Object-Oriented Systems Size Estimation

Abdullah Al-Shishani
Hashemite University, Jordan
Department of Software Engineering
abdullah.asendarz@gmail.com

***Abstract*—Software size estimation is one of the promising fields in software engineering, because cost estimation mostly depends on software size. This paper reviews existing size estimation techniques and present a new approach and tool for object oriented systems size estimation that is more accurate than current ones, taking in to consideration non-functional requirements. The goal is to make software size estimation more accurate in the earlier stages of software development life cycle.**

Keywords—Software size estimation, object oriented.

I. Introduction

Software industry is evolving eagerly, and it is important to estimate the cost, size and effort of a software project accurately. Accurate estimations are important because they affect management decisions, budget and deadlines.

There are many approaches and methods for size estimation, most of which use the following metrics:

- Lines of code (LOC)
- Function points (FP)
- Class points (CP)

in this paper an overview of these metrics is presented, and current approaches and methods for size estimation are discussed.

Most of the effort and cost estimation approaches and methods base their estimation on the LOC of the software, so a wrong estimation of LOC can lead to inaccurate results. The effort estimation is mainly based on the equation[5] :

$$E = A + B * (KLOC)^C \tag{1}$$

where E stands for estimated effort,A, B, and C are constants, and KLOC is the expected number of thousands of lines of code in the final software. It is clear that the equation has a high dependency on LOC estimated, thus a wrong estimation of the LOC can lead to wrong cost and effort estimation. In addition to that, software size may vary depending on the non-functional requirements of the system. Thus one cant estimate the size of a software accurately without knowing the non-functional requirements. In addition to that, each developer has his own programming style, which means even with the same functional and non-functional requirements, software size can vary depending on the developer's style of programming.

Most of the programming languages headed to object oriented methodologies. Which makes current software size measures for the procedural paradigm not suitable to capture specific object-oriented features, such as classes, inheritance, encapsulation, and message passing[3].

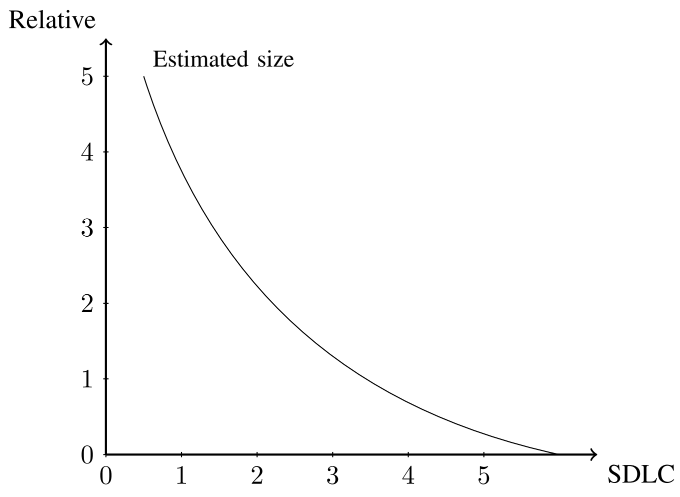

Fig. 1. Software size estimation through software development life cycle

Our method depends on the architecture and design patterns to be used in the system, and taking non-functional requirements in to consideration. One should note that developer's style of programming can effect the software size, however the estimation method uses the default model for both architecture and design patterns being used in the system. To estimate non pattern related code size, class points will be used [3].

In this paper, the metrics and methods of software size estimation are presented and discussed, then a new approach and tool for object oriented systems size estimation is presented.

The rest of the paper is organized as follows: Section 2 describes background; subsection 2.A talks about object oriented size estimation;Subsection 2.B talks about function points; Subsection 2.C talks about class points; Subsection 2.D presents the related work; Section 3 talks about the significance of software size estimation; Section 4 presents our approach for object oriented systems size estimation; Section 5 presents time schedule and Section 6 presents the conclusion.

II. Background

Software size estimation accuracy increases in the later stages of the software development life cycle (fig. 1) That is because it becomes much clear how the actual system is being built and what are the unexpected problems. The goal is to estimate the size of the software in the early stages of software development life cycle accurately.

The models and techniques available today for software size estimation are not yet reliable enough to be consistently used with existing cost estimation models [7].

The most popular sizing technique used is the PERT method where estimation is based on expert judgment. Such estimates are based on the experience of the judgment in projects of similar characteristics[9]. However, these estimations are expected to be underestimated or overestimated. It can be seen in Table I that 7 out of 10 projects were underestimated by experts.The underestimation happens for many reasons[7], including :

- The desire to please the management.
- Incomplete recall of previous experiences.
- Lack of enough knowledge of the particular project being estimated.

based on the for mentioned expert judgment we can clearly see the need for more accurate and clear estimation method.

Most of software estimation techniques use LOC to estimate the size of the software . However, Software size estimation using LOC can be inaccurate, mainly because each developer has his own programming style. Which means a single functionality has many possibilities of LOC. For example if an ordinary if statement in Java is being implemented, we have multiple possibilities :

- Multi-line if else statement .

```
if(condition)
   // condition true
else
   // condition false
```

- Multi-line if statement .

```
if(condition){
   // condition true
   return;
}
// condition false
```

- Single-line if statement .

```
condition ? // condition true : //
    condition false
```

Thus, it is almost impossible to estimate software size using lines of code, which raises a need for more accurate size estimation measure.

System non-functional requirements effect software size as well. In Weinbergs experiment [4], five development teams developed five systems with different sizes for the same functional requirements. Those teams were given variant non-functional requirements. one team was asked to minimize the amount of memory required for the program, another was to optimize program understandability, the third team was to minimize program length, the fourth team was to minimize the development effort, and the last team was to produce the clearest possible output. Based on that experiment it is clear that non-functional requirements can effect the final system size.

TABLE I. Actual and Predicted Sizes

Product	Actual Size	Predicted Size
1	70.919	34,705
2	23.015	32,100
3	34,560	22,000
4	23,000	9,100
5	25,000	12,000
6	52,080	7,300
7	7,650	28,500
8	25,860	8.000
9	16,300	30,600
10	17,410	2,720

A. *Object Oriented Size Estimation*

Most of the widely used programming languages are object oriented, which makes the need for a sizing method for object oriented systems more desirable. In [8][11], two size measures named Size1 and Size2 were used to estimate the size of object oriented systems. Size1 is the number of non-commented lines of code. Size2 is the total count of the number of data attributes and the number of external (or public) local methods in a class. Size2 is defined as :

$$Size2 = NOA + NEM \tag{2}$$

where NOA is the number of attributes of the class and NEM is the number of external methods of the class, i.e., methods which are available through the class interface to members of other classes [6]. Thus, the computed value of Size2 does not take private methods in consideration. [6] provided an extension to the equation to take local methods in consideration. So the size is defined as :

$$Size = NOA + NOM \tag{3}$$

where NOM is the number of local methods (i.e., NOM includes also nonpublic methods).

B. *Function Point*

”Not long ago, the Lines of Code (LOC) count was the only software size measure used for defining product measures”[3]. Using LOC to accurately estimate software size is almost impossible, because same functional requirement can be implemented in many ways with variant number of lines of code. Thus, accurate LOC can be gotten only on late phases of software development life cycle. The need to accurately estimate software size in the early stages of software development life cycle encouraged the development of more size estimation techniques. One such technique is the Function Point method, which was introduced by Albrecht [1] to measure the size of a data-processing system from the end-users point of view. Function point is based on the number of inputs, outputs , inquiries , master files and interfaces.

C. Class Point

Inspired by Function Point, Gennaro Costagliola Et. all[3] introduced Class Point. Class point depends on the number of attributes and methods in a class, which evaluates the complexity level of the class. Total Unadjusted Class Point value (TUCP) is computed using predefined complexity values of the class. Then based on influence degrees related to general system characteristics, the Total Degree of Influence (TDI) is computed. Then the TDI is used to determine the Technical Complexity Factor (TCF) using the following equation:

$$TCF = 0.55 + (0.01 * TDI) \tag{4}$$

Fianlly the value of Class Point (CP) is obtained by multiplying the Total Unadjusted Class Point value by TCF, using the following equation:

$$CP = TUCP * TCF \tag{5}$$

D. Related Work

There are many existing approaches and methods for software size estimation, few are for object oriented systems.

Gennaro Costagliola et al.[3] presented an approach for object oriented systems size estimation. Based on function points, they used Class Points to measure the size of the software. Class Points were based on the complexity of the classes of the system. Class Point depends on the class's methods and attributes. However, non-functional requirements were not considered.

Luiz A. Laranjeira [7] proposed a method for object oriented systems size estimation that takes advantage of a characteristic of object-oriented systems, the natural correspondence between specification and implementation, in order to enable users to come up with better size estimates at early stages of the software development cycle. However, internal structure, the design of the code and non-functional requirements were not considered.

Kirsten Ribu[10] presented a tool and approach for object oriented systems estimation based on use cases of the system. They size the system by measuring the size or complexity of the use cases in the use case model, in order to compute an early estimate of cost and effort. However, non-functional requirements were not considered.

Non of the existing methods, tools and approaches take non-functional requirements in consideration. As non-functional requirements can effect the size of the system, their effect on the size of the software shouldn't be ignored.

III. Significance

In any software project, it is important to deliver accurate cost and time estimations, and most of size and cost estimations are based on the size of the software estimated. Effort estimation depends on the size of the software as well, which can effect management decisions. Thus, it is significant to estimate the size of the software accurately.

IV. Methodology

Just like Class Point[3], our approach depends on the complexity of the class. It is based on the number of attributes and methods of the class (public and private). To predict the classes that the system will consist of, we depend on the possible design patterns to be used. Architecture system design can give a good plus for the estimation as well. For example, if abstract factory pattern will be used in the implementation, it will clearly have an interface, which is the abstract factory itself, and a number of concrete factories. Based on expert prediction, the number of concrete factories can be accurately estimated. For example, building a restaurant management system using Java, a requirement is to paint a floor plan. However, the requirement said that the floor plan will be decorated based on the location of the floor plan in the system (ex. Live, Plan, Template...), with 3 locations in the system. A good design pattern to use here would be abstract factory for painting the floor plan. With three concrete factories to paint the three different locations of the floor plan. An expert in painting using Java can accurately predict the operations needed to paint the floor plan. For example, paint the background, paint the tables on the floor plan, repaint the floor plan ... etc. As the main methods needed to paint the floor plan are defined, the rest is to count these methods. However, a non-functional requirement for the same feature was to to have an exact identical floor plans across all the systems working together with a gap of 30 seconds. Which means every single floor plan needs to be repainted every 30 seconds. Thus, we need to add another method that adds a scheduler to repaint the floor plan every 30 seconds. The final abstract floor plan interface would be like : .

```
/**
 * @author Asendar
 *
 */
public interface AbstractFloorplan {

        /**
         *
         */
        public abstract void
            addBackground();

        /**
         *
         */
        public abstract void addTables();
        /**
         *
         */
        public abstract void paint();

        /**
         *
         */
        public abstract void repaint();

        /**
         *
         */
        public abstract void
            addRepaintTimer();

}
```

As it is clear what the master interface looks like, it is easy to predict how the concrete factories will look like. However, one can not ignore the possibility of adding other methods to one or more of the concrete factories, that do not originally exist in the master interface. However, these added methods are considered as design pattern grime[2] which should not be added to the design pattern at all.

The existence of a non-functional requirement does not necessarily mean more methods or attributes to add. A non-functional requirement can be to make the final source code as abstract as possible and as concrete as needed. Thus, number of methods and attributes inside a class should be minimal. Which means that the developer who will develop the system should make each module as abstract as possible. For example, in Observer design pattern, instead of having 3 methods for firing different types of events, with a non-functional requirement of making the code as abstract as possible, the developer can omit all the methods that fire different types of events, and add a single event throwing method that can do all the functionality required.

One should note that software architectural design and design patterns used to build the software are (when first built) usually identical to the Structural Pattern Specification (SPS)[2]. Thus, when the software is first built, it do not has any methods or attributes that are not specified in the SPS. However, as the software evolves it tends to add more methods and attributes that do not exist in the (SPS).

Based on the predicted system architectural style and design patterns to use, and using class points[3], and taking non-functional requirements in to consideration, more accurate size estimation of a software project can be obtained.

The tool to estimate the size of the software to be implemented has a local storage of the class point equations and widely used design pattern SPSs. The possible design patterns to be used are provided to the tool to retrieve its SPS. And should provide the implementation details for a specific design pattern. For example, if a design pattern to be used is abstract factory, number of concrete factories should be provided, as well as the number of methods in the master abstract factory interface. The tool considers that each concrete factory will be identical to the abstract factory. However, methods or attributes to be added due to a non-functional requirement should be specified in the abstract factory master interface. After specifying all the required data, using class point approach[3] the size of the software to be built is calculated.

TABLE II. RESEARCH TIME LINE

Duration (Weeks)	**Task**
1	Research on Size Estimation
3	Related Work
2	New approach
7	Building the Tool
3	Evaluation

V. TIME SCHEDULE

Presenting our approach and building the tool time line is shown in Table II

VI. CONCLUSION

Accurate estimation of software size is almost impossible in the very early stages of the software development life cycle. And one can not ignore non-functional requirements during size estimation. In addition to that, developer's style of coding can significantly effect the size of a software. However, based on our understanding and after a research based in Jordan with software development organizations, every organization has its own standards for coding style, which means that developer's style of coding can be ignored in software size estimation process.

Design patterns and software architecture style can give a good plus in software size estimation, as each design pattern has a SPS and should be conformant to it's SPS.

Using Class Points and software potential architecture a more accurate size estimation can be achieved.

REFERENCES

[1] A.J. Albrecht. "Measuring Application Development Productivity". In: *Proc. Joint SHARE/GUIDE/IBM Application Development Symp.* (1979), pp. 83–92.

[2] Clemente Izurieta Et. all. "A multiple case study of design pattern decay, grime, and rot in evolving software systems". In: *Software Quality Journal* 21 (2013), 289323.

[3] Gennaro Costagliola Et. all. "Class Point: An Approach for the Size Estimation of Object-Oriented Systems". In: *IEEE Transactions on Software Engineering* 31 (2005), pp. 52–74.

[4] Gerald M. Weinberg Et. all. “Goals and performance in computer programming”. In: *The Journal of the Human Factors and Ergonomics* 16 (1974), pp. 70–77.

[5] Barry W. Boehm. “Software Engineering Economics”. In: Prentice Hall PTR Upper Saddle River, NJ, USA, 1981.

[6] B. Henderson-Sellers. “Object-Oriented Metrics: Measures of Complexity”. In: Prentice-Hall, 1996.

[7] L.A. Laranjeira. “Software Size Estimation of Object-Oriented Systems”. In: *IEEE Transactions on Software Engineering* 16 (1990), pp. 510 –522.

[8] W. Li and S. Henry. “Object-Oriented Metrics that Predict Maintainability”. In: *J. Systems and Software* 23 (1993), pp. 111–122.

[9] R. Pressman. “Software Engineering: A Practitioners Approach”. In: New York: McGraw-Hill, 1982.

[10] Kirsten Ribu. “Estimating Object-Oriented Software Projects with Use Cases”. Master of Science Thesis. University of Oslo, Nov. 2001.

[11] D. Kafura W. Li S. Henry and R. Schulman. “Measuring ObjectOriented Design”. In: 1995.