment on a subject, or establishes an explicit view on the meaning of a concept, it is not done with the intent to educate the reader, rather it is done with the intent of providing context so the reader can align their understanding with the ideas in the book. For more information on some of the relevant concepts, you can reference the literature provided in the Reference section at the end. Feedback is welcome at estimationbook.com.

The method presented here scales well over any size of a software project, provided the organizational structure and the development methods are such that continuously delivering working software is achieved.

Small projects, consisting of a few developers working for a couple of months are harder to control through the approach provided by this book, but you can still find useful ideas to guide your future discussions with clients and teammates.

For those who work in the context of legacy systems, the applicability of the approach depends on whether the system is healthy or not. For projects within healthy legacy systems you can apply the concepts almost directly. For projects within legacy systems with compromised codebase integrity, you can cherry pick ideas that make sense in your context and apply them when there is an opportunity.

This book will be of particular value to people who provide software project delivery as a service to other companies or departments within larger organizations. Teams utilizing modern development techniques and automation have often achieved the technical excellence required for reliable forecasting and intelligent project control. These teams often serve big corporate clients who are not well dispositioned, or are downright allergic to the concept of no estimation and no long-term planning. By providing a viable alternative to the dreaded work breakdown structures and bottom up estimation, this book will help delivery teams speak the language of business people without sacrificing software development performance or quality of delivery practices.

Finally, ”Estimation, Software Estimation” will provide clues for achieving a mindset which allows us to bridge an important gap - to see each other (delivery teams and clients) as human and to appreciate each other’s and our own needs and abilities.

#NoEstimates

Since this book has the word estimate in its title, it seems appropriate to address the idea of #NoEstimates.

#NoEstimates is about searching for alternatives to estimates. The method described in this book might actually fall in this category of alternative to the broadly accepted methods of estimation. However, it can be also seen as a logical extension of traditional estimation. I will leave it to the reader to make their own judgement as to whether the method described here is alternative or not - if that categorization matters to the reader at all.

Exploring alternatives is interesting when context is maintained. Still, much of the discussions in the #NoEstimates blogs and articles move haphazardly from project delivery to product development to software development. These are vastly different contexts. The book you are reading is about estimating, forecasting and controlling software *projects*.

In contrast, the concepts of business value and guided product experiments are primarily about product development. Software quality, continual integration and automation are primarily about software engineering. All three - product delivery, software delivery, and project delivery are complementary and when done right each draws on the strengths of the others. While product delivery and software delivery present just as interesting and important problems, they are not the main subject of this book and are only examined where it facilitates the discussion of intelligent project forecasting.

How Is The Book Organized

The concepts in the book build upon each other. Concepts covered earlier are needed to fully appreciate those covered later. Ideally the reader will take on a sequential approach and read the book from front to back.

That said, many of the sections in the book can be read without having read the previous sections or chapters. Some readers might be familiar with a few of the ideas presented in this book and could dive straight into a later chapter. The sections within chapters are kept brief, making it easier to locate topics when reading non-sequentially.

Chapter 0 initializes a minimalistic context for the rest of the book.

Chapter 7 describes a fictional, but realistic software development project where the estimation and forecasting techniques described here are exercised. It can be read independently or used as a reference.

The Tidbits chapter is a compilation of ideas and techniques that are important for supporting the practical application of this method.

Frequently Used Pronouns, Collective Nouns, and Terms

A few words appear repeatedly throughout the text:

"People" is often used as all the people on a project, which includes both client and the software delivery team. Or, "people" can sometimes refer to only one of these groups - either the client team or the software delivery team.

"We" is mainly from the perspective of the software delivery team, but it is also used from the perspective of all the people involved on a software project. Occasionally “we” stands for project managers and scrum masters only.

“Business people” and “client” are used for the people who request the software solution. Sometimes these people are external clients and sometimes they are from another department within the same company.

"Software developers" is generally meant as the programmers, designers and testers from the delivery team, but is sometimes used to describe all the people who contribute to the delivery of a software project, including the scrum master and project manager.

“The team” and “project team” - these and similar expressions are used to mean all the people on a project, including people from the client team who are actively engaged with the project.

“Delivery team” is basically “the project team” less the “business people”. If there are 5 scrum teams working on the same project then all of these are considered “the delivery team”, since the major discussion in this book focuses on estimating and forecasting complete projects and not individual team’s performance.

“Accurate” is generally used to mean both accurate and precise, since we are only interested in the pragmatic side of forecasting.

“Project” is a collaborative enterprise for achieving a particular valuable goal over a set period (and within certain limitations).

“Project management” is meant as the activities related to decision making and application of control over project parameters in a way that is most conducive to the project’s success. It also includes the activities related to securing the people’s wellbeing within the project boundaries.

“Project manager” is generally meant to describe a role and not a job title. Anyone who is participating in project management activities on the project can play the project manager’s role. A scrum master can often be a project manager, but so can a team lead, a director or a manager. Occasionally, the whole team can be the project manager.

“Project performance” captures the project’s progress towards success in terms of delivered functionality and expanded effort. It is not exactly the same as team performance, although both are certainly related. It is important to keep in mind that the definition of success might shift during a typical project as people adjust to reality.

“Invert, always invert”

~ Carl Jacobi,
19th Century mathematician

Chapter 0

These are the beliefs and assertions which give meaning to much of the discussion in this book.

"*Certainty or safety is a basic need*" - At some level every person needs safety. At the most elementary level a person needs physical and psychological safety. This is true even when a person engages in an inherently risky endeavour like starting a new software project.

"*A software team can deliver continuously within a controlled productivity range*" - Modern delivery teams have mastered proven software engineering practices and have repeatedly demonstrated that their productivity can remain within constant limits throughout the duration of a project. We will take "constant limits" to mean that if there are two comparable pieces of functionality then the team will complete these by expanding comparable amounts of effort. And we can expect this to hold true throughout the project (with the possible exception of the first few weeks when people are picking up speed).

"*Project control is more important than record keeping*" - On a software project, the primary responsibility of the people involved is to take actions with the intent to control and steer the project to success. Book keeping is of secondary or ternary importance. The benefit of forecasting is to pull certain important decisions earlier in the project's life. We are not forecasting to prove something right or wrong, nor are we estimating to keep a record and hold people accountable for the estimation numbers they produced.

"It is only worth forecasting when there is ability to act" - A forecast on its own does not change the outcome of a project. There must be a real possibility that we make control decisions, which lead to measured and timely actions, and change the project's parameters. If such readiness for action does not exist on a project, and will never exist regardless of new knowledge, then forecasting becomes useless.

Chapter 1

Let's have a short stroll and meet the people who are involved in a software project. It is important to look at their typical problems so that we have confidence in the adequacy of the forecasting method and the approach to project control that we develop later in this book. It is the people and their problems who establish the solution's adequacy.

The People

The Client (a.k.a. Business Person). These are the people who generate business solutions. They take business risks, sometimes having to power through considerable fears and doubt. They hustle and discover valuable things that other people need and are willing to pay for. Their ideas and resolve for action accomplish the visionary work needed for the creation of something new. Sometimes they need software tools in order to move their business ideas forward. Even when they take on a project for building a software solution, they are not software developers, rather they are business developers.

The Developer (a.k.a. Programmer, Tester, Designer). These are the people whose brains and hands perform the implementation work and bring a software solution into existence. They are skilled in software techniques and in making computer systems perform complex things. They understand how users interact with software. Many of these people can keep large amounts of information and abstractions in their brains. They enjoy seeing these abstractions materialize in the form of working software and sharing this miracle with the rest of us. A certain element of playfulness and doing things purely for the sake of having them done can be found in many software developers.

The Project Manager and the Scrum Master. These are the people charged with facilitating and organizing various project activities. They help other people make decision and thus have an effect on how a project is being controlled. Key activities affecting decision making on a project are forecasting and scope control. While project managers and scrum masters do not themselves produce estimates, they play a central role[1] in how these estimates are used into planning, and subsequently how the plans are enacted throughout the project's execution. The facilitation work that project managers and scrum masters perform is critical for the overall project tone and the quality of the work environment.

The Product Owner and the Business Analyst. These are the people who have the skill of converting the ideas of business people into a format consumable by software developers. Product owners can also make confident decisions on what is valuable and what is less valuable in a software solution. Business analysts, and many product owners, help describe, and to a degree define, the solution that brings about the capabilities desired by business people. Their work greatly affects the quality of information that developers get to process, the volume of implementation work and the end product suitability.

The Manager and the Team Lead. These are people who are tasked with getting the job done. They manage teams of developers with finite capabilities and less finite potential, and are responsible for creating the environment where developers can do their work well. These are also often the people who provide initial estimates for software projects. They may delegate the actual estimation to their teams, but they remain personally accountable for the information that gets communicated to clients. When it comes to project sizing, Managers and Team Leads often use experience and gut intuition for producing an overall project estimate that is safe and sufficiently accurate.

The Problems

A problem is something to be worked out or solved, but it is also something that allows people to self-validate and to ultimately grow as human beings. The way a person approaches their own problems has a noticeable effect on their quality of life and those around them. The way one person approaches other people's

[1] Technically, scrum masters are absolved from the responsibility of planning out the full project. They only focus on helping the team work better and remove impediments. However, I consider client discomfort and uncertainty to be major impediments for the team's work. As such, the scrum master is responsible for minimizing them. One of the things that can be done is to have a better project forecast, reliable long term commitment and a working plan for effective project control.

problems has a noticeable effect on the trust and relationships they will be able to establish.

Making a Decision. This is one of the main problems for businessman and businesswoman. Making decisions is how they make their living and create the ecosystems for other people to make their livelihood too. The actual mechanics of deciding is not what creates difficulty for them, but it is the build-up to that decision which costs them time, effort and comfort. It is a complex problem since the business context involves various bits of information and many unknowns that are all in flux. Business people deal with this complexity and handle the unknown by taking risks and venturing into business experiments.

The challenge that software projects present is the high variability of just about everything - which basically translates into more uncertainty. Some feel uncertainty is what business is all about.

Making a Promise. This is one of the main problems for makers. Software developers are makers. People who make things with their hands love seeing the products of their labor being used by other people. These people enjoy providing solutions to other people's problems. A maker wants to say "I will solve your problem. Just tell me what it is."

Whether the promise is explicit or implicit, it is real and both the maker and the client appreciate its power. How to make the promise carry through is the problem that makers wrestle with.

Making a Plan. Making a plan is a problem for anyone who is accountable for seeing a project to an end. On software development projects this is often the person tasked with project management responsibilities.

One way or another - every project ends. What activities happen between the project's start and the project's end, and what is the result of these activities, ultimately has a crucial impact on whether the project ends successfully or not. Consequently, identifying the activities with the best chance of contributing to a successful outcome is of primary interest for a planner.

Typically, there is an expectation, or at least a wish, for a reliable plan to be in place sooner rather than later through the life of a project. Even when the "plan" is to simply work through the mountain of challenges, the project manager is expected to guide the effort on a path of success and to have sufficient foresight into why the chosen path is the one leading to success.

side note

Who cares!

It is easy to say "Don't promise", and "Don't plan". It is less easy to say "Don't make decisions", although some people do say it in a roundabout way. Even though these approaches look seductively simple, and there is a hint of bravery in choosing them, none leads to a satisfactory solution. They invariably lead to unwelcome compromises that people need to accept at the expense of comfort and happiness.

In the context of a single project, managers have a similar problem to planners, they want to see the project on time. And business analysts have a similar problem to makers, i.e. they need to see their ideas materialize and produce valuable results.

The Desires

An adequate solution should solve a problem and align with desires. So let's make a few more generalizations. You will notice this as a pattern that applies to estimation and forecasting too - by making careful simplifications we can get good enough understanding of a complex problem.

To make decisions on reliable information. For business people this is one of the primary desires. By the nature of what they do, they need to make so many decisions that any opportunity for making a quick and clear decision, based on little but reliable information, is welcome.

The next best thing is the knowledge that reliable information will be available at a defined moment in time. When businessmen and businesswomen are faced with uncertainty they can tolerate it for a while, but it helps them a lot if they have an idea of what to expect once the waiting is over.

Why is this important? It is important so that we can properly sequence the input we are providing to people who are making decisions. We need to appreciate what suits them best at a given moment in the life of the project. This makes communications much more meaningful and allows us to move through seemingly difficult situations with ease.

To be able to work. Makers and artists, which software engineers are, enjoy working. They like being useful and spending time tinkering with whatever happens to be in their field of interest. What they typically don't enjoy is to deal with something they cannot perceive to be real, valuable and true. They markedly dislike the situations where they are the originator of things with questionable worth.

For makers, it is preferable to describe a complex problem in a complex way, rather than sacrifice the truth and provide simplistic and untrue answer. Makers can bend a little and deal with uncertainty, but only for short periods of time. They prefer to spend their time making things.

Why is this important? Because it is important to understand that software developers detest estimation, and forecasting by extension, since they can not be proven to be true. They are only a guess. And for makers a guess represents little value. For this reason, we need to be sensitive and empathetic to their dislike when we need their input and cooperation. When developers see that we understand the binary unsustainability of our own request, they will oddly be more willing to help. But it is also important for another reason - when developers see that our forecasting efforts are ultimately designed to provide them with a more sensible environment for work, there is a material improvement in the relationship's dynamics.

To be able to apply control. For people who make plans and who are responsible for delivering a project, like project managers, team leads, and scrum masters, it is highly desirable to have control over how things roll out towards the project objective. Of course a project manager is not judged by their ability to strictly follow a rigid plan. Rather, the project manager is ultimately valued for their ability to deliver a satisfactory project, changing plans if necessary, and even steering in the absence of a ratified plan.

Guiding things along a known plan of action is typically easier than making all the right calls in real time without the benefit of planning and anticipation. A plan provides a useful reference and an opportunity to rehearse some scenarios in advance. It puts us in anticipatory mode and not in reactive mode of being. In this way we can roughly gauge if things are panning out well and we can set interim course direction which helps us move through obstacles without getting distracted by too much fear and unnecessary considerations. The more control a planner can exert on how work is being completed during the execution of a project, the more likely it is the end result agrees with the plan and, by assumption (please see the side note), with the expectations and wants of the people affected by the project.

side note

Project Management? So 20th century!

Waterfall is a method for software development and project management, which relies heavily on thorough and exhaustive pre-planning. It has come to be that Waterfall is a bad thing in many cases, mainly for not being able to adjust to reality. Planning and software project management, as practices, have been associated with Waterfall for so long that many people treat them as equivalent. We need to separate them though, because they are not the same. There are ways to plan and to manage a project that are dramatically different than what Waterfall has established as a standard.

The Manifesto for Agile Software Development says - we (the software developers) value responding to change over following a plan. Suggesting that following the plan might be the better thing to do sounds like a contradiction to the manifesto.

However, there are two assumptions, or rather oversimplifications, which are baked into this particular postulate of the Manifesto for Agile Software Development. The first is that we value responding to change only when we have assessed that the change merits a response. We are not merely responding to any change that comes along. The other is that we value responding to change over following a *static* plan, not over following a plan in general. A plan can be revisited and adjusted (even within the constraints of a contract). When the plans adapt to the relevant changes in reality, we can confidently follow these plans, while simultaneously responding to change.

Of course here we are talking about effective project control - steering the project reliably into producing the desired outcomes. For example, having power control over people's overtime is not effective control in this sense, while minimizing context switching by suggesting or enforcing a smaller WIP (work in progress) rule, or by improving feedback time, can be considered an application of adequate project control.

side note

What Control?

In this book, when we talk about control, we are talking about *project control*. And we are explicitly not talking about control over people.

There are three primary project controls we can manipulate:

- **Scope** - controls the "what" of the solution
- **Effort** - represents the power we apply towards building the solution
- **Duration** - represents the time we have available for finishing the project

By adjusting each of these controls within the project envelope, we can affect the project's progress and ultimately can drive the project towards desired objectives. There are other project aspects that can be recognized as distinct secondary controls. They have direct and indirect impact on the primary controls, and they are also very important in their own right for managing the less tangible outcomes of a project. These are the Environment, Software Quality, Metrics, and Value.

Environment, the well being and collaborative capacity of people, can be treated as a secondary control. It almost directly converts to Effort - motivated people deliver more effectively. In this sense Environment can be considered part of the Effort primary control, and by improving the environment we increase the available effort that can be expanded towards the project goal.

Software Quality, the well being and capacity for change in the code base, can also be treated as a secondary control. However, for most teams this is only a theoretical control since the quality that the team can attain is constant (and maximum) within the envelope of a single project. Lowering the quality is of course possible, but it can not be considered a control since it doesn't make sense. In any case, improving software quality, if it can be done sustainably within the project, also translates to Effort because once the team is working at an improved quality level they get to expend less effort for achieving a comparable result.

Metrics, the health of the adopted development processes. Driving towards more controlled processes improves the predictability of events on the project and the likelihood of a forecast being close to the actual outcome. To the extent that an improved process can be proven to facilitate better productivity, we can consider it an effective project control. However, we should not forget that individuals and interactions come before processes and tools. Enforcing rigid processes will backfire when creativity and thinking are the primary activities of people (which is the case on software development projects).

Value of the end product. Prioritizing functionality, so that we first complete the more valuable pieces, is a critically important derisking technique. Prioritization is a variation of the Scope control. Occasionally people will discover valuable functionality that has not been previously recognized as an objective for the project. Pivoting for value represents *product control*, not project control. However, the value of the end result is so crucial for the project success, that the project should accommodate changes to scope where this has been deemed the correct course of action. If the change in scope can not be contained within the current project envelope then the whole project needs to be reframed.

However, it should be noted that within the context of a given project the appropriate, and sufficient value is assumed to be guaranteed by the project's definition itself. A project starts with a specific goal, and it is expected that this goal has enough value to justify the project.

With forecasting we are seeking insight on where and when to apply *project control.*

Summary

There are a few typical roles on any software project. Sometimes these roles might be fulfilled by people whose daily roles are different. For example, a developer might be a maker primarily, but can double as a project manager and planner. It is less likely to see a business person play the role of a maker and deliver solid software code, but it is not unheard of.

Clients, developers and facilitators (scrum masters, project managers, team leads) have various problems and desires. Most of the time they prefer to do what they do best. People don't mind occasionally spending time on things they do not consider their primary interest, and they don't mind temporarily hanging in suspense, but they seek the things that make them comfortable and happy. They desire some form of certainty and autonomy.

Outside these generalizations lies a diverse set of highly nuanced human wants, fears and relationships. But the bulky characterizations we discussed in this chapter define the landscape on most software development projects.

Chapter 2

In this chapter we look at simplicity and explore why is it good to keep things simple. We will go over a few concepts which allow to view complex matter through simplified but correct models.

A model is a composition of ideas which help us understand a subject that the model represents. A correct model is one which introduces little or no skew to the particular aspect of interest, i.e. there is a reliable mapping between what we understand through the model and what we would see if we were to look directly at the subject we are observing. A model can provide significant simplification and still be correct for a given purpose. For example, if we are considering shipping books through the mail then modeling a book as a solid box with some weight might be a decent simplification.

An oversimplified model distorts reality in a way that makes the model unacceptable for what we are trying to accomplish. An overly complicated model is too hard to work with and is not worth the knowledge it provides - even if it's a correct model. For example, if we model the book as simply a "unit" it might be oversimplified. Consider: "I will ship 4 units." - it doesn't carry the necessary information for size and weight. On the other hand: "I will ship 4 doohickeys for reading, each consisting of two 8″x11″ flat unbendable cardboard surfaces hinge-attached longitudinally along the 11″ edges and encompassing a set of 250 collated flexible paper sheets suitable for printing" is overly complicated, useless, and probably wrong, even if we could derive the weight of each doohickey.

When looking for simple solutions it is important to keep in mind that we still want a true representation of the reality. Sometimes, to stay true, we need to combine a few simple concepts, and even build a more complex model - as long as it is not significantly more complex than it needs to be, the solution can be considered simple in the context of the specific problem.

Why is simplicity important in the context of software estimation, forecasting and planning? One of the reasons is that these activities do not represent a primary interest for any of the people on the project. Business people want to make decisions and move on. Developers want to be working and creating software solutions. And project managers want the project to be rolling, and they want to be able to control it towards a successful outcome.

And while everyone on the project understands the importance of a schedule, it is not the schedule or the plan itself that makes anyone happy. Thus, it is best if this aspect of the project planning[2] is approached in a simple and painless way.

It is useful to examine simplicity on its own before we try to capture it within the specific techniques for estimation, forecasting and planning. This will allow us to confidently accept the validity of the method discussed later. Whence this chapter.

What's Wrong with (overly) Simple Answers?

Simple answers are very much okay and desirable when they reflect a similarly simple reality.

But software projects are rarely simple. Thus, providing a simple answer to almost any question about a software project risks oversimplification.

For example, let's consider the simple question, "When is this project going to end?" This can be easily answered by producing a calendar date - July 2nd. A simple answer like this leaves too many unanswered questions which are implied in the original inquiry. Is the project going to be successful? Is everyone going to feel it was a successful project or only some people? Is the project goal project going to be met fully or partially? Is there anything else, that needs to happen after this project ends, for it to be a useful project?

It is unlikely that someone on a project cares only about the end date. The reality for which people care is complex and it is also constantly changing. Producing overly simple answers in this context is not adequate. But sometimes people get lazy

[2] It is important to distinguish between the portion of planning, which takes care of the schedule, and project planning in general. Project planning encompasses a diverse set of business activities which can be immensely interesting to many people. We are not talking about that planning here and we will limit ourselves to discussing the part of it which deals with estimation, forecasting and scheduling.

and want to “nail” the problem, with a silver bullet[3] - a single metric, or a single best practice, a single simple answer. And instead of simplifying the matter they only introduce uncertainty, increase the likelihood of a misunderstanding, and ultimately reduce the chances for a successful outcome.

What’s Wrong with Complicated Answers?

Complicated answers do not instill trust. They require too much intellectual investment to be understood and they are only adequate when everyone is deeply interested in the subject matter in question.

When producing a project forecast we need to be careful with exposing the underlying complexities. People will increasingly doubt the method with each layer of complexity.

What people need is a simple and fast process - producing simple and trustworthy answers.

Complicated answers are not always unwanted. Some of the speculations we make during forecasting depend on non-trivial relationships among things. We want to hide these when possible, but if someone is explicitly inquiring about the source of our reasoning, we should be able to dive in and provide satisfactory answers regardless of how complicated the reality around them happens to be. In a scenario like this, maintaining simplicity will frustrate people more than inadvertently over-complicating an answer.

Simple Constructs that Capture Complex Reality

Range. Range is the more adequate alternative of a singular value in almost any project context. For example, if a client asks “How long do you think this project will be?”, an answer of “11 months” can be suspicious, while the equally simple answer of “Between 10 and 12 months” can be satisfactory and more adequate.

[3] “No Silver Bullet — Essence and Accidents of Software Engineering” https://en.wikipedia.org/wiki/No_Silver_Bullet

Ranges allow us to say things that are sufficiently true. These statements generate all the trust needed without exposing the underlying complexities and uncertainties.

Of course, if we answer “Between 6 and 16 months”, we are just as correct (or more) but the uncertainty increases unacceptably. The client will have the same doubts as if we hadn’t provided an answer at all, only now they will also question our ability to adequately understand their problems. The anser “Between 10 months and 20 days, and 11 months and 10 days”, which can be correct and very precise, triggers similar suspicions.

To build a meaningful range we need to know two things:

- What is an adequate accuracy for the listener. This can also be called “acceptable error”. It is a value that represents how far off the true answer someone is willing to go and still consider the result good.

- What is a meaningful and sufficiently true value that the range should encompass.

Often, guessing is acceptable replacement of knowledge, so we don’t really need to know the above two, but only to be able to guess them with some level of comfort. Try this next time you are challenged to produce a seemingly simple answer, especially when it is about estimates and dates - instead of a number, produce a range. If people still insist on a singular answer then use “most likely” and pick a value within the range. We talk about probabilities later.

Conditionals. Conditionals help us when there are a few possibilities that would make for an unacceptably wide range if we were to wrap them all into a single range.

Sometimes an answer can be dramatically different based on various factors. Instead of producing a single range encompassing the many possible outcomes, it is better to have an answer that represents the disparate possibilities.

For example, when talking to a client, instead of saying, “This project might take between 8 and 12 months,” we can say, “This project will take between 10 and 12 months if we add Chris to the team. And it will take between 8 and 10 months if we add Avery."

By using conditionals we provide the client with a clearer picture at the expense of minimal extra complexity. The end result is a simple reality for the client.

If we spare the conditionals and provide a wider range, say "between 8 to 12 months" from the example above, we overload the listener with a different type of complexity - they need to consider whether the range is acceptable for them in its entirety. It may be that the project must end in less than 10 months. The client can wrongly assume that it is more likely for us to be on the upper end of the provided range, which is 12 months, and they can decide not to do the project with us. We were technically correct but didn't provide the needed information.

Boundary. Setting boundaries is an important technique for simplifying the reality and for staying safe. It is the emanation of responsibility. When we stay within reasonable boundaries, the people we work with get the healthy message that we are taking measures for our own safety, and as a result they tend to see us as more trustworthy. However, let's quickly consider what a reasonable boundary is and how does the nature of the work change the meaning of reasonable.

People often say "Don't drive beyond your headlights". This is as valid in software project context as anywhere else.

One of the difficult things when communicating ideas is to make sure that other people understood accurately what we had in mind. For example, when we say that the estimates show it will take 10 months to complete the project, it is likely that people understand this as "The project will take 10 months to complete." But what we said is that an estimate we made is 10 months, not the (actual) length of the project.

Let's see how setting a boundary can expose the important limits in our statements and help with this problem. We may say this instead: "We only have looked at the next two modules with enough detail to speculate on effort cost and duration. Based on what we have estimated, and on all the assumptions we made, we think it will take us 1.5 months to complete this. We have identified a few risks as well. If you want us to discuss the assumptions or the identified risks, it can help us all stay more aligned through the next 2 months. If the projection for these two modules turns out correct, we will be comfortable forecasting the whole project at 16 to 18 months.”

The boundaries we set in this statement are:

- Only part of the system has been assessed
- There are assumptions, so we only feel safe within them being true
- There are also some risks

Now, it’s true that the statement is not twitter friendly[4]. But it sets boundaries within a relatively concise articulation, and these boundaries make it more likely that the client’s understanding of what we said is closer to what we meant. A listener might need some extra time for processing it, and we might need to furnish help, but this extra time is in the order of minutes. If we spare the few minutes here and try to be short, we risk people working with incorrect understanding for months.

This is important because when we allow the discrepancies in people’s understandings to accumulate for long periods it drains their trust and will eventually exhaust it - people become convinced that there is no chance for common understanding. When this happens towards the end of a long and complex project, there might not be enough will-power left to discuss issues rationally and make collaborative decisions that work for everyone. Instead, it is likely that people start blaming each other. Someone will invariably say “But you said 12 months! We are now only two weeks from that!” - estimation and forecasting have turned into tools for assigning blame and controlling people, not tools for controlling projects.

Let’s now see how the meaning of reasonable boundary can change based on the essence of the work. Consider the driving through the night analogy again. While we are driving the vehicle, it is reasonable to only stay within the headlights. It might be even more reasonable to stay within half of what the headlights cover. But let’s imagine that we stop driving and take a short break on the side of the road. There is nothing wrong with envisioning how long it will take us to the next little

[4] Many people these days take pride in emulating Steve Jobs and emit ridiculously succinct messages (in emails or otherwise), only to then have other people perform inordinate amount of juggling and mental trickery before finally arriving at the “correct” rendition of the intent in the commanding message. We need to be careful when pretending to be Steve Jobs!

town, so we can find a restaurant to eat, or how long it will take to cross a continent, if that's what the trip is about.

When the nature of our work is to envision the future, it becomes unproductive to maintain the same boundaries that make sense when we work on the immediate tasks propelling us towards that same vision.

When the future is unknown, we can still maintain enough limiting conditions - for example, we can say, "Provided we maintain the same pace, we will deliver the first phase of the project in 6 months, so 9 months in total. Based on the high level specifications it seems the second phase is similar in size. It is likely to take a similar amount of time, but this is a speculation which I'm not comfortable with at the moment."

We can share our fears and not push through by force and commitment only. Fear is a type of boundary too, and we state that we are not willing to venture too far beyond it. This way we stake a claim into the unknown and still maintain boundaries that make sense to us. Opening up with our fears takes courage. And we show others that we are both brave and smart. It is important that we are brave at work, and it is also important to stay responsible. As we get more data we can sharpen the boundaries and engage with commitments without fear.

Void. Void is the construct signifying the absence of something. The importance of this construct is that it allows us to not have to invent information about something that is simply not there.

When we can intelligently deal with uncomfortable situations we are creating trust. It is much more responsible to acknowledge an (ideally temporary) inability to provide information than to supply fake data masquerading as information. Whether we work for other people or we risk our own money, being responsible is the behavior which creates and builds trust the most. Being irresponsible can destroy trust quicker than any other misstep, even when it doesn't cause material loss.

Business people can handle void fairly well if they know when information will be available. They can also handle cost associated with obtaining information sooner. The cost might be in money or in increased risk. This allows us to say: "I cannot provide meaningful information for the target date at the moment, but if we start now, with the team we considered, I can provide an outlook with 90% certainty in two months, and with 70% certainty in four weeks."

Business people are capable of managing this type of situation well, and provided we can supply the additional information as promised, void is more meaningful than something that we only wish to be true.

Probability. This leads us to the next construct - probability. Because it is such a central notion in estimation and forecasting, it has its own section in this book. But let's quickly sketch it here as a tool for simplifying the reality, and talk more about its other properties in the next chapter.

Probability is in a way similar to a range. With a range we guarantee the value is within the upper and lower limits. With a probability we avoid providing that guarantee, but provide a measure of how likely it is that the actual value matches some target instead.

When it comes to software estimations, probabilities do translate to a range almost directly. For example if I say, "I'm 80% certain the project will end 10 months from now" this can be translated like "The project will end between 10 and 12 months from now." Where if I say, "I'm 90% certain the project will end 10 months from now" then this can be interpreted as "The project will end between 10 and 11 months from now".

It is interesting to note that, when we make software projections, we talk about things taking longer than expected and not shorter - "I'm 80% certain that the project will end 10 months from now" rarely means the project might turn out to be 8 months. There is an implied "at least" in front of the number of months - "I'm 80% certain that the project will end *at least* 10 months from now."

Summary

Software projects are complicated. However, there are communication tools we can use to simplify things considerably and still have a correct understanding of the relevant issues. These tools are common structures that make it easier for us to comprehend problems, to find and share options, and make decisions.

People have different demands for information and different tolerance levels to simplification. We need to learn how to use these simplification tools in ways that fit organically with everyone on the project - clients, developers and managers. If we can help others feel comfortable we can secure a better environment for effective collaboration.

Things need to stay as simple as possible, especially in the context of estimation and forecasting. But sometimes we need to think beyond simplicity in order to capture it truly. We are discussing a few slightly more complex phenomenons in the next chapter and this gets us ready to start on the core of this book.

Chapter 3

In this chapter we touch on statistics and probabilities. We will look at a few interesting applications of statistics and will focus on how they affect decision making, because the main purpose of intelligent forecasting is to facilitate decision making and project control.

Probability

What is probability? What does it mean to you? For example, what does it mean if I say that you have 80% chance of winning $100 when you invest $100?

It doesn't mean much as a number on its own. It doesn't tell us whether you will win $100 or not. It only starts meaning something when we put it in context of other things. For example, are you willing to take on the 20% probability that you lose $100? Do you only have $100 available? Are you only allowed to bet one time, two times, or more? Is it your goal to win the extra $100 in the first place? The answers to these questions provide real meaning to the probability number.

Probability is a number that helps us decide if we want to try something with the intent of obtaining a certain outcome. If we are not interested in the specific outcome, the probability number is of only superficial interest. But if we do care about the outcome then this number starts having a more material meaning.

When we talk about a singular event, probability represents a binary reality - something either happens or it doesn't happen. The probability number informs us how likely are we to win or lose. If we are willing to take the risk of losing we might proceed. If the risk is too high, or the potential prize too insignificant, we might chose to not proceed. But regardless of the likelihood, if we decide to proceed, we still either win or lose.

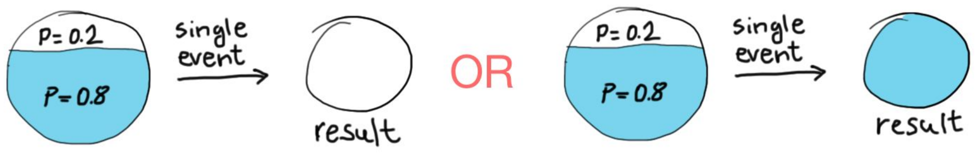


Fig. With a single event there will be only one result and although the blue result is more likely, there is no guarantee there will be a blue result.

When we work with a stream of events, we observe something else - the probability number starts describing how frequently the desired outcome occurs. It is now almost guaranteed that the desired outcome will happen - we just don't know how many times exactly. As the number of events gets bigger, the guarantee becomes stronger, and so does the likelihood of the actual number of successful outcomes being closer to the probability number.

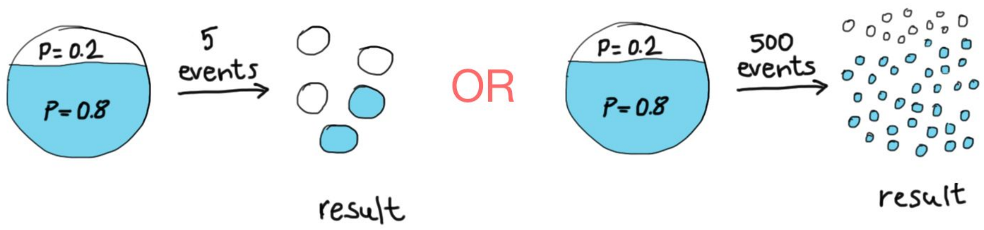


Fig. At 5 events the actual number of blue dots might not represent 80% of the total. At 500 events the actual number of blue dots is very close to 80% of the total.

When we apply a probability number on multiple independent events, things line up to what is known as normal distribution curve. It is a line that describes what is the likelihood of things happening as we move along a range of probable states.

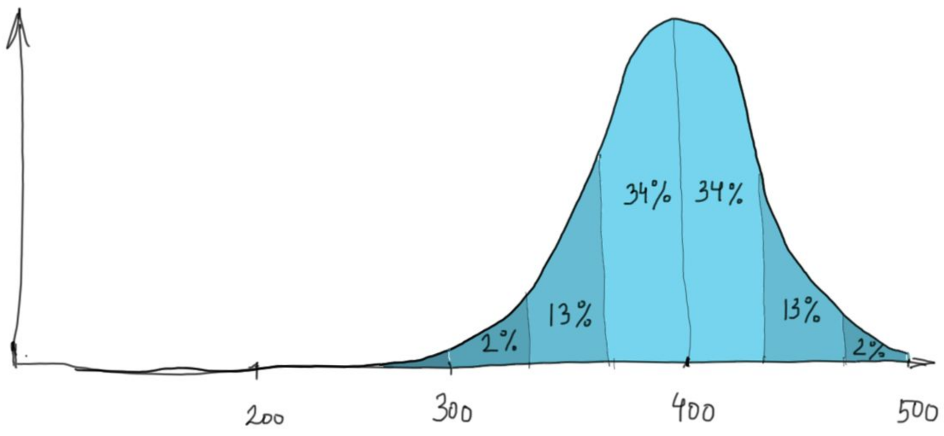


Fig. If we ran 1000 independent experiments with P=0.8 for a blue dot, and with 500 samples in each experiment, and if we measure the actual number of blue dots per experiment, we would have a result similar to the graph above. In 68% of the cases the number of blue dots will be between 370 and 430, and in 95% the number of blue dots will be between 330 and 470.

In a normal distribution, the most likely outcome is called "mode" (400 is the mode above). Almost all, 99.6%, of the possible outcomes fall within within the ±3 standard deviations from the mode. A standard deviation is a span on the dimension of interest, which span helps statisticians discuss the probability distribution easier. Each shaded area above is one standard deviation. Approximately 70% of all outcomes fall within ±1 standard deviation from the mode, and approximately 95% fall within ±2 standard deviations from the mode. These numbers might seem contrived, but it turns out they apply over a very large set of natural phenomena.

Let's say we have a team of 2 developers and have estimated (guessed) a project at 1 ½ years. Let's say we have information to think that the absolute best we can do is 1 year, and based on other assumptions we think that 2 ½ years is the worst we can do. We can build a cumulative probability curve describing how likely the estimates seem based on what we know at this moment.

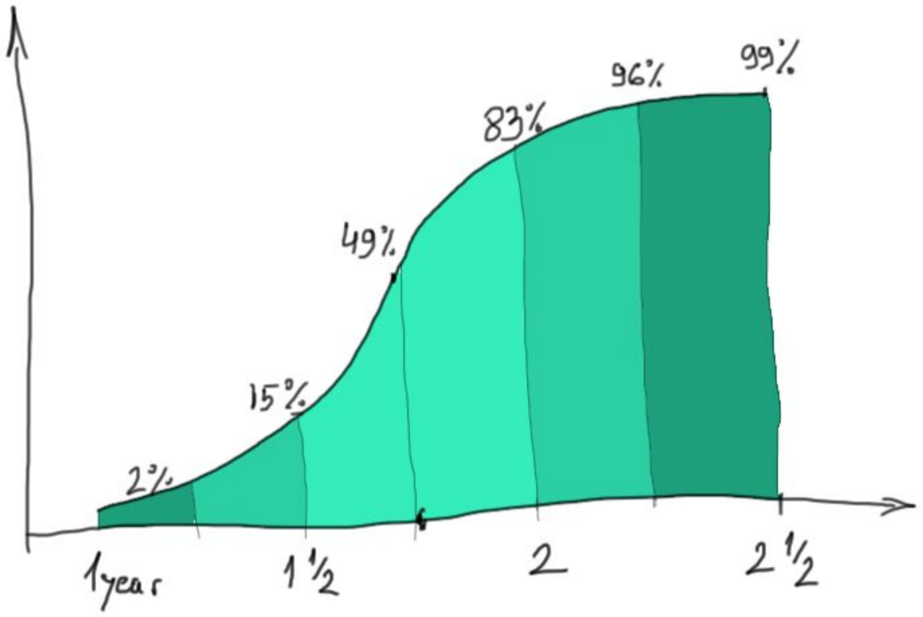


Fig. Cumulative distribution curve indicating about 80-85% probability that the project will be less than 2 years, and only about 15-20% that it will be less than the estimate of 1 ½ years. .

Even with a grossly oversimplified model like this, we can start making some crude decisions and we can improve communications on the project.

Getting Accurate Information Based on Imprecise Inputs

The Central Limit Theorem tells us that the sum of many random and independent variables is approximately normally distributed regardless of the specific distributions of the variables. This is of great significance, and is a powerful tool we can use for project planning, because projects are such a sum of large number of variables. True, they are not always independent, and they are not always completely random, but it is a good enough approximation that we can use to bypass hefty sets of complexities.

…

Control, Lack of Control, and Precision

The statistical controls we are discussing here can be rather loose when there is a low number of sample variables on which to model the distribution. For example, if we have only delivered 3 or 4 chunks[5] from a system, and we are trying to use the available information to forecast the remaining 97 chunks, then we cannot apply statistical controls yet. Our decisions will be poorly informed. We need to work with a sample of at least 20-30 variables, and ideally more than 50, before statistical principles start to work reliably for us.

What this means in practice is that we cannot approach the management of a project purely as a statistical problem.
...

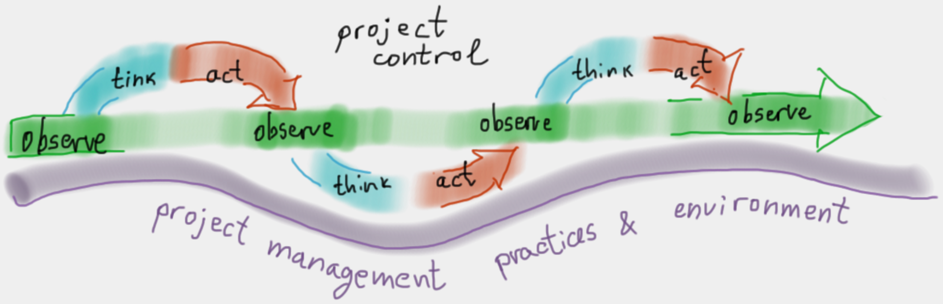
[5] A chunk can be of any size and is not the same as a user story or a functional specification. A chunk represents a set of functionality which we have decided to estimate together. We will talk about resolution and splitting the whole scope in chunks later. For a large project a single chunk can represent 3-4 weeks worth of work for a single person or a programming pair, and for smaller projects a chunk can comprise 4-5 days worth of work.

Chapter 4

...

side note

The essence of project control is *observe, think, act, observe.* It is a variation of the PDCA cycle (Plan-Do-Check-Act) of many continuous processes. The essence of project management, and planning, is to secure the environment where this type of control is feasible.



...

Estimating the work. Now that we have the estimation ranges defined, we need to break the known scope of work into pieces, and start estimating. We estimate in days worth of effort per single track[6], i.e. if the developers had the day exclusively for delivering working software - programming, designing and testing. If we observe that most chunks get in the S and M buckets we can just keep going. Some pieces will end up being XS, and some will end up L or even XL, but the bulk of the pieces should be in the S and M ranges.

[6] Sometimes a track might be a small team. I worked on a project where two programmers would couple with a single tester. The three developers would be working almost exclusively together. On that project we estimated the functionality that the whole trio could complete in a day/week worth of work.

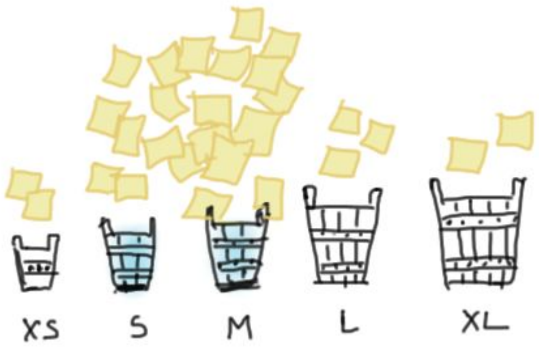


Fig. Aim to have chunks split in a way that makes them suitable for the S and M buckets. It is okay to have a few spill over into the other buckets.

…

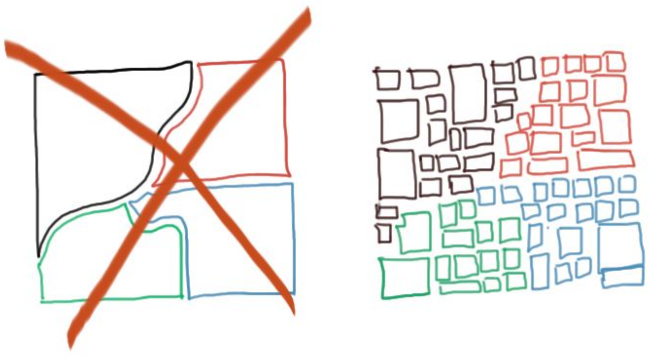
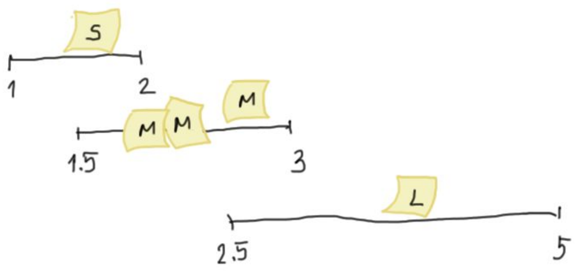


Fig. Do not ask developers to provide high precision estimates for intricate functionality based on detailed specification. Ask them to provide high resolution estimates even if each individual estimate is of lower accuracy.

Estimating at a very high resolution is wasteful. We need to pay the cost of business analysts and developers looking at too many details, and scrum masters or project managers having to track too many pieces of data. It can slightly speed up the forecasting when we start collecting data, and it can provide a (false) sense of certainty, but it costs more than the value it contributes. We should work at a resolution that makes sense and be careful every time we lead people into thinking we are working at too fine of a precision. Most likely we can not achieve that precision for the final forecast[7], and no one needs it - people only need an adequate guidance towards the objective.

...



$$\text{lower} = 1 \times 1 + 3 \times 1.5 + 1 \times 2.5 = 8$$

$$\text{upper} = 1 \times 2 + 3 \times 3 + 1 \times 5 = 16$$

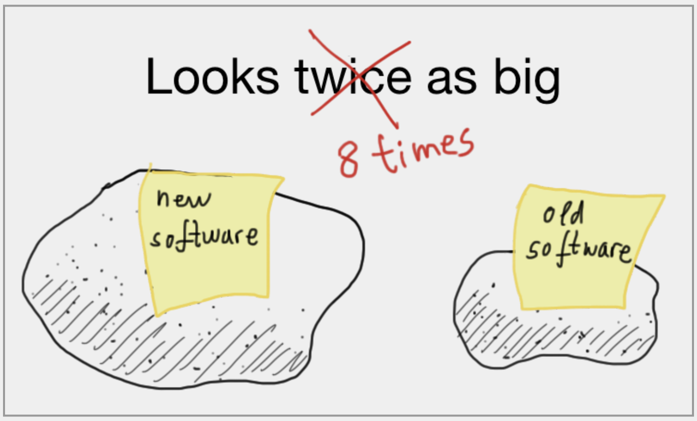
Fig. We can calculate the lower and upper estimation limits for the whole project by summing the lower and upper range limits of all pieces. The numbers here can be days, weeks, or months worth of effort.

Keep in mind that this is the implementation effort. For duration range we need to adjust for calendar time.

side note

Beware of the relative estimation technique which is discussed in many books on agile software development. It looks something like this picture

[7] I worked with a new CTO some time ago and he announced that we were going to start estimating and delivering all projects with a precision of +/- 1 day. He was reasoning that if the airlines are able to schedule transcontinental flights with precision of minutes, we should be able to deliver 3-4 month long projects with a precision of 1 day. The frequency of deployment to production was once every three weeks because of dependencies with marketing campaigns and with other departments. Even if we were able to forecast within 1 day of the actual delivery, it would bring small practical value.



and says that a "stone" that looks twice the size of another "stone" is twice as big. However, every schoolkid in 10th grade can tell us that this is not true. A stone that looks twice as big as another stone will be 8 times larger, because a stone is three dimensional, and two raised to the third power is eight ($2^3 = 8$). In software, the work is often multi-dimensional and the effect of this visual "illusion" can be even larger. (Some of the typical dimensions are: business rules, software architecture and design, database access, performance, security, usability, accessibility, visual design, automation and documentation.)

When initially estimating the chunks of work, we would do better if we looked at each one separately and give it our best shot of placing it in the right bucket, without comparing it too much with chunks we have already estimated. This way we don't inadvertently compound the estimation "errors".

The technique of relative sizing is adequate when applied on an iteration level estimation in "story points", because at that time developers are intimately familiar with the context. Story points are not useful for initial project estimation and forecasting, but they serve a different and very important role. We will talk about it later in this chapter.

Adjusting for calendar time. It is wise to not confuse effort estimates with calendar time. Such a confusion is a naive mistake, but sometimes people do it when rushed to produce a convenient answer. When we ask developers whether something takes 2-3 days, we are really asking whether it is 2-3 days of implementation work (a.k.a programming), not whether it is 2-3 days on the calendar. If developers need to attend meetings for 3-4 hours a day, the effective time for implementation work becomes 2 hours per day, and a piece of work estimated at 2-3 days might take 8-12 days on the calendar, i.e. 2-3 weeks.

One way to translate between effort and calendar time is to add certain buffers that make sense in the particular situation. Meeting time is only one of the things we need to consider when converting effort estimates to calendar time. On a longer project people need to take vacation, which might be up to 10% of the calendar time. We can apply this buffer without asking people for actual vacation plans[8]. There are many other activities that do not directly contribute to software. (There is a longer discussion on this type of safety buffers in the "Navigating Issues" section of Chapter 6.)

Multiplying the optimistic and pessimistic end of the effort range by 1.5 might be an honest thing to do when converting to calendar time as long as we can explain what goes into the extra .5 (50%) that we are factoring in. We can also produce a "realistic" estimate by using PERT calculation, or by locating a point reasonably spaced between the optimistic and pessimistic limits.

$$\text{lower (effort)} = 8 \rightarrow 12 \text{ calendar}$$
$$\text{upper (effort)} = 16 \rightarrow 24 \text{ calendar}$$

$$\text{Optimistic} = 12,\ \text{Pessimistic} = 24,\ \text{Most Likely} = 20$$

$$\text{PERT} = \frac{O + 4 \times M + P}{6} = \frac{12 + 4 \times 20 + 24}{6} \approx 19$$

Fig. PERT formula for realistic estimate is E = (O + 4xM + P)/6, where O is optimistic, M is most likely, and P is a pessimistic prognosis. (The effort-to-calendar conversion ratio is 1.5)

...

[8] Treating people with respect should be considered a project control. By planning around people's needs instead of having people adjust their needs according to a forced plan, we improve the chances for a wholehearted commitment throughout the duration of the project.

The Other Two Pillars

The Central Limit Theorem works very well for the application described so far. But it is not sufficient by itself. There are two other crucially important principles that must be satisfied for us to be able to forecast intelligently. They are "Sustainable pace of work" and "Done".

Sustainable pace of work. We often see software development teams, who work in an agile style, estimating user stories in story points. The number of points delivered within an iteration is called as velocity. Story points and velocity don't relate directly to the estimation or forecasting for the whole project. One reason for this is that they change meaning throughout the project, as they are affected by what the team learns each iteration. Another reason is that sometimes we don't have enough detail to estimate everything in points. Even if we had enough detail, it would not make sense to spend time understanding features and dependencies at the level needed for story points.

However, story points and measuring velocity, are of paramount importance for the forecast's worthiness. Their value is in setting a sustainable pace for the team. A team not working in a sustainable way cannot deliver a project that can be reliably forecasted. One of the three principles on which the method of intelligent forecasting is based is that the team must be capable of working consistently, and story points are the tool for ensuring this consistency.

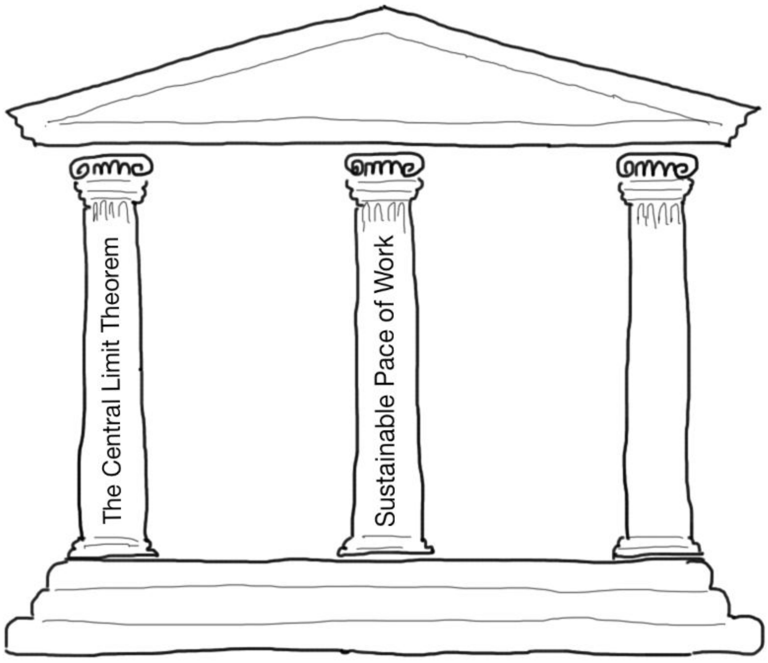
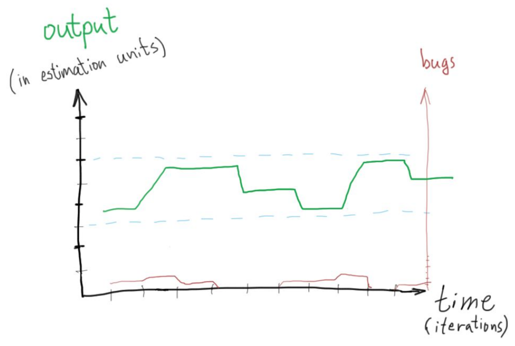


Fig. Statistics alone is not enough to support intelligent forecasting. The project team must be capable of sustaining an optimal pace of work throughout the duration of the project.

Sustainable pace does not necessarily mean a constant pace, as teams typically need time to reach their optimal pace. Teams can also innovate and accelerate. During innovative periods the pace of delivery might suffer at first and then it can pick up again. There is also normal fluctuation based on the work complexity. As long as the team's productivity remains within a satisfactory bandwidth we can consider this mode of operation sustainable.

If there are short bursts of output followed by deep troughs of endless bug fixing and zero or negative architectural progress then the team is not capable of delivering in a sustainable way.



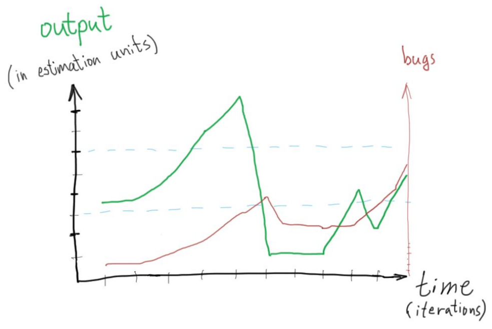


Fig. The diagram on the left shows sustainable work, and the one to the right shows unsustainable work. Even if the average output ends up relatively equal for the depicted period, the project on the right is too volatile and it is also accumulating bugs, which will eventually drive it out of usable power.

Engineering techniques like automated unit testing, test driven development (TDD), refactoring, continuous integration and continuous delivery (CI and CD) greatly contribute towards a team's capability for sustainable work. These practices are not the domain of iterative delivery models, and if a team is practicing them proficiently they can maintain sustainable pace.

If these software delivery practices are absent, and if regression issues consume progressively larger bandwidth, we need to account for it in the forecast. Regardless of whether we use scrum or we follow a big waterfall plan, we have to be honest about the team's capabilities. Deteriorating code base and regression issues can render the team's productivity to zero quickly.

Done. Done is a concept I first encountered in a formalized way when working with Scrum. So I attribute it to Scrum. Wherever it originates, probably in XP, the effect it has on our ability to intelligently forecast, plan and manage the project is binary - i.e. if we cannot work within a definition of Done[9], we cannot intelligently plan or manage a project.

[9] Definition of Done is a set of conditions that the team determines as sufficient for guaranteeing valuable software. For a more thorough description please see here: https://www.scruminc.com/definition-of-done/

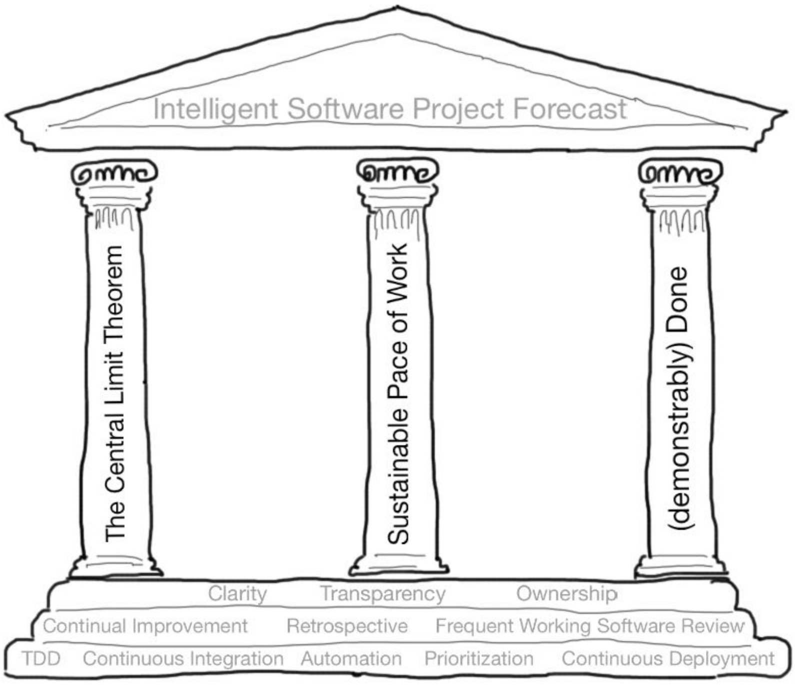
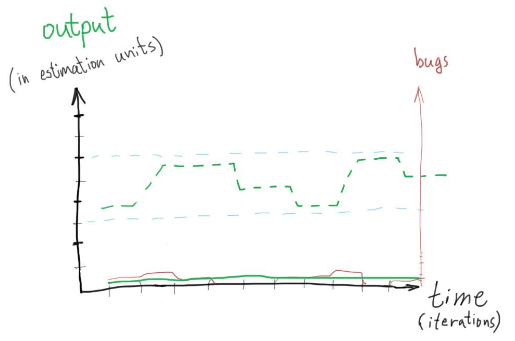


Fig. "Done" is the third pillar to support scientific forecasting. The project team must be capable of disciplined software delivery within desired and agreed upon parameters.

"Done means coded to standards, reviewed, implemented with unit Test-Driven Development (TDD), tested with 100 percent test automation, integrated and documented."

Scrum Inc.

If we are delivering code, but we cannot claim that a specific piece of functionality is done, then for all practical purposes we should not include the effort we expanded towards this code in the progress tracking. And if none of the stories can be claimed as done, we should not account for any of the effort expanded - i.e. even if we labored for 1 month we have to account for zero delivered scope.



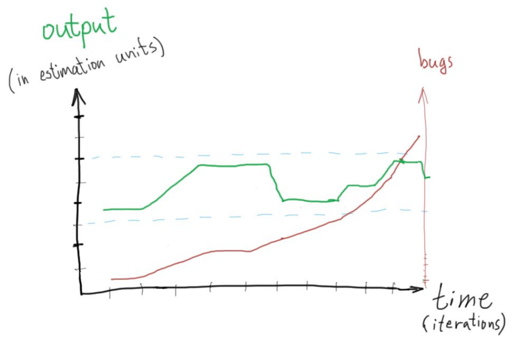


Fig. When we can not claim Done then the useful output is imaginary. The diagram on the left shows a project pretending to be delivering consistently, but without anyone validating that the software is Done. The bugs might be underreported since the software is not being tested or demonstrated to clients. The diagram on the right shows a project with accumulating bugs, but people still insist that the software is Done. Ideally people should agree to not claim Done when there are uncontrollable functionality bugs, or they will get disillusioned further into the project.

It is often the case on waterfall projects that people claim 80% readiness on some functionality, only to have this same status reported week after week without the ability to identify any single piece that is 100% done. On such projects the assessment for completeness is expected by a developer. The 80% readiness level claimed by the developer is not suitable for forecasting, since it is not validated by a business person or a user. Project managers and business people on such dysfunctional projects usually say that they need 100% complete functionality before engaging with verifying it, thus leaving the developers in a perpetual state of uncertainty.

Please note that this is dramatically different than having small pieces of functionality claimed as 100% done and confirmed by a business person. In this case we can still say we are 80% done with a larger component of work, but it actually means that about 80% of the work towards the large component is done. We can reasonably expect to have about 20% work left. When we asked the product owner to assess the completeness levels of work items (in the Mapping activity earlier), it was based on "demonstrably done" functionality, not on how many lines of code developers believe are needed before the code is done.

Working within a definition of done is more difficult on waterfall projects. This is primarily because business people wait for the hand off at the end of the development phase. However, the technical practices of test automation and refactoring, which enable a team to work within a definition of done, are not exclusively reserved for iteratively run projects. Teams using these techniques can get very close to the benefits of working within a definition of done, even if they can not secure the continual interaction with business partners.

The two pillars that support predictability and confidence - “Sustainable pace of work” and “Done”, go hand in hand. If a team does not maintain sustainable pace they will start cutting corners and step outside the definition of done; if the team does work in a sustainable pace but starts casually going out of the definition of done then they will very soon become incapable of sustaining the pace - they will get bogged down in problems. “Done” gives us confidence about the validity of present claims, and “Sustainable pace” provides for predictability of projections.

Software delivery practices like test driven development, automation and continuous integration, and product development practices like relentless prioritization, continuous deployment, and continual usability testing greatly facilitate a mode of operation where teams can consistently work within a definition of done and deliver at a pace which is sustainable and commercially viable.

•••

Chapter 5

Adjustments

In software projects, like in most situations involving human relationships, it is worth the effort to provide meaningful information to people so that they gain situational awareness and make intelligent choices. Complete awareness is of course neither possible nor needed, but a small effort can go a long way towards significant improvements and avoiding major contentions. Uncertainty of 30%-50% or more is likely to create unneeded pressure and turmoil on any project. Many people operate at these levels of uncertainty routinely and for long periods. On the other hand, verbally maintaining that we work at 10-15% uncertainty, without being able to support the claims with data, is as unnerving if not more so. If we can find a workable method for bringing the uncertainty level reliably to 10-15%, supported by meaningful data, it will spare the team and clients valuable energy, unnecessary worry and loss of time.

We were already careful to not confuse effort estimate with calendar time, and we autocorrected the "plans" based on estimation input by tracking the actual completion times. We are now going to look at a few more adjustments to the forecasting model that will further enhance our understanding of the project's progress and the accuracy of the resulting projections.

Available Thrust

Now that we have a forecast diagram we need to figure out how to translate the information into intelligent decisions in order to adjust the project's progress closer to the path we want. Plotting the diagram and looking at the forecast is only the observational part of the effort. Applying project control is the real purpose of this observation. Remember, once the end date of a project is near, we have very limited ability to make significant changes. Consequently, we want to apply adequate control as early in the project as possible. The more stabilized the project performance is in its early and mid stages, the greater the probability of arriving at the desired destination.

I was delighted when I read that the burndown chart in Scrum is based on Jeff Sutherland's experience landing fighter jets[10]. I, too, enjoy flying and have always found flying and project management to be close in many ways. When I was developing this forecasting technique, one such important similarity that took a central spot in the model was *available thrust*. The available thrust of an airplane is the force with which the propeller or jet engine can push the airplane forward. The available thrust determines the maximum angle of climb the airplane can perform.

[10] "Scrum: The Art of Doing Twice the Work in Half the Time"

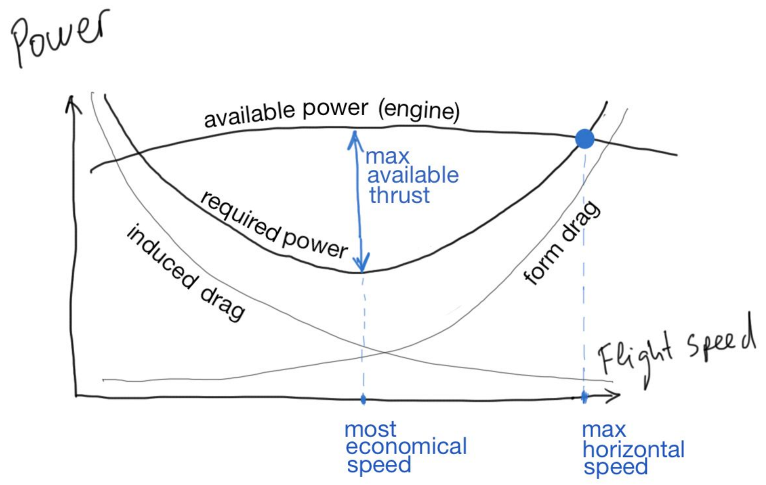


Fig. There are a few forces acting on an airplane in flight and this graph captures how the required power for sustaining horizontal flight changes with speed. When plotted together with the available power, some important speeds can be determined. The difference between required and available power defines the excess available thrust.

At first, when trying to produce a projection, I was working with "percent allocation" of people on the project. However, allocation alone did not properly reflect the contribution people were making to software and it also did not properly reflect the discrepancy between the actual team composition and my plans.

Since I was working as a scrum master I knew about all the intricate issues people were having on the project. All these impediments were taking away from the time that people had available to devote specifically to writing software. Some impediments were random or temporary by nature, and some were part of the process.

For example, I found that because we were working with an offshore team there were occasional issues with using the English language. Sometimes we had to repeat things multiple times, sometimes we had to hold an entire meeting a second time to clarify issues that were at least partly the result of the inability to use common language. I "calculated" that these diversions cost us 3% to 5% of our work time. This is about 15-20 min a day for the whole team, and it adds up. Another time-sink was caused by a series of process related meetings, and these meetings provided close to zero value for us as a delivery team. In the first two months we attended the meetings twice a week for an hour and a half each. This alone was consuming 10% to 15% of the effective time for computer programming.

I knew that the situation would improve with time - people would start communicating more effectively, and we would be not attending the process related meetings for much longer. Everyone was allocated 100% to the project, but I still wanted to reflect that the real contribution was less. Thus, I came up with the concept of available thrust to signify that although the team may be assigned 100% of the time on the project, the thrust people put into developing working software can be handicapped by all sorts of reasons. (I couldn't use the term "utilization" as I dislike it when applied to people or teams. It implies viewing people as utilities, which I didn't appreciate. Additionally, it was not representing my perception correctly, since people are still utilized when they are in a meeting, only they are not always utilized very effectively. And I also felt that "available thrust" is a more positive term and contains within itself the notion of progressive forward movement, something I kind of wanted for the project's forecast.)

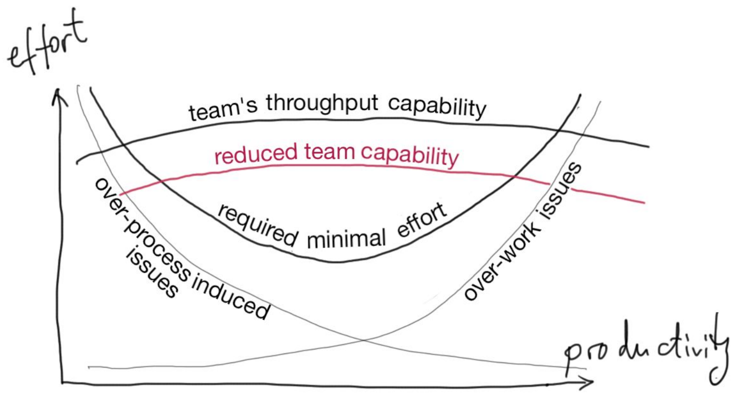


Fig. The actual team's throughput capability depends on multiple factors and this affects a number of project conditions, such as maximum sustainable productivity and team's ability to climb out of a setback. On the left the team is handicapped by bureaucracy, and on the right by being hasty.

I needed to expose this relationship because I wanted to claim a certain increase in the angle of the projection every time the team was able to remove any resistance factor. Thus, instead of just being able to say that performance will improve, I could speculate by how much it will improve. This allowed me to aim at the target with great precision. It also helped me have an adequate response when someone had overly optimistic expectations for the benefits from an expected improvement. Or when someone was too negative and was diverting time into unneeded solutioning of transitory problems. Often such arguments are fear driven and not based on real data. When meaningful data was available I was able to shift discussions towards constructive outcomes more often.

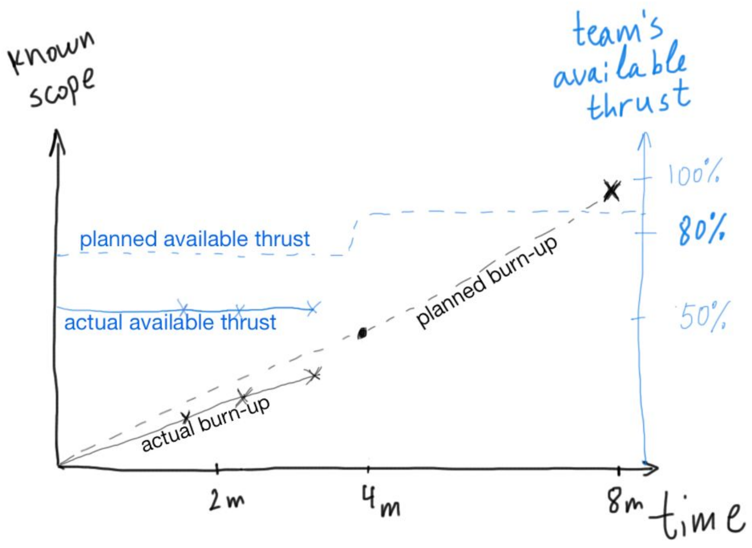


Fig. The plan here was to have 70% available thrust for the first half of the project and 85% for the remainder. The actual available thrust for the first three months is 50%. The project team can determine which course of action has a real likelihood of getting the project to a desired state - whether it is freeing capacity, reducing scope or adding new people.

I also wanted to reflect the fact that my team might not be at the planned staffing level. For example, if I had planned to start with 2 developers, and then to have 6 developers working by week 8 of the project, but there were only 4 developers by that time, then even if they were all fully dedicated to writing software for 80% of their time, this is still only 50% of the theoretical available thrust (4 x 80% / 6 = 53%). This is important because it is surprisingly easy for people to forget what the plan is in terms of staffing commitments, yet to retain unfaltering commitments to scope and deadlines. By plotting everything on the timeline, clarity, with all its benefits, is restored.

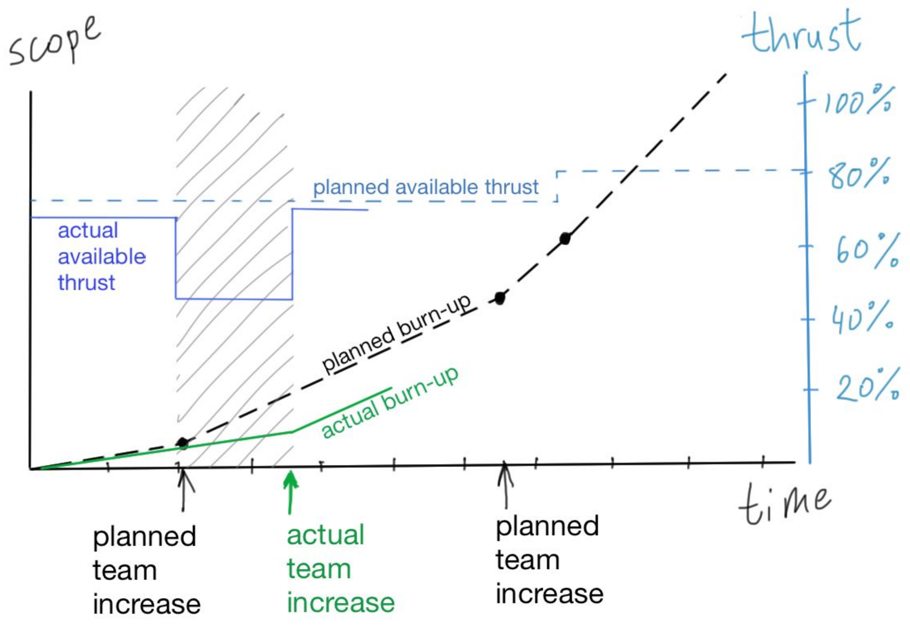


Fig. When we fail to increase the team as planned, the project starts underperforming, yet the current delivery team performs as expected. With available thrust depicted, people have a clear picture from which to produce a new plan of action.

With available thrust we can also account for more exotic factors affecting the project, like team dissatisfaction and project politics. If we have ample information to make an argument that team members are so dissatisfied that they actually need time during the day to cope with their frustrations - more frequent walks outside, more conversations, more arguments - we can assign a % of time for these additional activities and subtract from the available thrust. Similarly, we can assign cost to politics and process - if team members are being pulled into meetings only to serve as a backup for arguments, or if they are pulled into unnecessary process related meetings, a cost in terms of percentage available thrust can be assigned.

Context switching. We can "generate" available thrust, and up the angle of the projected line, by minimizing the amount of context switching. Context switching is caused by constant interruptions and fragmentation of the work in progress when working on too many things simultaneously, even if all of them contribute to the software solution.

By reducing context switching, and providing an environment where programmers can zone in on their work for longer periods, we can increase the available thrust compared to an environment where these factors are not in consideration. Measuring how much effective time we gain by minimizing context switching is not very difficult. The non-scientific way to do it is to ask a developer about how much time a day they feel is being lost from interruptions, or to add 5-15 minutes for each interruption and get an average number of interruptions.

Since context switching can easily contribute to 30-50% effective time loss (you read this correctly), it is imperative for a scrum master or project manager to familiarize themselves intimately with ways to minimize it:

- keeping an open and functioning communication network within the team (communication does not equal chatter)
- ensuring people are available for feedback when needed
- guarding the team from undesired external communications (when someone, a manager maybe, comes with a random and unrelated request)
- scheduling meetings according to how developers work[11]
- applying WIP limits and facilitating short cycle times
- having stories clearly specified[12]
- promoting direct interaction, communication and collaboration between all team members (improving sociometrics)[13]

These are all examples of things we can do to minimize context switching for the team. The time loss is not the only negative that comes with context switching. Sometimes brilliant ideas will disappear or never appear because the creative context was not preserved for long enough. These are losses that are difficult to measure, but not so difficult to feel.

Climbing faster. When we want to drive the angle up by adding more people to the project, we need to recognize that this is not a linear relation. Adding 30% more people does not necessarily translate in 30% gain in rate of climb.

---

[11] Maker's Schedule, Manager's Schedule - http://www.paulgraham.com/makersschedule.html

[12] Clearly specified requirement does not mean voluminous specifications. To make it more complicated, a clear specification does not guarantee shared knowledge and ease of communication. A great way to specify requirements is the subject of the book "Specification by Example"

[13] Having teams where people freely communicate with each other is crucial for many aspects of high performance. Where this has a positive effect on context switching is that a novice developer will not feel the need to interrupt a chain of people and ask for assistance in initiating a discussion with another project member.

People are the single most important factor on a project, and it can affect the project both ways - adding a person might have a positive effect on team's performance, and removing a person might also have a positive effect. We also need to account for warm up time when adding people. This typically shows as a short dip in the team's output followed by an increase in productivity (if the new people are contributing).

Something to keep in mind is Brook's law, which states:

"Adding people to a late software project makes it later"

We need to only add people before the project is late. By acting promptly on the available forecast indications we can add capable team members early enough to ensure net positive effect on the project. Of course it helps if scaling is planned and is not an afterthought.

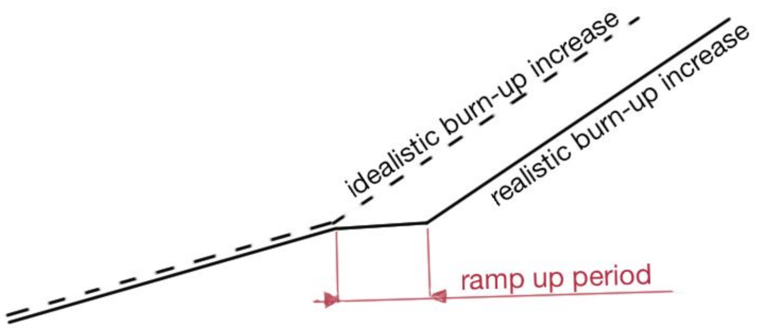


Fig. The ramp up period depends on the experience of the team members, the complexity of the domain, the relative enlargement of the team, and other factors. If there are two or three planned team increases throughout the project, the cumulative "delay" from ramp up activities can be substantial and should be accounted for in planning.

When applying project control by adding more people, we need to follow up with new data collection, measurements and plotting. If the expected performance improvement is not supported by the data we need to readjust the forecast and expectations.

Takeoff and Level Off

Take-off and **Level off.** Usually, work on a project starts slower and then accelerates a little before it stabilizes at a sustainable pace. If we simply draw a straight line from the zero point through a single data point, the performance line is more slant than needed. If we have more than one data point then the true project performance is captured better. For this reason, if we ever forecast on a single data point, which can be the case on a shorter project, we should adjust for this effect manually and slightly up the burn-up angle of the projection line.

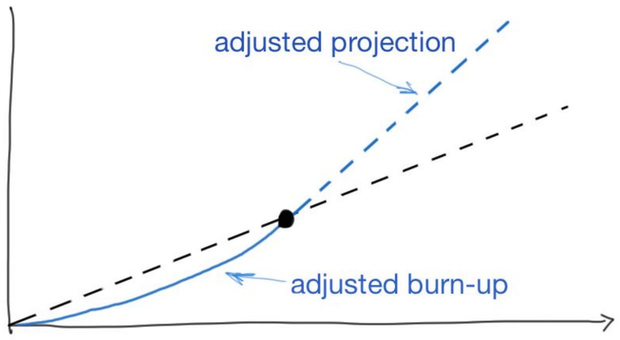


Fig. Adjusting for take-off is not only accounting for the ramp-up activities at the onset of the project, but is also properly reflecting the real team's productivity at the time of the single data reading.

Somewhat similar visually, but for a different reason, is the situation at the end of a project. Assuming a project ends with a deployment or a hand-off, then if we are aiming straight for the final date, we are effectively planning for a controlled crash on the last day of the project. Even if it is only for other people's comfort, we should consider allowing for a slow-down and level-off prior to the project's final day. The project performance line should gradually go horizontal an iteration or two before the end date so that people have time to work with the software in a stabilized state. This means that we need to account for the level-off duration as it requires a slightly steeper performance slope throughout the bulk of the project.

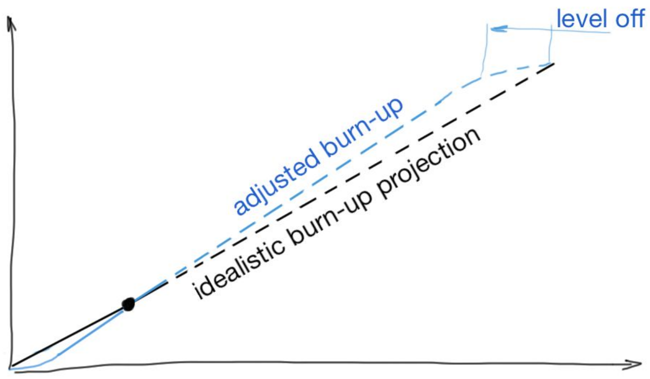


Fig. The required level-off at the end of the project effectively pushes the target scope completion sooner, thus steepening the necessary burn-up rate. This is an important consideration, especially when behind schedule and figuring out a burn-up rate that will put the project back on target.

The level-off adjustment might seem small but we can't underestimate the increase in project performance that is needed to support it. The adjustment only seems small on the piece of paper on which we are plotting the graph. In reality it requires thoughtful application of the team's energy resources and a sustained effort of shielding the team from detractors throughout the project.

...

Scope

Dealing with new scope and scope creep. When people are faced with an increase in scope, the tensions often run high and project teams get demoralized. Developers feel that business people are trying to load them with out of scope work without comprehending the project realities, and business people feel that developers are attempting to avoid work without comprehending the need for success.

This is one more area where charts and diagrams can simplify things significantly and bring everyone closer to agreement. For this reason alone, an intelligent forecast pays for itself multiple times along the life of a single project - removing tension and allowing people to focus on meaningful problems.

When new scope starts sneaking in, we need to:

- position the new target on the forecasting diagram
- track new scope and original scope separately
- demonstrate the situation
- outline the available options - aim at the new target; cease the addition of new scope; extend the project; defer other scope; or anything else that can be inferred from the forecast.

This clarity in choices dissipates tension in the team significantly. The situation is very different than adding new scope directly to a product backlog and mixing it with original scope, without a clear picture of the impact that this is having on target dates and expected features.

side note

Scope Destinations

When dealing with scope there are a few things we can do other than completing it:

- **Deferral** of scope is when we agree to postpone implementation until all other scope is complete, i.e. there is an agreement that deferred scope can be omitted from the project deliverables and still have a successful project.

- **Descoping** is when we remove scope from the project with the explicit understanding that we will no longer revisit it.

- **Reprioritization** is when we change the importance of scope with the intent to focus on the more important functionality earlier, but we still need the less important scope to be part of the project deliverables.

Additionally, if scope is organized by releases, it is useful to have fake releases labeled "*Deferred*" and "*Orphaned*". Deferred scope should be kept visible on the release level because it is still for consideration. Orphan scope is scope that has not been assigned to a release. But just because it has not been assigned to a release does not mean there is an explicit agreement to not complete it as part of the project. It is dangerous because sometimes it becomes "invisible" and then all of a sudden shows up in the most inconvenient moment.

Scope, and the approach that people take to change in scope, is another junction where multiple competing concepts confuse the situation and make it easier to get in conflict. These competing concepts are: product, project, and contract type.

Let's look at the common confusion between "product" and "project" first. The product oriented view is that we can simply prioritize work by business value, in an ever evolving backlog, and with this we guarantee that what is valued most gets completed first. However, for the people who look at the effort from the project oriented view, it is also important to know when and at what cost will the whole effort be complete. Both views are meaningful from the perspectives of the people who maintain them. It becomes important to have a tool that can bridge the gap and help people develop an understanding and empathy for the different view.

The other pair of concepts that are easy to confuse, and affect people's attitude to scope change, are "project" and "contract". There are two main contract types in software development - one is "Fixed Price" and the other is "Time & Material". Both contract types apply some structure to the relationship of the parties. A project on the other hand, by definition, is an enterprise with a specific goal. The goal is often defined in terms of specific functionality or capabilities. The issue here is that a Fixed Price contract puts a lot of stress on software developers, while Time & Material contract puts the stress on the client. Clients working on a project with Fixed Price contract can become insensitive to the pains of software and product developers, who are trying to stay within a set budget. And software developers working on a project with Times & Material contract might become insensitive to the client, who is trying to accomplish a defined set of functionality before their money runs out. It is important to keep in mind that the fact that a project has a "Time & Material" contract, does not invalidate the fact that the project also has a set goal (at least in the hearts and hopes of the people who initiated the project).

When we use a simple visual tool things become clearer, with less possibility for misunderstanding. Finding satisfactory solutions becomes a controllable effort, and people are free to make clear choices from mutually complementing options. In this sense the value of a forecast goes beyond its potential accuracy. The forecast becomes a tool for communication between people who struggle finding a common language, thus saving time, projects, and relationships. This is the true power of a good forecast.

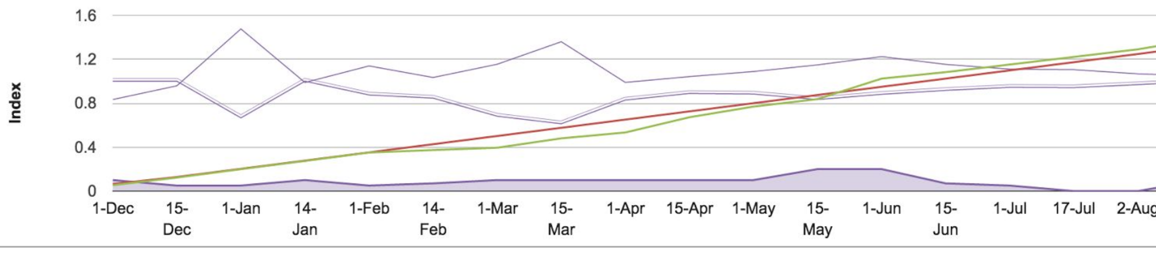
There is nothing bad with scope creep. As it happens, people discover new things they had not thought about at the start of a project or things they simply forgot to mention. This newly discovered scope often deserves to be included in the project's definition of success. As long as people are aware that it is "new", and they are still willing to keep it in the action plan, then the extra scope cannot be good or bad. It becomes simply an additional factor to consider. What is not good, is when people pretend that adding new work is as simple as adding new specifications to a backlog, and has no implication on the rest of the project parameters like cost, duration, complexity, work environment and more.

To facilitate ...

Chapter 6

In this chapter we will look at a tool providing additional insight into the health and progress of a project. We will also look at a few situations that can be expected on a typical project.

Performance Index



The scope forecast that we explored in previous chapters enables us to apply measured project control towards desired functionality and calendar targets. Scope and schedule are important project dimensions, but to be confident and resolute in the decision making process we need a clear view of the energy and financial state of the project. This will enable us to not only steer the controls to where we want, but also to have an informed expectation for the longer term capability of the project.

The indication of available thrust already provides deep insight into the project's condition. However, available thrust is more about the team's output than about the project's. The team might be hitting all the planned functionality targets, but at a higher cost and thus the scope forecast alone is not sufficient. We can plot the planned[14] and actual dollar expenses, along with a few index values that provide

[14] For projects with a fixed scope and fixed budget contract, the planned cost is clear. For projects with time and material contracts there is no agreed upon cost, but that does not prevent us from planning or establishing a desired cost, or at least indicating the projected cost.

information for the financial health of the project. A set of simple indications which improve awareness of the project’s financial and energy state are:

- money burn - planned cost and actual cost
- value index - scope (“value”) we have delivered to date compared to planned
- relative cost index - money we have spent per unit of scope compared to planned
- borrowing index - the overtime we have borrowed from employees in order to be where we are

...

Chapter 7

=========== Chapter 7 is work in progress =============

The Big Kahuna

Adriana put her bag on the chair and her coffee cup on the desk. She brushed the snow off her coat and took it off so she can soak the warmth of the office air. Then she typed a quick messenger note to Josh, one of the Scrum Masters with whom she wanted to talk about an upcoming project, "Josh, can you please meet me for a quick talk in my office. We might have a new project coming."

Josh wrapped up with what he was doing and headed to Adriana's office. "Hi, Adriana, how are you!"

"Hi Josh! A little cold actually. It's freezing outside." she searched for something in her bag and when she found it she continued "The company name is DT International, and they want to build a mobile version of their daytime trading system."

"They need an estimate in order to sign a contract. We need to work out something if we want to stay in the game. They are talking to at least one other software company."

"Ok. Do we know if they are looking for a 'Fixed budget' contract, or if they are open to 'Time & Material'? And do you know if there is a set calendar deadline?"

"Yes, they need it ready before next November, as there is an audit in January and they need to be in compliance with new regulations. Big kahunas like these are usually not open to Time & Material. We need to give them a fixed number and it will be difficult to get them to pay more, we need to get it right from the first time."

"Hmm. Ok." said Joshed with just a little concern. "But they know that we work iteratively and we need them involved and making decisions throughout the project, right?... Do we know who are the lead developers who can work on this project? I assume we have some documentation too.”

“Yes, I just emailed you the high level specifications. What they need from us is to design and develop the mobile interface, and all the functionality is described in the specification. They will provide APIs to their backend systems. And yes, we made it clear that we work iteratively and that the product will evolve with active collaboration from both ends.”

“And who are the developers?”, asked Josh

“Oh, yes. I’ve already talked with Jackie. She might be already working on an estimate, and you should go talk to her. I expect both of you to come up with the estimate and a plan.” Adriana was looking at her phone and scrolling through the appointments in her calendar. “How does it sound? Do you think you have what you need for now?”

“For now, yes. How soon do we need to have an estimate?”

“We have 3 days to get back to them with an estimate. We will have some more time to plan and adjust after that but probably not more than a week or two. ”

“Ok. I’ll talk with you later today or tomorrow.” Josh was already building hypothetical project plans in his mind and was considering the risks. Developers loved the agile methodology. And fixed scope projects were hard to implement in a truly agile fashion. He had to secure the environment and the working relationships that developers got to anticipate and in fact to rely on in order to deliver high quality products.

Estimating

“Hi Jackie! I’m happy we will be working together again. Did you start looking at it already?”

“Hi Josh! Me too. I’m glad we’ll get to be on the same team again.” Jackie was flipping between a couple of files on her screen and it was visible she already initiated the first stab at comprehending the problem.

“How does it look?” asked Josh.

“I haven’t looked at all of it yet. It looks big, but nothing scary so far.” said Jackie.

“Do you know who else will be working with us?” Josh knew that Jackie is one of the more experienced developers and was wondering whether the things that didn’t look scary to her would scare some of the other people who would eventually join the team.

“I’m thinking to create a WSB and estimate the hours of work.” continued Jackie. “It worked well for the last project.”

“This one is much bigger though. Do you think we can break it down to hours? Seems like a lot of work. And is there enough detail in the spec?”

“I know, right! It’s stressing me out already, but we need to provide an accurate estimate. If we don’t break it down to the smallest possible detail we can’t estimate accurately enough.”

Josh was looking at the screen and seemed unconvinced. As if he wanted to say “Let’s try something else.”

“Do you have something else in mind?” asked Jackie. “I mean, what else can we do other than breaking it down to a fine WSB and then summing up all the hours?”

“What do you think about estimating it in ranges?” said Josh. “It will be easier, less stressful, and it represents the reality of what we know at the moment better than an hour estimate. What do you think?”

“I like the less stressful part, but I’m not sure that this is going to give us an accurate estimate.”

“How many pages is this high level specification document?” asked Josh.

“It’s about 45-50 pages, if we discount for some of the charts. Why?”

“And if we start in a two weeks we will have a little over 11 months, right?” said Josh without pausing his thread of thought.

“Well yes, but we can’t really count the time around Christmas for much. I’d say 11 months sharp. And this is if we can get all the setup quick enough.” Jackie was already feeling slightly pressured by the discussion of the limited time available for the job.

“That’s true. So let’s plan as if we only had 10 months.” said Josh and continued calculating something in his mind. “If we split each page in about 3-4 sections that feel like independent chunks of related functionality, we’ll end up with 130-180 chunks, right?” Josh wanted to finish the explanation of his approach. “We can then assign a t-shirt size to each chunk and total up all the ranges. Over 100 pieces has to be enough to give us pretty good confidence. Statistically.”

“But this will be a very wide range. I’m not sure that’s what Adriana needs.” said Jackie.

“We’ll narrow it down later. But for now it is what we can produce with the information we have.” Josh seemed confident. “And I don’t want to put us in a corner by providing a single number. It is a collaborative exercise. Let’s see if everyone involved feels comfortable with a broader range first. And instead of hours let’s use days or even weeks - 72 hours is difficult to comprehend as a measure of effort.”

Jackie thought for a few seconds and said “Okay. I’m in! What do we do next?” Jackie seemed eager to see the result of this approach as it was already feeling somehow natural.

"Well let's look at a few of these chunks that we have and see how big they seem to you." said Josh. Even though she agreed to the experiment Jackie was still doubtful.

"But please think of the effort as if someone else will be coding it." said Josh. "I know you are one of the quicker programmers so when estimating, please imagine it's someone a little less skilful, but still competent enough."

"Cool. I'm starting to get interested in this method of estimating. Almost sounds like fun." she smiled. "And actually, we code in pairs, so it might be better to estimate it with that in mind. It's close to lunch. Do you want to grab something?" And the two of them took their jackets and went out of the office.

After lunch they got in a nice conference room which they decided will be their office for the remainder of the day.

"Yes, you are right. It will be much better to estimate having in mind that it's a programming pair doing the work." said Josh. He was happy that Jackie was such an experienced developer and was making the plan more realistic. "And let's assume we also have one tester working with the programming pair. So with all these assumptions, how are these first few chunks of work looking?"

Jackie was reading through the description of an editable personal information form that had to be initialized by an external service. A few more details were spelled out around the usage of the form and the information on it. "Hm, this looks like at least a few days of work. With all the mobile platforms we need to test and support it might be more than a week actually." Jackie was thinking back to similar solutions she had programmed in the past and she knew there might be tricky places when dealing with personal information. She haven't worked on a banking application before so this made her willing to go with a safer estimate rather than an overly aggressive one.

"So would you say about 2 weeks is adequate for this?" asked Josh. "I also expect that some of it will need to change when the users provide feedback. Not sure yet how that process will be handled but you rely on their feedback during Sprint Reviews, right?"

"That's right. If we don't have that process working well I will not be a happy programmer on this project. A broken feedback process adds a lot of uncertainty. Honestly, I'm not even sure if I want to be on this project if we are only going to simulate being Agile." Jackie was obviously passionate about the way the programmers approached work and she didn't want the subject to be just an afterthought.

The next few chunks of work looked about the same. One looked a little more than 2 weeks, one looked about 3 weeks, and there were a few more that seemed a week and a half, but both Jackie and Josh thought it's fair to count them as 2 week effort.

"Ok. Let's take the 2 week period as a mean and build a few ranges around it. I think this will speed up the estimation for the remainder of the chunks as otherwise we are looking at too much detail." said Josh.

"Yeah, with this speed I was afraid we have to stay here through the night." laughed Jackie. "And you know how much we programmers dislike overtime." She said it with a smile, but Josh was paying attention. He didn't like overtime either. An occasional instance when everyone is so deeply engaged and in rhythm that they forget to stop working at the end of the day was ok and even could be a fun team building experience. But having to consistently work overtime just to sustain the project was one of the things no one in the company liked. Most people had diverse experiences with other companies and other software development methodologies, and overtime was often part of the culture. Readiness to work overtime was even regarded as a virtue in many of these places and people who wanted to live their private lives were sometimes considered slackers, or poor team players.

An almost empty spreadsheet file was taking up the whole screen. "Okay. Let's do this!" said Josh.

"You can see the ranges. They are overlapping a little, but just place each chunk where you feel it belongs most. I'll be recording the results here on the right side."

Tidbits (Chapter 8)

A soup of related concepts, heuristics and practical details is mixed up in this chapter with the intent to provide good starting points for anyone who applies intelligent forecasting in practice.

Software Development Laws

Brook's Law. Brook's Law talks about the effect of adding "manpower" to a software project:

> *"Adding manpower to a late software project makes it later." (Brooks 1975)*

Sometimes generalised as:

> *"Adding people to software development slows it down"*

It is important to note here that the generalized form is just that, a generalization. It is true, but only for a set period that follows immediately after adding people to the project. Once the new team members have had time to ramp up and start contributing, the net throughput of development might increase (depending on many other factors).

The original version of this principle is important for project control because it tells us that people should be added to a project only in anticipatory mode, i.e. planned. If we are reactively adding people because we have realized that the project is late, then it is too late.

As we said earlier in the book, forecasting can help with this situation since it puts us exactly in the anticipatory mode needed for sensible project control. We can decide whether more people are needed in advance of "being late".

Gall’s Law. Gall's Law is about growing software systems. It explains how complex but working systems become in existence:

> *"A complex system that works is invariably found to have evolved from a simple system that worked. A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over with a working simple system."* (Gall 1986)

This principle can be applied to software teams as well and has important implications on scaling teams or working on projects with large teams (10 small scrum teams working on the same project comprise a very large project team).

> *"A complex team that works is invariably found to have evolved from a simple team that worked. A complex team designed from scratch never works and cannot be patched up to make it work. You have to start over with a working simple team"*

This law basically says that we need to grow teams, and not scale them or “architect” them. This is important, since growing a team does not have to start with adding more people. We should first exhaust the practical opportunities for growth and improvement within the small and simple team, including teaching them new skills. One of the benefits of this is that when we have a team of high performing individuals, adding new individuals causes much less disturbance. The new people tend to accelerate quicker because most of their interactions are providing support and building their confidence. This feeds into an organically sustainable growing mechanism - as long as we don’t exceed the intake capacity of the highly proficient team.

Many frameworks for applying structure at scale attempt to slice and dice a large project team into small groups and hope to achieve the efficiency of the small group, but simply applied at scale. My experience with this approach is not positive. Large teams that are broken in multiple small teams are still large teams. They need to be grown. Once we start with a normal sized team and grow it beyond it’s natural capacity, we can add more people and then split it in two. Then keep growing each of the teams gradually until we are able to split in two again. Of course, we need to account for a many great number of other factors if we are to build a high performing team. But starting with a large team, or many small teams, is a sure way to fail.

[a diagram for growing vs. scaling up]

Starting a project with a large team puts the project in a state of being late as soon as it has started (because the team can not be patched to function). Projects like this operate under Brook's law at all times and we are stuck in a death spiral where the more people we add the less chance we have for making it out unscathed.

Parkinson's Law and **Hofstadter's Law**

"Work expands so as to fill the time available for its completion" (Parkinson's Law)

"It always takes longer than you expect, even when you take into account Hofstadter's Law." (Hofstadter 1980)

These two principles are true when we look at a project as a single monstrous task and we grand-total the small tasks into a single large effort. With the method of intelligent forecasting, we in fact treat each chunk of work as a small project with its own probability of being on time, early, or late. We then utilize the Central Limit Theorem to work on all the probabilities together, accounting for the effect of each one, while not focusing on any individually. We are also tracking the project performance with additional project indexes, which provide a fine grained survey and prevent the work from simply expanding and compressing within the full project envelope.

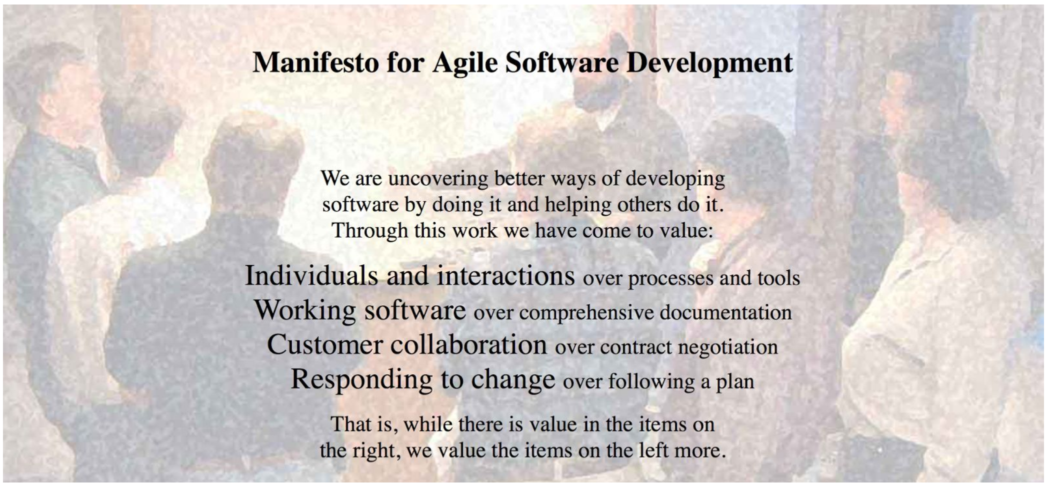
Conway's Law

"Organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations" (Conway 1968)

Conway’s Law can be applied to multiple aspects of software teams and the systems they build. In the context of forecasting, it reinforces Gall’s Law because if we want to start with a small and simple system, we also need a small team working on it. Otherwise, if we start with a complex team, even if the system starts small, it will inherit the complexities of the large team through an overly complex design and architecture. A complex team will not be able to produce a simple solution. This will immediately affect the team’s ability to work in a sustainable way and the result is often legacy systems even before they have been deployed to production.

Manifesto for Agile Software Development

The Manifesto for Agile Software Development is a document that gives valuable perspectives on a few core dimensions. I'll share my thoughts here, and stay close to what is relevant to forecasting projects. The manifesto is a document that is worth reflecting on, and people working with scrum or other frameworks related to agile development should take time to discuss what applies to their projects and where they want to seek improvements.



Manifesto for Agile Software Development

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more.

Fig. The Manifesto as printed on http://agilemanifesto.org

The original manifesto text is below, in slightly <u>larger font and is underlined</u>, and my thoughts are inline, and in slightly *smaller and slanted type*.

<u>Manifesto for Agile Software Development</u>

<u>We are uncovering better ways of developing software</u> *(A "hint" that this is primarily about software development and not about project delivery. Better software development enables better project delivery but does not replace it or guarantee it.)* <u>by doing it and helping others do it.</u>
<u>Through this work we have come to value:</u>

Individuals *(clients, managers, developers and scrum masters are individuals. We need to treat them as such and value their problems and issues)* and interactions over processes and tools *(agile development, scrum, XP, #noEstimates, and estimates, all fall in this group. We should utilize the tools that facilitate interactions with individuals.)*
Working software over comprehensive documentation
Customer collaboration *(working together implies taking care of each other's needs. It cannot be one sided where developers only care for better software and business people only care for business)* over contract negotiation *(positioning yourself well for a possible negotiation, and enabling other people to reason productively during negotiation, is good. Contract negotiation is different than negotiation.)*
Responding to change *(knowing what the change means, in the context of the project, is important for an adequate response. Baselining, tracking, forecasting and planning are not useless activities as they put change in context.)* over following a plan *(trying to follow a static plan at any cost is the issue. Having a plan that you intend to follow, but you are willing to revisit and adjust, is not a bad thing.)*

You can read more and reflect here:

http://agilemanifesto.org
http://agilemanifesto.org/principles.html

Common Gaps, Environment Needs and Tiny Details

Project set up. A common gap on many software development projects is to have the parameters of the project negotiated, and to only then assemble a team with the mandate to complete the project as defined. This creates issues on multiple levels.

If the delivery team and the PM (or the sole SM) are not involved in the project setup then they have missed out on the opportunity to affect and manage one of the most important stages of the project - its initiation. Once the project is set up, the chances for success are irreversibly affected. Having participation from as many of the team members as possible is an important step towards minimizing risk.

Many projects are defined within a Statement Of Work (SOW) that never gets presented to the delivery team. The team is simply handed a set of requirements and a final date. This is not the best way to approach a team if we want them to get to a high performing state. People can sense that they are not being treated equally, and predictably retract their full and committed participation in the endeavour. After all, if people are to develop a sense of ownership, then they need to be treated as proper owners of the problem. Also, important consideration in the form of assumptions and dependencies might get completely overlooked by the people who "crafted" the SOW, thus exposing the project to additional risk.

One of the first steps towards a sensible working environment on any software development project is to have the full delivery team familiarized and ideally involved with the creation of the SOW. This lays the groundwork for a participative team and improves the risk profile of a project significantly. The delivery team should define the assumptions which must hold true throughout the execution of the project, as well as they need to identify any major dependencies and risks that need to be worked out in order to have a realistic chance for success.

Project start. When the delivery team jumps into implementation work right away, there is a huge risk introduced to the project immediately. The likelihood for a successful project diminishes greatly when the delivery team is not allowed the opportunity to familiarize themselves with the business problems, main users of the proposed system and their primary pain points, the typical journeys these users take through the current processes. Sometimes in an effort to save money, a select group of individuals will go over some of these topics, and will produce written documents, only to then have the documents stashed somewhere out of reach of the delivery teams. People can go through year-long projects and then accidentally stumble on useful descriptions of personas, success criteria, and intricate flow charts with useful insight.

This is not what we want to save money on. Shared knowledge and understanding is crucial for optimal flow and a solid solution. A preferable approach is to take a few full days and have the delivery team and clients go over the pertinent items together. There are various tools for facilitating this process and it is best if completed by a professional facilitator. An investment like this will pay off on any project longer than 2-3 months.

Project Manager vs. Scrum Master. The job title, of course, does not matter. However, the attitude does, and the roles are different. Many Project Managers (PMs) approach project management remotely and do not immerse themselves fully in neither the details of the implementation work, nor in the relationships within the team. A Scrum Master (SM) on the other hand has to be intimately involved in everything that goes into the team, and also be willing to serve the people on the team and provide assistance, guidance or anything else that is needed to keep the team delivering great work. On the other hand, many SMs would refuse to expand sizable amount of energy on long term activities like project planning. An SM is mainly concerned with the work of the team within the scope of an iteration or two.

My approach, and recommendation is to always care about the team first. Not the forecast or the project plan. This gives a more useful perspective for the whole forecasting exercise as it becomes primarily a tool for helping the team and the project move forward. When the PM aspect takes precedence over the SM aspect then the person responsible for managing the project might get confused that the forecast or project plan actually drives what is happening. This is usually contra-productive and backfires, as people and reality refuse to conform to the imposed plan. So for me it has always worked best to have the forecast and the plan be driven by what is the real capability of the situation, but fully realizing the impact that the forecast and plan have on the situation and various possibilities.

SCARF

"SCARF: A Brain-Based Model for Collaborating with and Influencing Others" This model is based on neuroscience and provides a useful framework for any type of social interactions, including negotiations. SCARF is an acronym for "Status, Certainty, Autonomy, Relatedness, Fairness" and the model is based around people's reward and threat responses on these five aspects of an interaction.

Status - relative importance to others (in the context of the interaction)
Certainty - ability to predict the future
Autonomy - the sense of having control over events
Relatedness - the perception of safety with others
Fairness - the perception of fair exchanges between people

Here is how an intelligent and reliable forecast facilitates the practical application of SCARF within the boundaries of the project:

Status. Within the context of a project, the status of a client is that of someone who has asked for help in accomplishing something that is of value to them. And our status is that of someone who has been entrusted with seeing their project to a successful end. When we promote the use and application of precise forecasting, we establish ourselves on the controls of the project and we are allowing the client to take their natural status of someone who is being provided a service. This normalizes the relationship, triggers the client's reward response, and allows for the right type of interactions to occur throughout the duration of the project. It is their project, but they have already entrusted us with it, and by not shying away from the responsibility to control the project, we cocreate the correct social structure.

Certainty. By providing a clear outlook and prognosis, based on provable tracking data, we are delivering the ultimate in clarity and certainty to the client, and to the team. When we confidently indicate the likelihood of not being able to achieve a goal well in advance of the expected time, we provide the client with the opportunity to work out solutions while there is still time. And when we confidently state that there is a high likelihood of achieving a goal, we allow the client to divert their precious energy to solving other important problems. In both cases, we are triggering the reward response in the client - either by allowing them ample space for working out the project problems, or by allowing them the comfort of actually knowing that things are on track. (A very different sensation than the one provided by claimed progress, but combined with a continual inability to provide an outlook for the final result.)

Autonomy. By providing clear and viable options for steering the project, we provide the client with the autonomy of choosing which option suits their situation the best. The client can steer the project where they want, yet still within the safety of what we have determined is doable. This way we maintain the "status" aspect of the relationship and we are still responsible for the ultimate effect that the application of a control has on the project, but we allow the client to choose which control to apply.

Relatedness. By being transparent in presenting the factors affecting the project's progress, and by openly providing multiple options for dealing with difficult situations, we show the client that we are working for them and that we are one team. With a willingness to let go of negotiations that have become inadequate for one reason or another, and by allowing clients to freely re-negotiate scope while maintaining a clear picture of the involved costs, we become immediately relatable. Simultaneously we continually maintain the client's perception of our positive ability to carry out our part of the relationship. This shows that not only do we care, but we actually can, and will, do whatever it takes to get the job done (because we are basing everything on the provable capability of the situation).

Fairness. By laying down all the options, and by demonstrating regard for every percentage of throughput that we can apply towards the project's success, we create an environment where every action is treated fairly within the envelope of the project. Everything has a clear cost and a clear benefit. Choices have clear expected outcomes, and there are no secrets or intentional gotchas. By creating a fair playing field we allow the client to have a reward response in this aspect of the negotiation as well.

"Negotiation" - The dictionary definition of the word is "*discussion aimed at reaching an agreement*". However, I enjoy thinking about this word with its usage in the expression "negotiating turns". This way the whole project and all the people on it become one unit with an immediate objective of handling an obstacle. Negotiations on a software project take on a completely new meaning when I think of the word this way.

Invert, always Invert

Carl Gustav Jacob Jacobi (1804 – 1851) was a German mathematician. One of his maxims was: '*Invert, always invert*'. He believed that the solution of many complex problems can be facilitated by expressing the inverse form of the problem.

By studying the inverse of the complex problem we are presented with interesting options, and a valid solution often emerges. Here is how this principle is exercised in a few of the ideas in this book:

- To gain the ability to control the project with precision we agreed to rely on things over which we don't have control and can not evaluate with precision. By inverting the focus from precision to imprecision, we allow for valuable statistical properties to emerge in support of our objective.

- In order to be able to change the parameters of the project we commit to a specific outcome, seemingly putting ourselves in a situation that does not permit change of parameters. By inverting the typical order of events, and committing before we know the exact details, we allow details to emerge once we are in motion and we are not simply working out a predefined solution, but we are actively designing the problem itself. In true collaboration with clients we examine what success means and we adjust the details of its very definition in a way that supports the project's practical ability within a defined envelope.

- By inverting our focus from personal dynamics and high performing teamwork, and instead invest energy into developing a tool which is sourced from a place that seems diametrically opposed to the human aspect of the project, namely statistics and scheduling, we allow space for the human relationships to develop around the critical supports of trust and safety (which are facilitated by the tool). This then allows high performance team dynamics to emerge and carry out the project.

Appendix

Cheatsheet

[a step by step poster/infographic style of cheatsheet]

Variations of the Intelligent Forecasting Method

Using a story map directly. If the forecast needed by business people is in broad strokes, this approach can work well. It is a good alternative to zero forecasting when the relationship with the client is strong and the team is performing well. In essence it is very similar to what is described in this book.

This approach also provides no high-fidelity guidance for applying proper project control, it does not indicate trouble soon enough, and it does not (directly) facilitate differentiating “added scope” from “original scope”.

All stories as 1-point stories. This approach is worth considering if we already have many stories that are similarly sized. Getting the whole backlog into similarly sized small stories can be difficult and expensive, though.

Something else to be aware of with this approach is that some stories are still going to be riskier than others. If we do the right thing, and take on the risky and difficult stories first, then we might have a slower start. Plotted on 1-dot-per-story data this will indicate we are tracking too slow for the project. We then need to convince people not to worry, since we expect to be going quicker through the 1-dot stories planned for later.

In practice, this method is better suited for iteration level planning and maintaining consistent delivery pace across iterations. It can save time in place of the planning poker game and it also nudges the team towards smaller stories, which can improve the flow significantly.

Spreadsheets and Sample Charts

Index

References

https://leanpub.com/xanpan2/read
https://www.amazon.ca/Xanpan-Centric-Agile-software-development-ebook/dp/B012ZSY9HO/ref=sr_1_17?s=books&ie=UTF8&qid=1481824924&sr=1-17&keywords=team+software

The Case For Project Management

https://www.leadingagile.com/2011/10/the-case-for-project-management/?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+LeadingAgile+%28LeadingAgile%29

http://www.jamesshore.com/In-the-News/Estimates-or-No-Estimates.html

http://www.allaboutagile.com (this seems to be coupled with www.leadingagile.com and also very good)

https://en.wikipedia.org/wiki/Sample_size_determination

https://en.wikipedia.org/wiki/Central_limit_theorem

http://www.agilenutshell.com/cone_of_uncertainty https://www.gilb.com/blog/when-you-can-measure-what-you-are-speaking-about-and-express-it-in-numbers-you-know-something-about-it

http://connection.mit.edu/wp-content/uploads/sites/29/2014/12/hbr-new-science-teams-2012.pdf

https://en.wikipedia.org/wiki/Angle_of_climb

https://en.wikipedia.org/wiki/The_Mythical_Man-Month

http://ithinktowin.blogspot.ca/2009/05/change-management-subtle-difference.html

https://en.wikipedia.org/wiki/Scientific_method

Software Quality Management / Systems Thinking

Specification by Example

https://en.wikipedia.org/wiki/Conceptual_model

https://viaconflict.wordpress.com/2015/04/30/using-the-scarf-model-to-navigate-the-psychological-landmines-of-negotiations/