UML Based Effort Estimation In Component Based Systems

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering

in

Software Engineering

By:

VINEET KHERA

80731025

Under the supervision of:

Dr RAJESH KUMAR

Associate Professor

SMCA, Thapar University

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

THAPAR UNIVERSITY

PATIALA – 147004

JUNE 2009

Certificate

I hereby certify that the work which is being presented in the thesis entitled, **"UML based effort estimation in component based systems"**, in partial fulfillment of the requirements for the award of degree of Master of Engineering in Software Engineering submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of **Dr. Rajesh Kumar** and refers other researcher's works which are duly listed in the reference section.

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

(Vineet Khera)

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

(Dr. Rajesh Kumar)

School of Mathematics & Computer Application

Thapar University, Patiala

Countersigned by

(SEEMA BAWA)

Professor & Head

Computer Science & Engineering. Department

Thapar University, Patiala

(R.K.SHARMA)

Dean (Academic Affairs)

Thapar University, Patiala

Acknowledgement

No amount of words can adequately express the debt, I owe to Dr. Rajesh Kumar, Associate Professor, School of Mathematics & Computer Application, for his kind support, motivation and inspiration that triggered me for the thesis work. I owe him lots of gratitude for having me shown this way of research. He could not even realize how much I have learnt from him.

I wish to express my gratitude to Ms. Seema Bawa, Professor & Head, Computer Science & Engineering Department, for their excellent support and encouragement right from beginning of this course. I am also thankful to all the faculty and staff members of the Computer Science & Engineering Department for providing me all the facilities required for the completion of this work.

I would also like to express my gratitude to Mr. Arun Sharma, Professor, Amity University, Noida for the support and guidance he has given to me. It was my pleasure working with him during this thesis work.

Most importantly, I would like to give God the glory for all of the efforts I have put into this report.

Vineet Khera

80731025

Abstract

Software development has come a long way from traditional software development, which is characterized by the structured programming paradigm introduced in the late 60's and early 70's to contemporary development practices, which characterize a software application as interacting, independent components. Traditional software effort estimation models capture this monolithic view of software development. In these models, software effort is projected at the large-grained system level whereas to accurately predict effort in Component Based Software Development (CBSD), a fine-grained approach is needed to identify and classify the relevant cost factors. Effort estimation in CBSD is concerned with deriving estimations for small, concurrent development projects. The small projects allow for quantitative and direct measurements of the factors influencing cost. CBSD is principally based on the core concepts of Object Orientated Techniques where the role of UML is unbeatable. Different UML constructs can be used at different stages of software development for estimation of resources like efforts & cost etc.

In this study, the literature was reviewed for various estimation techniques based on UML diagrams and their pros and cons in context of CBSD were underlined. The important aspect of CBSD is its functional and non-functional requirements from the integrated set of components. These were incorporated into newly defined set of technical and environmental factors and other major issues like use case granularity etc. were collected during the computations. Karner's use case point method was adapted and the revised framework was applied on two case studies. Validation was done on another similar work done by Bente Anda *et* al. and the results varied just about 6.15% with respect to actual efforts made. This study thus, shows how complexity factors can be calibrated for a specific context and produce relatively accurate estimates.

Table of contents

List of Tables

List of Figures

Chapter 1

Introduction

`

Software development has improved considerably from traditional software practices i.e. the structured programming techniques to contemporary development practices, which characterize a software application as interacting, independent components. Traditional software effort estimation models capture this monolithic view of software development. In these models, software effort is projected at the large-grained system level whereas to accurately predict effort in Component Based Software Development (CBSD), a fine-grained approach is needed to identify and classify the relevant cost factors. Effort estimation in CBSD is concerned with deriving estimations for small, concurrent development projects. The small projects allow for quantitative and direct measurements of the factors influencing cost. CBSD is principally based on the core concepts of Object Orientated Techniques where the role of UML is unbeatable. Different UML constructs can be used at different stages of software development for estimation of resources like efforts & cost etc. In the following chapters, we shall investigate different practices for effort estimation based on UML constructs and shall bridge the gap while applying them to CBSD.

1.1 Estimation

There can be three different approaches in developing highly trustworthy software systems. The first is developing new methodologies to improve software quality. An example of this is object-oriented, component-based software development. The second approach is process improvement. This approach has improved software quality and reliability. The third approach is software measurement. We need accurate metrics for measuring software systems and predicting the effort required for development.

The methodologies used in the first two approaches affect how the software is measured. For example, object-oriented methodology generates new metrics relating to object oriented technology. Process improvement can be enhanced through metrics of each process. In the research of software measurement in terms of effort estimation, several criticisms exist: lack of a theoretical basis, lack of desirable measurement properties, being insufficiently generalized, being too implementation technology dependent, being a subjective measurement based on expert decision and being too labor-intensive for collecting information [7,16]. Software metrics were used as the basic foundation of prediction of effort. The traditional approaches focused on source code or expert decision-based analysis to provide accurate information for calculation. Analogy based, the most pragmatic form of estimation gives accurate estimation, is very simple to apply to similar projects, and provides rapid estimation with detailed documentation. It results in increasing unreliability and confronts difficulties with real environment and given data. The other approach of function analysis is better and reliable for size estimation. This can be applicable in the early stage of project life cycle, and is language & platform independent. This still involves manual/high labor cost and is not applicable to latest software development methodology. This is not even ideal in the requirement capture period. Another approach COCOMO-I poses live effort estimation, transparent algorithm, local calibration and free implementation on the cost of being highly dependent on size input, small data set to determine the parameter heuristics.

As apparent, common problems with these approaches are lack of early estimation, over-dependence on expert decision, and subjective measurement of each metric. A different approach is required to overcome these existing difficulties. We move upstream in the software development process to requirement analysis and design. Currently, UML diagrams are widely used in the software development industry for requirement analysis and detail design before jumping into the coding processes.

1.2 Unified Modeling Language (UML)

Modeling has been an essential part of engineering, art and construction for centuries. Complex software designs that would be difficult for a user describe textually can readily be conveyed through diagrams. Modeling provides three key benefits: visualization, complexity management and clear communication. UML is a visual language for specifying, constructing and documenting the artifacts of systems. It is a technique which is adopted by software professionals all over the world and used for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. It can be used with many processes models, throughout the software development life cycle and across different implementation technologies. It is a graphical notation for object-oriented analysis and design. UML provides a framework for describing a set of models that capture the functional and structural semantics of any complex information system.

1.2.1 Benefits of using UML

UML was designed by Rational Software in order to provide a standard for software modeling languages. It is now owned and controlled by the Object Modeling Group, an independent organization. UML has quickly become the de-facto standard modeling language in the software industry. UML is said to be a universal language because it can be applied in many areas of software development. Here are few reasons for why should UML be used for software construction at a universal level:

i) Software construction generally needs a plan and there is a common perception among general management that software is both cheap to produce and easy to change. In contrary, software is extremely complex and, once placed in a particular structure, often long-winded and difficult to change. A building architect creates a drawing of the building from more than one direction, ensuring that the structure is appropriate for the purpose of the building and that the dimensions are all consistent. Only when this visual model of the proposed work

has been approved is the job of laying bricks begun. Therefore UML as a modeling language is a useful tool for software construction.

ii) Considering multiple dimensions and levels of detail, a software is very abstract and hard to visualize. A visual modeling language, such as UML, allows software to be visualized in multiple dimensions so that a computer system can be completely understood before construction begins. Furthermore, UML can be used to produce several models at increasing levels of detail. The overall scope of the software can quickly and easily be defined at the start of the project with a high level model allowing for accurate estimation. Increasing levels of detail can then be added to each part of the software as it is constructed, until finally the software appears as code. The code can then be tested against a test model that is derived from the original model of requirements.

iii) UML is appropriate for both new system developments and improvements to existing systems. It is a fallacy to use a new modeling technique on an old system; the old system will have to be completely “re-documented” in the new style in order for any change to take place. The truth is that only those parts that are affected by the change will need to be modeled. If they were badly documented or not documented at all, then a lot of work would need to be done to understand them in order to make the change.

iv) Undocumented or badly documented software has a reduced value as a company asset. With so much of the company's business process embedded in obtuse software code, a company is vulnerable to losing an understanding of and control over the business rules by which it operates. Gradual re-engineering of the company's bespoke computer software in a way that makes the functionality of the software transparent brings those important assets back under control. It removes the company from dependence of key personnel who may leave at any time and improves understanding of the company's critical business processes. It even improves the quality of software engineering staff because those who want

to do a proper job will look for companies that model with UML. It includes user-definable extension mechanisms so that it can be adapted for specific environments. Industry standard extensions now exist for business modeling and web-based applications development.

v) The use case driven nature of modeling with UML ensures that all levels of model trace back to elements of the original functional requirements. This traceability between models comes without the extra effort of creating and maintaining a 'traceability matrix' which can be a complex and time consuming job. The result is that the impact of a requested change can quickly be estimated. A change in requirements can be traced though analysis and design models into those components affected and even lines of code. The code affected can then be traced back to the requirements and total regression testing effort calculated.

vi) UML accommodates incremental development and re-development of software systems. UML models respond well to an incremental development environment. It is possible to develop only those parts of the model that are required to satisfy the new requirements, and the code needed to fulfill those requirements is in place. This approach ensures that only the work needed to fulfill the current requirement is done, while still developing in such a way that maximizes re-use, maintainability and extensibility.

1.2.2 UML for estimation (Motivation)

UML is a widely recognized standard to describe software systems using object-oriented concepts through visualization. It provides a well-structured architecture and overview of a system through various diagrams representing different viewpoints of the target system. Useful information extracted from UML diagrams provides the benefit of a language-independent measurement in the upstream level of software development.

To glean useful information early in the software development process, we focus mainly on two UML diagrams *viz.* use case diagram and class diagram. While requirement elicitation is captured through a use case diagram, detail design information is provided in class diagrams. The early work starts from use case points, which was first given by Karner in 1993 [18] as a software project effort estimation model. His work has been inspiration for researchers since then and many have elaborated his approach in different contexts. Use case point approach has gained significant popularity among industry professionals as well as research scholars. There are many approaches available that exploits knowledge from use case diagrams and class diagrams in the process of predicting efforts for a system.

Use case diagrams are not the only source of early estimation. Analysis class diagrams can also provide handful of information pertaining to effort required to build the software. Among such approaches, one details Class Points with the help of use case diagrams, interaction diagram and state diagrams. It also proposes optional use of interaction diagram and state diagram, which can provide deep insight and more accurate prediction for efforts required for a system. Another approach of UML points provides simple calculation, easy to implement, and provides reasonable cost estimation in the upper stage of software development. It exploits use case diagram and class diagrams for more information. We shall have a brief lookup on all three approaches and few others in later sections.

1.3 Component Based Software Development (CBSD)

"A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture."

CBSD is a process that emphasizes the design and construction of computer based systems using reusable software components. CBSD encompasses two parallel engineering activities: domain engineering and component based development as shown in Figure 1 [28]. Domain engineering explores an application domain with the specific intent of finding functional, behavioral, and data components that are candidates for

reuse. These components are placed in reuse libraries. Component based development collects requirements from the customer, selects an appropriate architectural style to meet the objectives of system to be built, and then follows the sequence as

Search -> Select -> Create/Adapt -> Integrate -> Maintain

In addition custom components are engineered to address those aspects of the system that cannot be implemented using existing components.

Component-based approaches for the development of information systems are a widely growing engineering discipline that deals with the cycle of developing components and developing with components. Software engineers try to borrow concepts of components composition from other engineering disciplines to component software. Since the early nineties when Microsoft introduced a component-based development environment (the Visual Basic environment and its pluggable components), several component-based approaches and components models were introduced (COM, DCOM, EJB, CORBA etc.).

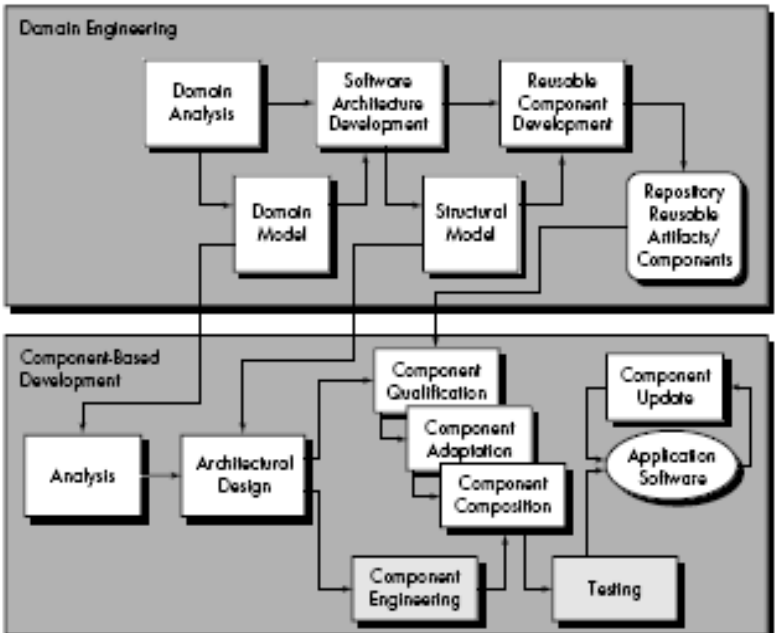

Figure1: Component based software development

Component based software development is focused on assembling existing components to build a software system, with a potential benefit of delivering quality systems by using quality components. It deviates from the conventional software development process in a way that it is integration centric as opposed to development

centric. The quality of a component based system using high quality components thus depends on the quality of its components, the framework and the integration process used. CBSD has thus brought along potential risks and challenges to the techniques and methods for estimating costs, measuring and assessing the quality attributes of a component based system, given the fact that the quality of an individual Component influences the quality of the overall system. There is a need for a new set of metrics to assess the effectiveness of CBSD. Unlike the traditional metrics, they do not depend on the component's code size, which is generally unknown. Cost estimation is a key requirement for component based software systems before the actual development activities can proceed. Cost is a function of the enterprise itself, its particular development process, the selected solution, and the management and availability of the resource during the development project. A component based system is usually complex and testing the final product is time consuming and costly. Thus it is necessary to have testability analysis of the component based system, to indicate the difficulty level of component based system testing. There is a lack of component based performance evaluation models. Effective maintenance involves detailed observations of the behavior of a system and is driven by software complexity. However, component based system presents a unique maintenance challenges. Unlike the traditional software systems, these cannot be done by drilling down to view or change the source code of the component, but are restricted to reconfiguring and reintegrating components. The reliability measurement of a component based system comprises of two main activities: first, the reliability measurement of individual component and second, the reliability measurement of the whole system based on the component context model.

In past, the significant advantages of CBSD over traditional development approaches including white-box reuse have been reported. Industry analysts have also argued that component based development approaches are an important building block for framework like web services, and that the keen interest in Web services has made CBSD a mainstream technology. Predicting efforts and cost in CBSD becomes more complicated when we consider the cost required for searching and integrating reusable components.

1.4 Thesis organization

This work encompasses all the terms described in earlier sections and correlate them into one context. Our focus is to provide a framework for effort estimation using UML diagrams specifically for component based systems. This thesis, hence, is organized in following manner.

Chapter 1 (Introduction): contains all introductory topics that are required to understand the content and context of this work. This chapter describes effort estimation and its relevance for software development; UML and its benefits in software development life cycle (SDLC); CBSD and its major issues and lastly context division of this thesis.

Chapter 2 (Literature Survey): contains the literature survey for existing and related research with respect to this work. It also gives an outline of comparison for different techniques and provides various pros and cons for the respective technique under review.

Chapter 3 (Problem Statement): outlines the gaps in existing work; specific problem statement of problem and brief analysis of the problem.

Chapter 4 (Proposed framework): describes the proposed solution with respect to the existing research gap; guidelines to solve the problem and exact procedure of solving the problem.

Chapter 5 (Evaluation and validation): contains the description of case studies on which the evaluation was done, respective result sheet and the validation with respect to existing similar work.

Chapter 6 (conclusion): concludes the work with final remarks and also underlines few issues that are left un-answered in this work but are worth mentioning while discussing about effort estimation for component based systems.

Chapter 2

Literature Survey

Past Karner's work, there has been continuous research in UML based effort estimation. Many researchers have even used class diagrams and few also included sequence diagrams, state diagrams etc. but most of the work primarily relies on use case diagrams. The very nature of being early analysis activity of use case diagram makes it obvious choice in predicting efforts required to build the desired system. Early analysis classes can also enhance the accuracy of estimation but current work requires more details, which can only be provided through design classes. We reviewed many popular and established techniques based on UML diagrams and found the use case point approach as most popular, easy and convincing approach. In this chapter, we shall elaborate three methods in detail because of the major difference in their approach, and later on few other techniques based on use case points will be discussed.

2.1 Use Case Points

Use case diagrams contain the functional behavior of the target system, determined during the requirement analysis phase. Introduced by Karner [18], UCP effort estimation was an extension of existing estimation methods, such as function point analysis. Figure 2 depicts the main flow of effort estimation based on the UCP calculation through Karner's [18] approach.

In this process, first of all actors are characterized as simple, average or complex and the total unadjusted actor weight (UAW) is calculated by counting the number of actors in each category, multiplying each total by its specified weighting factor, and then adding the points. Next, categorization of the use cases is done 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).

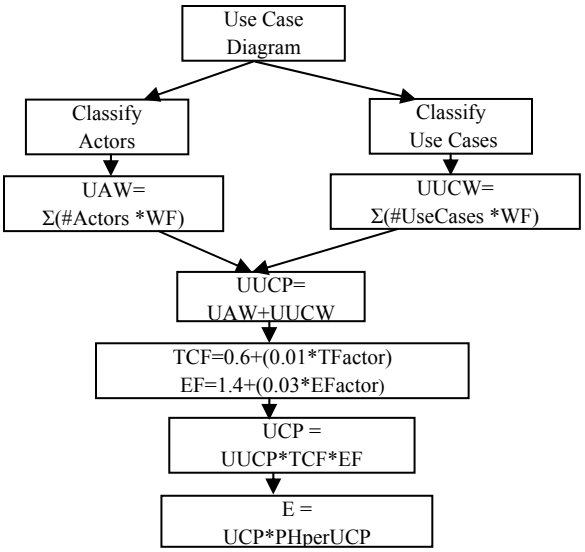

Figure 2: Karner's approach

Next, the use case points are adjusted based on the values assigned to a number of technical factors and environmental factors. Each factor is assigned a value between 0 and 5 depending on its assumed influence on the overall project. This step has 3 different formulae:

1. The Technical Complexity Factor:

   TCF = 0.6 + (.01 * Tfactor)

2. The Environmental Factor:

   EF = 1.4 + (-0.03 * Efactor)

3. Adjusted use case points:

   UCP = UUCP * TCF * EF

Finally, the UCP is multiplied by a historically collected data representing productivity, such as a factor of 20- staff hours 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 provided a detailed table describing how each factor was determined and what value was assigned at each step [18].

Pros & Cons of Karner's method

Karner's approach being the earliest work in UML based estimation provided a foundation for researchers in this domain. It considers 13 technical and 8 environmental factors directly affecting use cases and hence the effort estimation. However, it has weak points when applied to general software projects. UCP lacks information, only counting the number of actors and use cases. This method also relies heavily on the estimating expert regarding the weighting of UAW/UUCW and the technical and environmental factors. The determined value of each of these factors will be highly dependent on the expert's opinion, and will therefore increase variance in the final results. Besides this, Karner's work omits the inclusion of "*include*" and "*extend*" use cases and does not even clarifies the granularity of a use case.

2.2 Class Points: Fast&&Serious Method

Carbone and Santucci emphasized that Object Oriented (OO) metrics should catch the three canonical dimensions of OO software: functionality (behaviour of objects), amount of communication among objects, and percentage of reuse through inheritance [23]. They named it Fast&&Serious, which combines several measures extracted by UML diagrams and associating each class with a size estimation, in terms of source lines of code (SLOC). This method works on the data about the project under analysis extracted from Rational Rose, a CASE tool used for design specification using UML; and produces an estimation of the software complexity in terms of SLOC. For comparison purpose it is converted in terms of traditional Function Point mehod using a backfire index.

It is a method that starts analyzing the class diagram and on the basis of the diagram level of detail, applies a rough (Fast) or a detailed (Serious) estimation method. Numerical data computed for all the classes belonging to the class diagram is further assessed through additional pieces of information coming from other UML diagrams (i.e., use cases, sequence diagrams, and state diagrams).

The Fast&&Serious [23] measurement process consists of six major steps, some of them optional. Following table lists them, where a * denotes an optional step.

Step	Description
1	Determine the type of method to apply: Fast or Serious
2	Calculate the complexity of each class (Class Point: CP) in the class diagram under analysis.
3	Assess the CPs by use case diagram.
4	Assess the CPs by interaction diagrams.
5	Assess the CPs by state diagrams.
6	Sum the CPs computed in phases 2, 3, 4, and 5 obtaining the number of SLOC of the whole system.

Table 1: Fast&&Serious Steps

Step 1: Fast or Serious?

In this step the class diagram are analyzed and for each class the following metrics are extracted:

- DIT (Depth in Inheritance Tree)
- MPC (Number of Methods per Class)
- NAC (Number of Association per Class)
- PMS (Percentage of methods with signature)

Average of PMS is used to choose the method to adopt in the next steps. For Fast method, PMS should be greater than a threshold value (say 70%), otherwise Serious method is to be applied. DIT, MPC, and NAC will be used in next steps.

Step 2: Computing class complexity

In this step a complexity value for each class is calculated with the help of few intermediate support results. For each class, the number of Class Point (CP) is calculated as follows:

$$CP(c) = 2 * SP(c) + 3 * BP(c)$$

Where CP(c) estimates the complexity of class c; SP(c) as State Points being the sum of weights of each list attributes in class c; and BP(c) as Behavioral Points calculated for each method in class c. Note that if a class c has no attributes and methods, it is not included in the estimation process.

Step 3: Assessing class complexity using Use Case Diagrams

This step intends to assess the CP(c) computed in the previous step using the information extracted from use case diagrams. This is done by first associating a complexity value with each use case and each actor; then using CUCA(c) and CAA(c) to assess the complexity of c, and finally finding the increment/decrement percentage for CP(c). For example CUCA(c) = 8, CAA(c) = 15 gives the increment of 9% through Use Case assessing as given by Carbone and Santucci [23]. Hence, if CP(c) = 1000 the new value for CP(c) shall be

$$CP(c) = 1000 + 1000*9/100.$$

Step 4: Exploiting Interaction Diagrams

Similar to step 3 where the strategy was to assess the CP of a class when new information is available, the further focus is to exploit interaction diagrams; in particular sequence diagrams.

For each sequence diagram sed_j where j=1...nsed and *nsed* is the number of available sequence diagrams, we look at the instances of *c* in each sed_j calculating numMess(*c*, sed_j) as the number of messages sent or received from instances of class *c* in sed_j. Finally, let totMess(*c*) as the sum of numMess(*c*, sed_j) and rsed(*c*) as the ratio between the number of sequence diagrams wherever *c* appears. Here, totMess(*c*) gives an idea of the amount of communication involving class *c*, while rsed(*c*) corresponds to the percentage of code that communicates with such a class. These values are useful to find the percentage of increment/decrement for CP(*c*) as given by Carbone and Santucci [23].

Step 5: Exploiting State Diagrams

A State Diagram *std* may be directly associated with a class *c* or to a method of *c*. We extract the following values from each *std*: the number of states as numSta(*std*) and the number of actions (i.e., entry and exit actions associate with a state) as numAction(*std*). We assess CP(*c*) after computing the total amount of states totSta(*c*) and total number of actions totAction(*c*) for the class *c* and finding the percentage of increment/decrement as guided by Carbone and Santucci in State diagram assessing table [23].

Step 6: Computing the whole system size

So far, we have a CP(*c*) associated with each class in the class diagram (CD); CP(*c*) is correlated with the size of *c* and of the whole system (in terms of Java SLOCs) through the following formulae:

$$Size(c) = 4 + 10 * CP(c)^{0.7}$$

$$Size(System) = \sum_{ci \in CD} Size(c_i)$$

The number of FP associated with the size (System) may be obtained using the backfire index for the Java language [23].

Pros & Cons of Fast&&Serious method

Fast&&Serious [23] presents a novel method for estimating OO software projects size exploiting many UML diagrams. The main features of this approach are: the method can be totally automated; it exploits all the principal UML diagrams in its estimation process; and its output is in terms of SLOC, so can be compared with well known standard like FP. Yet, it is to note that in this method introduces several formulas whose constants have to be fixed performing exhaustive tests; and it uses preliminary figures derived by some training experiments. Also its implementation on incremental large scale development like component based systems is yet to be validated.

2.3 UML Points

Kim *et* al. [19] proposed a new approach that is easier to calculate, excludes the expert's decision, and focus more on the diagram itself. They used UML points to estimate project effort and size measurement. UML points are calculated by adding use case points and class points. According to their work [19], use case diagram has much information about the early development stage's concept and the target system's dynamic viewpoint. The developer uses this diagram for communicating with the customer to decrease the conceptual gap between them, which has sufficient knowledge of the target system. It generates few intermediate results before computing use case points. Use case Points (UCP) – This definition represents the use case points of the target system.

$$UCP = \Sigma(NOA+NOUC+NOR)$$

Where NOA = Number of actors, NOUC = Number of use cases, and NOR = Number of roles.

The ratio values ANA_UC and ANR_UC are easily calculated by using given equations [19] to provide a more general overview of use case points. Specifically, use case diagram is part of communication between developer and customer to reduce conceptual gaps between them, so it has sufficient knowledge about the target system. Therefore, the value of the use case points can be used as an input for this effort estimation model. For instance, if the value of NOA is high then the system will have a

great deal of interaction with its environment. UCP is calculated by adding up all of the use case points.

In object-oriented development, the class diagram has a great deal of quantification information based on the design document. It contains the structural functionality of the target system and its class hierarchy, which are the logical blocks of the developed system. The class points approach was first introduced in 1998. This was based on the function points analysis approach to represent the internal attributes of a software system in terms of counting. Figure 3 describes three major steps to measure a target system in terms of class points as described by Kim *et* al. [19]. Each step consists of major activities required to gather quantification information of classes.

The first step is to identify and classify the classes into four system types: problem domain type, human interaction type, data management type, and task management type, each in terms of the characteristics of the target system. This classification will be helpful in distinguishing between complex systems and will provide easier comparison among them. After identification and classification of classes, the class points will describe the complexity level of each class, as determined by the number of external methods, the number of services requested, and the number of attributes. Finally the class points will be calculated by applying a technical complexity factor of the target system. This technical complexity factor was determined by the degree of influence of 18 different target software system characteristics, each on a scale of 0 to 5. The detailed procedure and equations of this measurement are described by Kim *et* al. in [19].

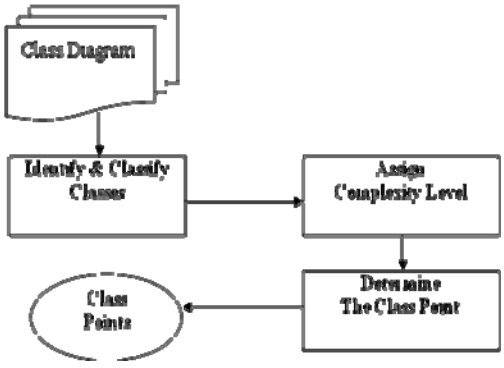

Figure 3: Three steps of the Class Points

This definition of class points provides increased understanding of a system's architectural complexity as compared to earlier work. It is defined by Kim *et* al. in [19] as: Class Points (CP) – The class points of the target system.

$$CP = \Sigma (NOC+NOIR+NOUR+NORR+ NOM+NOCA+NOASS)$$

Where NOC = Number of Classes, NOIR = Number of Inheritance Relationships, NOUR= Number of Use Relationships, NORR= Number of Realize Relationships, NOM = Number of Methods, NOCA = Number of Class Attributes and NOASS = Number of Associations.

Kim *et* al. used different equations to gather fundamental information from class diagrams to recognize its structural complexity These equations can easily be calculated to provide relative information about structural complexity of the class diagrams. The CP, finally, will be calculated by adding up all of the class point values as in equation given in [19].

These UML-based use case points and class points provide the project manager and developer a better understanding of the architectural complexity of the target system. The size measurement UML points can be used to estimate project effort. UML points are calculated by adding use case points and class points. UML points, in turn, are applied on historical data as in Karner's method to generate size and effort estimation [19].

Pros & Cons of UML point method

UML point method requires additional information such as number of inheritance/uses/realize relationships, number of parameters and number of classes. Unlike earlier class point definitions, which used expert decision on component type, complexity level, TDI (Total Degree of Influence), and TCF (Technical Complexity Factor) it automates the procedure through a tool. This approach has similar benefits to use case points described in the previous section. However, it has few weak points when applied to general software projects; it requires comprehensive design information from

class diagrams. This cannot be fulfilled through early design classes. Moreover, its implementation on incremental development or component based systems is not yet validated, which makes it doubtful for non-functional requirements of a CBS.

Method	Pros	Cons
USE CASE POINTS	- Simple, easy to apply method. - Provides 13 technical 8 environmental factors for guidance.	- Lacks information, only counting the number of actors and use cases. - Does not address use case granularity. - Not addressing a change in requirements or use cases. - TCF & EF are outdated.
CLASS POINTS	- Exploiting many UML diagrams. - Method can be totally automated; and its output is in terms of SLOC, so can be compared with well known standard like FP.	- Uses expert decision on component type, complexity level, TDI (Total Degree of Influence) etc, which are highly dependent on the expert's decision. - Uses preliminary data derived by some training experiments. - Its implementation on incremental large scale development like component based systems is yet to be validated. - Goes deep into UML design (i.e.) interaction & state diagrams, hence makes it a late design activity instead of early analysis one.
UML POINTS	- Uses an automatic size measurement tool, the UML point generator, to extract information from use case diagrams and class diagrams. - Size measurement for the UML design specification at the early design phase.	- Current work focuses on class and use case diagrams. - Relies on design information from class diagrams, which cannot be provided through early analysis class diagram. - Yet to applied to incremental large scale developments like Component Based Systems

Table 2: Comparison of UML based estimation techniques

2.4 Other UCP based approaches

Though having few shortcomings in its approach, Karner's work has been continuous motivation for further research. Many researchers have taken his study as a base work for their research. The amount of work done in this domain clearly undermines the potential it puts for future work. It has also been widely accepted and acknowledged by industry professional while considering it as the earliest effort estimation technique. We have also reviewed few techniques in this domain and also summarized a few in the following section.

2.4.1. Use Case transactions & Entity Objects:

Robiolo and Orocosco [13] worked to find an alternative method to estimate effort through use case diagram. Their work primarily focused towards improving the magnitude of relative error in the effort estimation as well as the time at which the estimation of effort is performed. It was done through statistical computations for first objective and use case diagrams for second.

In this approach, two alternative variables were used to express a notion of size: transactions and entity objects and finally effort were estimated through mean productivity value. Evaluation was done on two case studies, which were designed to check if such theoretical proposal was useful to estimate effort in a real life environment. Transactions and entity objects may be calculated from the use cases textual descriptions whereas function points are obtained at a later stage. There are following two formulae for computing size of the application (Size Ap) as given in [13].

Size Ap. = ∑ Number of Use Case Transactions.

Size Ap. = ∑ Number of Module Entity Objects.

The improvement of effort estimation was achieved by using the size notion transactions and the technique mean productivity value for applications within the same environment. This restriction is explained by the fact that productivity is a characteristic

that belongs to a set of persons under certain conditions, thus the estimations can only be applied to projects carried out under similar circumstances. Moreover, it is possible to track a project because transactions and entity objects can be easily counted in each primary use case and that they may be used to measure the changes imposed by the client throughout the development process. Thus, productivity—quotient between size and effort—may be easily calculated as changes occur. Such productivity value may be compared to the productivity value used to make the original estimation in order to measure the deviation resulting from the modifications inherent to a real life development cycle. The validation of transactions and entity objects as size metrics were left for future work.

2.4.2. Use case Size Points & Fuzzy USP

Another work by Braz and Vergilio [3] exploit the information available through use case diagram and proposes a metric, named USP to deduce its size. Another metric proposed was named FUSP that uses the concepts of the Fuzzy Set Theory to create gradual classifications of the sections of a use case, eliminating some problems when classifying elements through tables of categories. The approach spans through 9 steps and computes use case size points (USP) as

$$USP = UUSP * (FTA - FAA)$$

Where UUSP is unadjusted use case size points, FTA is technical factor adjustment and FAA is environmental adjustment factor. FTA and FAA are determined through a new set of technical (14) and environmental (5) factors [3].

The other metric FUSP (Fuzzy Use Case Size Points) is computed in two steps. In the first step, the functionalities present in the use case receive a continuous classification. In the second step, Fuzzyfication and Defuzzyfication of the linguistic terms are performed.

They compared four metrics namely FP, UCP, USP, FUSP and emphasized on the correct measure of use case granularity. The validation was done on MIS like real time project. Validating on complex systems which are incrementally developed or large scale component based development was not addressed in their approach. In their future work, they mentioned some improvements could be done in metric FUSP by enlarging the number of elements considered or increasing the difference among the complexity of the categories to take better use of the Fuzzy Theory.

2.4.3 Use Case Points

In another approach based on UCP, Diev reviewed several situations [11] that occur when a use case model is used for estimation. He emphasized that to provide accurate use case based estimate it is necessary to take into consideration the existing system's design as well as the details of the project.

He extended the earliest work done by Karner but also correlated it with system's complexity and the supplementary efforts needed for various management activities. His work was primarily based on application of UCP in financial domain. Alike other researchers, he also addressed the issue of use case granularity with the help of use case transactions. According to his approach, UCPm i.e. modification to use case points is given in [11] as

$$Size=UUCP * TCF \text{ and}$$

$$Effort=size * EF * BSC * R + supplEffort$$

Where UUCP is unadjusted use case points, TCF is technical complexity factor, EF is environmental factors (as given in Karner's method [18]) and BSC is base system's complexity, R is Person-Hour per use case point and SupplEffort is the supplementary efforts required to manage the system.

This work was an extension of Karner's method only, yet it demonstrated the issues like use case granularity, complexity of the system and the supplementary efforts

required for management activities like configuration management, project management etc.

2.4.4 Use Case Points

Extending the UCP approach, Anda *et* al. [2] investigated the application of use cases in estimating software development effort in a multiple case-study, in which four companies were chosen to actually develop a system based on the same requirements specification. The teams from the four companies had very similar qualifications, and the functionality of the four resulting systems was almost equivalent. The teams followed different development processes and placed different emphasis on the quality of the code. Employing use case point method, necessary effort was estimated to 413 hours where the actual effort of the development teams lied between 431 to 934 person hours [2].

In this work too, it is addressed that there is some adaptation required to handle the increase in effort due to development process and quality requirements on the code. The other issue emphasized was again the granularity of the use case. They also discussed it with the number of transactions in a use case. They had taken average complexity for TCF and EF and in some way sidelined the need of these. However, the case studies in question were explicitly made out of the scope of component based development. Hence, it did not concern about the nature of technical and environmental factors of component based development.

2.4.5 Use Case Points

Mohagheghi *et* al. [24] adapted and tested use case points method on a large industrial system with incremental changes in use cases. With main assumptions of use cases potentially being used as a measure of the size of a system, changes in these to be used as a measure of changes in functionality between releases. They implemented the UCP method in correlation with COCOMO-II method to predict the effort required to build a large and incremental real time system. Their method was independent of

automated UCP method tools, paradigms or programming languages, and could promote high quality use cases. This method is cheap, transparent and easy to understand and also suitable when development is being outsourced. They proposed an overhead factor (OF) to be employed for complex systems. They observed the granularity of each use case with the number of transactions it contains, the effect of changes it incorporates in each of the iteration of development cycle, and computed the efforts required in the form of primary (new work) and secondary (modification) efforts [24].

Though addressing much of the escaped issue, it does not correspond well with component based system development. Again, the TCF and EF are arbitrarily taken as average or rather with no effect on size of the system. It also excluded the influence of nonfunctional requirements but put forward to be included in the technical factors. Proposed overhead factor (OF) was used to estimate the total effort based on effort used to develop the system before system test and was empirically derived to be 2. This empirical observation relies on judgments made by researchers and is subjected to further adjustments [24]. As suggested, future studies can help to understand the degree of modification in incremental development of a system (use cases, code and integration costs), and how the method works on other types of systems.

2.5 Metrics for CBS

There are many metrics available for component measurement. Encompassing static complexity metrics to dynamic behavioral metrics, these metrics can be applied on individual components. In [26], Narasimhan and Hendradjaya defined two sets of metrics, static and dynamic complexity of the component. In another similar work, while defining metrics for component quality, Cho *et* al. [8] used a completely different context for component complexity and quality. Component complexity, Component packaging, Component interaction density, Component plain complexity, component dynamic complexity, Component integration complexity, component time-to-market complexity are few metrics from [8,26], which can be used for different complexity measures of a component.

Though there are so many metrics available for component measurement, very few address the sizing of components. Gill and Grover suggested component size metric to be the number of use cases realized by the component [15]. They also discussed SLOC as the general metric for components sizing but also stated it as inefficient. Moreover, we know that reusing the same component or code segment should not account for new cost or efforts required for the system in question. Components in CBSD can come from any source, be it third party vendors, in house libraries or development from scratch. It is not always possible to know the exact sizing of a component so we bank on the concept of use case based sizing for component based systems as well.

Chapter 3

Problem Statement

In the last chapter, we discussed the approaches that make use of UML diagrams for effort estimation of the software based on object oriented techniques. CBSD, however, has different issues while calculating efforts required for development. In the following sections, we shall precisely discuss the gaps in the current work, define the statement of problem and analyze the same in the preview of possible solution.

3.1 Gaps in research

During our study, we underlined following gaps in various research works done in the domain of UML based effort estimation. These are identified while considering present technological advances, process maturities and the fields never touched upon.

3.1.1 CBS not addressed

While reviewing through the current studies, it was observed that the work is centered towards domain specific systems only. In other words, the specific sizing metrics are not present for a process model, which could cover larger variety of domains. Researchers have worked toward estimating efforts of an application based on object oriented concepts because they could be modeled through UML diagrams. Though in one instance, it has been scaled to cover incremental development as well but the nature of component based systems is slightly more complex than this. A component based system essentially follows distributed and location transparent approach. More often, component based systems are object oriented too and can be represented through UML. They ask for standardized processes and procedures and hence incorporate their own complexity

issues. Functional aspects can always be included when we estimate the size of the system but concerns to non-functional requirements are always left untouched. This account for the need of a method or even a adaptation of a established method which may consider various issues of component based system development.

3.1.2 TCF and EF outdated

In widely accepted and worked upon method of Use case point, Technical complexity factor and Environmental factors play as adjustment or tuning factors. Technical complexity factors describe the multiplying factor by which use case points should be raised up for incorporating various functional and nonfunctional issues. Originally specified by Karner, these factors have been modified, left unchanged and sometimes normalized to produce little or no effect on respective studies. While considering present scenario of component based system development, these factors seem to be outdated and irrelevant with its decomposed, distributed and concurrent nature. Hence, there should be a revision of TCF and EF in order to incorporate it into the estimation process of a component based system.

3.1.3 No critical review, summary or comparison of work done is present

Till date, research in UMl based effort estimation demonstrates its interest in the base theory being extended or improved. People have reviewed the inline approaches used for effort estimation or defining new metrics. Thorough comparison of different approaches is as such nowhere present. There should be some normalized comparison for these approaches so that future directions can be outlined.

3.2 Statement of problem

Based on our literature survey, we define our area of work as “UML based effort estimation for Component Based Systems”. In this work, we analyze the existing work

done and its shortcomings and incorporate few modifications into UCP approach along with a modified set of TCF and EF in the context of CBS.

3.3 Analysis of problem statement

In order to solve the said problem statement, the sequence of work is as defined below:

1. Review, summarize and compare existing work
2. Suggest modified and/or new technical complexity factors and environmental factors
3. Taking inputs from industry professionals, research scholars and academicians to define their rating and weight criterion.
4. Specifying the classification and rating criteria for new set of TCF and EF with proper guidelines.
5. Implementing revised approach on two case studies.

Chapter 4

Proposed Framework

We have already identified few issues while discussing gaps in research. We had also discussed how to solve those issues in the analysis sections of previous chapter. In the following sections, we shall propose the framework for estimating efforts through use case diagrams for component based systems. Considering non-functional requirements as intrinsic features of a component based systems, we have outlined new technical complexity factors and for standardized procedures followed in component based system development, we have modified environmental factors too. In the following sections we shall define them and also give the guidelines for using them in the process of predicting the effort required for software development.

4.1. Defining UC granularity

a. Each main/ alternate/ exceptional flow to be counted as a use case
b. All include & extend relations are to be separately counted.
c. A use case more than 15 transactions to be split accordingly for counting purposes.

4.2. Analyzing use case diagram

Use cases are the earliest output of requirement modeling. Getting deep into class diagram and other UML diagrams shall account for higher level of design details. We should not go in this direction as it deviates from the original motive of early decision about effort computations.

4.3. Determining modified complexity factors

So far, we have discussed that quality of component based system largely depends upon quality of the components forming that system. Also, a component based system is assumed to be location independent and a decoupled system. Considering these and other such non-functional requirements, following are the revised sets of technical and environmental factors.

4.3.1 Technical factors

	Factor	Weight
T1	Distributed system	0.5
T2	Component performance	1.5
T3	End-user efficiency	1
T4	Internal complexity	1.5
T5	Component reusability	1
T6	Usability	1
T7	Maintainability	2
T8	Security	1.5
T9	Third party integration	1.5
T10	User education	1
T11	CASE tools	-0.5
T12	Interface complexity	1

Table 3: Revised Technical Complexity Factors

4.3.2 Environmental factors

	Factor	Weight
E1	Domain experience	1
E2	Stable requirements	2
E3	Part time workers	-0.5
E4	Technology newness	1
E5	Component model	1
E6	Internal library resources	-0.5
E7	Organization maturity	0.5

Table 4: Revised Environmental Factors

4.4. Computing efforts

In this study, we have adapted Karner's method as discussed in section 2.1 to estimate the effort required. Following equations are used as in sequence to compute unadjusted actor weight (UAW), unadjusted use case weight (UUCW), unadjusted use case points (UUCP), technical complexity factor (TCF), environmental factor (EF) and finally use case points (UCP) respectively. The UCP thus got are multiplied by **"r"**, which is PersonHour per UCP as established in the industry ranging from 18 to 36 PH/UCP. For our computation sake, we have taken this value to be 18.

1. $UAW = \sum Actor_i * W_i$
2. $UUCW = \sum UC_i * W_i$
3. $UUCP = UAW + UUCW$
4. $TCF = 0.6 + (0.01 * TFactor)$

5. *EF=1.4+(-0.03*EFactor)*

6. *UCP=UUCP*TCF*EF*

4.5 Guidelines for complexity factors

In this section, we describe the criterion on what the different values can be assigned to respective technical or environmental factor. The classification largely varies from 3 to 6 categories depending upon the nature of the complexity factor. The value for each of the classification varies from 0 to 5.

4.5.1 Technical complexity factors:

Distributed System

It is believed that component based systems should follow a distributed, reconfigurable and location transparent architecture; if not perfectly distributed. Hence, this technical factor becomes intrinsic feature of a component based system and its overall effect on cost should be minimal.

Rating	Description
1	Virtually centralized, all components integrated to one place e.g. Windows OS
2	A mix of local and remote components, largely a localized arrangement
3	Majority of the components are location transparent
4	Highly distributed, nearly all components are remotely located like Web services
5	Extremely distributed environment, databases are also distributed.

Table 5: Guidelines for Distributed system

Component Performance

Performance of a component based system is largely depends upon performance of individual components. However, as they are composed into a system with some adaptation and integration efforts, it becomes necessary to count on minimal performance of a component. Hence, it is weighted with good effect on effort required.

Rating	Description
0	Not mandatory to have very efficient components
1	Little effect. Few components required to be efficient.
2	Somewhat important. Overall system performance is necessary.
3	Important but can be compromised for availability e.g. Web-based applications
4	Very much performance based system.
5	Highly important. Zero-tolerance systems like real time systems.

Table 6: Guidelines for component performance

End User Efficiency

How much efficiency is required from the end user so that he can operate the system very well? It is certainly affected by the complexity of the system. A 0 value to this factor means the system is easy to use and no efficiency required from a user whereas a 5 value indicates good technical skills required operating the system.

Rating	Description
0	No effect.
1	Little knowledge is enough.
2	Good knowledge is required to operate efficiently.
3	Professionally trained user is required.
4	An experienced and trained professional is required.
5	Highly skilled user is required.

Table 7: Guidelines for end-user efficiency

Internal Complexity

How much complex is the internal working of the system? A 0 value means easy-to-integrate components are available that has very simple interfaces as well as simple implementation (which is over-optimism). An average value of 3 indicates a good complexity with overall easy-to-integrate components.

Rating	Description
0	Very simple system
1	Few complex components but largely a simple implementation.
2	Complex system.
3	Internally complex but easy-to-integrate
4	Sophisticated system. optional or minimal efforts for integration.
5	Highly complex system. Integration efforts may be necessary.

Table 8: Guidelines for internal complexity

Component Reusability

It indicates the amount of code developed from scratch to be included into the system and determines the extent of code reusability. Higher the reusability desired, higher shall be the complexity.

Rating	Description
0	Not important.
1	Minimal influence of reusability.
2	Somewhat important. Overall design should be reusable, if not components.
3	Reusability desired. Few components should adhere to it.
4	Relatively high reusability desired. Coupling is also important factor.
5	Highly cohesive components to support maximum reuse.

Table 9: Guidelines for component reusability

Usability

The ease of use is said to be usability. How much usability is required determines how many GUIs and help features are to be provided in the system. A good, easy to use interface with context sensitive help should be rated average i.e. 3 here.

Rating	Description
0	Not so important
1	Not so feature oriented. No specific requirements.
2	Usable, can be aided by documentation only.
3	Fairly usable, provides help features.
4	Highly user friendly e.g. web application
5	A very good user friendly environment with all help features and graphics like OS.

Table 10: Guidelines for usability

Maintainability

It is the ease with which, we can implement the change in the system. A high maintenance system shall account for overall high development efforts. In case there is a frequency of changes required after the system is deployed, it should account for high maintenance. It also describes volatility of the components i.e. how frequently a component is changed. Hence its effect on effort prediction should be considered very important.

Rating	Description
0	Highly decoupled system.
1	Reconfigurable low coupling system.
2	Reconfigurable, documented and standardized interfacing.
3	Medium coupling, changes would not be largely localized.
4	Strong coupling between components.
5	Tightly coupled system. High maintenance required.

Table 11: Guidelines for maintainability

Security

The amount of security required is bound to increase in case of component based systems. As the components integrated could come from third party vendors, whose security features may not be that much reliable as compared to their repository counterparts, it becomes utmost important to consider this aspect on high note.

Rating	Description
0	Not so important.
1	Little features of security sufficient. E.g. User login.
2	Security desired. Data transfer should be controlled and managed.
3	Secure system. Authentication & authorization required over the network as well.
4	Considerable security required. User accounts, privileges are to be maintained. E.g. OS & databases.
5	Security critical environment, e.g. payment gateway etc.

Table 12: Guidelines for security

Third Party Integration

How many third party components are to participate in the system? Large number of COTS though reduce the overall development efforts yet increase in adaptation and integration cost. This factor shall definitely depend how many components we are successful to gather from in-house or shared libraries.

Rating	Description
0	No third party components. All to be recovered from in-house libraries.
1	Maximum of in-house components.
2	Mixture of third-party components and in-house components.
3	Maximum third-party components, few to be developed in-house.
4	Very few to be integrated. Mostly to be developed.
5	Maximum to be developed and integrated with few third-party components.

Table 13: Guidelines for third party integration

User Education

Does the user require to be educated to operate the system? How complex is the system for user perspective is the key for this factor. Here, the extent of separate user training required is taken into consideration.

Rating	Description
0	No user education required.
1	Documentation is sufficient.
2	Documentation with embedded help files is sufficient.
3	An orientation program with practical demonstration is required.
4	Separate user-training is needed.
5	User training, online help and possibly certification required.

Table 14: Guidelines for user education

CASE Tools

A development process is very much eased out with the help of Computer Aided Software Engineering (CASE) tools. If the availability is high, overall effort required to build the system are bound to be low. Hence, the factor shall account accordingly.

Rating	Description
0	No CASE tools usage.
1	IDE used.
2	IDE with documentation software.
3	IDE with documentation and diagram tools
4	CASE tools for design, development and testing.
5	Maximum use of CASE tools. Includes tools for configuration management & project management.

Table 15: Guidelines for CASE tools

Interface Complexity

How much complex is the interfacing of the components? More complex interfacing shall include high testing efforts and also high maintenance efforts.

Rating	Description
1	Easy interfacing.
3	Complex interfacing, yet manageable without much efforts.
5	Highly complex interfacing. Requires adaptation code.

Table 16: Guidelines for interface complexity

4.5.2 Environmental factors

Domain Experience

It determines how much the people working in the project are familiar with the domain and technical details of the project? Least efforts shall be required for already educated team to explain the problem statement.

Rating	Description
0	No experience
1	Simple awareness of the idea
2	Few are aware of the domain, very few with knowledge in the domain.
3	Knowledge of the domain. Few are experienced as well.
4	Most of the team has either experience or knowledge of the domain.
5	All are experienced.

Table 17: Guidelines for domain experience

Stable Requirements

Is the client clear about what he wants? The most important factor for stable requirements is clear expectations from client. For a client with highly volatile nature, this factor is to be given highest weight.

Rating	Description
0	Quite stable.
1	Very few changes in the requirements.
2	Little changes approximately 20% in meetings with customer.
3	Changes in requirement amounts upto 50%.
4	Highly volatile requirements, changes around 60-70%.
5	No stability. Customer does not have any idea of what is actually needed.

Table 18: Guidelines for stable requirements

Part Time Workers

Part time workers like consultants etc. add the value into development process. They input expert knowledge for the shortest duration they stay into the process. This factor shall be computed accordingly.

Rating	Description
0	No part-time staff.
1	Very few part-time consultants.
2	Approximately one-third of team is composed of part-time members.
3	Approximately half of the team is of part-time experts.
4	Mostly are part-time experts.
5	All are part-time staffers.

Table 19: Guidelines for part-time workers

Technology Volatility/Newness

A potential cause to software risk is the newness of the technology being implemented. It is also affected by its volatility, which implies the frequency with which it keeps changing. Hence, this factor is taken with strong impacts on efforts required to develop the system.

Rating	Description
0	Easy technology. One week is required to pickup.
1	Atleast two weeks required to be familiar with the technology.
2	A month may be required to be familiar.
3	Special training for the technology is required.
4	Special training along with help during the project is required.
5	Difficult. Needs only skilled and experienced people.

Table 20: Guidelines for technology volatility

Component Model

In the absence of a defined criterion for component certification, the quality of a component can be determined by the quality of component model it follows. The standards, practices and procedures defined in a component model warranties the quality of component development. Lowest value indicates no component model followed and highest means a detailed, standardized and accepted component model.

Rating	Description
1	No component model
3	Old but established component models like COM etc.
5	Latest and sophisticated component models are followed.

Table 21: Guidelines for component model

Internal/Shared Library Resources

A successful component based development largely depends upon availability of suitable components. Presence of internal or shared library resources definitely lowers the efforts required in searching, retrieving and selecting components with respect to the given requirement set. A 0 value indicates no such resources whereas the maximum value is indicative of a detailed repository.

Rating	Description
0	No shared resources.
1	Internally managed collection of components.
2	Documented collection of reusable components.
3	Small repository with shared resources and/or libraries.
4	Management initiative towards repository management.
5	Dedicated infrastructure for repository management.

Table 22: Guidelines for internal or shared library resources

Organization Maturity (Certification/CMM Level)

Organization maturity in terms of its quality certification and/or CMM level defines a minimum guarantee level for quality of development process. A highly placed organization is assumed to have quality skilled staff, defined and repeatable processes along with procedures for defect prevention and continuous improvement. This surely reduces the dependability on experts, reduced effort requirement for quality software development.

Rating	Description
0	No certification
3	Any CMM level with ISO certification
5	CMM level 5 certified organization.

Table 23: Guidelines for organization maturity

Chapter 5

Evaluation and Validation

To evaluate the proposed framework, we have taken two classroom projects as case studies. These projects are component based systems; developed in latest technologies and follow popular component models.

First case study is a web services based eShopping web site that is composed of multiple interfaces, has different user contexts and many non-functional requirements. The desired system should allow a customer to browse the catalog through internet, make purchases online, and receive products at home. Customer should be allowed to make payments through credit or debit card or through an online bank account. The payment should be secured through external agency. It should allow a sales person to manage the catalog; an administrator to manage user accounts; shipping clerk to update inventory.

The information requirement of the system is to generate order form, product selections, customer information, billing information and shipping information. The external interfaces required for this system include internet, payment gateway, security authorization agency and the shipping agency. Non functional requirements for this application include availability, usability, performance, security, portability and performance. The system is to be built through VB.Net and ASP.Net on .Net framework.

Following is the detailed procedure for computing UUCP for this case study.

	actors		type	weight		
1	customer		simple	1		
2	sales_person		simple	1		
3	shipping clerk		simple	1		
4	credit verification company		complex	3		
5	security verification company		complex	3		
6	administrator		simple	1	**UAW**	**10**

	use case	transactions	type	weight		
1	Login	3	simple	5		
2	add to shopping cart	2	simple	5		
3	view shopping cart	1	simple	5		
4	browse products	1	simple	5		
5	perform product search	2	simple	5		
6	Register	2	simple	5		
7	update customer info	1	simple	5		
8	Checkout	7	complex	15		
9	update product info	3	simple	5		
10	update inventory	3	simple	5		
11	ship product	4	average	10		
12	verify card	7	complex	15		
13	enter shipping & Credit Card info	3	simple	5	**UUCP**	**90**

	TCF	weight	value	tcf(i)		
1	Distributed system	0.5	4	2.00		
2	Component performance	1.5	3	4.50		
3	End-user efficiency	1	3	3.00		
4	Internal complexity	1.5	3	4.50		
5	Component reusability	1	4	4.00		
6	Usability	1	4	4.00		
7	Maintainability	2	2	4.00		
8	Security	1.5	5	7.50		
9	Third party integration	1.5	3	4.50		
10	User education	1	1	1.00		
11	Case tools	-0.5	4	-2.00	**tfactor**	**40.00**
12	Interface complexity	1	3	3.00	**TCF**	**1.000**

	EF	weight	value	Ef(i)		
1	Domain experience	1	4	4.00		
2	Stable requirements	2	2	4.00		
3	Part-time workers	-0.5	1	-0.50		
4	Technology newness	1	0	0.00		
5	Component model	1	5	5.00		
6	Internal library resources	-0.5	3	-1.50	**efactor**	**13.50**
7	Organization maturity	0.5	5	2.50	**EF**	**0.995**

The second case study is a CRM based customer support site that is composed of few external interfaces, has different user contexts for clerk, management and marketing and also requires extensive management reporting. The desired system must allow a customer to browse the catalog through internet, make online orders, and receive items through shipping. Customer should be allowed to make backorders, which could be traced back. The marketing clerk should be allowed to enter special promotional schemes and produce reports. It should allow a sales clerk to manage the catalog, customer information and order, backorder information. There should be provision for management reporting for all users' contexts.

The information requirement of the system is to generate order form, item lookups, order and backorder information, billing information and shipping information, catalog information and customer information. The external interfaces required for this system include internet, merchandising agency and the shipping agency. Non functional requirements for this application include availability, usability, portability and performance. The system is to build at Java platform following EJB component development methodology.

Following is the detailed procedure for computing UUCP for this case study.

	actors	type	weight		
1	customer	simple	1		
2	clerk	simple	1		
3	management	simple	1		
4	merchandising	simple	1		
5	marketing	simple	1		
6	shipping	simple	1	**UAW**	**6**

	use case	transactions	type	weight		
1	create new order	3	simple	5		
2	lookup item availability	2	simple	5		
3	update order	3	simple	5		
4	lookup order status	2	simple	5		
5	create order return	3	simple	5		
6	record back-order	3	simple	5		
7	record order completion	1	simple	5		
8	provide catalog info	3	simple	5		
9	maintain customer info	5	average	10		
10	create charge adjustment	3	simple	5		
11	distribute promotional package	3	simple	5		
12	manage catalog	5	average	10		
13	create spl promotion	3	simple	5		
14	maintain prdocut info	5	average	10		
15	produce report	7	complex	15	**UUCP**	100

	TCF	weight	value	tcf(i)		
1	Distributed system	**0.5**	3	1.5		
2	Component performance	**1.5**	3	4.5		
3	End-user efficiency	**1**	3	3		
4	Internal complexity	**1.5**	2	3		
5	Component reusability	**1**	3	3		
6	Usability	**1**	3	3		
7	Maintainability	**2**	2	4		
8	Security	**1.5**	3	4.5		
9	Third party integration	**1.5**	3	4.5		
10	User education	**1**	1	1		
11	Case tools	**-0.5**	3	-1.5	**tfactor**	**33.5**
12	Interface complexity	**1**	3	3	**TCF**	**0.935**

	EF	weight	value	tcf(i)		
1	Domain experience	1	3	3		
2	Stable requirements	2	3	6		
3	Part-time workers	-0.5	2	-1		
4	Technology newness	1	0	0		
5	Component model	1	2	2		
6	Internal library resources	-0.5	3	-1.5	**efactor**	11
7	Organization maturity	0.5	5	2.5	**EF**	1.07

The following table summarizes the results obtained through revised technical and environmental factors with respect to component based systems. Both of the projects have different functional and non-functional requirements, nature of development and operating requirements.

	UAW	UUCW	UUCP	TCF	EF	UCP	r	efforts (PH)
eShopping	10	90	100	1	0.995	99.50	18	**1791**
CRM	6	100	106	0.935	1.07	106.05	18	**1909**

Table 24: Evaluation results

To validate our results, we have taken the case study used by Anda *et* al. [2]. In their work, they took minimal effect of technical and environmental factors while computing efforts and got the estimate of 413 person hours as opposed to 431 to 943 hours of actual efforts spent by different organization. The difference of actual efforts comes with the development methodologies followed by different organizations. This indicates that more sophisticated, reusable, documented and quality development procedure required more efforts in building the same system as compared to a similar development with less formalized, quality and adhoc development process.

Following table summarizes the comparison of results with that of [2].

	UAW	**UUCW**	**UUCP**	**TCF**	**EF**	**UCP**	**r**	**efforts (PH)**
Anda *et* al.	7	50	57	0.6	0.605	20.69	20	**413**
Our approach	7	50	57	0.795	1.085	49.17	18	885

Table 25: comparison of results

We have already incorporated different non-functional requirements into technical complexity factors along with those environmental factors that result into quality software development. Assigning appropriate weight to these factors resulted into a better estimation for the same project. Our estimate varied by an acceptable amount of 6.15%, which is quite satisfactory when compared to the similar estimation technique adapted by Anda *et* al. [2]. Anda *et* al. compared their results with the least value of 431 person hours among for value set whereas our estimate approximate two upper end values in the same set. In this way, we can say that revised complexity factors and use case granularity guidelines have given far more accurate results than existing work.

Chapter 6
Conclusion and Future Work

In this work, we aligned the established and popular UCP method with widely acknowledged and popular process model i.e. component based system development. Unlike other development strategies, CBSD inherently incorporates functional and non-functional requirements into the desired system. We addressed these issues with the help of newly defined technical factors and environmental factors. The results from this study support previous claims that the UCP method may support early estimation of software development effort for CBS as well. We also addressed the specific concerns of CBSD in the context of early effort estimation and the encouraging results acknowledge the work. The future work in this context could be development of an automated tool for fetching information directly from UML diagrams and to apply this framework on large industrial development. However, during this study we encountered few issues that are not covered under effort estimation but directly affects the overall cost of the system and are worth mentioning here.

6.1 Un-answered issues

We incorporated few observations like component model and organization maturity in TCF and EF. However there are some unanswered issues what we encountered while doing the study. These are related to the other costs of searching and retrieving, adaptation and maintenance and the project management costs. These are the factors that affect the overall cost of development but cannot be included as effort required to build the system. They are discussed as followed.

6.1.1 Search & retrieval cost

The very first step in CBSD is finding the potential candidates for component reuse. This requires browsing in-house and shared libraries and also checking with third party vendors for the components that match the requirement set upto an acceptable extent. Browsing may include thoroughly inspecting the surrogates, component signatures and/or the documentation available with the component. The whole process essentially depends upon the type of cataloging used for components in the libraries. COTS component are also offered with detailed documentation, which helps in in-depth analysis of the components with respect to stated requirements. The components thus searched are retrieved from the libraries and/or procured from respective vendors. These retrieved components are further filtered on the basis of their contextual distance from requirement set. Components thus extracted are termed as "selected" and further passed on to adaptation step, where requisite interface compatibilities are framed with the help of glue code or wrapper code (wherever required).

Failure and success of this step completely depends upon the quality of surrogates and the documentation available with the component. It remains easy to retrieve from in-house libraries for the components of domain matching to the system in question. The functional information is easily available with the organization and hence efforts required in searching and retrieving are essentially lesser. Lesser number of components retrieved shall increase in the number of components developed and in turn, directly affect the overall cost of the system. Though that would definitely increase the scope of libraries and may help in sophisticated development of current and future project; the very purpose of cost minimization shall be defeated for current assignment.

There is as such no address for such technical issue in terms of cost involved. There are available metrics for cost of component repository management or the component itself but not for component search and retrieval. Work is needed to predict the efforts required to search and retrieve components per stated requirement. Otherwise, there may be a formula for cost of this step, which defines cost as the function of number of hits and misses for each requirement considering repository size as well.

6.1.2 Integration cost

Cost of integrating components in component based system is very vital and depends on the organization's specific internal and external factors. The COCOTS model [1] is based on the general form of COCOMO model and uses alternative set of cost drivers to predict cost of component integration. Five groups of factors influence the component integration cost as indicated by set of drivers: product and vendor maturity, customization and installation facilities, usability, portability and previous product experience. Minkiewicz [26] identifies the major cost drivers based on the CBSD life cycle activities and cost associated with component inclusion, integration and maintenance. The cost drivers are identified for the evaluation and selecting of component, purchase of the selected components, integration of the component into the software system, system level testing to establish that all of the component work together correctly and evaluate component upgrades and upgrade component solutions required. However, these proposed cost drivers need analysis for their validation. Mili *et* al. [25] presented an integrated cost model for software reuse. The paper discusses the economic point of view of the software reuse. The proposed cost model evaluates software reuse in economic terms and evaluates them on the basis of investment analysis functions. Ellis [12] proposes seventeen cost drivers (e.g. vendor maturity, product maturity, portability, previous product experience, etc.) and describes the cost of component integration based on these cost drivers. He emphasized that this integration cost is to be addressed while accounting for the total cost of the system.

6.1.3 Management cost

In CBSD, the management phase costs occur throughout the component based system life cycle including the same during evolution and maintenance phases. The most important characteristic of CBSD is the separation of its interfaces from implementations. Since large components have complex interfaces, they are prone to changing requirements, conditions and different versions of components etc. For example, the context based constraint (CoCon) mechanism specifies one requirement for a group of

indirectly associated components that share a context, which refers the state, situation or environment of each component. The mechanism requires that monitoring points be determined for each of them [15,31]. Reasoning about the behavioral composition is also important. Stutzke [33] presented a cost estimation model based on volatility of the components. The component volatility is defined as the frequency with which a component vendor releases new version. He proposed a formula for the calculation of cost associated with using component products as

Additional Cost = Component volatility * architectural coupling * apparent interface size * (Cost of screening the component + cost of making changes to the component)

From our earlier discussions, we again emphasize that these costs are not directly part of the effort required to build the system but certainly worth counting while computing overall development cost of the system. Few cost models, however, incorporate the integration cost for the system and also perceive the system in context of component reuse, yet it does not discuss upon search, retrieval cost or the cost of volatility of the component. In future, we strongly wish to put these three costs and others (if any) in line while calculating cost required for a component based system.

References

[1] Abts C., Boehm B., Clark B., “COCOTS: A COTS software integration cost model”, proceedings of ESCOM-SCOPE 2000 conference, April, 2000.

[2] Anda B., Benestad H. C., Hove S. E., "A multiple-case study of software effort estimation based on use case points", 2005 International Symposium, p 407-416, 2005

[3] Rodrigo B. M., Vergilio S. R., “Software Effort Estimation Based on Use Cases”, proceedings of the 30th Annual International Computer Software and Applications Conference, p 221 – 228, 2006.

[4] Brereton P., and Budgen D., “Component-based systems: A classification of issues” IEEE Computer 33, 11, pp 54–62, 2000

[5] Lionel B., Sandro M., and Basili V. R., “Property-Based Software Engineering Measurement,” IEEE Transactions on Software Engineering, Vol. 22, No. 1, pp. 68-86, January 1996.

[6] Cechich A., Vallecillo A., Piattini M., “Assessing component based systems, Component Based Software Quality”, LNCS 2693, pp 1–20, 2003.

[7] Chidamber S. R. and Kemerer C. F., “A Metrics Suite for Object Oriented Design,” IEEE Transactions on Software Engineering, 20(6), pp. 476-493, June 1994.

[8] Cho E.S., Kim M. S., Kim S. D., "Component metrics to measure component quality", Proceedings of Software Engineering Conference, APSEC 2001, pp 419- 426, 2001.

[9] Cockburn Alistair, “Writing Effective Use Cases”, Addison-Wesley, 2001.

[10] Costagliola G. and Tortora G., “Class Point: An Approach for the Size Estimation of Object- Oriented Systems,” *IEEE Transactions on Software Engineering*, Vol. 31, No. 1, pp. 52-74, Jan. 2005.

[11] Diev S., “Use cases modeling and software estimation: applying use case points”, ACM SIGSOFT Software Engineering Notes, Vol 31, Issue 6, Pages: 1 – 4, November, 2006

[12] Ellis T., “COTS integration in software solutions—A cost model”, In Proceedings of Systems Engineering in the Global Marketplace, NCOSE international symposium. 1995, St Louis, MO, USA.

[13] Gabriela R. and Orosco R., "Employing Use Cases to early estimate effort with simpler metrics", Innovations in Systems and Software Engineering, Volume 4, Number 1, pp. 31-43, April 2008.

[14] Gao J. Z., Tsao H.-S.J., Wu Y., “Testing and quality assurance for component based software”, Artech House, 2003

[15] Gill N. S., Grover P. S., “Component based measurement: few useful guidelines”, ACM SIGSOFT Software Engineering Notes 28 (6), pp 1–6, 2003.

[16] Gill N S., Grover P. S., “Software Size Prediction Before Coding,” ACM SIGSOFT Software Engineering Notes, Vol. 29, No. 5, pp. 1-4, September 2004.

[17] Jacobson I., “Object-Oriented Software Engineering. A Use Case Driven Approach”, Addison-Wesley 1993.

[18] Karner G.. “Metrics for Objectory”. Diploma thesis, University of Linköping, Sweden. No. LiTH-IDA-Ex-9344:21, December 1993.

[19] Kim S., Lively W., Simmons D., “An Effort Estimation by UML points in the early stage of software development”, proceedings of the 2006 international conference on software engineering research & practice, p 415-421, June, 2006,

[20] Kusumoto S., Matukawa F., Inoue K., Hanabusa S., and Maegawa Y., “Estimating Effort by Use Case Points: Method, Tool and Case Study,” Proceedings of the 10th International Symposium on Software Metrics METRICS’04, pp. 292- 299, September 14-16, 2004.

[21] Leicher A., Bubl F., “External requirements validation for component based systems”, In CAiSE 2002, LNCS 2348, 2002, Springer, Berlin.

[22] Mahmood, S., Lai, R., Kim, Y.S., Kim, J.H., Park, S.C. and Oh, H.S., “A survey of component based system quality assurance and assessment”, Information and Software Technology 702 47, pp 693–707, 2005.

[23] Massimo C., Giuseppe S., “Fast&&Serious: a UML based metric for effort estimation”, 6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, June, 2002.

[24] Mohagheghi P., Anda B., Conradi R., "Effort estimation of Use Cases for incremental large-scale software development", International Conference on Software Engineering (ICSE), pp. 303 – 31, 2005.

[25] Mili A, Chmiel S F, Gottumukkala R , Zhang L, “An integrated cost model for software reuse”, In Proceedings of the 22nd international conference on Software engineering, pp 157–166, 2000

[26] Minkiewicz A. F., “The real costs of COTS”, In Proceedings of IEEE Aerospace Conference, pp 2863–2869, 2001.

[27] Narasimhan V. L., Hendradjaya B., “Theoretical considerations for software component metrics”, Proceedings of World Academy of Science, Engineering and Technology, Volume 10, pp 165-170, December 2005.

[28] Pressman R. S., “Software engineering A Practitioner’s Approach”, Fifth Edition, Tata McGraw Hill.

[29] Ravichandran T. and Rothenberger M. A., “Software reuse strategies and component markets”, Communications of the ACM, August 2003/Vol. 46, No. 8

[30] Simao R.P.S., Belchior A.D., Quality characteristics for software components, Component Based Software Quality LNCS 2693, pp 184–206, 2003.

[31] Smith J., “The estimation of effort based on use cases”, Rational Unified Process (RUP) white papers, 1999

[32] Smith R. K, “Effort Estimation in Component-Based Software Development: Identifying Parameters”, In the proceedings of the 36th ACM Southeast Regional Conference, Marietta, Georgia, USA, pp 323-325,1998.

[33] Stutzke R., “Costs impact of COTS volatility”, In Proceedings of Workshop on COCOMO 2.0. 1995

[34] Zhihao C., Hihn J., and Lum K., “Selecting Best Practices for Effort Estimation”, IEEE Transactions on software engineering, vol. 32, no. 11, november 2006

List of papers

1. Paper titled "UML based effort estimation" ***published and presented*** in National Conference "NCANIT-09" held at Guru Jhambeshwar University, Hisar on 24-25 March'2009, pp 537-542.

2. Paper titled "Development issues in component based systems" ***published and presented*** in National Conference "ETSNT'09" held at Amity University, Noida on 17-18 April'2009, pp 169-174.

3. Paper titled "UML based effort estimation for CBS" ***communicated*** to ACM SIGSOFT Software Engineering Notes.

Abbreviations

CASE	Computer Aided Software Engineering
CBS	Component Based Systems
CBSD	Component Based System Development
CD	Class Diagram
COCOMO	Constructive Cost Model
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CP	Class Point
CRM	Customer Relationship Management
DCOM	Distributed Component Object Model
DIT	Depth in Inheritance Tree
EF	Environmental Factor
EJB	Enterprise Java Bean
FP	Function Point
FUSP	Fuzzy Use Case Size Points
MPC	Methods per Class
NAC	Number of Association per Class
NOA	Number of actors
NOASS	Number of Associations
NOC	Number of Classes
NOCA	Number of Class Attributes
NOIR	Number of Inheritance Relationships
NOM	Number of Methods
NOP	Number of Parameters

NOR	Number of roles.
NORR	Number of Realize Relationships
NOUC	Number of use cases
NOUR	Number of Use Relationships
PMS	Percentage of methods with signature
OO	Object Oriented
OF	Overhead Factor
SDLC	Software Development Life Cycle
SLOC	Source Line Of Code
TCF	Technical Complexity Factor
TDI	Total Degree of Influence
UAW	Unadjusted Actor Weight
UUCP	Unadjusted Use Case Point
UUCW	Unadjusted Use Case Weight
UC	Use Case
UCP	Use Case Point
UML	Unified Modeling Language
USP	Use case Size Points
UUSP	Unadjusted Use Case Size Points
FTA	Technical Factor Adjustment
FAA	Environmental Adjustment Factor

Appendix A1: Questionnaire

Questionnaire for industry professionals/ Academicians/ Research Scholars
Effort estimation in component based systems

Please rate the factor among the values.

Each value indicates the extent upto which the factor contributes while estimating effort/cost for a component based system. For example a 0.5 value indicates the factor produces least effect for predicting the efforts required; max value (2) means highly influential; whereas a negative value indicates that it helps in reducing the efforts/cost required while developing a component based software.

Technical complexity factors

Factor	Our estimate	Your Estimate
Distributed system	0.5	
Component performance	1.5	
End-user efficiency	1	
Internal complexity	1.5	
Component reusability	1	
Usability	1	
Maintainability	2	
Security	1.5	
Third party integration	1.5	
User education	0.5	
Case tools	-0.5	
Interface complexity	1.5	

Environmental factors

Factor	Our estimate	Your Estimate
Domain experience	0.5	
Stable requirements	2	
Pat-time workers	-1	
Technology newness	0.5	
Component model	1	
Internal library resources	-1	
Organization maturity	-0.5	

Name Designation

Organization Total Experience