International Journal of Information Technology and Management
Vol. 17, Issue No. 3, November-2022, ISSN 2249-4510

DOI: https://doi.org/10.29070/qnfwhh75

Estimate the effort required to develop an Object-Oriented Software utilizing class point approach

Manas Prasad Rout [1] , Prof. Sabyasachi Pattnaik [2]

1. Research Scholar, F.M. University, Balasore, Odisha, India ,
2. Professor, F.M.University, Balasore, Odisha, India

Abstract: The time and effort put into creating a piece of software may have a significant impact onwhether or not it is commercially successful. As a second step, we examine the found classes' localmethods and their interplay with the rest of the system to ascertain their complexity. In softwaredevelopment, estimation is the process of establishing the most precise measure of size required to designand test the programmed system, based on the software's needs. For calculating rough estimates of thesize of Objective Oriented OO products, we provide a technique that has resemblance to FP (FunctionPoint) we call it Class Point. Particularly, two metrics like number of methods and number of attributesare used for user objects that include design patterns.

Keywords: effort, software development, object oriented software, class point approach, estimation, complexity, size, design patterns, metrics, user objects

---------------------------X---------------------------

INTRODUCTION

Estimating how much time a software project will take is a constant struggle. As the approaches to be implemented are unknown at this point in the process, the project managers are unable to make firm decisions regarding the next steps to take. Researchers and practitioners alike have come to regard object-oriented technology as the standard approach to creating new software. Encapsulation, inheritance, polymorphism, abstraction, cohesion, and coupling are only a few of the capabilities provided by the OO programming model that help to manage the development process in a methodical manner. The size of a piece of software is a crucial metric. Given that the line of code and the function point (FP) were both used for the procedural programming concept, traditional software estimation techniques like the Constructive Cost Estimation Model (COCOMO) and the function point analysis (FPA) have proven to be unsatisfactory for measuring the cost and effort of all types of software development. Data and procedures are kept separate in a procedurally oriented design, but are brought together in an object-oriented one. For the second time, the FP method to effort estimate is only viable during the coding phase of the software development life cycle. For this reason, an alternative method of estimating labor requirements must be used at the earliest phases of software development.

Researchers and practitioners that employ statistical approaches like fuzzy, machine learning, etc. to draw inferences from the data should keep in mind that the difficulty of learning/estimating relationships from samples is only one aspect of the entire experimental process. As a consequence, it is crucial to think about data collected from prior projects to get excellent outcomes in predicting software size. It has always been crucial to have reliable estimates at hand when making bids or analyzing the financial viability of projects. In order to generate a more precise answer, the class point approach (CPA) is used to estimate the project work, since classes are the fundamental logical unit of an object-oriented system. From this class diagram, we can derive the CPA. As a result, during the software development life cycle (SDLC), the design phase, CPA may also be used to estimate how much work will go into the project. If you want to estimate the size of a class using CPA, you may use one of two metrics: CP1 or CP2. A pair of metrics, the number of external methods (NEM) and the number of services requested (NSR), are used to determine CP1, while a third variable, the number of attributes (NA), is used to determine CP2 (NOA). The number of locally specified public methods (NEM) is a measure of a class's interface size. The NSR may be used as a gauge for how well the various parts of a system interact with one another. Class NOA may be quantified with the use of the NOA. Calculating the technical complexity factor (TCF) for either the FPA or CPA requires looking at the impact of various system-level features. However, CPA disregards non-technical aspects including management efficacy, developer expertise, security, stability, maintainability, and portability. In light of these six non-technical considerations, CPA has been refined in this study to determine how much work must go into creating a software product. Fuzzy logic was used to determine the degree of difficulty for each class before assigning class points. Multi-layer perceptron (ANN), multivariate adaptive regression splines (MRS), projection pursuit regression (PPR), constrained topological mapping (CTM), K nearest neighbor regression (KNN), and radial basis network are just some of the adaptive methods of regression techniques used to improve prediction accuracy (RBFN). In order to back up the conclusion, a comparison of the outcomes from both methods has been supplied.

The next step is to assess the complexity of the identified classes by analyzing their local methods and the ways in which they interact with the rest of the system. We may do this by using suitable metrics for OO classes. The criteria and approaches utilized to finish this stage differ significantly from CP1, the previous one. As a matter of fact, in CP1 the complexity of each class is determined by the number of required services and the number of external methods (NSR). In an OOP system, the interface size for a given class may be roughly estimated using the NEM, which is derived from the number of publics, local methods. NSR is used to determine how closely related components of a system are to one another and to quantify that relationship. Again, this affects a specific subset and is proportional to the quantity of services ordered by the other subsets. In addition to the above metrics, CP2 additionally takes into account the Number of Attributes (NOA) to determine a class's complexity. After figuring out the level of difficulty associated with each class, we can use the data and its structure to assign a relative value to each category. Then, the result for Total Unadjusted Class Points is calculated using a weighted sum (TUCP). Similar to how FPA determines a score, the Class Point is arrived at by adjusting the TUCP by a value based on system-wide global features. In addition to proposing these measures, we also provide the results of a theoretical validation and an empirical validation. In reality, it is commonly accepted that a software measure is only acceptable and effective once it has been verified.

When developing software, estimation is the process of finding the most precise measure of size required to design and test the programmed based on the software's needs. The size estimates are used in the planning, iteration, budgeting, investment analysis, pricing, and bidding processes. Academics and professionals in the software industry have been grappling with the difficulties of software project estimation since at least the 1960s. A substantial amount of literature shows that the time, money, and maintenance impacts of planned size are proportional to their size. Software estimation is essential for effective management of the software development life cycle's multiple phases (SDLC). As the programmed evolves over time, there must be control over the subsequent release. As a result, it's important to undertake some estimation work before really developing the product. Both internal and external factors might be used in the exploration and estimate processes. It is possible to make educated guesses about the size, complexity, and modularity of a software project. When we talk about "external qualities," we're referring to attributes that may be measured by considering the project's surroundings. External attributes include things like reliability, clarity, and ease of upkeep. Because they are highly dependent on context, it is notoriously difficult to quantify exterior features. As a result, research into the intrinsic property of "size" has yielded an estimation model.

LITERATURE REVIEW

TülinErçelebi Ayyildiz et.al (2018) This research examines the relationships between the number of software classes and methods in object-oriented software and the number of unique nouns and verbs in requirements artefacts in the issue domain. With this goal in mind, we assessed 14 completed software development projects at a CMMI Level-3 accredited defense sector business. A linear regression analysis-based model for size estimation is developed on the basis of the high correlation between the measurements. On these 14 software projects, the method's predictive performance is evaluated. As a further observation, it has been shown that the metrics of the issue domain correlate strongly with the amount of work put into the development process. As a result, the metrics of the issue domain are also employed in a linear regression analysis to estimate the effort. The anticipated work utilizing the COSMIC Function Point (CFP) approach to measuring size is also compared with the assessed estimates of work. The results demonstrate that, in comparison to the work calculated via CFP size measurement, the effort estimated using the suggested technique is more accurate.

Mr. RUSHIKESH S. RAUT (2020) Object-oriented programming (OOP) is being more widely used in the real world, which bodes well for the future of the software business and the development of software engineering. The in-depth study of object-oriented programming necessitates familiarity with certain aspects. We have learned the constructors and destructors, as well as the benefits and drawbacks of object-oriented programming, from this work.

Joseph Yoder et.al (2019) There is a large and growing body of knowledge in object-oriented programming and design that has been amassed over the past 50 years. This body of knowledge suggests that lowly coupled, highly cohesive, extensible, comprehensible, and robust software designs are some of the most important qualities in a system. However, this may not be sufficient due to the growing complexity and heterogeneity of modern software. In this article, we'll look at some of the ways in which object-oriented design has proven difficult to implement in practice. The complexity of modern real-world problems requires developers to create not just one but multiple models for the same problem, and we discuss how existing quality assessment techniques for object-oriented design should go beyond high-level metrics. Finally, we recommend a list of issues for researchers and practitioners to work on in the near future. This article is an appeal for more grounded studies in the area of object-oriented software design.

Gurusiddaraj Konded et.al (2019) Object-Oriented Software Engineering's (OOSE) popularity has grown as the software industry and software engineering have progressed, making OOSE a more practical choice for the real world's software-intensive environments. OOSE has come a long way from its early days in software assessment and design, and is now widely regarded as one of the software integrations processes that contribute to a successful product. The OOSE is an effective methodology for software development because it brings together Object-Oriented Analysis (OOA) models, Object-Oriented Design (OOD), and Object-Oriented Programming (OOP). After software analysis and design, OOSE offers the potential of OOP throughout development and production. This article examines the broad concepts and problems that impact commercial software development.

Osman Gazi Yildirim et.al (2021) In order to keep students interested and motivated in programming classes, instructors often employ computer games as a teaching aid. The first author, who is also the course teacher, conducted this participatory action study to improve the quality and efficiency of the Object-Oriented Programming course he teaches. Questionnaires and semi-structured interviews were used in the initial phase of the action study to gather data for issue characterization and solution. After this, a plan of action was developed, and one of its components was a revamped Object-oriented Programming course. The action plan's goals were reflected in the integration of Unity 3D's The Karting Microgame Template into the course, with students utilizing C # to implement additional game components like as a bonus collecting system, scoring, collision mechanics, etc. Afterwards, a plan of action was put into motion. Twenty-nine post-secondary students enrolled at Turkey's Computer Technology Department for the implementation phase during the 2019-2020 spring semester. Object-oriented programming exam results, student and researcher journals, and focus group interviews were used to evaluate the success of this implementation after it had been put into place. This report details the challenges researchers faced, student perspectives on the study's execution, and the researchers' own personal experiences.

RESEARCH AND METHODOLOGY

These design patterns may be thought of as a collection of user objects, and these items are divided into four distinct categories. The 23 design patterns established by Gamma et al.are organized by default into the categories shown in Table 1. These classifications are arbitrary and determined only by the commonality of the underlying components that the design patterns are applied to.

The PDT component stores patterns made up of objects/classes that stand in for real-world things in the system's application domain. Bruegel et al.outlines a centralized database for managing accidents, and they use items like Incident and Field Officer and Emergency Report as examples.

HIT stands for "human interaction type," and it describes things made to satisfy the demands of data visualization and user interface design. Regarding the aforementioned scenario, HIT is the owner of the objects Emergency Report Form and Report Emergency Button.

DMT stands for "data management type," and its associated objects provide storage and retrieval operations. As shown in, one example of a DMT component is the Incident Management subsystem, which includes the classes responsible for executing SQL queries to create and retrieve database entries that represent Incidents.

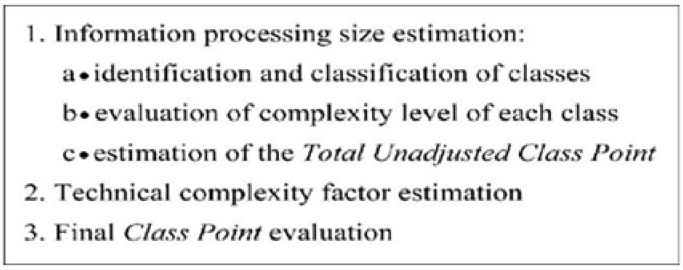
TMT objects are in charge of task definition and management. d. Task management type. In the given example, there are two objects that provide this function: Manage-Emergency Control and Report Emergency Control. Objects that provide data transfer across modules hosted on multiple hosts are likewise part of the task management category. Classes like Message and Connection are common examples of those belonging to this subsystem.

DATA ANALYSIS

The Class Point Approach

Here, we introduce the Class Point method, which can be used to calculate an approximate OO product size at the system level. It was designed by adapting the principles of FP analysis to the OO paradigm and integrating established OO metrics in an appropriate way. The Class Point technique was initially introduced in, and from there, it was improved, resulting in the development of two measures, CP1 and CP2, which are described here. The purpose of the suggested technique is, in fact, to provide a strategy that lets us fine-tune estimations all through the development and make use of fresh data as it becomes accessible. When first conceptualized, the CP1 measure's purpose was to provide developers with a rough size estimate as work began. A calculation like that

Table 1: The Class Point Method


1. Information processing size estimation:
   a• identification and classification of classes
   b• evaluation of complexity level of each class
   c• estimation of the *Total Unadjusted Class Point*
2. Technical complexity factor estimation
3. Final *Class Point* evaluation

Applying CP2, which needs data that are often accessible later in the development phase for calculation, may provide extra depth. When we were first conceptualizing the Class Point approach as an extension of FP, we opted against utilizing direct mappings of FP logical files and transactions to classes and methods. There is a clear delineation between data and operations in the procedural paradigm. In contrast, the notion of an operation is intrinsic to the idea of a class in the OO paradigm because of the close relationship between the operations and the data they are used to control. Therefore, the Class Point approach zeroes emphasis on classes, because they are the units to be counted and weighted according to their complexity levels, just as with functions in FPA. Each class's complexity is calculated using data about its local methods, its interaction with other parts of the system, and, if accessible, its properties. Although FPA's categorization of data function types and transaction function types was developed with business applications in mind, the Class Point measurements' classification of classes is neutral with respect to the nature of the applications themselves. Our technique is somewhat similar to POP counting; however, we differentiate between classes themselves rather than methods. A further difference from POPs is that data are also included in the Class Point measurements, and the information required to complete the counting is often accessible at the design phase rather than being substituted by estimated values.

The Class Point methodology, used for estimating CP1 and CP2, is described in Section 3.1. The steps that were taken to arrive at the suggested measures are laid forth in Section 3.2.

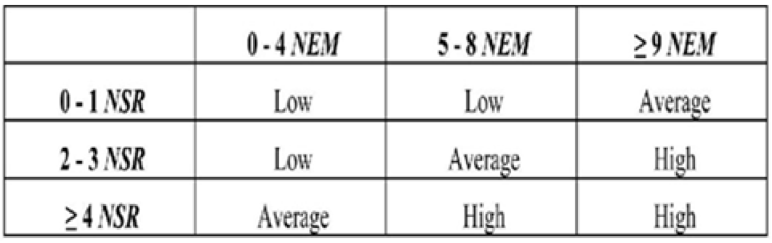
The Class Point Method:

Class Point size estimate may be broken down into three distinct stages that mirror those of the FP methodology. In Table, we see a rough outline of this procedure.

Identification and Classification of User Classes

The Class Points system begins with an analysis of the design requirements to determine the various classes. There are four main categories of system components.

Table 2: Evaluating the Complexity Level of a Class for CP1



	0 - 4 *NEM*	5 - 8 *NEM*	≥ 9 *NEM*
0 - 1 *NSR*	Low	Low	Average
2 - 3 *NSR*	Low	Average	High
≥ 4 *NSR*	Average	High	High

Categories issues according on whether they are of the problem domain, human interaction, data management, or task management varieties (TMT). Classes representing entities in the system's domain of use are kept in the PDT component. Accordingly, in a decentralised information system for accident management, such as the one described in, Incident, Field Officer, and Emergency Report are all examples of PDT classes. Training programmed of the HIT kind are fashioned to meet the need for graphical representations of data and human-computer interaction. Emergency Report Form and the Report Emergency Button are two examples of such categories that relate to the first. DMT covers classes that allow for the archiving and retrieval of information. The Incident Management subsystem of DMT is responsible for running SQL queries to save and retrieve Incident records from the database. Lastly, TMT classes are developed for task management and, as such, define and govern activities like Manage-Emergency Control and Report Emergency Control. This section also integrates communications between different computer-based subsystems and manages connections with external systems. Common examples of classes in this subsystem are Message and Connection. This classification was introduced by Coad and Nicola to make it simpler to modify and reuse classes. Regardless of the application domain or architectural style, such class types may be recognized in any OO system.

Evaluation of a Class Complexity Level

To determine a class's complexity level, the Class Point method considers how each class component behaves. The method used to establish this difficulty level is where CP1 and CP2 diverge. In particular, CP1 takes into consideration the total number of external methods and the total number of services requested, whereas CP2 additionally makes use of the total number of characteristics.

The size of a class's interface may be estimated by counting the number of locally specified public methods, or the Number of External Methods (NEM). The NSR quantifies the degree to which various parts of the system are dependent on one another. Again, this only applies to a single class and is based on how many services across classes you need. Both metrics may be found in the design documentation. Indeed, activities that define any OO design process include identifying the classes with their characteristics and methods and building interaction (or cooperation) diagrams to demonstrate which external services are required for a class to complete the desired duties. For their part, Li and Henry suggested a comparable metric to NSR. To quantify the degree to which one class depends on another, a metric called Message Passing Coupling (MPC) was first proposed in. NSR counting is simpler than MPC, therefore it may be done in the early stages of design and analysis.

As shown in Table 2, CP1 class complexity is determined by the sum of the NSR and NEM range values. Thereafter, classes are given weights that reflect the complexity and nature of the data they contain. Classes with more than nine NEM and an NSR greater than 2 are considered to be of high complexity. Section 3.2 will explain the thinking behind Table 2.

One other factor in determining a class's complexity is the number of characteristics, which is factored into the computation of CP2. Since NOA is an important metric, CP2 takes it into account as a separate variable, which leads to Tables 3a, 3b, and 3c. Each table is indexed on NSR, NEM, and NOA, and is associated with a certain NSR range.

Estimating the Total Unadjusted Class Point

When we know how complicated each class is, we can calculate the value of the Total Unadjusted Class Points (TUCP). TUCP table completion is required for this purpose (see Table 4). For each row and column in the table, the number of classes and their relative complexity are shown.

In this way, the TUCP is determined by adding together and weighting the four parts of the application.:

$$TUCP = \sum_{i=1}^{4} \sum_{j=1}^{3} w_{ij} \times x_{ij},$$

where xij is the number of classes of components of type I (problem domain, human interaction, etc.) and complexity level j (low, medium, or high), and win represents the weighting value for type I and complexity level j.

Technical Complexity Factor Estimation

TCF is calculated by weighting 18 generic system attributes according to their impact on the application (from 0 to 5).

Table 3: CP2 Evaluation of the Complexity Level of a Class

0 - 2 *NSR*	0 - 5 *NOA*	6 - 9 *NOA*	≥ 10 *NOA*
0 - 4 *NEM*	Low	Low	Average
5 - 8 *NEM*	Low	Average	High
≥ 9 *NEM*	Average	High	High

(a)

3 - 4 *NSR*	0 - 4 *NOA*	5 - 8 *NOA*	≥ 9 *NOA*
0 - 3 *NEM*	Low	Low	Average
4 - 7 *NEM*	Low	Average	High
≥ 8 *NEM*	Average	High	High

(b)

≥ 5 *NSR*	0 - 3 *NOA*	4 - 7 *NOA*	≥ 8 *NOA*
0 - 2 *NEM*	Low	Low	Average
3 - 6 *NEM*	Low	Average	High
≥ 7 *NEM*	Average	High	High

(c)

Table 4: valuating the TUCP

System Component Type	Description	Complexity			
		Low	Average	High	Total
PDT	Problem Domain	...*3=...	...*6=...	...*10=...	...
HIT	Human Interaction	...*4=...	...*7=...	...*12=...	...
DMT	Data Management	...*5=...	...*8=...	...*13=...	...
TMT	Task Management	...*4=...	...*6=...	...*9=...	...
TUCP		***Total Unadjusted Class Point***			

as seen through the eyes of a designer. The estimated levels of effect are shown in Fig. 1's Processing Complexity table. Total Degree of Influence (TDI) is calculated using the following formula, and it is comprised of influence degrees connected to such broad system features.

TCF ¼ 0:55 þ ð0:01

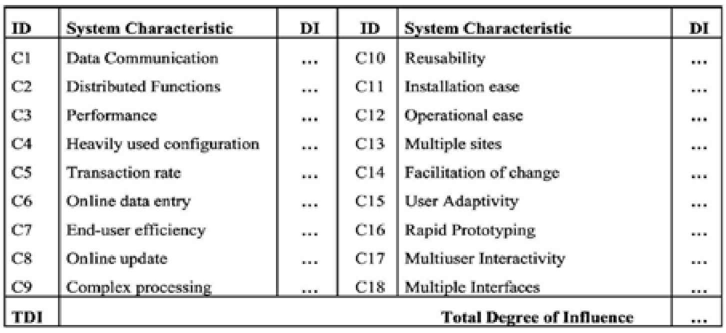
TDIÞ:

By multiplying the sum of the unadjusted class points by the standard deviation, we may get the adjusted class point (CP).

CP ¼ TUCP

TCF:

In relation to the unadjusted CP count, the adjustment factor may have an effect of -45% (corresponding to a TDI of 0) or +45% (equivalent to all degrees being set to 5). In the Appendix, you'll find a Class Points computation sheet (Fig. 6). Importantly, the Technical Complexity Factor is calculated by considering not only the factors taken into consideration by the FP approach, but also the following factors, which were developed with object-oriented systems in mind: 15 - User Adaptation Prototyping, Rapid 17 User Interaction Eighteen. Several Connections In light of this fact, we've decided to add the aforementioned features to object-orientation.



ID	System Characteristic	DI	ID	System Characteristic	DI
C1	Data Communication	...	C10	Reusability	...
C2	Distributed Functions	...	C11	Installation ease	...
C3	Performance	...	C12	Operational ease	...
C4	Heavily used configuration	...	C13	Multiple sites	...
C5	Transaction rate	...	C14	Facilitation of change	...
C6	Online data entry	...	C15	User Adaptivity	...
C7	End-user efficiency	...	C16	Rapid Prototyping	...
C8	Online update	...	C17	Multiuser Interactivity	...
C9	Complex processing	...	C18	Multiple Interfaces	...
TDI				Total Degree of Influence	...

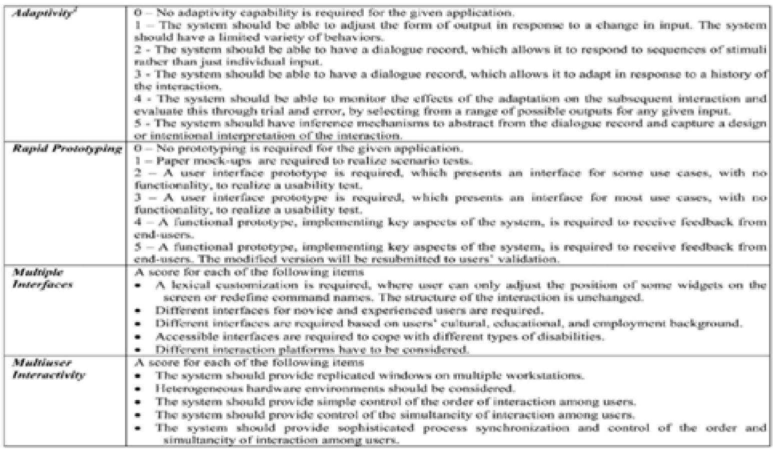
Figure 1: The Processing Complexity table

provides the best theoretical framework for designing useful GUIs (GUIs). It is generally accepted that objects provide a more accurate representation of the UI components and that direct manipulation of those items is more easily supported [53]. Also, the ability to reuse software components and an OO user interface go hand in hand with an OO-designed and -implemented programmed. The development of these cutting-edge interfaces typically consumes the majority of software project resources due to the costs of their design, implementation, debugging, and modification, despite the availability of powerful and useful object-oriented tools for building interactive graphical applications. As a result, it is crucial to utilize a reliable size prediction in order to properly plan the creation of the interactive software project. Table 5 provides some rules of thumb that may be used to gauge how much of an impact each of the added variables really has.

The Class Point Definition

Professional Guidance This section explains how we arrived at the current definition of the Class Point metrics. The opinions of a panel of 12 engineers versed in the creation of interactive OO applications support this claim. Based on these observations, we may describe a class's complexity by looking at its methods, characteristics, and interactions with other classes. This was further bolstered by an examination of Bunge's ontology. A lot of focus in recent years has been on using this kind of ontology in the context of object-orientation [21]. In reality, Bunge's qualities underpin the concept of object and are preoccupied with the significance and delimitation of worldly representations. Concepts and objects, given names as distinct entities, each with its own set of defining characteristics, make up these models. As a result, we may define an object as a concrete, singular entity whose set of discrete qualities can be taken into account as a whole. Assigned methods and instance variables represent an object's attributes in object-oriented programming. The object and all of its characteristics are considered to be a representation of the application domain. For our purposes, we have paid close heed to Bunge's definition of complexity as the "numerosity of its composition." This idea maps into the complexity of an object class in an OO framework, which may be thought of as the sum of all the characteristics that can be associated with it. For their WMC metric, Chidambaram and Kemmerer took use of the existing notion of object complexity. However, in their view, methods are what really add to a class's complexity, as outlined by Bunge's definition. Because they felt characteristics were less time-consuming than techniques, they opted not to include them in the complexity assessment. However, from our perspective, two classes with the same number of methods and differing quantities of attributes cannot be equally time intensive. However, in the CP1 measure, properties of a class are not taken into account for evaluating the complexity level since they may not be accessible in early design. When available, the number of characteristics is used in the assessment of the complexity level in the CP2 measure.

Table 5: Guidelines to Determine the Influence Degree of the Introduced Factors



Adaptivity[1]	0 – No adaptivity capability is required for the given application. 1 – The system should be able to adjust the form of output in response to a change in input. The system should have a limited variety of behaviors. 2 - The system should be able to have a dialogue record, which allows it to respond to sequences of stimuli rather than just individual input. 3 - The system should be able to have a dialogue record, which allows it to adapt in response to a history of the interaction. 4 - The system should be able to monitor the effects of the adaptation on the subsequent interaction and evaluate this through trial and error, by selecting from a range of possible outputs for any given input. 5 - The system should have inference mechanisms to abstract from the dialogue record and capture a design or intentional interpretation of the interaction.
Rapid Prototyping	0 – No prototyping is required for the given application. 1 – Paper mock-ups are required to realize scenario tests. 2 – A user interface prototype is required, which presents an interface for some use cases, with no functionality, to realize a usability test. 3 – A user interface prototype is required, which presents an interface for most use cases, with no functionality, to realize a usability test. 4 – A functional prototype, implementing key aspects of the system, is required to receive feedback from end-users. 5 – A functional prototype, implementing key aspects of the system, is required to receive feedback from end-users. The modified version will be resubmitted to users' validation.
Multiple Interfaces	A score for each of the following items • A lexical customization is required, where user can only adjust the position of some widgets on the screen or redefine command names. The structure of the interaction is unchanged. • Different interfaces for novice and experienced users are required. • Different interfaces are required based on users' cultural, educational, and employment background. • Accessible interfaces are required to cope with different types of disabilities. • Different interaction platforms have to be considered.
Multiuser Interactivity	A score for each of the following items • The system should provide replicated windows on multiple workstations. • Heterogeneous hardware environments should be considered. • The system should provide simple control of the order of interaction among users. • The system should provide control of the simultancity of interaction among users. • The system should provide sophisticated process synchronization and control of the order and simultancity of interaction among users.

Insights from a proposed taxonomy of adaptable systems have informed the adaptability criteria presented in. The definition of the Technical Complexity Factor and the allocation of weights used in calculating the Total Unadjusted Class Point value (TUCP) have been the subject of much debate among OO engineers.

The engineers are given a list of component types and asked to assess the relative importance of each of the three degrees of complexity for each. We first provided the engineers with four sets of weight triplets to get things rolling. Such baseline values were arrived at by giving various classes within each kind a weight depending on the amount of effort data they provided. In order to begin with low, medium, and high weights for each kind, we first averaged the given weights and then averaged all the weights below the resultant number and all the weights above. The values were examined by engineers, who then compared and debated their own revisions until an agreement was achieved on the weights shown in Table 4.

For the Technical Complexity Factor, we advocated for the inclusion of User Adaptivity, Rapid Prototyping, and Multiple Interfaces in addition to the original 14 components of Function Point analysis for interactive applications. Indeed, the user-centered approach that should be followed in the GUI design process necessitates that a great deal of development work be committed to achieving a high degree of usability of the interface in order to match user's wants and preferences. As a result, prototyping is generally acknowledged as crucial for GUI development. Users should be able to test the prototype and provide suggestions on how to improve the final product. Adding adaptability features for the user and creating other views of the same programmed might also need more work. This may be necessary to provide the user the freedom to choose the mode of interaction that is most appropriate for their specific goals, background, and skill level. Engineers were asked for their feedback on the potential variables. They validated the hypothesis that these elements may significantly alter the time required to create an interactive application. Multiuser Interaction was also discovered throughout the interviews with the individuals. The idea for such a suggestion came from the realization that, in order to allow users to concurrently interact with a shared user interface, it is necessary to take into account a number of distinct factors during the development of a multiuser interface. Such activities include multi-user event processing, window replication, process synchronization, and more.

CONCLUSIONS

Many software development effort/cost models use it as a metric for foretelling how much time and money will be needed to complete a given piece of software. Although FPA can only be used to procedural business systems, many academics believe that the FP method can be successfully applied to other types of systems with little tweaking. Here, we go through the final methodology, the meta-methodology that informed its development, and the methodology criteria that were used to validate it. The size estimates are used in the planning, iteration, budgeting, investment analysis, pricing, and bidding processes. The Gamma Group's definition of grouping for the items that make up the 23 design patterns presents a significant issue for any software projects when attempting to estimate how much time and effort will be required to complete the product. It has always been crucial to have reliable estimates at hand when making bids or analyzing the financial viability of projects. The goal of the suggested strategy is to provide us a way to improve our estimations as the project progresses and take use of newly available data.

References

1. TülinErçelebiAyyildiz et.al (2018) Size and Effort Estimation Based on Problem Domain Measures for Object-Oriented Software

2. Mr. Rushikesh S. Raut (2020) Research Paper on Object-Oriented Programming (OOP) e-ISSN: 2395-0056 p-ISSN: 2395-0072

3. Joseph Yoder et.al (2019) Current Challenges in Practical Object-Oriented Software Design

4. GurusiddarajKonded et.al (2019) A Survey Paper on Object Oriented Software Engineering ISSN 2321 0613

5. Osman Gazi Yildirim et.al (2021)An Action Research Study on the Development of Object-Oriented Programming Course

6. Amit Verma, Navdeep Kaur Gill, "Analysis of Watermarking Techniques", International Journal of Computer Science and Technology (IJCST), Vol. 7, Issue 1, pp. 153-156, Jan- March 2016.

7. Kaur, l., kaur, n., ummat, a., kaur, j., &kaur, n. (2016). research paper on object-oriented software engineering. international journal of computer science and technology, 36-38.

8. Kalinga, E. A. (2018). Learning by doing in teaching and learning Object-Oriented Analysis and Design approach to software development. Proceedings of the 12th International Multi-Conference on Society, Cybernetics and Informatics (IMSCI 2018).

9. Clarke, P. J., &Pierantonio, A. (2018) Teaching modeling: A software perspective. Computer Science Education, 28(1), 1-4. doi:10.1080/08993408.2018.1486535

10. Attane, P., &Kanjug, I. (2020, November). A Study of Learner's Mental Model and Motivation Using Constructivism Online Learning Environment to Promote Programming in Rural School. In International

Conference on Innovative Technologies and Learning (pp. 361-366). Springer, Cham.

11. Avcı, Ü. &Ersoy, H. (2018). Bilgisayar Programlama Derslerinde Öğrenme Motivasyonu Ölçeğinin Türkçe Uyarlaması: Geçerlilikve Güvenilirlik Çalışması. Journal of Higher Education & Science/YüksekögretimveBilimDergisi, 8(1).

12. Balla, T., &Király, S. (2020). A Discussion of Developing a Programming Education Portal. Central-EuropeanJournal of New Technologies in Research, Education and Practice, 1-14

13. Zainal Abidin, N. H., Arsad, R., Muslim, N., & Masrom, S. (2020). Computer game application for JAVA programming language learning. Mathematical Sciences and Informatics Journal (MIJ), 1(1), 77-89.

14. Zhu, J., Alderfer, K., Furqan, A., Nebolsky, J., Char, B., Smith, B., ... &Ontañón, S. (2019, August). Programming in-game space: how to represent parallel programming concepts in an educational game. In Proceedings of the 14th International Conference on the Foundations of Digital Games (pp. 1-10).

15. Wong, Y. S., Hayati, M. Y. M., & Tan, W. H. (2016, September). A propriety game-based learning game as learning tool to learn object-oriented programming paradigm. In Joint International Conference on Serious Games (pp. 42-54). Springer, Cham.