A
Dissertation
On

Estimation of Object Oriented Function Points based on UML Design Specification

Submitted in Partial fulfillment of the requirements
For the award of Degree of

MASTER OF ENGINEERING
(Computer Technology and Application)
Delhi University, Delhi

Submitted By:
VARUN BARTHWAL
(University Roll No. 12216)

Under the Guidance of:
Dr. Daya Gupta
Head Of Department
Department Of Computer Engineering
Delhi College of Engineering, Delhi

DEPARTMENT OF COMPUTER ENGINEERING
DELHI COLLEGE OF ENGINEERING
DELHI UNIVERSITY
(2008-2009)

TABLE OF CONTENT

CERTIFICATE

DELHI COLLEGE OF ENGINEERING
(Govt. of National Capital Territory of Delhi)
BAWANA ROAD, DELHI - 110042

Date:___________

This is certified that the work contained in this dissertation entitled "Estimation of Object Oriented Function Points based on UML Design Specification**."** by Varun Barthwal is the requirement of the partial fulfillment for the award of degree of Master of Engineering in Computer Technology & Application at Delhi College of Engineering. This work was completed under my direct supervision and guidance. He has completed his work with utmost sincerity and diligence.

The work embodied in this major project has not been submitted for the award of any other degree to the best of my knowledge.

Dr.Daya Gupta
Head Of Department
Department of Computer Engineering
Delhi College of Engineering

ABSTRACT

Object - oriented programming is widely used approach to develop a software system, several techniques have been defined by researchers and practitioners to measure the size of object-oriented software system. Unified modelling language (UML) is most popular way to implement a object oriented software system, how to map this UML to function point analysis various approaches are suggested by software practitioners, we have used UML design specification to map the UML to function point analysis. Various rules were proposed earlier, these rules can be applied on UML design specification to estimate function points. We have used UML class diagram for data function analysis and UML sequence diagram for transaction function analysis than we have applied transformation rules and guidelines to estimates function point. We have developed a tool based estimation technique for object oriented software metrics based on UML design specification using COCOMO II.

ACKNOWLEDGEMENT

I am thankful to the Almighty because without his blessings this work was not possible. It is a great pleasure to have the opportunity to extent my heartfelt gratitude to everybody who helped me throughout the course of this project.

It is distinct pleasure to express my deep sense of gratitude and indebtedness to my learned supervisor Dr. Daya Gupta for his invaluable guidance, encouragement and patient reviews.
His continuous inspiration has made me complete this dissertation. He kept on boosting me time and again for putting an extra ounce of effort to realize his work.

I would also like to take this opportunity to present my sincere regards to my teachers Mrs. Rajni Jindal, Dr. S.K. Saxena, Mr. Manoj Sethi, Mr. Manoj Kumar and Mr. Vinod Kumar for their support and encouragement.

I am grateful to my parents, brother and sister for their moral support all the time, they have been always around on the phone to cheer me up in the odd times of this work.

I am also thankful to my classmates for their unconditional support and motivation during this work.

Varun Barthwal
M.E.(Computer Technology and Application)
Department of Computer Engineering
Delhi College of Engineering, Delhi-42

1. **INTRODUCTION**

1.1General Concept

Today is the era of developing of Object oriented software e.g. banking application, commercial application and in various fields. Developing a quality, cost- effective software within a specified time period is still a challenging task. In order to attain it, it is necessary to manage the entire software development processes based on the effective project plan. Therefore software development process should include correct estimation of various software metrics like size, effort invested, development time, quality, risks and resources of software. Many Researchers feel that size estimation should be done in the early phase of the development life cycle that is on the transformation model. There are various models for effort, cost, quality estimation used by software practitioners. There have been proposed a lot of effort models and most of them include software size as an important parameter. In the models, LOC (lines of codes) is often adopted. However, using LOC as the software size has difficulties because the definition of LOC is very vagueness and LOC depends on the programming language. Function point is a measure of software size that uses logical functional terms business owners and users more readily understand. Since it measures the software requirements or business models, the measured size stays constant despite the programming language, design technology, or development skills involved. Also, it is available early in the development process, making its use opportune for planning the design and development projects. Up to the present, various FPA versions based on the Albrecht's version have been proposed. IFPUG (International Function Point Users Group) version [2] frequently used in software organizations. In industrial practice, it is desirable to have a reliable cost estimate available already before software is actually built. One of the more popular approaches to estimating the software size is Function Point Analysis (FPA) [2,7]. Detailed FPA measurement rules were proposed for the design specifications using the UML (Unified Modeling Language) and develop the function point measurement tool.

Rational Rose is widely used in software development organizations which provide inputs to estimate function points. The Unified Modeling Language (UML) [11,12] was developed to provide a common language for object oriented modeling. It was designed to be extensible in order to satisfy a wide variety of needs and was also intended to be independent of particular programming language and development methods. In this thesis we have present our work that is to automate this process completely in early development life cycles then we will used COCOM II estimation techniques to calculates rest of the software metrics e.g. effort, cost, development time etc.

1.2 Motivation

It is very important to determine the size of a proposed software system yet to be built based on its requirements, i.e., early in the development life cycle. Given a size estimate, it is usually possible to estimate the effort, cost, development time and rest of the software metrics, to build this system. The most widely used approach to size estimation is Function Point Analysis (FPA). It is not clear, however, how function points can be reasonably counted for object-oriented requirements specifications. To estimates size of object oriented system in early development life cycle is quiet necessary and to automate this process is still changeling task. Our main goal is to automate this process completely in early development life cycles then we will used COCOMO II estimation techniques to calculates rest of the software metrics e.g. effort, cost, development time etc. This thesis presents an algorithm to calculates software size in function points and an architecture which calculates rest of the software metrics, e.g. effort, development time, cost etc. along with the support of tool that have been constructed to automate the metrics estimation.

Harput suggest a semi-automatic transformation model to estimate the size of object oriented system which is based on class diagram, use-cases and sequence diagram. Kusumoto has calculated function point for java source code. Uemura and Kusumoto also developed a tool, based on UML design specification. Our FPA algorithm follow all rules suggested by Harput and inspired by approach given by Uemura and Kusumoto.

1.3 Related Work

There are various techniques for measuring size of traditional software like LOC, function point etc. Commonly used techniques to measure size of software, Line of code (LOC), function count, object point and statement count. Early approaches were centred on function point measures such as Albrech method[1],IFPUG method[9], Mark II method[6],COSMIC full function point method [7], IBM German point Method [8]. Caldiera had a approach used for Objet Oriented Programming (OOP) is Function Point Count for OO system [11], application Point and Multimedia Point [13], D.J Ram an S.V.G.K Raju presented Object Oriented Design Function Points [3] and Object Point Count by Sneed [12], he proposed object points as a measure of size for OO software. Object points are derived from the class structures, the messages and the processes or use cases, weighted by complexity adjustment factors. Problem with this approaches were that they required judgment on the part of measurer, hence they were not accurate. Recent approaches for size estimation of OOP are web object [34], statement count [15], automated function count for OOP [4] and class point. Harput proposed rule based function point estimation from transformation Model [2]. D. Janaki Ram and S. V. G. K. Raju used all the available information during the Object oriented design phase to estimate Object Oriented Design Function Points (OODFP). They have suggested a counting procedure to measure the functionality of an object oriented system during the design phase from a designers' perspective. They have used all the available information during the oriented system design phase to estimate Object Oriented Design Function Points (OODFP). It considers all the basic concepts of oriented system systems such as inheritance, aggregation, association and polymorphism. Kusumoto measured function point from source code based on static and dynamic information collected by execution of set of test cases [4]. This approach is not suitable for project planning but can be used only for maintenance metrics when coding and implementation part have completed. Some research concentrate on new emerging paradigm of Object Oriented (OO) software and design a tool that provide unifying framework for calculation of different kind of software metrics like size, cost, time, effort, productivity, maintenance metrics, and quality metrics [15][16]. Edilson J. D. Candido, Rosely Sanches, Estimate the size of web applications by using a simplified function point Method[39]. Sneed and Huang [10], presents an effort estimation technique for maintaining a large-scale web application by measuring and tracking the size and complexity of web based system, they used a combination of function-points and static impact analysis to trace the change request to different components of web application and then measure their size and complexity.

H.Sneed [9] described an ongoing project to improve the maintenance process. A repository has been constructed on the basis of a relational database and populated with metadata on a wide variety of software artifacts at each semantic level of development – concept, code and test, this repository is used to perform impact analysis and cost estimation of change requests prior to implementing them. Sneed constructed a tool to navigate through the repository, select the impacted entities and pick up their size, complexity and quality metrics for effort estimation. Giovanni Cantone [19] introduces a conversion model (UML to FP) for establishing the link, and presents a pilot study for comparing the Function Point counts provided by the model with those provided by a Function Point certified expert. K.Koteswara Rao,Srinivasan, Nagaraj and Jitender Ahuja introduced the idea of using UML Relationships as the starting point and gave the brief introduction on how to get the building blocks of the function point analysis out of the diagrams, Relationships, they have focused on UML relationship, generalization, association, dependency and realization. In order to map the UML elements to Function Point Analysis entities, they develop guidelines, rules, heuristics, and flexibility specifications, which also constitute the requirements of an analyzer and semi-automatic converter.

1.4 Organization

The remainder of thesis is organized as follows: Chapter 2 provides an overview of function points metrics and related approach given by D.Ram, Kusumoto approach to measure it. COCOMO II is introduce in Chapter 3 in which early design model is discussed and how function points is converted into source lines of codes also describe there. Chapter 4 describe the tool architecture which we have developed and its design scheme and also in this section we have propose algorithm to estimate function points for object oriented system. Implementation details are given in 5th section of thesis in which required platform and additional tool have been introduced. Finally Conclusion and future work are mention in chapter 7.

2. SOFTWARE METRICS

2.1 Basic Software Metrics

Size of software system is considered as the basic metrics in software metrics model. Size can be estimated in Lines of code or function point. Lines of code cannot be estimated correctly before software completion because it varies due to language complexities of different language. While Function points are technologically independent, consistent, repeatable, help normalize data, enable comparisons and set project scope and client expectation. So here size is estimated in terms of function point.

2.1.1 Function Points

Function points measure the information processing content of software systems. Function points measure the size of an application from the customer's point of view. The aspects of a software system that can be measured accurately are these:

- Inputs to the application.
- Outputs from the application.
- Inquiries by the end users.
- Data files updated by the application.
- The interface to other applications.

2.1.2 FPA Process Overview

The FPA process involves:

1. Identifying the function point counting boundary. A boundary indicates the border between the software system being measured and the external application or the user domain. A boundary determines what functions are included in the function point count.

2. Determining the unadjusted function point count (UFPC). The unadjusted function point count reflects the specific countable functionality provided to the user by the application.

2.1.3 Internal logical files (ILF)

An internal logical file (ILF) is a user identifiable group of related data maintained within the boundary of the application.

An ILF must be a group of data that is maintained within the application and satisfies specific user requirement. Data stores that were created for technical reasons or for storage of intermediate values are not counted. Extra capabilities automatically provided are not counted unless the customer specifically requests them.

ILF Complexity

RECORD ELEMENT TYPE	DATA ELEMENT TYPE		
	1-19	**20-50**	**>51**
1	**LOW(7)**	**LOW(7)**	**AVG(10)**
2 TO 5	**LOW(7)**	**AVG(10)**	**HIGH(15)**
6 OR MORE	**AVG(10)**	**HIGH(15)**	**HIGH(15)**

TABLE 2.1

2.1.4 External Interface Files (EIF)

An External Interface File (EIF) is a user identifiable group of logically related data maintained outside the boundary of the application. One example of an EIF is a file or table containing names of codes read by the system being counted but maintained by some other application. The group of data is logical and user identifiable, and satisfies a specific user requirement, referenced by the application, not maintained by the application, is also an ILF in another application

EIF Complexity

RECORD ELEMENT TYPE	DATA ELEMENT TYPE		
	1-19	**20-50**	**>51**
1	**LOW(5)**	**LOW(5)**	**AVG(7)**
2 TO 5	**LOW(5)**	**AVG(7)**	**HIGH(10)**
6 OR MORE	**AVG(7)**	**HIGH(10)**	**HIGH(10)**

TABLE 2.2

2.1.5 External input (EI)

An external input (EI) processes data that come from outside the application boundary. An external input is the facility provided to the customer to insert, update, and delete records of an ILF. It may maintain one or more ILFs. For example, an external input may maintain department and employee information. The information entered will be stored in one or more ILFs. Another example may be the maintenance of system parameters, which will be used by the processes of the software system being developed. Data are received from outside the application boundary, input is the smallest business transaction as seen by the user, comprehensive and self contained.

External input complexity

RECORD ELEMENT TYPE	DATA ELEMENT TYPE		
	1-19	**20-50**	**>51**
1	**LOW(5)**	**LOW(5)**	**AVG(7)**
2 TO 5	**LOW(5)**	**AVG(7)**	**HIGH(10)**
6 OR MORE	**AVG(7)**	**HIGH(10)**	**HIGH(10)**

TABLE 2.3

2.1.6 External output (EO)

An external output (EO) is a process that generates data sent outside the application boundary, for example, the external output the customer views in the form of reports, messages, etc. External outputs also include the files the application generates to be used as transactions by another application. An external output may be generated using one or more ILFs or EIFs. Data are sent outside the application boundary. The output is meaningful to the customer's business, comprehensive and self contained. Data in the ILF or EIF is not changed by the external output. Count only unique external output.

External output complexity

RECORD ELEMENT TYPE	DATA ELEMENT TYPE		
	1-19	**20-50**	**>51**
1	**LOW(5)**	**LOW(5)**	**AVG(7)**
2 TO 5	**LOW(5)**	**AVG(7)**	**HIGH(10)**
6 OR MORE	**AVG(7)**	**HIGH(10)**	**HIGH(10)**

TABLE 2.4

2.1.7 External query (EQ)

An external query is a process made up of an input-output combination that results in data retrieval. It has two parts, the screen on which the customer specifies the request (search criteria) and the resulting display. Count each unique request and display combination. The external query is unique if it has a format different from other external queries in either the request or display parts, or if the customer requests processing logic different from other external queries with the same format. On an external query, the customer enters data for control purposes to direct the search. An external query differs from an external input since it does not modify an ILF. Though it reflects the immediate retrieval of current data for display, it differs from external output in that external output reflects the manipulation and reformatting of data (usually in report form). The media

(screen or paper) is not the basis for distinguishing external queries from external output since external output can also be displayed on a terminal. An external output may be generated using one or more ILFs or EIFs.
The output is comprehensive, self contained and immediately required for the customer's business. When there is a one-to-one relationship between requests and displays, count only displays. Also, count just the displays if one request results in multiple displays. In either case, the count of the displays will equal the external query count. If several unique request panels result in the same display, count the requests instead of the display, for example, a display of customer information that results from completing a screen of name information, a screen of address information, or information about a specific purchase. In these cases there is one display but three external queries, since there are three different processes that get the same display.

External query complexity

RECORD ELEMENT TYPE	DATA ELEMENT TYPE		
	1-19	**20-50**	**>51**
1	**LOW(5)**	**LOW(5)**	**AVG(7)**
2 TO 5	**LOW(5)**	**AVG(7)**	**HIGH(10)**
6 OR MORE	**AVG(7)**	**HIGH(10)**	**HIGH(10)**

TABLE 2.5

2.2. IFPUG version

IFPUG version is a modified-version of the Albrecht’s function point. In the modification, the evaluation of the complexity of the software was objectively established and the rules of the counting procedures were also described minutely and precisely. In the IFPUG version, the counting procedure of function point consists of the following seven steps[2].

Step1 (Determine the Type of Function Point Count): Select the type of function point from the following three ones:(1) Development project function point count, (2)Enhancement project function point count and (3)Application function point count.

Step2 (Identify the Counting Boundary): A boundary indicates the border between the application or project being measured and the external applications or the user domain. A boundary establishes which functions are included in the function point count.

Step3 (Count Data Function Types): Data function types represent the functionality provided to the user to meet internal and external data requirements. Data function types are classified into the following two types: Internal logical file(ILF) and External interface file(EIF).

Step4 (Count Transactional Function Types): Transactional function types represent the functionality provided to the user for the processing of data by an application. They are defined as the following three types: External input(EI), External output(EO) and External inquiry(EQ). The definition of transactional functions are described as follows: External input(EI): An external input processes data or control information that comes from outside the application's boundary. The external input itself is an elementary process. External output(EO): An external output is an elementary process that generates data or control information sent outside the application's boundary. External inquiry(EQ): An external inquiry is an elementary process made up of an input-output combination that results in data retrieval. The output side contains no derived data. Here, derived data is data that requires processing other than direct retrieval and editing of information from internal logical files and or external interface files. No internal logical file is maintained during processing.

Then, assign each identified EI or EO a functional complexity based on the number of file types referenced (FTRs) and data element types (DETs).A file type referenced is ,(1) An internal logical file read or maintained by a function type, or (2) An external interface file read by a function type. Also, assign each EQ a functional complexity based on the number of file types referenced (FTRs) and data element types (DETs) for each input and output component. Use the higher of the two functional complexities for either the input or output side of the inquiry to translate the external inquiry to unadjusted function points. For each of EI, EO and EQ, there is a FTR/DET complexity matrix.

Step5 (Determine the Unadjusted Function Point Count): As the result of Step3 and Step4, the counts for each function type are classified according to complexity.

Step6 (Determine the Value Adjustment Factor): The value adjustment factor (VAF) indicates the general functionality provided to the user of the application. VAF is comprised of 14 general system characteristics that assess the general functionality of the application.

Step7 (Calculate the Final Adjusted Function Point Count):

The final adjusted function point count is calculated using a specific formula for development project, enhancement project or application based on the result of Step1.

2.3 Object Oriented Design Function Point

D.J Ram has given this approach to estimate size based on all information available in design phase.

Data Function Types

According to Ram classes are mapped into data functions. A logical file is divided into two types depending on the application boundary. The complexity of an ILF/EIF depends on the DETs and RETs it has. A DET is a simple data type such as int, char, float, string etc. Object reference, a complex data type is considered as a RET. So, in case of aggregation RET should be considered. The inherited data is visible to all the methods in a derived class. So, inherited data should he included to calculate the complexity of a derived class.

Transaction Function Types

Methods in a class are candidates for transactional function types. It operates on the data within that class, arguments and return values. The complexity of a method depends on the DETs and FTRs. The inherited methods will he coded only once in the base class. So, methods that are inherited from a base class should not he considered for estimating the complexity of a derived class. If any derived class overrides a method, its complexity should be considered for that derived class alone. Using the signature of a method, it is possible to identify the communicating objects. So, association should he considered for the method from where it invokes the required method(s). A single valued association is considered as a DET and a multivalued association is considered as a FTR. Method without any arguments and return type, then its complexity is considered as one DET.

Complexity of Class

The complexity of a class is classified low if a class processes less than 50% of data that is visible to it , average if a class processes 51 % to 70% of data that is visible to it and high if a class processes more than 70% of data that is visible to it.

	Complexity Value
LOW	0.3
AVERAGE	0.6
HIGH	0.9

TABLE 2.6 Complexity Value Of Class

The complexities are mapped to a numerical value based on observations across different projects. These values are presented in Table 2.6.

Unadjusted Function Point

Unadjusted Function point (UFP) of the Object Oriented system is calculated as follows:

1. Calculate the function points for each class in the design. It is obtained by adding the function points of its data function and transactional function.
2. Estimate Complexity Value of Class.
3. UFP of a class is obtained by multiplying its function points with Complexity Value of Class.
4. Add UFP of each class to the get the UFP of the Object Oriented system.

2.4 Kusumoto’s Dynamic approach

Kusmoto suggests dynamic approach to calculate size of java source code. They have developed a function point measurement tool to measure function points from java source code, they proposed measurement rules to count data function and transactional function types based on IFPUG method and used dynamic information collected from the program execution based on a set of test cases which should correspond to all functions of the target program. In order to measure function point, it is necessary to extract the logical file and transaction function from the target program. Complexity of logical file based on the number of data element type (DET) and the record element type (RET) and for method complexity is determined by number DET and file type references (FTR). Tool generates syntax information log and dynamic information log files reading java source code.

Kusumoto’s tool includes following components

Syntax analyzer

It analyses the target program and collect syntax information used in the function point calculation of it into syntax information file.

Executor

It executes the target program using a set of test cases and collects information about program execution and store it into execution log file.

Function point calculator

It calculates the value of function point based on the data of syntax database, execution log database using the specified data function classes and boundary classes.

2.5 Function Point to Unified Modelling Language: Conversion Model by Giovanni Cantone

Cantone considers convertibility of the elements of the Unified Modeling Language into entities of the Function Point Analysis; they introduced a model for establishing the link. In order to map the Unified Modeling Language elements to Function Point Analysis entities, some guidelines, rules, heuristics, and flexibility specifications, developed by Cantone. Cantone aimed to develop map, a usage strategy, and a tool to support analysis of UML-documented applications, UML-FP conversion, and FP counts. Consequently, they have chosen to describe UML-FP mapping by placing conversion items in the form of rules and tool specific items in form of flexibility requirements. Cantone introduced a tool For a given set of parameter values, tool will be configured as a certain Automatic Analyzer and Counter and hence will enact the related mapping model.
Cantone introduced some rules, guidelines some flexibility requirements to estimate function points based on UML diagrams e.g. Use cases, Class diagram and Sequence diagram.

Data Function types

To estimate data function types, they used class diagram. The UML CD elements that are useful for counting Function Points are: Class, Class Stereotype, Attribute, Relationship, and Responsibility, Operation or Method (simply Method, in the followings). Class Stereotype includes three basic UML stereotypes, (i) Entity: these classes represent the key concepts of the application system; their main responsibilities are to store and manage information in the application system. (ii) Control: these classes model the control behavior of one or, in some cases, more Use Cases of the application system.
(iii) Boundary: these classes model the interaction between the external world and the internal logic of the application system. Entity classes are the candidates for logical files, they agreed with Caldiera[11] approach that all Logical Files are ILF, those files excepted that are mapped to classes encapsulating external components, which are identified as EIF, (e. g. other applications, external services, library functions).

Transactional Function types

Cantone focused on use-case diagram and sequence diagram to estimate transactional function. Communication patterns are used to detect transactional function types, in sequence diagram. External inputs can be identified by system directed messages sequences and external outputs and external query depends upon the message sequences which are actor directed. Complexity of transactional function is determined by total number of arguments of candidate messages and entity classes.

2.5 Harput's Transformation Model

Harput has proposed a semiautomatic transformation model to estimates Object Oriented Function Point (OOFP) early in the software development cycle. He proposed eighteen rules for Data function types and nine rules for transaction function types that specify a semi-automated transformation from an object-oriented requirement model to an FPA model. Harput presented the rules for mapping classes and associations to data function types as well as the rules for mapping use cases and functional requirements to transactional function types.

Rules for data function types

Rule 1 Classes or groups of classes in the information model are mapped to internal logical files (ILFs). If there is no information model available, then classes or groups of classes in the domain model are mapped to ILFs. In this case, however, only those classes are to be mapped to ILFs which represent entities the system to be built is required to maintain information about.

Rule 2 Some of the classes or groups of classes in the domain model are mapped to external interface files (EIFs). Classes which have already been mapped to ILFs according to Rule 1 for data function types may not be mapped to EIFs.

Rule 3 A single class can be mapped to one file.

Rule 4 All classes in a subtree of a generalization hierarchy can be mapped together to one file.

Rule 5 Leaf classes can be mapped together with all their ancestors to one file.

Rule 6 Classes which are connected through an aggregation can be mapped together to one file.

Rule 7 Attributes of classes represent the data element types (DETs) of the files.

Rule 8 Regardless of the number of the attributes in the mapped classes, every file has at least one DET.

Rule 9 Every file has at least one record element type (RET).

Rule 10 Some of the classes mapped to a function point file represent the RETs of this file. Which of the mapped classes represent RETs depends on the mapping method as given in the following rules, and they are to be determined by the FPA expert.

Rule 11 If a single class is mapped to a file, then one RET is counted for this file.

Since there is only one class being mapped in this case, it is the only one which can be counted as a RET.

Rule 12 If the classes in a generalization hierarchy are mapped as a group to one file, then a RET can be counted for each leaf class or, alternatively, a RET can be counted for each class in the hierarchy.

Rule 13 If a leaf class together with all its ancestors is mapped to a file, then a RET can be counted for the leaf class only or, alternatively, a RET can be counted for each class from leaf to root.

Rule 14 If classes which are connected through an aggregation are mapped together to one file, then a RET is counted for the aggregating class and for each aggregated class.

Rule 15 n-ary associations (with $n > 2$) can be decomposed into binary associations. If an n-ary association found in the domain or information model is to be mapped to function point files, it needs to be decomposed to binary associations first

Rule 16 Binary associations between classes can be mapped to files.

Rule 17 Associations and aggregations can increase the DET counts of those files by one which have been created by mapping the connected classes.

Such relations which are not mapped to files can increase the DET count of files.

Rule 18 Files that were created by mapping associations to them, have at least two DETs.

Rules for transactional function types

Rule 1 Use cases with given pre- and post conditions can be viewed as functional requirements for the composite system.

Rule 2 Messages in UML sequence diagrams can be viewed as functional requirements for the system to be built.

Rule 3 Functional requirements for the composite system consisting of the system to be built and the users can be mapped to transactions. Functional requirements for the system to be built can also be mapped to transactions.

Rule 4 Several functional requirements for the system to be built can be mapped together as a group to a transaction.

Rule 5 If a functional requirement for the composite system is mapped to a transaction, the related functional requirements for the system to be built must not be mapped to transactions, and vice versa.

Rule 6 The FPA expert has to determine the type of the transactions.

Rule 7 The file types referenced (FTRs) of the transactions are determined through the classes in the domain or information model that have been mapped to files. These classes can be explicitly referenced from the functional requirements or from messages in sequence diagrams, respectively. The files which these classes have been mapped to are the FTRs of those transactions which the corresponding functional requirements or messages have been mapped to.

Rule 8 For each transaction identified and for each message in a UML sequence diagram corresponding to this transaction that contains an object as a parameter, DETs can be counted as follows: for each attribute of such an object in a UML class diagram, one DET can be counted for each field according to its data type, if this attribute crosses the system boundary (but it may be counted only once).

Rule 9 For each transaction identified, if at least one corresponding message in a UML sequence diagram exists for a system response message, a confirmation or verification, then count one additional DET for this transaction.

2.5.1 Estimation of data function types

Identifying Data Functions

An **information model** in software engineering is a representation of concepts, relationships, constraints, rules, and operations to specify data semantics for a chosen domain of discourse. It can provide sharable, stable, and organized structure of information requirements for the domain context.

In problem solving a **domain model** can be thought of as a conceptual model of a system which describes the various entities involved in that system and their relationships. The domain model is created in order to document the key concepts, and the domain-vocabulary of the system being modeled. The model identifies the relationships among all major entities within the system, and usually identifies their important methods and attributes. In UML, a class diagram is used to represent the domain model.

Harput suggests that domain model is used for data function types identification, information model can also be taken but mostly it is not available, only those classes of domain model are to be mapped to internal logical files which represent entities the system to be built is required to maintain information about (see Harput rule 1,2).

If data members of classes can be modified or renewed than these classes should be considered as a candidate for internal logical files else these are candidates for external interface files [21].

All Logical Files are ILF, only those files are accepted as EIF that are mapped to classes encapsulating external components (e. g. other applications, external services, library functions) [10]. Only those objects are kept as Logical File candidates that both include some attributes, and exchange data with non-Actor objects: “Objects that have attributes changed by the operations of other objects are regarded as ILF and others are regarded as EIF”, according to Uemura[21] .

Entity: These classes represent the key concepts of the application system; their main responsibilities are to store and manage information in the application system. All and only classes stereotyped Entity are logical file candidates, ILF or EIF [18].

Simple attribute: This represents a basic data type; one DET is counted for each of such attributes ("e.g. integers, strings etc." are counted as 1 DET each). Attributes do map DET one to one (Harput rule for data function types 7).

Complex attribute: One RET is counted for each of such attributes.

In UML, a class diagram is used to estimation of data function types.

Single Class

A single class can provide information about data function types if system to be built has to maintain information only about that particular class.

DET Count: Attributes of class represent the Data Element Types (DETs) of the files (Harput rule for data function types 7).
RET Count: At least One Record Element Types RET is counted for a single class (Harput rule for data function types 9,11).

Association
Binary association is used for mapping classes into logical files. If association is not binary then first it converted into binary then mapped to files. Self- Associations are never mentioned (Harput rule for data function types 15,16).

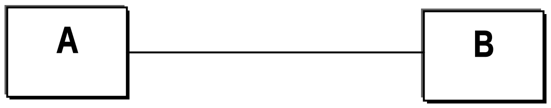


Fig. 4.2 Association in Class diagram

DET Count: Data element types are increased by one due to associations or in other words at least two DETs can be taken for those logical files which are mapped by associations. Association having multiplicity not greater than one, increase FP complexity by one DET each (Harput rule for data function types 17,18).

RET Count: Association having maximum multiplicity is not just one, increase FP complexity by one RET each.

Aggregation

An aggregation may signify that an instance of one class can contain an instance of another class. Entire aggregation structure is counted as a single logical file these classes can be taken as candidates for data function types (Harput rule for data function types 6).

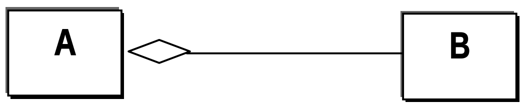


Fig. 4.3 Aggregation in Class diagram

DET Count: Aggregation having multiplicity not greater than one, increase FP complexity by one DET each (Harput rule for data function types 17).

RET Count: If classes which are connected through an aggregation are mapped together to one file, then a RET is counted for the aggregating class and for each aggregated class. In an Aggregation, RET increment affects the Logical File complexity of the aggregating class. Aggregation having maximum multiplicity is not just one, increase FP complexity by one RET each (Harput rule for data function types 14).

Generalization

These classes represent the information in the application system are analyzed for estimation of data function types. Leaf classes are mapped together with all their ancestors to one file or in class hierarchies, total path from root to leaf class is provides one logical file, e.g. in figure given below {A,B,D}, {A,B,E} and {A,C} can be taken as data function types (Harput rule for data function types 5).

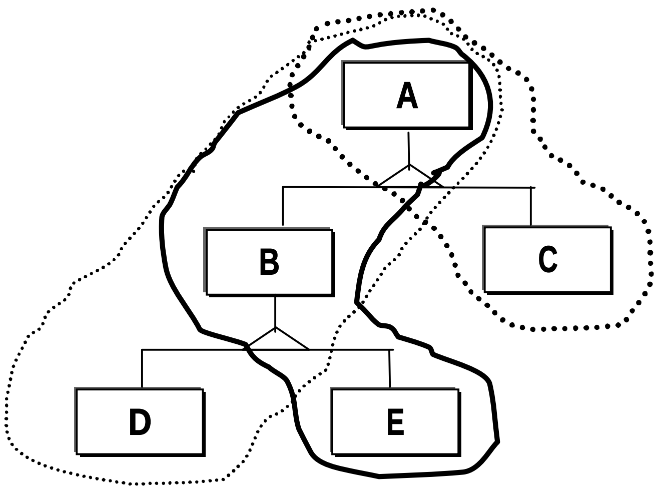


Fig. 4.4 Generlization in Class diagram

DET Count: Total number of attributes in one logical file (Harput rule for data function types 7).

RET Count: RET can be counted for each class from leaf class to root in generalization hierarchy (Harput rule for data function types 13).

For each logical file in a generalization hierarchy each class represent one RET e.g. in given figure {A,B,D}, {A,B,E} and {A,C} are taken as logical files having RET 3,3 and 2 respectively.

2.5.2 Estimation of Transactional Function Types:

In sequence diagram, two kinds of messages sequences should be considered [19]

1. **Actor directed messages sequences (ADMS)**
2. **System directed messages sequences (SDMS)**

Messages between actor to actor and entity objects to entity objects should not be considered as candidates for transactional function types.

Messages: Messages help to find the Elementary Processes of the application system. Here the problem is to map messages or message sequences to Elementary Processes.

We have taken messages as candidates for transactional function types (Harput rule for transactional function types 2).

External Input (EI): External inputs are those messages which occur in System Directed Messages Sequences (SDMS), i.e. from actor to application.

DET count: A data element type (DET) counts of a transactional function is the number of arguments in messages directed to entity objects types (Harput rule for transactional function types 8, 9).

FTR count: The file type reference (FTR) count of a Transactional Function is the number of entity objects that participate in the message exchange types (Harput rule for transactional function types 7).

External Output (EO): Messages which occur in Actor Directed Messages Sequences (ADMS), i.e. from system to actor. When arguments of all the messages in an ADMS include some but not all attributes of the objects read through messages sequence. It means that the message contains derived data. Then, we regard it as an External Output.

DET count: A data element type (DET) counts of a transactional function is the number of arguments which are attributes of entity objects in messages or message sequences (Harput rule for transactional function types 8, 9).

FTR count: The file type reference (FTR) count of a Transactional Function is the number of entity objects that participate in the message exchange (Harput rule for transactional function types 7).

External Query (EQ): Those messages which occur in Actor Directed Messages Sequences (ADMS), i.e. from system to actor. When arguments of all the messages in an

ADMS include all the attributes of the objects read through messages sequence. Then, we regard it as an External Query.

DET count: A data element type (DET) counts of a transactional function is the number of arguments which are attributes of entity objects in messages or message sequences (Harput rule for transactional function types 8, 9).

FTR count: The file type reference (FTR) count of a Transactional Function is the number of entity objects that participate in the message exchange (Harput rule for transactional function types 7).

3. SOFTWARE METRICS ESTIMATION: COCOMOII

3.1 General Software Metrics

Effort, Development time, cost and productivity are considered as a general software metrics. COCOMO II model is adopted for estimating these metrics. COCOMO II requires software size in terms of LOC. In first layer we estimate size in unadjusted function point.

3.2 Relating UFPs to SLOC

COCOMO II is used for calculation of other software matrices. The unadjusted function points have to be converted to source lines of code in the implementation language (Ada, C, C++, Pascal, etc.). Table given below shows the number of lines of codes per function point.

Programming Language	SLOC/UFP
ADA 95	49
C	128
C++	55
COBOL (ANSI 85)	91
FORTRAN 95	71
HTML 3.0	15
JAVA	53
LISP	64
PROLOG	64
VISUAL C++	34

TABLE 3.1 SLOC/UFP

For many years, software engineers and computer scientists have used phrases as “high level language ” and “low level language” without precisely defining a terms. Now with reasonable good justification, language can classify according to the number of statements they require to encode one function point:

High level language , less than 50

Mid level language, 51-99 and

Low level language, more than 100

3.2 COCOMO II

COCOMO II is tuned to modern software life cycles. The original COCOMO model has been very successful, but it doesn't apply to newer software development practices as well as it does to traditional practices. COCOMO II targets the software projects of the 1990s and 2000s, and will continue to evolve over the next few years.

The primary objectives of the COCOMO II effort are:

- To develop a software cost and schedule estimation model tuned to the life cycle practices of the 1990's and 2000's.
- To develop software cost database and tool support capabilities for continuous model improvement.
- To provide a quantitative analytic framework, and set of tools and techniques for evaluating the effects of software technology improvements on software life cycle costs and schedules.

COCOMO II is really three different models:

- The Application Composition Model
- The Early Design Model
- The Post-Architecture Model

3.2.1 Effort Estimation

In COCOMO II effort is expressed as Person-Months (PM). A person month is the amount of time one person spends working on the software development project for one month. This number excludes time typically devoted to holidays, vacations, and weekend time off. The number of person-months is different from the time it will take the project to complete; this is called the development schedule or Time to Develop, TDEV. For example, a project may be estimated to require 50 PM of effort but have a schedule of 11 months.

$$PM = A \times (Size)^{E} \times \prod_{i=1}^{n} EM_i$$

where $A = 2.94$

Scale Factors

The exponent E in equation is an aggregation of five scale drivers that account for the relative economies or diseconomies of scale encountered for software projects of different sizes. If $E < 1.0$ the project exhibits economies of scale. If the product's size is doubled, the project effort is less than doubled. For small projects, fixed start-up costs such as tool tailoring and setup of standards and administrative reports are often a source of economies of scale. If $E = 1.0$ the economies and diseconomies of scale are in balance. This linear model is often used for cost estimation of small projects. If $E > 1.0$ the project exhibits diseconomies of scale.

Scale drivers	Very low	Low	Nominal	High	Very High	Extra High
PREC	6.20	4.96	3.72	2.48	1.24	0.00
FLEX	5.07	4.05	3.04	2.03	1.01	0.00
RESL	7.07	5.65	4.24	2.83	1.41	0.00
TEAM	5.48	4.38	3.29	2.19	1.10	0.00
PMAT	7.80	6.24	4.68	3.12	1.56	0.00
	or the estimated Process Maturity Level (EMPL)					

- **TABLE 3.2 Scale Factors (E) COCOMO II estimation model**

Early Design Model Cost Drivers

COCOMO II uses a set of effort multipliers to adjust the nominal person-month estimate obtained from the project's size and exponent drivers

This model is used in the early stages of a software project when very little may be known about the size of the product to be developed, the nature of the target platform, the nature of the personnel to be involved in the project, or the detailed specifics of the process to be used. This model could be employed in either Application Generator, System Integration, or Infrastructure development sectors. The Early Design model uses KSLOC or unadjusted function points (UFP) for size. UFPs are converted to the equivalent SLOC and then to KSLOC. The application of project scale drivers is the same for Early Design and the Post-Architecture models. In the Early Design model a reduced set of cost drivers is used as shown in Table given below. The Early Design cost drivers are obtained by combining the Post-Architecture model cost drivers.

Early Design	Post-Architecture Cost Drivers
RCPX	RELY, DATA, CPLX, DOCU
RUSE	RUSE
PDIF	TIME, STOR, PVOL
PERS	ACAP, PCAP, PCON
PREX	APEX, PLEX, LTEX
FCIL	TOOL, SITE
SCED	SCED

TABLE 3.3

Cost Driver	Extra Low	Very Low	Low	Nominal	High	Very High	Extra High
RCPX	0.73	0.81	0.98	1.0	1.30	1.74	2.38
RUSE	---	---	0.95	1.0	1.29	1.81	2.61
PDIF	---	---	0.87	1.0	0.83	0.63	0.50
PERS	2.12	1.62	1.26	1.0	0.83	0.63	0.50
PREX	1.59	1.33	1.12	1.0	0.87	0.71	0.62
FCIL	1.43	1.30	1.10	1.0	0.87	0.73	0.62
SCED	---	1.43	1.14	1.0	1.0	1.0	---

TABLE 3.4 Early Design Cost Driver

3.2.2 Schedule Estimation

Nominal-Schedule Estimation Equations

Both the Post-Architecture and Early Design models use the same functional form to estimate the amount of effort and calendar time it will take to develop a software project. These nominal-schedule (NS) formulas exclude the cost driver for Required Development Schedule, SCED. The amount of effort in person-months, PM_{NS}, is estimated by the formula:

$$PM_{NS} = A \times Size^{E} \times \prod_{i=1}^{n} EM_i$$

$$\text{where } E = B + 0.01 \times \sum_{j=1}^{5} SF_j$$

$TDEV_{NS}$, it will take to develop the product is estimated by the formula:

$$\mathrm{TDEV_{NS}} = \mathrm{C} \times \left(\mathrm{PM_{NS}}\right)^{\mathrm{F}}$$
$$\text{where } \mathrm{F} = \mathrm{D} + 0.2 \times 0.01 \times \sum_{j=1}^{5} \mathrm{SF_j}$$

The value of n is 16 for the Post-Architecture model effort multipliers, EM_i, and 6 for the Early Design model. The values of A, B, C, D, SF_1, …, and SF_5 for the Early Design model are the same as those for the Post-Architecture model. The values of EM_1, …, and EM_6 for the Early Design model are obtained by combining the values of their 16 Post-Architecture counterparts.

The subscript NS applied to PM and TDEV indicates that these are the nominal-schedule estimates of effort and calendar time. The effects of schedule compression or stretch-out are covered by an additional cost driver, Required Development Schedule. Size is expressed as thousands of source lines of code (SLOC) or as unadjusted function points (UFP). Development labor cost is obtained by multiplying effort in PM by the average labor cost per PM.

The values of A, B, C, and D are:

A = 2.94	B = 0.91
C = 3.67	D = 0.28

The initial baseline schedule equation for the COCOMO II Early Design and Post-Architecture stages is:

$$\mathrm{TDEV} = [\mathrm{C} \times (\mathrm{PM_{NS}})^{(\mathrm{D}+0.2\times(\mathrm{E}-\mathrm{B}))}] \times \frac{\mathrm{SCED\%}}{100}$$
$$\text{where } \mathrm{C} = 3.67, \mathrm{D} = 0.28, \mathrm{B} = 0.91$$

In Equation, C is a TDEV coefficient that can be calibrated, PM_{NS} is the estimated PM *excluding* the SCED effort multiplier, D is a TDEV scaling base-exponent that can also be calibrated. E is the effort scaling exponent derived as the sum of project scale drivers and B as the calibrated scale driver base-exponent. SCED% is the compression / expansion percentage in the SCED effort multiplier rating scale.

3.3 Advanced Software Metrics

Maintenance and quality metrics are advanced software Metrics. Impact analysis [13][14] is used to calculate maintenance metrics , while quality metrics is estimated by software tester on the basis of different quality parameter. Impact analysis estimates the maintenance size of project.

Software Maintenance

Software maintenance is defined as the process of modifying existing software while not changing its primary functions. COCOMO II model assume that software maintenance cost generally has the same cost driver attributes as software development costs. Maintenance includes redesign and recoding of small portions of the original product, redesign and development of interfaces, and minor modification of the product structure. Maintenance can be classified as either updates or repairs. Product repairs can be further segregated into corrective (failures in processing, performance, or implementation), adaptive (changes in the processing or data environment), or perfective maintenance (enhancing performance or maintainability). The SCED cost driver (Required Development Schedule) is not used in the estimation of effort for maintenance because maintenance cycle is usually of a fixed duration. The RUSE cost driver (Required Reusability) is not used in the estimation of effort for maintenance due to the extra effort required to maintain a component's reusability is roughly balanced by the reduced maintenance effort due to the component's careful design, documentation, and testing. The RELY cost driver (Required Software Reliability) has a different set of effort multipliers for maintenance. For maintenance the RELY Cost driver depends on the required reliability under which the product was developed. If the product was developed with low reliability it will require more effort to fix latent faults. If the product was developed with very high reliability, the effort required to maintain that level of reliability will be above nominal. The scaling exponent, E, is applied to the number of changed KSLOC (added and modified, not deleted) rather than the total legacy system

KSLOC. The effective maintenance size $(\text{Size})_m$ is adjusted by a Maintenance Adjustment Factor (MAF) to account for legacy system effects.

RELY Descriptors:	slight inconvenience	low, easily recoverable losses	moderate, easily recoverable losses	high financial loss	risk to human life	
Rating Levels	Very Low	Low	Nominal	High	Very High	Extra High
Effort Multipliers	1.23	1.10	1.00	0.99	1.07	n/a

Table 5.9 . RELY Maintenance Cost Driver

The maintenance effort estimation formula is the same as the COCOMO II Post-Architecture development model (with the exclusion of SCED and RUSE):

$$PM_M = A \cdot (\text{Size}_M)^E \cdot \prod_{i=1}^{15} EM_i$$

Sizing Software Maintenance

COCOMO II differs from COCOMO 81 in applying the COCOMO II scale drivers to the size of the modified code rather than applying the COCOMO 81 modes to the size of the product being modified. Applying the scale drivers to a 10 million SLOC product produced overlarge estimates as most of the product was not being touched by the changes. The scope of “software maintenance” follows the COCOMO 81 guidelines in [Boehm 1981; pp.534-536]. It includes adding new capabilities and fixing or adapting existing capabilities. It excludes major product rebuilds changing over 50% of the existing software, and development of sizable (over 20%) interfacing systems requiring little rework of the existing system. The maintenance size is normally obtained via Equation given below, when the base code size is known and the percentage of change to the base code is known.

$$(\text{Size})_M = \left[(\text{Base Code Size}) \times \text{MCF}\right] \times \text{MAF}$$

The Maintenance Adjustment Factor (MAF) is discussed below. But first, the percentage of change to the base code is called the Maintenance Change Factor (MCF). The MCF is similar to the Annual Change Traffic in COCOMO 81, except that maintenance periods other than a year can be used. Conceptually the MCF represents the ratio in Equation below:

$$\text{MCF} = \frac{\text{Size Added} + \text{Size Modified}}{\text{Base Code Size}}$$

A simpler version can be used when the fraction of code added or modified to the existing base code during the maintenance period is known. Deleted code is not counted.

$$(\text{Size})_M = (\text{Size Added} + \text{Size Modified}) \times \text{MAF}$$

The size can refer to thousands of source lines of code (KSLOC), Function Points, or Object Points. When using Function Points or Object Points, it is better to estimate MCF in terms of the fraction of the overall application being changed, rather than the fraction of inputs, outputs, screens, reports, etc. touched by the changes. The Maintenance Adjustment Factor (MAF). COCOMO II uses the Software Understanding (SU) and Programmer Unfamiliarity (UNFM) factors from its reuse model to model the effects of well or poorly structured/understandable software on maintenance effort.

$$\text{MAF} = 1 + \left(\frac{\text{SU}}{100} \times \text{UNFM} \right)$$

The Software Understanding increment (SU) is obtained from Table 5.10. SU is expressed quantitatively as a percentage. If the software is rated very high on structure, applications clarity, and self-descriptiveness, the software understanding and interface-checking penalty is 10%. If the software is rated very low on these factors, the penalty is 50%. SU is determined by taking the subjective average of the three categories.

	Very Low	Low	Nominal	High	Very High
Structure	Very low cohesion, high coupling, spaghetti code.	Moderately low cohesion, high coupling.	Reasonably well-structured; some weak areas.	High cohesion, low coupling.	Strong modularity, information hiding in data / control structures.
Application Clarity	No match between program and application world-views.	Some correlation between program and application.	Moderate correlation between program and application.	Good correlation between program and application.	Clear match between program and application world-views.
Self-Descriptive-ness	Obscure code; documentation missing, obscure or obsolete	Some code commentary and headers; some useful documentation.	Moderate level of code commentary, headers, documentation.	Good code commentary and headers; useful documentation; some weak areas.	Self-descriptive code; documentation up-to-date, well-organized, with design rationale.
SU Increment to ESLOC	50	40	30	20	10

Table 5.10 Rating Scale for Software Understanding Increment SU

UNFM Increment	Level of Unfamiliarity
0.0	Completely familiar
0.2	Mostly familiar
0.4	Somewhat familiar
0.6	Considerably familiar
0.8	Mostly unfamiliar
1.0	Completely unfamiliar

Table 7. Rating Scale for Programmer Unfamiliarity (UNFM)

4. TOOL ARCHITECTURE AND DESIGN

4.1 Tool Architecture

In this section architecture of the tool is described which estimates unadjusted function point in early development life cycle of object oriented software. Harput transformation rules are applied to estimate function points. In this section architecture of tool is presented which estimates early design software metrics in layered approach, in first layer object oriented function points are calculated based on UML design specification on applying harput rules [20]. These function points are converted into source line of codes (SLOC), primary input for COCOMO II to calculate General Software Metrics, which is done in layer two of the tool. The most fundamental calculation in the COCOMO II model is the use of the Effort Equation to estimate the number of Person-Months required developing a project. Most of the other COCOMO II results are derived from this quantity. In this model, some of the most important factors contributing to a project's duration and cost are the Scale Drivers. By using COCOMO II we can estimate effort in person month and development time. Now other metrics can be converted to by means of the following techniques.

1. PM to Dollars – On the basis of hourly salary
2. Productivity = FP/PM
3. Productivity = KLOC/PM
4. Development Cost = $/FP
5. Development Cost = $/LOC
6. Documentation= pages-of-documentation/FP
7. Documentation = pages-of-documentation/KLOC

Advanced software metrics are quality and maintenance of project. These metrics are calculated after completion of implementation phase of software development life cycle (SDLC). Here we are dealing with early design phase so these advanced metrics are not to be considered. Our main approach is to estimate software metrics in early design phase.

Fig 4.1 shows the architecture of tool.

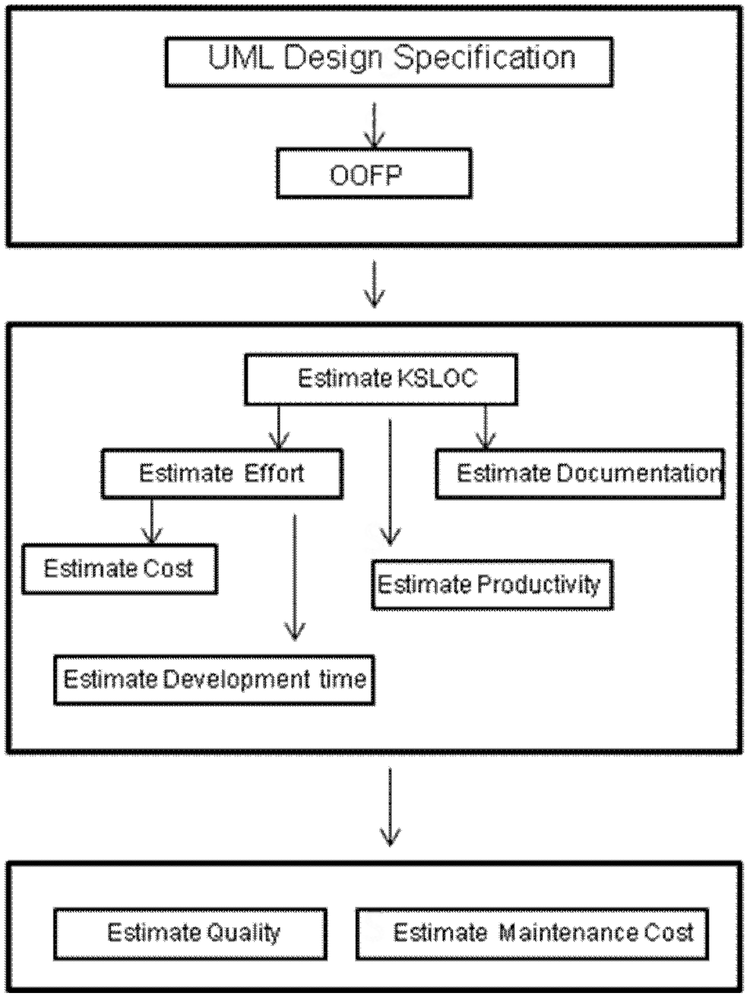


Fig. 4.1 Architecture of Tool

4.2 Design Approach

4.2.1 First layer Design

First layer of tool estimates function point based on UML design specification. UML design (Class diagram, Sequence diagram) developed in Rational Rose, as an input resource. We used Rational Rose Class diagram to estimate Data function types and Sequence diagram for Transactional function type estimation. Class diagram and sequence diagram by Rational rose generates design specification in UML syntax which is analyzed by analysis unit by applying Harput transformation rules. Both analysis unit and counting unit follow the rules to estimates unadjusted function points (UFP).

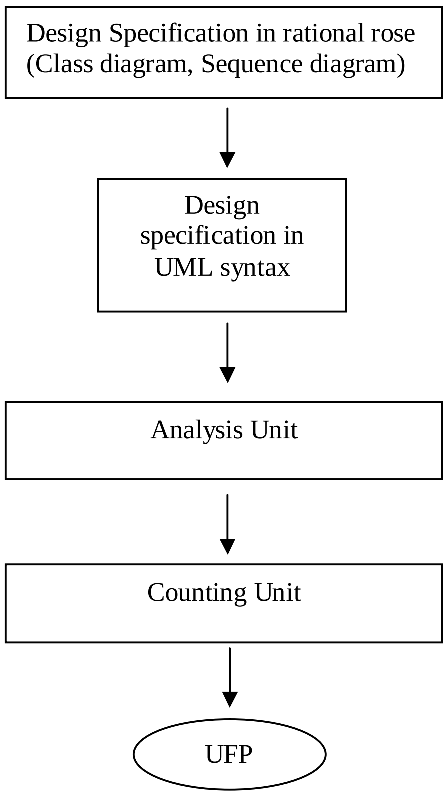

Design Specification in rational rose (Class diagram, Sequence diagram)
Design specification in UML syntax
Analysis Unit
Counting Unit
UFP

Fig. 4.2 Size estimation in First layer

4.2.2 First Layer Design Algorithm

Algorithm to estimate data function type:

```
Step 1: Identification of ILF and EIF
        LF_SingleClass()                 // According to Harput rule 3
        LF_Generalization()              // According to Harput rule 4,5
        LF_Association()                 // According to Harput rule 15, 16
        LF_Aggregation()                 // According to Harput rule 6

Step 2: Estimation of DET: DET_calc()   // According to Harput rule 7,8,17,18

Step 3: Estimation of RET
        RET_SingleClass()                //According to Harput rule 9
        RET_ Generalization()            // According to Harput rule 12,13
        RET_Association()                // According to Harput rule 9
        RET_Aggregation()                // According to Harput rule 14

Step 4: Complexity estimation of files: cmplx_est()

Step 5: DataFunctions()
```

Algorithm to estimate transactional function:

```
Step 1: Identification of External Input (EI), External Output (EO) and External Enquiry
(EQ)                                       // According to Harput rule 2

Step 2: Estimation of DET: DET_calc()     // According to Harput rule 8,9

Step 3: Estimation of FTR: FTR_calc()     // According to Harput rule 7

Step 4: Complexity estimation of files: cmplx_est()

Step 5: TransactionalFunctions()
```

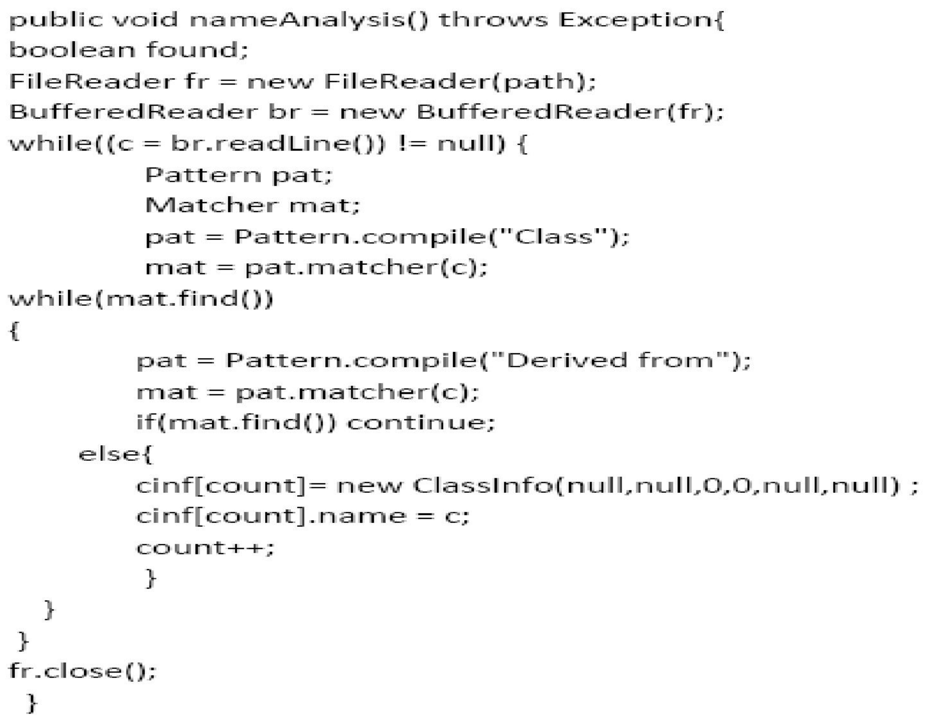
Unadjusted Function Calculation:

```
Step 1: UFP=DataFunctions() + TransactionalFunctions()
```

According to algorithm given function point measurement is done as follows:

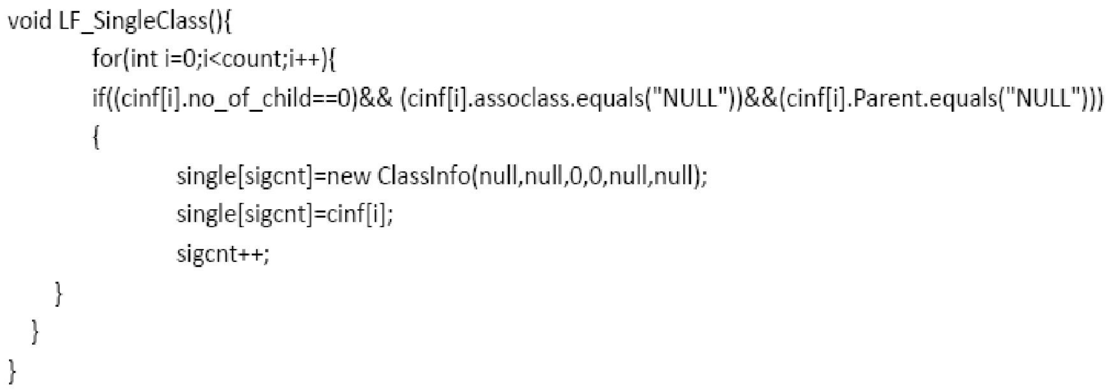
First Data function types are estimated

In step 1 identification of data function types is done. Function In step 1 LF_SingleClass(), LF_Generalization(), LF_Association() and LF_Aggregation() return the set of classes in each logical file e.g. LF_Generalization() returns the all the set of classes which are candidates for data function types or we can say in step 1 data function types are evaluated. Fig 4.3 shows the pseudo code in which first class information is extracted from UML syntax textual description.



```
public void nameAnalysis() throws Exception{
boolean found;
FileReader fr = new FileReader(path);
BufferedReader br = new BufferedReader(fr);
while((c = br.readLine()) != null) {
          Pattern pat;
          Matcher mat;
          pat = Pattern.compile("Class");
          mat = pat.matcher(c);
while(mat.find())
{
         pat = Pattern.compile("Derived from");
         mat = pat.matcher(c);
         if(mat.find()) continue;
     else{
         cinf[count]= new ClassInfo(null,null,0,0,null,null) ;
         cinf[count].name = c;
         count++;
          }
  }
 }
fr.close();
 }
```

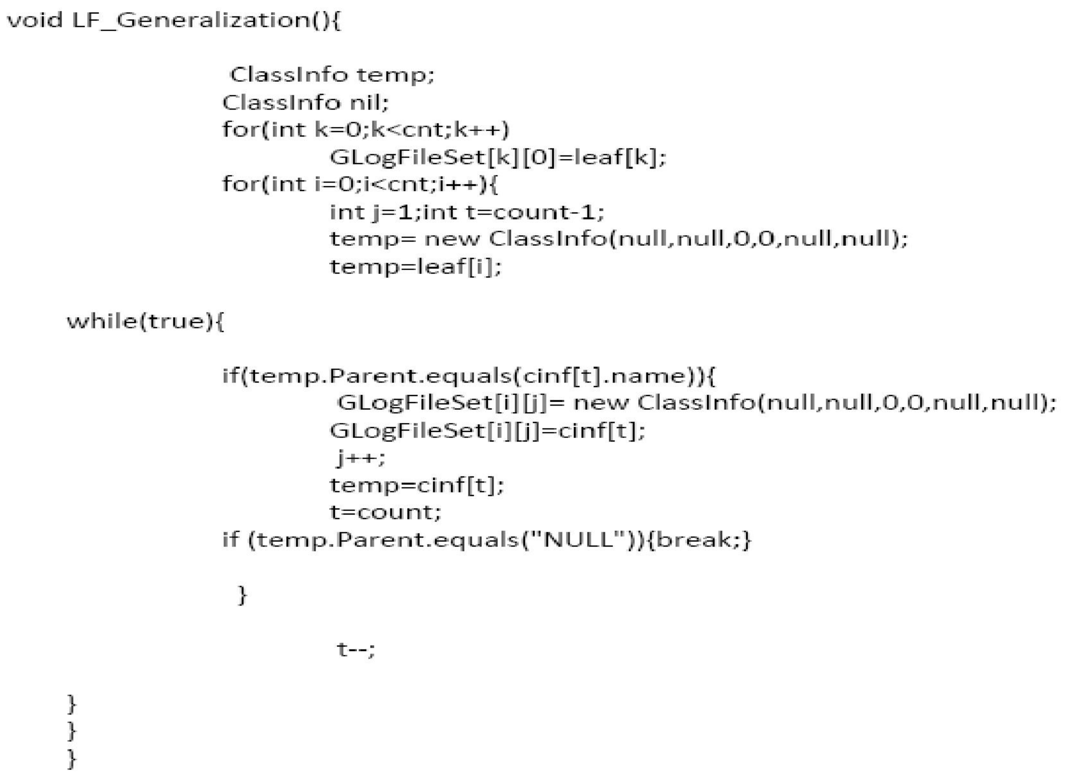
Fig 4.3 Pseudo code to analyze class information



```
void LF_SingleClass(){
        for(int i=0;i<count;i++){
        if((cinf[i].no_of_child==0)&& (cinf[i].assoclass.equals("NULL"))&&(cinf[i].Parent.equals("NULL")))
        {
                single[sigcnt]=new ClassInfo(null,null,0,0,null,null);
                single[sigcnt]=cinf[i];
                sigcnt++;
    }
  }
}
```

Fig 4.4 Pseudo code to LF_singleClass

Fig 4.4, 4.5, 4.6 and 4.7 shows the pseudo code for LF_Generalization, LF_Association and LF_Aggregation.

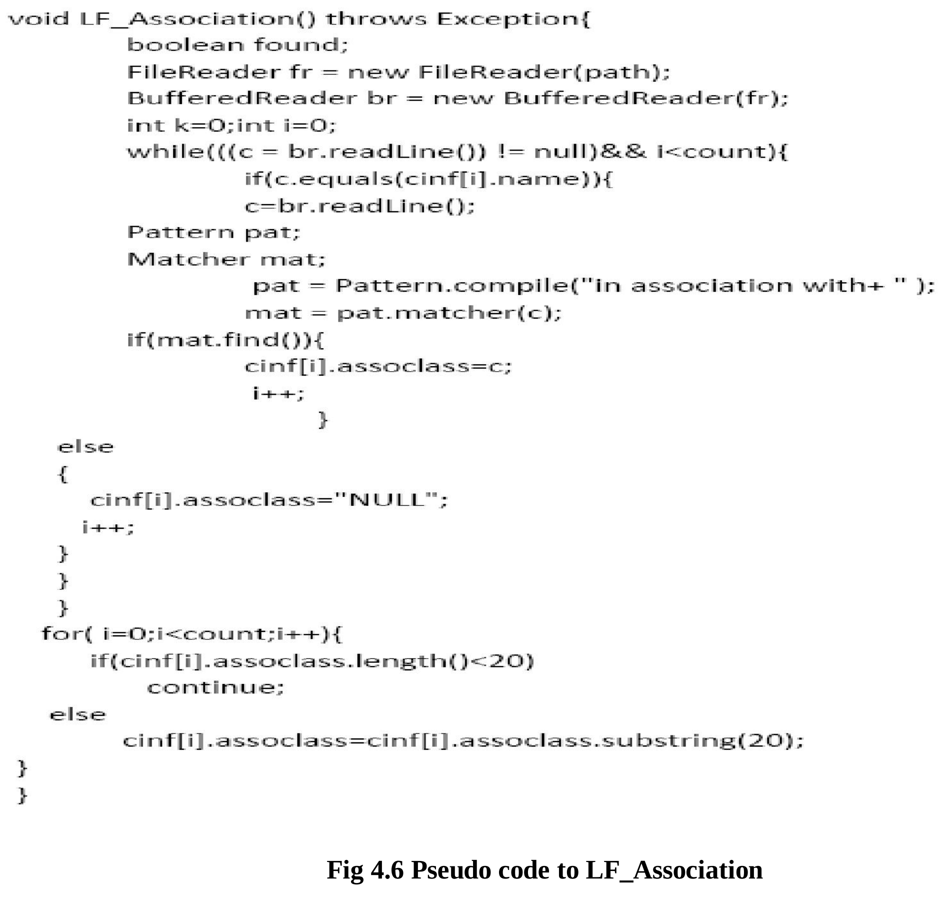


```
void LF_Generalization(){

                   ClassInfo temp;
                  ClassInfo nil;
                  for(int k=0;k<cnt;k++)
                           GLogFileSet[k][0]=leaf[k];
                  for(int i=0;i<cnt;i++){
                           int j=1;int t=count-1;
                           temp= new ClassInfo(null,null,0,0,null,null);
                           temp=leaf[i];

     while(true){

                  if(temp.Parent.equals(cinf[t].name)){
                            GLogFileSet[i][j]= new ClassInfo(null,null,0,0,null,null);
                           GLogFileSet[i][j]=cinf[t];
                            j++;
                           temp=cinf[t];
                           t=count;
                  if (temp.Parent.equals("NULL")){break;}

                    }

                            t--;

     }
     }
     }
```

Fig 4.5 Pseudo code to LF_Generalization

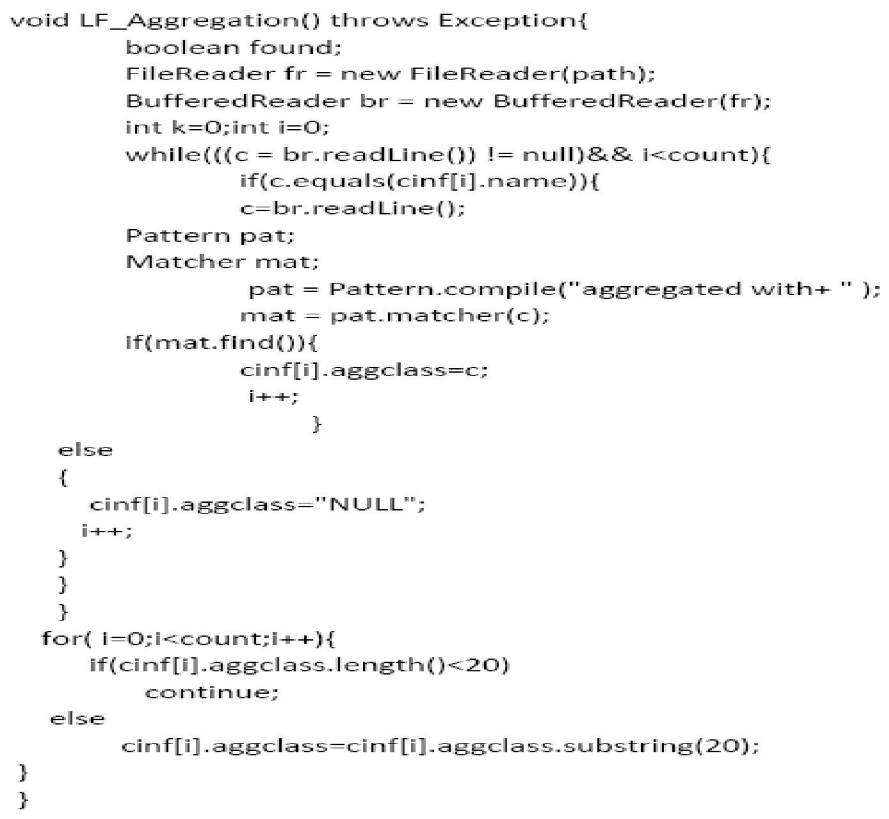
Step 2 determines total numbers of data element types for each logical file, here DET_calc() is used for estimation of DET this function will take each data function types as arguments and returns the total numbers of data element types of corresponding data function.

Step 3 estimates record element types of each data function here RET_SingleClass() which returns the numbers of record element types of data function mapped by only one class, RET_Generalization() takes data function mapped by generalization hierarchy and returns total numbers of RET for corresponding data function e.g. in figure 2 {A,B,D}, {A,B,E} and {A,C} are taken as logical files having RET 3,3 and 2 respectively. Same for RET_Association and RET_Aggregation().



```
void LF_Association() throws Exception{
        boolean found;
        FileReader fr = new FileReader(path);
        BufferedReader br = new BufferedReader(fr);
        int k=0;int i=0;
        while(((c = br.readLine()) != null)&& i<count){
                if(c.equals(cinf[i].name)){
                c=br.readLine();
        Pattern pat;
        Matcher mat;
                 pat = Pattern.compile("in association with+ " );
                mat = pat.matcher(c);
        if(mat.find()){
                cinf[i].assoclass=c;
                 i++;
                     }
   else
   {
     cinf[i].assoclass="NULL";
    i++;
   }
   }
   }
  for( i=0;i<count;i++){
     if(cinf[i].assoclass.length()<20)
         continue;
   else
        cinf[i].assoclass=cinf[i].assoclass.substring(20);
}
}
```

Fig 4.6 Pseudo code to LF_Association



```
void LF_Aggregation() throws Exception{
        boolean found;
        FileReader fr = new FileReader(path);
        BufferedReader br = new BufferedReader(fr);
        int k=0;int i=0;
        while(((c = br.readLine()) != null)&& i<count){
                if(c.equals(cinf[i].name)){
                c=br.readLine();
        Pattern pat;
        Matcher mat;
                 pat = Pattern.compile("aggregated with+ " );
                mat = pat.matcher(c);
        if(mat.find()){
                cinf[i].aggclass=c;
                 i++;
                     }
   else
   {
      cinf[i].aggclass="NULL";
     i++;
   }
   }
   }
  for( i=0;i<count;i++){
     if(cinf[i].aggclass.length()<20)
         continue;
  else
        cinf[i].aggclass=cinf[i].aggclass.substring(20);
}
}
```

Fig 4.7 Pseudo code to LF_Association

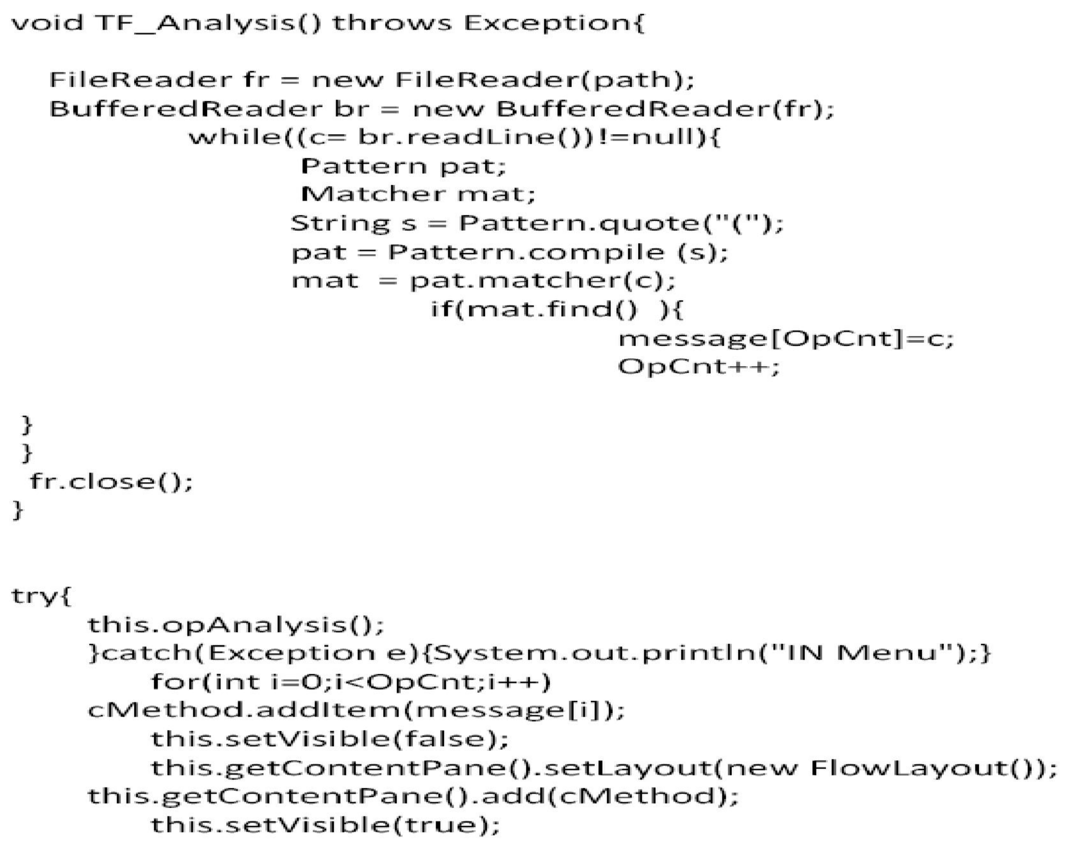
Cmplx_est() is a function which assign complexity to each data function based on DET/RET complexity matrix.

Finally in step 5 DataFunction() will returns the total data function with all details.

Transactional function types are calculated as follows

In step 1 and step 2 identification of transactional function is done on the basis of messages sequences (ADMS, SDMS).

Identify_EI (), Identify_EO() and Identify_EQ() will return the type of transactional functions on the basis of messages sequences, if messages is SDMS than identified transactional function is regarded as External input, and if messages sequence is ADMS and messages contain derived in their arguments than this will be external output otherwise external query is taken as transactional function.



```
void TF_Analysis() throws Exception{

   FileReader fr = new FileReader(path);
   BufferedReader br = new BufferedReader(fr);
            while((c= br.readLine())!=null){
                    Pattern pat;
                    Matcher mat;
                   String s = Pattern.quote("(");
                   pat = Pattern.compile (s);
                   mat  = pat.matcher(c);
                             if(mat.find()  ){
                                          message[OpCnt]=c;
                                          OpCnt++;

 }
 }
  fr.close();
}


try{
      this.opAnalysis();
      }catch(Exception e){System.out.println("IN Menu");}
          for(int i=0;i<OpCnt;i++)
      cMethod.addItem(message[i]);
          this.setVisible(false);
          this.getContentPane().setLayout(new FlowLayout());
      this.getContentPane().add(cMethod);
          this.setVisible(true);
```

Fig. 4.10 Transactional Function Analysis

Step 2 and step three calculates total number of data element types and file types references.

Step 4 will assign complexity to each transactional function based on their complexity table.

Finally in step 5 TransactionalFunction() will returns the total data function with all details

4.2.4 Second layer Design

In this layer general software metrics are estimated e.g. effort, development time, cost, productivity etc. SLOC is the input for COCOMO II estimation technique, so unadjusted function points, estimated in first layer are converted into Source lines of code (see table 3.1) then scale drivers and cost drivers for early design model (see table 3.2 and 3.4) are to be set and finally we use COCOMO II formula to estimate general software metrics. We have introduced COCOMO II in chapter three, here COCOMO II is used to estimate general software metrics. Our first layer output is size of object oriented system in unadjusted function points (UFP), this is converted into source lines of code (SLOC) (see table 3.1).

Scale drivers (see table 3.2) are adjusted according to relative economies or diseconomies of scale encountered for software projects of different sizes.

We are using early design model of COCOMO II, using this model effort is estimated in nominal person-month then set of effort multipliers (see table 3.4) to adjust the nominal person-month. Early design model uses KSLOC to evaluate effort in person month.

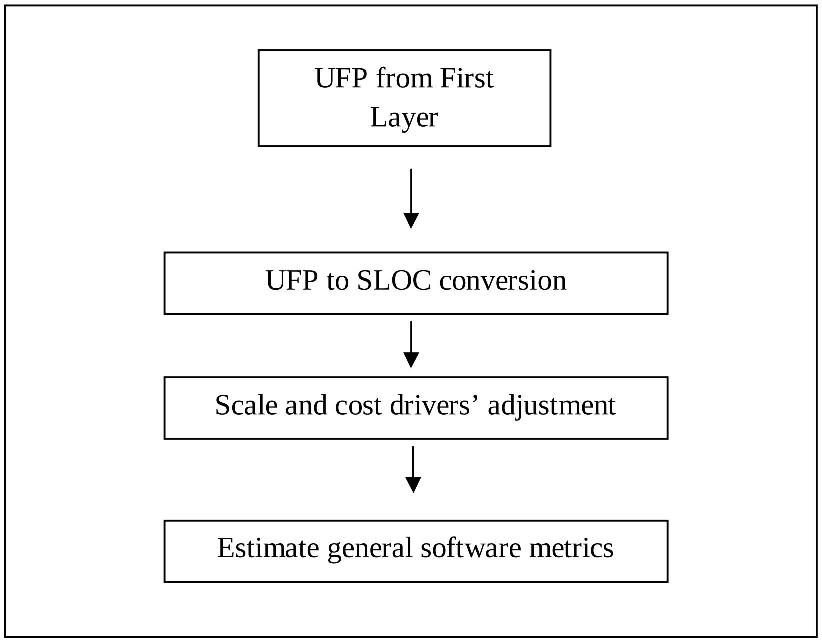

UFP from First Layer
UFP to SLOC conversion
Scale and cost drivers' adjustment
Estimate general software metrics

Fig. 4.9 General software metrics estimation in second layer

5. IMPLEMENTATION DETAILS

5.1 Rational Rose

Rational rose is most popular UML design tool. We have used rational rose enterprise edition 2000. First we develop UML diagram (Class diagram and sequence diagram) then Rational rose generates documentation report which is based on UML design specification and taken as input resource to estimate function points. Figure 5.1 shows the Class diagram of ATM system developed in rational rose.

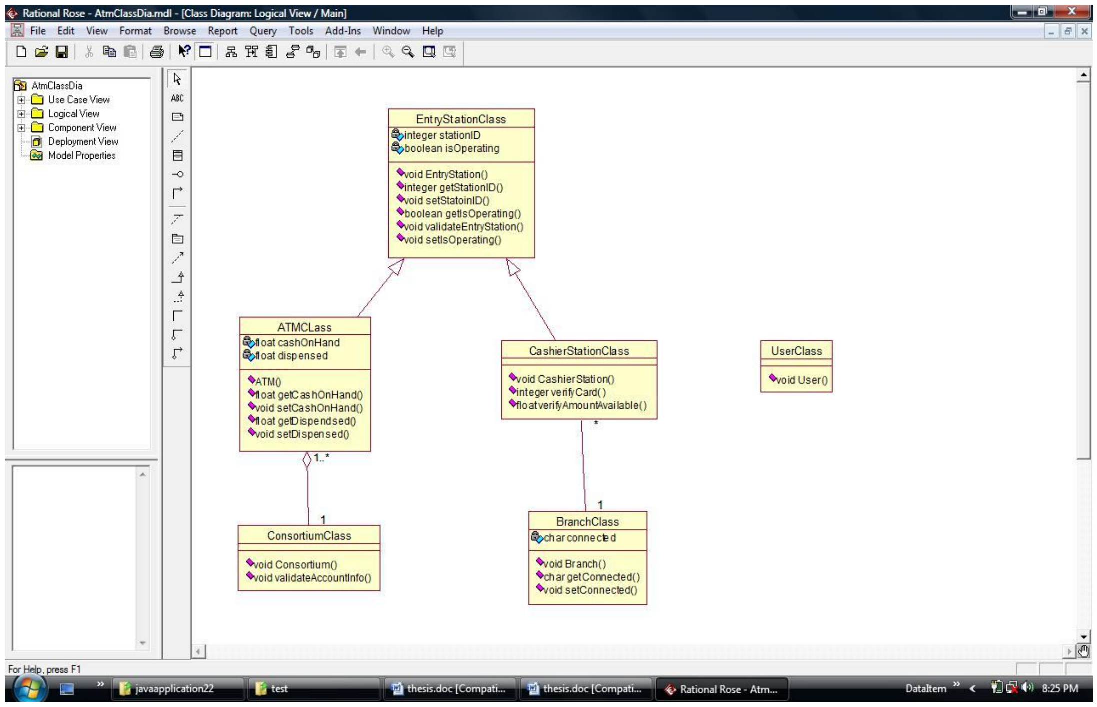
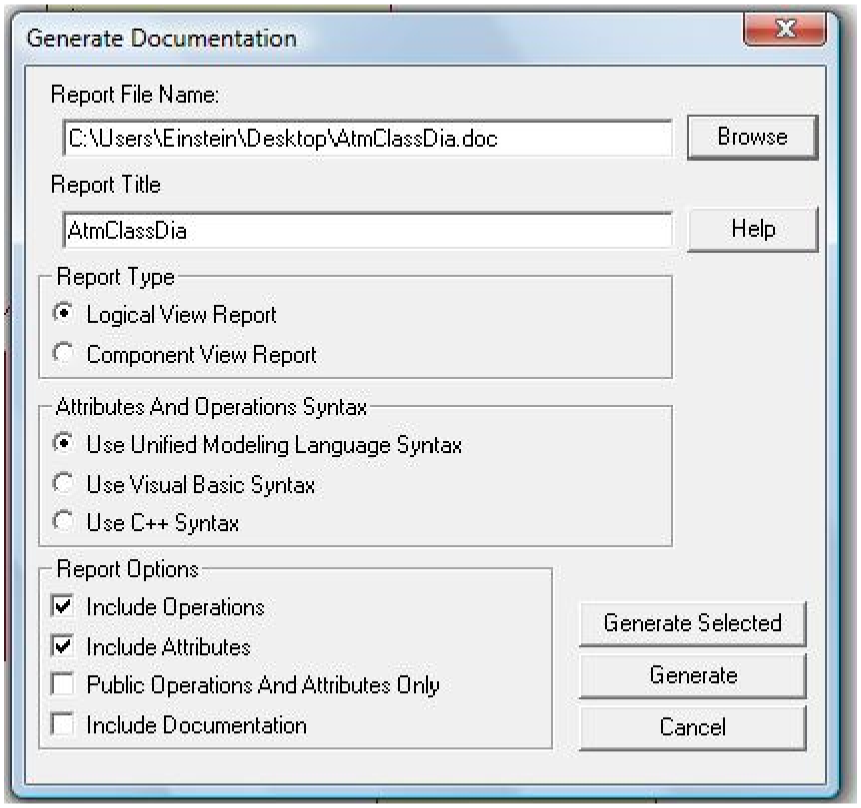


Fig. 5.1 Rational Rose Class diagram for ATM system



Generate Documentation

Report File Name: C:\Users\Einstein\Desktop\AtmClassDia.doc

Report Title: AtmClassDia

Report Type
- [✓ Logical View Report]
- Component View Report

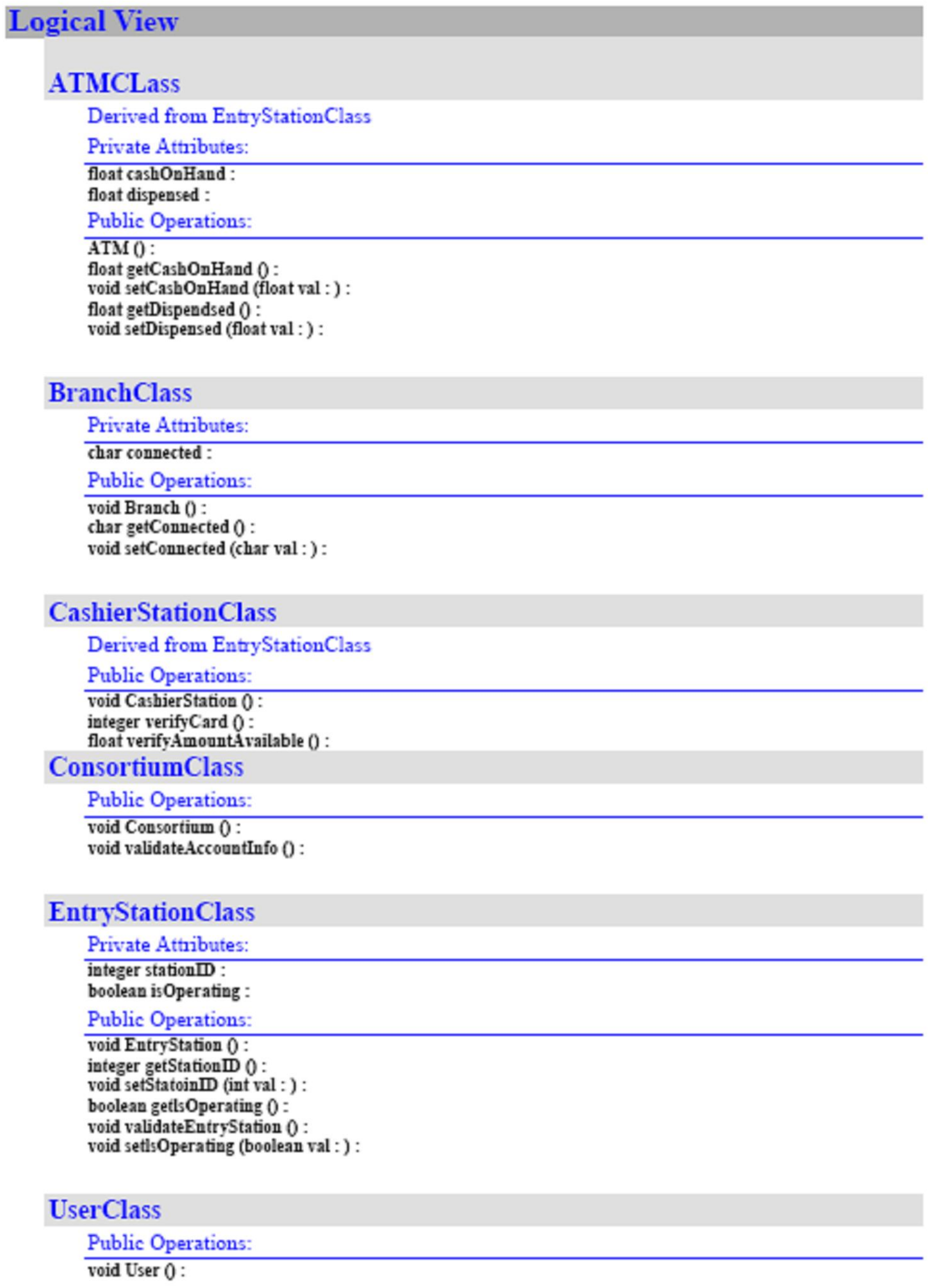
Attributes And Operations Syntax
- [✓ Use Unified Modeling Language Syntax]
- Use Visual Basic Syntax
- Use C++ Syntax

Report Options
- [✓ Include Operations]
- [✓ Include Attributes]
- Public Operations And Attributes Only
- Include Documentation

Fig. 5.3 Document generation using Rational rose in UML syntax

Step to generate UML documentation

1. Locate the generated file into directory.
2. Assign Report title.
3. Assign Report type as logical view report.
4. Select Unified modeling language syntax.
5. Press Generate to generate documentation.



Logical View

ATMCLass

Derived from EntryStationClass

Private Attributes:

float cashOnHand :
float dispensed :

Public Operations:

ATM () :
float getCashOnHand () :
void setCashOnHand (float val :) :
float getDispendsed () :
void setDispensed (float val :) :

BranchClass

Private Attributes:

char connected :

Public Operations:

void Branch () :
char getConnected () :
void setConnected (char val :) :

CashierStationClass

Derived from EntryStationClass

Public Operations:

void CashierStation () :
integer verifyCard () :
float verifyAmountAvailable () :

ConsortiumClass

Public Operations:

void Consortium () :
void validateAccountInfo () :

EntryStationClass

Private Attributes:

integer stationID :
boolean isOperating :

Public Operations:

void EntryStation () :
integer getStationID () :
void setStatoinID (int val :) :
boolean getIsOperating () :
void validateEntryStation () :
void setIsOperating (boolean val :) :

UserClass

Public Operations:

void User () :

Fig. 5.4 Design specification in UML syntax

Fig 5.4 shows the design specification in UML syntax generated by rational rose. This is the input for first layer of the tool (see fig 4.1). Tool read this file line by line and extracts required data set to estimates size in Unadjusted Function Points

5.2 Platform Used

We have used JAVA (JDK 1.6.0) as a platform to implement this tool. The JDK is a development environment for building applications, applets, and components using the Java programming language. The JDK includes tools useful for developing and testing programs written in the Java programming language and running on the Java platform.

5.2.1 Netbean IDE

We have used the tool Netbean IDE (Integrated development environment) to develop our application under the environment of JDK. Netbean IDE is development tool based on JAVA (1.6.0) developed by Sun Microsystems Inc., it is supported by Windows vista, XP operating system. Netbean tool have editor view and design view, Using Design view of netbean tool user can use JDK's swings and AWT utility, in easy way i.e. pick and locate. Design view is shown in fig 5.5. in editor view user can develop JAVA application using JDK's different types of utility, e.g. swings, beans, abstract window toolkit(AWT). Netbean tool have its own GUI window using which user can easily perform the task e.g. run, debug, compile, edit etc. Editor view is shown in fig 5.6.

We have used JAVA swings to design the Graphical User Interface (GUI), to develop the tool. Fig 5.5 and 5.6 shows the Netbean development GUI design and program editor window.

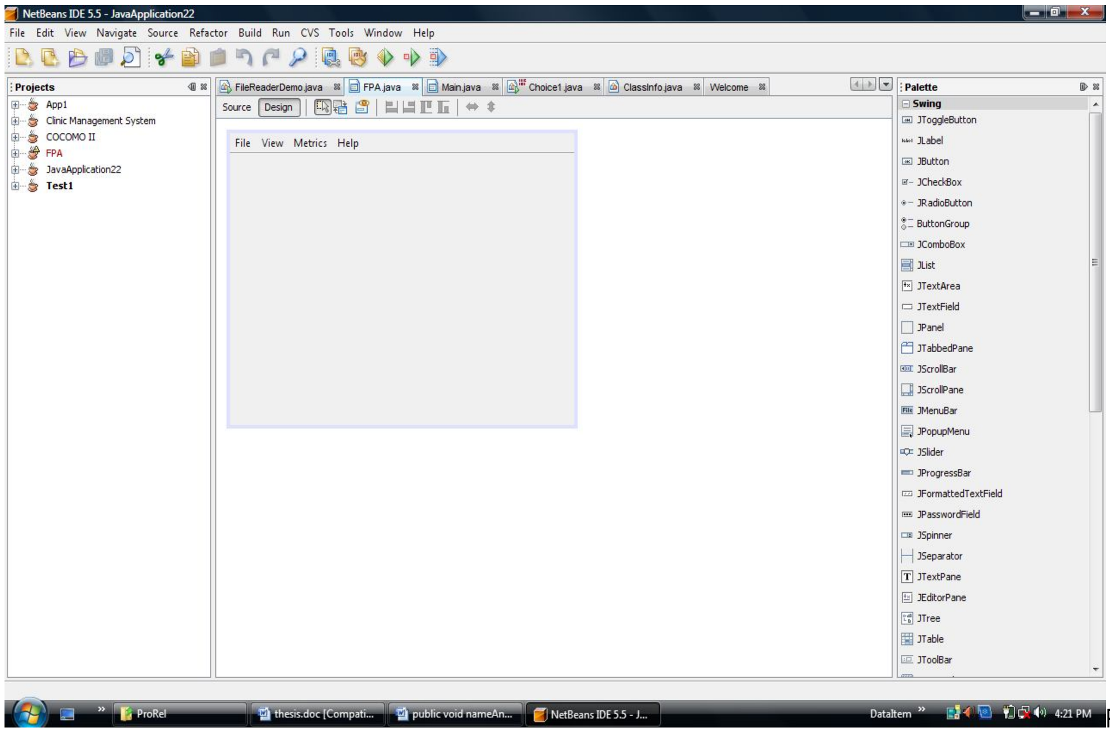
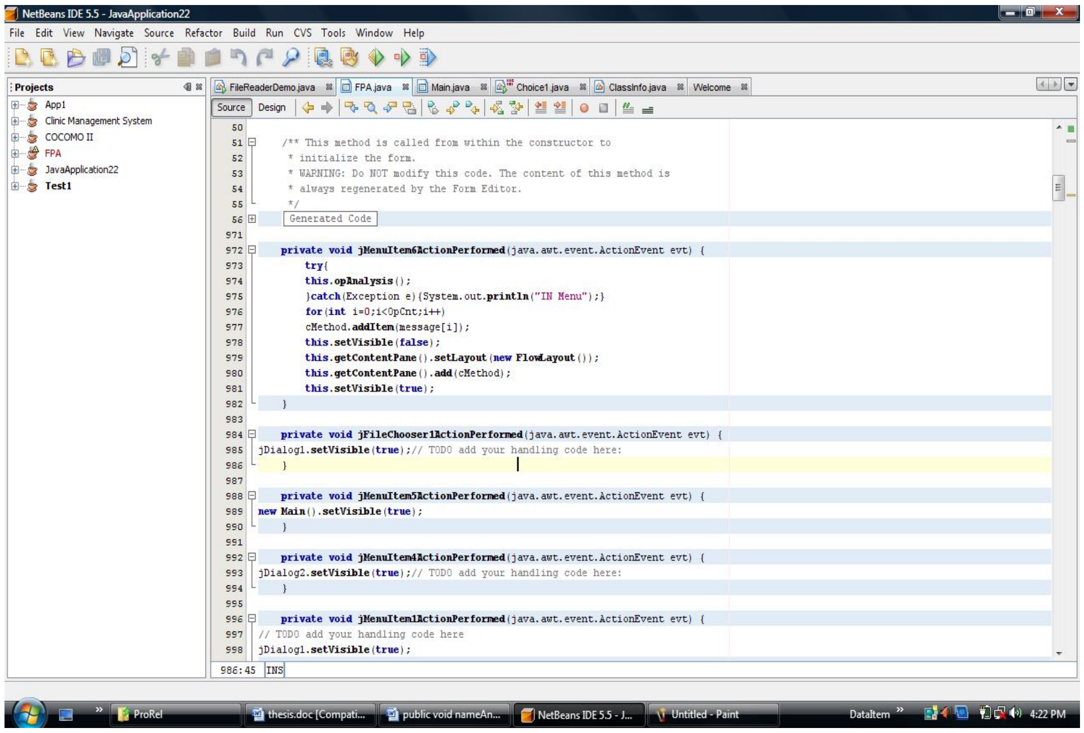


Fig.

5.5 GUI design in Netbean IDE tool



```

    /** This method is called from within the constructor to
     * initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is
     * always regenerated by the Form Editor.
     */
    Generated Code

    private void jMenuItem6ActionPerformed(java.awt.event.ActionEvent evt) {
        try{
        this.opAnalysis();
        }catch(Exception e){System.out.println("IN Menu");}
        for(int i=0;i<OpCnt;i++)
        cMethod.addItem(message[i]);
        this.setVisible(false);
        this.getContentPane().setLayout(new FlowLayout());
        this.getContentPane().add(cMethod);
        this.setVisible(true);
    }

    private void jFileChooser1ActionPerformed(java.awt.event.ActionEvent evt) {
jDialog1.setVisible(true);// TODO add your handling code here:
    }

    private void jMenuItem5ActionPerformed(java.awt.event.ActionEvent evt) {
new Main().setVisible(true);
    }

    private void jMenuItem4ActionPerformed(java.awt.event.ActionEvent evt) {
jDialog2.setVisible(true);// TODO add your handling code here:
    }

    private void jMenuItem1ActionPerformed(java.awt.event.ActionEvent evt) {
// TODO add your handling code here
jDialog1.setVisible(true);
```

Fig. 5.6 Program editor in Netbean IDE tool

5.2.2 Software Architecture of tool

Fig 5.7 shows the software architecture of tool, UML rational rose tool is used to design class diagram and sequence diagram, then its documentation report is generated in UML syntax which is analyzed by syntax analysis unit design in JDK platform in netbean IDE editor, syntax analysis unit will transfer all required data to which is stored in runtime database, than counting unit fetch data from memory to estimate function point, finally it calculates basic and general software metrics.

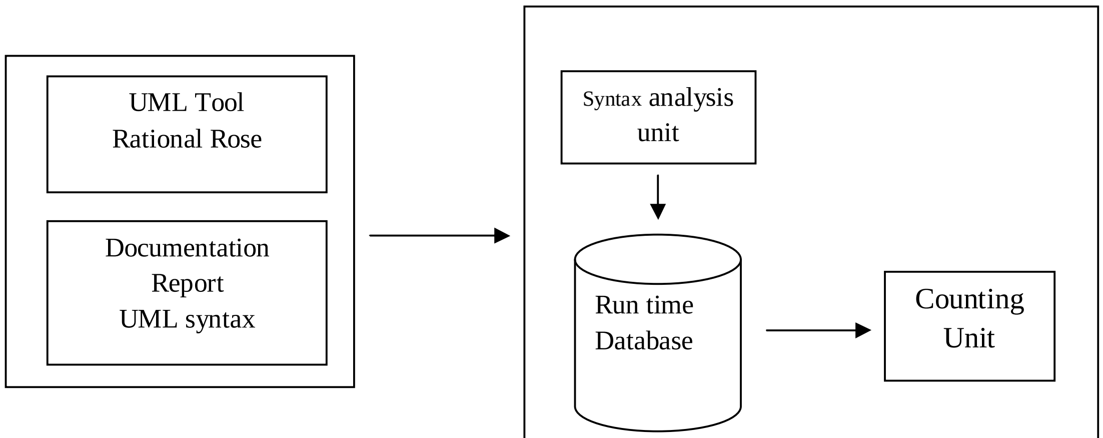


Fig. 5.7 Software Architecture of tool

Fig 5.7 shows dialog box which browse input text file and load it into memory, then tool will analyze input file to estimate basic software metrics. After uploading the input file we analyze transactional function as given in fig 5.8.

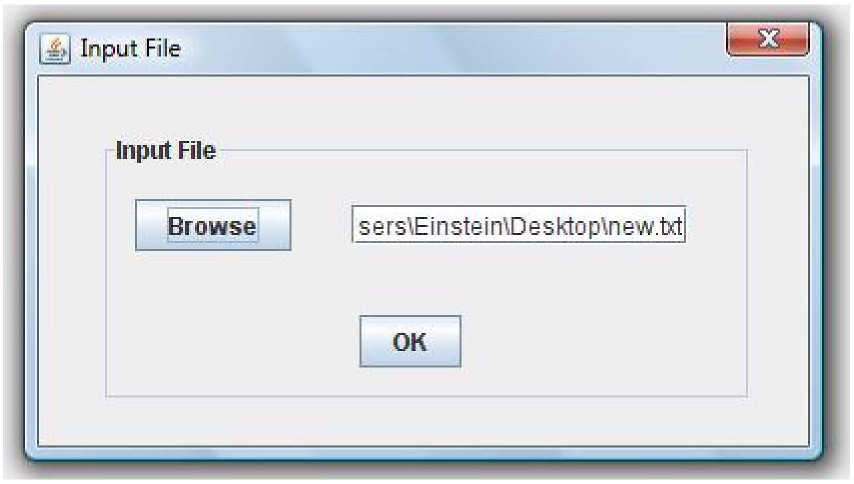
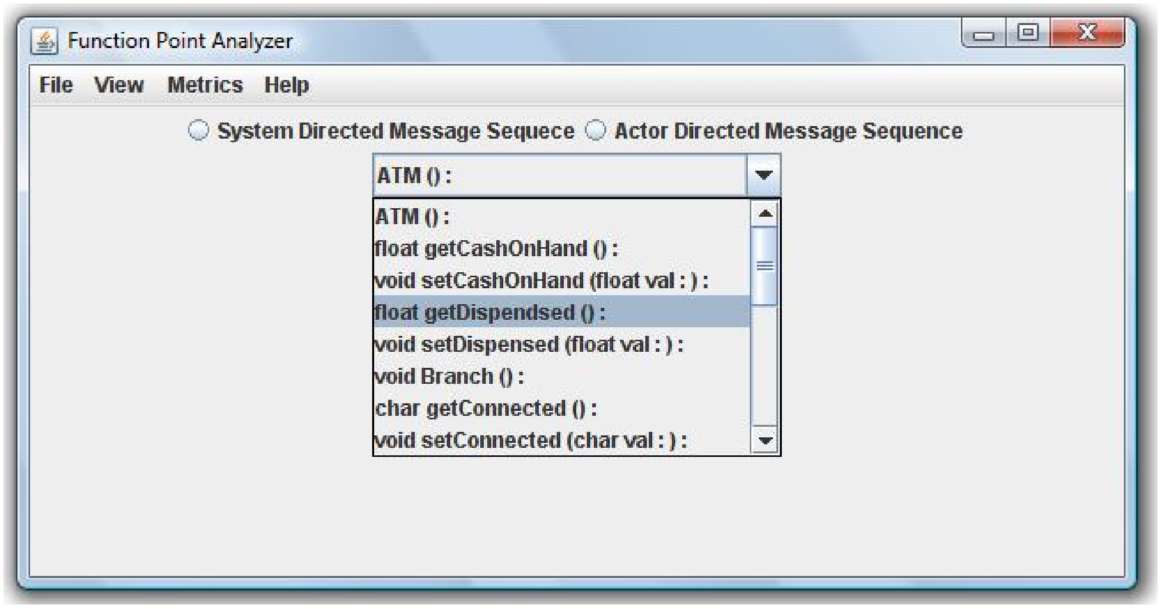


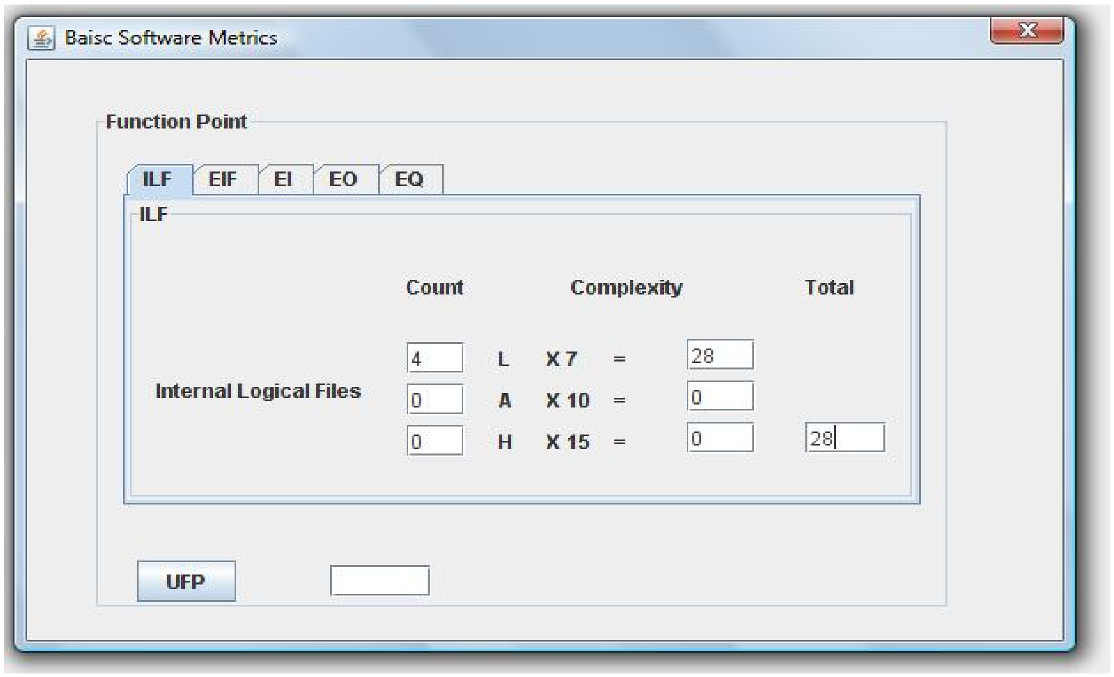
Fig. 5.8 Input File



Function Point Analyzer
File View Metrics Help
System Directed Message Sequece
Actor Directed Message Sequence
ATM () :

```
ATM () :
float getCashOnHand () :
void setCashOnHand (float val : ) :
float getDispendsed () :
void setDispensed (float val : ) :
void Branch () :
char getConnected () :
void setConnected (char val : ) :
```

Fig. 5.9 Transactional function analysis

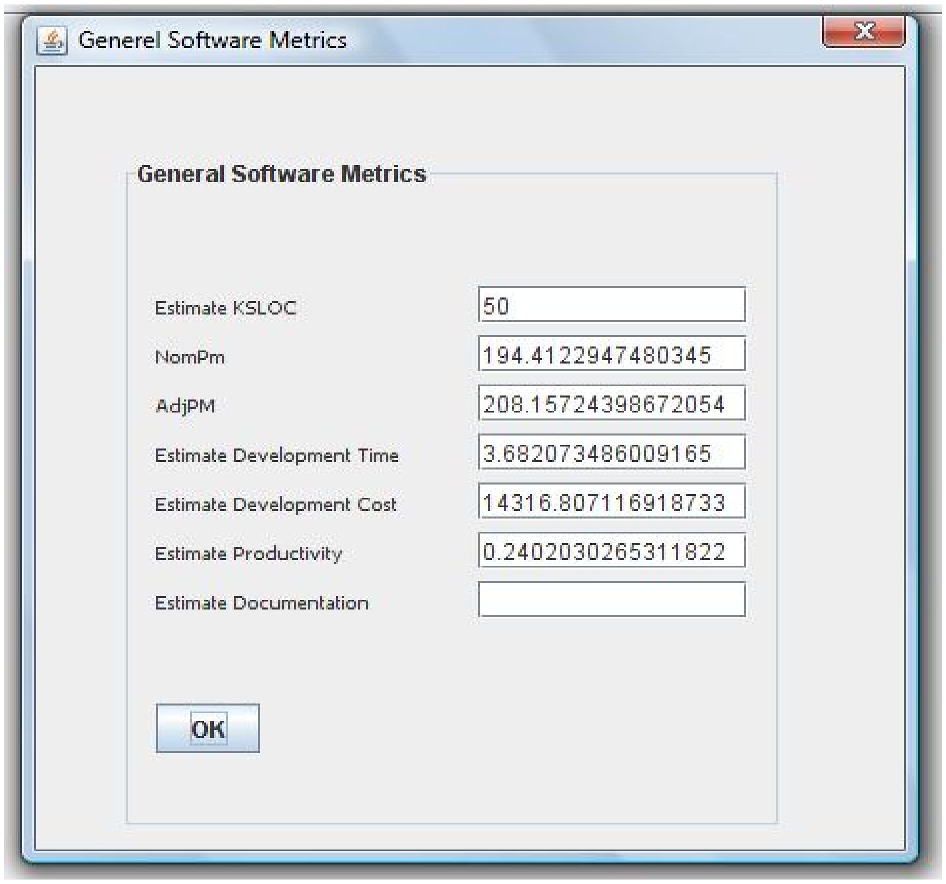


	Count		Complexity			Total
Internal Logical Files	4	L	X 7	=	28	
	0	A	X 10	=	0	
	0	H	X 15	=	0	28

Fig. 5.10 Basic Software Metrics

Fig 5.6 shows the window of our Function Point Analyzer (FPA) in which input file which describes specification details is given to the tool then it estimate datafunction and transactional function and finally calculates unadjusted function points metrics.

The output of FPA tool is converted into source lines of codes (see table 4.1)


Generel Software Metrics

General Software Metrics

Estimate KSLOC	50
NomPm	194.4122947480345
AdjPM	208.15724398672054
Estimate Development Time	3.682073486009165
Estimate Development Cost	14316.807116918733
Estimate Productivity	0.2402030265311822
Estimate Documentation	

Fig. 5.11 General Software Metrics

Fig 5.7 shows tool window in which general software metrics are estimated using COCOMO II technique.

6. CASE STUDY OF HOSPITAL MANAGMENT SYSTEM

In this chapter case study of Hospital Management System is described. we have applied all Harput rules to estimate size of Hospital Management System. Class diagram and sequence diagram of various events of Hospital management system are taken as a reference to estimates function poins, than we have applied Harput rules.

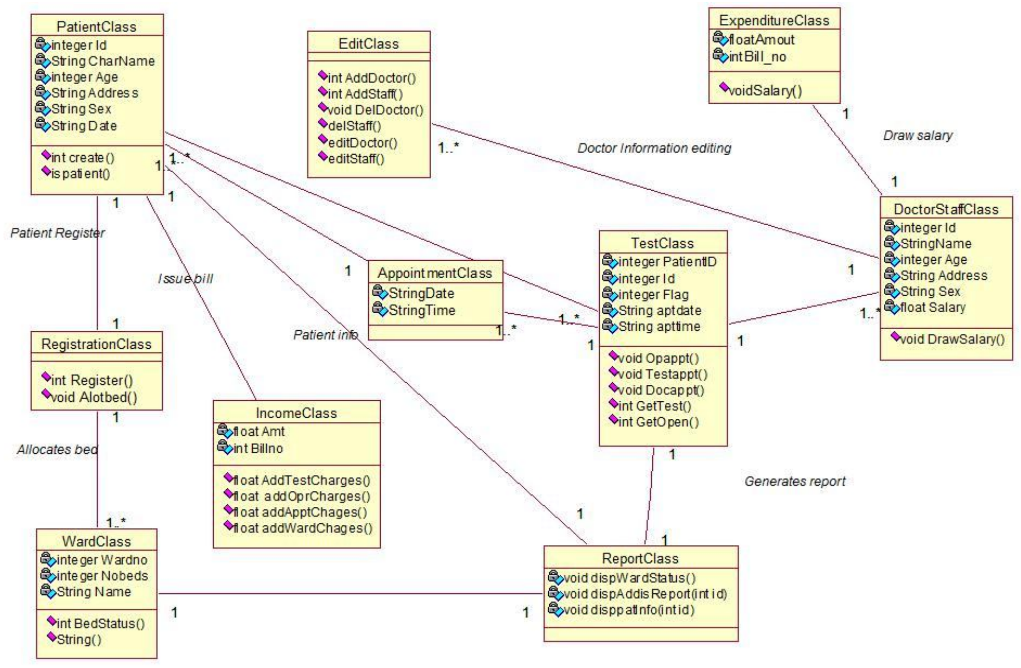


Fig. 6.1 Class diagram of Hospital Management System

Size is estimated in Unadjusted function point which is converted into source lines of codes (SLOC), see table 3.1. SLOC is the necessary input to COCOMO II, applying COCOMOII we have calculated rest of the software metrics.

Data function type estimation

Objects that have operations which change the attributes of other objects in exchanging the data are regarded as Internal Logical Files, hence we will take all classes in given class diagram as candidates for internal logical files. According to Harput Rule for Data Function Types 7, Attributes of classes represent the data element types (DETs) of the file e.g. (Patient, Registration) having 6 data element types. We have associated classes. having binary association so we have mapped these classes according to rule e.g. (Patient, Registration) due to association considered as a single logical file, (see Harput rule for data function types 16). According to harput for rule data element types 17, associated classes is increased e.g. (Patient, Registration) 7 data element type, 6 due to total no of attributes and 1 due to rule. On the basis of complexity matrix, low complexity is assigned to each logical file given in Table 6.1. Total number of DETs and RETs are decided according to Harput rules for associated classes, according to rule data element types are total number of data member in associated classes and record element types can be determined by total no of classes and multiplicity between associated classes, e.g. (Patient, Appointment) having total number of DETs are 9 and RETs are 3, two due to number of classes and 1 increment due to multiplicity between classes.

InternalLogical Files	DET	RET	Complexit
Patient, Registration	7	2	LOW
Patient, Appoinment	9	3	LOW
Patient, Income	9	2	LOW
Patient, Test	10	3	LOW
Patient, Report	7	2	LOW
Registration, Ward	4	3	LOW
Ward, Report	4	2	LOW
DoctorStaff, Test	10	3	LOW
DoctorStaff, Edit	7	3	LOW
DoctorStaff, Expendr	8	2	LOW
Test, Report	3	2	LOW
Test, Appointment	5	2	LOW

Table 6.1 Internal Logical Files

Transactional function types estimation:

These messages from given sequence diagrams for different events are taken as candidates for transaction function types:

1. *addApptCharges(int id)*
2. *addApptCharges(int id)*
3. *addTestCharges(int id)*
4. *addTestCharges(int id)*
5. *addWardCharges(int id)*
6. *allotbed(int id)*
7. *getOper(int id)*
8. *delDoctor(int id)*

9. delStaff(int id)

10.editDoc(int id)

11.editStaff(int id)

Admission:

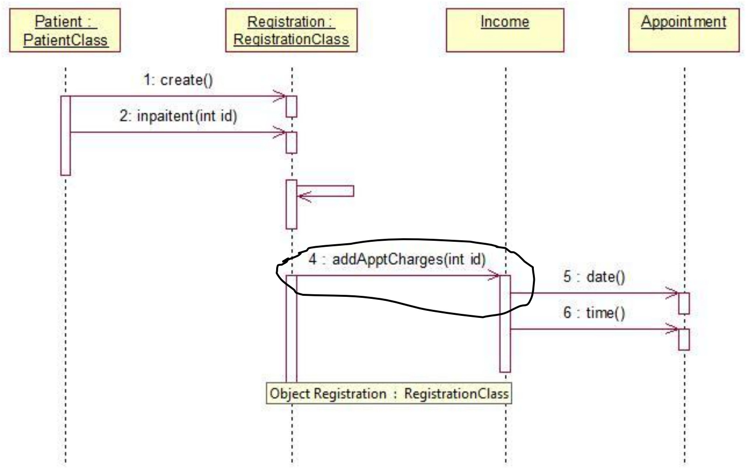

Patient : PatientClass
Registration : RegistrationClass
Income
Appointment
1: create()
2: inpaitent(int id)
4 : addApptCharges(int id)
5 : date()
6 : time()
Object Registration : RegistrationClass

Fig. 6.2 Sequence diagram for Admission event

Message addApptCharges(int id)

Actor: Income

Non-Actor: Registration

Here communication from non-actor to actor, so it can be identified as external output.

DET: 1, only one argument candidate message have.

FTR: 1, only one entity class is there.

Test Appointment:

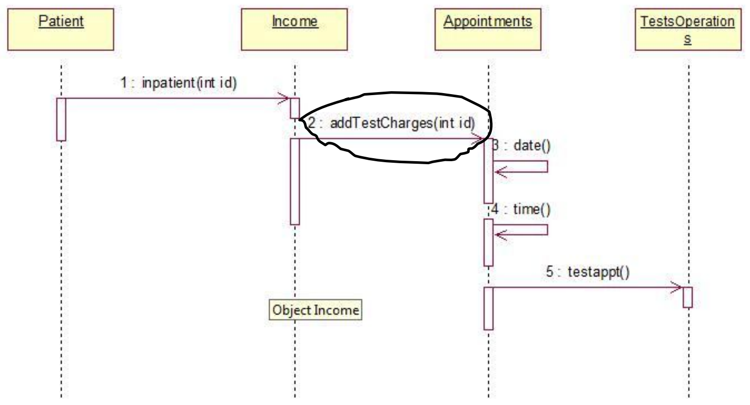


Fig. 6.3 Sequence diagram for Test Appointment event

Message: addTestCharges(int id)

Actor: Income

Non-Actor: Appointment

Here communication from actor to non-actor, so it can be identified as external input (EI).

DET: 1, only one argument candidate message have.

FTR: 1, only one entity class is there.

Doctor Appointments:

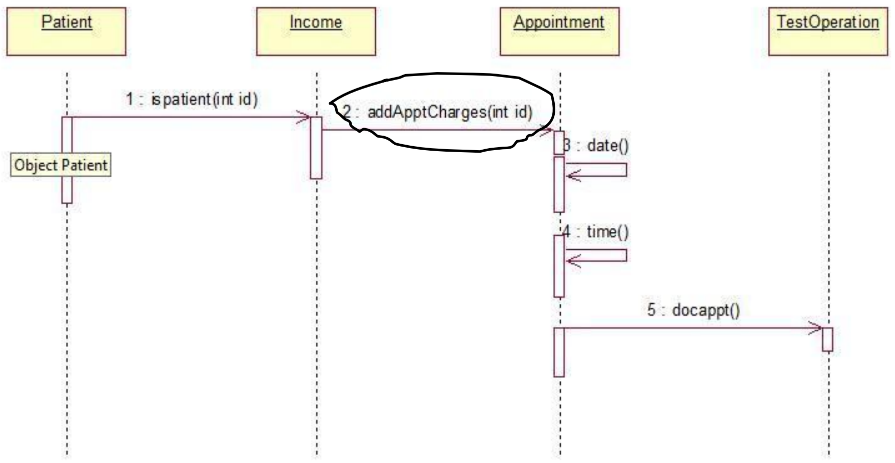


Fig. 6.4 Sequence diagram for Doctor Appointment event

Message: addApptCharges(int id)
Actor: Income

Non-Actor: Appointment

Here communication from actor to non-actor, so it can be identified as external input (EI).

DET: 1, only one argument candidate message have.

FTR: 1, only one entity class is there.

Bed Allotment:

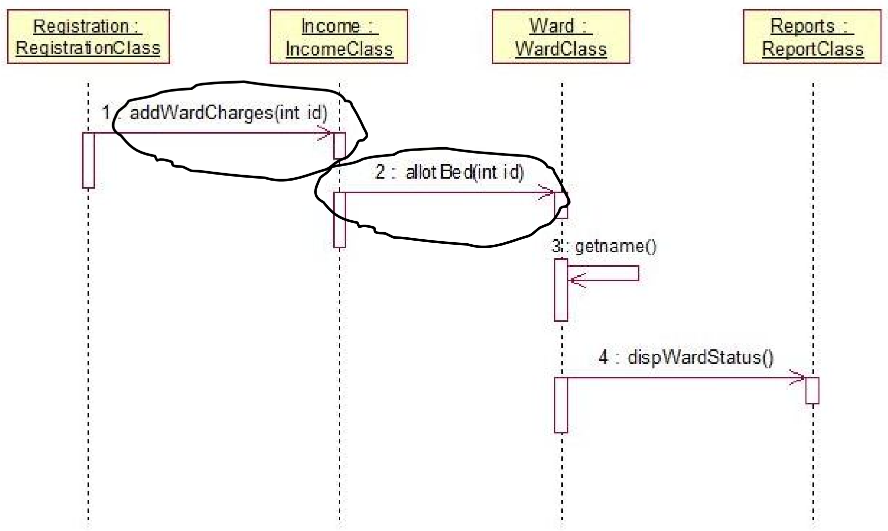


Fig. 6.5 Sequence diagram for Bed Allotment event

Message: addWardCharges(int id)

Actor: Income

Non-Actor: Registration

Here communication from non-actor to actor, so it can be identified as external output (EO).

DET: 1, only one argument candidate message have.

FTR: 1, only one entity class is there.

Message: allotBed(int id)

Actor: Income

Non-Actor: Ward

Here communication from actor to non-actor, so it can be identified as external input (EI).

DET: 1, only one argument candidate message have.

FTR: 1, only one entity class is there.

Undergo Operation:

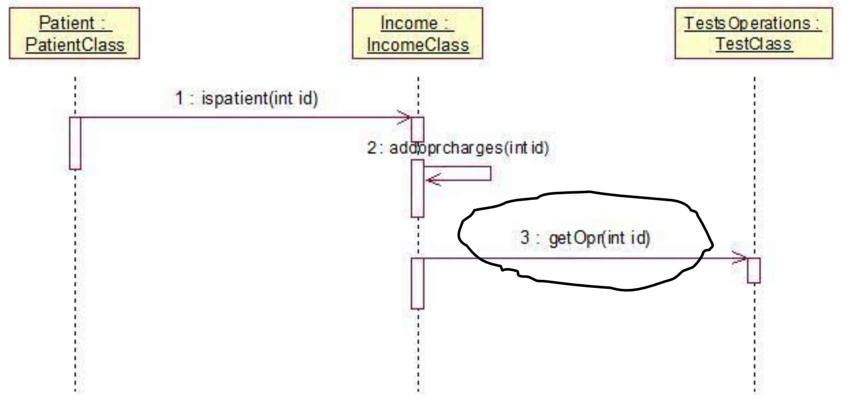


Fig. 6.6 Sequence diagram for Undergo Operation event

Message: getOpr(int id)
Actor: Income

Non-Actor: TestOperation

Here communication from actor to non-actor, so it can be identified as external input (EI).

DET: 1, only one argument candidate message have.

FTR: 1, only one entity class is there.

Login:

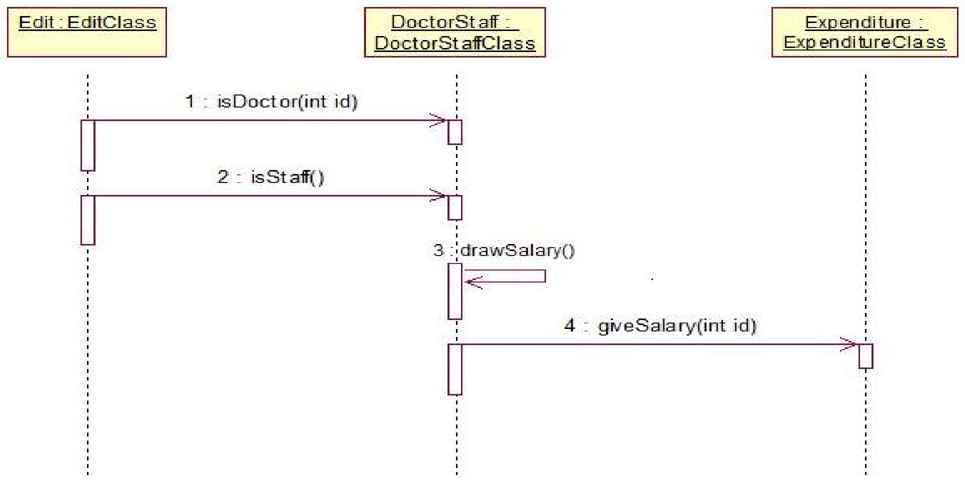


Fig. 6.7 Sequence diagram for Login event

Draw salary:

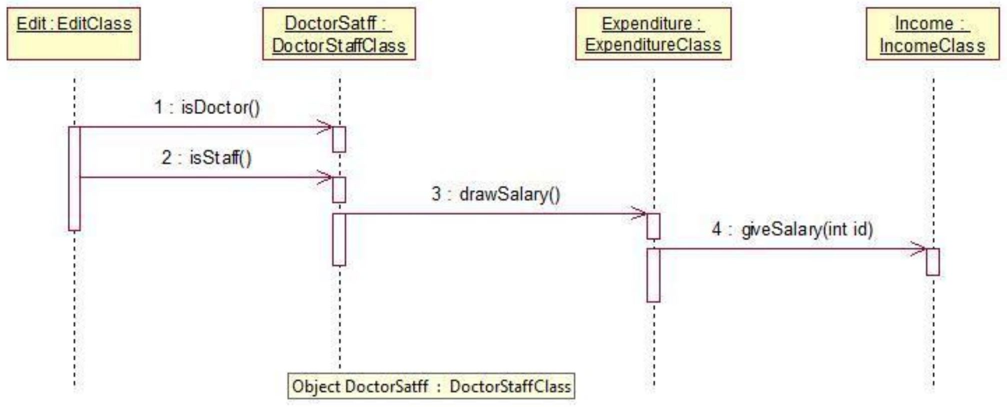


Fig. 6.8 Sequence diagram for Draw salary event

Login and Draw salary events have no candidate message for transactional function types. Here some messages are having no arguments so they should be discarded and rest are between actor to actor or non-actor to non-actor so they cannot be taken as a candidates for transactional function types.

Delete Doctor/staff:

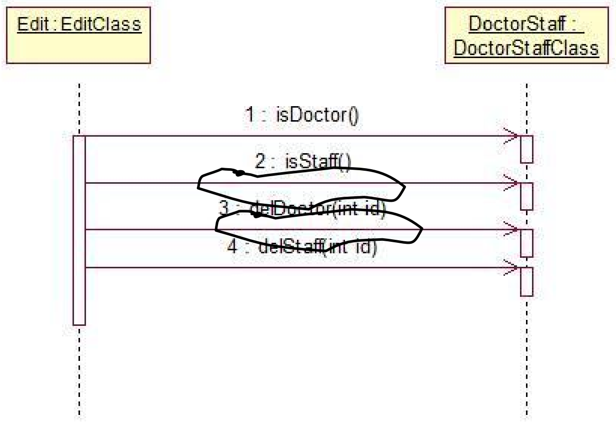


Fig. 6.9 Sequence diagram for Delete Doctor/staff event

Message: delDoctor(int id), delStaff(int id)

Actor: DoctorStaff

Non-Actor: Edit

Here communication from non-actor to actor, so it can be identified as external output (EO).

DET: 1,1, only one argument candidate message have.

FTR: 1,1, only one entity class is there.

Edit Doctor/staff:

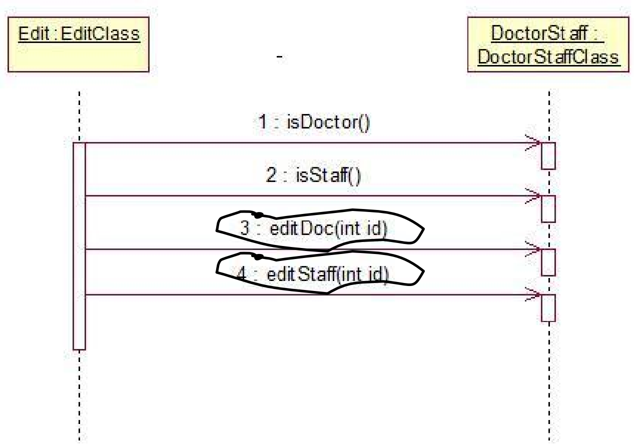


Fig. 6.10 Sequence diagram for Edit Doctor/staff event

Message: editDoctor(int id), editStaff(int id)

Actor: DoctorStaff

Non-Actor: Edit

Here communication from non-actor to actor, so it can be identified as external output (EO).

DET: 1,1, only one argument candidate message have.

FTR: 1,1, only one entity class is there.

Prescribed Test:

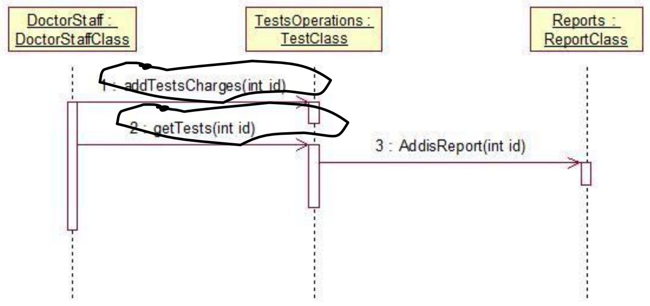

DoctorStaff : DoctorStaffClass
TestsOperations : TestClass
Reports : ReportClass
1 : addTestsCharges(int id)
2 : getTests(int id)
3 : AddisReport(int id)

Fig. 6.11 Sequence diagram for Prescribed test event

Message: getTestCharges(int id),getTest(int id)

Actor: DoctorStaff

Non-Actor: TestOperation

Here communication from actor to non-actor, so it can be identified as external input.

DET: 1,1, only one argument candidate message have.

FTR: 1,1, only one entity class is there.

Messages	Transaction function	Complexity
addApptCharges(int id)	EO (L)	1 DET, 1FTR
addApptCharges(int id)	EI (L)	1 DET, 1FTR
addTestCharges(int id)	EI (L)	1 DET, 1FTR
addTestCharges(int id)	EI (L)	1 DET, 1FTR
addWardCharges(int id)	EO (L)	1 DET, 1FTR
allotbed(int id)	EI (L)	1 DET, 1FTR
getOper(int id)	EI(L)	1 DET, 1FTR
delDoctor(int id)	EO (L)	1 DET, 1FTR
delStaff(int id)	EO (L)	1 DET, 1FTR
editDoc(int id)	EO (L)	1 DET, 1FTR
editStaff(int id)	EO (L)	1 DET, 1FTR
getTest(int id)	EI (L)	1 DET, 1FTR

Table 6.2 Transactional Function types

Total Internal Logical Files: 12 with low complexity

Total External Input: 6 with low complexity

Total External Output: 6 with low complexity

Total UFP: 12*7 + 6*5 + 6*5 = 144

Total SLOC: 144*53 = 7362 (UFP to SLOC conversion ratio for JAVA) see Table 3.1

Size	144 UFP
SLOC	7362
Effort	22.91 PM
Development time	3.67 Month
Staff	2.5
Productivity (SLOC/PM)	321.2

Table 6.3 General Software Metrics

7. CONCLUSION

In this thesis, we have applied detailed function point analysis rules for design specification developed based on the UML. Our tool estimates object oriented software metrics in early life cycle phase, based on information of software in early design we have applied Harput rules and some guidelines to estimate size metrics then we have applied COCOMO II techniques to calculate rest of the software metrics. Tool architecture and design is only for object oriented software. We have used Harput transformation rules and Uemura approach to automate function point estimation but still fully automatic model transformation still seems to be out of reach. Compared with FPA, the estimation error range will decreased as we are accounting for the complexities of generalization, aggregation and association which are not considered in traditional function point measurement techniques. This approach easily estimate the effort for a software development project based on its size using FPA.

REFERENCES:

[1] A.J. Albrecht, "*Measuring Application Development Productivity*", Proc. IBM Applications Development Symp., Monterey, Calif. ,Oct 14-17, 1979.

[2]Harput V,Kaindl H,Kramer S.,"Extending Function Point Analysis to Object-Oriented Requirements Specifications ",procceding on 11th IEEE International Software Metrics Symposium (METRICS 2005).

[3] D.J Ram, S.V.G.K Raju," *Object Oriented Design Function Points*", -7695-0825-1/00 2000 IEEE.

[4] Kusumoto S.,Imagawa K.,Inoue K.,Morimoto S.,"Function Point Measurement from Java Program" Proceedings of the ICSE'2002,florida ,USA.

[5] International Function Point User Group (IFPUG), Function Point Counting Practices Manual, Release 4.0, IFPUG, Westerville, Ohio, April 1990.

[6]Symons,C.:"*Function-Point Analysis: Difficulties and Improvements*." *IEEE Transactions on Software Engineering*, Vol. 14, Nr. 1, January 1988, pp. 2-11.

[7]Common software Measurement International Consortium, COSMIC–FFP version 2.0(2000).http:// www.cosmicon.com/

[8] Poensgen, B. and Bock, B. *Function-Point An]alyse*, dpunkt.verlag, Heidelberg, 2005.

[9] Sneed, H. "*Impact Analysis of Maintenance Tasks for A Distributed Object-Oriented System*" *Proceedings of 17th International Conference on Software Maintenance*(ICSM 2001: Florence, Italy, November 7-9, 2001) IEEE CS Press, pp. 180-189.

[10] Sneed H.M, Huang S," *Sizing Maintenance Tasks for Web Applications*", procceding on 11th European Conference on Software Maintenance and Reengineering (CSMR'07) 2007.

[11] G. Caldiera, *G.* Antoniol, R. Fiutem, and C. Lokan. "*Definition and experimental evaluation of function points for object-oriented systems*"In *Proc. of the 5'h InternationalSymposium on Software Metrics,* pages 167-178, November 1998.

[12] Sneed, H.M.: "Estimating the Development Costs of Object-Oriented Software." *Proceedings of 7th European Software Control and Metrics Conference,* Wilmslow,UK, 1996, p. 135.

[13] Cowderoy, A.J.C. "Size and Quality Measures for Multimedia and Web-site Production." *Proceedings of the 14th International Cocomo Forum*, 1999.

[14] Reifer, D.: "Web Development: Estimating Quick-to-Market Software." *IEEE Softeware,* November/December 2000.

[15] M. Sadiq., Shabbir Ahmed, "*Computation of Function Point of a Software on the basis of average complexity*", International Conference on Advanced Computing and Communication Technologies,(ICACCT 07),Panipat, Haryana, India, 2007.

[36] Gupta D.,Kaushal S.,Sadiq M.,"software Estimation tool based on three layer model for software engineering metrics" , ICMIT 2008.

[17] Prof. Ellis, COCOMO II.2000.0 Horowitz University of southern California, Center for software engineering, 1995

[18] Edilson J. D. Candido,Rosely Sanches, "Estimating the size of web applications by using a simplified function point Method",IEEE 2004.

[19] G. Cantone, D. Pace, and G. Calavaro. Applying function point to unified modeling language: Conversion model and pilot study. In Proceedings of the 10th International Symposium on Software Metrics (METRICS'04), pages 280–291. IEEE Computer Society, 2004.

[20] Fetcke, T., Abran, A. and Nguyen, T., "Mapping the OOJacobson Approach into Function Point Analysis", IEEE Proceedings of TOOLS-23'97, 1997.

[21] T. Uemura, S. Kusumoto, and K. Inoue. Function Point Measurement Tool for UML Design Specification. In Proceedings of the Sixth IEEE International Symposium on Software Metrics, 1998.