Introduction

An engineering discipline needs to be able to estimate the cost of a construction project before construction begins. Software cost estimation seeks to develop, enhance, and apply analytical modeling techniques to the estimation of time and effort required for the completion of software development projects. The need for estimation is driven by the cost considerations of industrial-strength development projects, in the public as well as the private sector. There are three sources of input for predictive modeling: program or system decomposition, experience and historical data, and empirical models. Two types of estimation models are examined: cost models and constraint models. Early models focused on estimating cost and effort alone. More recent work has turned to parameterizing the tradeoffs inherent in design decisions that affect not only cost and resource allocation issues, but many software quality attributes. Several publicly available tools are also examined which implement one or more cost models, constraint models, or a combination thereof.

There have been many efforts at developing software cost estimation techniques, with a variety of bases: model-based (e.g. SLIM, COCOMO, Price-S), expertise-based (e.g. Delphi), learning-oriented (e.g. neural or case-based models), dynamics-based (e.g. Abdel-Hamid-Madnick), regression-based (e.g. OLS, Robust), and composite techniques (e.g. Bayesian-COCOMO II). This study will confine itself to model-based techniques.

An ideal indicator of success in this field would be full-adoption/full-win, i.e. all the software projects that were correctly estimated and carried out using rigorous cost estimation methods would come in on time and on budget (and, for some systems, on target with regard to other quality metrics).

History of the Technology: Early Stages

Attempts at developing software cost estimation methods began in the early 1960’s. The earliest example from this period is Delphi, which arose from attempts at Rand Corporation to develop systematic methods for forecasting complex processes, using expert consensus. The seminal paper "Report on a Long-Range Forecasting Study," by T. J. Gordon and Olaf Helmer, was published as a Rand paper in 1964. Delphi was the first use of collected cost-related data, derived from actual computer systems developed as far back as World War II. Another early system was Nelson’s SDC (1966).

Nearly 10 years passed between these early efforts and the development of several software cost estimation methods that would form the basis of a true emerging technology. Beginning in the mid-1970’s, several competing methods appeared, each based on one of two competing measures of software project size: Source Lines of Code...
(SLOC) and Function Points (FP). It is worthwhile to examine these measures in more detail before continuing with the narrative of technological maturation.

SLOC is a fairly intuitively obvious measure, one which was already in use for measuring the size of finished projects; during the 1960’s, SLOC was counted in terms of 80-line punch cards. Cost estimation techniques which rely on this measure assume that the cost of a software system can be predicted by estimating the number of functional lines of source code the final system will contain. Developments in SLOC estimation methods over time have included weighted considerations for partial code reuse. Criticisms of this type of estimation method have arisen from a number of sources, including differences between the number of lines of code required to write routines in different programming languages.

Function points are a less obvious metric, developed by Allan Albrecht in the late 1970’s, as an attempt to measure systems from a functional perspective, rather than through counting lines of code. Since function points are language and platform independent, they will theoretically maintain a consistent count for a given system design, regardless of the mechanisms of that design’s implementation. More will be said about function points in the descriptions of the individual models which use this metric.

Between 1974 and 1981, several important software cost estimation models emerged for general use, mainly driven by the needs of industrial software development. One of the first of these was the Price-S model, developed by Freiman and Park originally for Martin Marietta Price Systems internal use, incl. Apollo program (soon RCA Price, then GE Price), and released as a proprietary model in 1977. This was the first commercially available detailed parametric software cost model to be extensively marketed and used. The required inputs were SLOC, and additional productivity and complexity factors; the output was effort in person-hours or person-months. Today, Price-S is commercially available from Price Systems, and has been updated to include three submodels. The Acquisition submodel forecasts costs and schedules. The Sizing submodel estimates software size in SLOC, function points, or Predictive Object Points (POPs, introduced in 1998 for sizing object-oriented development projects). The Life-cycle Cost submodel is intended only for early costing of the maintenance and support phase of the software development life cycle, and is used in conjunction with the Acquisition submodel. Other important SLOC-based models were developed contemporaneously by Wolverton at TRW, and by Walston and Felix at IBM.

In 1979, Albrecht published his description of the Function Point counting method of software estimation. This is an appropriate point at which to describe Function Points in some detail. Function points measure systems from a functional perspective, using five major classifications of functionality for estimating cost at any point in the life cycle from detailed design onward. An underlying assumption of the Function Point count method is that boundaries can be drawn around a computer system to be measured prior to classifying components. External Inputs are processes in which data crosses a boundary from outside to inside, while External Outputs are processes in which data crosses a boundary from inside to outside. External Inquiry refers to a process that results in data
retrieval from both internal logical or external interface files, and involves both input and output processes. Internal Logical Files remain entirely within an application boundary. External Interface Files contain data which resides entirely outside an application boundary, e.g. within another application. Components are classified and then ranked high, low, or average to give them a weighting factor based on the number of data element types and record element types they will require.

A second function point related method appeared soon after: Software Life-Cycle Model (SLIM), developed by Larry Putnam of Qualitative Software Measurement. SLIM modeled the development lifecycle as a Rayleigh distribution of project personnel vs. time, using function points or SLOC as its primary inputs. However, additional factors were required for estimation: a Manpower Buildup Index (MBI) and a technology constant or productivity factor (PF). Calibration to individual projects was achieved by deriving the MBI and PF from previous project data or, if previous data unavailable, a set of questions answered by the project development team. SLIM’s outputs were required project effort in man-hours, schedule, and defect rate.

The proliferation of thought and effort in software cost estimation reached critical mass in 1981, which saw the publication of no less than three important methods of cost estimation, as well as a seminal text in the field, an invaluable aid to concept dissemination. In 1981, SOFTCOST appeared from JPL, Bailey and Basili’s method appeared from work done at NASA, and Barry Boehm published the full-length textbook, Software Engineering Economics. Although Boehm has published a paper describing his SLOC-based COnstructive COst MOdel (COCOMO) in 1978, the book set the method in a context of real-world software engineering projects and economic decision analysis, and described application of COCOMO in a way that made it appear as reasonable a part of the software development process as cost-benefit analysis. It did not look simple, but it made sense in context, integrating the language of software engineering with the language of microeconomics. Even practitioners who did not read the formulae, began to use the vocabulary.

Original COCOMO was derived from, and calibrated using, an extensive dataset of completed projects at TRW. The model had three modes (e.g. aircraft postflight data reduction): an organic mode, for projects with low complexity, small teams, and informal requirements; a semi-detached mode, for projects with intermediate complexity, mixed teams, and rigid or less-than-rigid requirements (e.g. aircraft flight-training simulator); and an embedded mode, for projects with highly rigid requirements (e.g. aircraft on-board collision avoidance system). COCOMO also had three models. The Basic model simply calculated required effort as a function of program size, counted in SLOC. The Intermediate model estimated required effort as a function of program size and a set of cost drivers. The Advanced model estimated required effort as a function of program size, cost drivers, and also phase of the software development life cycle, with a separate estimate determined for each phase of the life cycle. The Basic COCOMO equation is: \( E = aKLOC^b \). The selection of mode provides weighting factors \( a \) and \( b \) for this equation; another weighting factor for the Intermediate and Advanced models is the product of 15 cost drivers (multipliers) in four categories: product, computer, personnel and project.
The life cycle phases addressed by the Advanced model are: requirements planning and product design (RPD), detailed design (DD), code and unit test (CUT), and integration and test (IT). The Advanced model breaks also weights the 15 cost drivers separately for each life cycle phase.

**History of the Technology: Maturation**

Late in the 1980’s, two key signs of technology maturation appeared: the development of several new models that built on and extended the concepts of the 1974-1981 models, and attempts at experimental validation of parts of the 1974-1981 models by external parties.

**Experimental Validation**

Empirical studies must formulate an hypothesis or question to test, observe a situation, abstract the observations into data, analyze the data, and draw conclusions with respect to the tested hypothesis. Some types of empirical studies are controlled experiments, observational studies, case studies, surveys, and prototyping studies. Controlled experiments are difficult to perform with software cost estimation models of complex real-world projects, because it is difficult to enforce or even measure compliance with all planned parameters over the long time it takes to complete such a project, given all the human elements involved, both internal and external. However, extensive work has been done to assess the internal consistency, and comparative validity, of the underlying metrics on which the models are based: SLOC and function point counts.

SLOC counts have to some extent been automated, at least once the detailed design phase of software development has been reached. Since function point counts have not been automated, and are less amenable to automation that SLOC counts, people must create them, and they have been attacked as subject to inconsistency since their introduction. An important study, published in 1987 (Kemerer 1987), performed an empirical comparison of four different methods: SLOC vs. function point cost estimation methods, and proprietary vs. non-proprietary methods. For this study, 16 projects were obtained from a data-processing applications company, and the models were applied post hoc. Parameters for applying models to completed projects were derived through interviews and surveys. No significant difference was found between proprietary and nonproprietary models, and the function points model was validated by the data. All the models, however, were deemed to require calibration to be generalizable.

Another important empirical study by the same external evaluator appeared six years later (Kemerer 1993), seeking to answer the question of interrater reliability of function points counts (the extent to which independent counts of function points agree). By this time, a standard for function point counting had appeared (IFPUG 3.0), which used the Albrecht Standard method. The study measured interrater reliability of the Albrecht Standard method, interrater reliability of a newer, alternative function point counting method, and intermethod reliability of these two methods. The other method selected for comparison was the Entity-Relationship counting method, which had potential for FP count automation. The interrater reliability of both methods was found to be high, and
intermethod reliability also strongly correlated, which was a strong validation of the ability of people to understand and apply the function point counting process, and thereby the underlying metric of the function point estimation models.

Empirical studies of this kind are representative of empirical studies of programmers, a subset of empirical studies of software engineering. Empirical studies of programmers involve human subjects research methods, and mainly addresses areas of software development in which variations in ability, skills, and/or experience of personnel significantly affects the outcome of the development process in terms of cost, schedule, or quality. Human subjects research standards are regulated by the National Institute of Health (NIH), and participants in software engineering experiments are subject to the same protections as in medical or psychological experiments. Human subjects add complexity to experimentation. Among other issues, one must select a sample that will generalize to the population you are trying to understand, and carefully control for confounds by using a sufficient sample size, randomizing and blinding to treatment, and counterbalancing conditions. Extraneous variables such as age, domain-specific knowledge, education, experience, gender may also cause confounds. Despite these difficulties, however, developers, engineers, and architects are integral and non-interchangeable parts of software development, and research on their internal process and its effect on software development is no less important in predicting and refining the development process than the study of how any other part of the software functions.

**Expanded and Enhanced Models**

Beginning in the late 1980’s, expansions and enhancements of the early models appeared to address additional and alternative outputs of estimation. An early example of this was SEER-SEM. Developed at Galoreth and based on Jensen’s 1983 model, SEER-SEM was a proprietary parametric estimation model using SLOC and/or function points as inputs, with additional cost drives for size, personnel, complexity, environment, constraints, platform & application, development method, and standards. This model also allowed user calibration through a customizable knowledge base. Outputs, no longer the simple effort and schedule, included capability metrics, trade-off analyses with comparison of alternatives, and was later enhanced to add risk analysis. JPL SCT, which appeared a few years later, also focused on cost and risk estimation.

The COCOMO model was updated in 1995 with COCOMO II, based on a database of 161 completed projects. In this version, the Basic and Intermediate models were replaced and many cost drivers either replaced or updated. Three additional development submodels were added. The Applications Composition submodel, for rapid development, took a new kind of input: object points (a count of screens, reports, and inputs), and was weighted by 3-level complexity factor. The Early Design model, for exploration of alternative architectures, used function points or SLOC input, along with five scale factors and seven effort multipliers, and was calibrated using the effort multipliers from Post-Architecture model. The Post-Architecture model, for when top level design is complete, used SLOC and/or function point input, as well as five scale factors and 17 effort multipliers; this model was not yet completely calibrated as of 2002.
In the apparently well-calibrated category, ESTIMACS, a late-1990’s entry from Software Productivity Research, and based on work by Capers Jones, was calibrated using a database of 8000 software development projects. The inputs used were function points, or another metric, feature points, a variation on the function point concept, along with software environment factors. Outputs are four levels of required effort: project, phase, activity, and task; additional outputs are resources, deliverables, costs, schedules.

Models have also begun to appear to address evolving paradigms of software engineering. SELECT Estimator, released in 1998 by SELECT Software Tools, and based on ObjectMetrix, a proprietary model, is specifically designed for large scale, object oriented, distributed systems, and assumes an incremental development lifecycle. The primary input is scope of project, including applications, classes, user cases, packages, components, services, and additional factors include complexity, reuse, and technology. The outputs are simple: total effort in person-days, schedule in months, and total development cost. WEBMO is the newest of the models; appearing in 2002, it seeks to address web development projects specifically. And the COCOMO family of models has expanded to include several models targeted at tradeoff analysis between cost and additional quality attributes such as reliability.

**Post-Research Maturity**

It is difficult to tell exactly what the adoption rates of cost estimation models are in software development, outside of government development, where formal estimation techniques are required as part of budgeting and accountability. But models and tools are still struggling for acceptance in commercial software development practice. Some say that this is due to social politics, some to the difficulty of fitting the models to complex realities. Whatever the reason, given that the majority of software projects still come in late, over budget, and often at a lower level of quality than desired (let alone planned), one must conclude that either the models themselves are seriously flawed, or that they are implemented ineffectively, incompletely, and/or inconsistently. A belief that software cost estimation models work is inconsistent with a belief that it has reached Redwine and Riddle’s furthest stage of maturation, i.e. 70% adoption.

However, the concepts and language of software cost estimation have permeated the software engineering and development culture, to the point where what may once have been revelations are now consultant-speak. This is an indication of a mature technology, as is the development of related technologies. According to Barry Boehm [Boehm 2003], software engineering has changed. Software is a much bigger part of systems than it was 30 years ago, strongly affecting the cost, schedule, and value of systems. A high proportion of software failures is caused by value-oriented issues: lack of user input, incomplete or changing requirements, lack of resources, unclear objectives, and unrealistic expectations and time frames. Value considerations, he says, must be integrated into existing and emerging SE principles and practices, creating value-based software engineering, in which factors considered equipotential in cost estimation models...
(including his own) are now deliberately weighted for value by the development teams working together with the stakeholders.

The framework proposed for this recognition of a de facto paradigm shift is interesting in two respects. First, software cost estimation is but one (rather small) part of the agenda, which contains as much attention to human factors as to technical factors, if not more, and incorporates software cost estimation in a larger context of software development as it is. Second, it is not interesting in that any of the components is particularly novel, but in that the different components draw heavily on knowledge derived from disciplines outside computer science, including economics and psychology, and that vocabulary transference has been successfully made over the last 25 years from these disciplines to the professional software engineering community, largely through the dissemination of software cost estimation concepts and their corollaries. Perhaps the original goal has not been reached because a purely technological solution is inadequate; in this case, broadening the scope of the problem to include the human side of the process may be the only way to reach that original goal.

Bibliography

Software Cost Estimation


Boehm, B.W. Software Engineering. IEEE Transactions on Computers, C25(12), December 1976, pp.1226-1241. Defines “software engineering” and surveys the state of the field, particularly as it relates to management approaches and general applicability of techniques.


This paper suggests a roadmap for the future, placing cost estimation in a larger context of software engineering economics.


Empirical study of application of Halstead’s program length metric, McCabe’s control complexity metric, and program bandwidth measure to a real-time software system.

Description of software cost estimation techniques underlying SLIM estimation model.

Personal narrative of development of SLIM cost estimation method.

Evaluation of three techniques for estimating software cost when information is missing from the model, with results favoring hot-deck imputation.

Introduction of Mark II approach to function point based software cost estimation.

Description of software cost estimation method developed at IBM.

Informative site on the history of computer science from computer scientist at Technische Universität Berlin, son of computing pioneer Konrad Zuse.

**Empirical Studies of Programmers**

Controlled experiment comparing individual and group performance in creating scenario profiles for architectural design modifications.

Controlled experiment comparing performance of 20 computer science undergraduate students modifying “good” and “bad” object-oriented and structured program designs.

Controlled experiment comparing comprehension and performance between procedural and object-oriented programmers during a code modification task.


Comparison of empirical literature in software cost estimation and software inspections research with content of Journal of Empirical Software Engineering.


Survey of empirical studies about testing techniques, categorized by technological maturity of knowledge involved in each study.


Contains brief introduction to history of empirical studies of programmers in the context of human-computer interaction.


Controlled experiment comparing performance of 59 data processing professionals using seven code walkthrough/inspection methods on a small sample program.


Survey of the current state of empirical studies in SE. Call for better study design and data collection techniques, and increased involvement with other areas of software engineering research.


Controlled experiment comparing performance of 24 computer science graduate students using ad hoc, checklist, and scenario methods in a limited specification requirements inspection task.

Controlled experiment comparing performance of 48 computer science graduate students using ad hoc, checklist, and scenario methods in a limited specification requirements inspection task.

