Fingers in the Air: a Gentle Introduction to Software Estimation

Giovanni Asproni, gasproni@asprotunity.com
Asprotunity Limited, www.asprotunity.com

Introduction

Estimation is a fundamental activity of every project. A project has always some goals, and the purpose of estimation is to give a reasonably accurate idea of what is necessary in terms of time, costs, technology, and other resources, to achieve them. Since resources are always limited, an accurate estimate is an invaluable tool for deciding if the goals are realistic, as well as for creating and updating the project plans.

All parties involved in a project need to know something about estimation:

- Customers need a way to check that what they are told is correct, and also if the project is worth their investment
- Managers need to manage the project and also provide estimates to their managers and to the customers in order to set their expectations properly, and enable them to take informed decisions
- Developers need to provide estimates to their managers for the tasks they have to perform, so that managers can produce or update the project plans

Unfortunately, the software development community has a very poor record in estimating anything—in fact is very common for projects to run over-time and over-budget, and deliver poor quality products with fewer features than originally planned.

Part of the problem is that software is quite difficult to estimate. In fact, huge differences in individual productivity, the fact that creative processes are difficult to plan, the fact that software is intangible, and the fact that during the life of the project anything can change - e.g., scope, budget, deadlines, and requirements - make software estimation a challenging task.

However, in my experience, the main cause of poor estimates is that the various stakeholders are often unaware of what estimates are for, and what they should look like. This lack of knowledge means that they have no way to judge if the project goals and the associated expectations are realistic. The result can be either overestimation which causes the allocation of an excess of resources to the project, or, more often, gross underestimation which often leads to "death march" projects [You], which are characterized by "project parameters" that exceed the norm by at least 50%, e.g.:

- The schedule has been compressed to less than half the amount estimated by a rational process
- The staff is less than half it should be in a project of the same size and scope
- The budget and resources are less than half they should be
- The features, functionality and other requirements are twice what they would be under normal circumstances

The rest of the article is an introduction to the software estimation process aimed at project managers, developers and customers who want to get a better understanding of the basics this subject, and avoid to make their projects a death march one.

Some definitions: estimates, targets, and commitments

A typical conversation between a project manager and a programmer talking about estimates often goes along these lines

- PM: “Can you give me an estimate of the time necessary to develop feature xyz?”
- Programmer: “One month”
- PM: “That's far too long, we've got only one week!”
- Programmer: “I need at least three”
- PM: “I can give you two at most”
- Programmer: “Deal!”

The conversation above is an example of the “guess the number I'm thinking of” game [You]: the programmer, at the end, comes up with an “estimate” that matches what is acceptable for the manager, but, since it is *his* estimate, the manager will hold him accountable for that. Something is obviously wrong, but what is it exactly?

In order to answer this question, let me introduce three definitions - estimate, target, and commitment.

An **estimate -** as defined in “The New Oxford Dictionary of English” - is “an approximate calculation or judgement of the value, number, quantity, or extent of something”.

In general we make estimates because we cannot directly measure the value of the quantity because [Stu]:

1. The object is inaccessible (e.g., continuous operational use)
2. The object does not exist yet (e.g., a new software product)
3. Measurement would be too expensive or dangerous

A very important thing to notice is that the definition above implies that an estimate is an objective measure. If I asked you to estimate my height, you would probably have a look at me and come up with a number irrespective of the fact that I wish to be taller than what I actually am - if I told you that your estimate had to be at least 2 metres, you'd probably laugh at me (I'm a bit more that 1.7 metres tall), yet, this is exactly what the manager in the conversation above was doing.

Two examples of estimates from a real project setting could be:

- Implementing the search functionality will require between two and four days
- The development costs will be between forty and sixty million dollars

A **target** is a statement of a desirable business objective. Some examples of targets are:

- Release version 1.0 by Christmas
- The system must support at least 400 concurrent users
- The cost must not exceed three million pounds

A **commitment** is a promise to deliver specified functionality at a certain level of quality by a certain date.

Some examples of commitments are:

- The search functionality will be available in the next release of the product
- The response time will improve by 30% by the end of the next iteration

Estimates, targets and commitments are independent from each other, but targets and commitments should be based on sound estimates, or, as written in [McC]:

"The primary purpose of software estimation is not to predict a project's outcome; it is to determine whether a project's targets are realistic enough to allow the project to be controlled to meet them".

In other terms, the purpose of estimation is to make proper project management and planning possible, and allow the project stakeholders to make commitments based on realistic targets.

Back to the conversation at the beginning of this section. What the project manager was *really* asking the programmer was to make a commitment, not to provide an estimate. Even worse, the commitment was not based on a sound estimate, but only on an unstated target that the manager had in mind. However, it should be clear now that **estimates are not commitments, and they are not negotiable**.

Why does this kind of conversation happen then? In many cases there is a simple answer: more often than not, people are well meaning, but are (especially managers) under heavy pressure to deliver, and they ask questions hoping to receive the answers they are wishing for. On the other

hand, the people providing the estimates (like the developer in the previous dialogue), sometimes find it easier to just give the answer that will keep the other party happy, and deal with the consequences later.

It doesn't need to be like that. Every time I find myself in a situation similar to the one above - with someone asking for a commitment and calling it estimate - the first thing I do is to make very clear the difference between the two. This always has the effect of changing the nature of the conversation into a more meaningful one. Now that we have some sound definitions, let's have a closer look at the estimation process itself.

What to estimate

When we talk about estimation, usually the first thing that comes to the mind is time. However, estimation is important for any factor that can impact the success of a project, e.g.:

- Time
- Cost
- Size
- Quality
- Effort
- Risk
- Etc.

Often, some of the factors listed above are actually constrained, i.e., they can vary only between a minimum and a maximum value (time and cost are some typical ones). If that's the case the other factors will have to be estimated accordingly, so that either all the constraints are respected, or, if not possible to achieve that, the project goals are modified, or the availability of some resources is increased.

Accuracy and precision

"It's better to be approximately right than precisely wrong". Warren Buffet.

When dealing with estimates, we necessarily deal with issues of accuracy and precision. In general, estimates should be as accurate as possible, but they can't be precise (after all they are approximate measurements). However, these two concepts are often used wrongly as synonyms - in fact a measurement can be accurate without being precise, or precise without being accurate.

In this context, accuracy refers to **how close** to the real value a number is, while precision refers to **how exact** a number is (number of decimal places). For example:

- Accurate. Task x will take between two and four days
- Precise. Task x will take 2.04 days

If task x above is finished in three days, then estimate 1 was accurate (but not precise), while estimate 2 was precise (but inaccurate, and wrong).

When estimating a quantity it is always important to match the precision to the accuracy of the estimate appropriately. For example, estimating in hours of work a project that is going to take a couple of years is likely to be a big mistake since the error is almost certainly going to be bigger than one hour - it is more likely to be a few months - and the high precision can actually be dangerous due to the false sense of confidence it conveys.

Uncertainty

Estimating software is hard for several reasons: big differences in individual productivity, the fact that creative processes are difficult to plan, the fact that software is intangible and so difficult to measure, and the fact that during the life of the project, anything can change - scope of the product, budget, deadlines, or, as often happen in the software world, requirements.

For these reasons, estimates have always a degree of uncertainty. Uncertainty is usually described and managed using probabilities - an estimate, typically, comes with a best, a worst, and a likely outcome with some probabilities attached. Single point estimates always have a probability less than 100% (usually closer to 0% than to 100%).

The best possible accuracy, according to studies conducted by Barry Boehm [Boe], can be described using the cone of uncertainty, which gives a measure of the estimation error depending on the development phase the project is in. In Figure 1 there is the cone for the case of sequential projects with no requirements volatility (they don't change during the lifetime of a project). The horizontal axis represents the phase the project is in, while the vertical one represents the error factor - e.g., in the initial product definition phase the estimate can be inaccurate by a factor of 4 on either side (over or under estimation). This means that if an initial time estimate for the project is 100 days, it could actually take as many as 400 (4x100) or as few as 25 (0.25x100).

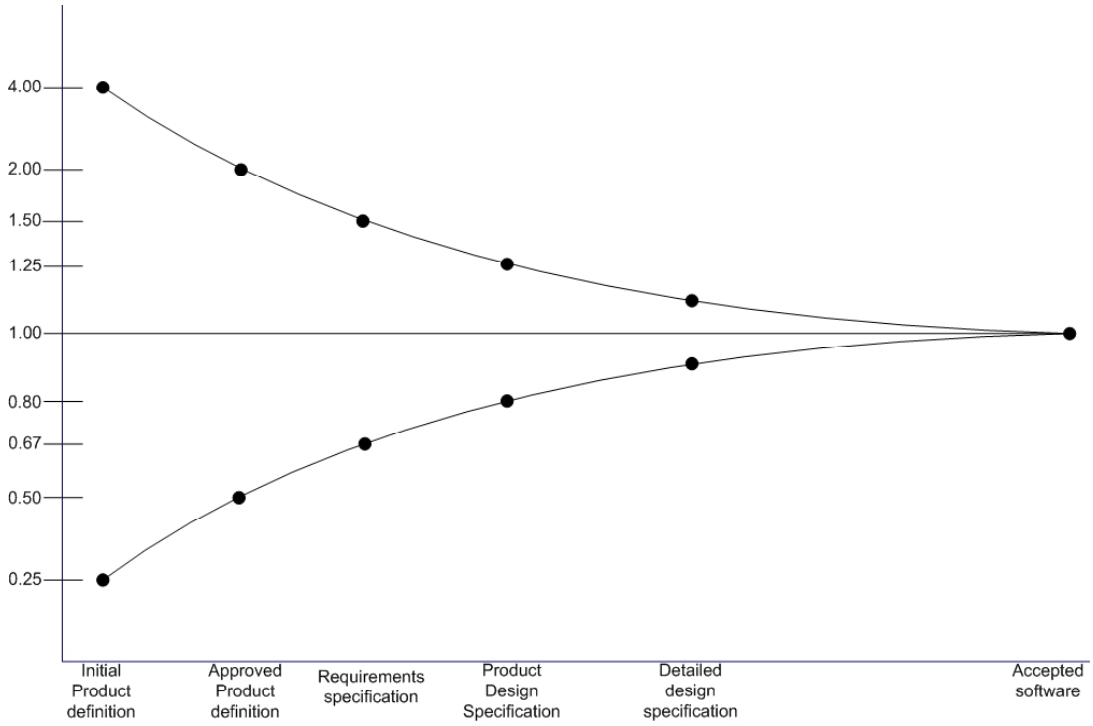

Figure 1: Cone of Uncertainty for a sequential process

As you can see, depending on the phase the software is in, the estimation error can be quite high - up to 400% over or under estimation (and in case of requirement volatility it can be even bigger), but, if the project is well managed, the accuracy increases over time.

The same cone can be used for iterative projects as well, as depicted in Figure 2. The main difference is that there are no phases any more but iterations, and the estimation error goes down iteration after iteration (again, if the project is well managed).

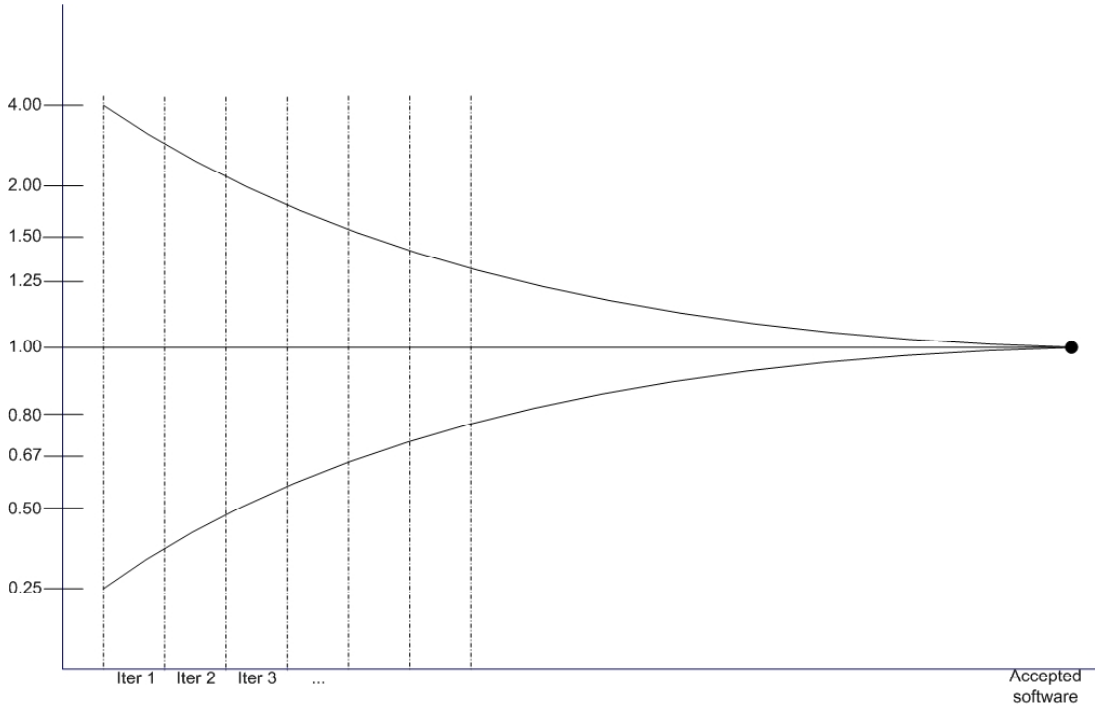

Figure 2: Cone of Uncertainty for an iterative process (Source: [Lar])

It is important to stress a few things.

First, the cone represents best case accuracy. Expert estimators give estimates somewhere inside the cone; less experienced ones can actually go outside the cone. This means that even an expert is bound to over estimate or under estimate a project by 400% during the initial phases of the project.

Second, the cone doesn't narrow itself. The fact that a project is at its 16th iteration doesn't necessarily mean that estimations will be more accurate. In fact, if any of the following is true

- Poor requirements definition
- Lack of user involvement
- Poor design
- Poor coding practices
- Bad planning
- Etc.

Estimates will always have a poor accuracy, and the project will have a high chance of being late and over budget, or, even worse, failing.

Third, estimation is an on-going activity: estimates should be updated whenever there is new information available. This implies that planning is an on-going activity as well. This is independent on the development process used - iterative, waterfall, or something in-between.

Some of you may be wondering why, with this variation, projects are often late and over-budget, and almost never early and under budget - after all the cone of uncertainty says that both things are possible. The answer is, usually, pressure to deliver: in an increasingly competitive world we want to do more in less time and fewer resources. On top of that, it might be difficult to explain to the boss that the project goals are far too ambitious given the available resources, and so people find it easier to say yes to “aggressive” targets.

Making commitments

At some point, even if estimates are only approximations, there will be some decisions to take and commitments to be made. So, when is the best time to commit? As you can see from Figure 3, in the case of a sequential process, a good time is sometime between the completed product definition and the design specification depending on the amount of risk that the stakeholders are willing to take (and the corresponding potential benefits).

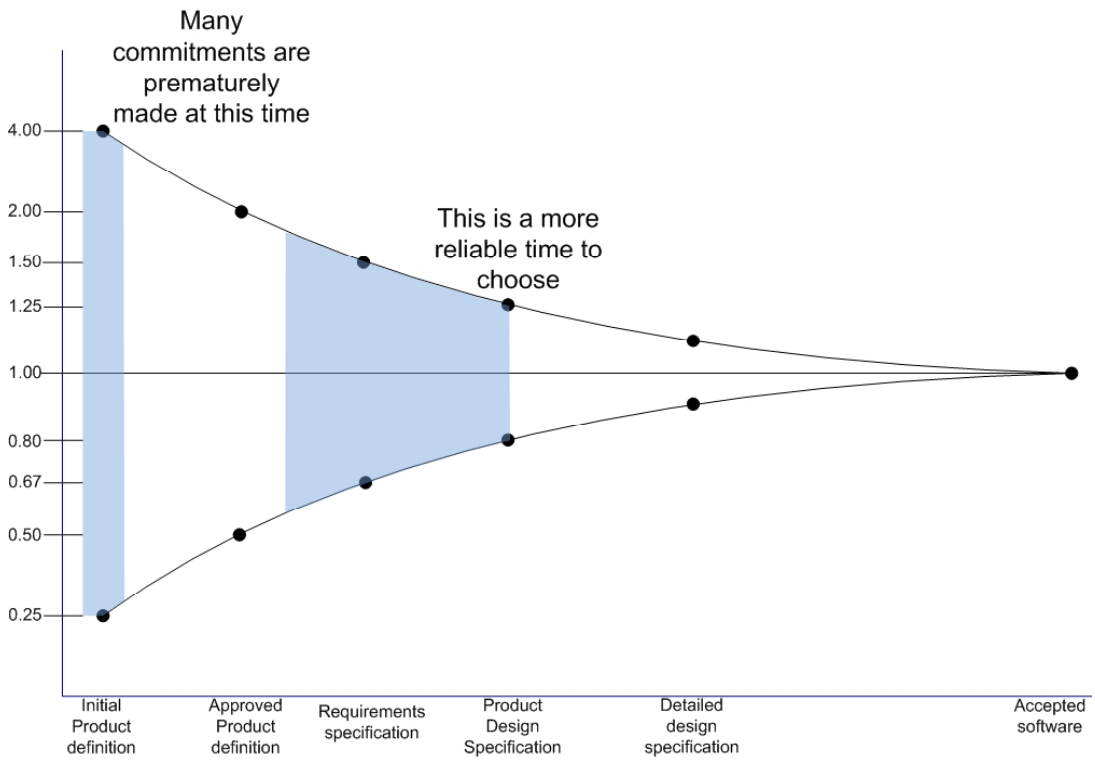

Figure 3: When to commit for a sequential process (adapted from [Lar])

The equivalent example for an iterative process is shown in Figure 4. The main difference is that, in this case, the commitment is made after a few iterations when the stakeholders consider the estimation error acceptable.

In both cases - of the sequential process, and of the iterative one - the initial period of product definition (or the first few iterations) can be used to prototype the most important parts of the product to get a better understanding of what to expect, and to be able to give more accurate estimates, and set more realistic targets and commitments.

Unfortunately, in many cases commitments are made very early when only very few things are known about the product and the potential issues. This is the main reason for time and cost over-runs and also one of the biggest source of defects in the final product - when the deadline gets closer people are prone to cut corners and lower quality in the hope to meet the deadline.

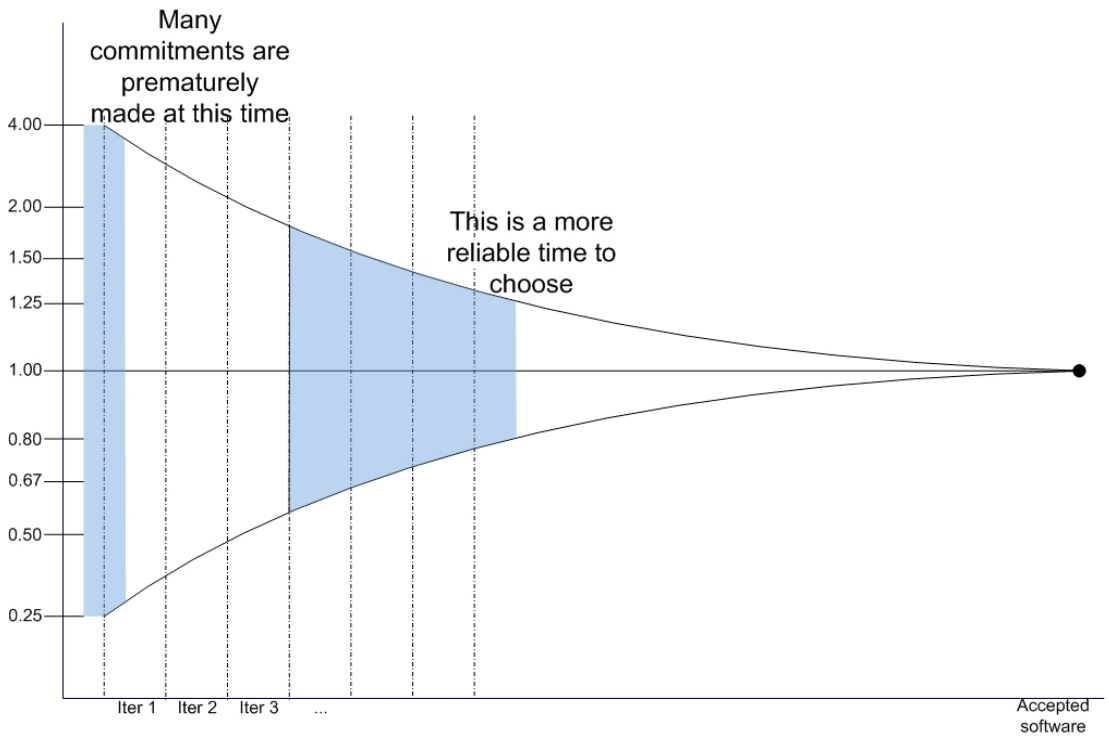

Figure 4: When to commit for an iterative process (adapted from [Lar])

Underestimation and overestimation

Accurate estimates are rare. Is it better to over-estimate or under-estimate?

Both approaches have problems.

Under-estimation tends to reduce the effectiveness of planning, reduce the chance of on-time delivery (developers already tend to be 20%-30% too optimistic), increase the chance of cutting corners, and cause destructive project dynamics which, in turn, will cause overtime, angry customers, and poor quality.

On the other hand, over-estimation tends to cause the cause problems due to Parkinson's law which says that work will expand to fill available time, and to the student's syndrome (adapted to software development) - if developers are given too much time, they'll procrastinate until late in the project, and probably they won't finish on time.

However, as shown in figure 5, over-estimation tends to cause fewer problems than under-estimation. In fact, the penalty paid for overestimation tends to increase linearly, while the one paid for underestimation tends to increase exponentially.

This makes sense also at an intuitive level. For example, when the time for developing the product is underestimated, as the deadline approaches the pressure mounts, the team starts to work over-time and cut corners, but this causes the occurrence of many defects which are very difficult to fix because corners have been cut, and the developers are tired because of the overtime, and due to this even more problems pop up, and so on and so forth - all the issues impact each-other causing an exponential growth in cost effort and schedule, and an exponential loss in quality. In case of overestimation, even if something bad happens, there is usually time to recover, and there are fewer knock-on effects.

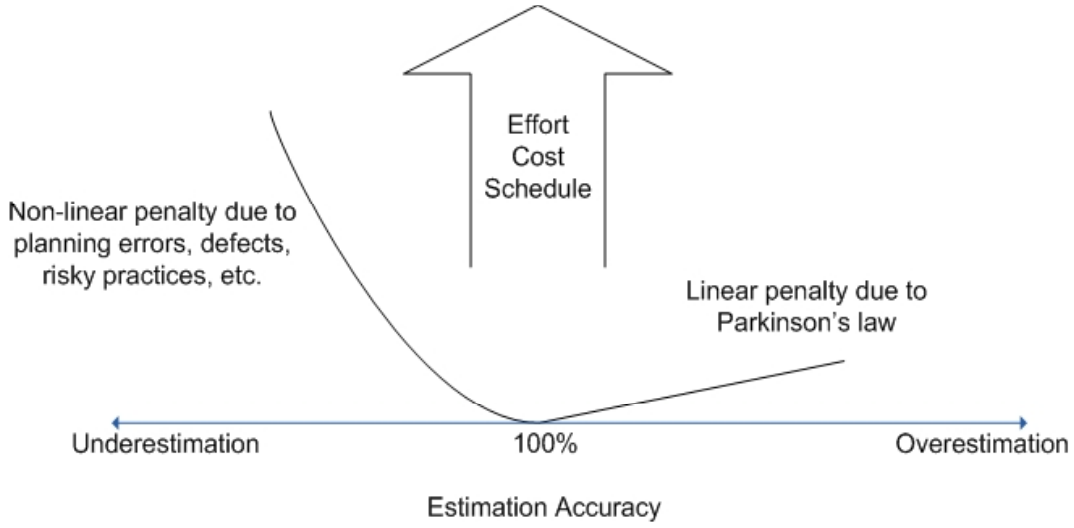

Figure 5: Under-estimation and over-estimation effects (taken from [McC]

Some estimation techniques

Some common estimation techniques include

- Count, compute, and judge. If you can, count, if you can't count compute and use judgement only as a last resort. It can be used during all stages and can have a very high accuracy
- Mathematical models. Based on formulas obtained by studying the data available from past projects, their main advantage is the availability of software tools that can be used for predicting as well as simulating outcomes. Their big disadvantage is that they tend to depend on a very high number of calibration parameters which can be very tricky to configure appropriately. Probably the most famous one is COCOMO II by Barry Boehm [Boe]
- Calibration and historical data. Used to convert counts to estimates - story points to days, lines of code to effort, etc. Calibration uses three kinds of data (depending on what is available): industry data, which refers to data from other companies that develop the same kind of software; historical data, which is data available from the organization running the project; or project data, which is data from the project itself. The availability of historical and project data allows the creation of highly accurate estimates. The main disadvantage is that there has to be some data available to start with
- Analogy. This one is about creating estimates based on a similar (in terms of what is being estimated, e.g., size, complexity, etc.) past projects. Its accuracy is not very high, but it is quite simple to implement. Its main drawback is that it can be highly subjective - different estimators may have different ideas about how similar the projects involved are
- Proxy. Sometimes it is extremely difficult to produce direct estimates, but it is possible to find a proxy correlated to the quantity being estimated which is easier to deal with. A typical example is story points as a proxy for size. This technique can achieve great accuracy, but, to be used effectively, it is necessary to have a fair amount of experience and high discipline
- Expert judgement. This technique is based on the on the judgement of experts - e.g., senior developers who worked on similar systems. It is by far the most widely used, and it comes in various flavours, from individual to group techniques. It can have high accuracy, but it has some important disadvantages - finding the experts may not be easy, and since they are human, they may be subject to external pressure to cut their estimates.

The list above is by no means exhaustive. There are many more listed and explained in much more depth in [McC] and [Stu].

Each technique has its pros and cons. Often, but not always, there is a tension between simplicity, cost, and accuracy - the simpler the technique is to apply the lower are its cost and also its accuracy. Also, the accuracy of each technique, usually, depends on when it is used during the project - early stages, in the middle, or at the end. For these reasons, different techniques can (and should) be used together in order to produce the best results.

A typical approach is to use expert judgement - or, even better, some historical data - at the beginning of the project, and then collect data directly from the project to be used to calibrate and refine the estimates using any of the techniques described above as appropriate. When the project is done, the data collected can be used for estimating future projects in the company, improving the accuracy of the initial estimates and the overall ability of the company to deliver.

It is worth noting that, independently on how you start, the better way to act to improve the accuracy of your estimates is to collect project data as you go and use it to calibrate them. An analogy used by some agile teams to describe this approach is “yesterday's weather” from the observation that today's weather matches yesterday's weather more often than not. In fact, this is a much better approach than the, unfortunately, very common one based on wishful thinking - the we’ll do it better this time syndrome - which assumes that things will go better than in the past only to realize that is not the case when it is already too late.

Final remarks

There are a few very important things which are independent on the estimation techniques used, and are worth remembering when producing estimates.

First of all the quantities being estimated have to bee measurable ones, otherwise the estimation might be impossible. For example how would you estimate the impact on the schedule (or cost, or effort, etc.) of the following requirement (which, incidentally, is not fictional, but comes from a real specification of a system I worked on)?

“The system must be scalable to support future business growth”.

A better one would have been

“The system must be able to support 400 concurrent users with a maximum response time of 1 second”.

Second. The estimates should be provided by the people that have to do the work since they are the ones that are in the best position to know the effort, time, etc. required.

Third. Remember to include everything. For example, time estimates produced by developers should include all technical tasks that should be performed (build environment, computer set-up, supporting the build, etc.); all side tasks like e-mails, meetings, phone calls; time-off due to holidays, possible illness, etc.

Finally, all underlying assumptions should be explicit - availability of resources, expected quality, etc. Even programmers, believe it or not, are able to provide accurate estimates when they remember to include all the things listed above [McC].

Conclusion

In this article I described some of the basics of software estimation.

The most important points to remember are:

- Estimates, targets and commitments are different things
- Estimates are not negotiable
- Estimates cannot be precise, and always have a degree of uncertainty attached
- Estimation is an on-going activity, and new estimates must be made as soon as there is new information available
- There are several estimation techniques, each one with its pros and cons

Of course there is much more to know about this subject - For example, I deliberately avoided discussing all the human issues that can have an impact in the estimation process - and I strongly suggest all the books in the references, in particular [McC] and [Coh] are probably the best ones to get started with.

References

[Boe]: Boehm, Barry W. et al., Software Cost Estimation with Cocomo II, Prentice Hall, 2000

[Coh]: Cohn, Mike, Agile Estimating and Planning, Prentice Hall, 2005

[Lar]: Larman, Craig, Agile & Iterative Development: A Manager's Guide, Addison Wesley, 2004

[McC]: McConnell, Steve, Software Estimation: Demystifying the Black Art, Microsoft Press, 2006

[Stu]: Stutzke, Richard D., Estimating Software Intensive Systems, Addison Wesley, 2005

[You]: Yourdon, Edward, Death March, Prentice Hall, 2004