*Bogdan Stępień**

SOFTWARE DEVELOPMENT COST ESTIMATION METHODS AND RESEARCH TRENDS

Early estimation of project size and completion time is essential for successful project planning and tracking. Multiple methods have been proposed to estimate software size and cost parameters. Suitability of the estimation methods depends on many factors like software application domain, product complexity, availability of historical data, team expertise etc. Most common and widely used estimation techniques are described and analyzed. Current research trends in software estimation cost are also presented.

Keywords: *software project effort, size estimation, software cost*

METODY ESTYMACJI KOSZTÓW PRODUKCJI OPROGRAMOWANIA

Wczesna estymacja rozmiaru i czasu zakończenia projektu jest kluczowa dla efektywnego planowania i śledzenia postępów pracy. W celu rozwiązania problemu estymacji rozmiaru i kosztów produkcji oprogramowania opracowano wiele metod. Użyteczność różnych metod estymacji zależna jest od wielu czynników, takich jak obszar zastosowania oprogramowania, złożoność produktu, dostępność danych historycznych, doświadczenie zespołu itd. W niniejszym artykule zostały przedstawione i przeanalizowane najczęściej stosowane techniki estymacji, jak również najnowsze kierunki badań.

Słowa kluczowe: *koszt produkcji oprogramowania, estymacja rozmiaru*

1. Introduction

The emphasis on software cost estimation has been increasing gradually over last three decades. Today, it is especially strong and visible since it provides the link between the general concepts of economic analysis and the world of software engineering. The software cost estimation technique is also an essential part of the foundation for the good software management.

There are several approaches used by models to estimate software development cost (effort to produce the software). Some are based on analogy, some on theory, and others on statistics, but all of them consider size of the software product as the

*Post-graduate student at the Faculty of Electrical Engineering, Automatics, Computer Science and Electronics, AGH University of Science and Technology, Cracow, Poland, Bogdan.Stepien@motorola.com

most influential factor in predicting effort. Other factors that also affect effort are for example product complexity, the experience of the development team, development tool support, project coordination complexity, maturity of the technology in which the software product is to be produced.

The aim of this article is to provide an overview of some software size and cost estimation techniques and to define their strengths and weaknesses. At the end of the article, some of the areas of current and future research in software estimation techniques will also be described.

2. Software Size Estimation Techniques

2.1. Lines of Code Counting

Although the first dedicated books on software metrics were not published until mid-1970's, the history of active software metrics dates back to the mid-1960's when the Lines of Code (LOC) metric was used as the basis for measuring programming productivity and effort.

The LOC metric was, and still is, used routinely as the basis for measuring programmer productivity (LOC per programmer month) and as such LOC was assumed to be a key driver for the effort and cost of developing software. Indeed, the early resource prediction models (such as [17] or [4]) used LOC or related metrics as the key size variable in predictive (regression-based) models.

Table 1
Strengths and weaknesses of the LOC method

Strengths	Weaknesses
Simple metric and directly related to the size	Indication of construction methods used
Suitable for estimation using the Wideband Delphi process	LOC cannot be estimated reliably in the early phases of the development cycle unless data are available from similar, completed projects
	Counting must always follow the same rules defining the LOC measure and must be independent of factors like coding style

Lines of Code (LOC) counting is the simplest way to estimate the size of a software product. It provides simple and well understood metric. Usually LOC estimation is performed by software developers experienced in similar projects and the method is suitable for the Wideband Delphi process (this process will be described in section 3.2.1). The experts analyze the work packages and based on their own experience, they derive the LOC estimate needed to fulfill the requirements of each work package.

Defining a line of code is difficult (see Tab. 1) due to conceptual differences involved in accounting for executable statements and data declarations in different pro-

gramming languages. The goal is to measure the amount of intellectual work put into program development, but difficulties arise when trying to define consistent measures across different languages. To minimize these problems, the Software Engineering Institute (SEI) definition checklist for a logical source statement is used in defining the line of code measure. Pragmatically, there seems to be no real reason to choose one definition over another, so long as the same definition is used consistently.

2.2. Function Point Analysis

The concept of function points had its roots in the '70s with companies like IBM. Albrecht [1, 2] introduced the concept of function points as a software size measure while considering the more general problem of measuring application development productivity. His objective was to develop a software size measure that was independent of the implementation technology. To measure productivity, a measure of work product (or output) has to be defined. For this Albrecht chose *the function value to be delivered to the user.*

Function points are useful estimators since they are based on information that is available early in the project life cycle. To calculate the function value delivered to the user, the number of inputs, outputs, inquires, and files (including interfaces to the other programs) from the user perspective is counted, weighted, and summed. The user function types should be identified, as defined below:

- **External Input (Inputs)** – Count each unique user data or user control input type that (i) enters the external boundary of the software system being measured and (ii) adds or changes data in a logical internal file.
- **External Output (Outputs)** – Count each unique user data or control output type that leaves the external boundary of the software system being measured.
- **Internal Logical File (Files)** – Count each major logical group of user data or control information in the software system as a logical internal file type. Include each logical file (e.g., each logical group of data) that is generated, used, or maintained by the software system.
- **External Interface Files (Interfaces)** – Files passed or shared between software systems should be counted as external interface file types within each system.
- **External Inquiry (Queries)** Count each unique input-output combination, where an input causes and generates an immediate output, as an external inquiry type.

Each instance of these function types is then classified by complexity level. The complexity levels determine a set of weights, which are applied to their corresponding function counts to determine the Unadjusted Function Points quantity (see Fig. 1). This is the Function Point sizing metric used as input by COCOMO II estimation model (described in section 3.4.1).

Step 1. Determine function counts by type. The unadjusted function counts should be counted by a lead technical person based on information in the software requirements and de-sign documents. The number of each of the five user function types should be counted (Internal Logical File (ILF), External Interface File (EIF), External Input (EI), External Output (EO), and External Inquiry (EQ)).

Step 2. Determine complexity-level function counts. Classify each function count into Low, Average and High complexity levels depending on the number of data element types contained and the number of file types referenced. Use the following scheme:

For ILF and EIF			
Record Elements	Data Elements		
	1–19	20–50	51+
1	low	low	avg
2–5	low	avg	high
6+	avg	high	high

For EO and EQ			
Record Elements	Data Elements		
	1–5	6–19	20+
0–1	low	low	avg
2–3	low	avg	high
4+	avg	high	high

For EI			
Record Elements	Data Elements		
	1–4	5–15	16+
0–1	low	low	avg
2–3	low	avg	high
4+	avg	high	high

Step 3. Apply complexity weights. Weight the number in each cell using the following scheme. The weights reflect the relative value of the function to the user.

Function Type	**Complexity-Weight**		
	Low	Average	High
Internal Logical Files	7	10	15
External Interfaces Files	5	7	10
External Inputs	3	4	6
External Outputs	4	5	7
External Inquiries	3	4	6

Step 4. Compute Unadjusted Function Points. Add all the weighted functions counts to get one number, the Unadjusted Function Points.

Fig. 1. Function Point Count Procedure

The standard Function Point procedure involves assessing the degree of influence (DI) of fourteen application characteristics on the software project determined according to a rating scale from 0.0 to 0.05 for each characteristic. The sum of 14 ratings is added to a base level of 0.65 to produce a general characteristics adjustment factor that ranges from 0.65 to 1.35. Each of these fourteen characteristics, such as distributed functions, performance, and reusability, thus has a maximum of 5% contribution to estimated effort.

Table 2
Strengths and weaknesses of the function points

Strengths	Weaknesses
Not dependent on the construction method	Not a technology independent measure of size
Can be used early in the project life cycle	The calculation of function points is complex and tends to take a black box view of the system
A normalized size	Suitability varies for different classes of software systems

Albrecht's original work [1] has grown and mutated over the years – function-point counting has now its own standards group, the International Function Point Users Group. IFPUG has classes on function-point counting and reference manuals with all of the rules. A function-point counting spreadsheet and other resources are available from the [13]. Strengths and weaknesses of the method are summarized in Table 2.

2.3. Use-case Points

Use-case Points are counted from the use-case analysis of a system. They are counted during the early phases of an object-oriented project that captures its scope with use cases. Each use-case is scaled as easy, medium, or hard to produce a point count. The use-case points can be also adjusted for the project's technical and personnel attributes, and then directly converted to hours in order to obtain a rough idea of a nominal project schedule.

The Use-case Points method was developed by Gustav Karner of Objectory (now a part of Rational Software). In 1993, he did research on deriving estimates of a project's required staff hours from the use cases. His work is an extension and modification of Albrecht's work on function points [2] and his method should be used in conjunction with other estimating methods.

Actors are defined as anything external to the system that interfaces with it. Examples include people, other software, hardware devices, data stores, and networks. Use cases describe the things actors want the system to do, such as querying the status of an existing order.

The Use-case Point counting procedure starts with determining for each actor, whether it's simple, average, or complex. You count how many of each kind you have and multiply each by its weighing factor. After adding these products we get the total unadjusted actor weights (UAW). Then, for each use case, you determine whether it's simple (three or fewer transactions), average (four to seven transactions), or complex (eight or more transactions) by counting its transactions, including secondary scenarios. Each use-case type is multiplied by the weighting factor and after adding these products we get the unadjusted use-case weights ($UUCW$). The sum of the UAW and the $UUCW$ gives the unadjusted use-case points ($UUCP$): $UAW + UUCW = UUCP$

The Use-case Points method employs a technical and environmental factors multiplier that attempts to quantify areas such as ease of use and programmer motivation. Those factors, when multiplied by the unadjusted use-case points, produce the adjusted use-case points, an estimate of the size of the software.

To estimate effort, Karner proposed a factor of 20 staff hours per use-case point, although many other factors can affect such a rate, including time pressure, uniqueness of the architectural solution and programming language.

2.4. Object Points

Object Point estimation is a relatively new software sizing approach, but it is well-matched to the application composition phase of software product development. It is also a good match to associated prototyping efforts, based on the use of a rapid-composition Integrated Computer Aided Software Environment (ICASE) providing graphic user interface builders, software development tools, and large, composable infrastructure and applications components. In these areas, it has compared well to Function Point estimation on a nontrivial (but still limited) set of applications [3]. The Object Points are used for sizing in Applications Composition estimation model of the COCOMO II.

Figure 2 presents the baseline COCOMO II Object Point procedure for estimating the effort involved in Applications Composition and prototyping projects [5]. The productivity rates in the figure are based on an analysis of the year-1 and year-2 project data in [3].

Definitions of terms in Figure 2 are as follows:

- *NOP*: New Object Points (Object Point count adjusted for reuse);
- *srvr*: number of server (mainframe or equivalent) data tables used in conjunction with the SCREEN or REPORT;
- *clnt*: number of client (personal workstation) data tables used in conjunction with the SCREEN or REPORT;
- *%reuse*: the percentage of screens, reports, and 3GL modules reused from previous applications, pro-rated by degree of reuse.

Step 1. Assess Object-Counts: estimate the number of screens, reports, and 3GL components that will comprise this app. Assume the standard definitions of these objects in your ICASE environment.

Step 2. Classify each object instance into simple, medium and difficult complexity levels depending on values of characteristic dimensions. Use the following scheme:

For Screens			
Number of Views contained	# and source of data tables		
	Total <4 (<2 srvr <3 clnt)	Total <8 (2/3 srvr 3-5 clnt)	Total 8+ (>3 srvr >5 clnt)
< 3	simple	simple	medium
3–7	simple	medium	difficult
> 8	medium	difficult	difficult

For Reports			
Number of Views contained	# and source of data tables		
	Total <4 (<2 srvr <3 clnt)	Total <8 (2/3 srvr 3–5 clnt)	Total 8+ (>3 srvr >5 clnt)
0 or 1	simple	simple	medium
2 or 3	simple	medium	difficult
4+	medium	difficult	difficult

Step 3. Weigh the number in each cell using the following scheme. The weights reflect the relative effort required to implement an instance of that complexity level:

Object Type	**Complexity-Weight**		
	Simple	Medium	Difficult
Screen	1	2	3
Reports	2	5	8
3GL Compoment			10

Step 4. Determine Object-Points: add all the weighted object instances to get one number, the Object-Point count.

Step 5. Estimate percentage of reuse you expect to be achieved in this project. Compute the New Object Points to be developed, *NOP = (Object-Points) (100-%reuse)/ 100.*

Step 6. Determine a productivity rate, *PROD = NOP / person-month*, from the following scheme:

Developers' experience and ICASE maturity	Very Low	Low	Nominal	High	Very High
PROD	4	7	13	25	50

Step 7. Compute the estimated person-months: *PM = NOP / PROD.*

Fig. 2. Baseline Object Point Estimation Procedure in COCOMO II

3. Software Effort Estimation Techniques

3.1. Theoretical Models

A theory-based effort estimation model was introduced by [17]. It is based on the probability distribution called the Rayleigh curve. This curve presented on Figure 3 expresses manpower distribution on a project over time. The curve is modelled by the differential equation

$$\frac{dy}{dt} = 2Kate^{-at^2} \tag{1}$$

where $\frac{dy}{dt}$ is the staff build-up rate, t is the elapsed time from the start of design to product replacement, K is the area under the curve and represents total life-cycle effort (including maintenance), and a is a constant that determines the shape of the curve.

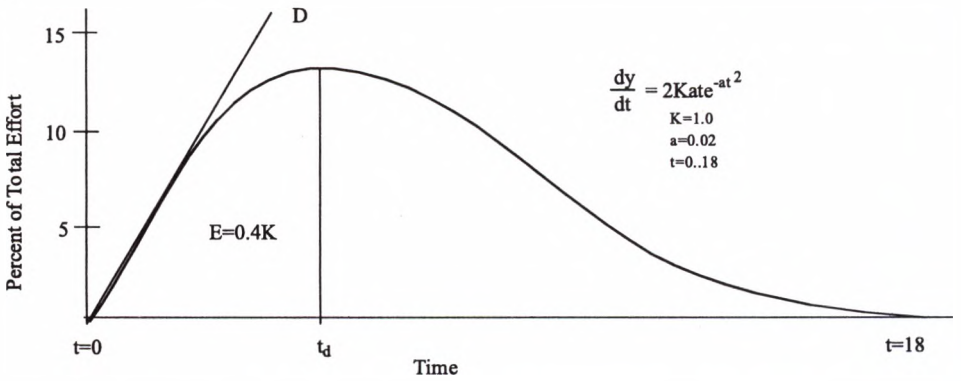

Fig. 3. Rayleigh Model

Putnam used productivity to link the basic Rayleigh manpower distribution model to software size and technology factors. Productivity has been defined as the size of the software product, S, divided by the development effort, E:

$$P = \frac{S}{E} \tag{2}$$

To find E in the Rayleigh model, Putnam made the assumption that the peak staffing level (top of the curve) corresponded to the development time. With this assumption, the area under the curve represented development effort, E. E was found to be approximately 40% of K, the total life-cycle effort which is the total area under the curve.

Putnam observed from project data that the more productive projects had an initial slower staff buildup and the less productive projects had an initial faster staff buildup. He associated the initial staff buildup of a project with the difficulty of the

project, D. The difficulty is represented on the Rayleigh curve as the slope of the curve at time $t = 0$. By taking the derivative of Rayleigh equation and setting $t = 0$, difficulty is defined as:

$$D = \frac{K}{t_d^2} \tag{3}$$

Putnam links the Rayleigh manpower distribution and software development effort. He assumes that there must be a relation between difficulty, D, and productivity, P and he finds this relationship to be:

$$P = \alpha D^{-\frac{2}{3}} \tag{4}$$

By combining the equations (2), (3), (4) and the assumption that $E = 0.4K$, we get the cube root of total life-cycle effort K:

$$\frac{S}{0.4K} = \alpha \left(\frac{K}{t_d^2}\right)^{-\frac{2}{3}} \tag{5}$$

$$S = 0.4\alpha K^{\frac{1}{3}} t_d^{\frac{4}{3}} \tag{6}$$

$$K^{\frac{1}{3}} = \frac{S}{0.4\alpha t_d^{\frac{4}{3}}} \tag{7}$$

Equation (8) introduces a technology factor, C, which is the product of 0.4 and α. The technology factor accounts for differences among projects such as hardware constraints, personnel experience, and programming environment. Putnam suggests using 20 different values for C ranging from 610 to 57,314.

$$K = \frac{S^3}{C^3} \frac{1}{t_d^4} \tag{8}$$

Development effort, E, is found by substituting $E = 0.4K$:

$$E = 0.4 \frac{S^3}{C^3} \frac{1}{t_d^4} \tag{9}$$

Some Rayleigh curve assumptions do not always hold in practice (e.g. flat staffing curves for incremental development; less than t^4 effort savings for long schedule stretchouts). Putnam has developed several model adjustments for these situations. It can be seen from Equation (9) that the effort E increases as the third power of the size S if the schedule remains constant. For a fixed program size, the effort E increases with the inverse of the fourth power of t_d. The optimum development schedule can be calculated from Equation (10) and it agrees with most statistical models used in practice today.

$$t_d = 2.4E^{\frac{1}{3}} \tag{10}$$

Strengths and weaknesses of the theoretical models in general are summarized in Table 3.

Table 3
Strengths and weaknesses of the theoretical models

Strengths	Weaknesses
Objective, repeatable, analyzable formula	Subjective inputs
Efficient, good sensitivity analysis	Assessment of exceptional circumstances
Objectively calibrated to experience	Calibrated to past, not future

3.2. Expertise-Based Techniques

Expertise-based techniques are useful in the absence of quantified, empirical data and are based on prior experience of experts in the field (Tab. 4). Based on their knowledge and understanding of the proposed project, experts arrive at an estimate of the cost/schedule/quality of the software under development. The obvious drawback to this method is that an estimate is only as good as the expert's opinion.

Table 4
Strengths and weaknesses of the expertise-based techniques

Strengths	Weaknesses
Easily incorporates knowledge of differences between past project experiences	However at the same time the output is no better than the experts
Handles assessment of the exceptional circumstances, interactions and is representative	Estimates can be biased due to incomplete experience or human nature
	Subjective estimates that may not be analyzable

Examples of Expertise-based techniques include the Delphi technique, Rule-Based Systems and the Work Breakdown Structure each of which are described in the following subsections.

3.2.1. The Delphi Approach

The Delphi approach was originated at The Rand Corporation in 1948 originally as a way of making predictions about future events – thus its name, recalling the divinations of the ancient Greek oracle. Since then, it has been used as an effective way of getting group consensus.

The aim of the Delphi method is to combine expert opinion and prevent bias due to position, status or dominant personalities. The method involves a panel of experts who each respond separately to a specific enquiry via a series of questionnaires. Their responses are anonymous in the sense that none of the others know who is included in the group or where each response originated from. As initial responses are made separately, new ideas may be introduced by individuals which other members of the panel have not previously considered. Responses obtained from the panel are collated

by a central coordinator and sent back to the respondents in a synthesized form. Then the process is repeated. The aim of each iteration is to gradually produce a consensus amongst the group, or alternatively for responses to become stable, since there is no guarantee that a consensus will result and a range of opinions or responses may be produced instead of a single answer.

Farquhar performed an experiment at Rand Corporation in 1970 where he gave 4 groups the same software specification and asked the groups to estimate the effort needed to develop the product [9]. Two groups used the Delphi technique and two groups had meetings. The groups that had meetings came up with an extremely accurate estimate as compared to the groups that used the Delphi technique. To improve the estimate consensus obtained by the Delphi technique, Boehm and Farquhar formulated an alternative method, the wideband Delphi technique [4].

The wideband Delphi approach can be described with following steps:

1. Coordinator provides Delphi instrument to each of the participants to review.
2. Coordinator conducts a group meeting to discuss related issues.
3. Participants complete the Delphi forms anonymously and return it to the Coordinator.
4. Coordinator feeds back results of participants' responses.
5. Coordinator conducts another group meeting to discuss variances in the participants' responses to achieve a possible consensus.
6. Coordinator asks participants for re-estimates, again anonymously, and steps 4–6 are repeated for as many times as appropriate.

3.2.2. Rule-Based Systems

This technique has been adopted from the Artificial Intelligence domain where a known fact fires up rules which in turn may assert new facts. An expert system is built based on IF-THEN rules for representing the specialist knowledge gained from a human expert (such as an experienced project manager). In this case it is the knowledge about how to estimate a project cost. The expert system applies that knowledge automatically to make decisions.

If the knowledge area is specific enough and well isolated, a reliable cost models based on historical data can be constructed. As a result, the estimation procedure can be automated and the project managers gain the ability to easily and quickly predict the cost. An example rule from a rule-based system developed by Madachy is shown below [15]:

```
IF Required Software Reliability = Very High AND
   Personnel Capability = Low THEN Risk Level = High
```

3.2.3. Work Breakdown Structure

This technique of software estimating involves breaking down the product to be developed into smaller and smaller components until the components can be independently estimated. The estimation can be based on analogy from an existing database of completed components, or can be estimated by experts, or by using the Delphi technique described above. Once all the components have been estimated, a project-level estimate can be derived by rolling-up the estimates.

As discussed in [4], a software Work Breakdown Structure (Fig. 4, 5) consists of two hierarchies, one representing the software product itself, and the other representing the activities needed to build that product. The product hierarchy describes the fundamental structure of the software, showing how the various software components fit into the overall system. The activity hierarchy indicates the activities that may be associated with a given software component.

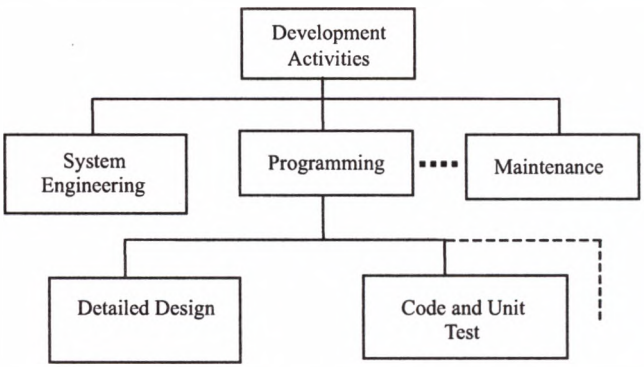

Fig. 4. An Activity Work Breakdown Structure

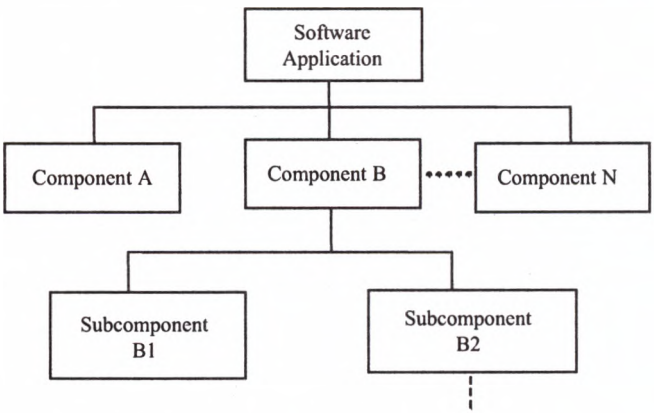

Fig. 5. A Product Work Breakdown Structure

3.3. Learning-Oriented Techniques

Learning-oriented techniques use prior and current knowledge to develop a software estimation model. Neural networks and Analogy estimation are examples of Learning-Oriented Techniques.

3.3.1. Neural Networks

In the last decade, many researchers explored neural networks as an alternative to the other software cost estimation methods. Neural networks are based on the principle of learning from example, no prior information is specified or supplied to the network. Neural networks are characterized in terms of three entities, the neurons, the interconnection structure and the learning algorithm.

Most of the software models developed using neural networks use backpropagation trained feed-forward networks (see Fig. 6). As discussed in [11], these networks are architected using an appropriate layout of neurons. The network is trained with a series of inputs and the correct output from the training data so as to minimize the prediction error. Once the training is complete, and the appropriate weights for the network arcs have been determined, new inputs can be presented to the network to predict the corresponding estimate of the response variable.

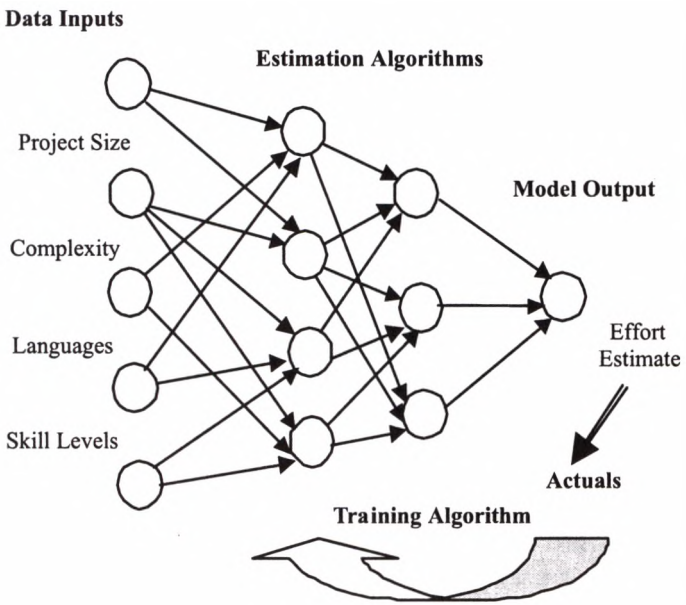

Fig. 6. A Neural Network Estimation Model

Wittig [19] developed a software estimation model using connectionist models (synonymous with neural networks as referred in this section) and derived very high prediction accuracies. Although, Wittig's model has accuracies within 10% of the actuals for its training dataset, the model has not been well-accepted by the software engineering community due to its lack of explanation (Tab. 5).

Table 5
Strengths and weaknesses of the neural networks

Strengths	Weaknesses
Accuracy compares favorably with other methods	Requires large training sets in order to give good predictions
The method is objective and repeatable	Accuracy is sensitive to decisions regarding the net topology
Can be applied when only partial information about project is available	Little explanation value – such models do not help us understand needed software effort

Neural networks operate as "black boxes" and do not provide any information or reasoning about how the outputs are derived. And since software data is not well-behaved it is hard to know whether the well known relationships between parameters are satisfied with the neural network or not. For example, both theory and other data sources agree that if you're developing a software product for future reuse, more effort is required to make the components less dependent on other components.

3.3.2. Analogy Estimation

This method of effort estimation is based on comparison of a planned project with previous projects that have similar characteristics. This model uses experts or stored historical project data to determine the effort required to develop a software product. For a new product it must be determined what subcomponent level is practical for estimation. There must be an estimate of how many components will likely be in the product. Experts compute the high, low, and most likely estimates for effort required based on the differences between the new and previous projects. The method can provide a detailed estimate of effort depending on how deep into the sub-components the analogies are made (Tab. 6).

Table 6
Strengths and weaknesses of the analogy estimation

Strengths	Weaknesses
Based on representative experience	Historical data and experience may be not representative
High accuracy in case of very similar projects	

Case-based reasoning is an enhanced form of estimation by analogy. A database of completed projects is referenced to relate the actual costs to an estimate of the cost of a similar new project. Thus a sophisticated algorithm needs to exist which compares completed projects to the project that needs to be estimated. After the current project is completed, it must be included in the database to facilitate further usage of the case-based reasoning approach. Case-based reasoning can be done either at the project level or at the sub-system level. Case studies represent an inductive process, whereby estimators and planners try to learn useful general lessons and estimation heuristics by extrapolation from specific examples.

3.4. Statistical Models

Statistical models use data to derive the values for model coefficients. Regression analysis is used to establish the relationship between model parameters and software development effort. There are two forms of statistical models: linear and non-linear.

3.4.1. COCOMO II

COCOMO II effort estimation model is based on regression. It consists of three sub-models, each one aiming to offer increased fidelity the further along one is in the project planning and design process (Tab. 7).

Table 7
Strengths and weaknesses of the COCOMO II estimation

Strengths	Weaknesses
Objective and not influenced by politics	Size dependent estimation method
Repeatable, versatile and initially calibrated	Needs to be calibrated to achieve better predicability

The original COCOMO (Constructive Cost Model) model was first published in [4], and reflected the software development practices of the day. In the last two decades, software development techniques changed dramatically, for example the software components became reusable, and new systems can be built using common off-the-shelf software. That is why the authors formulated a new version of the model called COCOMO II, which provides the following three sub-models for estimation of software projects cost:

1. **Application Composition** model involves prototyping efforts to resolve potential high-risk issues such as user interfaces, software/system interaction, performance, or technology maturity. It uses object points for sizing.
2. **Early Design** model involves exploration of alternative software/system architectures and concepts of operation. It involves use of function points for software product sizing and a small number of additional cost drivers.

3. **Post-Architecture** model involves the actual development and maintenance of a software product. It uses source instructions and/or function points for sizing, with modifiers for reuse and software breakage.

In the COCOMO II method the software development effort (in person months) is modelled using the following equation:

$$\text{Effort} = A \times (\text{Size})^E \times \prod_i \text{EM}_i \tag{11}$$

where A is a multiplier that scales the effort according to the specific project conditions, $Size$ is the estimated size of a project in Kilo Source Lines Of Code (KSLOC) or Unadjusted Function Points (UFP), E is an exponential factor that accounts for the relative economies or diseconomies of scale encountered as a software project increases its size, and EM_i are the effort multipliers. The coefficient E (scale exponent) is determined by weighing the predefined scale factors SF_i and summing them via following formula:

$$E = 0.91 + 0.01 \sum_i \text{SF}_i \tag{12}$$

Five scale factors has been defined – precedentedness, development flexibility, architecture/risk resolution, team cohesion and process maturity. The number of the effort multipliers depends on the model and varies from 7 in case of Early Design model to 17 in Post Architecture model. The example effort multipliers are: reliability, complexity, reuse, experience, schedule acceleration and others.

The development time $TDEV$ is derived from the effort according to the following formula:

$$\text{TDEV} = C \times (\text{Effort})^F \tag{13}$$

Latest calibration of the method shows that the multiplier C is equal to 3.67 and the coefficient F is determined is a similar way as the scale exponent:

$$F = 0.28 + 0.002 \sum_i \text{SF}_i \tag{14}$$

When all the factors and multipliers are taken with their nominal values, the equations for effort and schedule are as follows:

$$\text{Effort} = 2.94 \times (\text{Size})^{1.1} \tag{15}$$

$$\text{TDEV} = 3.67 \times (\text{Effort})^{3.18} \tag{16}$$

4. Current Research Areas

The research of new and more accurate size and effort estimation methods led to revising former models and approaches to this problem. New estimation methods include many variants of the Function Point Analysis, Putnam model, application of fuzzy logic, neural networks and multi-agent systems. Examples of these new approaches are described in this section.

4.1. Unified Modelling Language

Many researchers are currently examining the elements of the Unified Modelling Language (UML) in order to find the relationships with the elements of the estimation methods. For example Stutzke [18] proposed a way to use the UML elements to estimate the size in Unadjusted Feature Points. Feature Points method is a refinement to the Function Point Analysis which introduces changes to improve applicability to systems with significant internal processing (e.g., operating systems, communications systems) – this allows accounting for functions not readily perceivable by the user, but essential for proper operation.

Other example in this area is a mapping between UML elements and Full Function Points (another variation of the Function Points Analysis targeted towards realtime system and embedded applications) proposed by [6].

Some areas such as accounting for reuse or research on how productivity depends on the architecture choice, or the development process still need to be investigated. These methods also require validation and assesing of their accuracy.

4.2. Full Function Points

Full Function Points (version 1.0) was proposed in [16] with the aim of offering a functional size measure specifically adapted to realtime software. The field tests have shown that Full Function Points is also suited to measuring the functional size of MIS (management information systems) software. This fact coupled with the feedback received from organizations which have used Full Function Points since version 1.0 was released in 1997, have motivated the authors to improve the method. Many improvements proposed led to the next generation of functional size measurement method – version 2.0 of the COSMICFFP measurement method.

The COSMIC-FPP method is designed to be applicable to software from the domains of application software, real-time software and their hybrids. The method involves applying a set of rules (see [8] for detailed description of the rules) and procedures to a given piece of software as it is perceived from the perspective of its Functional User Requirements. The result of the application of these rules and procedures is a numerical "value of quantity" representing the functional size of the software, from user's perspective.

4.3. Fuzzy Analogy

An extension (fuzzification) of the Analogy Estimation estimation method was proposed in [12]. The new approach, called Fuzzy Analogy, is based on reasoning by analogy, fuzzy logic and linguistic quantifier. Fuzzy Analogy is composed of three steps – identification of similar projects, evaluation of similarity between projects and adaptation. The categorical data from the similar projects, such as factor data of the COCOMO model, are represented by fuzzy sets rather then classical sets.

This methods can handle correctly the imprecision and the uncertainty when describing software project. Fuzzy Analogy is also applicable when the variables are numeric (no uncertainty). First software prototypes and emprical validation of the approach were just started.

4.4. Automation

The automation of the estimation process reduces the measurement costs and speeds the process. There are two main areas of research in the automation of the software functional size measurement process. The first one covers methods based on the source code analysis (retro-engineering). An example framework for automating Function Points counting from source code can be found in [20].

The other one includes methods based on specifications and case-tools. The functional size measure can be automatically generated from designs in UML (see section 4.1) once mapping between the UML elements and the estimation method rules is defined. Formalization of the IFPUG defnition of function points using the formal specifcation language B was proposed in [10]. The goals of the formalization were to provide an objective defnition of function points (which should reduce variance due to interpretation) and to automate function point counts for B specifcations.

5. Summary and Conclusions

This article has presented an overview of a variety of software estimation techniques classifying them in broad categories. The strengths and weaknesses of each of these approaches have been discussed, suggesting in which situation one technique might be more appropriate to use than another. Current research trends and examples of new methods were also presented.

As it can be seen for the article, there is no silver bullet method for software estimation. The ideal size estimation method would define a relatively simple metric directly related to product size, would not depend on chosen construction technology and could be applied starting early in project life-cycle.

The current software size metrics are either simple and construction method dependent or are complex and have limited applicability. Also not all of them are easy or possible to use in early project phases. The comparison of chosen characteristics of the size estimation methods is presented in Table 8.

Since most of the software effort estimation methods take the product size as an input parameter, it is crucial to chose the best possible size estimate in order to obtain stratifying effort predictions. To minimize risk of method inaccuracy at least two independent size estimation methods shall used to derive an average size estimate.

The ideal effort estimation technique would be repeatable and objective, would take into consideration historical project data and could handle various exceptional circumstances that can have impact on development time. Currently no method satisfies all of these criteria.

Table 8
Characteristics of the size estimation methods

Size estimation method name	Complexity of the metric and method	Construction method independent	Suitable for early project phases
Lines Of Code	Low	No	No
Function Points	High	Yes	Yes
Use-case Points	Medium	Yes	Yes
Object Points	Medium	Yes	No

Although new techniques based on rule systems, agents or neural networks were developed, they are not widely used in the real-life projects. They have not gained popularity with the software engineering community either because of limited applicability or poor results and their black-box approach to estimation. Chosen characteristics of widely used effort estimation methods are presented in Table 9.

Table 9
Characteristics of the effort estimation methods

Effort estimation method name	Repeatable	Objective	Historical data used
Putnam Model	Yes	Yes	No
Wide Band Delphi	No	No	Yes
COCOMO II	Yes	Yes	Only by recalibration
Analogy	Yes	No	Yes

The main conclusion we can draw from this article is that the key to arriving at solid estimates is to use a variety of methods and tools and then to investigate the reasons why the estimates obtained using one method might differ significantly from those provided by another. Also during a project, the estimates shall be revised often to help to keep a software project on track.

References

[1] Albrecht A. J.: *Measuring Application Development Productivity.* Proceedings of the Joint SHARE, GUIDE, and IBM Application Development Symposium, Oct. 14–17, 1979

[2] Albrecht A. J., Gaftney J. E.: *Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation.* IEEE Transactions on Software Engineering, vol. 9, No. 2, November 1983

[3] Banker R., Kauffman R., Kumar R.: *An Empirical Test of Object-Based Output Measurement Metrics in a Computer Aided Software Engineering (CASE) Environment.* Journal of Management Information Systems, 1994

[4] Boehm B. W.: *Software Engineering Economics.* Englewood Cliffs, New Jersey, Prentice-Hall 1981

[5] Boehm B. W., Clark B., Horowitz E., Westland C.: *Cost Models for Future Software Life Cycle Processes: COCOMO 2.0.* Annals of Software Engineering Special Volume on Software Process and Product Measurement, Arthur J. D., Henry S. M. (Eds.), Amsterdam, The Netherlands, J.C. Baltzer AG, Science Publishers 1995

[6] Bévo V., Lévesque G., Abran A.: *Application of FFP method from a specification with UML notation: First test and questions raised.* International Workshop on Software Measurement 1999

[7] Conte S., Dunsmore H., Shen V.: *Software Engineering Metrics and Models.* Benjamin/Cummings, Menlo Park, Ca. 1986

[8] Common Software Measurement International Consortium, *COSMIC-FPP Measurement Manual*, version 2.1, 2001

[9] Farquhar J. A.: *A Preliminary Inquiry Into the Software Estimation Process.* RM-6271-PR, The Rand Corporation, 1970

[10] Diab H., Frappier M., St-Denis R.: *Counting Function Points From B Specifications.* International Workshop on Software Measurement, 1999

[11] Gray A. R., MacDonnell S. G.: *A Comparison of Techniques for Developing Predictive Models for Software Metrics.* Information and Software Technology 39, 1997

[12] Idri A., Abran A., Khoshgoftaar T. M.: *Fuzzy Analogy: A New Approach for Software Cost Estimation.* International Workshop on Software Measurement, 2001

[13] International Function Point Users Group, *http://www.ifpug.org*

[14] Kitchenham B.: *Software Development Cost Models.* [in:] R. Rook (Ed.), Software Reliability Handbook, London, U.K., Elsevier 1990

[15] Madachy B.: *Heuristic Risk Assessment Using Cost Factors.* IEEE Software, May/June 1997

[16] Pierre D., Maya M., Abran A., Desharnais J.: *Adapting Function Points to Real Time Software.* IFPUG Conference, Fall 1997

[17] Putnam L.H.: *A General Empirical Solution to the Macro Software Sizing and Estimating Problem.* IEEE Transactions on Software Engineering, July 1978, 345–361

[18] Stutzke R. D.: *Using UML Elements To Estimate Feature Points.* International Workshop on Software Measurement, 1999

[19] Wittig G. E., Finnie G. R.: *Using Artificial Neural Networks and Function Points to Estimate 4GL Software Development Effort.* Australian Journal of Information Systems, 1994

[20] Ho V. T., Abran A.: *A Framework for Automating Function Points Counting from Source Code.* International Workshop on Software Measurement, 1999