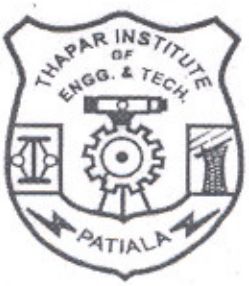
A Framework for Effort Estimation Using Use Case

A thesis
submitted in the partial fulfillment of the requirements
for the award of degree
of

Master of Engineering
in
Software Engineering



Submitted by
Gaurav Singhal
(8033105)

Under the guidance of
Sh. R. S. Salaria
Assistant Professor & Head,
CSED, TIET, Patiala

COMPUTER SCIENCE & ENGINEERING DEPARTMENT
THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
(DEEMED UNIVERSITY)
PATIALA-147004 (PUNJAB)

May 2005

Candidate's Declaration

I hereby declare that the thesis entitled **"A Framework for Effort Estimation Using Use Case"** submitted by me in the partial fulfillment of the requirements for the award of degree of **Master of Engineering in Software Engineering** at Thapar Institute of Engineering & Technology (Deemed University), Patiala, is an authentic record of my own work carried out under the supervision and guidance of Sh. R. S. Salaria.

The matter presented in this thesis has not been submitted by me for the award of any other degree or diploma of this or any other university.

Gaurav Singhal

(Gaurav Singhal)

This is to certify that the above statement made by the candidate is true and correct to the best of my knowledge and belief.

15/05/08

(Sh. R. S. Salaria)

Supervisor
Assistant professor
Computer Science & Engineering Department
Thapar Institute of Engineering & Technology
Patiala

Countersigned by:

20/05/08

(Sh. R. S. Salaria)

Head
Computer Science & Engg. Department
Thapar Institute of Engg. & Technology
Patiala

(Dr. D. S. Bawa)

Dean (Academic Affairs)
Thapar Institute of Engg. & Technology
Patiala

Acknowledgement

The thesis would not be considered complete unless I pay my sincere thanks to some important people whose guidance and support was invaluable for my thesis work.

Firstly, I would like to thank **"Maa Vaishno"** who was by my side at all the tough moments, for not letting me down at the time of crises and showing me the silver lining in the dark clouds.

My sincere gratitude to my research guide, Sh. R. S. Salaria, Assistant Professor and Head, Computer Science & Engineering Department, TIET, Patiala, for providing me incalculable guidance, suggestions and support throughout the course of this thesis work, and I owe much to his wisdom, generosity and patience.

I would like to give special thanks to Sh. Rajesh Bhatia, Assistant Professor, Computer Science & Engineering Department, and other faculty members for always having an open door and providing their invaluable suggestions to me.

I would also like to thank Mr. Mahesh Dubey, Assistant Systems Engineer (Consultancy), TCS, Gurgaon and Mr. Manish Kumar, Software Engineer, Flexitronics Software Systems, Gurgaon Software who were always there at the need of the hour and helped me in hundred little ways that meant a lot.

Last but certainly not the least, I thank my parents and my brother for providing their uncanny guidance and giving me a great head start on life. I am indebted to their priceless moral support and inspiration.

Gaurav Singhal

(GAURAV SINGHAL)

Abstract

We cannot control what we cannot measure. Therefore in order to compare, to control and to predict we need software metrics. The need to improve productivity and software quality has put forward the research on software metrics technology.

Various object oriented metrics are proposed by the researchers each targeting a particular phase of software development life cycle which is divided into four phases - *analysis, design, implementation, and testing*. The main purpose of the metrics in analysis phase is estimation of the resources that might be needed for the software project.

Unfortunately, the software profession is notoriously inaccurate when estimating cost and schedule. Reliable early estimates are difficult to obtain because of the lack of detailed information about the future system at an early stage. However, early estimates are required when bidding for a contract or determining whether a project is feasible in the terms of a cost-benefit analysis.

The Unified Modeling Language (UML) has become the standard Object-Oriented design language in software engineering. An important part of the UML is the Use Case notation. Since Use Case modeling is increasingly being utilized as the method of choice to describe the software and system requirements and as a basis of design, development, testing, deployment, configuration management and maintenance; the attributes of a use case model may therefore serve as measures of the size and complexity of the functionality of a system. Many organizations use a system's use case model in the estimation process. Estimating software with use cases is still in the early stages. This thesis describes a "Framework for Effort Estimation using Use Case". It uses use case model, accompanied with environment and technical factors, to estimate the resources that is needed for the software project. This thesis also provides the various guidelines for assigning values to the various environmental and technical factors. These factors are very important while estimating the effort.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 INTRODUCTION

Software Metrics plays a vital role in quantifying resources for software development, attributes of software product and process. Metrics were used in conjunction with prediction models to predict different aspects of software product. Nowadays, software engineers recognize the need of software metrics to understand, manage and improve the total software development process. In order to do this, metrics relevant to software engineering process and products have to be collected and analyzed.

"Measurement is the process by which numbers or symbols are assigned to attribute of entities in the real world in such a way so as to describe them according to clearly defined rules".

Measurement is very important in software industry because:

- ✓ Software metrics can help to fully understand design and architecture information of the software system.
- ✓ Software metrics can evaluate quality of the software and provide cost estimation of software project.
- ✓ Software metrics can help to determine the effect of object technology, especially reuse technology applied in the software development according to some quantitative evaluation such as productivity, quality, lead-time, maintainability, reusability etc.

1.2 MEASUREMENT FOR WHOM

Measurement lies at the heart of many systems that govern our lives. Economic measurements determine price and pay increases. Measurements in radar systems enable us to detect aircraft when direct vision is obscured. Medical system measurements enable doctors to diagnose specific illness. Measurements in atmospheric systems are the basis of weather prediction.

"Without measurements technology cannot function".

Each of us uses measurement in everyday life. Price acts as a measure of value of an item in a shop, and we calculate total bill to make sure the shopkeeper gives us correct change. We use height and size measurements to ensure that our clothing will fit properly. When making a journey, we calculate distance, choose our route, measure our speed, and predict when we will arrive at our destination.

1.3 WHY MEASUREMENT?

Measurement helps us to understand our world, interact with our surroundings and improve our lives. It gives us a way of comparing, controlling and predicting. Measurement gives us a way tracking progress and rescheduling, if necessary. It also provides an access of product quality [30].

"You cannot control what you cannot measure",

----- TOM DEMARCO, 1982.

Measurement is the process by which numbers and symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules.

Thus, measurement captures information about attributes of entities. An *entity* is an object (such as a person or room) or an event (such as a journey or the testing phase of a software project) in the real world. An *attribute* is a feature or property of an entity. Typical

attributes include the area or color (of a room), the cost (of a journey), or the elapsed time (of the testing phase).

1.4 MEASUREMENT IN SOFTWARE ENGINEERING

Measurements have been widely used in many engineering disciplines, yet in the computer software industry there still have been doubts about their use in this field. With the project programmer and manager focusing on software productivity and software quality, there exists a need for the better technique of software development and software metrics during the process of development. There are some main reasons that software metrics became very important in software industry [17]:

- ✓ Software metrics can help to fully understand both the design and architecture information of the software system. Additionally, it can help start to understand the process of development by applying the process evaluation in the activity of software development.

- ✓ Software design metrics can aid to discover the underlying errors in the software design at the early stage of software development life cycle. Software metrics can also assists the task of software test.

- ✓ Software metrics can evaluate the quality of the software and provide cost estimation of software project. It becomes easier to estimate and plan new activities based on them. Utilizing the measurement of the current activity, it is easy to control progress and improve the process to make it more cost effective in the future.

- ✓ Software metrics can help to determine the effect of the object technology, especially reuse technology applied in the software development according to some quantitative evaluation such as productivity, quality, lead time, maintainability, reusability etc.

Thus, if we neglect measurement in software engineering then,

- ✓ We fail to set measurable targets for our software products. For example, we promise that the product will be user-friendly, reliable and maintainable without specifying clearly and objectively what these terms actually mean. As a result, when the project is complete, we cannot tell if we have met our goals.

Projects without clear goals will not achieve their goals clearly.

----TOM GILB (1988).

- ✓ We fail to understand and quantify the components costs of software projects. So, we cannot hope to control costs if we are not measuring the relative components of cost.

- ✓ We do not quantify or predict the quality of the products we produce.

- ✓ We allow anecdotal evidences to convince us to try yet another revolutionary new development technology, without doing a carefully controlled study to determine if the technology is efficient and effective.

1.5 REPRESENTATION CONDITION OF MEASUREMENT

The representation condition asserts that a measurement mapping M must map entities into numbers and empirical relations into numerical relations in such a way that the empirical relations preserve and are preserved by the numerical relations.

In fig 1.1, we see that the empirical relation "taller than" is mapped to the numerical relation ">" [17]. In particular, we can say that

A is taller than B if and only if M (A)>M (B)

This statement implies that:

- ✓ Whenever Joe is taller than Fred, then *M (Joe)* must be a bigger number than *M (Fred)*.

- ✓ We can map Joe to a higher number than Fred only if Joe is taller than Fred.

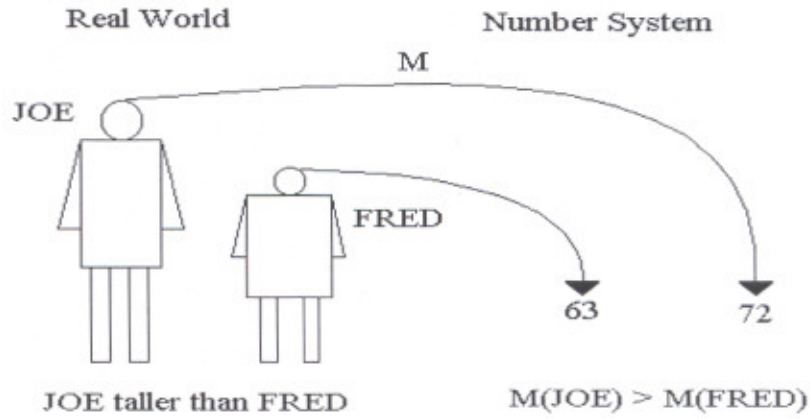


Fig. 1.1 Representation Condition

1.6 KEY STAGES OF FORMAL MEASUREMENT

There are 5 stages in a measurement process [17];

- ✓ Identify attribute for some real world entities.
- ✓ Identify empirical relations for Attributes.
- ✓ Identify numerical relations corresponding to each empirical relation.
- ✓ Define mapping from real world entities to number.
- ✓ Check that numerical relations preserve and are preserve by empirical relations (Representational Condition).

1.7 CHARACTERISTICS OF SOFTWARE METRICS

There are several characteristics of software metrics. These are given below [30, 17]:

- ✓ Simple and computable

- ✓ Persuasive
- ✓ Consistent
- ✓ Programming language independent
- ✓ Field tested

1.8 CLASSIFICATION OF SOFTWARE METRICS

Software metrics can be classified in to three parts [19]:

1.8.1 Product Metrics

Product metrics are a measure of the software product at any stage of its development, from requirements to installed system. Product metrics may measure the complexity of the software design, the size of the final program (either source or object code), or the number of pages of documentation produced. Example: size, reuse, functionality, modularity, correctness, coupling, cohesiveness etc.

1.8.2 Process Metrics

Process metrics are a measure of the software development process, such as overall development time, type of methodology used, or the average level of experience of the programming staff. Example: time, effort, number of requirement changes, number of coding faults found etc.

1.8.3 Resource Metrics

Resource metrics is used to measure the inputs to the software engineering activity, such as hardware, software, documentation, knowledge, and human resources. Example: speed, memory size, price, communication level of team etc.

1.9 OBJECT ORIENTED PARADIGM

Object oriented design and development are popular concepts in today's software development. Object oriented design and development focuses on objects as the prime agents involved in the computation; each class of data and related operations are collected into a single system entity. There are various characteristics of the object oriented paradigm which differentiates them from the traditional structured approach [16, 19].

1.9.1 Localization

Localization is the characteristics the software that indicates the manner in which the information is concentrated within a program. As the conventional methods localized the information around functions, which are typically implemented as procedural modules, in OO context information is concentrated by encapsulating both data and process within the bounds of a class or objects. Since the class is the basic unit of an object oriented system, localization on based on objects and metrics are applied to classes or objects as the complete entity.

1.9.2 Encapsulation

Encapsulation encompasses the responsibilities of the class, including its attributes (and other classes for aggregate objects) and operations, and the state of the class as defined by specific attribute value. The attributes and methods are encapsulated into a single unit called class. Encapsulation encourages measurement at a higher level of abstraction.

1.9.3 Information Hiding

A well designed OO system encourages Information Hiding. It suppresses the operational details of the program component. Only the information necessary to access the component is provided to those other components that wish to access it.

1.9.4 Inheritance

Inheritance is a mechanism that enables the responsibilities of one object to be propagated to other object. Inheritance occurs throughout all levels of class hierarchy.

1.9.5 Abstraction

Abstraction is the process that enables the designer to focus on the essential details of a program component (either data or process) with the little concern for the lower level details. Abstraction is a relative concept. As we move to the higher level of abstraction we ignore more and more details i.e. we provide a more general view of the concept and vice versa.

1.9.6 Polymorphism

Polymorphism refers to a programming language's ability to process objects differently depending on their data type or class. Polymorphism allows two or more objects respond to the same message. An analogy of polymorphism to daily life is how students response to a school bell. Every student knows the significant of the bell. When the bell (message) rings, however, it has its own meaning to different students (objects). Some students go home, some go to the library, and some go to other classes. Every student responds to the bell, but how they response to it might be different.

1.10 WHY NOT CONVENTIONAL METRICS FOR OBJECT ORIENTED SYSTEM?

Following are the reasons why the conventional metrics cannot be used for object-oriented system:

- ✓ The key difference between object oriented programming and structured programming is that the former identifies the object in the applications, while the latter models the application as a set of functions.

- ✓ Since the traditional software metrics aims at the procedure oriented development and it cannot fulfill the requirements of the object-oriented software.
- ✓ There are no metrics for the concepts like encapsulation, inheritance, coupling, cohesion etc, so conventional metrics cannot be used for the OO paradigm.

It is because of all these limitations a set of new software metrics adapted to the characteristics of object technology was proposed. Accordingly object oriented metrics then became an essential part of object technology as well as good software engineering.

1.11 THESIS ORGANIZATION

Chapter 2 gives an overview of object oriented software metrics classified according to the different phases of software development. Chapter 3 discusses the use cases and use case diagram in general and discusses their various viewpoints. Chapter 4 discusses the various inadequacies of the object oriented software metrics with main focus on the analysis phase object oriented software metrics. Chapter 5 discusses the proposed method for effort estimation. Chapter 6 discusses the various guidelines for the various environmental and technical factors. Chapter 7 concludes the work and discusses the future scope.

Chapter 2

Classification Framework for OO Metrics

Metrics are key components of any engineering discipline, software engineering is no exception. Software metrics is often used to refer to broad range of measurements for computer software. Since the 1950, when software engineering were greatly researched on, software metrics were developed continuously and became an important research area in the field of software engineering. The need for software metrics is now fully recognized by the software engineering community.

In recent years, OO technologies have emerged as a dominant software engineering practice and are often heralded as the silver bullet for solving software problems. OO development has proved its value for systems that must be maintained and modified. OO software engineering adopts a systemic life cycle where system development is divided into four phases [19]: Analysis, Design, Implementation and Testing.

- ✓ **Analysis:** The process of analysis is intensified and focused specifically on software. To understand the nature of the program(s) to be built, the software engineer ("analyst") must understand the information domain for the software, as well as required function, behavior, performance, and interface.

- ✓ **Design:** The design process translates the requirements into a representation of the software that can be assessed for quality before coding begins.

- ✓ **Implementation:** The design must be translated into a machine-readable form. The implementation step performs this task.

- ✓ **Testing:** Once code has been generated, program testing begins. The testing process focuses on the logical internals of the software, ensuring that all statements have been tested, and on the functional externals; that is, conducting tests to

uncover errors and ensure that defined input will produce actual results that agree with required results.

As OO technologies has some new characteristics, such as data abstraction, encapsulation, inheritance, polymorphism, information hiding and reuse, traditional software metrics do not readily lend themselves to the OO notions. Therefore, new ways of measuring OO software are largely researched on.

After many years of research, many OO metrics are proposed, each targeting at a specific phase of the OO development life cycle.

2.1 ANALYSIS PHASE

In Object Oriented Analysis, use case models are often employed to capture the functional requirements of a software project. There is a fair amount of work on describing and formalizing use cases, but there is a lot less on deriving metrics from use cases. The aim and objective of measuring use cases is not to determine the quality and design of the use case model, but as a guide for estimation of the resources that might be needed for the software project. This information is important for project manager to plan ahead the amount of man-power and system resource that is needed. Effective software project estimation is one of the most challenging and important activity in software project life-cycle.

One example of better known estimation metrics for the Analysis Phase is development by Gustav Karner of Objectory AB (now known as Rational Software) in 1993, known as the "*use case points method of software estimation*". It uses use case model, accompanied with environment and technical factors, to estimate the resources that is needed for the software project. The results of use case points method indicates that it has potential to be a reliable source of estimation, and it can have a strong impact on estimating the size of software development projects.

Use case point is used to estimate the size and schedule of software development projects. Early estimate of effort based on use cases can be made when there is some understanding

of the problem domain, system size and architecture at the stage at which the estimate is made. As estimation of a project is important, this information can help project managers to estimate the resources like software developers, testers that are needed the project during the early phases.

Karner is one of the better use case metrics, but the values generated are estimated and only serves as a guide. Even though, it is difficult to derive a precise and accurate use case metrics, we should not avoid use cases for estimation and rely instead on the analysis and design realization that emerges, because relying on design and implementation phase metrics would delay the ability to make early estimates and will not be satisfactory for a project manager. It is better for the project manager to obtain estimates early for their planning purposes, and then refine them at the later phases.

2.1.1 Karner's Approach

This section gives a brief overview of the steps in the use case points method as described in [9]. First, categorize the actors in the use case model as simple, average or complex and calculate the total unadjusted actor weight (UAW) by counting the number of actors in each category, multiplying each total by its specified weighting factor, and then adding the points. Next, categorize the use cases as simple, average or complex, depending on the number of transactions, including the transactions in alternative flows. Then the unadjusted use case weights (UUCW) are calculated by counting the number of use cases in each category, multiplying each category of use case with its weight and adding the products. The UAW is added to the UUCW to get the *unadjusted use case points (UUPC).*

Next, the use case points are adjusted based on the values assigned to a number of technical factors as in table 2.1 and environmental factors as in table 2.2 [26].

Table 2.1 Technical Complexity factors

Factor	Name	Weight
T1	Distributed System	2
T2	Performance Objectives	2
T3	End user efficiency	1
T4	Complex Internal Processing	1
T5	Reusable Code	1
T6	Installation Ease	0.5
T7	Easy use	0.5
T8	Portable	2
T9	Easy to change	1
T10	Concurrent	1
T11	Security objectives	1
T12	Direct access to third parties	1
T13	User training facilities	1

Table 2.2 Environmental Complexity factors

Factor	Name	Weight
E1	Familiarity with project	1.5
E2	Application experience	0.5
E3	Objects-oriented Experience	1
E4	Lead analyst capability	0.5
E5	Motivation	1
E6	Stable requirements	2
E7	Part-Time Staff	-1
E8	Difficult programming language	-1

Each factor is assigned a value between 0 and 5 depending on its assumed influence on the overall project.

The Technical Factor (TCF) is calculated multiplying the value of each factor (T1 – T13) in Table 1 by its weight and then adding all these numbers to get the sum called the *TFactor*. Finally, the following formula is applied:

$$TCF = 0.6 + (.01 * Tfactor)$$

The Environmental Factor (EF) is calculated accordingly by multiplying the value of each factor (E1 – E8) in Table 2 by its weight and adding all the products to get the sum called the *Efactor*. The formula below is applied:

$$EF = 1.4 + (-0.03 * Efactor)$$

The *Adjusted use case points* is calculated as:

$$UCP = UUCP * TCF * EF$$

Finally, the UCP is multiplied by a historically collected figure representing productivity, such as a factor of 20 staff hrs per use case point, to arrive at a project estimate [9]. The result is an estimate of the total number of person hours required to complete the project.

2.2 DESIGN PHASE

OO Design is a continuity of OO analysis. It extents and updates the analysis model, by including class design, interactions between subsystems as well as some data storage attributes that may be needed. Unlike in the analysis phase when our measurements aim to obtain estimates for the software project, measurements in the design phase aims to indicate the quality of the design and detect possible design faults and error. There are three design metrics suite which are most popular in the industry. They are

- ✓ Chidamber and Kemerer Metrics Suite (CK Suite),
- ✓ Abreu Metrics Suite (MOOD Suite),

- ✓ Lorenz and Kidd Metrics Suite (LK Suite)

2.2.1 Chidamber and Kemerer Metrics Suite - *Design Metrics*

In aim of measuring the key notions of OO software, Chidamber and Kemerer, in 1994, developed a set of six metrics to identify certain design and code traits in OO software, like inheritance, coupling and cohesion etc [20]. Out of the six metrics developed, three are classified as design metrics and remaining as code metrics.

The three design metrics in the CK OO metrics suite are Depth of Inheritance Tree (DIT), Number of Children (NOC) and Lack of Cohesion in methods (LCOM) and can be summarized as:

2.2.1.1 Depth of Inheritance Tree *(DIT)*

This metric measures the maximum level of the inheritance hierarchy of a class. The root of the inheritance tree inherits from no class and has a DIT count of 0. CK suggest that DIT can be used to indicate the complexity of the design, potential for reuse.

2.2.1.2 Number of Children *(NOC)*

This metric counts the number of immediate subclasses belonging to a class. NOC was intended to indicate the level of reuse in a system and a possible indicator of the level of testing required.

2.2.1.3 Lack of Cohesion in Methods *(LCOM)*

This metric purports to measure the lack of cohesion in the methods of a class. It is based on the principle that a variable occurring in many methods of a class causes that class to be less cohesive than one where the same variable is used in few methods of the class. CK view a lack of cohesiveness as undesirable as it is against encapsulation. Lack of cohesion could imply that the class should probably be split into two or more subclasses.

2.2.2 Metrics for Object Oriented Design (MOOD Suite)

Unlike CK metrics, MOOD, as the name suggests, is a metrics suite that targeted specifically to obtain measurements for the design phase. The emphasis behind the development of the metrics is on the features of OO design, namely inheritance, encapsulation and coupling [21].

In the first proposed of MOOD, eight metrics were in the suite. It was later refined to six, the two metrics removed are Clustering Factor and Reuse factor. Each of the six metrics in MOOD suite is expressed as a quotient where the numerator is the actual use of a particular mechanism (i.e. inheritance, information hiding, coupling and polymorphism) in the system being measured and the denominator is the maximum possible use of the same mechanism. The value of each metric would then range from 0 (total absence), to 1 (maximum possible presence). The six metrics are summarized as:

2.2.2.1 Attribute Hiding Factor *(AHF)*

This metric is the ratio of hidden (private and protected) attributes to total attributes and is proposed as a measurement of encapsulation and information hiding.

2.2.2.2 Method Hiding Factor *(MHF)*

This metric is the ratio of the total inherited methods and total methods defined. Similar to AHF, it is proposed as a measurement of encapsulation and information hiding.

2.2.2.3 Coupling Factor *(COF)*

COF is defined as the ratio of the maximum possible number of couplings in the system to the actual number of couplings not imputable to inheritance. Abreu views coupling as increasing complexity, reducing potential reuse and limiting understandability and maintainability.

2.2.2.4 Polymorphism Factor *(POF)*

POF is defined as the ratio of the actual number of possible different polymorphic situation for a class to the maximum number of possible distinct polymorphic situations for the class. Polymorphism arises from inheritance, and Abreu claims that in some cases, overriding methods reduce complexity, so increasing understandability and maintainability.

2.2.2.5 Attribute Inheritance Factor *(AIF)*

AIF is defined as the ratio of the sum of inherited attributes in all classes of the system under consideration to the total number of available attributes for all classes. AIF is suggested to express the level of reuse in a system. It is claimed, however, that too much reuse causes deterioration in understandability and testability.

2.2.2.6 Method Inheritance Factor *(MIF)*

MIF is defined as the ratio of the sum of inherited methods in all classes of the system under consideration to the total number of available methods for all classes. Just as for AIF, MIF is suggested to express the level of reuse in a system. It could also claim to be an aid to assessment of testing needed.

2.2.3 Lorenz and Kidd Metrics Suite

There were many metrics suggested by LK. However, the ten metrics described give a fair cross-section of the broad areas covered. Unlike the CK metrics, most of the LK metrics are direct metrics, and include more directly countable measures, e.g., the Number of Methods *(NM)* metric, and the Number of Variables *(NV)* metric. Although relatively simple to collect, doubt can be cast on the usefulness of such metrics because they give only a limited insight into the architecture of the system under investigation. The ten metrics are as under [16]:

2.2.3.1 Number of Public Methods *(PM)*

This simply counts the number of public methods in a class. According to LK, this metric is useful as an aid to estimating the amount of work to develop a class or subsystem.

2.2.3.2 Number of Methods *(NM)*

The total number of methods in a class counts all public, private and protected methods defined. LK suggest this metric as a useful indication of the classes which may be trying to do too much work themselves; i.e., they provide too much functionality.

2.2.3.3 Number of Public Variables per class *(NPV)*

This metric counts the number of public variables in a class. LK consider the number of variables in a class to be one measure of its size. The [act that one class has more public variables than another might imply that the class has more relationships with other objects and, as such, is more likely to be a key class, i.e., a central point of co-ordination of objects within the system.

2.2.3.4 Number of Variables per class *(NV)*

This metric counts the total number of variables in a class. The total number of variables metric includes public, private and protected variables. According to LK, the ratio o[private and protected variables to total number of variables indicates the effort required by that class in providing information to other classes. Private and protected variables are therefore viewed merely as data to service the methods in the class.

2.2.3.5 Number of Methods Inherited by a subclass *(NMI)*

This metric measures the number of methods inherited by a subclass. No mention is made as to whether that inheritance is public or private. In a language such as C++, we have to consider the possibility that the inheritance may be private, Then, any classes using methods from a subclass would not necessarily have access to all of the inherited methods.

2.2.3.6 Number of Methods Overridden by a subclass *(NMO)*

A subclass is allowed to re-define or override a method in its superclass with the same name as a method in one of its superclasses. According to LK, a large number of overridden methods indicate a design problem, indicating that those methods were overridden as a design afterthought. They suggest that a subclass should really he a specialization of its superclasses, resulting in new unique names for methods.

2.2.3.7 Number of Methods Added by a subclass *(NMA)*

According to LK, the normal expectation for a subclass is that it will further specialize (or add) methods to the superclass object. A method is defined as an added method in a subclass if there is no method of the same name in any of its superclasses.

2.2.3.8 Average Method Size *(AMS)*

The average method size is calculated as the number of non-comment, nonblank source lines *(NCSL)* in the class, divided by the number of its methods. *AMS* is clearly a size metric, and would be useful for spotting outliers, i.e., abnormally large methods.

2.2.3.9 Number of times a Class is Reused *(NCR)*

The definition of *NCR* given by LK is somewhat ambiguous. We assume the metric is intended to count the number of times a class is referenced (i.e., reused) by other classes. In this sense, we could view reuse in a similar way to coupling. We could then consider *NCR* as a measure of the extent of inter-class communication, and in this respect, a high value for *NCR* is undesirable.

2.2.3.10 Number of Friends of a class *(NF)*

This metric measure, for each class, the number of friends of that class. Friends allow encapsulation to be violated; and as such should be used with care. A high number or friends within a class could indicate a potential design flaw, an oversight in design, which has filtered through to the coding stage; we note in passing that friends are a

concept specific to the C++ language. *NF* is a measure of class coupling, since friends may rely on a particular class (or classes) to operate properly.

2.3 IMPLEMENTATION PHASE

Code metrics are used as measurements for the implementation phase and can be obtained by looking into the program codes. CK metric suite defines three code metrics specifically for OO program codes [20].

2.3.1 Chidamber and Kemerer Metrics Suite - *Code Metrics*

The three code metrics proposed by CK are given below -

2.3.1.1 Weighted Methods per Class (*WMC*)

The WMC is a count of the methods implemented within a class or the sum of the complexities of the methods (method complexity is measured by cyclomatic complexity).

2.3.1.2 Coupling between objects (*CBO*)

Coupling between Object Classes *(CBO)* is a count of the number of other classes to which a class is coupled.

2.3.1.3 Response for a Class (*RFC*)

The *RFC* is the count of the set of all methods that can be invoked in response to a message to an object of the class or by some method in the class.

2.4 TESTING PHASE

Today's software applications impose new challenges for software testing. Some of the commonly known challenges are complex application scenarios, extensive third party integration and increased security concerns. These factors inherently make testing of application for more complex and critical than traditional software testing.

The main purpose of metrics in testing phase is to find the total number of test cases, to find the effort required to do testing. A number of metrics are proposed for this but no one is popular in the industry.

Suresh Nageswaran has proposed a metric for the test phase known as "Test Effort Estimation using use cases" [20]. Test Effort Estimation provides estimation for test effort based on the use case model. This metric use "*use case points method of software estimation*" given by Gustav Karner as the basis.

Chapter 3

Use Case and Use Case Diagram

No system exists in isolation. Every interesting system interacts with human or automated actors that use that system for some purpose, and those actors expect that system to behave in predictable ways. A use case specifies the behavior of a system or a part of a system and is a description of a set of sequences of actions, including variants that a system performs to yield an observable result of value to an actor [7].

We can apply use cases to capture the intended behavior of the system we are developing, without having to specify how that behavior is implemented. Use cases provide a way for the developers to come to a common understanding with the system's end users and domain experts. In addition, use cases serve to help validate the architecture and to verify the system as it evolves during development. These use cases are realized by collaborations whose elements work together to carry out each use case [14].

A well-designed house is much more than a bunch of walls thrown together to hold up a roof that keeps out the weather. When we work with our architect to design our house, we'll give strong consideration to how we'll use that house. If we like entertaining, we'll want to think about the flow of people through our family room in a way that facilitates conversation and avoids dead ends that result in bunching. As we think about preparing meals for our family, we'll want to make sure that our kitchen is designed for efficient placement of storage and appliances. Even plotting the path from our car to the kitchen in order to unload groceries will affect how we eventually connect rooms to one another. If we have a large family, we'll want to give thought to bathroom usage. Planning for the right number and right placement of bathrooms early on in the design will greatly reduce the risk of bottlenecks in the morning as our family heads to school and work. If we have teenagers, this issue has especially high risk, because the emotional cost of failure is high.

Reasoning about how we and our family will use our house is an example of use case-based analysis. We consider the various ways in which we'll use the house, and these use cases drive the architecture. Many families will have the same kinds of use cases - we use houses to eat, sleep, raise children, and hold memories. Every family will also have its own special use cases or variations of these basic ones. The needs of a large family, for example, are different from the needs of a single adult just out of college. It's these variations that have the greatest impact on the shape of our final home [7].

One key factor in creating use cases such as these is that we do so without specifying how the use cases are implemented. For example, we can specify how an ATM system should behave by stating in use cases how users interact with the system; we don't need to know anything about the inside of the ATM at all. Use cases specify desired behavior; they do not dictate how that behavior will be carried out. The important point is that it lets us (as an end user and domain expert) communicate with our developers (who build systems that satisfy your requirements) without getting hung up on details.

In the UML, all such behaviors are modeled as use cases that may be specified independent of their realization. A use case is a description of a set of sequences of actions, including variants that a system performs to yield an observable result of value to an actor. There are a number of important parts to this definition.

3.1 USE CASE

"Use Case is a sequence of transactions in a system, whose task is to yield a measurable value to an individual actor of the system".

----Ivar Jacobson's

The use case construct is used to define the behavior of a system or other semantic entity without revealing the entity's internal structure. Graphically, a use case is rendered as an ellipse. Every use case must have a name that distinguishes it from other use cases. A name is a textual string.

Figure 3.1 Use Case

For example, Manage Book is a use case as shown in figure 3.1 whose purpose is to order new books, put new books in to the shelf and to report damaged books. The use case gives us no idea about how this purpose is carried out. "Manage Book" is the name of the use case.

3.1.1 Viewpoints about Use Cases

The following are the various viewpoints about the use cases [7, 8, 14, 32]-

- ✓ A use case describes a set of sequences, in which each sequence represents the interaction of the things outside the system (its actors) with the system itself (and its key abstractions). These behaviors are in effect system-level functions that we use to Visualize, specify, construct, and document the intended behavior of our system during requirements capture and analysis. A use case represents a functional requirement of your system as a whole.

- ✓ A use case may have variants. In all interesting systems, we'll find use cases that are specialized versions of other use cases, use cases that are included as parts of other use cases, and use cases that extend the behavior of other core use cases. We can factor the common, reusable behavior of a set of use cases by organizing them according to these three kinds of relationships namely generalization, include, and extend relationships.

- ✓ A use case carries out some tangible amount of work. From the perspective of a given actor, a use case does something that's of value to an actor, such as calculate a result, generate a new object, or change the state of another object.

- ✓ We can apply use cases to our whole system. We can also apply use cases to part of our system, including subsystems and even individual classes and interfaces. In each case,

these use cases not only represent the desired behavior of these elements, but they can also be used as the basis of test cases for these elements as they evolve during development. Use cases applied to subsystems are excellent sources of regression tests; use cases applied to the whole system are excellent sources of integration and system tests.

3.2 ACTORS

An actor represents a coherent set of roles that users of use cases play when interacting with these use cases [8, 29]. Typically, an actor represents a role that a human, a hardware device, or even another system plays with a system. An instance of an actor, therefore, represents an individual interacting with the system in a specific way. Although we'll use actors in our models, actors are not actually part of the system. They live outside the system.

Actors are rendered as stick figures. We can define general kinds of actors (such as Library System User) and specialize them (such as Librarian) using generalization relationships as shown in figure below.

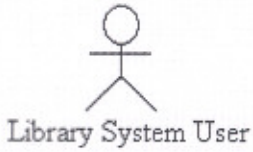


Figure 3.2 Actor

For example, *Library System User* is a simple actor as shown in figure 3.2 while *Librarian* is a specialized actor derived form *Library System User* as shown in figure 3.3. The specialized actor, *Librarian*, have the entire feature that exist in Library System user but have some additional features.

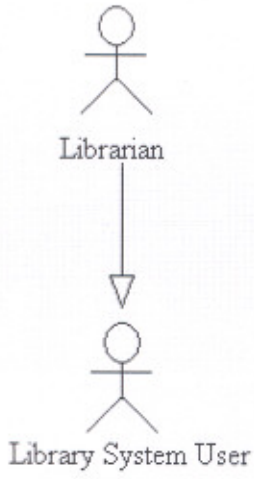


Figure 3.3 Generalized Actor

Actors may be connected to use cases only by association. An association between an actor and a use case indicates that the actor and the use case communicate with one another, each one possibly sending and receiving messages.

3.3 RELATIONSHIPS IN USE CASES

In all interesting systems, we'll find use cases that are specialized versions of other use cases, use cases that are included as parts of other use cases, and use cases that extend the behavior of other core use cases. We can factor the common, reusable behavior of a set of use cases by organizing them according to these three kinds of relationships namely generalization, include, and extend relationships [7, 8].

3.3.1 Generalization

It is relationship between a more general element and a more specific element that is fully consistent with the first element and that adds additional information. Generalization among use cases is just like generalization among classes i.e. the child use case inherits the behavior and meaning of the parent use case; the child may add to or override the behavior of its parent; and the child may be substituted any place the parent appears.

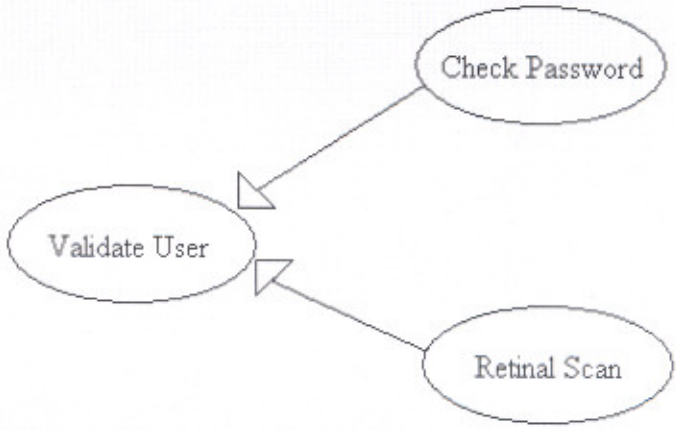


Figure 3.4 Generalization among use cases

For example, in figure 3.4 we have *Validate User* which is responsible for verifying the identity of the user. We have two specialized children of this use case namely *Check password* which validates user by checking his/her login id and password and *Retinal Scan* which validates user by his/her retinal scan. Both of these specialized use cases behaves just like *Validate User* and may be applied anywhere *Validate User* appears yet both add their own behavior.

3.3.2 Include

An include relationship from use case A to use case B indicates that an instance of the use case A will also include the behavior as specified by B. We use an include relationship to avoid describing the same flow of events several times, by putting the common behavior in a use case of its own (the use case that is included by a base use case). The include relationship is essentially an example of delegation - we take a set of responsibilities of the system and capture it in one place (the included use case), then let all other parts of the system (other use cases) include the new aggregation of responsibilities whenever they need to use that functionality. We can think of include as the base use case pulling behavior from the supplier use case.

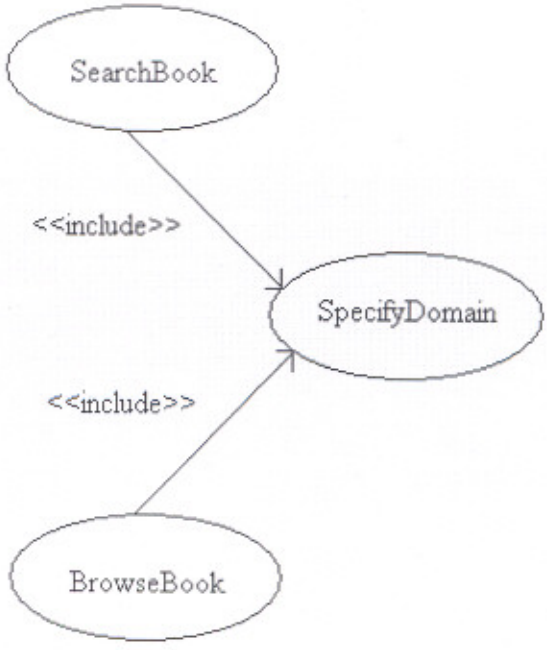


Figure 3.5 Include among use cases

For example, figure 3.5 shows an include relationship form *SearchBook* to *SpecifyDomain* and from *BrowseBooks* to *SpecifyDomain*. Here *SpecifyDpmain* is the common behavior of both *SearchBook* and *BrowseBooks*.

3.3.3 Extend

An extend relationship, i.e. a generalization with the stereotypes <<extend>>, defines that a use case may be extended with some additional behavior defined in another use case. The extend relationship includes both a condition for the extension and a reference to an extension point in the related use case i.e. a position in the use case where additions may be made. Once an instance of a use case reaches an extension point to which an *extend* relationship is referring, the condition of the relationship is evaluated. If the condition is fulfilled, the sequence obeyed by the use-case instance is extended to include the sequence of the extending use case. We use an extend relationship to model the part of a use case the user may see as optional system behavior. In this way, we separate optional behavior from mandatory behavior.

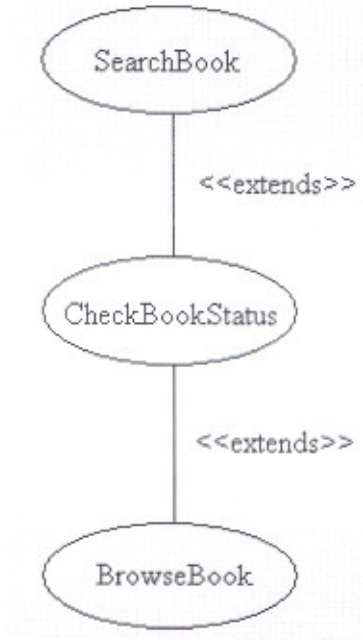


Figure 3.6 Extend among use cases

For example, figure 3.6 shows the <<extends>> relationship between *CheckBookStatus* and *BrowseBooks*, and between *CheckBookStatus* and *SearchBook*. The functionality provided by *CheckBookStatus* is not executed every time but it's the optional behavior which is executed only when user of *SearchBook* finishes searching the book and now wants to check his availability. User can't directly check availability through *CheckBookStatus*.

3.3.4 Association

An association is a structural relationship that describes a set of links, a link being a connection among objects. Actor may be connected to use case only by association. An association between an actor and a use case indicates that the actor and the use case communicate with one another, each one possibly sending and receiving messages.

3.4 USE CASE DIAGRAMS

Use case diagrams are one of the five diagrams in the UML for modeling the dynamic aspects of systems (activity diagrams, statechart diagrams, sequence diagrams, and

collaboration diagrams are four other kinds of diagrams in the UML for modeling the dynamic aspects of systems). Use case diagrams describe what a system does from the standpoint of an external observer. The emphasis is on *what* a system does rather than *how*. A use-case diagram is used to communicate the high-level functions of the system and the system's scope. By just looking at use-case diagram, we can easily tell the functions that our system provides [24, 25].

The main purpose of the use-case diagram is to help development teams visualize the functional requirements of a system, including the relationship of "actors" (human beings who will interact with the system) to essential processes, as well as the relationships among different use cases.

A use case diagram is a diagram that shows a set of use cases and actors and their relationships.

3.4.1 Contents of Use Case Diagrams

Use Case diagram commonly contain [7, 8]:

- ✓ Use cases
- ✓ Actors
- ✓ Extend, include, generalization, and association relationships

Use Case diagrams may also contain notes and packages. Packages are used to group elements of our model into larger chunks.

3.4.2 Example of Use Case Diagram

A use case diagram in Figure 3.7 shows the actors of the Library System and their use cases. Actors are *Librarians* and *Borrowers*. The actor *LibrarySystemUser* is a generalization of *Librarian* and *Borrower*. *Librarians* use the Library System to manage the book catalogue and the loan records. *Librarians* only need to inform to the Library System when books are checked into and checked out of the system to be able to manage

loan records. Thus, the use cases *BorrowBook* and *ReturnBook*, associated with *Librarian*, are created. *LibrarySystemUsers* can connect to the system, list the books borrowed by a library user and search for books by author, title, year or a combination of these. Thus, the use cases associated with *LibrarySystemUsers* are *ConnectToSystem*, *ListBooksBorrowedByUser* and *SearchBook*. *ConnectToSystem* is considered as a use case since a *LibrarySystemUser* can login to the system just to check his/her password. *Borrowers* can browse the book catalogue without specifying any condition. Thus, the use case *BrowseBooks* is associated with *Borrowers*. Search and browse operations can be repeated over the result of the last search or browse operation.

The Library System must guarantee that only registered users can login to the system. Further, the system must guarantee that *Borrowers* can only perform services associated with *Borrowers*, and that *Librarians* can only perform services associated with *Librarians*.

Books can be selected while users are searching or browsing the book catalogue. Once a book is selected, users can check its availability. The use case *CheckBookStatus* is modeled as an extension of *SearchBook* and *BrowseBooks*. Indeed, a book must be selected by one of these operations before the user can check its status.

Some use cases have similar features in their behavior. For instance, *BrowseBooks* and *SearchBook* are both use cases where books can be specified. Thus, a new use case called *SpecifyBook* has been created to model this shared behavior. Unidirectional associations are created to model the <<extends>> relationship between *SpecifyBook* and *BrowseBooks*, and between *SpecifyBook* and *SearchBook*. Similarly, *SpecifyDomain* is also a common behavior of *SearchBook* and *BrowseBook*.

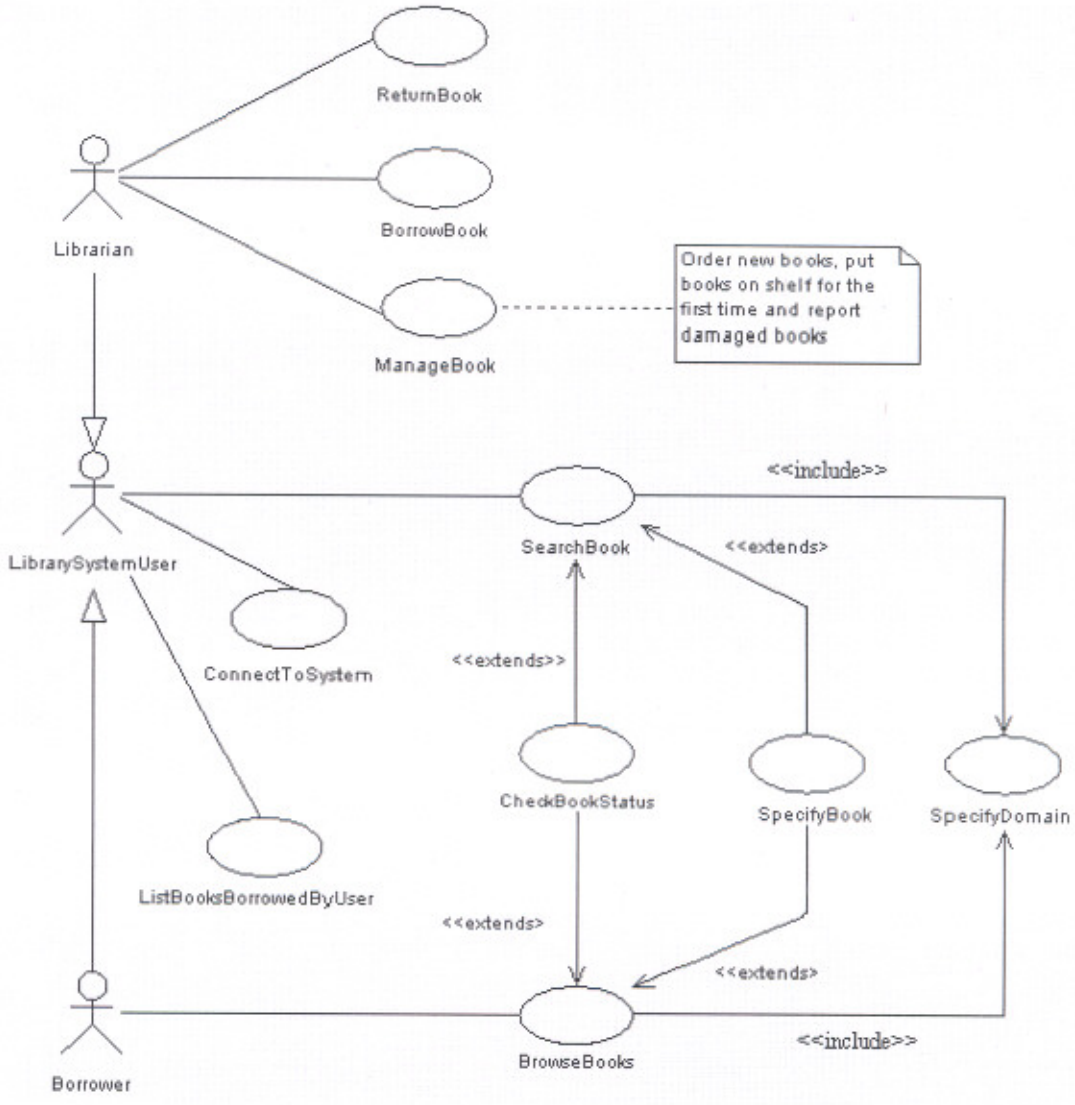


Figure 3.7 Use Case diagram

3.5 COMMON USES OF USE CASE DIAGRAM

We can apply use case diagrams to model the static use case view of a system. This view primarily supports the behavior of a system - the outwardly visible services that the system provides in the context of its environment.

Use case diagrams are helpful in following areas [10]:

- ✓ **Determining features (requirements)**

  New use cases often generate new requirements as the system is analyzed and the design takes shape.

- ✓ **Communicating with clients**

  Their notational simplicity makes use case diagrams a good way for developers to communicate with clients.

- ✓ **Effort Estimation**

  An emerging use of the Use Case is to estimate the effort required to build the system. One such approach for calculating the development effort is suggested by Gustav Karner as *"Karner's Approach"* and another approach to calculate the test effort required to test the system as is suggested by Suresh Nageswaran's *"Nageswaran's Approach"*.

Chapter 4

Inadequacies in the OO Metrics

All of the metrics suites, discussed in previous chapters, are the most popular one as far as object oriented approach is concerned but all these suites in all four development phases is having some inadequacies or inconsistencies which are discussed in the following sections.

4.1 INADEQUACIES IN KARNER'S APPROACH

The *use case point method of software estimation* is having a number of inadequacies which makes this approach less popular as compared to traditional metrics for effort estimation like *Albrecht Function Point Analysis Method* developed by Allan Albrecht of IBM.

All these inadequacies are listed below:

- ✓ **Related to Use Case Structure**

  The structure of use case is not uniform. It not only differs from company to company but also from project to project and from person to person with in the same company. So the final estimation has significant variance according to structure of Use Case documents. Use cases can differ in complexity, both explicitly as written, and implicitly in the required realization.

- ✓ **Related to Include, Extend, Generalized Use Cases and Generalized Actors**

  Karner said that only concrete actors and use cases are counted in [9]. This means that generalized actors, include use cases, extend use cases and generalized use cases should not be taken in to consideration while making the effort estimation.

But Bente Anda have a different opinion in there practical experience. In some use cases they found include and extended use cases has essential functionalities and reducing them will reduce steps and hence the estimation [3].

Also, generalization among use cases and among actors behaves exactly like generalization among classes, where the specialized use case inherits the behavior from the generalized use case. But it may be possible that a specialized actors or a specialized use case posses more functionality then generalized one. So in that case what to do?

- ✓ **Related to the Technical Factors and Environmental Factors**

The technical factors involved in developing system are assessed, similar to the Function Points. But having a view on the list of these factors, it is clear that there is no factor which is related to the testing i.e. Karner is not including the testing effort in his approach. But Estimating effort means estimating the amount of work to complete a project which includes both development and testing. The list of environmental factors also seems to be incomplete as there are several factors like working environment which should also be taken in to consideration because long work hours and bad work culture tend to decrease the quality and quantity of work being done in unit time.

- ✓ **Related to the Weight Assigned to Technical and Environmental Factors**

Karner has weighted the technical and environmental factors based on the experiences in the Objectory [9]. Also, Bente Anda has said that there are several other methods for effort based estimation which differs in the list of technical and environmental factors and their corresponding weights [1, 2, 15]. So, it is clear that the industries have modified weights assigned to the technical and environmental factors according to them i.e. the weights are not generalized. They are specialized to Objectory.

- ✓ **Related to the Rating Criteria of the Technical Factors and Environmental Factors**

  Karner suggested to rate the technical factors and environmental factors on the scale of 0, 1, 2, 3, 4, 5 [9]. 0 means that it is irrelevant and 5 means it is essential. But he does not gives any criteria which tells how to rate these factors making it impossible to tell when to assign 0, 1, 2, 3, 4, and 5. So the final result varies from person to person.

4.2 INADEQUACIES IN CK AND MOOD METRICS SUITE

CK and MOOD suite are the widely accepted suites in design and implementation phases but besides their wide acceptance they have some inadequacies and inconsistencies which are discussed in [21].

4.3 INADEQUACIES IN NAGESWARAN'S APPROACH

This metric proposed by Suresh Nageswaran's is using the work of Gustav Karner as the base. So there are some inadequacies which are common in both i.e. inherited from previous work and there are some inadequacies which are added by Suresh Nageswaran in his work.

The inadequacies which are inherited from *Use Case Point Method of Effort Estimation* are:

- ✓ Related to use case structure.
- ✓ Related to include, extend, generalized use cases and generalized actors.
- ✓ Related to the rating criteria of the technical factors and environmental factors.
- ✓ Related to the weight assigned to technical factors.

The inadequacies which are added to this are:

- ✓ **Related to Environment Factors**

  Suresh Nageswaran does not consider any environment factor in finding the test effort but factors like familiarity with project domain, testing experience, working environment, etc., play a major role in the final outcome. If the testing team has previous experience of testing on same project domain then testing effort required will be much less than the testing effort resulted from his approach.

Thus, all the popular object oriented metrics suites in all four software development phases are having inadequacies, which is the only reason why these metrics are not as popular as traditional one. All these suites are used by the industries but with some modification. The question is why modification? Can't we propose metrics that are general like *Albrecht Function Point Analysis Method*?

Chapter 5

Proposed Framework for Effort Estimation

In Object Oriented Analysis, use case models are often employed to capture the functional requirements of a software project; it makes sense to base the more difficult task of estimation of size and resources on them, as opposed to any other technique such as function points, lines of code etc. This information is important for project manager to plan ahead the amount of man-power and system resource that is needed. Effective software project estimation is one of the most challenging and important activity in software project life-cycle. Any prediction less or more will affect the project a lot.

The proposed framework includes several steps that need to be performed to estimate the effort. The steps are given below:

Step 1: Identify Actors

First Step is to identify the actors in the use case diagram. There are two types of actors, namely general actors and specialized actors, which are derived from general actors using generalization relationship. If two actors have 80 percent in common put them in generalization relationship and count them only once. This will increase the accuracy of the estimate.

Step 2: Determine the UAW (Unadjusted Actor weight)

The next step is to classify all the actors in to following classification. Table 5.1 will gives us clear idea of how to classify the actors. Second column gives us the criteria for making decision which type of actor falls in which category. The last column provides the factor of complexity.

Table 5.1 Classifying Actors

Classification	Criteria for classification	Value/Factor
Simple	Those which communicate to system through API.	1
Average	Those which have following properties • Actors who are interacting the system through some protocol (HTTP,FTP, or probably some user defined protocol) • Actor which are data store(Files, RDBMS)	2
Complex	Those which interact normally through GUI.	3

The total unadjusted actor weights (UAW) is calculated by counting how many actors there are of each kind, multiplying each total by its corresponding weighting factor, and adding up the products.

Step 3:Identify Use cases

There are four types of use cases - simple use cases, extend use cases, include use cases and generalized use cases. If two use cases have 80 percent functionality in common put them in generalization relationship and count them only once. This will increase the accuracy of the estimate.

Although Karner recommended that included and extending use cases should not be counted, the functionality described in these use cases must still be implemented [18]. If these use cases contain much of the essential functionality, it is necessary to include them in the counts. However, there is a danger that when writing included and extending use cases one may be more concerned with identifying opportunities for reuse, than with analyzing the problem and describing requirements [10].

Bente Anda. describe how omitting extending and included use cases resulted in an estimate that was much lower than the actual effort by first omitting these use cases

and obtaining an estimate that was much lower than the actual effort and then counting these use case and obtaining an estimate that was much closer to the actual effort[1, 3].

It seems difficult to form a precise rule for when to count extending and included use cases. But the findings of Bente Anda. make it clear that one should not always follow Karner's recommendations. In some projects, extending and included use cases contain a lot of the important functionality [1, 2, 3]. So, if in doubt, I believe that it is better to count extending and included use cases to be sure that all the functionality is sized/counted, in order to avoid underestimation.

Step 4: Determine number of UUCW (Unadjusted Use Case Weight)

The second step is to count use cases. Table 5.2 will gives us clear idea of how to classify the use cases. Second column gives us the criteria for making decision which type of use case falls in which category depending on number of scenarios and number of transactions. The last column provides the factor of complexity.

Table 5.2 Classifying Use Cases

Use case Type	Criteria for classification	Value/Factor
Simple	Greater than or equal to 3 transactions	5
Average	Between 4 to 7 transactions	10
Complex	Greater than 7 transactions	15

The total unadjusted use case weights (UAW) is calculated by counting how many use cases there are of each kind, multiplying each total by its corresponding weighting factor, and adding up the products.

Step 5:Determine Total UUCP (Unadjusted Use Case Point)

$$Total\ UUCP = Total\ UAW + Total\ UUCW$$

Step 6: Computing technical factor

The proposed framework also employs a technical factors multiplier corresponding to the Complexity Adjustment factor of the FPA method. Here we have considered 17 factors corresponding to both development and testing as shown in table 5.3. Weights to these factors are assigned on a scale of 0-2 as is suggested by Karner and values are assigned to each factor, depending on the degree of influence. 0 means no influence, 3 is average, and 5 means strong influence throughout.

Table 5.3 Technical Factors

Technical Factors			
Sr. No.	**Name**	**Weight**	**Description**
T1	Distributed System	1.5	Is the system having distributed architecture or centralized architecture?
T2	Performance Objectives	1.5	Does the client need the system to fast? Is time response one of the important criteria?
T3	End user efficiency	1	How's the ends users efficiency?
T4	Complex Internal Processing	1.5	Is the Business process very complex? Like complicated accounts closing, Inventory tracking, heavy tax calculation etc.
T5	Reusable Code	2	Do we intend to keep the reusability high? So will increase the design complexity.
T6	Installation Ease	0.5	Is client looking for installation ease? By default we get many installers which create package. But if the client is looking for some custom installation probably depending on module wise .One of our client has requirement that when the client wants to install he can choose which

			modules he can install. If the requirement is such that when there is a new version there should be auto installation. These factors will count when assigning value to this factor.
T7	Easy use	1	Is user friendly at the top priority?
T8	Portable	1.5	Is the customer looking for also cross platform implementation?
T9	Easy to change	1.5	Is the customer looking for high customization in the future? So that also increases the Architecture design complexity and hence this factor.
T10	Concurrent	1.5	Is the customer looking at large numbers of users working with locking support? This will increase the architecture complexity and hence this value.
T11	Security objectives	2	Is the Customer looking at having heavy security like SSL or have to write custom code logic for encryption.
T12	Direct access to third parties	1.5	Does the project depend in using third party controls? So for understanding the third-party controls and studying its pros and cons considerable effort will be required. So this factor should be rated accordingly.
T13	User training facilities	0.5	Will the software from user perspective be so complex that separate training has to be provided? So this factor will vary accordingly.
T14	Test tools	2	Are the testing tools available? How much is the proficiency of the tester on these

			modules he can install. If the requirement is such that when there is a new version there should be auto installation. These factors will count when assigning value to this factor.
T7	Easy use	1	Is user friendly at the top priority?
T8	Portable	1.5	Is the customer looking for also cross platform implementation?
T9	Easy to change	1.5	Is the customer looking for high customization in the future? So that also increases the Architecture design complexity and hence this factor.
T10	Concurrent	1.5	Is the customer looking at large numbers of users working with locking support? This will increase the architecture complexity and hence this value.
T11	Security objectives	2	Is the Customer looking at having heavy security like SSL or have to write custom code logic for encryption.
T12	Direct access to third parties	1.5	Does the project depend in using third party controls? So for understanding the third-party controls and studying its pros and cons considerable effort will be required. So this factor should be rated accordingly.
T13	User training facilities	0.5	Will the software from user perspective be so complex that separate training has to be provided? So this factor will vary accordingly.
T14	Test tools	2	Are the testing tools available? How much is the proficiency of the tester on these

			tools? Testing tools can automate the testing process reducing the manual testing effort.
T15	Documented inputs to test tools	1.5	These are the test cases which can be defined as sets of input parameter for which the software will be tested. These test cases can be used by the testing tools to perform regression testing by the capture and play phenomenon.
T16	Test-ware reuse	1	Are the testing equipments available? How much is the proficiency of the tester on these equipments? Test wares can automate the testing process reducing the manual testing effort.
T17	Complex Interfacing	1.5	How much complex is the interface of the software? As more complex is the interface, the more the effort required to test and manages that interface.

In order to assign weights to the above technical factors, we have made a questionnaire in which we have written the various factors that must be counted while making the estimation using use case and questionnaire to various industries in India and ask them to rate these factors on A, B, C, and D grades where A means highest Impact and D means lowest. The questionnaire sent is shown in appendix A and the various responses are shown in appendix B. Now Karner used a scale of 0-2 to weight these factors [9, 23]. We choose the same weights and assign 2 to A, 1.5 to B, 1 to C and 0.5 to D. Now if a factor has maximum responses of B then it is assigned 1.5 and so on for other factors too.

Step 7: Equation for technical factors

$$Tfactor = sum\ (t1....t17)$$

where t1, t2... t17 can be calculated by multiplying the value of each factor by its corresponding weight.

Step 8: Computing TCF (Technical Complexity Factor)

$$TCF = 0.6 + (0.01 * Tfactor)$$

Step 9: Computing Environmental Factor

There are other factors like trained staff, motivation of programmers, etc., which has quite a decent impact on the cost estimate. Here we have considered nine factors and these factors are associated with weights, and values are assigned to each factor, depending on the degree of influence. 0 means no influence, 3 is average, and 5 means strong influence throughout.

Table 5.4 Environmental Factors

Environmental Factors			
Sr. No.	**Name**	**Weight**	**Description**
E1	Familiarity with project	2	Are all the people working in the project familiar with domain and technical details of the project? So probably you will spend your most time in explaining them all know-how's.
E2	Familiarity with development process	1	Are all the people working in the project familiar the development process like Rational Unified Process or Incremental Application development? So probably you will

			spend your most time in explaining them all know-how's.
E3	Objects-oriented Experience	0.5	As use-case documents are inputs to Object oriented design. Its important that people on the project should have basic knowledge of OOP's concept.
E4	Lead analyst capability	2	How the analyst who is leading the project? Does he have enough knowledge of the domain?
E5	Motivation	1.5	Are the programmers motivated for working on the project? As instability in project will always lead to people leaving half way there source code and the hand over becomes really tough. This Factor you can put according to how software industry is going on? Example if the software market is very good put this at maximum value. As good the market more the jobs and more the programmers will jump.
E6	Stable requirements	2	Is the client clear of what he wants? I have seen clients expectations are the most important factor in stability of requirements. If the client is of highly changing nature put this value to maximum.
E7	Working Environment	1	Long work hours and bad work culture tend to decrease the quality and quantity of work being done in unit time.
E8	Part-Time Staff	-0.5	Are there part-time staffs in project like

			consultants etc?
E9	Difficult programming language	-1	How complex the language is? As more complex is the language more is the time required to learn that language.

Step 10: Equation for environmental factors

$$Efactor = sum\ (e1....e9)$$

where e1, e2… e9 can be calculated by multiplying the value of each factor by its corresponding weight.

Step 11: Computing EF (Environmental Factors)

$$EF = 1.4 + (-0.03 * Efactor)$$

Step 12: Calculating the AUCP (Adjusted Use Case Points)

$$AUCP = UUCP * TCF * EF$$

Step 13: Multiplying by Man/Hours Factor

Finally, the AUCP is multiplied by a historically collected figure representing productivity, such as a factor of 20-staff hrs per use case point, to arrive at a project estimate. The result is an estimate of the total number of person hours required to complete the project.

Karner proposed a factor of 20 staff hours per use case point for a project estimate [5, 6, 9]. While field experience of Gautam Banerjee has shown that effort can range from 15 to 30 hours per use case point, therefore converting use case points directly to hours may be an uncertain measure [5, 6]. Some other researchers have different opinions about this factor and have proposed some different value of effort per use case point.

Chapter 6

Guidelines for Complexity Factors

This chapter provides guide lines for assigning values from 0 to 5 to the environmental factors and some technical factors like portability, user training facilities and test tools. There are no guidelines as such provided by any governing body. Guidelines given by us are all based on our understanding. Section 6.1 gives the guidelines for various environmental factors while section 6.2 deals with the guidelines for some technical factors.

6.1 ENVIRONMENTAL FACTORS

There are some general factors which depend on the company and its staff and are very critical for the estimation of effort as the project is being developed by the staff within the company. These factors are called environmental factors. These factors will contribute towards the final effort needed to build the project.

6.1.1 Familiarity With Project

Are all the people working in the project familiar with domain and technical details of the project? If the team is not familiar with such details then a lot of time will go in explaining them all know-how's and this will increase the effort needed to built the system. Also some work approaches are not listed anywhere in literature and come only with experience. The following table can serve as a guideline to assign the weight to this factor based on the familiarity and the experience.

Table 6.1 Guidelines for familiarity with project

Rating	Description
0	None.
1	20 % percent familiarity with project.
2	40 % percent familiarity with project.
3	60 % percent familiarity with project.
4	80 % percent familiarity with project.
5	100 % familiarity with project.

6.1.2 Familiarity With Development Process

Are all the people working in the project familiar the development process like Rational Unified Process or Incremental Application development? So probably you will spend some time in explaining them all know-how's. The following table can serve as a guideline to assign the weight to this factor based on the familiarity.

Table 6.2 Guidelines for familiarity with development process

Rating	Description
0	None.
1	20 % percent familiarity with development process.
2	40 % percent familiarity with development process.
3	60 % percent familiarity with development process.
4	80 % percent familiarity with development process.
5	Expert.

6.1.3 Objects-Oriented Experience

As use-case documents are inputs to object oriented design. It's important that people on the project should have atleast basic knowledge of OOP's concept. The following table can serve as a guideline to assign the weight to this factor based on the experience.

Table 6.3 Guidelines for object-oriented experience

Rating	Description
0	No prior experience in Object Oriented Concepts.
1	Only theoretical experience.
2	2 years experience in Object Oriented Concepts.
3	4 years experience in Object Oriented Concepts.
4	10 years experience in Object Oriented Concepts.
5	15 years experience in Object Oriented Concepts full expert.

6.1.4 Lead Analyst Capability

How is the analyst leading the project team? How much is his experience? Does he have enough knowledge of the domain? Lead Analyst is the person who is responsible for the overall success or failure of the project. If he understands the project domain and have enough experience in that area then effort will reduce significantly. The following table can serve as a guideline to assign the weight to this factor based on the experience.

Table 6.4 Guidelines for lead analyst capability

Rating	Description
0	No lead analyst in the project.
1	3 years experience lead analyst in project.
2	5 years experience lead analyst in project.
3	8 years experience lead analyst in project.
4	10 years experience lead analyst in project.
5	Expert with 15 years of experience as lead analyst.

6.1.5 Motivation

Are the programmers motivated for working on the project? Are they are allowed and willing to take initiative? If it is not so, this will leads to instability among them and instability in project will always lead to people leaving half way there work and the

handling of the work then becomes really tough. This Factor you can put according to how software industry is going on? Example if the software market is very good put this at maximum value. As good the market more the jobs and more the programmers will jump. The following table can serve as a guideline to assign the weight to this factor.

Table 6.5 Guidelines for motivation

Rating	Description
0	No motivation.
1	Low motivation. Team works only when directed. No special initiative from team members.
2	5 % of team are high motivated and have self initiative. 90 % team members work only as directed; no special initiative exists from these team members.
3	20 % of team are high motivated and have self initiative. 80 % team members work only as directed; no special initiative exists from these team members.
4	50 % of team are high motivated and have self initiative. 50 % team members work only as directed; no special initiative exists from these team members.
5	High motivation and self initiation in all members.

6.1.6 Stable Requirements

Is the client clear of what he wants? We believe that client's expectations are the most important factor in stability of requirements. If the requirements are of highly unstable nature put this value to maximum. The following table can serve as a guideline to assign the weight to this factor.

Table 6.6 Guidelines for stable requirements

Rating	Description
0	No stability. Every meeting with customer changes around 80 % deviation from original requirements.
1	Requirements changing around 60 % of the original requirements.
2	Requirements changing around 40 % of the original requirements.
3	Requirements changing around 20 % of the original requirements.
4	Requirements changing around 5 % of the original requirements.
5	Stable.

6.1.7 Working Environment

Long work hours, company policies and bad work culture tend to decrease the quality and quantity of work being done in a unit time. If an employee is satisfied, he/she can give his/her best because he/she doesn't have any other things in mind. If he/she is not satisfied he/she will always think to switch to another company. And we know that any hand over in between of the project will derail the project. There are several working environment factors which govern how this point can be rated.

Table 6.7 Factors affecting working environment

Sr. No.	Working Environment Factors
1	Cooperative management
2	Flexible timings and flexible work
3	Employee Welfare (Facilities like Hospital, Insurance, etc.)
4	Better Wage Structure
5	Working Conditions (like proper lights, proper temperature, basic amenities etc.)

Based on the above factors, working environment can be rated as given below.

Table 6.8 Guidelines for working environment

Rating	Description
0	None of the above.
1	Any one of the above.
2	Any two of the above.
3	Any three of the above.
4	Any four of the above.
5	All five of the above.

6.1.8 Part Time Staff

Are there part-time staffs in project like consultants etc? More the part time staff more is the effort spends in building the project. The following table can serve as a guideline to assign the weight to this factor.

Table 6.9 Guidelines for part time staff

Rating	Description
0	No part time staff.
1	10 % of members are part-time staff.
2	30 % of members are part-time staff.
3	50 % of members are part-time staff.
4	80 % of members are part-time staff.
5	100 % of members are part-time staff.

6.1.9 Difficult Programming Language

How complex the language is? How much time the staff needs to pick up the language? As if more complex language is chosen more is the time needed to pick up the language and hence more is the effort needed. The following table can serve as a guideline to assign the weight to this factor.

Table 6.10 Guidelines for difficult programming language

Rating	Description
0	Easy with in one week the language can be picked up.
1	At least two week is needed to pick up the language.
2	At least one month is needed to pick up the language.
3	Special training needed for the language.
4	Special training needed for the language and need help during the project.
5	Difficult needs only experience people.

6.2 TECHNICAL FACTORS

6.2.1 Portable

A lot of effort is needed if client wants that the project have to run on various different platforms. Two or more platform can belongs to the same family of the operating system or two different families of the operating system. For example Windows 98 and Windows XP belongs to same family namely Windows while Windows 98 and Linux belongs to two different families. The following table can serve as a guideline to assign the weight to this factor.

Table 6.11 Guidelines for portable

Rating	Description
0	Application should cater to only operating system.
1	Application should cater to only one type of family of operating system. Means it will cater to windows OS family example windows 2000, win NT, windows 9x etc. Application should not cater to multiple families of OS.
2	Application should cater to at least two different family of OS example Windows and Linux.
3	Application should cater to three different family of operating system.

4	Application should cater to four different family of operating system.
5	Application should cater to five different family of operating system.

6.2.2 User Training Facilities

Will the software from user perspective be so complex that separate training has to be provided or the user can understands the software itself or with minor help? The following table can serve as a guideline to assign the weight to this factor.

Table 6.12 Guidelines for user training facilities

Rating	Description
0	No user training required.
1	Simple instructions are required to make user understand the system.
2	Help files are supplied to user which will be referred by user during using the software.
3	With help files user has to be provided with expert guidance in initial stage.
4	Special training needed to be provided.
5	Special training is required and certification has to be acquired in order that user is eligible to use the product.

6.2.3 Testing Tools

Automated testing tools can automate the testing process and testing can be done un-attended provided the tester has proficiency in their usage. The following table can serve as a guideline to assign the weight to this factor.

Table 6.13 Guidelines for testing tools

Rating	Description
0	No tool is available/ Used.
1	Tool available and testers have 20 % proficiency on those tools.
2	Tool available and testers have 40 % proficiency on those tools.
3	Tool available and testers have 60 % proficiency on those tools.
4	Tool available and testers have 80 % proficiency on those tools.
5	Tool available and testers are experts of those tools.

Chapter 7

Conclusion and Future Scope of Work

7.1 CONCLUSION

Software war for the best estimation has been going on for years. We are not saying that use case based effort estimation is the best way to do estimation. But definitely we have to measure and one day we have to unify on a common measurement principle. If we can say in real life that city "xyz" is 100 kilometers far, why cannot we say this project is of 10 KLOC complexity, 200 function points or 650 use case points. Different environments, different processes and different languages used by different companies further make it difficult to come to common grounds and hence common measurements.

Use case based method of effort estimation is obviously a very valuable addition to the tools available to the project manager but a detailed investigation of use case based method has led us to conclude that the method has several inadequacies. So, should we avoid use cases for estimation and rely instead on the analysis and design realizations that emerge? The problem with this is that it delays the ability to make estimates. It is better for the project manager to be able to obtain estimates early for the planning purposes, and then refine them iteration by iteration, rather than delaying estimation and proceeding in an unplanned fashion.

All of the estimation methods are susceptible to error, and require accurate historical figures for productivity in order to be useful within the context of the organization. Use case based method is comparable to the function point method that has been widely accepted by the companies. With standardization and support from national and international bodies like IFPUG that have helped the function point method to become widely accepted, this method also has the potential to become mature and one of the widely accepted effort estimation tool.

The environmental factors and the technical factors that are used as cost drivers in the use case based method are not always clearly understood. We have therefore specified the meaning of the specific factors, and defined guidelines for specific counting rules in order to obtain more consistent counts based on our understanding.

7.2 FUTURE SCOPE OF WORK

Some more work is to be done to gain more understanding about include, extend, generalized use case and generalized actor and counting their contribution towards the overall estimated effort. The guidelines proposed for various technical and environmental factors can be further refined and validated. In addition, the guideline for remaining technical factors can be proposed and validated. To have more general view on the weights of technical and environmental factors large number of software industries can be involved. Finally, the proposed framework for effort estimation using use case can be validated on actual projects.

Appendix - A

Questionnaire

Table A1 gives environmental factors and table A2 gives technical factors that affect the estimation of effort for both development and testing. These are assigned four grades - A, B, C, and D according to their impact on effort estimation. Here A means highest impact and D means low impact.

NAME: EXPERIENCE (in months): DESIGNATION:

Table A.1 Environmental factors

Factor	Grades
Familiarity with project	
Familiarity with development process	
Objects-oriented Experience	
Lead analyst capability	
Motivation	
Stable requirements	
Part-Time Staff	
Difficult programming language	
Working Environment	

Table A.2 Technical factors

Factor	Grades
Distributed System	
Performance Objectives	
End user efficiency	
Complex Internal Processing	
Reusable Code	
Installation Ease	
Easy use	
Portable	
Easy to change	
Concurrent	
Security objectives	
Direct access to third parties	
User training facilities	
Test tools	
Documented inputs	
Test-ware reuse	
Complex Interfacing	

Appendix - B

Responses to Questionnaire

Questionnaire was sent to four leading software industries out of which three companies and in total nine persons responded. The solutions to the questionnaire are arranged in the following tables in which table B1 shows the respondents, their experience, designation in company and the company name while table B2 to B4 shows the responses of different programmers and table B5 shows the final consolidation of the responses. Names are kept confidential on the request of respondents.

Table B.1 Information of respondents

Respondent	Experience (in months)	Designation	Company Name
Programmer 1	15	Assistant Systems Engineer (Consultancy)	TCS
Programmer 2	22	Software Engineer	Flexitronics Software Systems (Formerly Hughes)
Programmer 3	66	Technical Leader	-do-
Programmer 4	10	Software Engineer (RnD)	Quark Media House
Programmer 5	22	Software Engineer (RnD)	-do-
Programmer 6	15	Software Engineer (Quality Engineering)	-do-
Programmer 7	39	Software Engineer (RnD)	-do-
Programmer 8	34	Software Engineer	-do-
Programmer 9	13	Software Engineer	-do-

Table B.2 Responses from Programmer 1, 2 and 3

Factors	Programmer 1	Programmer 2	Programmer 3
Distributed System	A	B	B
Performance Objectives	B	B	B
End user efficiency	C	B	B
Complex Internal Processing	B	C	B
Reusable Code	A	A	A
Installation Ease	D	B	C
Easy use	D	C	C
Portable	D	C	C
Easy to change	D	B	B
Concurrent	C	C	C
Security objectives	C	C	C
Direct access to third parties	C	B	B
User training facilities	D	C	C
Test tools	D	A	A
Documented inputs	D	B	D
Test-ware reuse	D	C	C
Complex Interfacing	A	B	B
Familiarity with project	A	A	A
Familiarity with development process	B	C	B
Objects-oriented Experience	C	D	D
Lead analyst capability	A	B	A
Motivation	B	A	B
Stable requirements	A	A	A
Part-Time Staff	D	C	D
Difficult programming language	C	C	D
Working Environment	C	C	D

Table B.3 Responses from Programmer 4, 5 and 6

Factors	Programmer 4	Programmer 5	Programmer 6
Distributed System	A	A	B
Performance Objectives	C	C	B
End user efficiency	C	D	D
Complex Internal Processing	A	B	A
Reusable Code	A	B	B
Installation Ease	D	D	D
Easy use	B	C	C
Portable	B	B	B
Easy to change	B	A	D
Concurrent	B	B	B
Security objectives	A	A	A
Direct access to third parties	B	B	C
User training facilities	D	D	D
Test tools	A	A	A
Documented inputs	B	B	B
Test-ware reuse	C	C	B
Complex Interfacing	B	A	B
Familiarity with project	B	A	B
Familiarity with development process	A	B	B
Objects-oriented Experience	D	C	D
Lead analyst capability	C	A	D
Motivation	B	C	B
Stable requirements	A	A	B
Part-Time Staff	C	B	D
Difficult programming language	A	C	C
Working Environment	B	A	A

Table B.4 Responses from Programmer 7, 8 and 9

Factors	Programmer 7	Programmer 8	Programmer 9
Distributed System	B	B	A
Performance Objectives	A	B	C
End user efficiency	A	C	C
Complex Internal Processing	D	A	B
Reusable Code	A	B	A
Installation Ease	B	A	D
Easy use	A	A	C
Portable	A	B	B
Easy to change	A	B	B
Concurrent	A	B	B
Security objectives	A	B	B
Direct access to third parties	B	A	C
User training facilities	A	A	C
Test tools	A	B	C
Documented inputs	B	B	B
Test-ware reuse	A	C	C
Complex Interfacing	C	B	B
Familiarity with project	A	B	C
Familiarity with development process	C	C	B
Objects-oriented Experience	A	D	C
Lead analyst capability	C	A	D
Motivation	C	D	B
Stable requirements	B	A	B
Part-Time Staff	C	D	A
Difficult programming language	B	C	D
Working Environment	A	C	C

Table B.5 Consolidation of the responses

Factors	Grades			
	A	B	C	D
Distributed System	4	5		
Performance Objectives	1	5	3	
End user efficiency	1	2	4	2
Complex Internal Processing	3	4	1	1
Reusable Code	6	3		
Installation Ease	1	2	1	5
Easy use	2	1	5	1
Portable	1	5	2	1
Easy to change	2	5		2
Concurrent	1	5	3	
Security objectives	4	2	3	
Direct access to third parties	1	5	3	
User training facilities	2		3	4
Test tools	6	1	1	1
Documented inputs		7		2
Test-ware reuse	1	1	6	1
Complex Interfacing	2	6	1	
Familiarity with project	5	3	1	
Familiarity with development process	1	5	3	
Objects-oriented Experience	1		3	5
Lead analyst capability	4	1	2	2
Motivation	1	5	2	1
Stable requirements	6	3		
Part-Time Staff	1	1	3	4
Difficult programming language	1	1	5	2
Working Environment	3	1	4	1

Appendix - C

Acronyms

These acronyms are used through out the thesis so this section gives us a more clear picture what exactly they are.

Acronym	Full form	Definition
UAW	Unadjusted Actor Weights	A numeric sum of value of actors after giving the classification and before multiplying the technical complexity factor of the system.
UUCW	Unadjusted Use Case Weight	A numeric sum of value of Use cases after classifying and before multiplying the technical complexity factor of the system.
UUCP	Unadjusted Use Case Points	Sum of UAW and UUCW
API	Application Program Interface	Application programs used for accessing services provided by some lower-level module (such as operating system)
GUI	Graphical User Interface	A computer terminal interface, such as Windows, that is based on graphics instead of text.
Tfactor	Technical Factor	Total of all technical factor.
TCF	Technical Complexity Factor	Factor which defines the technical complexity of the project.
Efactor	Environment Factor	Total of all environment factor.
EF	Environment Factor	Factor which defines the general complexity of the project.

AUCP	Adjusted Use Case Points	This is the multiplication of UUCP, TCF, and EF.
OOP	Object Oriented Programming	A programming technology in which program components are put together from reusable building blocks known as objects.
UML	Unified Modeling Language	Stands for Unified Modeling Language. UML is a standard notation and modeling technique for analyzing real-world objects, developing systems, designing software modules in object-oriented approach.

Appendix - D

Glossary

Actor: In UML, someone or something outside the system that interacts with the system.

Include: In UML, a relationship from a base use case to an included use case specifying how the behavior defined for the included use case can be inserted into the behavior defined for the base use case.

Association: In UML, a relationship between an actor and a use case that indicates that the actor interacts with the system by means of the use case.

Extend: In UML, a relationship from an extending use case to a base use case specifying how the behavior defined for the extending use case can be optionally inserted into the behavior defined for the base use case.

Model: A semantically closed abstraction of a system or a complete description of a system from a particular perspective. Examples include use case, architecture, and domain models and code.

Object Management Group (OMG): An international standards organization that owns and maintains CORBA and UML standards.

Relationship: A semantic connection between model elements. Examples include associations, dependencies, and generalizations. Relationships to use cases include association, extend, and include.

System: A conceptual entity defined by its boundaries. Examples include companies, divisions, and sets of software applications, components, machines, and devices.

Unified Modeling Language (UML): A graphical language for visualizing, specifying, constructing, and documenting an object-oriented software-intensive system's artifacts.

Use case: In UML, a complete task of a system that provides a measurable result of value for an actor. More formally, a use case defines a set of use case instances or scenarios.

Use case diagram: A UML diagram that shows actors, use cases, and their relationships.

References

It would be completely selfish in my part that the whole work is my own wisdom. Lot of people has directly or indirectly contributed to my thesis work. I have referred numerous articles from WWW and many white papers and research papers. So this section has all the links and article referred.

[1] Bente Anda. *"Comparing Use Case based Estimates with Expert Estimates"*. Proceedings of the 2002 Conference on Empirical Assessment in Software Engineering (EASE 2002), Keele, United Kingdom, April 8-10, 2002.

[2] Bente Anda, Endre Angelvik, Kirsten Ribu. *"Improving Estimation Practices by Applying Use Case Models"*. Springer-Verlag, Berlin Heidelberg, pp. 383-397, 2002

[3] Bente Anda, Hege Dreiem, Magne Jorgensen, and Dag Sjoberg. *"Estimating Software Development Effort based on Use Cases – Experience from Industry"*. In M. Gogolla, C. Kobryn (Eds.): UML 2001 - The Unified Modeling Language. Springer-Verlag. 4th International Conference, Toronto, Canada, October 1-5, 2001, LNCS 218, 2001.

[4] Craig Larman. *"Applying UML and Patterns – An Introduction to Object-Oriented Analysis and Design and the Unified Process"*. Second Edition, Pearson Education, 2005.

[5] Gautam Banerjee. *"Use Case Estimation Framework"*. Annual IPML Conference, 2004.

[6] Gautam Banerjee. *"Use Case Points - An Estimation Approach"*. August 2001.

[7] Grady Booch, James Rumbaugh, Ivar Jacobson. *"The Unified Modeling Language – User Guide"*. Pearson Education, 2003

[8] Grady Booch, James Rumbaugh, Ivar Jacobson. *"The Unified Modeling Language – Reference Manual"*. Pearson Education, 2003

[9] Gustav Karner. *"Resource Estimation for Objectory Projects"*. Objective Systems SF AB (copyright owned by Rational Software), 1993.

[10] Ivar Jacobson. *"Use Cases -- Yesterday, Today, and Tomorrow"*.

[11] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar Overgaard. *"Object Oriented Software Engineering – A Use Case Driven Approach"*. Addison – Wesley, 1999.

[12] John Smith. *"The Estimation of Effort Based on Use Cases"*. Rational Software, Cupertino, CA.TP-171.October, 1999.

[13] Kirsten Ribu. *"Estimating Object-Oriented Software Projects with Use Cases"*. Masters Thesis, University of Oslo, Department of Informatics, November 2001.

[14] Mark Priestley. *"Practical Object Oriented Design with UML"*. McGraw-Hill International Editions, 2000.

[15] Mel Damodaran. *"Estimation Using Use Case Points"*, Computer Science Program, University of Houston-Victoria.

[16] Mark Lorenz and Jeff Kidd. *"Object Oriented Software Metrics A Practical Guide"*. PTR Prentice Hall, Englewood cliffs, New Jersey 07632.

[17] Pfledger & Fenton. *"Software Metrics – A Rigorous and Practical Approach"*, Thompson, 1996.

[18] Rakesh Agarwal, Santanu Banerjee and Bhaskar Gosh. *"Estimating Internet Based Projects: A Case Study"*. Quality Week 2001, Paper 6W2.

[19] Roger S. Pressman, *"Software Engineering – A Practitioner's Approach"*, fifth edition.

[20] S. R. Chidamber, C. Kermer. *"A Metrics Suite for object oriented design"*. IEEE Transaction on Software Engineering, Vol. 20 (6), pp.476-493, June 1994.

[21] Sunint Saini, R. S. Salaria. *"Inconsistencies of CK and MOOD Metrics"*, 2nd International Conference on Software Engineering Research, Management & Applications (SERA 2004), Los Angeles, CA, USA. May 5th -7th, 2004. pp. 160-165.

[22] Suresh Nageswaran. *"Test Effort Estimation Using Use Case Points"*. Quality Week 2001, San Francisco, California, USA, June 2001.

[23] Thomas Fetcke, Alain Abran, & Tho-Hau Nguyen. *"Mapping the OO-Jacobson Approach into Function Point Analysis"*. Technology of Object-Oriented Languages and Systems, TOOLS-23. IEEE Computer Society, Los Alamitos, CA, USA, pp. 192-202. 1998.

[24] http://www.alistapart.com/articles/whatstheproblem/

[25] http://bdn.borland.com/article/0,1410,31877,00.html

[26] http://www.c-sharpcorner.com/Code/2004/Dec/UseCasePoints.asp

[27] http://www.callista.se/ITPartner/Timeart.htm

[28] http://www.plenux.com/article_ucp.asp

[29] http://www.pmforum.org/library/papers04/pmusecase.htm

[30] http://sunset.usc.edu/classes/cs577b_2001/metricsguide/metrics.html

[31] http://www.uea.ac.uk/~a168955/effort_estimation/problems_UCPM.html

[32] http://www.umsl.edu/~sauter/analysis/488_f01_papers/wang.htm

Papers Communicated/Accepted/Published

Gaurav Singhal, R. S. Salaria; **Analysis of Use Case Point Method of Software Estimation**, nd International Conference on Distributed Computing & Internet Technology **(ICDCIT 2005)**, December 22-24, 2005, Kalinga Institute of Industrial Technology, Bhubaneswar, India.

(Communicated)