



THE UNIVERSITY *of* EDINBURGH

Edinburgh Research Explorer

UKHEC Report on Software Estimation

Citation for published version:

Kavoussanakis, K, Sloan, T & Simpson, A 2001, UKHEC Report on Software Estimation. UKHEC, EPCC, University of Edinburgh.

Link:

[Link to publication record in Edinburgh Research Explorer](#)

Document Version:

Publisher final version (usually the publisher pdf)

General rights

Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy

The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.



UKHEC Report on Software Estimation

Project Title: UKHEC

Document Title: UKHEC Report on Software Estimation

Document Identifier: EPCC-UKHEC 0.1

Distribution Classification: Public

Authorship: K. Kavoussanakis, Terry Sloan

Document history:

Personnel	Date	Summary	Version
Kostas Kavoussanakis, Terry Sloan		First Draft	0.1

Approval List: *Head of Applications Group*
Alan Simpson

Contents

1	Motivation	3
1.1	Disasters - The importance of estimation	3
1.2	What determines a successful software project	3
1.3	When to estimate	4
1.4	Relevance to academic research projects	4
1.5	History of Software Estimation	5
2	Projects and Estimates	7
2.1	Software project as a journey	7
2.2	What is a project	7
2.3	What is estimation	8
2.4	Making the estimate	8
3	Estimation Methods	10
3.1	Estimation by Analogy	10
3.2	Expert Opinion	10
3.2.1	Delphi	11
3.2.2	Work Break Structure (WBS)	11
3.3	Top-Down, Bottom-Up	12
3.4	Function Point Analysis	13
3.4.1	Cosmic-FFP	14
3.5	COCOMO	15
3.5.1	COCOMO II	16
3.6	ObjectMetrix	16
4	The Estimation Process	18
4.1	The silver bullet syndrome	18
4.2	Development model	18
4.3	Keep track of your project	18
4.4	Estimate in ranges	18
4.5	What's the day today?	19
4.6	Make sure	19
4.7	Bring in the right people	19
5	Estimation Tools	20
5.1	KnowledgePLAN	20
5.2	Costar	20
6	Further sources of information	21
6.1	General Books	21
6.2	WWW Sites	21

Motivation

1.1 Disasters - The importance of estimation

The history of the computer industry is littered with war stories concerning projects which have dramatically over-run their budget and failed to meet the client's expectations. As highlighted by McConnell in [1], several surveys have found that about two-thirds of all projects substantially overrun their estimates. Further, that the average large project misses its delivery date by 25 to 50 percent, and the size of the average schedule slip increases with the size of the project. The following table, from Boehm [2], lists the first and last cost and schedule estimates for a number of failed projects and helps to illustrate the pitfalls of inaccurate estimation.

Table 1: Software Overrun Case Studies from Boehm [2]

Project	First;Last Estimate		Status at Completion
	Cost (\$M)	Schedule (months)	
PROMS (Royalty Collection)	12;21+	22;46	Cancelled, Month 28
London Ambulance	1.5;6+	7;17+	Cancelled, Month 17
London Stock Exchange	60-75;150	19;70	Cancelled, Month 36
Confirm (Travel Reservations)	56;160+	45;60+	Cancelled, Month 48
Master Net (Banking)	22;80+	9;48+	Cancelled, Month 48

These case studies beg the obvious question why do these disasters keep happening. According to Construx Software Inc. [3] a prime cause of project overrun is the inherent difficulty of software estimation. Construx believe that effective software estimation is one of the most difficult software development activities and one of the most important.

Construx further state that underestimating a project will lead to

- under staffing,
- under scoping the quality assurance effort, and
- setting too short a schedule.

These in turn could lead to

- staff burnout,
- low quality, and
- loss of credibility as deadlines are missed.

On the other hand, over-estimating can be almost as bad: Parkinson's Law that work expands to fill available time comes into play, which means that the project will take as long as estimated even if the project was overestimated [3].

It is clear therefore that an accurate estimate is a critical part of the foundation of an efficient software project.

1.2 What determines a successful software project

The end of a software project is generally the point at which a project is judged to be a success or not. Typical criteria for determining success are listed below.

- Did the project finished on time ? In this report we will refer to this criteria as Time.
- Did the project finish within its budget ? We will refer to this as Cost.
- Did the software successfully do what the client wanted ? We will refer to this as Quality.

There are other criteria of course, such as is the client still talking to you, however these are generally additional to the Time, Cost and Quality criteria.

1.3 When to estimate

For many software projects, the time available, the budget available and the functionality to be implemented are determined before the start of the project. This is unfortunate since for most projects, this is when least is known about the work to be undertaken.

According to Sommerville [4], there is no simple way to make an accurate early estimate of the effort required to develop a software system. Initial estimates may have to be made on the basis of high level user requirements. The software may have to run on unfamiliar computers or use new development technology. The people involved in the project and their skills will probably not be known. All of these factors mean that it is difficult to produce an accurate estimate of system development costs at an early stage in the project [4].

This is further highlighted by McConnell [1] who states that the basic software estimation story is that software development is a process of gradual refinement. At the project start there is a fuzzy picture of what is to be built. The remainder of the project is spent trying to bring that picture into clearer focus. Since the picture of the software being built is fuzzy, the estimate of the time and effort needed to build are also fuzzy. Only when the software itself comes into focus will the estimate also become clear. Hence the argument that software project estimation is also a process of gradual refinement.

McConnell [1] goes on to say that at the time when developers are typically asked to provide a rough estimate, there can be a factor of 16 difference between high and low effort estimates. Further that even after requirements have been completed, you can only know the amount of effort needed to within about 50 percent.

Figure 1 (adapted from [4]) illustrates this changing uncertainty in the estimates from the project start to the final delivery. Due to the shape of the graph, this feature of estimation is sometimes referred to as the 'cone of uncertainty' [5].

The cone of uncertainty implies that it is not only difficult to estimate a project accurately in its early stages, it is theoretically impossible [5]. The person who claims to be able to estimate the attributes of a project even after the requirements phase is either a prophet or not very well-informed about the nature of software development [5].

Humphrey [6] illustrates this point in a more colloquial fashion.

"Projects usually get into trouble at the very beginning. They start with impossible dates, and nobody has time to think, let alone do creative or quality work. All that counts is getting into test, and the rush to test invariably produces a hoked-up product, a poor quality program, a late delivery, an unhappy customer, and disgruntled management."

Over the years this has been recognised and a number of estimations methods have been developed to help with this unknown. Further a number of software development techniques have also been developed to help manage the uncertainties in these estimates. Typically estimation methods are used in combination with these development techniques to try and ensure successful project outcomes.

1.4 Relevance to academic research projects

The idea to find an undisputed method to understand the size of a software project was hatched in the late sixties. However, up until today there is no undisputed, valid for all types of projects metric. The area is

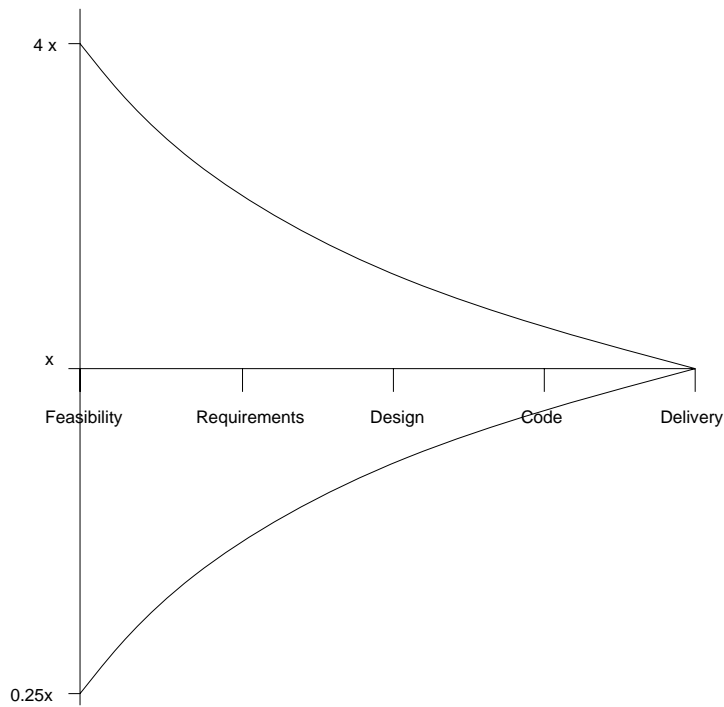


Figure 1: Uncertainty in estimates through a project's lifetime (Adapted from [4])

active both in terms of research and application, which leaves great hopes for the future.

However, the effort is concentrated where the money is, ie in MIS and recently in real-time systems. It is not just that money does not flow freely in academic environments. Projects which involve complex computations and the invention of challenging algorithms are even more complex and challenging to predict.

The facts above do not render this report out of context for academic environments. On the contrary, good estimation practices, applied with reason as explained in section 3, can keep the project on track and even earn some time for the tricky, interesting areas.

Additionally, the time, cost and quality are all defined in a grant proposal probably written more than a year before the project start. In the intervening period, many factors may have changed that directly affect these criteria. However, it is typical of funding bodies to not welcome modifications to the schedule or deliverables. This makes it even more important to standardise the estimation process first inside the individual research centres and then (quite a big step) across the centres in the UK. Such a standardisation can lead to better estimates and maximise the chances of success. A common culture can only benefit the exchange of ideas, personnel and even estimates.

1.5 History of Software Estimation

Boehm [7] contains a description of the history of software estimation. This section is essentially an extract from this.

Significant research on software cost modelling began with the extensive 1965 study of the 105 attributes of 169 software projects [8]. This led to some useful partial models in the late 1960s and early 1970s.

The late 1970s produced a flowering of more robust models such as SLIM [9], Checkpoint [10], PRICE-S [11], SEER [12] and COCOMO [13]. Although most of these researchers started working on developing models of cost estimation at about the same time, they all faced the same dilemma: as software

grew in size and importance it also grew in complexity, making it very difficult to predict accurately the cost of software development. This dynamic field of software estimation sustained the interests of these researchers who succeeded in setting the stepping-stones of software engineering costs models.

Just like in any other field, the field of software engineering cost models has had its own pitfalls. The fast changing nature of software development has made it very difficult to develop parametric models that yield high accuracy for software development in all domains. Software development costs continue to increase and practitioners continually express their concerns over their ability to predict accurately the costs involved. One of the most important objectives of the software engineering community has been the development of useful models that constructively explain the development life-cycle and accurately predict the cost of developing a software product. To that end, many software estimation models have evolved in the last two decades based on the pioneering efforts of the above mentioned researchers. The most commonly used techniques for these models include classical multiple regression approaches. However, these classical model-building techniques are not necessarily the best when used on software engineering data as illustrated in this paper.

Beyond regression, several papers [14] [15] discuss the pros and cons of one software estimation technique versus another and present analysis results.

Table 2 lists a number of examples of estimation techniques. These are classified into the following categories.

Model-Based These are techniques with a mathematical model as their cornerstone. They involve an algorithm which is most of the times derived by fitting data points from known projects. It is probably the most widely exercised method.

Expertise-Based These rely on the opinions of experts who have past experience of the software development techniques to be used and the application domain.

Learning-Oriented This covers manual estimation by analogy with previous projects through to the use of artificial intelligence techniques such as neural networks to produce estimates.

Dynamics-Based These techniques explicitly recognise that the attributes (eg. staff effort, skills, costs) of a software project change over its duration.

Table 2: Classification of some Software Estimation Techniques Boehm [7]

Classification	Estimation Techniques
Model-Based	SLIM COCOMO Checkpoint SEER
Expertise-Based	Delphi Rule-Based
Learning-Oriented	Neural Case-based (Estimation by analogy)
Dynamics-Based	Abdel-Hamid-Madnick
Composite	Bayesian-COCOMO II

Further information on the history of Cost Estimation can be found at [16].

In this report we will mainly focus on model-based and expertise-based techniques, as these are the ones most widely used. Function Point Analysis (see section 3.4 is not mentioned in Boehm's classification. We assume that the reason for that is that FPA is a sizing metric, as opposed to an estimation technique.

2 Projects and Estimates

2.1 Software project as a journey

Imagine driving on an important trip to a distant place you have not been before. No-one in their right mind set off to such a journey without knowing at least the name and the general direction of the destination. Other important considerations are the distance and the available routes that take you there. Armed with this information and a good map one can feel more comfortable about taking the trip.

The above are essential, however your comfort may be seriously compromised during the journey. Roadworks can hit you badly, but at least you have a good chance of having early warning, if you think to check. A puncture or a car fault can be less predictable, but they do happen. Timing can be important, who wants to cross London at 17:15 on a Friday afternoon?

We are not trying to put you off driving, after all driving somewhere is quite trivial these days. We are not exaggerating if we say that it is easy, the target is steady, there is plenty of information and if you are lucky, a friend or a colleague may have done it in the past so you have access to plenty of data. However, we wrote two paragraphs about the preparation without trying too hard.

Setting off on a software project is a lot harder. The biggest difference is that no matter how hard you try, the target is moving. This is inevitable as the product has a reason of existence only if it offers some advantage and often the original specification is superseded by events. The routes are not definite and the unlucky events are mostly unpredictable but also more frequent.

There are however striking similarities. Knowing the general direction is necessary in both cases. This is tackled fine in most projects, as the need for requirements capture and early design has been unquestionable in the past years. The timing is important, but this is not often acknowledged early enough. And, unlike what most people think, the distance to the target can be estimated. This is different to knowing the exact mileage, however route diversions due to unforeseen circumstances offer an analogue.

We will focus on software projects in the forthcoming sections.

2.2 What is a project

In any project, software or other, there are three conflicting factors: Duration, Cost and Quality¹. These are often viewed as a triangle (see figure 2). One cannot for example lessen the duration without affecting cost and/or quality.

In software engineering terms, functionality is the counterpart for quality, while duration and cost are still applicable. The correctness of the code (lack of bugs) is bundled into quality for simplicity.

The attributes we consider for software projects are *Effort* and *Schedule*. If these are available, then the cost and time can be defined.

Effort is the total number of time units (e.g. weeks or months) needed to complete the task. This may break down to effort from more than one persons, so as to take advantage of certain skills and parallelise the work to gain overall time.

The caveat here is that the more people one adds to a project the more one needs to work so as to coordinate them and the more they communicate so as to interact successfully, thus yielding overheads. This coordination may also lead to occasional idle periods of some of these people. *Schedule* is sometimes associated with the total time for the project, however the term usually includes the breakdown of effort per person at any given time, so as to track the coordination issues. It is a fact that the schedule is derived from effort.

¹To be precise, one wants to minimise duration and cost and increase quality, so it may make sense to consider quality⁻¹ to be the third factor.

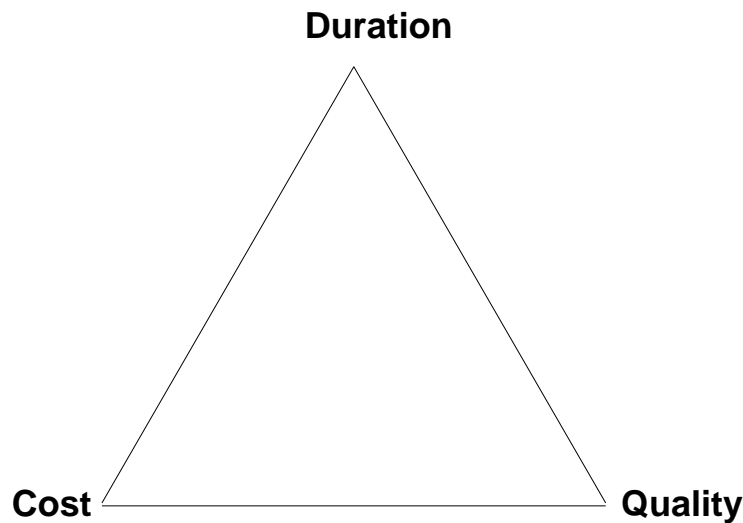


Figure 2: The software trade-off triangle (Adapted from [1])

2.3 What is estimation

It is essential for a project manager to know the effort, schedule and functionality of a project in advance. Perhaps there is no point in starting a project when there is not enough time to finish it or enough money to fund it or if the quality is so inadequate that the end product will be useless and unmarketable. However, the project factors change in the duration of the project, and they may change a lot. The worse thing is that one can seldom predict how they will change, yet we need to know all these before we start.

This is why we *estimate* software projects. There is no way to calculate in advance and expect the initial values to be correct. This does not render the estimates vain. On the contrary, it calls for better quality estimation techniques, which will yield more accurate early results and guide us to more targeted and thus effective contingency plans.

Software estimation is the act of predicting the duration and cost of a project. It is a complex process with errors built into its very fabric, however it is very rewarding when done the right way.

The estimation process does not finish until the project finishes. This is the answer of the project manager to the ever changing conditions of the project. It is the norm for project estimates to be quite off the final figures and become better as the project progresses and the information becomes more solid. In [17], it is stated that in the early stage of a projects, feasibility study, one can underestimate the size of a project by up to 4 times its final size, or overestimate it by the same range. However, the numbers can drop to 25% by the time the design document is finalised.² See also figure 1 for a graphic representation of the above.

2.4 Making the estimate

The usual approach taken for estimation is to set out and analyze the functional requirements and then derive the effort and schedule. This is where things become difficult and elusive. The functional requirements do not provide a solid background so as to deduce the explicit numbers that one needs for the effort definition. Even if the detail of the requirements has been set out, it is not immediately known how long it will take to develop the features, especially when the desired outcomes are genuinely novel. The biggest killer is *feature creep*, the unpredictable yet near-certain change of the functionality as the project progresses.

²The design document also evolves with time; the version that we are referring to is the last before implementation starts.

There are many more parameters involved, ranging from staff ability, to code reuse (both whether code is reused and whether the code written is to be reused in the future) to the language employed. Even if a similar project has happened in the past in an organisation, it is particularly hard to qualify what parts of it are the same as the current one and even harder to mix and match prior project components. Finally, the ability of the person or persons who estimate the projects varies, making it impossible to rely on the process both on a per project but more importantly on a per organisation level.

The idea to employ *software metrics* was cast. Instead of basing the estimates on the fuzzy description of the final functionality, software engineers looked for a solid backdrop to the process, a more wholistic approach. There is no definition of the project *size* (other than through the selected metric), however one could say that it corresponds to the area of the project triangle. It thus encompasses all three sides of the triangle and offers the solid, wholistic cornerstone of the estimation process.

Finding objective measurements of the size of the project has so many advantages. The process becomes transparent, repeatable and reliable. By transparent we mean that the process becomes so clear that the abilities of the practitioner can be rendered irrelevant. By repeatable we mean that the same process can be exercised by various people at different times and yield the same results. By reliable we mean that the end results, although never accurate, can be closer to the truth than any intuitive, esoteric method. The additional bonuses are there too. The productivity of staff and organisations can be monitored. The same results can be shared across the globe.

Section 3 of this report will focus on estimation techniques and examine one of the sizing metrics developed, Function Point Analysis in section 3.4. Section 4 sees estimation as a process and attempts to highlight good practices.

3 Estimation Methods

There are various methods for estimation, usually classified as in Table 2. The applicability of these methods to one's needs is debatable; it is quite common in bibliography to suggest trying various methods in parallel, especially when trying these for the first time, but also in general.

In this section we will list methods from various categories, starting with expertise-based ones and then proceeding with model-based methods. We also discuss Function Point Analysis, a sizing metric which can be used as the starting point of model-based methods.

3.1 Estimation by Analogy

This technique is generally applicable when other projects in the same application domain have been completed. The cost of a new project is estimated by analogy with these completed projects [4]. Construx [3] state that estimates based on such historical data from within an organization are more accurate than estimates based on rules of thumb or educated guesswork.

However, not all organisations have sufficient historical data to satisfactorily use analogy as means for estimation. For this reason, the International Software Benchmarking Standards Group (ISBSG) maintains and exploits a repository of international software project metrics to help software and IT business customers with project estimation, risk analysis, productivity and benchmarking [18]. ISBSG provide various publications and software tools to assist with the analysis of their data repository. At present this repository can be supplied on CD-ROM. It currently contains information on over 1200 projects. ISBSG also supply information on the types of metrics to collect from past projects in your own organisation which can be used in future estimations.

It should be noted that when using historical data that the important differences between past and future projects must be taken into account.

At a glance:

Pros:

- Can be accurate.
- Simple if the organisation repeats similar work.
- Estimates are immediately available.
- Encourages detailed documentation.

Cons:

- Can be very unreliable.
- It is very hard to assess the differences between the environments and thus assess the accuracy of the ISBSG data.

3.2 Expert Opinion

The contents of this section are extracted from Boehm [7].

Expertise-based techniques are useful in the absence of quantified, empirical data. They capture the knowledge and experience of practitioners seasoned within a domain of interest, providing estimates based upon a synthesis of the known outcomes of all past projects to which the expert is privy or which he or she participated. The obvious drawback to this method is that an estimate is only as good as the expert's opinion, and there is no way usually to test that opinion until it is too late to correct the damage if that opinion proves wrong. Years of experience do not necessarily translate to high levels of competency. Moreover, even the most highly competent of individuals will sometimes simply guess wrong. Two

techniques have been developed which capture expert judgement but that also take steps to mitigate the possibility that the judgement of any one expert will be off. These are the Delphi Technique and the Work Breakdown Structure.

3.2.1 Delphi

The Delphi technique [19] was developed at the Rand Corporation in the late 1940s originally as a way of making predictions about future events - thus its name, recalling the divinations of the Greek oracle of antiquity, located on the southern flank of Mt. Parnassos at Delphi. More recently, the technique has been used as a means of guiding a group of informed individuals to a consensus of opinion on some issue. Participants are asked to make some assessment regarding an issue, individually in a preliminary round, without consulting the other participants in the exercise. The first round results are then collected, tabulated, and then returned to each participant for a second round, during which participants are again asked to make an assessment regarding the same issue, but this time with the knowledge of what the other participants did in the first round. The second round usually results in a narrowing of the range in assessments by the group, pointing to some middle ground regarding the issue of concern. The original Delphi technique avoided group discussion; the Wideband Delphi technique [13] accommodated group discussion between assessment rounds.

This is a useful technique for coming to some conclusion regarding an issue when the only information available is based more on 'expert opinion' than hard empirical data.

3.2.2 Work Break Structure (WBS)

Long a standard of engineering practice in the development of both hardware and software, the WBS is a way of organizing project elements into a hierarchy that simplifies the tasks of budget estimation and control. It help determines just exactly what costs are being estimated. Moreover, if probabilities are assigned to the costs associated with each individual element of the hierarchy, an overall expected value can be determined from the bottom up for total project development cost [20]. Expertise comes into play with this method in the determination of the most useful specification of the components within the structure and of those probabilities associated with each component.

Expertise methods are good for unprecedented projects and for participatory estimation, but encounter the expertise-calibration problems discussed above and scalability problems for extensive analyses. WBS-based techniques are good for planning and control.

A software WBS actually consists of two hierarchies, one representing the software product itself, and the other representing the activities needed to build that product [13]. The product hierarchy (see figure 3) describes the fundamental structure of the software, showing how the various software components fit into the overall system. The activity hierarchy (see figure 4) indicates the activities that may be associated with a given software component.

Aside from helping with estimation the other major use of the WBS is cost accounting and reporting. Each element of the WBS can be assigned its own budget and cost control number, allowing staff to report the amount of time they have spent working on any given project task or component, information that can then be summarized for management budget control purposes.

Finally if an organization consistently uses a standard WBS for all its projects, over time it will accrue a very valuable database reflecting its software cost distributions. This data can be used to develop a software cost estimation model tailored to the organization's own experience and practices.

At a glance:

Pros:

- Applicable to very original projects.
- Inherent local calibration.

- WBS can lead to a well documented process.

Cons:

- Big dependence on experts' abilities.
- Big dependence on experts' presence; how easy does Delphi cope with high staff turnover in a small organisation?

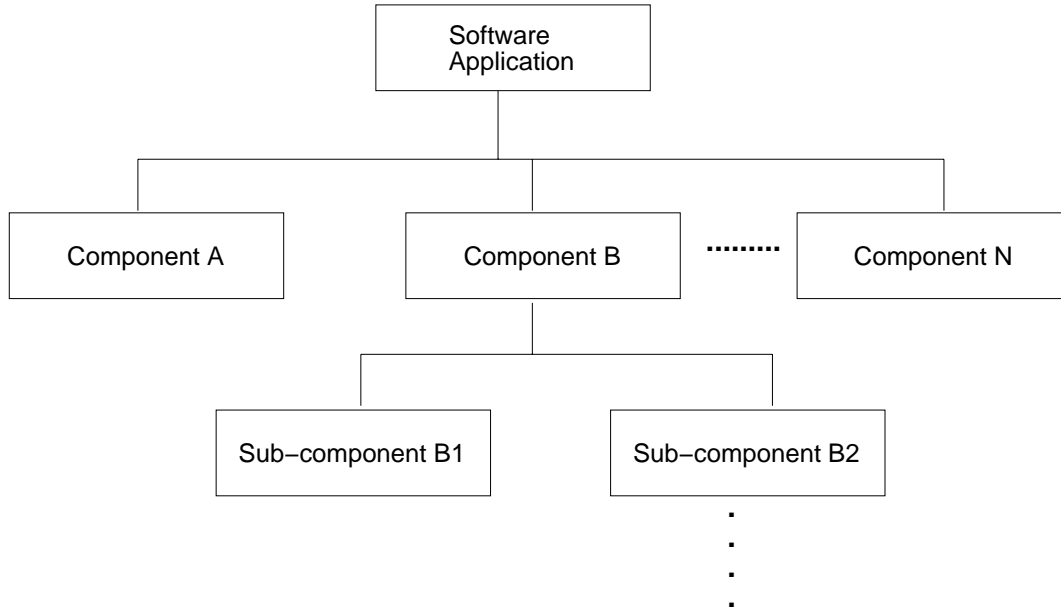


Figure 3: Product work breakdown structure (Adapted from [7])

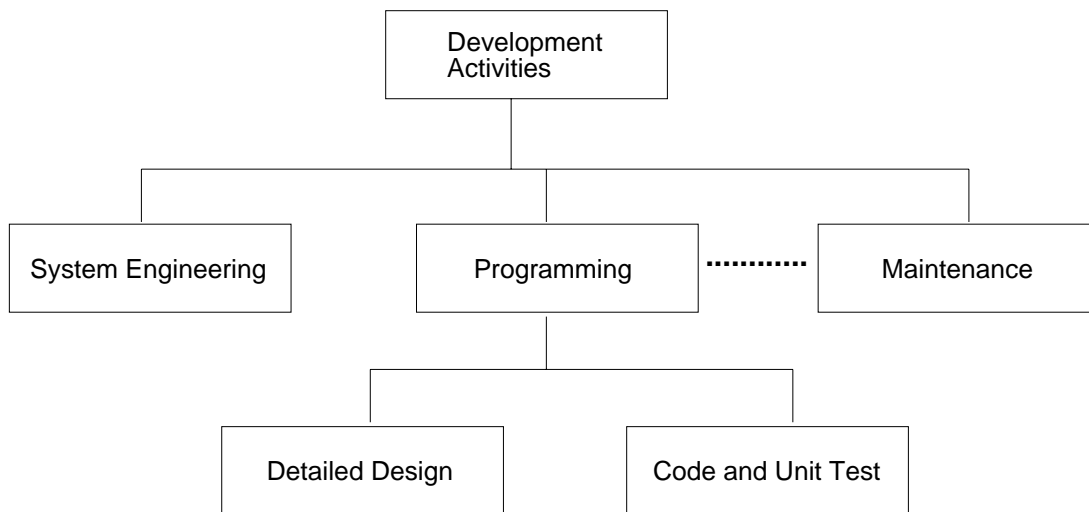


Figure 4: Activity work breakdown structure (Adapted from [7])

3.3 Top-Down, Bottom-Up

As explained in Sommerville [4], expert judgement and estimation by analogy can be tackled using either a top-down or bottom-up approach.

Top-down This approach starts at the system level. The estimator starts by examining the overall functionality of the product and how that functionality is provided by interacting sub-functions. The costs of system-level activities such as integration, configuration management and documentation are taken into account [4].

Bottom-up By contrast, this approach starts at the component level. Here, the system is decomposed into components and the effort required to develop each of these is computed. These costs are the added to give the effort required for the whole system development [4].

The disadvantages of the top-down approach are the advantages of the bottom-up approach and vice versa. Top-down estimation can underestimate the costs of solving difficult technical problems associated with specific components such as interfaces to non-standard hardware. By contrast, bottom-up is also more expensive since there must be an initial system design to identify the components to be costed [4].

3.4 Function Point Analysis

A widespread metric to count the size of projects is source lines of code (SLOC). It is very intuitive to use this as a measurement of size, since after all, this is what one sees at the end of a project or even at the end of a phase.

The method has various drawbacks. Although SLOCs can be counted in the end, it is quite hard to express a prediction of the size of a project in lines of code before a single line is written. And this is when one needs the estimate the most. In addition to that, there have been disagreements about what is a line of code. Do comments count? If yes isn't it generally harder and more time consuming to write a line of code? If no, then why bother? The other issue is counting lines with multiple instructions (e.g. `d = c++` in C, which is tantamount to `c++; d = c`, the latter being two lines, the former being more succinct and usually desirable). If the above and other minor points are not taken into account there can be no universally accepted counting process, and if they are paid attention the process becomes very cumbersome and tedious.

Function Points were introduced in 1977 by A. Albrecht of IBM to measure "the size of computer applications and the projects that build them" [21]. Function Points measure the size of a project based on the features that the developers have to implement. Contrary to SLOC, the expected functionality of a project is more or less available early in the project lifecycle, and definitely before starting to code. There is a specified framework for the counting of these features to ensure that nothing is left out. Counting also yields a classification of these features in one of the following five categories:

- **Inputs:** Instance through which the user can alter a program's data.
- **Outputs:** Instance generated by the program for use by the user or another program.
- **Inquiries:** The combination of an input and its resulting output. Outputs are inquiries requiring processing and possibly sophisticated formatting.
- **Logical Internal Files:** Logical blocks of data or information unreachable by the user belonging solely to the program.
- **External Interface Files:** Logical blocks of data originating from or addressed to other programs (not the user).

At the same time an estimation of the level of complexity (low, medium or high) is also performed. Having classified the features to be implemented, the practitioner uses tables to weigh the features according to the perceived difficulty of development. The summation of the weighted features yields the *Unadjusted Function Point Count*. The final step is to evaluate the necessity of 14 listed factors (which include reusability, multiple site usage, online data entry etc) and the identification of the end-product as either development, enhancement or application. These are parameters in a formula which is applied to the unadjusted count and yield the final Function Point count.

The paragraphs above are an over-simplistic attempt to summarise the 200 pages consisting the Function Point counting manual. Although they cannot be used as a prescription for someone to count, they can form the basis for discussion. One sees an immediate drawback from the terminology used. Inquiries and external interface files are very relevant to database applications, but not so much for real-time systems or physical and mathematical problem solvers. However, the feature-based approach has undoubted benefits. The most immediate one is that no-one can count FPs unless they have thought about the end-product very well. Moreover, the desirable features (or a first approximation of them) are available very early in the project life-cycle to allow for a rough estimate-in the early stages estimates can only be rough anyway.

Many people seem to show disbelief in the numbers and formulae applied. Where do these come from? Why are they valid for everyone? The first thing to note is that Function Points only estimate size, not effort. The effort differs from developer to developer and from site to site. The framework provides a standard method for any number of (qualified) people to look at a proposed project and extract more or less the same result for its extent. It is up to the practitioner then to look at historic data in their environment both for similar projects and the abilities of the developers to be involved and derive the effort and scheduled for the project in question. It is important to note that the metric is not influenced by the choice of language, tools used or the target platform.

Perhaps the biggest impediment for FPA is the fact that the process is manual and very time-consuming [22]. The abilities of various consultants ranges from 100 to 500 Function Points per hour.

The main problem of limited applicability is still valid. Attempts to modify the framework so as to adopt them for other types of applications have led to fragmentation. The International Function Point Users Group is the prevalent group of users. It both leads the effort to keep the method up to date with the modern developments and the training of qualified counting personnel. Moreover, there is an effort to unify the various approaches into an ISO standard (ISO/IEC 14143-1998), the Functional Size Measurement. The effort is underway. One of the big strengths of FPA is the existence of the IFPUG which meets twice annually and reviews the process, although the standard FPA method has lost momentum as software development has extended towards other directions.

3.4.1 Cosmic-FFP

The Cosmic-Full Function Points (FFP) method was developed in 1997 to address the shortcoming of FPAs not being applicable to real time systems. This category of applications includes those which involve communications or are event-driven. This is a great improvement to FPAs, as the category includes applications for everyday life (from lifts to telephone exchanges) and is still applicable to the data-intensive MIS software the FPAs always addressed. However, as it is clarified very early in the "Cosmic-FFP Measurement Manual" [23], it still misses projects which involve tricky computations, extensive calculations and complex problems, like weather forecasting and simulations.

Not all hope is lost though, according to the manual, one can define local extensions to cover such cases. These extensions are nothing but identified special cases for which a number of *functional size units (FSUs)* is agreed *across the organisation*. Obviously this requires a well organised archive of historic information about the Centre and some time for the unit weight to be tested. The manual also prescribes a different way to report such extensions.

The method is under development. It has recently undergone field tests for acceptance (published in April 2001) but these were by no means exhaustive [24]. The effort is embraced by the IFPUG but progress is slow.

At a glance:

Pros:

- Prescriptive, reliable method for sizing.
- Can be applied reasonably early in project lifetime.
- Immune to language and platform idiosyncrasies.

- Large user base-active effort.

Cons:

- Manual, fiddly process.
- Disagreement about its applicability across the various types of modern projects.
- Not ideal in the requirements capture period; although the method can be applied, it is too much work for such a volatile description.

3.5 COCOMO

The CONstructive COSt MOdel (COCOMO) was launched in 1981 by Barry Boehm [13]. It is often mentioned as COCOMO 81 to discern from its followup, COCOMO II, discussed below. The model assumes that the size of a project can be estimated in *thousands of delivered source instruction* and then uses a non-linear equation to determine the effort for the project. The formula is

$$Effort = a * size^b$$

for "large" projects and

$$Effort = a * size + b$$

for projects that can be achieved with 2-3 people teams.

The parameters a and b are those that hold the interest. The idea is to define them based on the characteristics of your project. These characteristics are then compared with historical data to yield the right numbers for these parameters. The model is really simple, as we will explain shortly. All the work has gone into working out the tables of numbers for the two parameters.

First the executioner has to decide whether the project is Organic, Embedded or Semi-Detached. This translates into "simple and familiar", "complex and/or novel" and "something in between" respectively.

Then they can choose between the Basic, uncomplicated and less precise (and perhaps less accurate as well) method or the Intermediate one. In the first case they just substitute the numbers from the table. So, according to the Basic method, a semi-detached project whose size is estimated to 1,000 delivered source instructions can be done in $3 * 1 + 1.12 = 4.12$ man months (using the small project equation).

The Intermediate method requires a bit more work to define the *Effort Adjustment Factor*, which is multiplied with the rest of the effort equation. The EAF is also worked out from tables. The user needs to think about a series of project characteristics (*cost drivers*) in the categories "Product Attributes", "Computer Attributes", "Personnel Attributes" and "Project Attributes". There are a number of characteristics in each of the categories for which the user must decide whether their impact is pronounced or not in a scale of 1-6. For each of these decisions yields a factor which is multiplied with all the others to give the EAF. Pretty simple really.

The Detailed method is a refinement to the Intermediate, separating the application of the cost drivers to Module, Subsystem or System level. We will not go into much detail, however we can point out that this model potentially offers better accuracy at the cost of harder work.

The issues with the method are similar to those for FPA. Again this is an empirical method with numbers representing the projects from which they were extracted. It was also introduced in 1981 and since then the application of software has changed significantly in scope. COCOMO II was introduced in 1997 and revised in 1998, 1999 and 2000. The revisions of COCOMO II only refer to the calibration of the model, however COCOMO II differs a lot from COCOMO 81.

3.5.1 COCOMO II

The biggest problem with COCOMO 81 was that modern software engineering practices obsoleted basic terminology and, more importantly, fundamental concepts. For example, the use of iterative development models made the definition of the start and end of a project phase obscure and the simple, waterfall-based algorithms of COCOMO 81 could not take that into account. The Organic, Semi-Detached and Embedded almost ad hoc classification has been replaced with five parameters (Precedentedness, Development Flexibility, Architecture or Risk Resolution, Team Cohesion and Process Maturity) thus taking into account more and more modern factors and allowing a fine tuning to the exponent.

COCOMO II [2] employs the Application Composition, Early Design, and Post-Architecture models in place of Basic, Intermediate and Detailed, thus discerning between the various stages in a project when each of the models would be more appropriate. The Application Composition model also allows for modern techniques like GUI prototyping to be taken into account. The author believes that the Detailed model has been dropped altogether from COCOMO II, however the models are listed above in order of detail that they provide.

The method has taken steps towards helping with the sizing of the project. Source lines of code are not useful when visual programming languages are employed and FPs are too fiddly. The COCOMO II developers have embraced *Object Points* [25]. Object Points are a count of the screens, reports and third generation language modules developed in the application, each weighted by a three level (simple, medium, difficult) complexity factor. For the purposes of COCOMO II the metric has been renamed "Application Points".

The year 2000 calibration of the model contains data from 161 projects. This is not a big enough number to inspire much confidence and there is always the issue with the correctness of the size input. However, the method's ability to calibrate with local data makes it pertinent in more environments.

At a glance:

Pros:

- Live effort.
- Transparent algorithm.
- Local calibration of methods possible.
- Free implementation available.

Cons:

- Too much dependency on the size input.
- Small data set for the parameter heuristics.

3.6 ObjectMetrix

ObjectMetrix [26] is a recent technique. It is an object-oriented method designed to support the modern, iterative software development models. Just like modern languages, it focuses on data rather than functionality to deduce the size of a project. The model discerns between GUIs ("user-facing business applications") and component-based backends.

The estimation comes in iterations. A crude description of the project (such as the one available from the User Requirements document) yields a "base metric". This assumes an iterative development lifecycle for each of the identified components and, we are led to believe, emanates from historical data.

In a next iteration this development lifecycle is broken up to activities (such as planning, design, programming etc) allowing the user to decide which of these will be carried out. The base metric per component is divided amongst the activities. This breakdown is the "activity profile" for each component.

The activity profile can then be weighted by means of "qualifiers". These include complexity, size, the team (size and ability) and others. An important one which makes a difference from other methods is reuse, accounting for both the positive impact of reusing code and the adverse effect of writing good, clean code for later reuse. The model can then use the weighted data to produce duration and cost estimates for the project.

ObjectMetrix is a commercial model. It is the underlying method for Optimize, the project management tool of the Object Factory (www.theobjectfactory.com).

The software boasts high accuracy and applicability to most types of Object-Oriented projects. It also claims that it is useable in the most early stages, as it requires minimal input to produce a result. We cannot judge the accuracy of this result, which is expected to be off the final figures anyway, as mentioned elsewhere in this report.

At a glance:*Pros:*

- Live commercial effort.
- Modern, OO design, support for iterative development models.
- Applicable very early in the project lifecycle.

Cons:

- Little known about its underpinnings.
- Not as widespread as other methods.
- Only commercial implementation is available.

4 The Estimation Process

In section 3 we talked about the nitty-gritty of estimation, the methods that are available to estimate the project. There is commercial and free software available which implements these methods. However, we believe that the most important issue about estimation is frame of mind. Most of this section is inspired by [1] and also personal experience.

4.1 The silver bullet syndrome

It happens to many people to get enthused about these methods and tools. On their own they are not sufficient. Nor are they applicable straight from the box. One has to experiment with them, probably use more than one at a time so as to evaluate and calibrate them. No two software development environments are the same.

4.2 Development model

Choose a development model that helps you deal with the software development tradeoffs (see figure 2 and the inherent difficulties of estimation. Many of the reasons for project over-runs are directly related to inappropriate choice of model [2]

This is further backed up by McConnell [1] who states that incorrect model choice can result in missing tasks and inappropriate task ordering which undercuts project planning and efficiency.

For example, if you have a project with fixed effort and a fixed cost, then the development model you use must allow you to trim the software functionality as the project proceeds. As illustrated by the cone of uncertainty 1, your initial estimates of what can be achieved within the fixed effort and costs will invariably be out. To ensure the project remains within its budget you must therefore be able to cut functionality. A timebox development approach [1] could be useful in such circumstances since its basic premise is to redefine the product to fit the schedule rather than redefining the schedule to fit the project.

For a speculative development in an area with many unknowns, estimation can be even more difficult. In such circumstances, a spiral development model [1], is much more appropriate. Here, the project starts on a small scale, explores the project risks, makes a plan to handle the risks, and then decides whether to proceed. This identification and reassessment of risks combined with decision-making on whether or not to proceed continues in an iterative manner throughout the lifetime of the project at regular intervals. This approach therefore recognises that early estimates will be wrong and thus allows the project to be cancelled before risk factors can engulf the schedule.

4.3 Keep track of your project

No matter which method of estimation one uses and no matter how accurate you/it has been in the past, do not overestimate it. A software project is a live entity with complex behaviour. The estimates must be closely monitored and updated, as highlighted in [17], even after the detailed design stage you can still expect to be off by 25%. Complement your estimates with a good tracking policy and always update the risks list and the priorities list. There are straightforward, affordable or even free pieces of software to help you perform these most important tasks.

4.4 Estimate in ranges

The tools and methods may give you absolute numbers about the end date of your project; don't be fooled. Revisit each of the workpackages that you split your project into and try to predict what can go

wrong. Put these in your risks list and try to evaluate their impact. While risk management deserves books, we can only afford a paragraph in this report.

Risks can hit you in two ways. One is for them to be very probable. The other is for them to cause extensive damage. And of course these two can come together. When evaluating such risks try to predict both their impact and their probability. If you multiply these (perhaps coming from gut-feeling) numbers you can get an indication of how severe this risk is.

When drafting your estimates you can start with raw numbers, eg "the GUI will take 3 months", and these can be inspired by tools. Associate these with a logical range associated with plausible risks or substantiated claims for time gains. The above example could look like "the GUI will take 3 months, +1 if the tool-generated code is useless, +0.5 if Jo goes on holiday, -1 if we can reuse the "File" menu functionality from project X". Please note that this is not the same as $3 + 1 + 0.5 - 1 = 3.5$. For a fuller explanation of this see McConnell [1]

4.5 What's the day today?

Be very careful not to forget basic laws of life. People work n days a week and have m days per annum holidays. Fill in the numbers for your environment and add these to your estimate. Do not expect these to spread evenly across the year. For example, few people work in Christmas time in European countries and especially in academic environments. How much work can you *really* do on a two month project starting on the first of December? How long will component X take if Jo goes on holiday in the summer?

4.6 Make sure

Providing an estimate is very complex, as discussed throughout this report. It is a shame that it is not taken seriously though. Many times you will be asked to provide an estimate thinking on your feet. Don't be tempted. Once you have given an estimate, no matter how much you stress "it is a rough one" or I am not sure, coming back a few days or weeks later with a number that is, say, four times as high will not go down very well. Although it is almost normal for the estimate at the requirements capture stage to be off by 400%.

The bottomline is, don't give an off-the-cuff estimate; anything you say can and will be used against you. Humphrey [6] gives a interesting commentary on ownership of estimates and their manipulation.

4.7 Bring in the right people

It is up to the project manager to consider the design of a piece of software themselves. When it comes to estimates, though, there are 2 groups of colleagues one cannot ignore.

The first group is those experienced in estimation. Their input is twofold. One is that they know how to appreciate the workload of components and are indispensable when combined with documented estimates from past projects. The other is that they know how to perform the task. Their advice is particularly important in environments with loose processes. This has been discussed extensively in section 3.2.

If you think you can or have to do without estimation veterans, you cannot do without the developers themselves. It is a common mistake to judge the effort from your own standards if it is not you who is going to do the job. Not everybody is the same and people have strong skills and strong weaknesses. These can affect the estimates by lengthening workpackages and shortening others. Their focus on the technical detail is also useful input as it can highlight problems in the implementation. Finally, do not discount the fact that by involving them you cause a feeling of ownership which will yield persistence and application to the project.

5 Estimation Tools

This contains examples of estimation tools from commercial vendors.

5.1 KnowledgePLAN

This tool from Software Productivity Research is a knowledge-based estimation, risk analysis and scheduling tool. It can be used to determine effort, schedule, cost, resource requirements and level of quality. It contains a wizard which takes you through the various stages of the estimation process.

It contains interesting Scope Creep functionality whereby a user supplies quality requirements and position in workplan and the tool estimates the impact of proposed changes to the effort, cost and quality of the project.

The knowledge base it uses is updated annually for various types of projects. More information can be found at <http://www.spr.com/Products/KnowledgePLAN/knowledgeplan.htm>. (**** Need to check this *****)

5.2 Costar

This COCOMO II based product is from Softstar Systems. It estimates project duration, staffing levels, effort and cost.

<http://www.softstarsystems.com>.

6 Further sources of information

6.1 General Books

For a general introduction to the area of software estimation, Steve McConnell's book *Rapid Development* [1] contains a number of excellent chapters on estimation and related areas. It also includes a list of books on general estimation for further reading. Similarly Ian Sommerville's book *Software Engineering* [4] contains a chapter on estimation with recommendations for further reading.

Yourdon's book *Deathmarch* [27] whilst not being specifically concerned with estimation does give valuable advice on how to survive a project if it looks like it will over-run.

6.2 WWW Sites

www.dacs.dtic.mil The Data & Analysis Center for Software (DACS) is USA Department of Defense Information Analysis Center. It acts as a software information clearing house and acts as source software information. This site contains a fairly extensive list of cost estimation resources and references. It includes a list of software estimating tool vendors.

References

- [1] Steve McConnell. *Rapid development: taming wild software schedules*. Microsoft Press, 1996.
- [2] B W Boehm, C Abts, A W Brown, S Chulani, B K Clark, E Horowitz, R Madachy, D Reifer, and B Steece. *Software Cost Estimation with COCOMO II*. Prentice Hall PTR, 2000.
- [3] Construx Software Inc. <http://www.construx.com/estimate>.
- [4] I Sommerville. *Software Engineering, Sixth Edition*. Addison-Wesley Publishers Limited, 2001.
- [5] Steve McConnell. *Software Project Survival Guide*. Microsoft Press, 1998.
- [6] W.S.Humphrey. Your Date or Mine. In *The Watts New Collection*. Software Engineering Institute, Carnegie Mellon University, <http://interactive.sei.cmu.edu/>, 2001.
- [7] B Boehm, C Abts, and S Chulani. Software Development Cost Estimation Approaches - A Survey. Technical Report USC-CSE-2000-505, University of Southern California - Center for Software Engineering, USA, 2000.
- [8] E Nelson. Management Handbook for the Estimation of Computer Programming Costs. Technical report, Systems Development Corporation, USA, Oct 1966.
- [9] L Putnam and W Myers. *Measures for Excellence*. Yourdon Press Computing Series, 1992.
- [10] C Jones. *Applied Software Measurement*. McGraw Hill, 1997.
- [11] R Park. The Central Equations of the PRICE Software Cost Model. In *4th COCOMO Users' Group Meeting, November 1988*, 1988.
- [12] R Jensen. An improved Macrolevel Software Development Resource Estimation Model. In *Proceedings 5th ISPA Conference, April, 1983*, pages 88–92, 1983.
- [13] B Boehm. *Software Engineering Economics*. Prentice Hall, 1981.
- [14] L Briand, V Basili, and W Thomas. A Pattern Recognition Approach for Software Engineering Data Analysis. *IEEE Transactions on Software Engineering*, 18(11), 1992.
- [15] T Khoshgftaar, A Pandya, and D Lanning. Application of Neural Networks for predicting program faults. *Annals of Software Engineering*, 1, 1995.
- [16] H Zuse. *History of Software Measurement*, Chapter 5 "Cost Estimation" at http://irb.cs.tu-berlin.de/~zuse/metrics/History_05.html.
- [17] B Boehm, B Clark, E Horowitz, C Westland, R Madachy, and R Selby. *Cost Models for Future Software Life Cycle Processes: COCOMO2.0* at <http://sunset.usc.edu/publications/TECHRPTS/1995/usccse95-508/usccse95-508.pdf>.
- [18] International Software Benchmarking Standards Group (ISBSG). *Software estimation, benchmarking, productivity, risk analysis, and cost information for software developers and business* at <http://www.isbsg.org.au>.
- [19] O Helmer. *Social Technology*. Basic Books, NY, USA, 1966.
- [20] B Baird. *Managerial Decisions Under Uncertainty*. John Wiley & Sons, 1989.
- [21] R Boehm. *Function Point FAQ* at <http://ourworld.compuserve.com/homepages/softcomp/fpfaq.htm>.
- [22] Edmond VanDoren. *Software Technology Review: Function Point Analysis* at http://www.sei.cmu.edu/activities/str/descriptions/fpa_body.html.
- [23] The Common Software Measurement Metric International Consortium. COSMIC-FFP Measurement Manual. Technical report, The Common Software Measurement Metric International Consor-

tium, available from

<http://www.lrgl.uqam.ca/publications/private/446.pdf>, 2001.

- [24] A Abran, C Symons, and S Cligny. An overview of COSMIC-FFP field trial results, also available from <http://www.lrgl.uqam.ca/publications/pdf/614.pdf>. In *ESCOM 2001, London, England, April 2-4*, 2001.
- [25] R J Kauffman R D Banker and R Kumar. An empirical test of object-based output measurement metrics in a computer aided software engineering (CASE) environment. *Management Information Systems*, 8:127–150, 1991.
- [26] The Object Factory Ltd. *Estimating Software Projects using ObjectMetrix* at <http://www.theobjectfactory.com/downloads/papers/ObjectMetrix.pdf>.
- [27] E Yourdon. *Death march: managing "mission impossible" projects*. Prentice Hall PTR, 1997.