A Software Code Measures Based on Requirement Engineering Documents

Dr. Zainab Mohammed Hussein
Al-Mansour University College
Software Engineering Department
zainab_hussain2012@yahoo.com

Dahlia Jasim Mohammed
Iraqi Commission for Computers and Informatics
Informatics Institute for Postgraduate Studies
dahliajanabi@gmail.com

Abstract

Research shows that Metrics are used by the software industry to estimate the software before creating it to impact the quality of the decision making earlier at the requirement stage, quantify the development, operation, and maintenance of the software.

In this paper a software code measures could be anticipated by the requirement measures gathered from the requirement engineering documents. Ten case studies have been analyzed, data-driven approach have been used to propose two mathematical models, one for requirement metrics which is the Requirement Model (RM) and another one for code metrics which is the Code Model (CM) based on the gathered and analyzed data from the tested systems. The results proved that the RM and CM values are approximately equal based on the relative error measurement and thus a method is gained for computing the code metrics in advanced phases of the software life-cycle and hence time and effort are saved, the cost is decreased, and low wastage is achieved.

Keywords: *Requirement Engineering Documents, Requirement Metrics, Code Metrics.*

1. Introduction

Measurement is the process of empirical, objective, assignment of numbers to properties of objects or events of the real world in such a way as to describe them. Formally, measurement is defined as a mapping from the empirical world to the formal, relational world. Consequently, a measure is the number or symbol assigned to an entity by this mapping in order to characterize an attribute [Kan04]. So metrics act as indicators that provide a quantitative feed-back to software developers about various aspects of the software and pinpoint problem areas in their systems [KKB08].

In an object-oriented environment, requirements are modeled as use-cases and they are implemented as methods of various classes defined in the class diagram and used in the behavioral diagrams. It is necessary to ensure that each and every requirement is addressed as use-case and every event of the use-cases are implemented as methods of classes used in the behavioral diagrams in a consistent manner [KKB08].

Estimation is the intelligent anticipation of the quantum of work that needs to be performed and the resources (human resources, monetary resources, equipment resources, and time resources) required to perform the work at a future date, in a defined environment, using specified methods. Software estimation is assuming more importance as a natural consequence of increased outsourcing of software development work. When any work is outsourced, it is necessary to come to an agreement with the supplier of the price to be paid for the assigned work. In an outsourcing scenario, software estimation is needed for the following reasons [Che09]:

1. To set a budget for the assignment.
2. To evaluate proposals received from different vendors for software development.
3. To reach an agreement with the selected supplier on the size of the software to be developed and the fee for developing the agreed-upon size of the software.

In this paper, a software code measures based on requirement engineering documents can be estimated by using a proposed software measurement tool that provide two models: the proposed

CM (Code Model) and the proposed RM (Requirement Model). These two models are based on the selected code and requirement metrics which are used to prove that it can be measure the code metrics earlier in the requirement phase of the software life cycle using the requirement documents.

2. Related Work

Ashish Sharma and D.S Kushawaha in 2010 [AK0501] proposed a complexity measure based on requirement engineering document, the major issue in development of quality software is precise estimation. Further this estimation depends upon the degree of intricacy inherent in the software i.e. complexity. This paper attempts to empirically demonstrate the proposed complexity which is based on IEEE Requirement Engineering document.

Ashish Sharma and Dharmender Singh Kushwaha in 2010 [AK0910] demonstrated empirically that the complexity of the code can be determined based on its IEEE software requirement specification document. Considering the shortcoming of code-based approaches, their proposed approach is able to compute the complexity of yet-to-be-written software immediately after freezing the requirement in the Software development Lifecycle (SDLC) process.

Luigi Lavazza and Gabriella Robiolo in 2010 [LR10] introduced the evaluation of complexity in Functional size measurement based on a UML Approach, this paper show that measurement-oriented UML modeling can support the measurement of both functional size and functional complexity from UML models.

Kenneth Lind and Rogardt Heldal in 2010 [KR10] explained the reasons behind the strong correlation between Functional Size Measure and Code Size Measure to obtain accurate code size estimation results.

Ashish Sharma and Dharmender Singh Kushwaha in 2011 [SK11] proposed a test metric for the estimation of the software testing effort using IEEE-Software Requirement Specification (SRS) document in order to avoid budget overshoot, schedule escalation etc., at very early stage of software development. Further the effort required to develop or test the software will also depend on the complexity of the proposed software.

Ashish Sharma and Dharmender Singh Kushwaha in 2012 [SK12] present a systematic and an integrated approach for the estimation of software development and testing effort on the basis of improved requirement based complexity (IRBC) of the proposed software.

3. Requirement Engineering

Requirements engineering is a major software engineering action that begins during the communication activity and continues into the modeling activity. Requirements engineering builds a bridge to design and construction. The bridge could begin at the feet of the project stakeholders (e.g., managers, customers, end users), where business need is defined, user scenarios are described, functions and features are delineated, and project constraints are identified. Requirements engineering provides the appropriate mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification, and managing the requirements as they are transformed into an operational system.

Requirement engineering document is a specification for a particular software product, program or set of program that performs some certain functions for a specific environment [SK0510]. The software requirements document (sometimes called the Software Requirements Specification or SRS) is an official statement of what the system developers should implement. It should include both the user requirements for a system and a detailed specification of the system requirements. Sometimes, the user and system requirement are integrated into a single description. In other cases, the user requirements are defined in an introduction to the system requirements specification. If there are a large number of requirements, the detailed system requirements may be presented as a separate document [Som11].

4. Unified Modeling Language

The UML is a general-purpose visual modeling language that is used to specify, visualize construct and document the artifacts of a software system. The emergence of UML as an industry standard for modeling system has encouraged the use of automated software tools that facilitate the development process form analysis through coding. It provides several diagram types that can be used to view and model the software system from different perspectives and/or at different levels of abstraction [NT05]. As UML is not a methodology it is left to the user to follow whatever processes they deem appropriate in order to generate the designs described by the diagrams. UML does not constrain this; it merely allows those designs to be expressed in an easy to use, but precise, graphical notation [Ken09].

4.1. UML Class Diagram

A class diagram is an abstraction for all the possible object diagrams that can be describes the types of the objects in the system and the relationships that exist between them [BS04]. A class is represented as a box with the name of the class inside, the name should always be singular and start with a capital letter. Optionally, the class diagram may also show the attributes and operations contained in each class. Figure (1) illustrates how a class can be drawn at several different levels of detail[LL05].

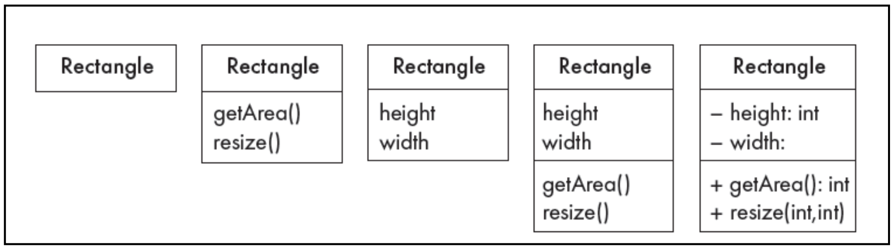


Figure (1) The Rectangle class at several levels of details

4.2. UML Sequence Diagram

The sequence diagrams describe how the system works over a period of time. Sequence diagrams are ‘dynamic’ rather than ‘static’ representation of the system. They show the sequence of method invocations within and between objects over a period of time. They are useful for understanding how objects collaborate in a particular scenario as shown in the example in the Figure (2) [Ken09]. There are three objects in this scenario. Time runs from top to bottom, and the vertical dashed lines (lifelines) indicate the objects ‘continued existence through time. The logic of a scenario often depends on selection (if) and iteration (loops). There is notation (interaction frames) which allow ifs and loops to be represented in sequence diagrams however these tend to make the diagrams cluttered. Sequence diagrams are generally best used for illustrating particular cases, with the full refinement reserved for implementation code [Ken09].

4.3. Use Case Diagram

The use case diagram shows the relationship between actors and use cases. Use case diagram explains about how the system is going to interact with the outside environment. The components are: actor, use case, relationship, and package. Actor is someone or something that is interacting with the system. Use cases are nothing but scenarios of the system. If the abstraction level of the diagram is more enough as an estimation of the function point can be derived from it. Different use cases must be consistent and describe the behavior of the system [RRR12]. An actor symbol is rendered as a stick man, and a use-case as an ellipse. An interaction between an actor and a use-case is presented as an unadorned line between the two as shown in Figure (3) [BS04].

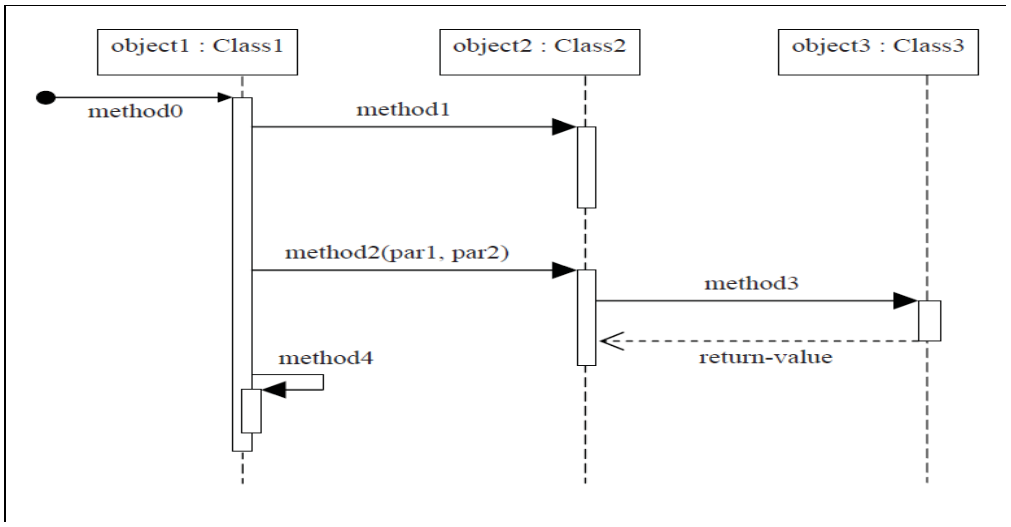


Figure (2) a sequence diagram example

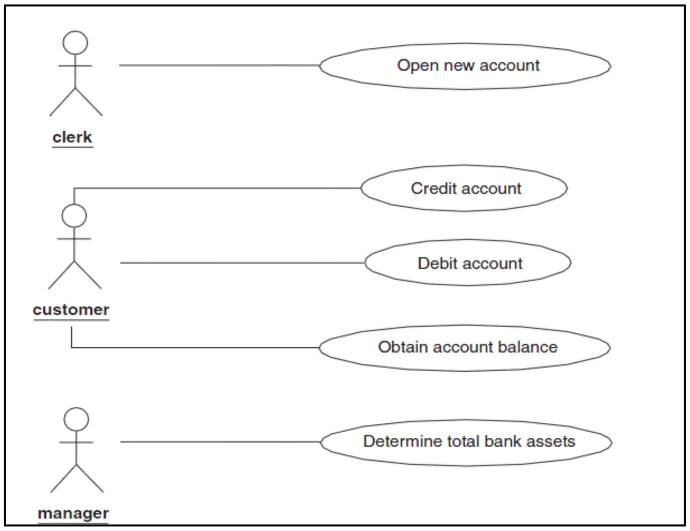


Figure (3) Community Bank Use Cases

5. Requirement Engineering Measurements

Technical work in software engineering begins with the creation of the requirements model. It is at this stage that requirements are derived and a foundation for design is established. Therefore, product metrics that provide insight into the quality of the analysis model are desirable. Although relatively few analysis and specification metrics have appeared in the literature, it is possible to adapt metrics that are often used for project estimation and apply them in this context. These metrics examine the requirements model with the intent of predicting the “size” of the resultant system. Size is sometimes (but not always) an indicator of design complexity and is almost always an indicator of increased coding, integration, and testing effort [Pre10]. Some of these metrics have been selected:

5.1 Use Case Points (UCP) metric

The use case points (UCP) method of software estimation was developed by Gustav Karner of Objectory System (later Rational Software and now IBM) in 1993 [Kar93]. It was developed for those software developers using Unified Modeling Language (UML) and Rational Unified Process (RUP) for software engineering and development. Each use case is comprised of actors. Each actor is classified into one of three levels based on complexity as shown in table (1) and each use case falls into one of three classes based on complexity. Each of these three classes categorizes use cases in one of two ways, either by number of transactions or by number of classes as shown in table (2) [Che09].

Table (1) Weighted actor

Complexity	Definition	Weight
Simple	A system interface	1
Average	A protocol-driven interface	2
Complex	A GUI	3

Table (2) Weighted use case

Complexity	Definition	Weight
Simple	1 to 3 transactions or 5 or fewer classes in the software	5
Average	4 to 7 transactions or 6 to 10 classes in the software	10
Complex	8 or more transactions or 11 or more classes in the software	15

To get the Unadjusted Use Case Points (UUCP), the weights of the actors and the use cases are summed together [Che09]:

$$\mathbf{UUCP = \sum_{i=1}^{6} n_i * W_i} \text{ .... } \mathbf{(1)}$$

where n_i is the number of items of variety i. W_i is the weight of variety i. i=1 to 6 according to the total types of the weighted use cases and actors. If there is no information about the implementation project environment and the environment we can use the UUCP for the estimation. Otherwise the UUCP would have to be adjusted to get a better estimation. Now the UUCP are adjusted with two types of factors: Technical complexity factor (TCF) and Environment complexity factor (ECF). Each of these factors shown in tables (3) and (4) are rated from 0 to 5, where 0 means the factor is irrelevant and 5 means the factor is most important. Each factor has a predetermined weight. The final value for each of the factors is the value obtained by multiplying the assigned value by its predetermined weight. Computation of the technical factor (TF) is shown in table (3) [Che09]:

$$\mathbf{TF = \sum_{i=1}^{13} T_i * W_i} \text{.. } \mathbf{(2)}$$

where T_i is the factors contributing to complexity of variety i, i=1 to 13 according to the total number of factors, (0.6) is a constant.

$$\mathbf{TCF = 0.6 + TF/100} \text{.. } \mathbf{(3)}$$

Table (4) shows the computation of the environmental factor (EF) such that [Che09]:

$$\mathbf{EF = \sum_{i=1}^{8} E_i * W_i} \text{.. } \mathbf{(4)}$$

$$\mathbf{ECF = 1.4 + (-0.03 * EF)} \text{.. } \mathbf{(5)}$$

where E_i is the factor contributing to efficiency of variety i, W_i is the weight of variety i, (1.4) and (-0.03) are constants. Now the use case points (UCP) is counted using the UUCP the unadjusted use case points, TCF the technical factor, and the ECF the environmental factor [Che09]:

$$\mathbf{UCP = UUCP * TCF * ECF} \text{ .... } \mathbf{(6)}$$

Table (3) Factors contributing to complexity

T_i	Factors Contributing to Complexity	W_i
T_1	Distributed systems	2
T_2	Response time	1
T_3	End user efficiency	1
T_4	Complex internal processing	1
T_5	Reusable code	1
T_6	Easy to install	0.5
T_7	Easy to use	0.5
T_8	Portable	2
T_9	Easy to change	1
T_{10}	Concurrent	1
T_{11}	Security features	1
T_{12}	Access for third parties	1
T_{13}	Special training	1

Table (4) Factors contributing to efficiency

E_i	Factors contributing to efficiency	W_i
E_1	Familiarity with project model used	1.5
E_2	Application Experience	0.5
E_3	Object-oriented experience of the team	1
E_4	Lead analyst capability	0.5
E_5	Motivation of the team	1
E_6	Stability of requirements	2
E_7	Part-time staff	-1
E_8	Difficult programming language	-1

5.2. Object Points (OP) metric

The object points (OP) method is used in COCOMOII (Constructive Cost Model) [Coc] as the size measure for a proposed software product. With the OP method, the software artifacts (screens, reports, and 3GL components) are enumerated. Each of these objects is rated on a complexity level as simple, medium, or difficult. The rules for classifying screens into different levels of complexity are shown in table (5). "Server" indicates tables on the server. "Client" indicates tables on the client machine. 3GL components have only one level of complexity, and that is the "difficult" level. These objects are given a weight that depends on the complexity level, as shown in table (6). Then all the objects are enumerated and summarized in a table, as shown in table (7) [Che09].

Table (5) Complexity levels for screens

	Number and source of data tables		
Number of views contained	Total <4 (1 server,1 or 2 clients)	Total between 4 and 8 (2 or 3 servers, 3 to 5 clients)	Total > 8 (>3 servers,> 5 clients)
2 or less	Simple	Simple	Medium
3 to 7	Simple	Medium	Difficult
8 or more	Medium	Difficult	Difficult

Table (6) Complexity levels for reports

	Number and source of data tables

Number of sections contained	Total <4 (1 server,1 or 2 clients)	Total between 4 and 8 (2 or 3 servers, 3 to 5 clients)	Total > 8 (>3 servers, > 5 clients)
1	**Simple**	**Simple**	**Medium**
2 or 3	**Simple**	**Medium**	**Difficult**
4 or more	**Medium**	**Difficult**	**Difficult**

Table (7) Weights for objects

	Weight based on complexity level		
Object	**Simple**	**Medium**	**Difficult**
Screen	**1**	**2**	**3**
Report	**2**	**5**	**8**
3GL components	**NA**	**NA**	**10**

5.3. The COSMIC – FFP (FSM) Method

COSMIC [Cos09] stands for the (Common Software Measurement International Consortium), FFP stands for (Full Function Point) and FSM is the (Functional Size Measure). The purpose of the COSMIC method is to provide a standardized method of measuring a functional size of software from the functional domains commonly referred to as 'business application' software and 'real-time' software.

This method is based on two phases. The COSMIC-FFP mapping phase takes as input the specification of Functional User Requirements (FUR). This specification may be at different levels of abstraction which generates the identification of different software layers as a result of a functional partition of the system. Later the boundary is identified, which is defined as a conceptual interface between the software under study and its users. The collection of FURs can be decomposed into a set of functional processes. Each functional process is a unique, cohesive and independently executable set of data movements, with data groups being defined as a distinct, non empty, non ordered and non redundant set of data attributes. The measurement method does not require identifying the data attributes. These might be identified if a sub-unit of measure is required. The measurement phase begins with the identification of the data movements of each functional process. A data movement, moves one or more data attributes than belong to one data group. The four valid types of data movement are: entry, read, write and exit [FAP12].

- An entry moves a data group from a user across the boundary into the functional process where it is required.
- A read moves a data group from persistent storage within reach of the functional process that requires it.
- A write moves a data group from inside a functional process to persistent storage.
- An exit moves a data group from a functional process across the boundary to the user that requires it.

The objective of this phase is to produce a quantitative value based on the measurement principle, which is established in COSMIC-FFP. The measurement function is applied to each instance of a data movement by assigning a numerical quantity, 1 Cfsu (Cosmic Functional Size Unit). Finally, the application of the aggregation function continues if the data movements of all the functional processes have been measured. This way, the functional size of a functional process is the sum of the functional sizes of individual data movements. Finally, the functional size of a software layer is defined as the sum of the functional sizes of its respective functional processes.

The measurement phase aims to produce a quantitative value that represents the software functional size. In order to do this, the data movement types are identified. Then, the measurement function is applied and finally, the respective aggregation functions are found [FAP12].

Step 1.-Identification of the data movements:

The four types of data movements (entry, read, write and exit) are identified basically in the respective message types defined in the Sequence Diagrams of the OO-Method Requirements Analysis Process (signal, query, service and connect). The relationship between the concepts of an entry data movement type and a signal message with input value is trivial. Nevertheless, it is necessary to indicate whether the data entry involves attributes of different data groups. Thus the rules to identify one entry for each different data group are [FAP12]:

Rule 1: "Accept each message labeled with the stereotype <<signal>> and with the input value as an ENTRY data movement". For the identification of READ data movements, we consider all the movements that recover attributes values pertaining to the same stored data group. The messages with the stereotype <<query>> represent data movements since they imply read the state of objects.

Rule2: "Accept each message labeled with the stereotype <<query>>as a READ data movement". The condition of a message represents a READ data movement because before execution, it implies recovering the value of the attributes involved in the condition in order to evaluate it.

Rule 3: "Accept each condition for some message type as a READ data movement". In the specification of a use case, it is possible to associate a precondition. It indicates a condition that must be satisfied before also the execution of the use case. In accordance with Rule 3, a precondition is also a READ data movement as defined in the following rule:

Rule 4: "Accept each precondition defined in the specification of a use case as a READ data movement". The following rule is not specifically associated to a sequence diagram but rather to the entire system since it is defined as a class property. This rule is considered as a complementary rule to the identification of read data movements and should be evaluated after the execution of any service in the quoted class.

Rule 5: "Accept each integrity constraint as a READ data movement". A service message allows us to create, destroy or update the state of objects. This message type implies a write data movement, since there is a change of state in these persistent objects.

Rule 6: "Accept each messages labeled with the stereotypes <<service/new>>, <<service/destroy>> or <<service/update>> as a WRITE data movement". Since a message with the stereotype <<connect>> implies the creation or destruction of a link between the objects of the respective classes, this type of message is also considered as a write data movement.

Rule 7: "Accept all message labeled with the stereotype <<connect>>as a WRITE data movement". The relationship between an exit data movement and a signal message with output value is trivial. Both concepts imply moving a data group from a functional process across the boundary to the user that requires it.

Therefore, the proposed rule is as follows:

Rule 8: "Accept each message labeled with the stereotype <<signal>> and with the output value as an EXIT data movement".

Step 2.-Applying the measurement function:

This step consists of applying the COSMIC-FFP measurement function to each data movement identified in each functional process (use case). In the equation described below (7), each instance of a data movement identified (entry, read, write and exit) receives a numerical size of 1 Cfsu (Cosmic Functional Size Unit) [FAP12].

$\mathbf{f(x) = 1Cfsu}$ **.. (7)**

Step 3.-Aggregation function at the functional process level:

This step consists of adding the results of the measurement function applied to all the data movements identified in each functional process. The aggregation function at this level (use case) is as follows [FAP12]:

Rule 9: "The functional size of a use case is equal to the sum of all data movements identified". However, two additional rules are defined due to the relationships that appear between use cases. In order to measure the functional size of a use case extended by one or more secondary use cases, the aggregation function is explained in the following rule:

Rule 10: "The functional size of a base use case extended by another secondary use case set is equal to the sum of the functional sub-processes identified in each secondary use case plus the functional sub-processes of the base use case". In a similar way, in order to a relationship include the functional size of a base use case is explained in the following rule:

Rule 11: "The functional size of a base use case that includes other secondary use cases is equal to the sum of the functional sub-processes identified in each included use case plus the functional sub-processes of the base use case". These two rules are expressed in the equation [FAP12]:

$$\mathbf{Size}\ (\mathbf{Base\_Use\ Case}) = \sum_{\mathbf{i=1}}^{\mathbf{n}} (\mathbf{Secondary\_Use\ Case_i}) + \mathbf{Size_P}(\mathbf{Base\_Use\ Case}) \quad \ldots\ldots\ldots\ldots\ldots \mathbf{(8)}$$

where i is the total number of use cases, p is the sub process.

Step 4.-Aggregation function at the software layer level:

This step consists of adding the results of the measurement function applied to all primary use case identified as functional processes in the software system delimited by the boundary. The secondary use cases are not considered in this step because they are not externals interactions. Therefore, the rule is as follows:

Rule 12: "The functional size of a software layer is equal to the sum of the functional sizes of all the primary use case (functional processes)". This rule is expressed in the following equation [FAP12]:

$$\mathbf{Size\ Layer1} = \sum_{\mathbf{i=1}}^{\mathbf{n}} \mathbf{Size}\ (\mathbf{Primary\_Use\ Case_i}) \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \mathbf{(9)}$$

where i is the total number of the primary use cases

6. Code Metrics

The code metrics are used to measure the software product (source code), at the end of the development stage (at the testing process), during design, and/or at the end of each deliverable. There are many object oriented code metrics, some measure the class, the project, or the properties of the object oriented (OO) concepts.

6.1. Chidamber and Kemerer (CK) metrics

Metric set proposed by Chidamber and Kemere [CK94], contains six object oriented design metrics. Three metrics have been selected, these metrics are used to measure software size and/or complexity, and they are listed as follows:

A. Weighted Methods per Class (WMC)

WMC relates directly to Bunge's definition of complexity of a thing, since methods are properties of object classes and complexity is determined by the cardinality of its set of properties. The number of methods is, therefore, a measure of class definition as well as being attributes of a class, since attributes correspond to properties.

Consider a Class, with methods M1, …, Mn that are defined in the class. Let $c_1, \ldots, c_n$ be the complexity of the methods. Then [Jam06]:

$$\mathbf{WMC} = \sum_{i=1}^{n} c_i \ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \ (10)$$

If all method complexities are considered to be unity, then WMC= n, the number of methods. This metric is used to measure the complexity and size of each class in the project with the following viewpoints [Jam06]:
The number of methods and the complexity of methods involved is a predictor of how much time and effort is required to develop and maintain the class.
The larger the number of methods in a class the greater the potential impact on children, since children will inherit all the methods defined in the class.
Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse.

B. Depth of Inheritance Tree (DIT)

DIT relates to Bunge's notion of the scope of properties. DIT is a measure of how many ancestor classes can potentially affect this class. Depth of inheritance of the class is the DIT metric for the class. In cases involving multiple inheritances, the DIT will be the maximum length form the node to the root of the tree. This metric can be used to measure the complexity and size [Jam06].

6.2. Lorenz and Kidd (LK) Class Size (CS)

A class size can be measured in a number of different ways. This set of metrics is proposed by Lorenz and Kidd [LK94]. These metrics deal with quantifying an individual class, they are summed together to measure the class size (CS) [JS04]:

A. Number of Class Methods in a Class

The number of methods available to the class and not its instance affects the size of the class. The number should generally be relatively small compared to the number of instance methods. The number of class methods can indicate the amount of commonalty being handled for all instances. It can also indicate poor design if service better handled by individual instances are handled by the class itself. As indication that this is occurring is an abundance of conditional logic based on data values [JS04].

B. Number of Class Variables in a Class

Class variables are localized globals, providing common objects to all the instances of a class. There are usually a relatively low number of class variables compared to instance variables. The class variables are often used to provide customizable constant values that are used to affect all the instances behavior. They might coordinate information across all instances, such as the determination of a unique value for a transaction number [JS04].

7. Relative Error Measurement

Error measurement is a potential source of non-sampling bias in many surveys. It occurs when the information to be obtained on one or more variables in the study is miss-measured. This happens, for example, as a result of an imprecise or inaccurate data collection instrument, complexities inherent to the variable being measured, and difficulties to the respondent inform the true response properly. This source of error is potentially a concern for the survey users because, if unaccounted for, it could affect the quality of the data collected and, as a possible consequence, distort the inferences for the parameters of interest [SS12].

Approximation error is the discrepancy between an exact value and some approximation to it. An approximation error can occur because the measurement of the data is not precise due to the instruments or approximations are used instead of the real data. One error measurement is selected, that is the Relative error which accounts for the relative size of error. The relative error is given by the difference between an experimentally determined or approximated value v_1 and the accepted value v_2 divided by the accepted value and multiplied by 100% [GL96].

$$\textbf{Relative error} = \frac{|v_1 - v_2|}{v_2} \times 100\% \ldots\ldots\ldots\ldots\ldots\ldots \textbf{(11)}$$

8. Building Estimation Models

A model can be built and used to estimate or predict the parameter of interest by using the metric values that can be measured. Such models are generally process-specific as the property of interest depends on the process. That is, if the process changes, the value of the property will change even of the input values (i.e., the metrics that have been measured) are the same. A model for the software process (or a part of it) can be represented as [Jal97]:

$$\mathbf{y = f(x_1, x_2, \ldots, x_n)} \ldots\ldots\ldots\ldots\ldots\ldots \textbf{(12)}$$

The dependent variable y is the metric of interest (e.g., the total effort, reliability, etc.). x_1, x_2, …, x_n are independent variables that typically represent some metric values that can be measured when this model is to be applied. The function f is really the model itself that specifies how y depends on these independent variables for the process. A model may be theoretical or data-driven. In a theoretical model, the relationship between the dependent and independent variables is determined by some existing relationships that are known. Such models are independent of data. Data-driven models are generally the result of statistical analysis of the data collected about the process from previous projects. In these models, one hypothesizes some model, whose actual parameters are then determined through the analysis of data. Many of the process models used in project management are data-driven. Collecting data for building such models is the major reason for the termination analysis phase of the management process [Jal97].

9. The Proposed Software Measurement Tool

The proposed software measurement tool is used to measure code metrics based on requirement engineering document by compute the proposed Requirement Model (RM) and the proposed Code Model (CM). This software measurement tool analyzes the requirement documents and the code of any tested software system to collect, analyzes, and save the required data that is needed to compute the selected metrics.

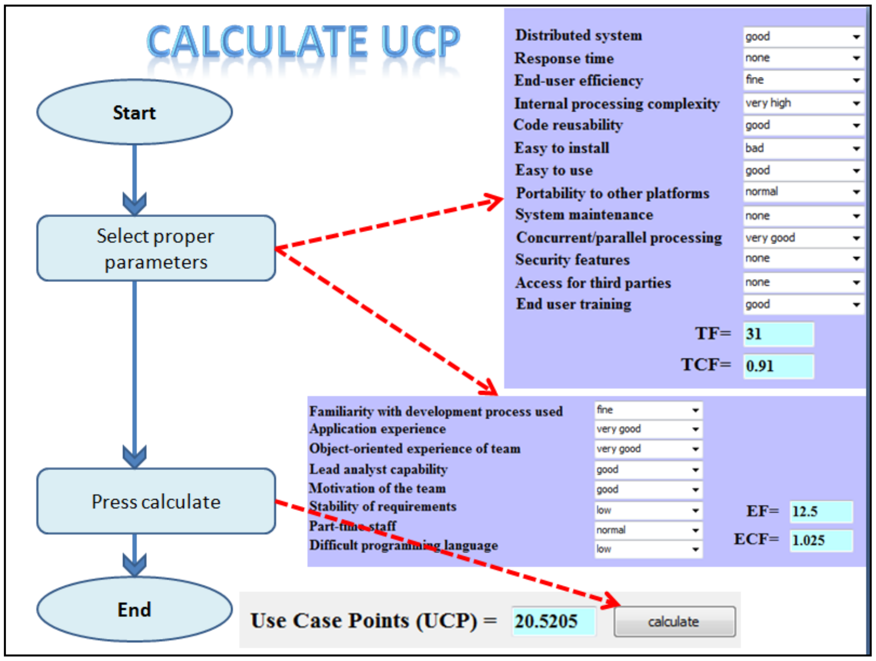
The proposed software measurement tool is capable of creating requirement documents for any software system by using the proposed UML tool where the required data needed to compute the selected requirement metrics are collected in the data-base automatically once the UML diagrams have been created, and then the proposed RM is computed automatically. The code metrics and the proposed CM are computed automatically just by selecting the project folder. The CM and the code metrics are used for comparison and prove purpose.

9.1 Computing the Requirement Metrics

There exist a number of requirement metrics, but some of them are selected because they concentrate on the object oriented size and complexity. The inputs of selected requirement metrics are: the requirement document (required system features from the customer and developer perspectives) and the UML diagrams created by using the UML Building tool provided by the proposed software measurement tool. The requirement metrics are computed automatically by the proposed system. One of the case studies is selected to demonstrate the computing process which is the (Central Repository) case study.

9.1.1. Use Case Points (UCP) Metric

The complexity of the use-case diagrams is classified according to the number of use-cases for each use-case diagram stored in the data-base. The actor's complexity is classified according to the actor type: a system interface, protocol driven, or GUI. Here the UUCP is computed automatically once the diagrams are created. Figure (4) describes the way of specifying the scales for the technical and the environmental factors from the GUI of the proposed software measurement tool. The scale starts from zero to five based on the customer desired features and the developer perspective of the system under construction. The scale are (none:0, very low:1, low:2, normal:3, high:4, very high:5) or (none:0, bad:1, fine:2, good:3, verygood:4, excellent:5) according to the factors. Then the Use Case Points (UCP) is computed automatically based on equation (6).



Distributed system	good
Response time	none
End-user efficiency	fine
Internal processing complexity	very high
Code reusability	good
Easy to install	bad
Easy to use	good
Portability to other platforms	normal
System maintenance	none
Concurrent/parallel processing	very good
Security features	none
Access for third parties	none
End user training	good
TF=	31
TCF=	0.91

Familiarity with development process used	fine
Application experience	very good
Object-oriented experience of team	very good
Lead analyst capability	good
Motivation of the team	good
Stability of requirements	low
Part-time staff	normal
Difficult programming language	low
EF=	12.5
ECF=	1.025

Use Case Points (UCP) = 20.5205

Figure (4) Calculate UCP Flowchart

a. Computing UUCP: The central repository case study has two use-case diagrams according to two actors (administrator and user) who communicate with the system via a GUI interface as shown in figures (5) and (6) respectively, such that the actors are considered complex, each one would be on weight 3. The use-case diagram in figure (5) has five transactions (use-cases), that makes it average of weight 10, and the second use-case diagram in figure (6) has one transaction (use-cases), that would make it simple of weight 5, and according to equation (1), **UUCP = (2×3) + (1×10) + (1×5) =21.**

b. **b. Computing the Technical Factor:** The Technical Factor (TF) is computed by multiplying the W_i by a scale that is specified according to the tested system functionality which is determined by the software developer to estimate the system at the requirement stage, such that for the Central Repository case study, the W_i and a scale for each factor are specified as shown in table (8) along with the reasons behind selecting each scale. By using this table, the TF and TCF are computed according to equations (2) and (3) as follows:

TF= (2×0) + (1×4) + (1×3) + (1×3) + (1×4) + (0.5×3) + (0.5×4) + (2×3) + (1×4) + (1×2) + (1×4) + (1×2) + (1×4) = 39.5

TCF = 0.6 + 39.5/100 = 0.995

c. Computing the Environmental Factor: The Environmental Factor (EF) is computed by multiplying the W_i for each factor by a scale that is specified according to the tested system functionality provided by the software developer to estimate the system at the requirement stage, such that for the Central Repository case study, the W_i and a scale for each factor can be selected as shown in table (9) and from this table, the EF and ECF are computed according to equations (4) and (5).

$$EF = (1.5\times3) + (0.5\times4) + (1\times5) + (0.5\times3) + (1\times3) + (2\times2) + (-1\times3) + (-1\times3) = 14$$

$$ECF = (1.4) + (-0.03\times14) = 0.98$$

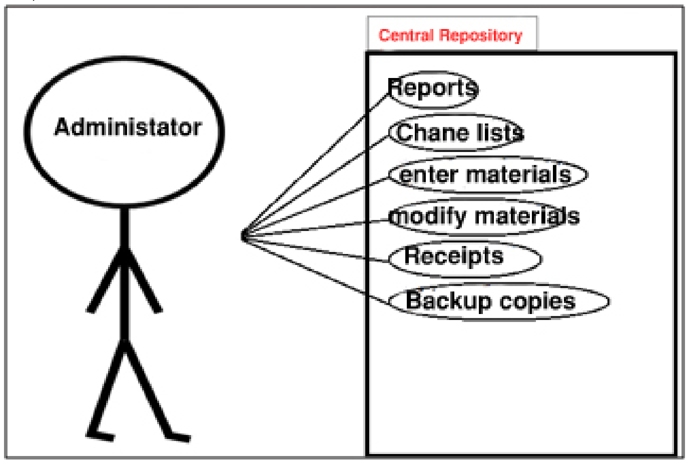


Figure (5) Use case diagram (Administrator responsibilities)

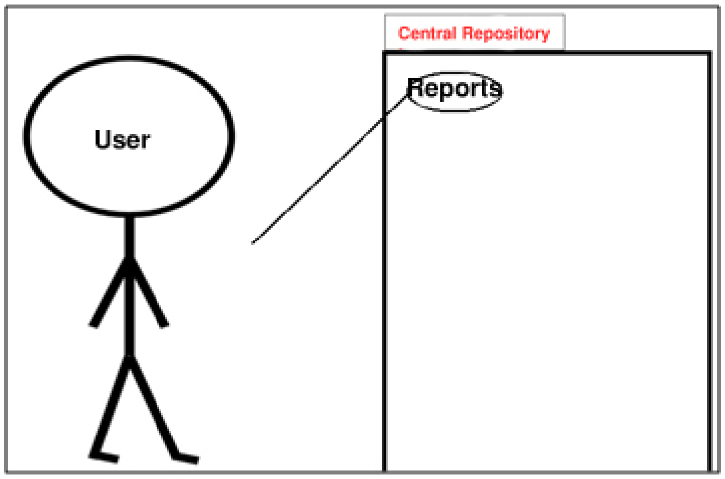


Figure (6) Use-case diagram (User responsibilities)

Table (8) TCF computation for the (Central Repository) case study

T_i	Factors Contributing to Complexity	W_i	Scales	Reasons
T_1	Distributed systems	2	0	Installed on a stand-alone PC
T_2	Response time	1	4	The data-base is stored on the same PC
T_3	End user efficiency	1	3	The user experience with computer systems is average
T_4	Complex internal processing	1	3	The system functions complexity levels are average
T_5	Reusable code	1	4	The programmer used an object-oriented approach
T_6	Easy to install	0.5	3	External data-base connection must be installed first before install the system
T_7	Easy to use	0.5	4	User friendly GUI

T_8	**Portable**	**2**	**3**	**Can be installed on one platform (windows) with different versi**
T_9	**Easy to change**	**1**	**4**	**Programmed with an OO approach that makes it easy to change.**
T_{10}	**Concurrent**	**1**	**2**	**It's a stand-alone PC application not a network application**
T_{11}	**Security features**	**1**	**4**	**The system is protected by a hardware device (USB flash)**
T_{12}	**Access for third parties**	**1**	**2**	**It's a stand-alone PC application**
T_{13}	**Special training**	**1**	**4**	**Staff training is required**

Table (9) ECF computation for the (Central Repository) case study

E_i	**Factors contributing to efficiency**	**W_i**	**Scales**	**Reasons**
E_1	**Familiarity with project model used**	**1.5**	**3**	**New requirement for software development**
E_2	**Application Experience**	**0.5**	**4**	**Based on the software developer experience**
E_3	**Object-oriented experience of the team**	**1**	**5**	**Based on the software developers team experience**
E_4	**Lead analyst capability**	**0.5**	**3**	**Based on the experience of the system analyst**
E_5	**Motivation of the team**	**1**	**3**	**Depends on the payment**
E_6	**Stability of requirements**	**2**	**2**	**Customer changing demands**
E_7	**Part-time staff**	**-1**	**3**	**The team worked at the same period of time**
E_8	**Difficult programming language**	**-1**	**3**	**Depends on the programming language difficulty**

d. Computing the Use Case Points: The UCP (Use Case Points) is computed according to equation (6) by multiplying the UUCP, TCF, and ECF together as follows:
UCP= 21 × 0.995 × 0.98 = 20.4771

9.1.2. Object Points (OP) Metric

The Proposed Snap shooting the User Interface tool is used to count the OP metric automatically once the snap-shooting process is finished along with selecting the type of each screenshot (screen, report, and 3GL components) and the number and source of data tables. The Central Repository case study is chosen to demonstrate the OP metric, this system is installed on one PC; this would specify the complexity level for the screens and reports as explained below:

a. **Specifying Complexity Levels for Screens:** The complexity level for the screens is determined according to tables (5) and (7) by specifying the number and source of data tables. The Central Repository system is listed in (3 to 7) number of views contained, and the system contains nine screens and their complexity level is as follows:

Three simple screens; their weight equals (1) hence the number and source of data tables (total >4). Figure (7) shows one of the simple screens for the Central Repository case study from the proposed software measurement tool GUI. Three medium screens; their weight equals (2) hence the number and source of data tables (total between 4 and 8). Figure (8) shows one of the medium screens for the Central Repository case study from the GUI of the proposed software measurement tool. Three difficult screens; their weight equals (3) hence the number and source of data table (total>8). Figure (9) shows one of the difficult screens for the Central Repository case study from the GUI of the proposed software tool.

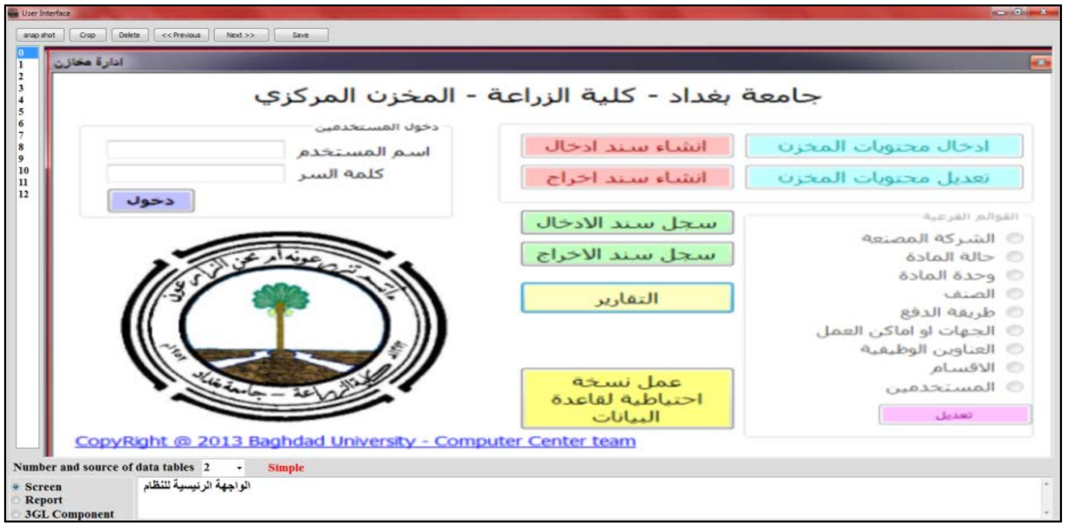
b. Specifying Complexity Levels for Reports : The case study contains three difficult reports, their weight equals (8) hence the number of sections contained is (2 – 3) and the number and source of data tables (total>8). The complexity level for the reports is specified according to tables (6) and (7). Figure (10) shows one of the reports for the Central Repository case study from the proposed software measurement tool GUI.

c. Specifying the 3GL Components: The 3GL components are special screens that are used by the system administrators to control the data architecture of the system, the data bases, and the interfaces. The complexity level of the 3GL components is considered difficult always. There is only one 3GL component such that the complexity level is difficult. Figure (11) shows the 3GL component for the Central Repository case study.

d. Compute OP: Finally, the OP is calculated by classifying each object according to the complexity level, then multiplying the total number of each object by its complexity level, and then sum the total for all the objects and their complexities as illustrated in table (10).

Table (10) the OP count for the Central Repository Case Study

Object	Simple	Medium	Difficult	Total
Screens	3 (screens) ×1	4 (screens) ×2	2 (screens) ×3	17
Reports	0 (reports) ×2	0 (reports) ×5	3 (reports) ×8	24
3GL components			1 (3GL) ×10	10
Total OP	51			

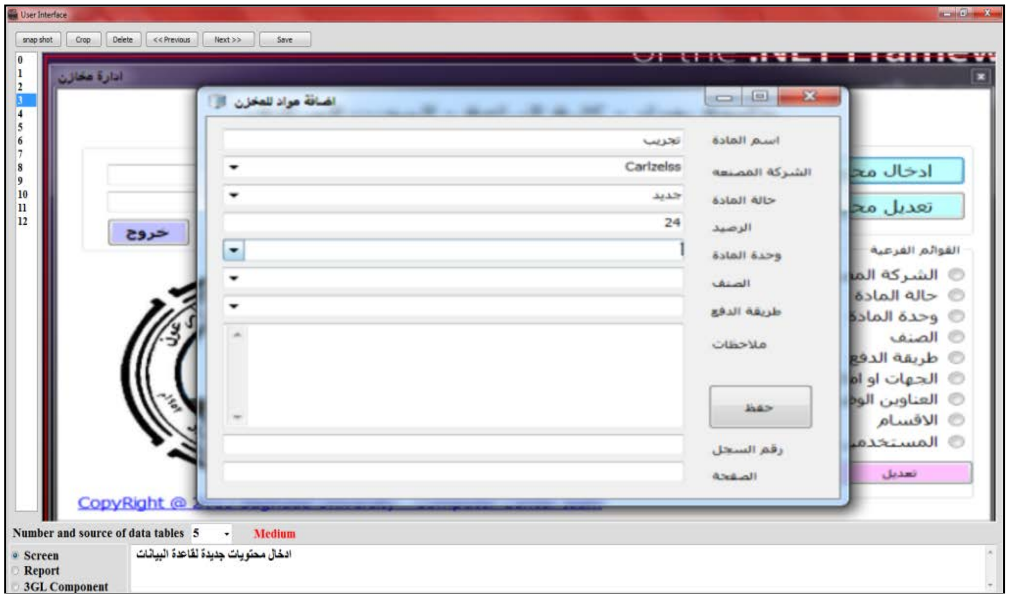


Number and source of data tables 2 Simple

[✓ Screen] Report 3GL Component

الواجهة الرئيسية للنظام

Figure (7) GUI Screen of the Central Repository Case Study (simple)



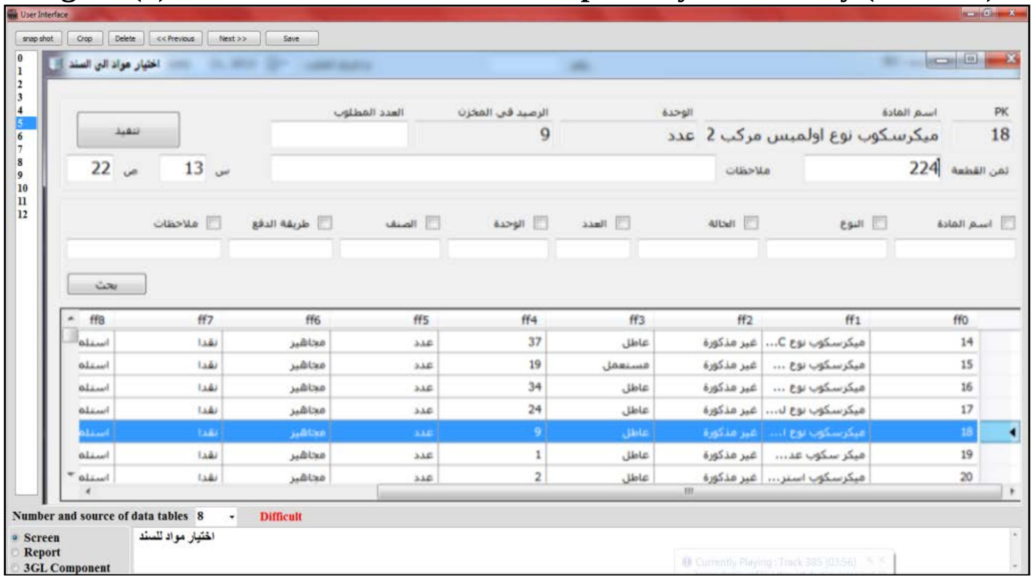
اسم المادة	تجريب
الشركة المصنعة	Carlzelss
حالة المادة	جديد
الرصيد	24
وحدة المادة	
الصنف	
طريقة الدفع	
ملاحظات	
رقم السجل	
الصفحة	

Number and source of data tables 5 Medium

[✓ Screen] Report 3GL Component

ادخال محتويات جديدة لقاعدة البيانات

Figure (8) GUI Screen of the Central Repository Case Study (medium)



PK	اسم المادة	الوحدة	الرصيد في المخزن	العدد المطلوب
18	ميكرسكوب نوع اولمبس مركب 2	عدد	9	

ثمن القطعة: 224
ملاحظات:
س: 13
ص: 22

☐ اسم المادة ☐ النوع ☐ الحالة ☐ العدد ☐ الوحدة ☐ الصنف ☐ طريقة الدفع ☐ ملاحظات

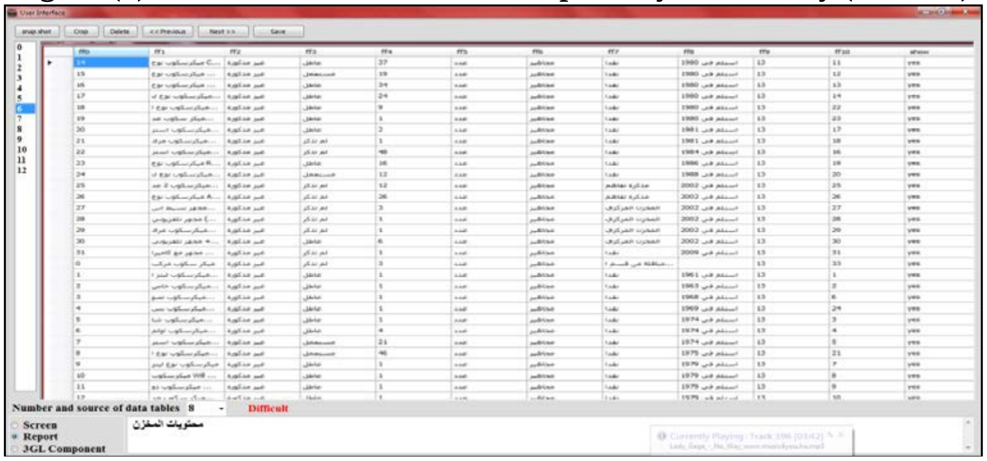
ff0	ff1	ff2	ff3	ff4	ff5	ff6	ff7	ff8
14	ميكرسكوب نوع C...	غير مذكورة	عاطل	37	عدد	مجاهير	نقدا	اسئلة
15	ميكرسكوب نوع ...	غير مذكورة	مستعمل	19	عدد	مجاهير	نقدا	اسئلة
16	ميكرسكوب نوع ...	غير مذكورة	عاطل	34	عدد	مجاهير	نقدا	اسئلة
17	ميكرسكوب نوع ل...	غير مذكورة	عاطل	24	عدد	مجاهير	نقدا	اسئلة
18	ميكرسكوب نوع ا...	غير مذكورة	عاطل	9	عدد	مجاهير	نقدا	اسئلة
19	ميكر سكوب عد...	غير مذكورة	عاطل	1	عدد	مجاهير	نقدا	اسئلة
20	ميكرسكوب استر...	غير مذكورة	عاطل	2	عدد	مجاهير	نقدا	اسئلة

Number and source of data tables 8 **Difficult**

[✓ Screen] Report 3GL Component

اختيار مواد للسند

Figure (9) GUI Screen of the Central Repository Case Study (difficult)

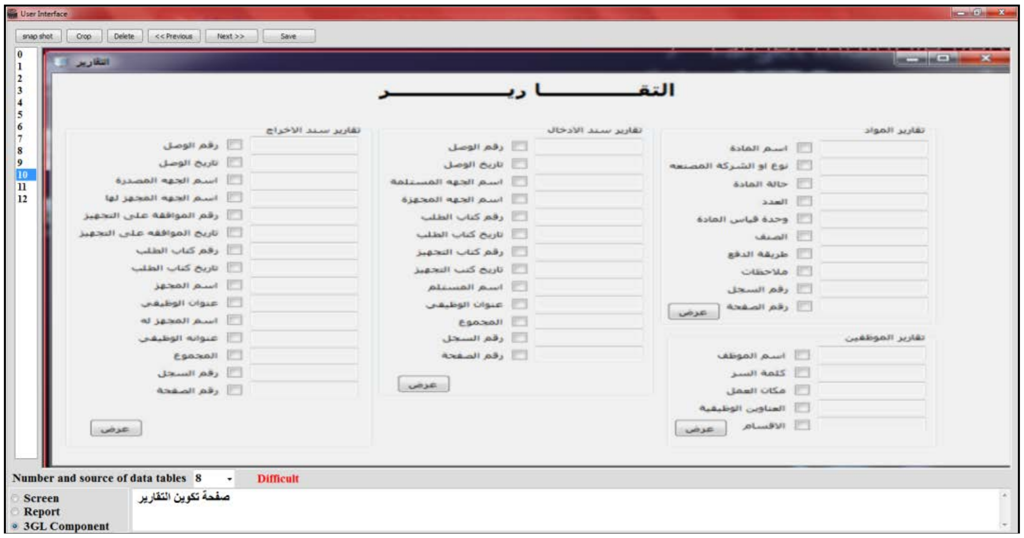


Number and source of data tables 8 **Difficult**

Screen [✓ Report] 3GL Component

محتويات المخزن

Figure (10) GUI Report of the Central Repository Case Study (difficult)



التقـــــــا ريـــــــــــر

تقارير المواد
☐ اسم المادة
☐ نوع او الشركة المصنعه
☐ حالة المادة
☐ العدد
☐ وحدة قياس المادة
☐ الصنف
☐ طريقة الدفع
☐ ملاحظات
☐ رقم السجل
☐ رقم الصفحة

تقارير الموظفين
☐ اسم الموظف
☐ كلمة السر
☐ مكان العمل
☐ العناوين الوظيفية
☐ الاقسام

تقارير سند الادخال
☐ رقم الوصل
☐ تاريخ الوصل
☐ اسم الجهه المستلمة
☐ اسم الجهه المجهزة
☐ رقم كتاب الطلب
☐ تاريخ كتاب الطلب
☐ رقم كتاب التجهيز
☐ تاريخ كتب التجهيز
☐ اسم المستلم
☐ عنوان الوظيفي
☐ المجموع
☐ رقم السجل
☐ رقم الصفحة

تقارير سند الاخراج
☐ رقم الوصل
☐ تاريخ الوصل
☐ اسم الجهه المصدرة
☐ اسم الجهه المجهز لها
☐ رقم الموافقه على التجهيز
☐ تاريخ الموافقه على التجهيز
☐ رقم كتاب الطلب
☐ تاريخ كتاب الطلب
☐ اسم المجهز
☐ عنوان الوظيفي
☐ اسم المجهز له
☐ عنوانه الوظيفي
☐ المجموع
☐ رقم السجل
☐ رقم الصفحة

Number and source of data tables 8 **Difficult**

Screen Report [✓ 3GL Component]

صفحة تكوين التقارير

Figure (11) 3GL Component of the Central Repository Case Study

9.1.3. Functional Size Measure (FSM)

The computation of the FSM depends on the UML sequence diagram. According to equation (7), each data movement is (entry, exit, read, and write) is identified, would increases the Cfsu by one while the acknowledgments are neglected because they are not considered as data movements. All the sequence diagrams for the system are measured and according to equations (8) and (9) the FSM is the sum of the data movements of all the sequence diagrams for the system.

For the Central Repository case study, there are three sequence diagrams illustrated in figures (12), (13), and (14). It can be seen from figure (12) that there are five data movements, such that FSM = 5 Cfsu. From the second sequence diagram shown in figure (13), it can be seen that there are seven data movements, such that FSM = 7 Cfsu. Finally there are seven data movements for the third sequence diagram shown in figure (14) such that FSM = 3 Cfsu. All the acknowledgements in all the sequence diagrams are neglected. According to equation (10), the final FSM = 5 + 7 + 3 = 15.

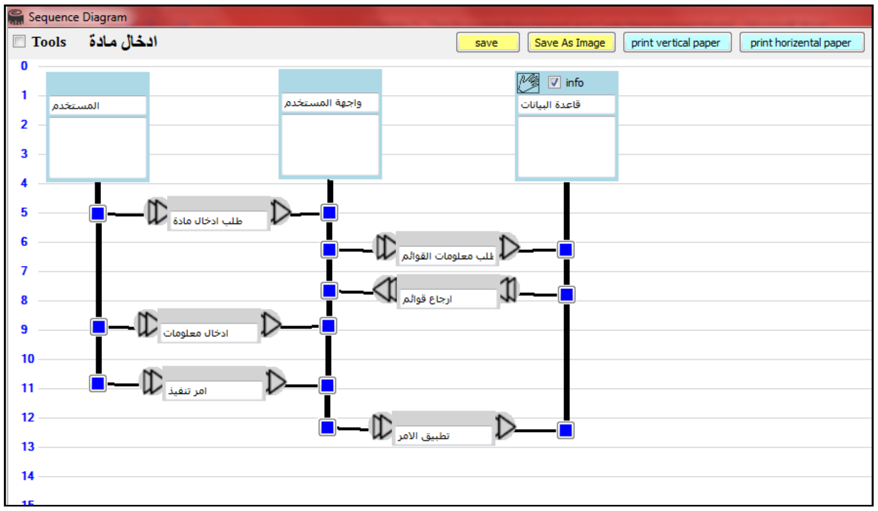


Figure (12) Requesting a material sequence diagram for the Central Repository Case Study

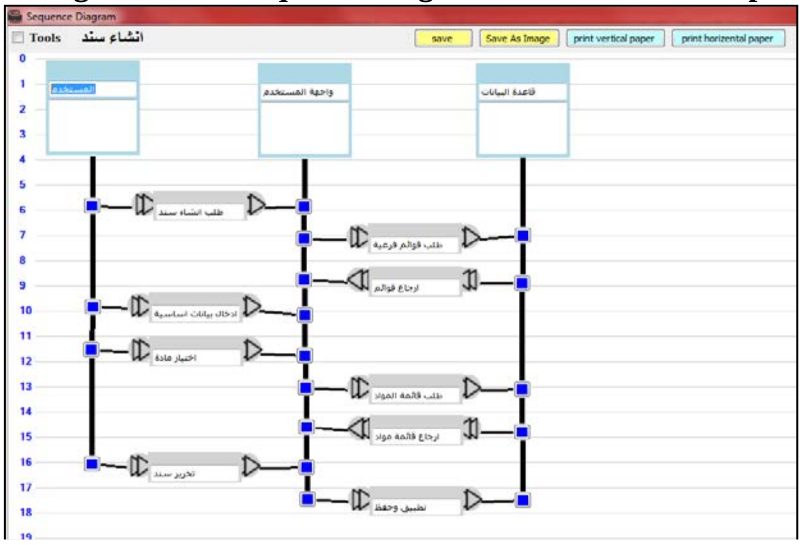


Figure (13) Creating a contract sequence diagram for the Central Repository Case Study

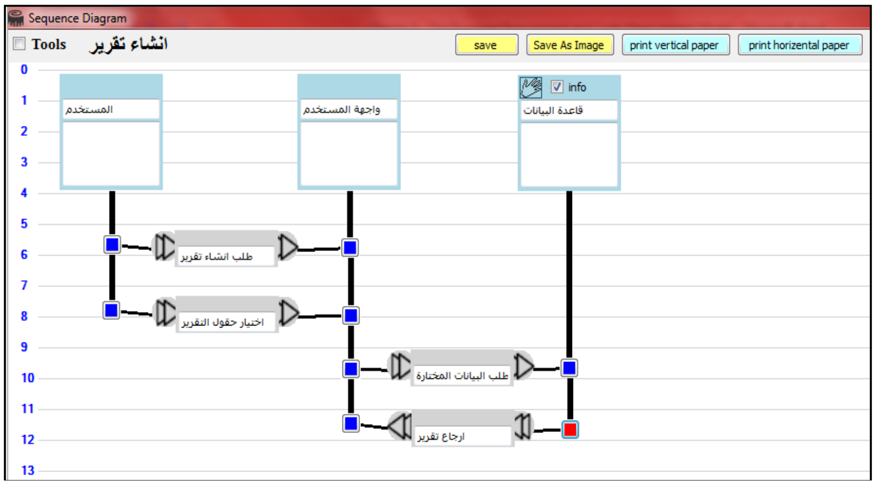


Figure (14) Creating a report sequence diagram for the Central Repository Case Study

9.2 The Code Measuring

This part of the proposed software measurement tool is responsible for comparison and proving purposes. The source code of the tested system is entered by selecting the (Java, C#, or VB) project folder or as a text file straight from their stored location. The proposed software measurement tool stores the source code of the tested system for documentation. The steps to measure the code metrics are as follows:

a. Brows the PC to choose the project folder: The tested system's source code files are automatically combined together in one list and the next operations are preformed sequentially. Figure (15) shows the browsing process to select the targeted system from its stored location.

b. Remove Comment lines: The comment lines are not part of the measurement process so the remaining lines are the pure source code. The total lines of code are counted automatically to show the size of the system after and before removing the comment lines. The comments in most of the programming languages are either (// comment line), (/* comment lines */) in JAVA and C#, or (' comment) in Visual Basic and the process of deleting the comment lines are shown in figure (16).

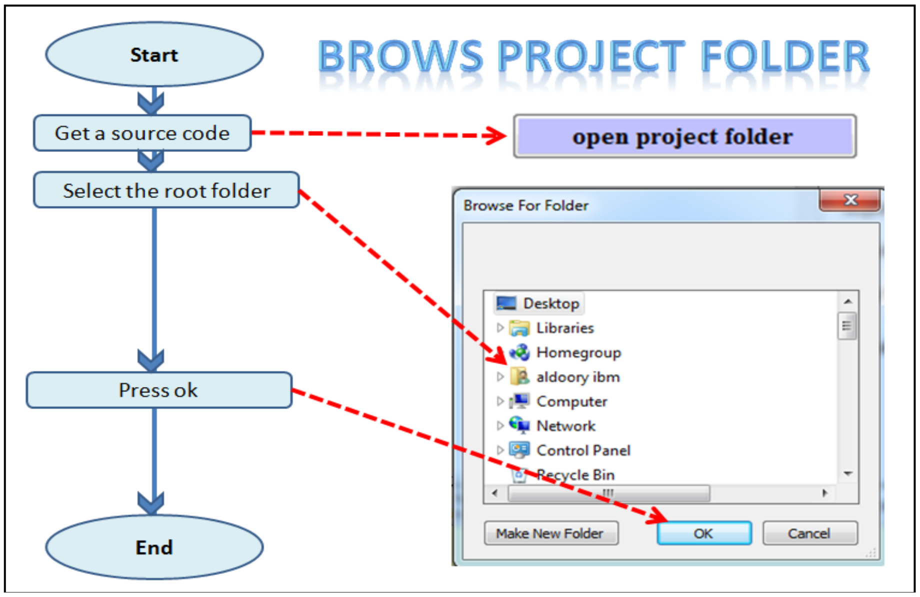


Figure (15) Open Project Folder Flowchart from the Proposed Software Measurement Tool

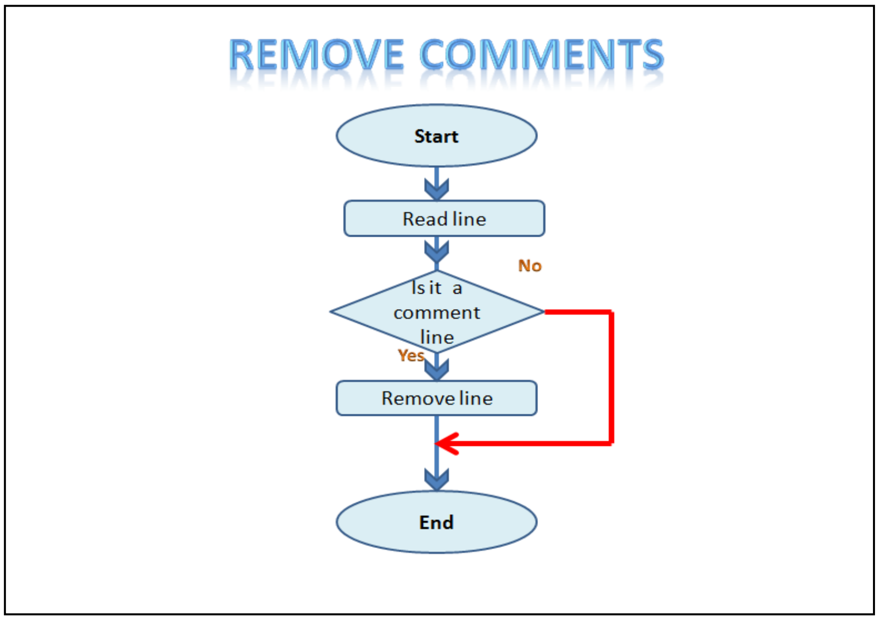


Figure (16) Remove Comment Lines Flowchart

c. Compute the Weighted Method per Class (WMC) metric : There are several methods to measure the complexity of a class. Based on equation (10), the WMC metric is programmed using the following proposed method: wherever a decision exists such as loops, conditional statement, switch statements, compound conditions, method calls (methods that are constructed by the programmer only), are encountered; then the DE (Decision Count) is increased by one such that:

Method Complexity = DE + 1 .... (13)

Figure (17) shows the flowchart for computing the WMC metric. Figure (18) shows the keyword count algorithm where the tested system source code is read line by line to search for certain keywords {if, for, while, goto, swith, &&, ||, foreach, do}. To compute the total number of method calls, a second method have been developed to store the function names in a list as shown in figure (19), then another method is created to look for the function calls in the tested system source code line by line as shown in figure (20). Figure (21) shows a method segment in the (Central Repository) case study which is programmed with C#, from this figure it can be seen that there are: one (IF) Statement, two call methods for the method GetFiles, each one increases the DE by 1; the method complexity for this method equals four.

d. Compute the Depth of Inheritance Tree (DIT) metric: The DIT metric focuses on the class levels in the class inheritance hierarchy, the root of the tree is considered level zero (DIT), and the level is increased when every node is encountered until the leaves are reached at the end of the inheritance tree, the leaves are in the highest level of the tree and this level represents the DIT metric. The DIT is a measure for both size and complexity. The DIT is programmed as show in figure (22), whenever a class inherits form another class, its father class is checked to see if it also inherits from another class, a counter starts form 0 that is increases by one whenever class inheritance is encountered, The DIT is performed for each class in the tested system. Figure (23) shows the class diagram for the (Central Repository) Case study, The father class is (form1), that makes it level 0, the class (R_Data_Structures) inherits from class (form1), so it's level 1, and class (access_2007_db_connection) inherits from class (R_Data_Structures) and that's makes it level 2.

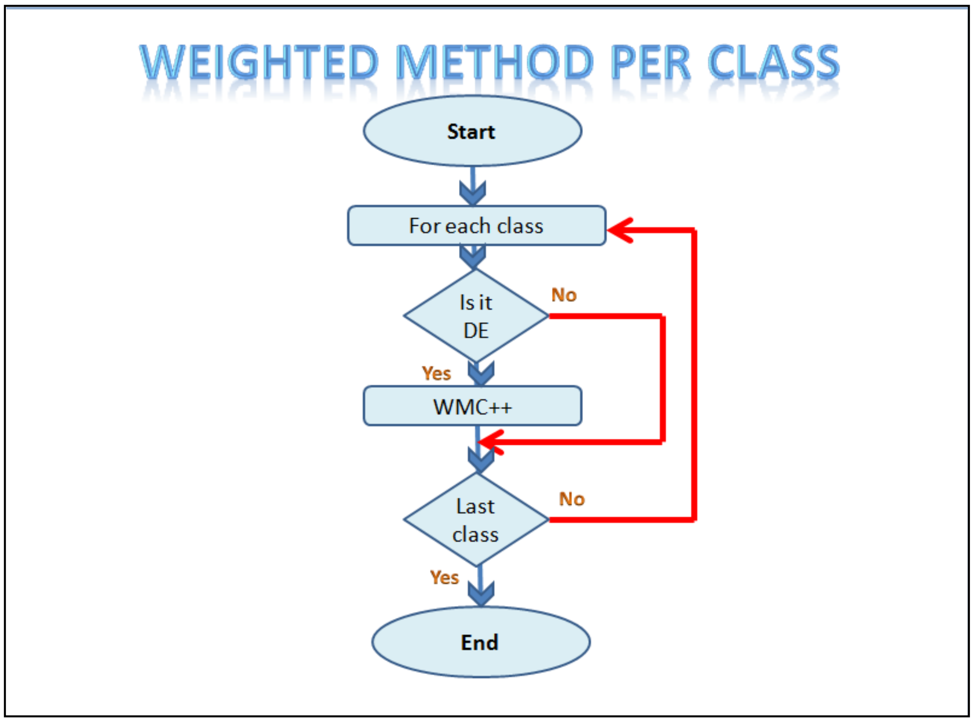
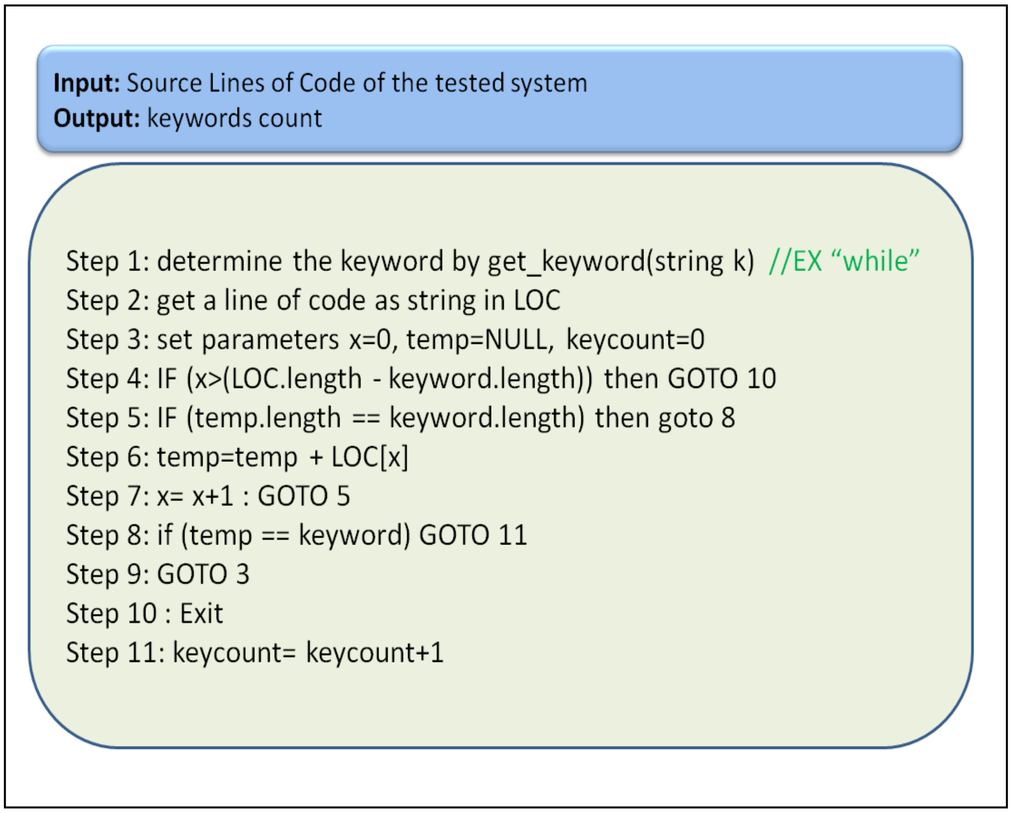


Figure (17) Compute WMC Flowchart



Input: Source Lines of Code of the tested system
Output: keywords count

```
Step 1: determine the keyword by get_keyword(string k)  //EX "while"
Step 2: get a line of code as string in LOC
Step 3: set parameters x=0, temp=NULL, keycount=0
Step 4: IF (x>(LOC.length - keyword.length)) then GOTO 10
Step 5: IF (temp.length == keyword.length) then goto 8
Step 6: temp=temp + LOC[x]
Step 7: x= x+1 : GOTO 5
Step 8: if (temp == keyword) GOTO 11
Step 9: GOTO 3
Step 10 : Exit
Step 11: keycount= keycount+1
```

Figure (18) Keyword Count Algorithm

```
Input: Source Lines of Code of the tested system
Output: Function names list

Step 1: get a new line of code as string in LOC from code list
Step 2: IF LOC start with {"void","int","string","double", "char","bool"}  GOTO 4
Step 3: GOTO 1
Step 4: IF LOC end with ")" return LOC in functions list
Step 5: GOTO 1
```

Figure (19) Build a List of Function Names Algorithm

```
Input: Source Lines of Code of the tested system + Function names list
Output: Functions calls count

Step 1: get a new line of code as string in LOC from code list
Step 2: IF LOC contain  '(' and LOC contain ')'  GOTO 3 ELSE GOTO 6
Step 3: for each item (itm) in functions list
Step 4: IF  LOC contain (itm) return (LOC) in Calls list
Step 5 : next ITEM
Step 6: GOTO 1
Step 7: FCcount=FCcount + Calls list.count
```

Figure (20) Computing Functions Calls Algorithm

```
public void Full_Path_Array(string From_This_Directoty, bool true_to_get_everything)
     {
      try
        {
         if (true_to_get_everything)
           {
             filePaths = Directory.GetFiles(From_This_Directoty,
"*.txt",searchOption.AllDirectories);
             }
             else
             {
filePaths=Directory.GetFiles(From_This_Directoty,"*.txt",SearchOption.TopDirectoryOnly);
             }
          }
          catch { }
       }
```

Figure (21) a method from the (Central Repository) Case study

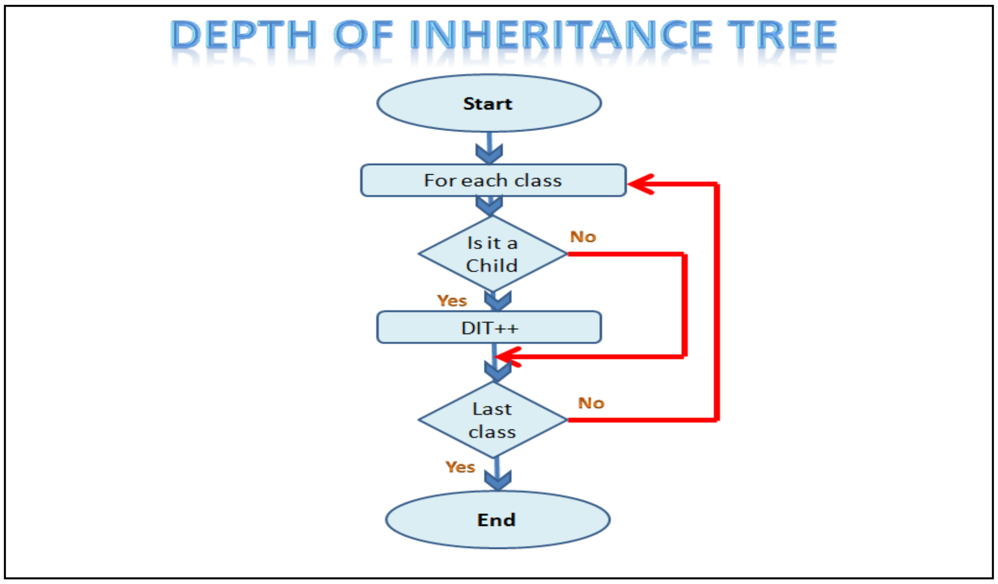


Figure (22) Compute DIT Flowchart

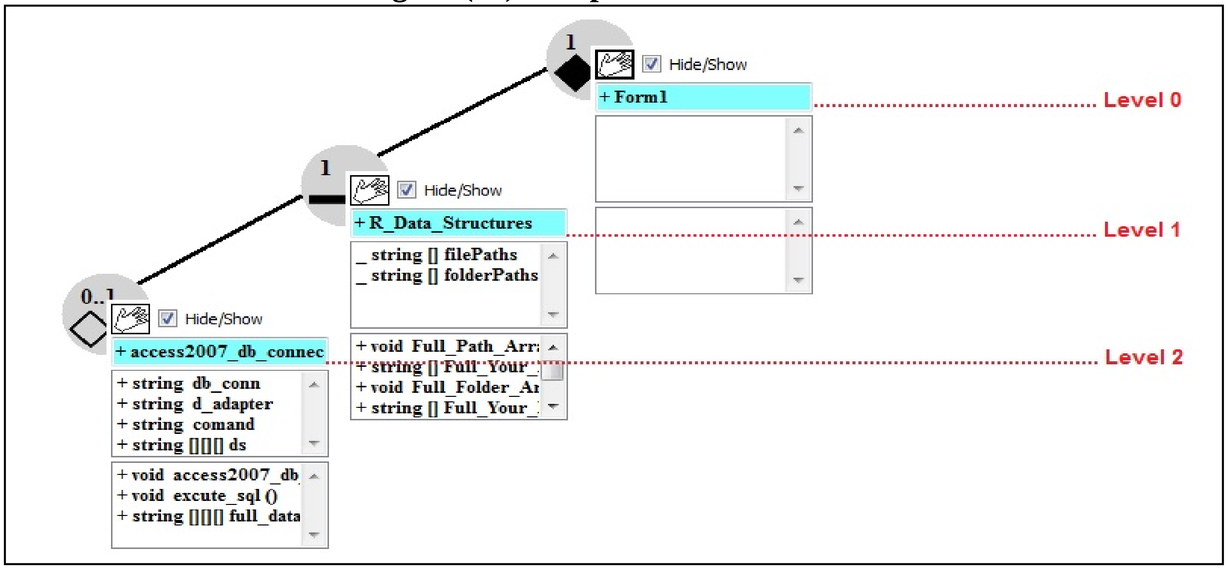


Figure (23) Leveled Class Diagram for the (Central Repository) Case Study from the Proposed Software Measurement Software tool

e. Computing the Class Size (CS) metric: The class size is the total number of the methods and attributes of each class, figure (24) shows the flowchart for computing the CS metric. A custom made algorithm works like a compiler to distinguish the attributes and methods built by the software developer of the tested system by transforming the source lines of code into string, then searches for attributes and functions based on the programming language structure to determine either the line is an attribute or function. Figure (25) shows a class within the (Central Repository) case study, which contains four attributes which are: OleDbConnection, OleDbDataAdapter, OleDbCommand, DataSet. And five methods which are: access2007_db_connection(string dbpath), access2007_db_connection(string dbpath, string user_id, string password), access2007_db_connection(string dbpath, string password), excute_sql(string sql_command), full_dataset(string sql_query). The CS is the total number of attributes and methods, so CS here is equal to nine. Figure (26) shows the code measures of the Central Repository case study from the GUI of the proposed software measurement tool.

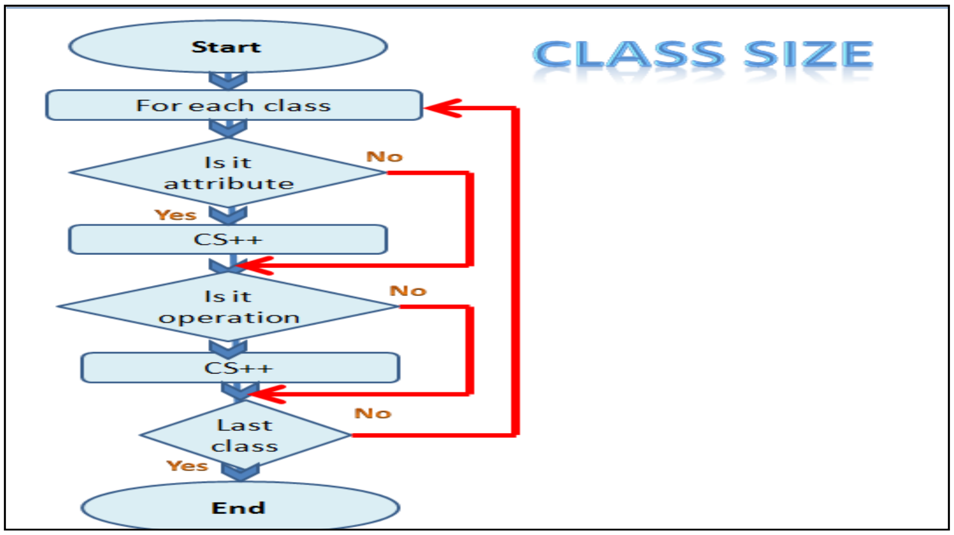
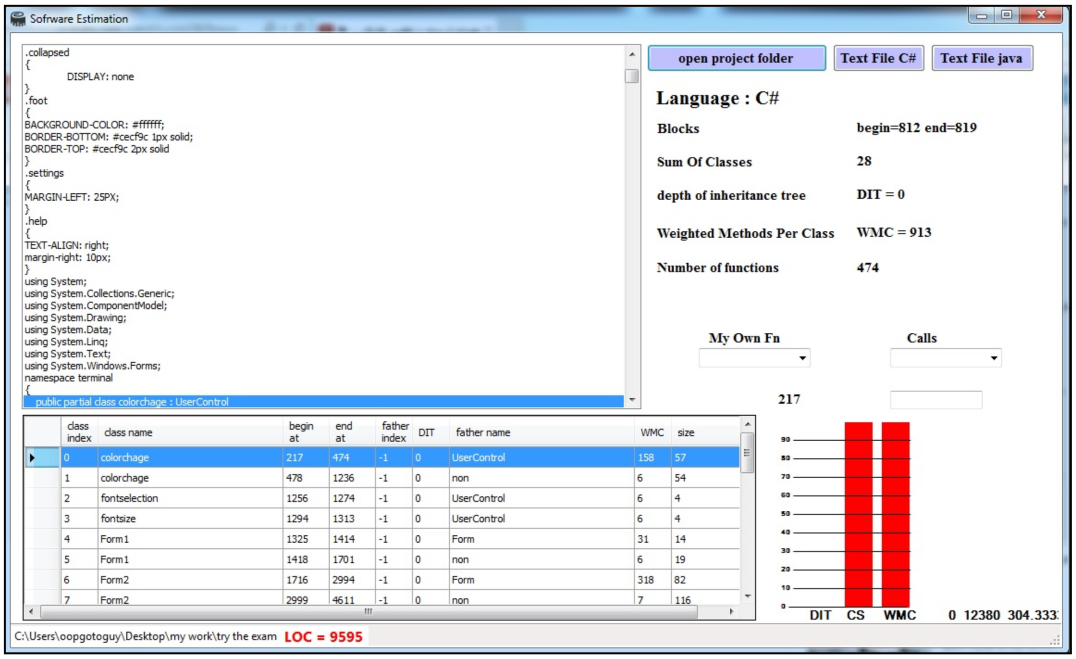


Figure (24) Compute CS Flowchart

```
public class access2007_db_connection : R_Data_Structures
 {
    public OleDbConnection db_conn;
    public OleDbDataAdapter d_adapter;
    public OleDbCommand comand;
    public DataSet ds;

    public access2007_db_connection(string dbpath)
    { ... }
    public access2007_db_connection(string dbpath, string user_id, string password)
    { ... }
    public access2007_db_connection(string dbpath, string password)
    { ... }
    public void excute_sql(string sql_command)
    { ... }
    public void full_dataset(string sql_query)
    { ... }
 }
```

Figure (25) a class from the (Central Repository) case study



```
.collapsed
{
        DISPLAY: none
}
.foot
{
BACKGROUND-COLOR: #ffffff;
BORDER-BOTTOM: #cecf9c 1px solid;
BORDER-TOP: #cecf9c 2px solid
}
.settings
{
MARGIN-LEFT: 25PX;
}
.help
{
TEXT-ALIGN: right;
margin-right: 10px;
}
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace terminal
{
  public partial class colorchage : UserControl
```

Language : C#

Blocks	begin=812 end=819
Sum Of Classes	28
depth of inheritance tree	DIT = 0
Weighted Methods Per Class	WMC = 913
Number of functions	474

217

class index	class name	begin at	end at	father index	DIT	father name	WMC	size
0	colorchage	217	474	-1	0	UserControl	158	57
1	colorchage	478	1236	-1	0	non	6	54
2	fontselection	1256	1274	-1	0	UserControl	6	4
3	fontsize	1294	1313	-1	0	UserControl	6	4
4	Form1	1325	1414	-1	0	Form	31	14
5	Form1	1418	1701	-1	0	non	6	19
6	Form2	1716	2994	-1	0	Form	318	82
7	Form2	2999	4611	-1	0	non	7	116

0 12380 304.333

C:\Users\oopgotoguy\Desktop\my work\try the exam LOC = 9595

Figure (26) the Code Measures of the Central Repository Case Study from the GUI of the Proposed Software Measurement Tool

9.3 Requirement Model (RM) and Code Model (CM)

Ten case studies have been collected and analyzed by using the selected code and requirement metrics. These metrics focus on the size and complexity concepts, for example, table (11) illustrates the final requirement metrics extracted from the requirement engineering documents for the Central Repository Case Study.

Table (11) Requirement Measures for the Central Repository Case Study

Requirement Metric	Value
UCP	20.4771
OP	51
FSM	15

The results have been gathered so it would be possible to simply aggregate the requirement metrics that are measured straight from the Object-Oriented system requirement engineering documents and according to the requirement metrics results. A mathematical model is proposed based on the concept of building an estimation model equation (12), this model is named the RM (Requirement Model) which its equation can be used to estimate the code measures based on requirements engineering documents.

RM = UCP + FSM + OP.. (14)

Where UCP is the Use Case Points metric, FSM is the Functional Size Measure metric, and OP is the Object Points metric

Based on the proposed equation (14) RM for the Central Repository case study is as follows:

RM = 20.4771 + 15 + 51 = 86.4771

To find and prove the RM model, the proposed CM (Code Model) is proposed using the selected code metric that are calculated for each class in every system. For example table (12) shows the final code metrics for the Central Repository Case Study. The CM (Code Model) equation is:

$$\mathbf{CM} = \frac{\sum_{i=1}^{n} \mathbf{WMC} + \sum_{i=1}^{n} \mathbf{DIT} + \sum_{i=1}^{n} \mathbf{CS}}{\mathbf{n}} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots \mathbf{(15)}$$

where WMC is the Weighted Method per Class metric, DIT is the Depth of Inheritance Tree metric, CS is the Class Size metric, and n is the number of classes.

Based on the proposed equation (14) the CM for the Central Repository case study is as follows:

$$\mathrm{CM} = \frac{188 + 2 + 77}{3} = 89$$

Table (12) Code Measures for the Central Repository Case Study

Code Metric	Value
WMC	**188**
DIT	**2**
CS	**77**
no. of classes	**3**

It can be seen that the RM $\cong$ CM and according to equation (11) the relative error between the RM and CM is computed as follows:

$$\text{Relative Error} = \frac{|89 - 86.4771|}{89} \times 100\% = 0.0283\%$$

So the relative error is very low, such that the proposed RM which is derived from the requirement engineering documents can be used to estimate the code metrics.

10 Results and Discussion

Ten case studies have been analyzed; the requirement metrics for all the case studies are shown in table (13). The final results for computing the proposed RM and the Proposed CM are shown in table (14), and the code metrics for the case studies are listed from in table (15-24) in the appendix. it can be seen that by measuring the relative error according to equation (11) for each case study and calculating the average of the ten case studies, the error value is insignificant and the relation is proved to be approximately linear between the proposed CM and RM formulas, and thus the CM could be estimated from the RM.

Figure (27) shows the scatter diagram for all the ten tested case studies, the labeled numbers on the diagram represent the number of the case study, where each case study has a set of its own (RM,CM) point. This figure shows the linear relationship between the RM and CM. This indicates that the ability to estimate the code metrics based on the proposed RM that is extracted from the requirement engineering document.

The system developers would benefit the most form anticipating the code measures at the requirement phase, hence they would be able to obtain the size and complexity measures of the source code in advance. By using the proposed software measurement tool, the programmers would have the privilege to anticipated the source code measure in advance, document the system by using the UML documentation and drawing tools and thus the customers would have a complete view of the system under construction along with its functionality.

11. Conclusions

1. The proposed software measurement tool enables the user to create requirement engineering documents by using the proposed UML tools to generate (use- case diagrams, sequence diagrams, and class diagrams). Then extracts the required information from these documents to compute the requirement metrics automatically.

2. It is possible to predict the CM (Code Model) from the RM (Requirement Model) that are extracted from the requirement engineering document during the first stage in the software life-cycle which is the requirement phase.
3. By measuring the code metrics based on requirement engineering documents, the software programmers would have a complete picture of what the system would be before the coding process; that would help them to work with no ambiguity.
4. Another benefits gained from measuring the code metrics at early stages of software development are saving time and reducing cost hence the system is fully understood and estimated earlier in the requirement phase.
5. The errors could be reduced where the system is anticipated at the requirement phase.
6. The proposed software measurement tool is capable of computing the code metrics (WMC, DIT, and CS) of a tested system source code in order to determine complexity and size of that system.
7. The proposed system stores the requirement information and the code measures in a data-base so all the tested systems are fully documented.
8. The proposed Requirement Model (RM) and Code Model (CM) have been computed for all the case studies, the results showed that the RM and CM are approximately equal and the relative error between the two models is insignificant, so it is proved that the proposed RM is capable of estimating the CM at the requirement phase of the software life-cycle.

Table (13) the Requirement Metrics for the Ten Case Studies

Case Study	UCP	OP	FSM	Number of requiremen Classes
Materialized View	**20.5205**	**55**	**18**	**3**
Al-Ibdaá for car spare parts	**16.641**	**31**	**10**	**12**
Electronic Signature	**43.3895**	**28**	**25**	**3**
Wav Visualizer	**16.3438**	**19**	**8**	**8**
Central Repository	**20.4771**	**51**	**15**	**3**
Image Finder	**7.3232**	**20**	**5**	**2**
Almasal	**18.5535**	**44**	**3**	**5**
Engineering Time, Cost Estimation	**11.9414**	**65**	**3**	**7**
ToDo List	**10.8173**	**13**	**4**	**2**
Group Paint	**7.777**	**10**	**3**	**2**

Table (14) The Results of the case studies and the Error Measurement

No.	Case Study	RM	CM	Relative error
1	**Materialized View**	**93.5205**	114	**0.1796%**
2	**Al-Ibdaá for cars spear parts**	**57.641**	67	**0.1396%**
3	**Electronic Signature**	**96.3895**	93	**0.0364%**
4	**Wav Visualizer**	**43.3438**	53	**0.1821%**
5	**Central Repository**	**86.4771**	89	**0.0283%**
6	**Image Finder**	**32.3232**	41	**0.2116%**
7	**Almassal**	**65.5535**	65	**0.0085%**

8	**Engineering time, Cost estimation**	**79.9414**	91	**0.1215%**
9	**ToDo List**	**27.8173**	25	**0.1126%**
10	**Group Paint**	**20.77**	21	**0.0109%**
Average		**60.3777**	65.9	65.9

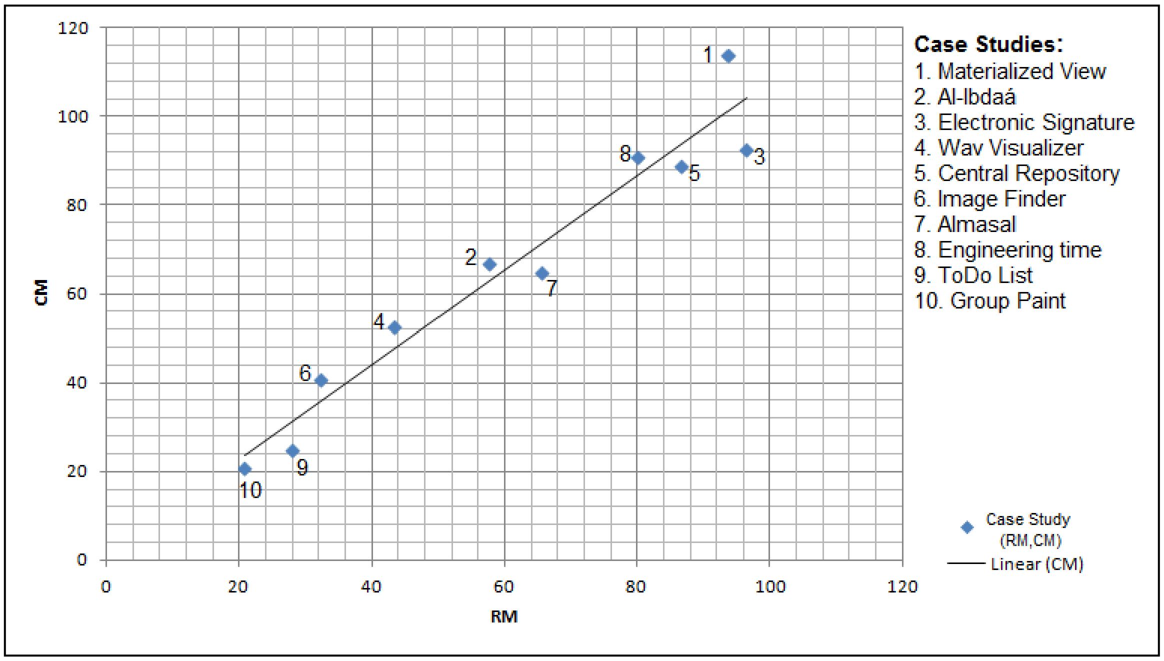


Figure (27) the Scatter Diagram for all the case studies

12 References

[BS04] K.Barclay and J. Savage, *"Object-Oriented Design with UML and Java"*, Elsevier, 2004.
[Che09] Murali Chemuturi, *"Software Estimation Best Practice, Tools & Techniques, A complete Guide for Software Project Estimators"*, J. Ross, 2009.
[CK94] Shyam R. Chidamber and Chris F. Kemerer, "A Metrics Suite for Object Oriented Design", IEEE Transactions on Software Engineering, Vol. 20, NO. 6, 1994.
[Coc] *"COCOMO II Model Definition Manual"*, version 1.4, University of Southern California.
[Cos09] *"The COSMIC Functional Size Measurement Method"*, Measurement Manual v 3.0.1, 2009.
[FAP12] Nelly Condori-Fernandez, Silvia Abrahao, and Oscar Pastor, *"Towards a Functional Size Measure for Object-Oriented Systems from Requirements Specification"*, IEEE, 2012.
[GL96] Gene H. Golub, and Charles F. Van Loan, *"Matrix Computations"*, Johns Hopkins Studies in Mathematical Sciences, 3rd Edition, 1996.
[Jal97] Pankaj Jalote, *"An Integrated Approach to Software Engineering"*, second edition, Springer, 1997.
[Jam06] Seyyed Mohsen Jamali, *"Object Oriented Metrics (A Survey Approach)"*, 2006.

[JS04] Jubair J. Al-Ja'afer and Khair Eddin M. Sabri, *"Chidamber-Kemerer (CK) and Lorenz and Kidd (LK) Metrics to Assess Java Programs"*, 2004.
[Kan04] Cem Kaner, *"Software Engineering Metrics: What Do We Measure and How Do We Know"*, IEEE, 10th International Software Metrics Symposium, 2004.
[Ken09] Simon Kendal, *"Object-Oriented Programming Using Java"*, Ventus Publising Aps, 2009.
[Kar93] Gustav Karner, *"Resource Estimation for Objectory Projects"*, 1993.
[KKB08] Ananya Kankilal, Goutam Kanjilal, and Swapan Bhattacharya, *"Metrics-based Analysis of Requirements for Object-Oriented Systems: An Emperical Approach"*, 2008.
[KR10] Linda Kenneth and Helda Rogardt, *"On the Relationship between Functional Size and Software Code Size"*, ACM, 2010.
[LK94] Mark Lorenz and Jeff Kidd, *"Object-Oriented Software Metrics"*, PTR Prentice Hall, 1994.
[LL05] Timothy C. Lethbridge and Robert Laganiere, *"Object-Oriented Software Engineering Practical Software Development Using UML and Java"*, 2nd Edition, Mc Graw Hill, 2005.
[LR10] Luigi Lavazza and Gabriella Robiolo, *"Introducing the Evaluation of Complexity in Functional Size Measurement: a UML-based Approach"*, ACM, 2010.
[NT05] Iftikhar Azim Niaz and Jiro Tanaka, *"An Object-Oriented Approach to Generate Java Code from UML Statecharts"*, International Journal of Computer & Information Science, Vol.6, No.2, June 2005.
[Pre10] Roger S. Pressman, *"Software Engineering, A Practitioner's Approach"*, Seventh edition, McGraw-Hill Companies, 2010.
[RRR12] Dr.GSVP Raju, K.Koteswara Rao, and M Sumender Roy, *"A Case Study Approach to Measure the Function Points from the Points of Relationships of UML"*, International Journal of Computer Applications, Volume 10, No. 10, 2012.
[SK0510] Ashish Sharma and D.S. Kushwaha, *"A Complexity Measure Based on Requirement Engineering Document"*, journal of computer science and engineering, volume 1, issue 1, May 2010.
[SK11] Ashish Sharma and Dharmender Singh Kushwaha, *"A Metric Suite for Early Estimation of Software Testing Effort Using Requirement Engineering Document and its Validation"*, International Conference of Computer & Communication Technology (ICCCT) 2011.
[SS12] Dami˜ao N. da Silva and Chris Skinner, *"Adjusting for Survey Measurement Error with Accuracy Variables"*, JSM, 2012

Appendix

Table (15) Code metrics for the Materialized View Case Study

Class Name	DIT	WMC	CS
Form1	0	160	23
Acess2007_db_connection	1	7	9
R_Data_Structures	1	105	37
Total	2	272	69

Table (16) Code metrics for Al_Ibda'a for Car Spare Parts Case Study

Class Name	DIT	WMC	CS
billform	0	51	16
billsrecords	0	109	32
brand	0	16	0
carbrand	0	16	7
customers	0	59	20
customer_list	0	11	5
form1	0	30	15
items	0	111	27

printing	0	17	8
qq	0	31	11
reseat_bill	0	26	9
storage	0	145	36
Total	0	622	186

Table (17) Code metrics for the Electronic Signature Case Study

Class Name	DIT	WMC	CS
WebCam	1	12	8
R_Data_Structures	1	105	37
Ramy_Protocol	2	87	28
Total	4	204	73

Table (18) Code metrics for Wav Visualizer Case Study

Class Name	DIT	WMC	CS
AudioFrame	0	17	11
FifoStream	0	28	18
R_Data_Structures	2	105	37
WaveInBuffer	2	14	15
WaveInRecorder	2	28	16
WaveOutBuffer	4	14	15
WaveOutPlayer	4	26	16
Total	14	232	128

Table (19) Code metrics for Central Repository Case Study

Class Name	DIT	WMC	CS
Form1	0	76	31
access2007_db_connection	1	7	9
R_Data_Structures	1	105	37
Total	2	188	77

Table (20) Code metrics for Image Finder Case Study

Class Name	DIT	WMC	CS
Form1	0	43	13
login	0	20	6
Total	0	63	19

Table (21) Code metrics for Almasal Case Study

Class Name	DIT	WMC	CS
Form1	0	92	38
Form2	0	41	19
Form3	0	56	22
Form4	0	29	14
Form5	0	13	5
Total	0	231	98

Table (22) Code metrics for Engineering Time, Cost Estimation Case Study

Class Name	DIT	WMC	CS
Form1	0	233	75

MyListBoxItem	**1**	**1**	**2**
Form2	**0**	**103**	**33**
Form3	**0**	**259**	**89**
Form4	**0**	**10**	**4**
Form5	**0**	**20**	**9**
Form6	**0**	**12**	**4**
Total	**1**	**638**	**216**

Table (23) Code metrics for ToDo List Case Study

Class Name	DIT	WMC	CS
Form1	**0**	**28**	**9**
Form2	**0**	**9**	**4**
Total	**0**	**37**	**13**

Table (24) Code metrics for Group Paint Case Study

Class Name	DIT	WMC	CS
Program	**0**	**1**	**1**
ramy	**0**	**9**	**4**
Total	**0**	**30**	**10**

المستخلص

تستخدم المقاييس من قبل صناع البرمجيات لتقييم جودة الأنظمة البرمجية قبل خلقها ولذلك تأثير كبير على جودة إتخاذ القرارات في مرحلة متقدمة خلال جمع المتطلبات، تقييم عمليات تطوير، تشيغل وصيانة النظام البرمجي. تعلمنا المقاييس بحالة النظام وخصائصه وتساعدنا في تقييم النظام بطريقة موضوعية.

في هذا البحث تمكنا من التنبؤ بمقاييس الشفرة البرمجيةبإستخدام مقاييس المتطلبات التي جمعت من وثائق هندسة المتطلبات. قد تم تحليل عشرة حالات دراسة وكذلك فرض نموذجين رياضيين احدهما لمقاييس المتطلبات البرمجية وهو نموذج المتطلبات والآخر لمقاييس الشفرة البرمجية وهو نموذج الشفرة البرمجية اعتماداً على جمع وتحليل البيانات للأنظمة المفحوصة. تتم عملية جمع هذه البيانات أوتوماتيكياً بإستخدام أداة القياس البرمجية المقترحة، تتكون هذه الأداة من جزئين: أداة لغة التصاميم الموحدة وأداة مقاييس الشفرة البرمجية.

النتائج اثبتت تقارب قيم نموذج المتطلبات ونموذج الشفرة البرمجية وذلك تم اثباتها بإستخدام مقياس الخطأ النسبي لهاتين القيمتين ونتج ذلك عن طريقة لحساب مقاييس الشفرة البرمجية في المراحل المتقدمة من دورة الحياة البرمجية وبذلك تم تقليل الوقت والجهد والتكلفة والضياع.