ELEN 7046– Software Technologies and Techniques

Assignment 3: Software Estimation Techniques

June 2015

Name	Student Nr
Adrian Freemantle	1307968

Abstract

Software effort estimation plays a crucial role in the success of any software project as it drives the allocation of budgets and determines the duration of the development effort. This information is most important at the early stages of a project in order to determine the software project's feasibility. Incorrect software effort estimates negatively impact productivity and the resulting software's profitability and quality. Model based, expertise based and learning oriented estimation categories are discussed in order to determine the relative merits, estimation techniques and applicable problem domains of each estimation technique. The author concludes that the field of software estimation cannot yet be considered to be a mature field and that no single technique is likely to provide a solution, instead multiple estimation techniques that are suitable to the problem domain should be used in order to derive an accurate estimate.

Table of Contents

1 Introduction

The success of any project is determined by its completion within the allocated budget, in time and with the required level of quality. Time and budgetary constraints need to be understood at an early stages of a project to determine its feasibly. In order to determine these factors estimates of effort are needed as soon as possible [1, p. 103].

The importance of early and accurate software effort estimates will be discussed along with the challenges of obtaining such estimates and the dysfunctions that arise from overestimation or underestimation.

Barry Boehm has argued that software effort estimation techniques can be categorised into six distinct groups; model based, expertise based, learning oriented, dynamics based, regression based and composite Bayesian [2].

This paper will evaluate the estimation techniques from the model-based, expertise-based and learning oriented categories to determine their advantages, disadvantages and applicable problem domains. These techniques will be evaluated to determine if there is a single best option or whether a variety of techniques should be used when estimating.

2 Software Effort Estimation

As software projects continue to grow in size, cost and complexity the need for early and accurate estimates is becoming more important than ever and at the same time more difficult to achieve [2].

In 1965 the System Development Corporation (SDC) evaluated 169 software projects on behalf of the United States Air Force in order to determine which attributes of a project affected the development effort. The outcome of this study was the 1966 publication of the first set of comprehensive cost-estimation models; the *Management Handbook for the Estimation of Computer Programming* (Nelson, 1966) [2].

Despite these developments, inaccurate estimates continued to be a problem. Rapid improvements in technology along with larger and more complex software projects resulted in a need for cost-estimation models that could *explain the development life-cycle and accurately predict the cost of developing a software product* [2].

The late 1970s and early 1980s saw the emergence of the first parameterised estimation models such as COCOMO and Function Points with refinements such as COCOMO II and COSMIC being published in the 1990s. New estimation techniques continued to be developed such Planning Poker in the 2000s and the emergence neural networks that could learn from previous case studies.

Over time the accuracy of the estimates made by these models have improved, but overestimation or underestimation continues to be a problem. As an example, a deviation of 20% for the actual cost of implementing a project as compared to the estimated cost is still considered to be a success [1, p. 108].

Overestimation of software effort may result in the project being overstaffed, resulting in higher management overhead and reduced productivity as each individuals productivity decreases as team size increases [3, p. 181]. Another form of inefficiency is the effect known as Parkinson's Law where *"work expands to fill the time available"* [1, p. 107]. In general overestimates result in reduced productivity and have a negative impact on the profitability of the project.

Underestimation of software effort is likely to cause the project to overrun both budgetary and time constraints. A popular opinion is that the project will still be completed in a shorter period than with a larger estimate (Parkinson's Law), however pressure on the team may negatively impact on the quality aspects of the final product. The pressure of constantly missed deadlines will also have a negative impact on staff morale and motivation which in turn has a negative impact on productivity [1, p. 107].

3 Software Effort Estimation Models

Software effort estimation is most often needed, and most difficult to obtain, during the earlier stages of a project in order to determine the feasibility of the endeavour. At this point the design will not yet be finalised and the team that will implement the project may not yet be identified. As a result early estimates frequently focus on determining the system size rather than the expected duration [1, p. 108] [3, p. 157]. This can be expressed as:

effort = (system size) x (productivity rate)

3.1 Algorithmic Models

Algorithmic models use the estimated size of the software as the primary input for deriving the estimated effort [3, p. 155]. The base formula used by most of these models can be expressed as:

$$E = a + b\text{KLOC}^c$$

Where E is the effort as measured in man-months, KLOC is the size of the software measured in 'thousands of lines of code' and the values of *a*, *b* and *c* are constants that are determined by the specific model being applied. It is clear from this base formula that the size of the project is seen as having a strong correlation with the resulting effort [3, p. 155].

The value of the exponent *c* is intended to reflect the economies of scale. Certain models assume that larger software projects are less expensive in relative terms due the economies of scale presented by the opportunity for code reuse and the greater distribution of fixed costs such as software tools over the total project. Other models assume increased costs due to the diseconomies of scale such as increased complexity of the code base and the inefficiencies introduced by larger teams [3, p. 159].

Algorithmic cost-estimation models can be determined via the results of careful experimentation in controlled conditions, however the findings of these experiments are

rarely applicable to large real-world projects. A more realistic approach is to derive a model from extensive regression analysis of historical data from real projects [3, p. 155].

> *"Regression analysis is a statistical tool for the investigation of relationships between variables… to ascertain the causal effect of one variable upon another" [4]*

The reliability of the resulting estimate derived from an algorithmic model depends on the quality of the data used to build the model and it's applicability to the project being estimated. New problem domains, the degree of innovation needed and changes in technology must be taken into consideration when selecting an estimation model. [3, p. 155]

Algorithmic cost-estimation models are considered to be most applicable to plan-driven projects which have longer delivery cycles as compared to agile projects which have short iterations with estimations being given in points or some other artificial unit [3, p. 157].

The accuracy of these models are disputed as estimation is typically done during the early stages of a project while there is very little information available. They do however provide insight into what factors have the largest impact on the overall development productivity and estimation can be repeated as addition information is made available [3, p. 157].

3.1.1 COCOMO and COCOMO II

The COnstructive COst MOdel (COCOMO) was published by Barry Boehm in 1981 based on his analysis of 63 diverse software projects and uses the base formula described in section 3.1. COCOMO provides a table of constants to be substituted for the values of b and c depending on the classification of the system as being 'organic', 'embedded' or 'semidetached' [1, p. 162].

Organic: small teams of skilled developers with experience in the problem domain where the projects are not exceptionally large.
Values of $b = 2.4$ and $c = 1.05$ are used.

Embedded: The software will run in an inflexible imbedded environment with significant constraints such as reliability or limited memory and processing capabilities.
Values of $b = 3.0$ and $c = 1.12$ are used.

Semidetached: The project may consist of a mix of various elements such as experienced and inexperienced staff with some constraints and possibly of a large nature.
Values of $b = 3.6$ and $c = 1.20$ are used.

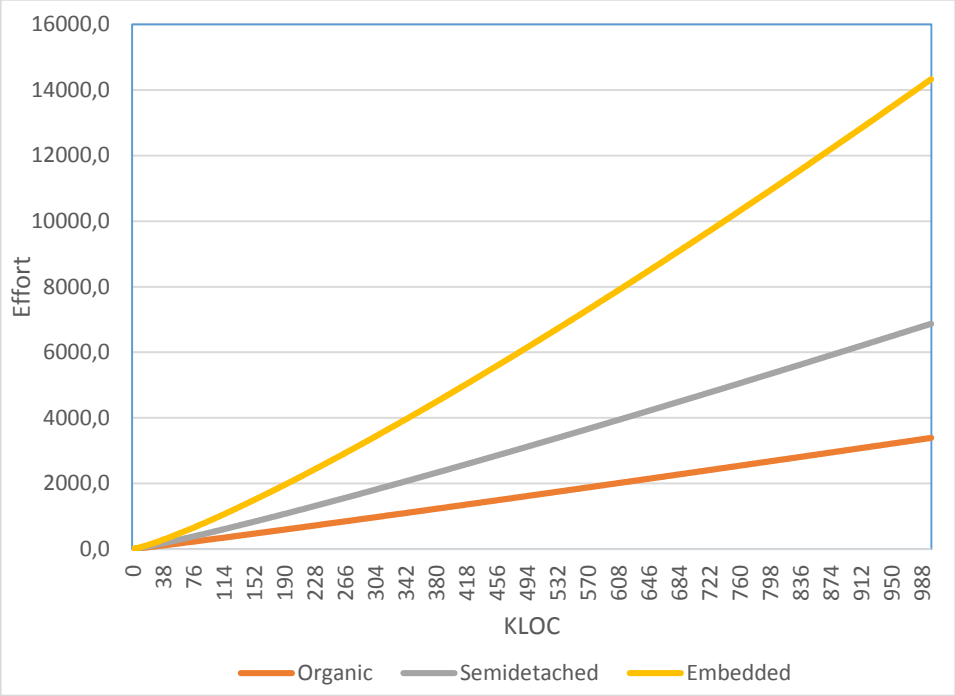

Figure 3-1 COCOMO effort per class type

COCOMO provided a simple but crude estimate that proved unreliable with the emergence of new software development processes and methodologies. This lead to the development and publishing of COCOMO II in 1995 [2].

COCOMO II introduced three different submodels to adjust for different development practices. These include the Application Composition model for prototyping stage, the Early Design model for the architectural design stage, and the Post-Architecture model for the development stage [2] [3, p. 168].

COCOMO II replaced the organic, semidetached and embedded modes with five scaling factors and a series of effort multipliers specific to each submodel. Scaling factors affect the value of the exponent *c* and thus impact the economies or diseconomies of scale while effort multipliers affect the value of multiplier *b* and are designed to account for factors that affect the overall productivity [3, p. 172] [1, p. 124]. The five scaling factors are:

- **Precedentedness** reflects the novelty, innovation required, experience with similar projects, uncertainty in the approach and implementation of the solution.
- **Development flexibility** is the degree to which conformance to specific constraints is required.
- **Architecture/risk resolution** is the level of uncertainty in the design and the requirements as well as the number of assumptions that have to be made.
- **Team cohesion** is a measure of the size and geographic distribution of the team.
- **Process maturity** is the maturity of the processes and procedures used by the organisation.

3.1.2 Albrecht Function Point Analysis

Function point analysis was developed for IBM in 1979 by Allan Albrecht as a means estimating the effort for a project without knowing the expected number of lines of code or which programming language it would be implemented in. This method uses the number of external inputs, outputs and the complexity of data structures as parameters for size calculation [2] [3, p. 165].

Albrecht identified five 'external user types' which could be categorised according to a complexity level in order to determine their weighting for calculating the Unadjusted Function Points (UFP). These five components are [1, p. 114]:

- Input types (I). The number of input transactions that result in changes to internal data.
- Output types (O). The number of output types for reports.
- Enquiry types (E). The number of transactions to display data to the user but that do not change internal data.
- Logical internal files (L). The internal records created by and maintained by the system.
- Interfaces (F). Data that is output or shared with between systems.

Each of these components have an associated weighting in the calculation of the UFP:

$$UFP = 4I + 5O + 4E + 10L + 7F$$

Function point analysis allows for the adjustment of UFP to adjusted function points (FP) through the application of a technical complexity adjustment (TCA). The TCA considers fourteen operational environment factors that can have an impact on the complexity of the final implementation. There is some contention over this adjustment as it does not appear to provide a more accurate result than the UFP [1, p. 116].

Various tables exist to convert FPs to an equivalent number of lines of code for a specific language. One FP is equal to 100 lines of COBOL, 53 lines of Java, and 34 lines of C++ [1, p. 166] [3, p. 167]. We can therefore see that 100 FP are equal to one KLOC of COBOL. This conversion allows for the use of KLOC as an input parameter for an algorithmic model such as COCOMO in order to derive an estimate of effort.

Function point analysis is most useful when estimating business applications where the focus is on the input and output of complex data structures as well as batch processing of data. It fails to provide accurate estimates for projects such as embedded systems, scientific systems or any other calculation intensive application due to the limited number of 'external user types' found in such systems [3, p. 165].

3.1.3 COSMIC

The Common Software Measurement International Consortium (COSMIC) project was created in 1999 to address the problems with using Function Points in real-time and embedded applications. The result has been the creation of the COSMIC Full Function

Point (FFP) which measures the internal interactions between components [2]. This is approach is suited to embedded and real time systems where internal interactions between components are more important than the 'external user types' measured in standard Function Point analysis. These internal interactions are categorised into four groups [1, p. 120]:

- Entries (E), points where data is moved into the component by a 'user' outside the component boundary.
- Exits (X), points where data is moved out of the component to a 'user' outside the component boundary.
- Reads (R), points where data is moved into the component from persistent storage.
- Writes (W), points where data is moved to persistent storage.

The FPP count for a component is the sum of the counts of all these data movements.

COSMIC FPP is not suitable for systems that where components perform complex internal processing of data as it only takes the number of data movements into consideration [1, p. 120].

3.2 Expert Judgement

An expert judgement, or expert opinion, is an estimate that is produced by an individual based on their expertise in the technology or problem domain [1, p. 112]. The accuracy of estimates produced using this technique are strongly affected by the skill and experience of the estimator and the number of assumptions that need to be made. It is difficult to verify the accuracy of the estimate or determine the relationships between various factors taken into consideration [2].

Expert judgements have been found to work well for estimating effort related to the maintenance or modification of existing software and the accuracy can be improved by having the expert justify their estimate and by combining the opinions of more than one estimator [1, p. 113].

3.2.1 Bottom-up Estimation

Bottom-up estimation is a technique that can best used during the early stages of a completely novel or unprecedented project or during the later stages of a project when more detailed information is available [1, p. 109].

The process starts by creating a Work Breakdown Structure by recursively decomposing the project into its component elements and associated tasks to the point where each task's estimated effort is around one to two weeks. The overall estimate is obtained by summing the estimates for the lowest level tasks in the Work Breakdown Structure [1, p. 109].

Early estimates made using this technique with no historical data will require many assumptions to be made by the estimator which in turn will negatively impact the accuracy

of the estimate. The process can however be repeated as more information becomes available [2].

3.2.2 Delphi Technique

Many of the problems of expert judgement can be negated through group estimation where the estimates of multiple experts are considered. A downside of this process that it can be difficult to reach consensus or the group may be dominated by a small number of individuals who exert undue influence on the final estimate [1, p. 227].

The Delphi technique was developed in the 1940s by the Rand Corporation as a group-decision making process that aimed to guide a group toward reaching consensus without any direct interaction between the group members. This same process can be applied to software effort estimation by having a group of experts estimate the same project, component or task. This helps reduce some of the problems associated with Expert Judgements by drawing from the collective experience of a group of experts [2].

The process starts by enlisting a group of experts and then presenting a project, component or task to be estimated. The members of the group do not consult with each other and return their initial estimates and justifications to the coordinator. The estimates are collated, reproduced and recirculated to the group members who may refine their own estimates and to comment on those of other group members. These results are again collated, reproduced and recirculated. This process continues until a consensus has been reached [1, p. 227] [2].

Boehm has argued that this process is most useful when no empirical data is available on which to base estimates other than the "expert opinion" of a group of individuals [2].

3.2.3 Story Points

Story points are a relative measure of a feature's overall size and complexity and are predominantly used in agile development processes where detailed up-front cost estimation is not done for the whole project [1, p. 183]. Story points do not have a relation to time and are instead on indication of the development team's estimate of how much effort one feature requires to implement in relation to another [5] [1, p. 183].

A common way in which a story point estimation session is started is to estimate a feature that is considered to be of an average size and provide it with a mid-range point value. This estimate then provides the benchmark against which all other features for the current iteration are compared. As an example, if feature A is estimated to be 10 points of effort and feature B is estimated to require three times more much effort than A, the estimate for B will be 30 points [5] [1, p. 183].

Over time a team's performance can be measured to determine their "velocity" or number of story points delivered per iteration. It is assumed that the team's velocity and estimation accuracy will improve over time, provided no changes are made to the team or their development processes [3, p. 157].

A significant disadvantage of this method is that a story point estimate and a team's velocity are only applicable to the current project for that specific team. One team may have a velocity of 100 story points per iteration while another may have a velocity of 50 points on the same project. It is not possible to infer from that information that one team is twice as productive as the other as the story points are a subjective measure for each team [3, p. 184].

3.2.4 Planning Poker

Planning poker was developed by James Grenning in 2002 as lightweight technique for story-point estimation. Planning poker, like the Delphi technique, is used to obtain consensus on estimates from a group of experts. The main difference between the two techniques is that planning poker aims to obtain faster consensus through direct interaction between team members [6].

> *"Planning poker combines expert opinion, analogy, and disaggregation into an enjoyable approach to estimating that results in quick but reliable estimates." [5]*

At the beginning of an estimation session each estimator is provided with a set of cards. Each set of cards will contain a sequence of numeric numbers that typically follow an exponential or Fibonacci number series ranging from 0 to 100 [6] [3, p. 183].

The team will be presented with each feature or user-story to be estimated after which they will select a card with a value representing their estimate. Once all team members have completed their selection the cards are revealed and placed on the table. If there is no consensus the highest and lowest estimators are asked to justify their estimates. The process is then repeated until a reasonable consensus has been reached [6].

An advantage of planning poker is that by having team members estimate together and justify their estimates, details around assumptions, design, implementation and domain specific knowledge are shared among the team [3, p. 183].

3.3 Learning Oriented

As with algorithmic models and expert judgement, learning-oriented techniques or estimation by analogy, use information about past projects to estimate effort for the current project. A significant difference is that learning-oriented techniques use inductive-based reasoning, where similar projects are analysed in order to extract estimation heuristics. The goal is that the derived estimation heuristics will provide guidelines that are more objective than expert judgement while being more specific than generic algorithmic models [2] [1, p. 133].

A disadvantage of learning-oriented techniques are that they require historical information from previously completed projects that are similar in nature to the current project being estimated. This problem can be circumvented by referring to external sources such the International Software Benchmarking Standards Group (ISBSG) which have created a project database covering the results of over 4800 projects [1, p. 108].

3.3.1 Estimation by Analogy

Estimation by analogy or case studies is a manual process through which estimators identify studies on previous projects with similar properties to the current project being estimated [1, p. 113]. The rule of analogy states that projects that share properties such as problem domains, constraints, technologies, team size and development methodologies are likely to have similar estimates [2].

Once the estimators have selected the relevant case studies they identify any differences between the project being estimated and the case studies being used. These differences are analysed in order to fine tune the current estimation model.

While estimation by analogy has been shown to be more accurate than algorithmic models [2], the accuracy of the estimate is dependent on the correct identification of similarities between the project being estimated and the case studies being considered. Availability of relevant case studies also poses a problem [1, p. 113].

3.3.2 Neural Networks

Neural networks are a form of artificial intelligence that are "trained" by analysing the results of a large number of historical case studies in order to "learn" how to estimate effort for a software project. The learning takes place by comparing specific input parameters for each case study with the actual results achieved. As the neural network "learns" from each case study its ability to estimate improves [2].

Unlike with human based estimation-by-analogy, neural networks are able to objectively process a vast number of case studies and can be more effective at reducing the variance in estimated effort. Neural network estimates have been shown to be accurate to within 10% of the actual costs incurred on a project [2].

A potential problem with neural networks is the possibility of over or under training. An extremely large number of case studies are required for the neural network to be sufficiently trained. If the selected case studies cover a narrow a spectrum of project types, the network will be over trained for that specific case and will not perform well in other scenarios. Should too few case studies be used, the network will be undertrained and will again perform poorly [2].

4 The Future of Software Effort Estimation

The number of estimation techniques, the differences in their approaches, continued dissatisfaction with their accuracy [3, p. 157] [2] along with multiple standards being issued for the same estimation techniques (no less than seven ISO standards for Function Points [1, p. 120]) are an indication that the field of software effort estimation is far from mature.

The rapidly changing nature of software development and increases in the size and complexity of software projects will most likely continue to present challenges to any estimation technique and it is unlikely that a one-size-fits-all solution will be found. Even

if an estimation technique could be derived that could accurately estimate the majority of software projects in today's development space, it would almost immediately be obsolete. Software solutions are continually being applied to new problem domains, using innovative architectures and development techniques for which no historical data or case studies exist.

In the end the best estimates will be derived by applying more than one estimation technique where the selected techniques are best suited to the specific problem domain and development environment.

5 Conclusion

Software effort estimation has been shown to play a crucial role in the success of any software project as it drives the allocation of budgets and determine the duration of the development effort. Early estimates were shown to be important at the early stages of a project in order to determine the software project's feasibility.

The challenges of obtaining of early and accurate software effort estimates was discussed and the consequences of incorrect estimates were highlighted.

Three of Boehm's software effort estimation categories were explored; model based, expertise based and learning oriented techniques. The differences these approaches where highlighted along with their relative advantages, disadvantages and applicable problem domains.

The author concluded that the future of software estimation could not yet be considered to be a mature field and that no single technique is likely to provide a single solution to all estimation problems.

6 References

[1] B. Hughes and C. Mike, “Software Project Management,” McGraw-Hill, 2009.

[2] B. Boehm, A. Chris and C. Sunita , “Software Development Cost Estimation Approaches – A Survey,” *Annals of software engineering*, vol. 10, no. 1-4, pp. 177-205, November 200.

[3] H. van Vliet, “Software Engineering - Principles and Practice,” John Wiley & Sons, 2008.

[4] A. Sykes, “An Introduction to Regression Analysis,” [Online]. Available: http://www.law.uchicago.edu/files/files/20.Sykes_.Regression.pdf. [Accessed 16 June 2015].

[5] M. Cohn, Agile Estimation and Planning, Prentice Hall, 2006.

[6] J. Grenning, “Planning Poker,” April 2002. [Online]. Available: http://renaissancesoftware.net/files/articles/PlanningPoker-v1.1.pdf. [Accessed 15 June 2015].