Software Cost Estimation

Hareton Leung
Zhang Fan
Department of Computing
The Hong Kong Polytechnic University
{cshleung, csfzhang}@comp.polyu.edu.hk

Abstract

Software cost estimation is the process of predicting the effort required to develop a software system. Many estimation models have been proposed over the last 30 years. This paper provides a general overview of software cost estimation methods including the recent advances in the field. As a number of these models rely on a software size estimate as input, we first provide an overview of common size metrics. We then highlight the cost estimation models that have been proposed and used successfully. Models may be classified into 2 major categories: algorithmic and non-algorithmic. Each has its own strengths and weaknesses. A key factor in selecting a cost estimation model is the accuracy of its estimates. Unfortunately, despite the large body of experience with estimation models, the accuracy of these models is not satisfactory. The paper includes comment on the performance of the estimation models and description of several newer approaches to cost estimation.

Keywords: project estimation, effort estimation, cost models.

1. Introduction

In recent years, software has become the most expensive component of computer system projects. The bulk of the cost of software development is due to the human effort, and most cost estimation methods focus on this aspect and give estimates in terms of person-months.

Accurate software cost estimates are critical to both developers and customers. They can be used for generating request for proposals, contract negotiations, scheduling, monitoring and control. Underestimating the costs may result in management approving proposed systems that then exceed their budgets, with underdeveloped functions and poor quality, and failure to complete on time. Overestimating may result in too many resources committed to the project, or, during contract bidding, result in not winning the contract, which can lead to loss of jobs.

Accurate cost estimation is important because:

- It can help to classify and prioritize development projects with respect to an overall business plan.
- It can be used to determine what resources to commit to the project and how well these resources will be used.
- It can be used to assess the impact of changes and support replanning.
- Projects can be easier to manage and control when resources are better matched to real needs.
- Customers expect actual development costs to be in line with estimated costs.

Software cost estimation involves the determination of one or more of the following estimates:

- effort (usually in person-months)
- project duration (in calendar time)
- cost (in dollars)

Most cost estimation models attempt to generate an effort estimate, which can then be converted into the project duration and cost. Although effort and cost are closely related, they are not necessarily related by a simple transformation function. Effort is often measured in person-months of the programmers, analysts and project managers. This effort estimate can be converted into a dollar cost figure by calculating an average salary per unit time of the staff involved, and then multiplying this by the estimated effort required.

Practitioners have struggled with three fundamental issues:

- Which software cost estimation model to use?
- Which software size measurement to use – lines of code (LOC), function points (FP), or feature point?
- What is a good estimate?

The widely practiced cost estimation method is expert judgment. For many years, project managers have relied on experience and the prevailing industry norms as a basis to develop cost estimate. However, basing estimates on expert judgment is problematic:

- This approach is not repeatable and the means of deriving an estimate are not explicit.
- It is difficult to find highly experienced estimators for every new project.
- The relationship between cost and system size is not linear. Cost tends to increase exponentially with size. The expert judgment method is appropriate only when the sizes of the current project and past projects are similar.
- Budget manipulations by management aimed at avoiding overrun make experience and data from previous projects questionable.

In the last three decades, many quantitative software cost estimation models have been developed. They range from empirical models such as Boehm's COCOMO models [5] to *analytical* models such as those in [30, 29, 8]. An *empirical model* uses data from previous projects to evaluate the current project and derives the basic formulae from analysis of the particular database available. An *analytical model*, on the other hand, uses formulae based on global assumptions, such as the rate at which developer solve problems and the number of problems available.

Most cost models are based on the size measure, such as LOC and FP, obtained from size estimation. The accuracy of size estimation directly impacts the accuracy of cost estimation. Although common size measurements have their own drawbacks, an organization can make good use of any one, as long as a consistent counting method is used.

A good software cost estimate should have the following attributes [31]:

- It is conceived and supported by the project manager and the development team.
- It is accepted by all stakeholders as realizable.
- It is based on a well-defined software cost model with a credible basis.
- It is based on a database of relevant project experience (similar processes, similar technologies, similar environments, similar people and similar requirements).
- It is defined in enough detail so that its key risk areas are understood and the probability of success is objectively assessed.

Software cost estimation historically has been a major difficulty in software development. Several reasons for the difficulty have been identified:

- Lack of a historical database of cost measurement
- Software development involving many interrelated factors, which affect development effort and productivity, and whose relationships are not well understood
- Lack of trained estimators and estimators with the necessary expertise
- Little penalty is often associated with a poor estimate

2. Process of estimation

Cost estimation is an important part of the planning process. For example, in the top-down planning approach, the cost estimate is used to derive the project plan:

1. The project manager develops a characterization of the overall functionality, size, process, environment, people, and quality required for the project.
2. A macro-level estimate of the total effort and schedule is developed using a software cost estimation model.
3. The project manager partitions the effort estimate into a top-level work breakdown structure. He also partitions the schedule into major milestone dates and determines a staffing profile, which together forms a project plan.

The actual cost estimation process involves seven steps [5]:

1. Establish cost-estimating objectives
2. Generate a project plan for required data and resources
3. Pin down software requirements
4. Work out as much detail about the software system as feasible
5. Use several independent cost estimation techniques to capitalize on their combined strengths
6. Compare different estimates and iterate the estimation process
7. After the project has started, monitor its actual cost and progress, and feedback results to project management

No matter which estimation model is selected, users must pay attention to the following to get best results:

- coverage of the estimate (some models generate effort for the full life-cycle, while others do not include effort for the requirement stage)
- calibration and assumptions of the model
- sensitivity of the estimates to the different model parameters
- deviation of the estimate with respect to the actual cost

3. Software sizing

The software size is the most important factor that affects the software cost. This section describes five software size metrics used in practice. The line of code and function point are the most popular metrics among the five metrics.

Line of Code: This is the number of lines of the *delivered source code* of the software, excluding comments and blank lines and is commonly known as *LOC* [10]. Although LOC is programming language dependent, it is the most widely used software size metric. Most models relate this measurement to the software cost. However, exact LOC can only be obtained after the project has completed. Estimating the code size of a program before it is actually built is almost as hard as estimating the cost of the program.

A typical method for estimating the code size is to use experts' judgement together with a technique called *PERT* [3]. It involves experts' judgment of three possible code-sizes: S_l, the lowest possible size; S_h the highest possible size; and S_m, the most likely size. The estimate of the code-size S is computed as:

$$S = \frac{S_l + S_h + 4S_m}{6}$$

PERT can also be used for individual components to obtain an estimate of the software system by summing up the estimates of all the components.

Software science: Halstead proposed the *code length* and *volume* metrics [13]. Code length is used to measure the source code program length and is defined as:

N= N1+N2

where N1 is the total number of operator occurrences, and N2 is the total number of operand occurrences.

Volume corresponds to the amount of required storage space and is defined as:

V = N log(n1+n2)

where n1 is the number of distinct operators, and n2 is the number of distinct operands that appear in a program.

There have been some disagreements over the underlying theory that supports the software science approach [14, 32]. This measurement has received decreasing support in recent years.

Function points: This is a measurement based on the functionality of the program and was first introduced by Albrecht [1]. The total number of function points depends on the counts of distinct (in terms of format or processing logic) types in the following five classes:

1. *User-input types*: data or control user-input types
2. *User-output types*: output data types to the user that leaves the system
3. *Inquiry types*: interactive inputs requiring a response
4. *Internal file types*: files (logical groups of information) that are used and shared inside the system
5. *External file types*: files that are passed or shared between the system and other systems

Each of these types is individually assigned one of three *complexity levels* of {1 = simple, 2 = medium, 3 = complex} and given a weighting value that varies from 3 (for simple input) to 15 (for complex internal files).

The *unadjusted function-point counts (UFC)* is given as

$$\text{UFC}=\sum_{i=1}^{5}\sum_{j=1}^{3} N_{ij}W_{ij}$$

where N_{ij} and W_{ij} are respectively the number and weight of types of class i with complexity j.

For example, if the raw function-point counts of a project are 2 simple inputs (W_{ij} = 3), 2 complex outputs (W_{ij} = 7) and 1 complex internal file (W_{ij} = 15). Then UFC = 2*3 + 2*7 +1*15 = 35.

This initial function-point count is either directly used for cost estimation or is further modified by factors whose values depend on the overall complexity of the project. This will take into account the degree of distributed processing, the amount of reuse, the performance requirement, etc. The final function-point count is the product of the UFC and these project *complexity factors*. The advantage of the function-point measurement is that it can be obtained based on the system requirement specification in the early stage of software development.

The UFC is also used for code-size estimation using the following linear formula:

LOC = a * UFC + b

The parameters a, b can be obtained using linear regression and previously completed project data. The latest *Function Point Counting Practices Manual* is maintained by the IFPUG (International Function Point Users Group) in http://www.ifpug.org/.

Extensions of function point: *Feature point* extends the function points to include algorithms as a new class [21]. An *algorithm* is defined as the set of rules which must be completely expressed to solve a significant computational problem. For example, a square root routine can be considered as an algorithm. Each algorithm used is given a weight ranging from 1 (*elementary*) to 10 (*sophisticated algorithms*) and the feature point is the weighted sum of the algorithms plus the function points. This measurement is especially useful for systems with few input/output and high algorithmic complexity, such as mathematical software, discrete simulations, and military applications.

Another extension of function points is ***full function point*** **(FFP)** for measuring real-time applications, by also taking into consideration the control aspect of such applications. FFP introduces two new control data function types and four new control transactional function types. A detailed description of this new measurement and counting procedure can be found in [35].

Object points: While feature point and FFP extend the function point, the object point measures the size from a different dimension. This measurement is based on the number and complexity of the following objects: screens, reports and 3GL components. Each of these objects is counted and given a weight ranging from 1 (*simple screen*) to 10 (*3GL component*) and the object point is the weighted sum of all these objects. This is a relatively new measurement and it has not been very popular. But because it is easy to use at the early phase of the development cycle and also measures software size reasonably well, this measurement has been used in major estimation models such as COCOMO II for cost estimation [6].

4. Cost estimation

There are two major types of cost estimation methods: *algorithmic* and *non-algorithmic*. Algorithmic models vary widely in mathematical sophistication. Some are based on simple arithmetic formulas using such summary statistics as means and standard deviations [9]. Others are based on regression models [38] and differential equations [30]. To improve the accuracy of algorithmic models, there is a need to adjust or calibrate the model to local circumstances. These models cannot be used *off-the-shelf*. Even with calibration the accuracy can be quite mixed.

We first give an overview of non-algorithmic methods.

4.1 Non-algorithmic Methods

Analogy costing: This method requires one or more completed projects that are similar to the new project and derives the estimation through reasoning by analogy using the actual costs of previous projects. Estimation by analogy can be done either at the total project level or at subsystem level. The total project level has the advantage that all cost components of the system will be considered while the subsystem level has the advantage of providing a more detailed assessment of the similarities and differences between the new project and the completed projects. The strength of this method is that the estimate is based on actual project experience. However, it is not clear to what extend the previous project is actually representative of the constraints, environment and functions to be performed by the new system. Positive results and a definition of project similarity in term of features were reported in [33].

Expert judgment: This method involves consulting one or more experts. The experts provide estimates using their own methods and experience. Expert-consensus mechanisms such as Delphi technique or PERT will be used to resolve the inconsistencies in the estimates. The **Delphi technique** works as follows:

1) The coordinator presents each expert with a specification and a form to record estimates.
2) Each expert fills in the form individually (without discussing with others) and is allowed to ask the coordinator questions.
3) The coordinator prepares a summary of all estimates from the experts (including mean or median) on a form requesting another iteration of the experts' estimates and the rationale for the estimates.
4) Repeat steps 2)-3) as many rounds as appropriate.

A modification of the Delphi technique proposed by Boehm and Fahquhar [5] seems to be more effective: Before the estimation, a group meeting involving the coordinator and experts is arranged to discuss the estimation issues. In step 3), the experts do not need to give any rationale

for the estimates. Instead, after each round of estimation, the coordinator calls a meeting to have experts discussing those points where their estimates varied widely.

Parkinson: Using Parkinson's principle “work expands to fill the available volume” [28], the cost is determined (not estimated) by the available resources rather than based on an objective assessment. If the software has to be delivered in 12 months and 5 people are available, the effort is estimated to be 60 person-months. Although it sometimes gives good estimation, this method is not recommended as it may provide very unrealistic estimates. Also, this method does not promote good software engineering practice.

Price-to-win: The software cost is estimated to be the best price to win the project. The estimation is based on the customer's budget instead of the software functionality. For example, if a reasonable estimation for a project costs 100 person-months but the customer can only afford 60 person-months, it is common that the estimator is asked to modify the estimation to fit 60 person-months’ effort in order to win the project. This is again not a good practice since it is very likely to cause a bad delay of delivery or force the development team to work overtime.

Bottom-up: In this approach, each component of the software system is separately estimated and the results aggregated to produce an estimate for the overall system. The requirement for this approach is that an initial design must be in place that indicates how the system is decomposed into different components.

Top-down: This approach is the opposite of the bottom-up method. An overall cost estimate for the system is derived from global properties, using either algorithmic or non-algorithmic methods. The total cost can then be split up among the various components. This approach is more suitable for cost estimation at the early stage.

4.2 Algorithmic methods

The algorithmic methods are based on mathematical models that produce cost estimate as a function of a number of variables, which are considered to be the major cost factors. Any algorithmic model has the form:

$$\text{Effort} = f(x_1, x_2, \ldots, x_n)$$

where $\{x_1, x_2, \ldots, x_n\}$ denote the cost factors. The existing algorithmic methods differ in two aspects: the selection of cost factors, and the form of the function f. We will first discuss the cost factors used in these models, then characterize the models according to the form of the functions and whether the models are analytical or empirical.

4.2.1 Cost factors

Besides the software size, there are many other cost factors. The most comprehensive set of cost factors are proposed and used by Boehm et al in the COCOMO II model [6]. These cost factors can be divided into four types:

- **Product factors**: required reliability; product complexity; database size used; required reusability; documentation match to life-cycle needs;
- **Computer factors**: execution time constraint; main storage constraint; computer turnaround constraints; platform volatility;
- **Personnel factors**: analyst capability; application experience; programming capability; platform experience; language and tool experience; personnel continuity;
- **Project factors**: multisite development; use of software tool; required development schedule.

The above factors are not necessarily independent, and most of them are hard to quantify. In

many models, some of the factors appear in combined form and some are simply ignored. Also, some factors take discrete values, resulting in an estimation function with a piece-wise form.

4.2.2 Linear models

Linear models have the form:

$$\text{Effort} = a_0 + \sum_{i=1}^{n} a_i x_i$$

where the coefficients $a_{1, \ldots,} a_n$ are chosen to best fit the completed project data. The work of Nelson belongs to this type of models [26]. We agree with Boehm's comment that "there are too many nonlinear interactions in software development for a linear model to work well" [5].

4.2.3 Multiplicative models

Multiplicative models have the form:

$$\text{Effort} = a_0 \prod_{i=1}^{n} a_i^{x_i}$$

Again the coefficients $a_{1, \ldots,} a_n$ are chosen to best fit the completed project data. Walston-Felix [38] used this type of model with each x_i taking on only three possible values: -1, 0, +1. Doty model [16] also belongs to this class with each x_i taking on only two possible values: 0, +1. These two models seem to be too restrictive on the cost factor values.

4.2.4 Power function models

Power function models have the general form:

$$\text{Effort} = a \times S^b$$

where S is the code-size, and a, b are (usually simple) functions of other cost factors. This class contains two of the most popular algorithmic models in use, as follows:

COCOMO (Constructive Cost Model) models

This family of models was proposed by Boehm [4, 5]. The models have been widely accepted in practice. In the COCOMOs, the code-size S is given in thousand LOC (KLOC) and Effort is in person-month.

A) *Basic COCOMO*. This model uses three sets of $\{a, b\}$ depending on the complexity of the software only:
   (1) for simple, well-understood applications, $a = 2.4$, $b = 1.05$;
   (2) for more complex systems, $a = 3.0$, $b = 1.15$;
   (3) for embedded systems, $a = 3.6$, $b = 1.20$.
   The basic COCOMO model is simple and easy to use. As many cost factors are not considered, it can only be used as a rough estimate.

B) *Intermediate COCOMO* and *Detailed COCOMO*. In the intermediate COCOMO, a nominal effort estimation is obtained using the power function with three sets of $\{a, b\}$, with coefficient a being slightly different from that of the basic COCOMO:
   (1) for simple, well-understood applications, $a = 3.2$, $b = 1.05$
   (2) for more complex systems, $a = 3.0$, $b = 1.15$
   (3) for embedded systems, $a = 2.8$, $b = 1.20$
   Then, fifteen cost factors with values ranging from 0.7 to 1.66 (see Table 1) are determined [5]. The overall *impact factor* M is obtained as the product of all individual factors, and the estimate is obtained by multiplying M to the nominal estimate.

Table 1: The cost factors and their weights in COCOMO II

Cost Factors	Description	Rating				
		Very low	low	nominal	High	very high
	Product					
RELY	required software reliability	0.75	0.88	1.00	1.15	1.40
DATA	database size	-	0.94	1.00	1.08	1.16
CPLX	product complexity	0.70	0.85	1.00	1.15	1.30
	Computer					
TIME	execution time constraint	-	-	1.00	1.11	1.30
STOR	main storage constraint	-	-	1.00	1.06	1.21
VIRT	virtual machine volatility	-	0.87	1.00	1.15	1.30
TURN	computer turnaround time	-	0.87	1.00	1.07	1.15
	Personnel					
ACAP	analyst capability	1.46	1.19	1.00	0.86	0.71
AEXP	application experience	1.29	1.13	1.00	0.91	0.82
PCAP	programmer capability	1.42	1.17	1.00	0.86	0.70
VEXP	virtual machine experience	1.21	1.10	1.00	0.90	-
LEXP	language experience	1.14	1.07	1.00	0.95	-
	Project					
MODP	modern programming practice	1.24	1.10	1.00	0.91	0.82
TOOL	software tools	1.24	1.10	1.00	0.91	0.83
SCED	development schedule	1.23	1.08	1.00	1.04	1.10

While both basic and intermediate COCOMOs estimate the software cost at the system level, the detailed COCOMO works on each sub-system separately and has an obvious advantage for large systems that contain non-homogeneous subsystems.

C) COCOMO II. Perhaps the most significant difference from the early COCOMO models is that the exponent *b* changes according to the following cost factors: precedentedness, development flexibility, architecture or risk resolution, team cohesion, and process maturity. Other differences include newly added cost factors and models for solidifying software architecture and reducing risk.

Putnam's model and SLIM

Putnam derives his model based on Norden/Rayleigh manpower distribution and his finding in analyzing many completed projects [30]. The central part of Putnam's model is called *software equation* as follows:

$$S = E \times (\text{Effort})^{1/3} t_d^{4/3}$$

where t_d is the software delivery time; E is the *environment factor* that reflects the development capability, which can be derived from historical data using the software equation. The size S is in LOC and the Effort is in person-year. Another important relation found by Putnam is

$$\text{Effort} = D_0 \times t_d^3$$

where D_0 is a parameter called *manpower build-up* which ranges from 8 (entirely new software with many interfaces) to 27 (rebuilt software). Combining the above equation with the software equation, we obtain the power function form:

$$\text{Effort} = (D_0^{4/7} \times E^{-9/7}) \times S^{9/7} \text{ and}$$

$$t_d = (D_0^{-1/7} \times E^{-3/7}) \times S^{3/7}$$

Putnam's model is also widely used in practice and SLIM is a software tool based on this

model for cost estimation and manpower scheduling.

4.2.5 Model calibration using linear regression

A direct application of the above models does not take local circumstances into consideration. However, one can adjust the cost factors using the local data and linear regression method. We illustrate this model calibration using the general power function model: Effort = $a \times S^b$.

Take logarithm of both sides and let Y = log(Effort), A = log(a) and X = log(S). The formula is transformed into a linear equation:

$$Y = A + b \times X$$

Applying the standard least square method to a set of previous project data $\{Y_i, X_i: i = 1, \ldots, k\}$, we obtain the required parameters b and A (and thus a) for the power function.

4.2.6 Discrete models

Discrete models have a tabular form, which usually relates the effort, duration, difficulty and other cost factors. This class of models contains Aron model [3], Wolverton model [39], and Boeing model [4]. These models gained some popularity in the early days of cost estimation, as they were easy to use.

4.2.7 Other models

Many other models exist and the following have been used quite successfully in practice.

Price-S is proprietary software cost estimation model developed and maintained by RCA, New Jersey [27]. Starting from an estimate of project size, type and difficulty, the model computes project cost and schedule.

SoftCost relates size, effort and duration to address risk using a form of the Rayleigh probability distribution [36]. It contains heuristics to guide the estimators in dealing with new technology and complex relations among the parameters involved.

The algorithmic models can be grouped as shown in Table 2.

Table 2. Classification of algorithmic models

Algorithmic Models					
	Linear	**Multiplicative**	**Power function**	**Discrete**	**Others**
Empirical	Nelson	Walston-Felix Herd et al	COCOMOS	Aron Boeing Wolverton	Price-S
Analytical			Putnam		SoftCost

Table 3 compares the strengths and weaknesses of different methods. From the comparison, we can conclude that

- No one method is the best for all projects.
- Parkinson and Price-to-win methods are not suitable for organizations which aim to win more business.
- Using a combination of techniques may provide the best estimation. For example, combining top-down estimation with expert judgement and analogy methods may provide a superior

result.

Table 3. Summary of strengths and weaknesses of different methods

Methods	Strengths	Weaknesses
Non-algorithmic		
Expert Judgment	Expert with the relevant experience can provide good estimation; Fast estimation	Dependent on the 'expert'; May be bias; Suffer from incomplete recall
Analogy	Based on actual project data and past experience	Similar projects may not exist; Historical data may not be accurate
Parkinson Price-to-win	Often win the contract	Poor practice; May have large overruns
Top-down	System level focus; Faster and easier than bottom-up method; Require minimal project detail	Provide little detail for justifying estimates; Less accurate than other methods
Bottom-up	Based on detailed analysis; Support project tracking better than other method, as its estimates address low level tasks	May overlook system level cost factors; Require more estimation effort compared to Top-down; Difficult to perform the estimate early in the lifecycle
Algorithmic	Objective, repeatable results; Gain a better understanding of the estimation method	Subjective inputs; Calibrated to past projects and may not reflect the current environment; Algorithms may be company specific and not be suitable for software development in general

4.3 Measurement of model performance

Various researchers have used different error measurements. The most popular error measure is *Mean Absolute Relative Error (MARE)*:

$$\text{MARE} = \sum_{i=1}^{n} (|(\text{estimate}_i - \text{actual}_i)/\text{actual}_i|) / n$$

where estimate_i is the estimated effort from the model, actual_i is the actual effort, and n is the number of projects.

To establish whether models are biased, the *Mean Relative Error (MRE)* can be used:

$$\text{MRE} = \sum_{i=1}^{n} ((\text{estimate}_i - \text{actual}_i)/\text{actual}_i) / n$$

A large positive MRE would suggest that the model generally overestimates the effort, while a large negative value would indicate the reverse.

The following criteria can be used for evaluating cost estimation models [5]:

1. *Definition* – Has the model clearly defined the costs it is estimating, and the costs it is excluding?
2. *Fidelity* – Are the estimates close to the actual costs expended on the projects?
3. *Objectivity* – Does the model avoid allocating most of the software cost variance to poorly calibrated subjective factors (such as complexity)? Is it hard to adjust the model to obtain any result the user wants?
4. *Constructiveness* – Can a user tell why the model gives the estimates it does? Does it help the user understand the software job to be done?

5. *Detail* – Does the model easily accommodate the estimation of a software system consisting of a number of subsystems and units? Does it give accurate phase and activity breakdowns?
6. *Stability* – Do small differences in inputs produce small differences in output cost estimates?
7. *Scope* – Does the model cover the class of software projects whose costs the user need to estimate?
8. *Ease of Use* – Are the model inputs and options easy to understand and specify?
9. *Prospectiveness* – Does the model avoid the use of information that will not be well known until the project is complete?
10. *Parsimony* – Does the model avoid the use of highly redundant factors, or factors which make no appreciable contribution to the results?

5. Performance of estimation models

Many studies have attempted to evaluate the cost estimation models. Unfortunately, the results are not encouraging, as many of them were found to be not very accurate. Kemerer performed an empirical validation of four algorithmic models (SLIM, COCOMO, Estimacs, and FPA) [22]. No recalibration of models was performed on the project data, which was different from that used for model development. Most models showed a strong over estimation bias and large estimation errors, ranging from a MARE of 57% to 800%.

Vicinanza, Mukhopadhyay and Prietula used experts to estimate the project effort using Kemerer's data set without formal algorithmic techniques and found the results outperformed the models in the original study [37]. However, the MARE ranges from 32 to 1107%.

Ferens and Gurner evaluated three models (SPANS, Checkpoint, and COSTAR) using 22 projects from Albrecht's database and 14 projects from Kemerer's data set [11]. The estimation error is also large, with MARE ranging from 46% for the Checkpoint model to 105% for the COSTAR model.

Another study on COCOMO also found high error rates, averaging 166% [23].

Jeffery and Low investigated the need for model calibration at both the industry and organization levels [19]. Without model calibration, the estimation error was large, with MARE ranging from 43% to 105%.

Jeffery, Low and Barnes later compared the SPQR/20 model to FPA using data from 64 projects from a single organization [20]. The models were recalibrated to the local environment to remove estimation biases. Improvement in the estimate was observed with a MARE of 12%, reflecting the benefits of model calibration.

There were also studies based on the use of analogy. With the use of a program called ANGEL that was based on the minimization of Euclidian distance in n-dimensional space, Shepperd and Schofield found that estimating by analogy outperformed estimation based on statistically derived algorithms [33].

Heemstra surveyed 364 organizations and found that only 51 used models to estimate effort and that the model users made no better estimate than the non-model users [15]. Also, use of estimation models was no better than expert judgment.

A survey of software development within JPL found that only 7% of estimators use algorithmic models as a primary approach of estimation [17].

6. New approaches

Cost estimation remains a complex problem, which continues to attract considerable research attention. Researchers have attempted different approaches. Recently, models based on artificial intelligence techniques have been developed. For example, Finnie and Wittig applied artificial neural networks (ANN) and case-based reasoning (CBR) to estimation of effort [12]. Using a data set from the Australian Software Metrics Association, ANN was able to estimate development

effort within 25% of the actual effort in more than 75% of the projects, and with a MARE of less than 25%. However, the results from CBR were less encouraging. In 73% of the cases, the estimates were within 50% of the actual effort, and for 53% of the cases, the estimates were within 25% of the actual.

In a separate study, Mukhopadhyay, Vicinanza and Prietula found that an expert system based on analogical reasoning outperformed other methods [24].

Srinivasan and Fisher used machine learning approaches based on regression trees and neural networks to estimate costs [34]. The learning approaches were found to be competitive with SLIM, COCOMO, and function points, compared to the previous study by Kemerer [22]. A primary advantage of learning systems is that they are adaptable and nonparametric.

Briand, El Eman, and Bomarius proposed a hybrid cost modeling method, *COBRA*: Cost estimation, Benchmarking and Risk Analysis [7]. This method was based on expert knowledge and quantitative project data from a small number of projects. Encouraging results were reported on a small data set.

7. Conclusion

Today, almost no model can estimate the cost of software with a high degree of accuracy. This state of the practice is created because

(1) there are a large number of interrelated factors that influence the software development process of a given development team and a large number of project attributes, such as number of user screens, volatility of system requirements and the use of reusable software components.
(2) the development environment is evolving continuously.
(3) the lack of measurement that truly reflects the complexity of a software system.

To produce a better estimate, we must improve our understanding of these project attributes and their causal relationships, model the impact of evolving environment, and develop effective ways of measuring software complexity.

At the initial stage of a project, there is high uncertainty about these project attributes. The estimate produced at this stage is inevitably inaccurate, as the accuracy depends highly on the amount of reliable information available to the estimator. As we learn more about the project during analysis and later design stages, the uncertainties are reduced and more accurate estimates can be made. Most models produce exact results without regard to this uncertainty. They need to be enhanced to produce a range of estimates and their probabilities.

To improve the algorithmic models, there is a great need for the industry to collect project data on a wider scale. The recent effort of ISBSG is a step in the right direction [18]. They have established a repository of over 790 projects, which will likely be a valuable source for *builders* of cost estimation model.

With new types of applications, new development paradigms and new development tools, cost estimators are facing great challenges in applying known estimation models in the new millenium. Historical data may prove to be irrelevant for the future projects. The search for reliable, accurate and low cost estimation methods must continue. Several areas are in need of immediate attention. For example, we need models for development based on formal methods, or iterative software process. Also, more studies are needed to improve the accuracy of cost estimate for maintenance projects.

References

1. A. J. Albrecht, and J. E. Gaffney, "Software function, source lines of codes, and development effort prediction: a software science validation", *IEEE Trans Software Eng.* SE-9, 1983, pp.639-648.

2. U. S. Army, *Working Schedule Handbook, Pamphlet No.* 5-4-6, Jan 1974.
3. J. D. Aron, *Estimating Resource for Large Programming Systems*, NATO Science Committee, Rome, Italy, October 1969.
4. R.K.D. Black, R. P. Curnow, R. Katz and M. D. Gray, *BCS Software Production Data*, Final Technical Report, RADC-TR-77-116, Boeing Computer Services, Inc., March 1977.
5. B. W. Boehm, *Software engineering economics*, Englewood Cliffs, NJ: Prentice-Hall, 1981.
6. B.W. Boehm et al "The COCOMO 2.0 Software Cost Estimation Model", *American Programmer*, July 1996, pp.2-17.
7. L. C. Briand, K. El Eman, F. Bomarius, “COBRA: A hybrid method for software cost estimation, benchmarking, and risk assessment”, *International conference on software engineering*, 1998, pp. 390-399.
8. G. Cantone, A. Cimitile and U. De Carlini, “A comparison of models for software cost estimation and management of software projects”, in *Computer Systems*: *Performance and Simulation*, Elisevier Science Publishers B.V., 1986.
9. W. S. Donelson, “Project planning and control”, *Datamation*, June 1976, pp. 73-80.
10. N. E. Fenton and S. L. Pfleeger, *Software Metrics*: *A Rigorous and Practical Approach*, PWS Publishing Company, 1997.
11. D. V. Ferens, and R. B. Gumer, “An evaluation of three function point models of estimation of software effort”, *IEEE National Aerospace and Electronics Conference*, vol. 2, 1992, pp. 625-642.
12. G. R. Finnie, G. E. Wittig, AI tools for software development effort estimation, *Software Engineering and Education and Practice Conference*, IEEE Computer Society Press, pp. 346-353, 1996.
13. M. H. Halstead, *Elements of software science*, Elsevier, New York, 1977.
14. P. G. Hamer, G. D. Frewin, “M.H. Halstead’s Software Science – a critical examination”, *Proceedings of the 6th International Conference on Software Engineering*, Sept. 13-16, 1982, pp. 197-206.
15. F. J. Heemstra, “Software cost estimation”, *Information and Software Technology*, vol. 34, no. 10, 1992, pp. 627-639.
16. J. R. Herd, J.N. Postak, W.E. Russell and K.R. Steward, "Software cost estimation study — Study results", *Final Technical Report, RADC-TR77-220*, Vol. I, Doty Associates, Inc., Rockville, MD, 1977.
17. J. Hihn and H. Habib-Agahi, “Cost estimation of software intensive projects: a survey of current practices”, *International Conference on Software Engineering*, 1991, pp. 276-287.
18. ISBSG, International software benchmarking standards group, http://www.isbsg.org/au.
19. D. R. Jeffery, and G. C. Low, “Calibrating estimation tools for software development”, *Software Engineering Journal*, July 1990, pp. 215-221.
20. D. R. Jeffery, G. C. Low, and M. Barnes, “A comparison of function point counting techniques”, *IEEE Trans on Soft. Eng.*, vol. 19, no. 5, 1993, pp. 529-532.
21. C. Jones, *Applied Software Measurement, Assuring Productivity and Quality*, McGraw-Hill, 1997.
22. C.F. Kemerer, “An empirical validation of software cost estimation models”, *Communications of the ACM*, vol. 30, no. 5, May 1987, pp. 416-429.
23. Y. Miyazaki, and K. Mori, “COCOMO evaluation and tailoring”, *Eighth Int. Conf. Software Engineering*, 1985, pp. 292-299.
24. T. Mukhopadhyay, S. S. Vicinanza, and M. J. Prietula, “Examining the feasibility of a case-based reasoning model for software effort estimation”, *MIS Quarterly*, vol. 16, no. 2, June 1992, pp. 155-172.
25. G. Myers, “Estimating the costs of a programming system development project”, Systems Development Div., Poughkeepsie Lab., IBM, May 1972.
26. R. Nelson, *Management HandBook for the Estimation of Computer Programming Costs*, AD-

A648750, Systems Development Corp., 1966.
27. R. E. Park, “PRICE S: The calculation within and why”, *Proceedings of ISPA Tenth Annual Conference*, Brighton, England, July 1988.
28. G.N. Parkinson, *Parkinson's Law and Other Studies in Administration*, Houghton-Miffin, Boston, 1957.
29. N. A. Parr, “An alternative to the Raleigh Curve Model for Software development effort”, *IEEE on Software Eng.* May 1980.
30. L. H. Putnam, “A general empirical solution to the macro software sizing and estimating problem”, *IEEE Trans. Soft. Eng.*, July 1978, pp. 345-361.
31. W. Royce, *Software project management: a unified framework*, Addison Wesley, 1998
32. V. Y. Shen, S. D. Conte, H. E. Dunsmore, “Software Science revisited: a critical analysis of the theory and its empirical support”, *IEEE Transactions on Software Engineering*, 9, 2, 1983, pp. 155-165.
33. M. Shepperd and C. Schofield, “Estimating software project effort using analogy”, *IEEE Trans. Soft. Eng.* SE-23:12, 1997, pp. 736-743.
34. K. Srinivasan and D. Fisher, “Machine learning approaches to estimating software development effort”, *IEEE Trans. Soft. Eng.*, vol. 21, no. 2, Feb. 1995, pp. 126-137.
35. D. St-Pierre, M Maya, A. Abran, J. Desharnais and P. Bourque, *Full Function Points: Counting Practice Manual*, Technical Report 1997-04, University of Quebec at Montreal, 1997.
36. R. Tausworthe, *Deep Space Network Software Cost Estimation Model*, Jet Propulsion Laboratory Publication 81-7, 1981.
37. S. S. Vicinanza, T. Mukhopadhyay, and M. J. Prietula, “Software-effort estimation: an exploratory study of expert performance”, *Information Systems Research*, vol. 2, no. 4, Dec. 1991, pp. 243-262.
38. C. E. Walston and C. P. Felix, “A method of programming measurement and estimation”, *IBM Systems Journal*, vol. 16, no. 1, 1977, pp. 54-73.
39. R. W. Wolverton, “The cost of developing large-scale software”, *IEEE Trans. Computer*, June 1974, pp.615-636.