A Comparison of Effort Estimation Techniques on Software Projects

A Comparison of Effort Estimation Techniques on Software Projects:

How Long Will Your Project Take?

By

Karl Cox

Cambridge Scholars Publishing

A Comparison of Effort Estimation Techniques on Software Projects:
How Long Will Your Project Take?

By Karl Cox

This book first published 2023

Cambridge Scholars Publishing

Lady Stephenson Library, Newcastle upon Tyne, NE6 2PA, UK

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

ISBN (10): 1-5275-0206-6
ISBN (13): 978-1-5275-0206-2

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The first time I heard about effort estimation for software projects was in 1997. I was studying for a master's degree in software engineering and was told by a very experienced project manager who had had a full career of managing large scale projects, that the best way to get 'close' to the reality of the final project effort was to:

1. Estimate the size of the project – typically with function points
2. Double the answer
3. Add 25%

Then you would be as close as you'd ever get. In other words, for big projects estimation techniques were of little value because the large projects only got larger, longer and much more expensive.

This did not fill me with confidence in what appeared to be such an important part of a project: how "big" is the product so that we can work out how long it is going to take? If the techniques to estimate project size, duration and cost are so inaccurate[1], why is there so much emphasis on estimation in the first place? I got curious and decided to read up on the subject.

Perhaps the founding father of software effort estimation is Professor Barry Boehm. He wrote the great tome on the topic called *Software Engineering Economics*, first published in 1981. Barry created the Constructive Cost Modelling estimation approach—COCOMO—that changed how industry looked at effort and cost estimation. The principles Barry defined for approaching effort estimation have been used pretty much everywhere in software engineering.

[1] One of my colleagues tells me when he was a software developer whatever the estimate calculated, he would multiply it by pi (3.14) in the hope of getting closer to the inevitably much larger actual.

I liked Barry's book and COCOMO, but I found it really hard to get my head round those cost drivers. Nonetheless, I got the concept of why estimation matters. I also read a brilliant book on software measurement by Professor Martin Shepperd[2], who at the time, was head of computer science research at Bournemouth University where I was studying. It's by far the best book I have read on the topic of software measurement and made a lot of sense to me. I soon found myself a member of Martin's empirical software engineering research group working on my PhD.

When I'd finished my PhD and got itchy feet, I ended up in Australia at the University of New South Wales working under Professor Ross Jeffery, also renowned for software measurement research. I'd jumped from one empirical software engineering research group into another. But hard numbers, formulae, equations, and piles of data points were not really my cup of tea—I got it but not the enthusiasm. I was more interested in the requirements end of the software lifecycle, which by its nature is much more qualitative than quantitative. I then worked in an empirical research group at an R&D company called NICTA whose strategy was to boost the Australian IT sector through the creation of useful software and systems applications. I found project managers in Australian government agencies still concerned with estimation of deadlines and costs, often because their projects were overrunning in both aspects. Why? Their projects were too large.

Interestingly, a lot of practical application has been achieved with estimation techniques because project managers do want solutions to this perennial problem. All the approaches discussed in this book are being used or have been used in industry. We will get on to the success or otherwise of estimation approaches a little later as we discuss each one. Further, the more I engage in teaching project management, I cannot avoid looking into estimation because it is so important to successful projects. Having worked on and managed real software development projects, I have found that any software development is ultimately dependent upon getting enough work done to satisfy the customer to the quality and scope they expect, to the agreed budget and more on schedule than less.

The beginning of each project is shrouded in mysteries: what is the deadline and is it realistic, what are the key requirements and are these volatile, how experienced is the project team and is it necessary to hire in

[2] Martin Shepperd (1995), *Foundations of Software Measurement*, Prentice Hall, ISBN: 0-13-336199-3.

contractors, what work is outsourced, what business processes are changed by this system and how, and is there enough budget to deliver a working application with the needed functionality at the right level of quality by the agreed deadline?

If you are a student reading this book, to answer the question in the book co-title: how long will your project take? It will take until the submission deadline. So that's it, then. End of book, thank you for reading… But is this everything you need to know as a computing student on this topic? Not really. I would like to run though several topics on estimation and examine some of the approaches to estimation that are past or at least ought to be by now, such as traditional function point analysis, to current ones such as calculating work-in-progress limits across a Kanban board, to understanding the current "financial" state of your project through earned value analysis.

Other aspects of project management are not addressed. You won't find huge discussion on different project management lifecycle methods—except where needed. I compared traditional development against agile in a book on managing your individual computing project[3] so I won't address this here. You won't find Gantt charts or any detail on Kanban boards. I wrote elsewhere about how computing students can apply three basic project management tools in their assignments in combination with business and requirements analysis tools.[4] The book you are reading now is only about estimation.

Why write a book on software project effort estimation for computing students?

Why do you need to consider estimation when you're doing a student project or assignment? Surely the submission deadline will tell you when you have to finish? Yes, that is one thing you don't need to estimate. But if you're working on a complicated group assignment, you will have to think about when tasks need to get done by so that everything can be completed on time. You'll also need to consider what the most efficient approach to

[3] Karl Cox (2017), *Managing your individual computing project – An agile approach for students and supervisors 2nd ed*, Create Space Publishing, ISBN: 978-1542778114.

[4] Karl Cox (2022), *Business Analysis, Requirements, and Project Management: A Guide for Computing Students*, Auerbach Publications – CRC Press, ISBN: 978-1032109756

the project to take will be. You'll need to consider how much time you need to allocate to certain tasks because this will tell you how much more time you can give to other tasks, such as writing up your project documentation report.

Please don't misunderstand me, I am not proposing you apply super-complex and highly advanced algorithmic approaches to estimation because, quite frankly, they are hopelessly inaccurate. The more complex the approach to estimating with the more parameters in play and variables to consider, the more likely your estimate will be miles out. Also, the more time you spend estimating, the more likely your estimates are going to be more wrong. You're trying to measure something yet to exist—just which part of this non-existent product are you measuring? This brings to mind a graph known as the "cone of uncertainty".

In figure 1-1 you can see this "cone" describing phases of a development project on the x-axis, and schedule over- and underestimates on the y-axis. The two curves on the project are furthest away from the x-axis at the beginning of the project. It is only when the project is complete that the two curves merge on the x-axis. The y-axis describes how far over or below the actual duration of the project you will be in your estimates. At the beginning of the project, you could be as much as two thirds overestimating the project (1.6x) or two thirds underestimating it (0.6x). In other words, it is highly unlikely you will have a within ballpark estimate (say within 20 per cent) until you have completed the specification.

I personally think even then that within 20 per cent will not be obtainable until you are much further into the design. It is only at the point where you have finished the project and handed over the software that you can be sure your estimate of project schedule is accurate. There is no point attempting to estimate the effort needed to complete a project at the very beginning of the project. Nowhere near enough is known about it at that point. You need to wait until further in.

Although the idea of the cone was written about over 40 years ago, not much has really changed in the world of estimating in terms of getting the schedule estimated perfectly from the start. If you're lucky you get a "ball-park figure" to begin with and hope this is close enough. As you progress through a project, that estimate should become more accurate as you learn more about the product and your project's parameters (environment and team). You will realise you are either over-estimating or under-estimating as per the curves on the graph in the figure.

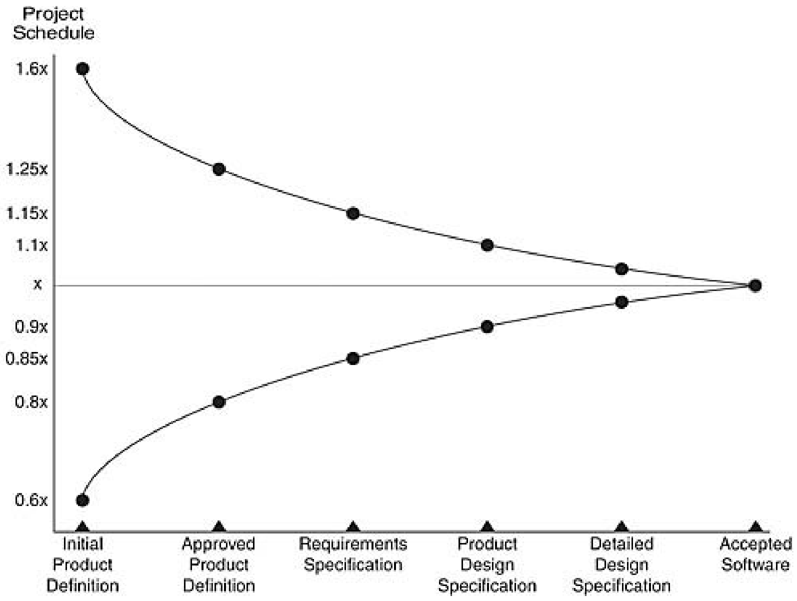

Figure 1-1. Cone of uncertainty. This was originally described by Barry Boehm as the "funnel curve" in his 1981 classic, *Software Engineering Economics*.

You might ask if all you are likely to get is an experience similar to that depicted in figure 1-1, then why should you bother estimating at all? It's true that your project—if you are a student—will have a hard deadline so maybe you don't need to bother? You might ask yourself if you are going to end up working in the industry and would that not mean working on projects? Those projects will have deadlines. The project manager on those projects—which could be you!—will have to estimate how the work needs to progress in order to meet the deadline. The deadline is often decided prior to the project kicking off. This often means that not a great deal is really known about client needs so the deadline is little more than a speculative guess, based upon some memory of projects being like this before. On any project, how do you know how long to take on writing a design document before you need to get to code? How long on average should you spend on programming on a project? According to Boehm, it is around 13 per cent! I've seen estimates of around 10-15 per cent from other sources. It's hard to believe, isn't it? Is your goal to deliver a software product where a minor task—in terms of effort—is the programming of the product? No, it does not make a lot of sense to me either. Certainly, my own experiences indicate this is a big underestimate. What is the rest of the project effort spent on? Well, a lot is put on getting the product's specification spot on. The idea of getting the whole blueprint correct before you build the product is very much a "traditional" view. In agile development, the blueprint should only come into play upon the particular user/technical story or requirement being programmed. Either way, analysis work is done to get a better understanding of the use of the function(s) in the context of those using it. Design work can take a lot of time, ensuring the database structure is done correctly and getting the business logic between the front end and back end correct. UML design has been around for 25 years and more, and it is pretty stable as the way to design your system on its inside. There's a lot of interface design work to do also and user experience design is really important to get spot on. Post programming, there is testing, too. We will exclude maintenance from our estimates because maintenance could go on for years, subject to contractual agreement between supplier and client. Looking back over the long list of things that may need to be done on the project, perhaps it is no wonder that programming is such a small chunk of the workload.

Productivity

A team's productivity rating is key to better development. Productivity means how efficiently you make or programme the product calculated as a daily, monthly or whole project rate. You may even base productivity on an individual developer rate. In other words, productivity means how fast can you get the work done? All the effort estimation in the world means very little without factoring in productivity. As an example, you might have to programme 20 screens for an ecommerce retail shopping web application. Historically on average, your project team can develop 0.75 screens per day. "Develop" means as a vertical slice: understand the requirements, do the design needed (screen, code, database, privacy, security, networks), programme the designs into a working screen and test the requirements against it. So, if there are twenty screens and a productivity rating of 0.75,

Project duration = number of screens / productivity rate
Duration = 20 / 0.75 = 26.67, which we will round up to 27 days.

Our main concern is to get the most efficient and effective working environment for your team. Being too productive—meaning here coding too quickly—can lead to a loss of interest in good quality assurance. When this happens, you have a big problem because the software product will have a lot of bugs and errors. Requirements will be misinterpreted. Even if those misinterpreted functions work on the programmed application, the product will fail to be useful to the customer and end users because it does not do what the client wants it to do. Or the product will not work properly. High productivity may mean reduced client satisfaction because of a higher bug rate. Is this really being productive? What if your productivity was two screens per day? In our little example,

Duration = 20 / 2 = 10 days.

Great! You built the application in less than half the time of the first estimate! But, as a consequence, what if you released a third more bugs? What is the cost of fixing those bugs? This is both a question of extra scheduled effort to bug fix and—perhaps more importantly—one of loss of reputation. Or confirmation of a bad reputation; customers know you work fast but they know you make more mistakes. This ultimately costs them more than if you took a more measured approach.

The opposite end of the spectrum is one that imposes too much quality control. This carries the risk wherein the delivery date will be pushed back unnecessarily, resulting in increased customer costs and extra costs for the development team. For the team, the extra costs may be offset costs or opportunity costs. Offset costs may be those of the allocated budget for the next project being dipped into so as to complete the current project. You are offsetting the next project against this current one, or at least prioritising it by your actions. Opportunity costs incur when unnecessary time is wasted which could have been put to much better use.[5]

[5] An example of opportunity cost: I knew a PhD student who had just completed her thesis and needed to get it printed. She was concerned that the price of printing locally was too high. So, she spent an entire week going from printer shop to printer shop across Sydney comparing prices. She triumphantly announced a week later that she had found a slightly cheaper printer resulting in saving $100 overall on the print. Fine, you might say, well done to her. Yet it was a case of penny-wise, pound-foolish. She spent around $50 travelling from printers to printers on bus and train tickets, and a similar amount on food, and she wasted an entire week doing so. The opportunity to have done something far more productive during that week was the cost of getting her

In the situation of too much quality assurance, there may grow a perfectionist's culture where the developers won't push for customer acceptance sign off because the developers always see ways to continually improve the product. This means a never-ending cycle of enhancement, testing, refusal to release the software, another round of enhancements needed—in the eyes of the developers only—and so on. The customer is now frustrated. If the development team asked the client to approve the product for release, the team might push back against the client's wishes of getting the product into proper usage right now. Ultimately, the team grow to believe the product is theirs because they created it.[6]

Failure to release adequate software can arise from a culture of fear. The development team never really believe their product is good enough. Without endless rounds of testing, revisions and refactoring, this fear will override any rational consideration or deliberation on project progress and the needs of the customer. The developers are worried the product will be rejected by the customer, so they never finish tinkering with it.

High productivity is important but there can be over-emphasis on it. I think you are better off being efficient and if this is close to "high productivity" then this is good enough. If your efficient best is still regarded as slow, then this slow productivity rate can be used as a weapon to undermine the confidence of the development team. If your team's productivity is noticeably slower

---

thesis printed at her local printers for $100 more. Of course, she didn't look at it the way I did.

[6] This might strike you as ridiculous, but I witnessed this for real on a project I worked on in Sydney. A government agency project had overrun by five years and was over budget by $60 million—it was originally contracted to be a one-year, three-million-dollar project. I was hired late into the project to look at opportunities to speed up end delivery but found a very big mess. I talked to a lot of staff on the project and found two people sitting forlorn in a corner of the office floor this large project was occupying. They had more or less sneered at me every time I walked past them so I decided to ask what their problem was. These two people turned out to be the customer representatives—and they had been banned from talking about their project with the development team and the project's management when they refused to pay $1 million for a function not specified or agreed to and entirely unwanted! How do you ban a customer from discussing their project?! How daft is that? The developers had taken over the project to the point where the customer was shut out. This happens surprisingly often especially when large systems integration (SI) companies get involved because the SI tries to take over the business of the project and not just build the product. My recommendation to speed up the Sydney project was to shut it down since nothing had been delivered and the requirements had grown from an initial 3,000 to 12,000. It was never going to succeed.

than other teams in your organisation, then management may want to find out why. The added pressure does not help. If your team is superfast, this may well be offset by the higher number of bugs you typically release. Being a tall poppy can quickly lead you to feeling like a sore thumb. Conversely, if your productivity is lower than other teams, why is this?

Perhaps your team is very inexperienced compared to others. You have a team composed of a high number of recent graduates, for instance. Naturally, because of lesser experience, your productivity rate is lower. But this should change over the course of a year once your team gain that experience. So, let's assume your productivity now is 0.5 screens per day.

Duration is 20 / 0.5 = 40 days.

This seems very slow compared to the superfast ten days duration or even the 27 days. Given you are inexperienced, does it mean there is also a risk of more bugs than average being released? This is possible. But it is hoped that the extra time taken ensures a better-quality product.

A simple way to address slow productivity is to include a good mix of experience in the project team. Staff who have 5- to 10-years' experience mixed with recent graduate intake can balance the productivity issues that might occur. Inexperienced staff will learn from the experienced and ultimately, their own productivity rates will increase.

You may find productivity is particularly slow on one project even with experienced developers. This could be because of a new client; staff are still feeling their way in how they work best with the client. Perhaps the client's business domain is slightly different to your previous experiences, and you need to learn their business on top of building their system for them? This may mean you need to develop business concept models and/or business process models to understand their business better before the normal development work can begin in earnest. This added work—vital in building a product that really does meet the needs of the client—is only done at the start with a new client. Since further projects are mostly enhancements to that original product or product line, extending and/or re-analysing the existing business concept models and business process models[7] takes much less time than starting from scratch.

[7] For an introduction to these business models, see: Karl Cox (2022), *Business Analysis, Requirements, and Project Management: A Guide for Computing Students*, Auerbach Publications – CRC Press, ISBN: 978-1032109756.

Your project environment may have changed and this can reduce productivity. Simple things like relocation of your office to a different floor or building can impact productivity hugely, even if the physical relocation happened before the project commenced. Impacts can be having to walk further to use the facilities, to being unable to warm up your lunch in the new building, to feeling too cold or too hot all the time because the ambient temperatures are not set to optimal yet. You're either shivering or snoozing! Perhaps you're using a new compiler or new application development suite meaning you need to get used to the new tools. This slows down productivity.

You may have a new project manager who you are not yet used to working with. Perhaps this new manager is more rigorous in collecting project data, meaning you have to produce higher quality and more accurate progress reports, taking time out from development work. Your business may have shifted direction slightly meaning the products you provide for them and/or external clients are different enough for your project to be slowed down whilst you get to grips with a potentially new product line.

Tactics for improving productivity range from going on a team building weekend to forcing your team to work longer hours, even the weekends! Neither are particularly brilliant, so I recommend you apply the following to your team to improve productivity.

Acceptance (which does not mean agreement). Accept that other people produce different quality work to you and do it in a different way. Provided the quality is good enough for your client and meets the standard set by the team, then this is fine. You do not have to agree with how work is done or that you could have done it better, but you should accept it. What you can't change, accept it! Life will become much easier for you if you do this. If you complain long enough about a colleague's work, you will end up having to do that work. Will you do a better job?

Communication. Ensure your team has an agreed way to keep in touch. You may find you become distributed if you have to work at home. How do you keep in touch? Having a communications plan helps provided it is followed and it is meaningful. As you should be in the same office, communication is easier but having regular, short meetings helps such as daily stand ups.

Understanding. Please don't misunderstand me! It isn't easy for me to look at someone doing something really badly—from my point of view—and not to jump in! But when I realise we are different, all of us, then I allow for some understanding of a different person's perspective on work. We get along better that way and our team is more successful.

Encouragement. Be positive! Not the fake happy-clappy "everything is super" approach, with whooping, cheering and back slapping. But you can say: "Well done, that work is spot on." Or "Good job." The impact on your colleagues is enormous when we all encourage each other sensibly. Keep the back slapping to the pub after work.

Unity. Work together as one unit. In Japan, IT teams tend to work as slowly as their slowest person so as not to isolate them or make them feel useless. This is Kanban work-in-progress limits in action as you will see later in the book. New graduates are treated like this to encourage them to work with the team. In unity is strength! I have seen this work well when working with companies in Japan. Soon the graduate speeds up and all is well.

Forgiveness. When something goes wrong, forgive the person who got it wrong and if it was you who made the mistake, forgive yourself first. Then think of how to learn from this mistake and to ensure it doesn't re-occur. If it does recur, then forgive the person/yourself and get on with learning from the mistake again. This is how we improve and become better at our work and as people. The worst thing you can do is point fingers and blame people, gossiping in the background. This generates a poisonous energy that affects everyone involved. It is very hard to eradicate once it is there. Treat others as you would have them treat you – the Golden Rule!

Sometimes staff are told to work weekends in order to get a product out the door because it is already past its deadline or very soon will be. This book is primarily targeted at students so this may not concern you because your deadlines are fixed and immoveable. When and where you work to meet the deadlines is entirely down to you. Even group work assessments are mostly done outside of class time. In an office environment, however, it isn't so easy to sacrifice your weekend for your employer. You may well have a family at home who have plans for the weekend that include you turning up. Employers expect you to manage your family life in your own way. If work life has to impinge on family life, employers tend not to really care too much about the impact. It's just business, they may say. Thankfully, there are more enlightened employers out there who do realise that family time and time away from the office is essential to employees. The more worn out you are from working overtime, the less productive you will be during normal office hours. One of the precepts of Agile is to give staff their weekend lives and to limit overtime to one day per sprint per developer. Some companies understand that many of their staff are young and have no ties, so they do allow these staff to work extra hours. But they need to be careful not to allow it too much. Burnout has been a very big problem in this industry and it still is. When you burnout, you are useless to your employer, you resent being at work and pretty soon quit, moving on to another job in the hope your new environment will refresh your energies once again. Companies with a high staff turnover rate get noticed. No one wants to work there because no one survives more than six months!

Consequently, clients notice that the quality of their products begins to fall as the development company staff begin to lose enthusiasm. This can lead to contractual problems where projects slip behind schedule as the productivity-quality relationship stumbles. The ultimate weapon a client has is to move their business elsewhere.

If you notice that your productivity rating is fluctuating wildly in the same team from project to project, then something in the nature of your projects needs investigating. Perhaps a cause is that for different clients, you need to use different development environments. Perhaps even different programming languages. You might now be a C# house but historically you were Java, and some of your older clients still have Java applications that need maintenance. Your project team is now very experienced with C# but you only have one very experienced Java programmer on your team (who is also a C# developer). This means when you need to work on a project for a Java client, only one team member can get the bulk of that work done. The productivity of other projects is reduced as a consequence. The client has to rely on the experienced Java programmer to get the project finished and there is almost no one else who can assist in the code work because they don't understand Java's syntax well enough. Perhaps this client can be convinced to convert to a C# application rather than Java? Perhaps the client is entirely unaware their product is a Java product anyway? It would be worth discussing this with the client. Could your team deliver a fully working C# version of the Java product over time? Could your client switch immediately to the C# version that's offered to your other clients? Is your client willing to transition its data and business functions just to support a C# environment rather than Java?

Perhaps each project you take on is a bespoke development. This means each project is unique and significantly different to any other project. Subject to how complex or different the project, this can result is highly fluctuating productivity rates. Taking an average rating doesn't really help in this case. You might instead be wiser streamlining your business. For example,

- We only develop C# applications.
- We are only a Windows OS product developer.
- We only build web applications.
- We only build B2B products[8].
- We only work in the retail banking sector.

In all honesty, the above is what companies already do anyway. That lone Java developer may wish to move elsewhere or may in fact also be a very competent C# developer. You would be wise to hold on to their experience and business knowledge, finding something the developer would like to do.

For students working on a group assignment, pushing for higher productivity can lead to frustration. It can lead to one person not trusting their team, even from day 1, because of a prior negative experience, and taking on the bulk of the work him or herself. This can lead to all kinds of problems. The rest of the team may build a resentment against the one person, may refuse to cooperate, may break away from that person and set up a second group which includes everyone minus that one person. None of which will be communicated to the tutor (I have experienced this many times!). Sometimes one or two students don't do any work and it is hard to accept or forgive this. But there may be genuine reasons why a group member isn't getting in touch or attending meetings. Rather than throw that person out of the group, it would be wiser to talk with staff to look at how that person can re-engage with the group, if possible, and sooner rather than later.

To manage groupwork productivity in a university setting, look at the consistency of the work distributed among the team. Who is doing the most work? Who appears to be doing nothing or very little? When a task is delegated, ensure someone works as a "buddy" or assistant on that specific item of work. This means if the work is beyond the capability of the first-

[8] B2B = business-to-business such as supplying control software for washing machine manufacturers; B2C = business-to-consumer, such as making a retail web application for the public to purchase our stock of clothing.

choice developer, the buddy can help out before the schedule slips. How much effort will each item of work take? Is coding a database more effort than designing a highly usable responsive screen? Is writing a requirements document less effort than programming a security control monitoring access rights to a dataset? Is managing a project considered productive? It is really important to realise that a genuine contribution is sometimes viewed as apparently insignificant work compared to other work. Programmers may feel they are doing the bulk of the work and those doing the analysis, documentation and management are only bit-part players. The fact is that the team succeeds or fails as a team, not because of one person. A successful team is where everyone contributes something of value. Writing a requirements document is a valuable input into the project. Conducting testing is a valuable contribution. Writing a report for the project—if required—is very valuable. Managing the project's progress is a vital activity for a successful outcome. Your success is judged at the group level not the individual.

A note on quality in relation to productivity

Software quality, ultimately, is the be-all end-all. In other words, if you have no- or low-quality software, your product isn't going to work. If it isn't going to work, it isn't going to be used. You can, therefore, be the fastest producer of software in the world, but if the quality of your product sucks, then you won't be in business for very long. No one will use your product because it is rubbish. When you go shopping online at say Amazon.your_nation, you can find a range of differently priced products. Some are expensive, you think, so you look for a better deal, so called cheap and cheerful. Read some of the reviews of these lesser-priced products and you'll be thinking "cheap and tearful" instead! This is because the overriding problem the cheaper products have is their lack of quality. They are too flimsy, or they arrive damaged, or are the wrong size and colour, have missing parts or the wrong parts, or are not compatible with your system as claimed in their advertisement. Or they break within minutes, days or weeks of usage despite claims to their robustness. The reviews make it clear: Don't waste your money! So, you don't and you spend more as a consequence, or you do and take a risk. It's the same with software quality. If it is poor you soon find you are looking for an alternative solution but by then it is too late, you are contractually in a wrangle with the development team, and you can't find a review on Amazon before your software is completed to warn you it will suck.

Poor quality can be defined in terms of bad interface design (the screen layout is counter-intuitive and you have to re-click buttons or links several times to get anything to happen), continual system error messages pop up (because the system files are not as compatible with your operating system as expected and that much needed driver file cannot be found), unexpected outputs from inputs (e.g. you work in a customer accounts department, input a customer invoice number that retrieves the customer's credit card details but not their order) or the function you needed isn't implemented as you expected (meaning your procedures have to change unnecessarily to meet the functioning of the software) and so on. Your productivity can be really high but if you deliver a low level of quality, then there's a problem. Conversely, slower productivity is no guarantee of high—or even good—quality because speed does not have a relationship inversely proportional to quality. However, a slower productivity rate can imply better quality because the assumption that more time on testing and even on understanding customer need has been taken. These are assumptions and may not come to fruition unless the manager in charge of the department imposes greater controls on the software development lifecycle such that quality assurance procedures and practices are put in place and adhered to, and that they undergo regular review.

Quality metrics include things like bug counts and size of bugs. There is mean time to failure—what is the average lifespan of the product before it fails to be useable? Other quality metrics range from performance to security to interoperability. There are complexity metrics that help in quality determination. Software quality is a major area for measurement but software quality metrics as a topic is beyond the scope of this book. I recommend you read Martin Shepperd's excellent chapter on software quality in his book *Foundations of Software Measurement* for starters. This is all I have to say about quality here. I will refer to it as we progress through the book but we are more focused on effort and productivity as you will see.

Person-day effort

One last thing to mention that I think it has been addressed in a roundabout way already. Duration of a project is the same as elapsed time. In other words, the duration of a project is the end-to-end number of days it takes to do something. My project starts March 10th and ends April 30th. How many days is that? There are 31 days in March. Subtract 10 from 31 and you have 21 days remaining. Add 30 days to that and you have an elapsed time of 51 days. I am not considering things like weekends and Easter holidays here. The raw end-to-end total called elapsed time is 51 days.

Effort is different in that it takes into consideration the number of people who do the work as well as the estimated duration for each task. For instance, if I am the only person doing my 51-day project, the effort is 51 person days. If there are two of us working on the project for 51 days, the effort is 102 person days even though the elapsed time remains 51 days. If the two of us can do the work in 40 days, then the elapsed time for the project is 40 days but the person-day effort is 80.

When you plan out a project, you have to take the number of staff into consideration. Primarily, you need to pay them, so you have to estimate the effort required in person-days as much as the elapsed time for the project. This book explains established techniques for calculating effort to build a product. When these estimates are made, the project manager will then take into consideration the person-day effort. Productivity is another way of explaining person-day effort. We can put two people on the task of implementing a major function that we estimate will take one person 10 days. Can we assume two people could do the work in 5 days? Is the task something that is easily divided into two pieces that both developers can work on full time and then join their efforts together at the end? We need to know what the work is before we can consider the impact on effort and duration. If it doesn't make sense to split the task then should we leave it as it is with one person working on it or add the other person and hope that together they can finish the task in less time than estimated?

The consequences of this consideration moving forwards are that the estimation techniques must take your team size into consideration. You should also be aware that not all the team will be deployed all the time. Analysts will start and at some point, before the end of the project, they will have completed their work. Designers will start after the analysts and programmers after the designers. Testers will work on the project only after some code has been developed. So, you cannot take it for granted your team will be working 100 per cent of the time on 100 per cent of the project.

What I would like you to take away after reading this book is that it is a good idea to estimate the effort needed for a project. Although you are going to be hampered by inexperience if you are a student reading this, when you go work in industry after graduating or during your placement year, you will realise that estimation is something managers will do for good reason.

As we progress through the book, you will see that some approaches to estimation take team size into explicit consideration whereas others do not except in terms of productivity as explained above. Many of the "successful" effort estimation approaches since then owe their origins to function points, which we look at in detail later, and COCOMO.

COCOMO

Barry Boehm's COCOMO and its successor, COCOMO II, are not really used any more as far as I am aware. But I give you an idea of COCOMO here because it did change the way industry worked, how projects were managed and how estimation was done, and as such is worthy of mention. COCOMO (COnstructive COst MOdel) applied to three classes of software projects:

- Organic projects – "small-sized" teams with "good" experience working with "less than rigid" requirements.
- Semi-detached projects – "medium-sized" teams with mixed experience working with a mix of rigid and less than rigid requirements.
- Embedded projects - developed within a set of "tight" constraints. It is also combination of organic and semi-detached projects (hardware, software, operational, ...).

The basic COCOMO equations take the form:

- Effort Applied (E) = $a_b(KLOC)b_b$ [person-months]
- Development Time (D) = c_b(Effort Applied)d_b [months]
- People required (P) = Effort Applied / Development Time [count]

where KLOC is the estimated number of delivered lines of code (expressed in thousands) for a project. The coefficients a_b, b_b, c_b and d_b are given in table 1-1.

Software project type	a_b	b_b	c_b	d_b
Organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Table 1-1. COCOMO coefficients

A coefficient is a constant factor or multiplier used to expressed greater or lesser uncertainty or complexity. An organic project—something we are used to doing—would have a coefficient of 2.4 (a_b) multiplied against estimated thousands of lines of code, multiplied by the b_b coefficient 1.05. This should result in a person-months effort. Development time is effort applied multiplied by the coefficients of c_b and d_b, 2.5 and 3.8 respectively. Team size is then calculated by dividing the effort by development time. Estimating using lines of code may be viewed as problematic because we would need to have done quite a bit of programming already before we can estimate lines of code with any accuracy. By which time, the project may be nearer the finish line than the start! There are also a significant number of complexity and environmental factors I am ignoring such as team experience and others. I will look at these factors more in chapters 2 and 3 especially in relation to function point analysis and use case points.

Was COCOMO any good? If it was reliable, I suspect it would still be in use. But this does not appear to be the case. However, its very existence changed how projects were run and that means it had a significant impact. Without COCOMO I am not sure other software sizing approaches would have emerged in quite the same way they did.

What this book covers

We will explore a number of more popular estimation techniques, at least two of which may appear a little too old to concern ourselves with. But we must look at them because they had a big impact on industrial practice and are still in use around the world today in places, so we do need to address them here.

Chapter 2 takes us back in time to explore function points. My goal is not to regale you with too much history but to get as quickly as possible to the calculations, so you know how to do them. I could have included COCOMO as a chapter but opted for function points (FP) because it is a technique still deployed on government projects though the versions used vary. This book explores function points Mark II designed for business systems that emerged out of Albrecht's original function point analysis (FPA). The main reason to examine Mark II FPA is because it was the FP approach more ideal to be taught on business computing and software engineering degrees that I have taught on. It also appears somewhat simpler than the full IBM-IFPUG Function Points approach for students to understand.

Chapter 3 looks at use case points. Use cases are a very popular way to diagram and document use of a software system from the user perspective and are core to UML, the Unified Modelling Language, the standard in designing object-oriented systems. Use case points was created when it became clear that sizing a system described in use cases was actually much more difficult than imagined and follows a similar structure to function point analysis.

Chapter 4 goes agile as we explore the size estimation technique of story points. You will have noted the repetition of "points" in function points, use case points and story points. There are similarities in the approaches, especially between function and use case. Story points only borrow the name. Nonetheless, they are very popular in estimating the effort or size of user stories, the popular agile way to document requirements. Coupled with story points, we will describe a technique commonly used to work out the points, "planning poker", or as it was originally known, Delphi estimation.

Chapter 5 moves away from the "points" approaches and describes earned value analysis (EVA), a financial indication of a project's progress both in terms of schedule and budget. EVA is a popular approach to pinpointing current progress (how is our project going and is it going to continue like this?) and is widely used across a diverse range of industries. Earned value is not a size estimation technique and is deployed differently to the other estimation approaches described.

Chapter 6 examines the estimation techniques deployed in Kanban agile projects. We explore work-in-progress rates that can be viewed as a measure of productivity, as well as techniques to estimate project duration and team size. Kanban is becoming more popular and has been advanced by such tools as the Trello[9] board. As such, we cannot ignore this rising star of agile development.

[9] Trello is a trademarked product name for a Kanban-style board software application owned by Atlassian. Try it out as it is fantastic and free: www.trello.com

Chapter 7 concludes the book and draws analogies between the approaches presented. We shall also look at where estimation has headed with a discussion of COSMIC function points analysis and “simple function points”, IFPUG’s standard body approach to estimating function points for agile projects.

The Appendix provides answers and where needed some brief discussion and explanation to the exercises set for you to do in chapters 2 through 6.

Table 1-2 shows two columns: “During-Actual” and “During-End”. These represent the two major points at which you can apply an estimation technique. During-Actual means at a point in the project we can examine exactly where we are compared to where we should be according to the plan and/or budget. During-End means we can calculate our end date at a point during the project (other than at the very start). But shouldn’t we be applying our estimation at the start of the project? Wouldn’t that help in determining budget, team and project duration? Shouldn’t we add to the table a column “Start-End”, meaning we could apply an estimation technique, such as story points, at the very start of the project to work out the end date? In fact, we cannot apply any sensible estimation approach right at the very start of a project unless we are taking a blind guess. The reason being that in order to have an estimate worth taking the time to consider, we need to know quite a bit about our project and have an especially good idea of the proposed product requirements. It is from these requirements (user stories, use cases, functional requirements, inputs-outputs) that we can begin to apply our estimates. The lag time from the start of the project to having a specification or a well-populated backlog of user stories is significant. It can take from weeks to months, respective to the size of the project. This time has to be accounted for in the schedule and in the budget, and deliverables need to be approved. Effectively, this is the analysis phase of a project—not to mention the business analysis that needs to be done prior to systems analysis. We can really only have a good grasp of the complexities of a project once we have gone through at least the majority of this early lifecycle phase. It is only at this point that we can apply our estimation approach because we need data to work with if our result is going to be in any way meaningful. Our size estimation is only for the code work to be done. That’s interesting if we are to believe that only 13 per cent of project effort is in the code work, as Boehm had calculated. Or if we are generous, 15 per cent. It appears we are putting huge effort into calculating around only 15 per cent of a project’s work! Is it worth all the fuss then or can we conclude that software development on projects is actually much more than 15 per cent? What happens in the remaining 85 per cent? Who calculates that and with what? In fact, it is the design and testing of the software, as well as the code, that we are estimating, which is much more than the programming work.

Chapter	Estimation Type / Technique / Representation	During-Actual	During-End
2	Function Points Mark II	☒	☑
3	Use Case Points	☒	☑
4	Story Points (including Delphi or "Planning Poker")	☒	☑
5	Earned Value Analysis	☑	☑
6	Kanban (WIP, duration, team size)	☒	☑

Table 1-2. Estimation approaches covered in this book and when they are best deployed

You will note in table 1-2 that all the estimation techniques can be applied to the During-End estimate. Only Earned Value Analysis helps represent where you currently are during a project. This is a core purpose of earned value. Work-in-progress—part of Kanban—is used to depict the amount of work you should be doing at any one point in time which is not necessarily the same thing as comparing where you should be against where you are. You could count items in the Done column of a Kanban board against those remaining to give an idea of percentage progress and hence estimate how much time is remaining.

Also note that Delphi is subsumed within Story Points because Delphi is a process employed to size stories and of itself a tool used in helping estimate size. Delphi estimation (now typically referred to as "planning poker" in the agile community) is a core practice that can be used across all approaches to estimate the relative size of a feature, function, requirement, user story or use case.

There's a lot to talk about with each approach and I will do that in each chapter on the specific approach. I would like to note, though, that we may also need to consider—in relation to table 1-2—whether we have the appropriate team size to address the schedule. Schedule estimates will change throughout a project and it's important to recognise when a significant change in team size may be required to complete closer to the deadline than the current trajectory might indicate. The different approaches to estimation—addressing size and schedule—can be used to *guesstimate* the necessary team size now required to complete on time or nearer on time. Notwithstanding Brooks' Law[10], the only approach tabled that explicitly looks at team size is part of the Kanban estimation suite. Brooks' Law states—from Fred Brooks' bitter experience!—that adding staff to a project that is already late will only make it later. This is borne out by his experience where many projects overrunning having had staff added late to help speed up the delivery only delayed the projects even further. Staff who join a project, no matter how experienced, need time to get up to speed. Who is going to help these new team members do just that? Those currently engaged flat out on the project is who. If these current staff have to pause working to help their new colleagues get up to speed (understanding the project context; the project environment—who is doing what, where the documentation is at; figuring out the requirements and so on), then inevitably the work rate will slow down on a project already running behind schedule. Hence, Brooks' experience that a project running late will only run later if staff are added late.

Limitations of this book

The explanations and examples provided in this book are limited in complexity because of the experience of the audience the book is targeted at. There is no doubt a lot more to say about each technique as I represent it. For further details, I recommend readers check the references provided throughout the book. I have also limited the techniques we look at because I think some are no longer in use and other newer ones may not be having any significant impact yet. However, I will briefly discuss two of the newer approaches in chapter 7 that are current incarnations of function points analysis.

[10] Fred Brooks (1995), *The Mythical Man Month*—Anniversary Edition, Addison-Wesley, ISBN: 978-0201835953