**Aim:** Finalise the problem statement and conduct the requirement gathering

Requirements identification is the first step of any software development project. Until the requirements of a client have been clearly identified, and verified, no other task (design, coding, testing) could begin. Usually business analysts having domain knowledge on the subject matter discuss with clients and decide what features are to be implemented.

In this experiment we will learn how to identify functional and non-functional requirements from a given problem statement. Functional and non-functional requirements are the primary components of a Software Requirements Specification.

Requirements

Somerville defines "requirement" as a specification of what should be implemented. Requirements specify how the target system should behave. It specifies what to do, but not how to do. Requirements engineering refers to the process of understanding what a customer expects from the system to be developed, and to document them in a standard and easily readable and understandable format. This documentation will serve as reference for the subsequent design, implementation and verification of the system.

It is necessary and important that before we start planning, design and implementation of the software system for our client, we are clear about it's requirements. If we don't have a clear vision of what is to be developed and what all features are expected, there would be serious problems, and customer dissatisfaction as well.

Characteristics of Requirements

Requirements gathered for any new system to be developed should exhibit the following three properties:

Unambiguity: There should not be any ambiguity what a system to be developed should do. For example, consider you are developing a web application for your client. The client requires that enough number of people should be able to access the application simultaneously. What's the "enough number of people"? That could mean 10 to you, but, perhaps, 100 to the client. There's an ambiguity.

Consistency: To illustrate this, consider the automation of a nuclear plant. Suppose one of the clients say that if the radiation level inside the plant exceeds R1, all reactors should be shut down. However, another person from the client side suggests that the threshold radiation level should be R2. Thus, there is an inconsistency between the two end users regarding what they consider as threshold level of radiation.

Completeness: A particular requirement for a system should specify what the system should do and also what it should not. For example, consider a software to be developed for ATM. If a customer enters an amount greater than the maximum permissible withdrawal amount, the ATM should display an error message, and it should not dispense any cash.

Categorization of Requirements

Based on the target audience or subject matter, requirements can be classified into different types, as stated below:

User requirements: They are written in natural language so that both customers can verify their requirements have been correctly identified

System requirements: They are written involving technical terms and/or specifications, and are meant for the development or testing teams

Requirements can be classified into two groups based on what they describe:

Functional requirements (FRs): These describe the functionality of a system -- how a system should react to a particular set of inputs and what should be the corresponding output.

Non-functional requirements (NFRs): They are not directly related what functionalities are expected from the system. However, NFRs could typically define how the system should behave under certain situations. For example, a NFR could say that the system should work with 128MB RAM. Under such condition, a NFR could be more critical than a FR.

Non-functional requirements could be further classified into different types like:

Product requirements: For example, a specification that the web application should use only plain HTML, and no frames

Performance requirements: For example, the system should remain available 24x7

Organizational requirements: The development process should comply to SEI CMM level 4

Functional Requirements: Identifying Functional Requirements

Given a problem statement, the functional requirements could be identified by focusing on the following points:

Identify the high level functional requirements simply from the conceptual understanding of the problem. For example, a Library Management System, apart from anything else, should be able to issue and return books.

Identify the cases where an end user gets some meaningful work done by using the system. For example, in a digital library a user might use the "Search Book" functionality to obtain information about the books of his interest.

If we consider the system as a black box, there would be some inputs to it, and some output in return. This black box defines the functionalities of the system. For example, to search for a book, user gives title of the book as input and get the book details and location as the output.

Any high level requirement identified could have different sub-requirements. For example, "Issue Book" module could behave differently for different class of users, or for a particular user who has issued the book thrice consecutively.

1. Conduct the literature survey
2. Shortlist at least 5 domain of project and possible nature of problem.
3. Hotel management system
4. Employee management system
5. Medical store management system – Apothecary
6. Department store management system
7. Ticket Reservation system
8. Identify the scope of the problems selected
9. Hotel management – login module, room service orders, checkout and bill module
10. Employee Management – login/register, list of employees, option to modify the details, to delete them, to search them, to sort the table by chosen parameter
11. Medical store management- suppliers and supplier information, customers and customer information, employee and employee information, inventory, bill generation, search item, place orders, treasurer.
12. Department store - suppliers and supplier information, customers and customer information, employee and employee information, inventory, bill generation, search item, place orders, treasurer.
13. Ticket reservation – number of busses available, timing, destinations and routes, reserve seats/tickets as per busses, bill generation, driver information/registration/contact, search/sort
14. Identify the end user of the solution
15. Hotel management system - staff
16. Employee management system - managers
17. Apothecary system – store owner/ employee
18. Department store management system – store owner/ employee
19. Ticket Reservation system – manager/ employee
20. Identify the functional requirements of the project
21. Supplier information storage
22. Customer information storage
23. Employee information storage
24. Inventory tracking and management
25. Searching an item in inventory
26. Bill generating system
27. Send order requests
28. Treasurer system
29. Login/Registration module
30. Identify the non-functional requirements of the project
31. Security
32. Availability
33. Reliability
34. Encryption
35. Identify the feasibility of the project

The medical store management system can be done as a desktop application because there is no real need for it to be a web application or an app to be circulated. Java is a language I am familiar with and by learning to use netbeans or something similar it can be executed. The cost will be minimal since no sophisticated software or hardware needs to be bought or incorporated.

1. Finalize one problem out of five selected

Medical store management system called Apothecary.

1. Frame the final problem statement.

Since medicine is an important commodity and can play an important role in the lives of people, its management/supply/storage should be done efficiently. This desktop application is aimed to make keeping track of the inventory and also its management.

1. List down the list of user requirements, system requirements.

User requirements: end device, basic understanding of computers and handling,

System requirements: 4GB RAM, Internal storage

1. Identify the ambiguities, inconsistencies, incompleteness from the requirements gathered.

Improper/ inaccurate data being submitted, outdated data being filled, not checking in with the system when selling something

1. List down the development plan for the selected problem statements

Development plan would involve learning the process to make a desktop application and developing the modules and then integrating them one at a time so as to minimize the number of bugs as well as guarantee proper functioning through testing.

**Aim:** Categorize projects using COCOMO, and estimate effort and development time required for a project

**Project Estimation Techniques**

A software project is not just about writing a few hundred lines of source code to achieve a particular objective. The scope of a software project is comparatively quite large, and such a project could take several years to complete. However, the phrase "quite large" could only give some (possibly vague) qualitative information. As in any other science and engineering discipline, one would be interested to measure how complex a project is. One of the major activities of the project planning phase, therefore, is to estimate various project parameters in order to take proper decisions. Some important project parameters that are estimated include:

**Project size:** What would be the size of the code written say, in number of lines, files, modules?

**Cost:** How much would it cost to develop a software? A software may be just pieces of code, but one has to pay to the managers, developers, and other project personnel.

**Duration:** How long would it be before the software is delivered to the clients?

**Effort:** How much effort from the team members would be required to create the software?

**COCOMO**

COCOMO (Constructive Cost Model) was proposed by Boehm. According to him, there could be three categories of software projects: organic, semidetached, and embedded. The classification is done considering the characteristics of the software, the development team and environment. These product classes typically correspond to application, utility and system programs, respectively. Data processing programs could be considered as application programs. Compilers, linkers, are examples of utility programs. Operating systems, real-time system programs are examples of system programs. One could easily apprehend that it would take much more time and effort to develop an OS than an attendance management system.

The concept of organic, semidetached, and embedded systems are described below.

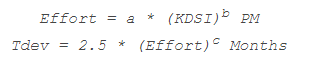
Organic: A development project is said to be of organic type, if the project deals with developing a well understood application. The development team is small. The team members have prior experience in working with similar types of projects

Semidetached: A development project can be categorized as semidetached type, if the team consists of some experienced as well as inexperienced staff. Team members may have some experience on the type of system to be developed

Embedded: Embedded type of development project are those, which aims to develop a software strongly related to machine hardware. Team size is usually large.

Boehm suggested that estimation of project parameters should be done through three stages: Basic COCOMO, Intermediate COCOMO, and Complete COCOMO.

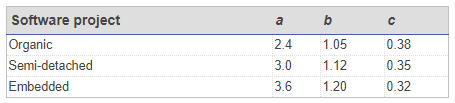
**Basic COCOMO Model**

The basic COCOMO model helps to obtain a rough estimate of the project parameters. It estimates effort and time required for development in the following way:   


Where,

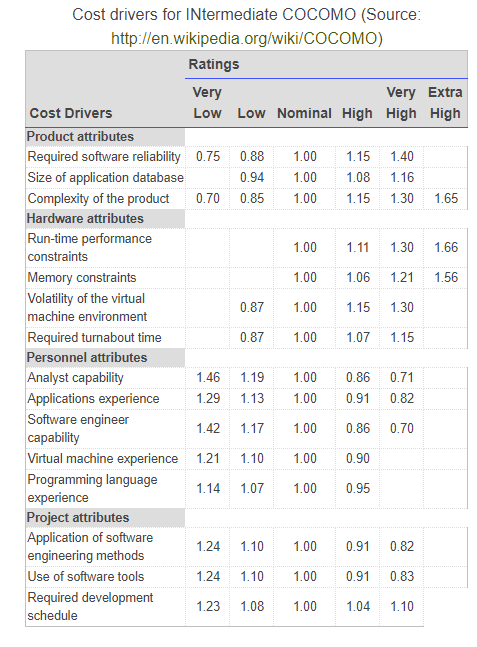
* KDSI is the estimated size of the software expressed in Kilo Delivered Source Instructions
* a, b, c are constants determined by the category of software project
* Effort denotes the total effort required for the software development, expressed in person months (PMs)
* Tdev denotes the estimated time required to develop the software (expressed in months)

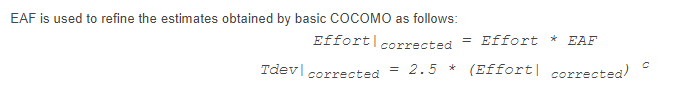
The value of the constants a, b, c are given below:



**Intermediate COCOMO Model**

The basic COCOMO model considers that effort and development time depends only on the size of the software. However, in real life there are many other project parameters that influence the development process. The intermediate COCOMO take those other factors into consideration by defining a set of 15 cost drivers (multipliers) as shown in the table below . Thus, any project that makes use of modern programming practices would have lower estimates in terms of effort and cost. Each of the 15 such attributes can be rated on a six-point scale ranging from "very low" to "extra high" in their relative order of importance. Each attribute has an effort multiplier fixed as per the rating. The product of effort multipliers of all the 15 attributes gives the **Effort Adjustment Factor (EAF).**





**Complete COCOMO Model**

Both the basic and intermediate COCOMO models consider a software to be a single homogeneous entity -- an assumption, which is rarely true. In fact, many real life applications are made up of several smaller sub-systems. (One might not even develop all the sub-systems -- just use the available services). The complete COCOMO model takes these factors into account to provide a far more accurate estimate of project metrics.

To illustrate this, consider a very popular distributed application: the ticket booking system of the Indian Railways. There are computerized ticket counters in most of the railway stations of our country. Tickets can be booked / cancelled from any such counter. Reservations for future tickets, cancellation of reserved tickets could also be performed.

On a high level, the ticket booking system has three main components:

* Database
* Graphical User Interface (GUI)
* Networking facilities
* Among these, development of the GUI is considered as an organic project type; the database module could be considered as a semi-detached software. The networking module can be considered as an embedded software. To obtain a realistic cost, one should estimate the costs for each component separately, and then add it up.

1. **Identify the type of Project (Basic or Intermediate)**
2. **Compute Time to develop, Effort required for Complete COCOMO model (Considering one basic module, one semidetached module and one embedded module).**

The project uses an intermediate model.

It can be divided in 3 modules:

**logic module:**

EAF: 1.46

KLOC: 0.2

Effort: 0.59

Corrected effort: 0.8614

Corrected Tdev: 2.362

**GUI/Front end:**

EAF: 1.04

KLOC: 0.4

Effort: 1.075

Corrected effort: 1.118

Corrected Tdev: 2.6

**Back end/Database:**

EAF:1.172

KLOC: 0.35

Effort: 1.2187

Corrected effort: 1.4283

Corrected Tdev: 2.8020

|  |  |  |  |
| --- | --- | --- | --- |
| Cost Drivers | Logic | Front | Back |
| **Product attributes** |  |  |  |
| Required software reliability | 1.15 | 1 | 1.40 |
| Size of application database | 1.3 | 1 | 1.3 |
| Complexity of the product | 0.85 | 0.85 | 1.15 |
| **Hardware attributes** |  |  |  |
| Run-time performance constraints | 1 | 1 | 1 |
| Memory constraints | 1 | 1.11 | 1.11 |
| Volatility of the virtual machine environment | 0.87 | 0.87 | 0.87 |
| Required turnabout time | 1 | 0.87 | 1 |
| **Personnel attributes** |  |  |  |
| Analyst capability | 0.86 | 1 | 0.71 |
| Applications experience | 1 | 1 | 1 |
| Software engineer capability | 1 | 1.17 | 0.86 |
| Virtual machine experience | 1 | 1 | 1 |
| Programming language experience | 1 | 1.14 | 0.95 |
| **Project attributes** |  |  |  |
| Application of software engineering methods | 1.24 | 1 | 1 |
| Use of software tools | 1.24 | 1.10 | 1 |
| Required development schedule | 1 | 1 | 1 |

**Aim:** Select appropriate Generic Process Model/ Evolutionary process Model/ Agile Model for your project.

A software process is a collection of various activities.  
  
**There are five generic process framework activities:**  
  
**1. Communication:**   
The software development starts with the communication between customer and developer.  
  
**2. Planning:**   
It consists of complete estimation, scheduling for project development and tracking.  
  
**3. Modelling:**

Modeling consists of complete requirement analysis and the design of the project like algorithm, flowchart etc. The algorithm is the step-by-step solution of the problem and the flow chart shows a complete flow diagram of a program.

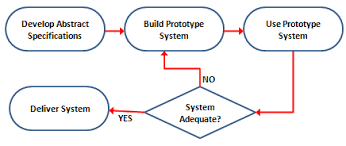
**4. Construction and Testing:**

Construction consists of code generation and the testing part. Coding part implements the design details using an appropriate programming language. Testing is to check whether the flow of coding is correct or not. Testing also check that the program provides desired output.

**5. Deployment:**

Deployment step consists of delivering the product to the customer and take feedback from them. If the customer wants some corrections or demands for the additional capabilities, then the change is required for improvement in the quality of the software.

1. **Identify the suitable process model through literature survey**
2. **Justify, why the model selected in Task 1 is most suitable for your project. Why other models are not suitable for your project.**

****

**This is the general process flow of an Evolutionary prototype model.**

Prototyping is a process which can be used as a standalone model or coupled with any other model. It assists in gaining a better understanding of the products when the initial requirements are not completely certain. The client communicates with the developer and tells him his needs and requirements. Then a quick plan is made ie a prototyping iteration and modeling occurs. Then the construction happens and the software is deployed. The stakeholders evaluate the prototype and give feedback which is used in the next iteration. This process goes on until all needs of the stakeholders are satisfied.

For apothecary, which will be a desktop application, the best suited model is the evolutionary prototype model.

The scale of this desktop application is large and the project is relatively simple and rudimentary. While scope of further development is present, the scope is small. The increments in the project will not be too large for this model to become cumbersome and the increment cycles will not be too many.

Other models:

Waterfall: this model is not suitable because here the scope for innovation is less and incorporating feedback of the customers is not encouraged, once deployed no changes are made to the software.

Evolutionary Spiral: this model is mainly used for risk assessment and reduction. But in my project the number of changes are less and extensive risk assessment is unnecessary.

Evolutionary concurrent: The project size is not big and neither are the time constraints too strict, so concurrency is not required and hence adds complexity rather than helping development.

Agile: While agile is also suitable, the project isn’t big enough or complex enough to require agile modeling. Communication doesn’t need to be prioritized since the requirements are straight forward. It also will not need too many iterations or modifications.

**Aim:** Design level-1, Level-2 and Level-3 **Data flow diagram**

**Dataflow Diagram:**

DFD provides the functional overview of a system. The graphical representation easily overcomes any gap between ’user and system analyst’ and ‘analyst and system designer’ in understanding a system. Starting from an overview of the system it explores detailed design of a system through a hierarchy. DFD shows the external entities from which data flows into the process and also the other flows of data within a system. It also includes the transformations of data flow by the process and the data stores to read or write a data.

A DFD shows what kind of information will be input to and output from the system, where the data will come from and go to, and where the data will be stored. It does not show information about the timing of processes, or information about whether processes will operate in sequence or in parallel.

The highest level, called the context diagram, is only an overview. More detail is typically needed for system analysts. We add detail to a DFD by creating “levels”. The first level added after the context diagram is called level “0”. Each new level breaks apart one process and “decomposes” the single process into a new, more detailed DFD. A complete DFD can have many (up to 6 or 7) levels depending on the complexity of system. Breaking the DFD into levels is referred to as “Decomposition”.

**Explanation of Symbols used in DFD**

**Process:** Processes are represented by circle. The name of the process is written into the circle. The name of the process is usually given in such a way that represents the functionality of the process. More detailed functionalities can be shown in the next Level if it is required. Usually it is better to keep the number of processes less than 7 . If we see that the number of processes becomes more than 7 then we should combine some the processes to a single one to reduce the number of processes and further decompose it to the next level.

**External entity:** External entities are only appear in context diagram. External entities are represented by a rectangle and the name of the external entity is written into the shape. These send data to be processed and again receive the processed data.

**Data store:** Data stares are represented by a left-right open rectangle. Name of the data store is written in between two horizontal lines of the open rectangle. Data stores are used as repositories from which data can be flown in or flown out to or from a process.

**Data flow:** Data flows are shown as a directed edge between two components of a Data Flow Diagram. Data can flow from external entity to process, data store to process, in between two processes and vice-versa.

**Context diagram and leveling DFD:**

We start with a broad overview of a system represented in level 0 diagram. It is known as context diagram of the system. The entire system is shown as single process and also the interactions of external entities with the system are represented in context diagram.

Further we split the process in next levels into several numbers of processes to represent the detailed functionalities performed by the system. Data stores may appear in higher level DFDs.

In a DFD with many levels it’s easy to forget which level you are on. That’s why each level has different numbering for the processes on the diagram. The ‘level’ corresponds to the number of decimal places required to define a process in it. Here’s how it works:

Context Diagram Process labeled “0”

Level 1 Processes labeled 1.0, 2.0, 3.0, .

Level 2 Processes labeled 1.1, 1.2, 1.3, .

Level 3 Processes labeled 1.1.1, 1.1.2,...

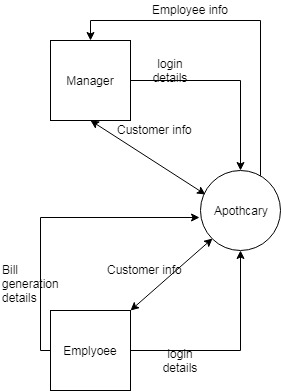
**Task to be completed:**

1. **Design Level 0, Level 1 and Level2 for your project.**

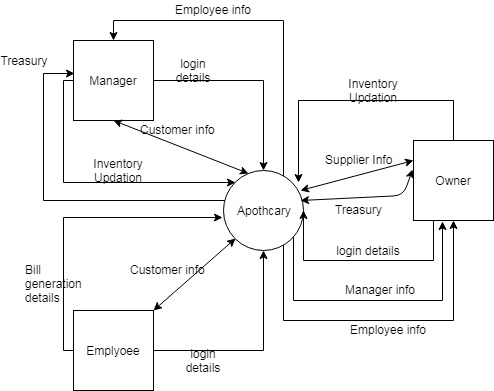
**(Use appropriate symbols for Entity, process, dataflow and database )**

Note :

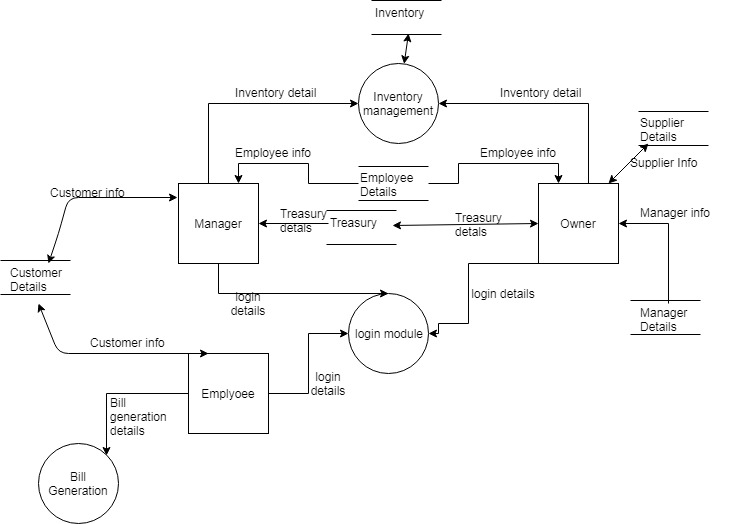
1. External entities only appear in context diagram i.e, only at level 0.
2. Keep number of processes at each level less than 7.
3. Data flow is not possible in between two external entities and in between two data stores.
4. Data cannot flow from an External entity to a data store and vice-versa.

****

**Level 0**

****

**Level 1**

****

**Level 2**

**Aim:** Design **State chart diagram** using case study’s Control specifications. Make use of Star UML software for design

**State Chart Diagram:**

Capturing the dynamic view of a system is very important for a developer to develop the logic for a system. State chart diagrams are popular UML diagram to visualize the dynamic behavior of an information system.

In this experiment, we will learn about the different components state chart diagram and how these can be used to represent the dynamic nature of an information system.

In case of Object Oriented Analysis and Design, a system is often abstracted by one or more classes with some well defined behaviour and states. A statechart diagram is a pictorial representation of such a system, with all it's states, and different events that lead transition from one state to another.

To illustrate this, consider a computer. Some possible states that it could have are: running, shutdown, hibernate. A transition from running state to shutdown state occur when user presses the "Power off" switch, or clicks on the "Shut down" button as displayed by the OS. Here, clicking on the shutdown button, or pressing the power off switch act as external events causing the transition.

Statechart diagrams are normally drawn to model the behaviour of a complex system. For simple systems this is optional.

A state is any "distinct" stage that an object (system) passes through in it's lifetime. An object remains in a given state for finite time until "something" happens, which makes it to move to another state.  All such states can be broadly categorized into following three types:

* **Initial**: The state in which an object remain when created
* **Final**: The state from which an object do not move to any other state [optional]
* **Intermediate**: Any state, which is neither initial, nor final

As shown in figure, an initial state is represented by a circle filled with black. An intermediate state is depicted by a rectangle with rounded corners.

State

A final state is represented by a unfilled circle with an inner

black-filled circle. Figure Representation of initial, intermediate, and final states of a statechart diagram Intermediate states usually have two compartments, separated by a horizontal line, called the name compartment and internal transitions compartment. They are described below:

* **Name compartment**: Contains the name of the state, which is a short, simple, descriptive string
* **Internal transitions compartment**: Contains a list of internal activities performed as long as the system is in this state

The internal activities are indicated using the following syntax: action-label / action-expression. Action labels could be any condition indicator. There are, however, four special action labels:

* **Entry**: Indicates activity performed when the system enter this state
* **Exit**: Indicates activity performed when the system exits this state
* **Do**: indicate any activity that is performed while the system remain in this state or until the action expression results in a completed computation
* **Include**: Indicates invocation of a sub-machine

Any other action label identify the event (internal transition) as a result of which the corresponding action is triggered. Internal transition is almost similar to self transition, except that the former doesn't result in execution of entry and exit actions. That is, system doesn't exit or re-enter that state. Figure shows the syntax for representing a typical intermediate state.

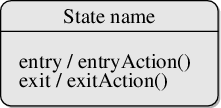


Figure: A typical state in a statechart diagram States could again be either simple or composite (a state congaing other states). Here, however, we will deal only with simple states.

### Transition:

Transition is movement from one state to another state in response to an external stimulus (or any internal event). A transition is represented by a solid arrow from the current state to the next state. It is labeled by: event [guard-condition]/[action-expression], where

* **Event** is the what is causing the concerned transition (mandatory) -- Written in past tense
* **Guard-condition** is (are) precondition(s), which must be true for the transition to happen [optional]
* **Action-expression** indicate action(s) to be performed as a result of the transition [optional]

It may be noted that if a transition is triggered with one or more guard-condition(s), which evaluate to false, the system will continue to stay in the present state. Also, not all transitions do result in a state change. For example, if a queue is full, any further attempt to append will fail until the delete method is invoked at least once. Thus, state of the queue doesn't change in this duration.

### Action

As mentioned in, actions represents behaviour of the system. While the system is performing any action for the current event, it doesn't accept or process any new event. The order in which different actions are executed, is given below:

1. Exit actions of the present state
2. Actions specified for the transition
3. Entry actions of the next state

Figure-03 shows a typical statechart diagram with all it's syntaxes.

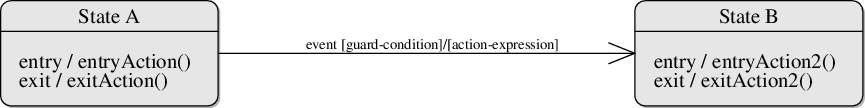


Figure: A statechart diagram showing transition from state A to B

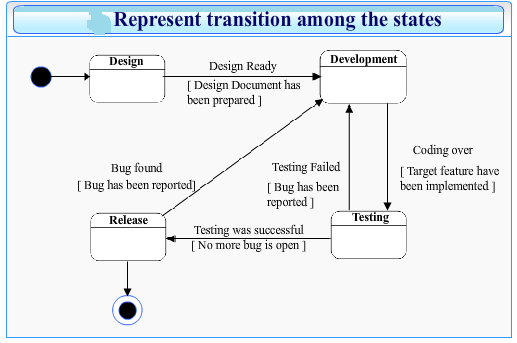
## Guidelines for drawing State chart Diagrams

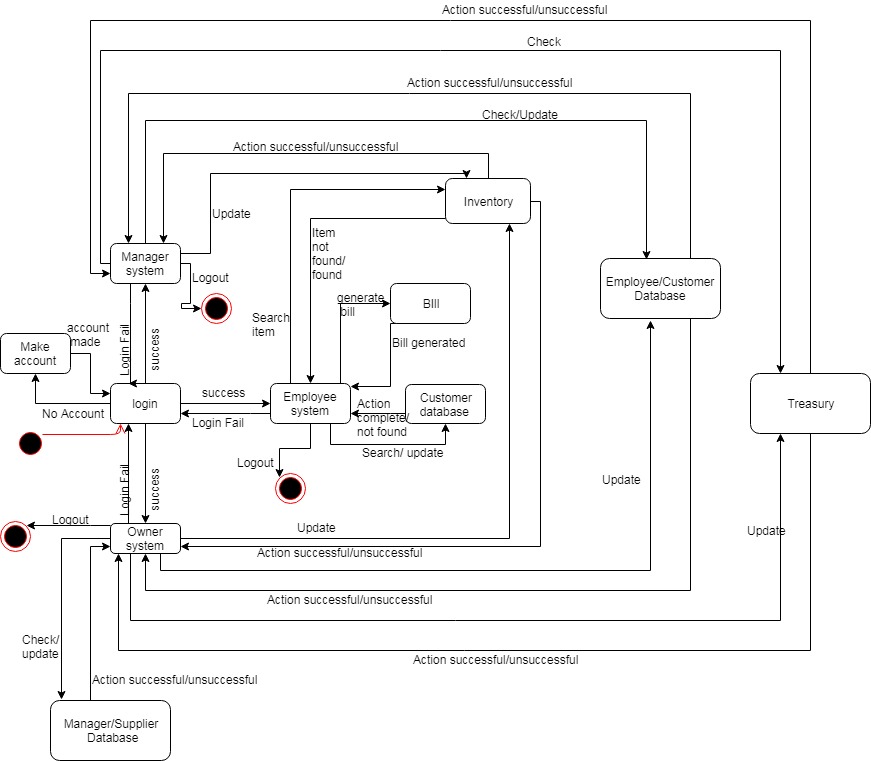
Following steps could be followed, as suggested to draw a statechart diagram:

* For the system to developed, identify the distinct states that it passes through
* Identify the events (and any precondition) that cause the state transitions. Often these would be the methods of a class as identified in a class diagram.
* Identify what activities are performed while the system remains in a given state
* List all sensors that are "read" by the software.
* List all interrupt conditions.
* List all "switches" that are actuated by the operator.
* List all data conditions.
* Recalling the noun-verb parse that was applied to the software statement of scope, review all "control items" as possible CSPEC inputs/outputs.
* Describe the behavior of a system by identifying its states; identify how each state is reach and defines the transitions between states.

### Drawing statechart diagram from a problem statement

Consider the development of a medium-scale software in, say, C++. In a software development life cycle typically four phased could be observed. It begins with planning and design, after which the development work starts. Once the required features have been implemented, the software is tested. If testing is successful, the software is released. If there is even a single failure, that is to be fixed, and tested again. However, certain bugs might become apparent only after someone uses the software in real scenario. As such, the software again enters into development phase, followed by testing, and then another release.

****

****

**Aim:** Design the appropriate **user interface diagram** for your project using three golden rules. Coding for your project according to the designs analyzed in EXP.4, 5

**User Interface design:**

While designing user interface, keep the three golden rules in mind.

* Place the user in control
* Reduce the user’s memory load
* Make the interface consistent

The goal of user interface design is to make the user's interaction as simple and efficient as possible, in terms of accomplishing user goals, what is often called as user-centered design.

**Coding:**

The coding depends on individual’s project. Any programming language can be used according to student’s interest.

**Coding** : The objective of the coding phase is to transform the design of a system into code in a high-level language and then to unit test this code. Good software development organizations normally require their programmers to adhere to some well-defined and standard style of coding called coding standards.

Coding Standards- A coding standard gives a uniform appearance to the codes written by different engineers. It enhances code understanding. It encourages good programming practices.

**Coding Standards and Guideline:**

Limiting the use of global data type. Contents of the headers preceding codes for different modules naming conventions for global variables, local variables, and constant identifiers. Error return conventions and exception handling mechanisms Representative Coding Standards. Do not use a coding style that is too clever or too difficult to understand. Avoid obscure side effects .Do not use an identifier for multiple purposes. The code should be well-documented.

**Code Review:**

Code review for a model is carried out after the module is successfully compiled and all the syntax errors have been eliminated. Normally, two types of reviews are carried out on the code of a module.

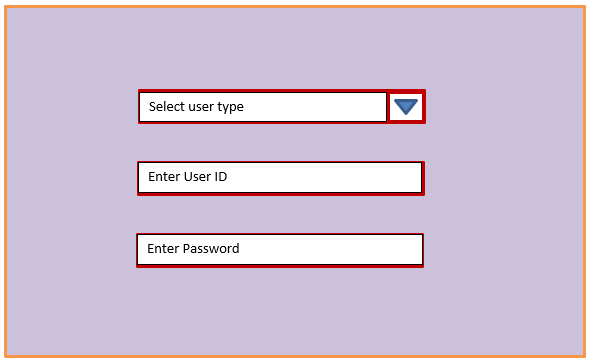
**Code Walk Through:** To discover the algorithm and logical errors in the code.

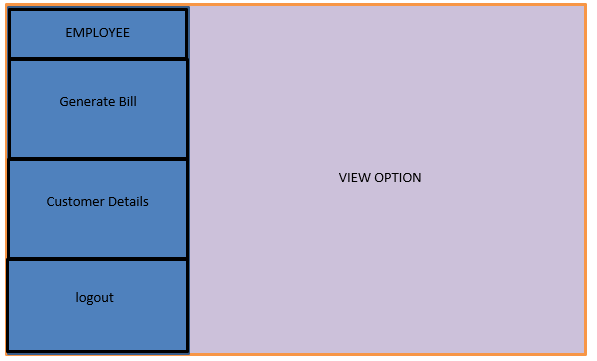
**Code Inspection:** The aim of code inspection is to discover some common types of errors caused due to oversight and improper programming.

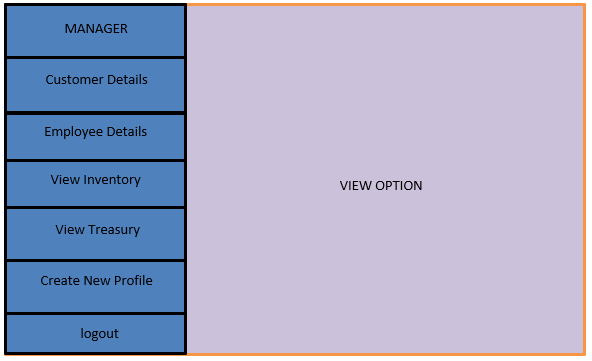
Software Documentation: Good documents are very useful and serves the following purposes. Good documents enhance understandability and maintainability of a software product. It helps the users in effectively using the system. Helps in effectively handling the manpower turnover problem. Helps the manager in effectively tracking the progress of the project.

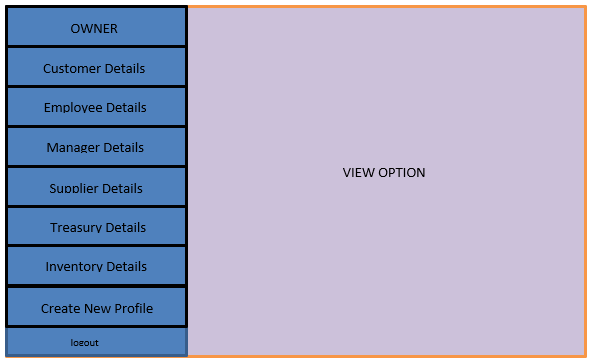
Software Documentation classified into the following: Internal documentation: These are provided in the source code itself

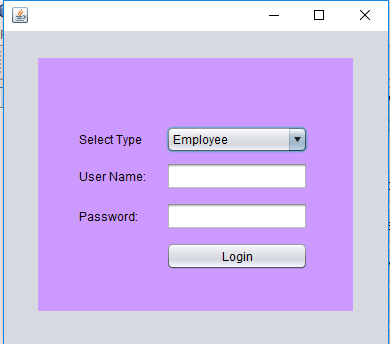
External documentation: These are the supporting documents that usually accompany a software product



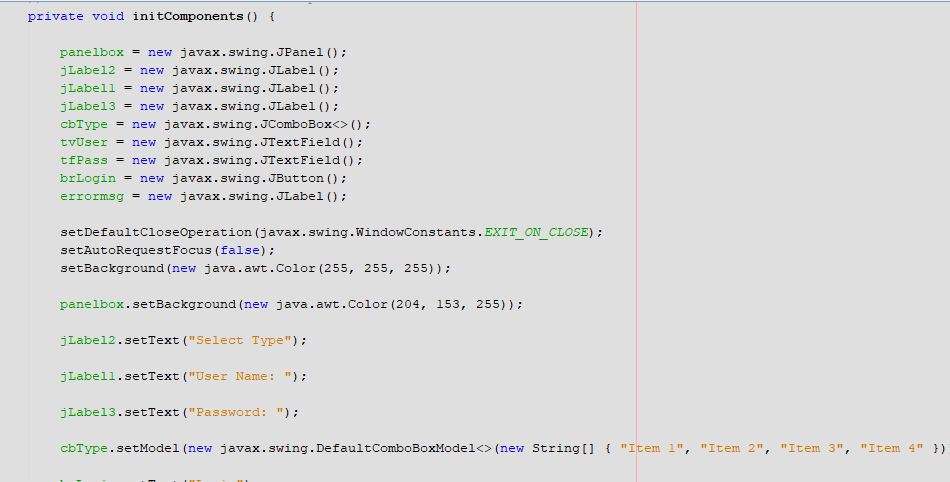


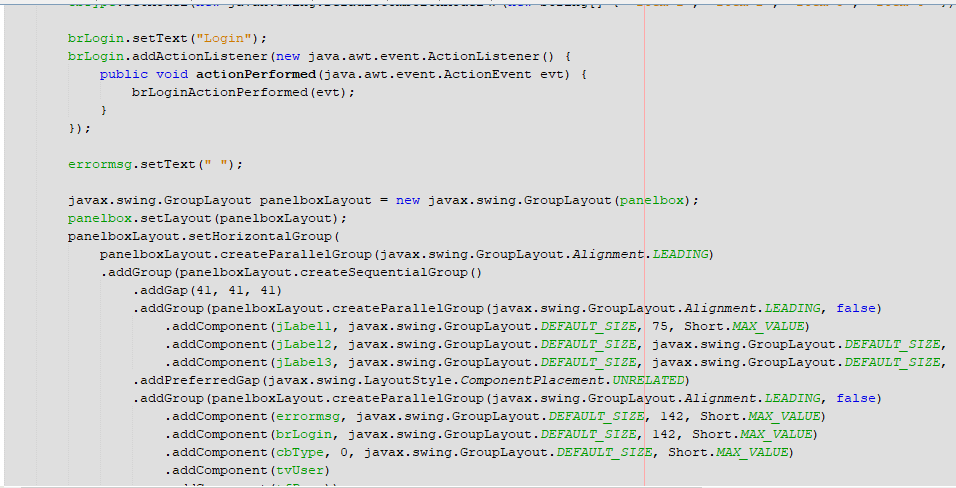


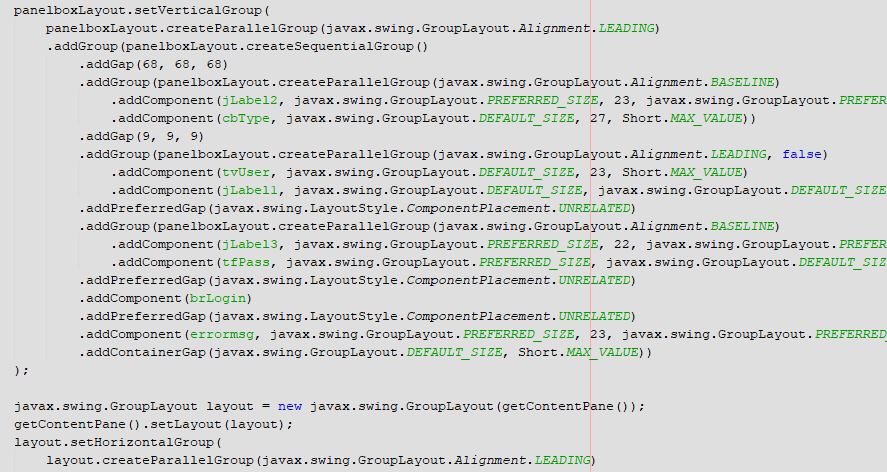


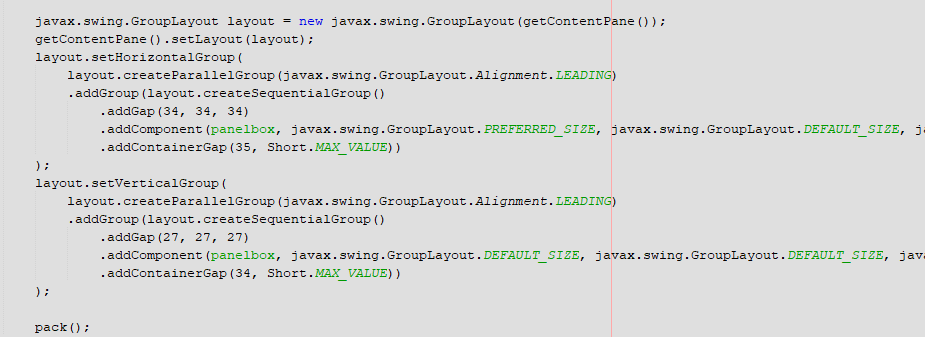


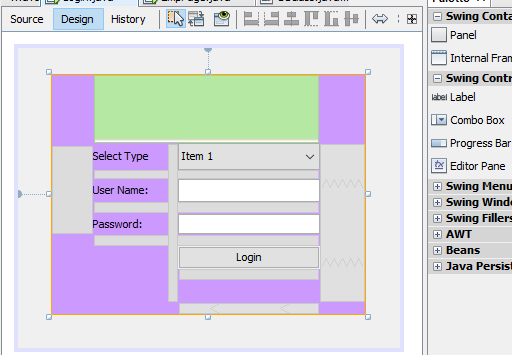
**The code for the UI:**











**Aim: Cost Estimation** of the proposed system, Schedule & Milestone using Function point estimation

Project Estimation

* Project scope must be understood
* Elaboration (decomposition) is necessary
* Historical metrics are very helpful
* At least two different techniques should be used
* Uncertainty is inherent in the process

**Function Point(FP) estimation:**

The function point metric (FP), first proposed by Albrecht , can be used effectively as a means for measuring the functionality delivered by a system.

* Function points are derived using an empirical relationship based on countable (direct) measures of software's information domain and assessments of software complexity

Information domain values are defined in the following manner:

* number of external inputs (EIs)
* number of external outputs (EOs)
* number of external inquiries (EQs)
* number of internal logical files (ILFs)
* Number of external interface files (EIFs)

**Information Weighting Factor**

**Domain Value Count Simple Average Complex**

External Inputs \_\_\_\_\_ x 3 4 6 = \_\_\_\_\_

External Outputs \_\_\_\_\_ x 4 5 7 = \_\_\_\_\_

External Inquiries \_\_\_\_\_ x 3 4 6 = \_\_\_\_\_

Internal Logical Files \_\_\_\_\_ x 7 10 15= \_\_\_\_\_

External Interface Files \_\_\_\_\_ x 5 7 10 = \_\_\_\_\_  
**Count total** \_\_\_\_\_\_\_\_

**FP=count total\*[0.65+(0.01\*sigma(Fi)]**

External Inputs (EI) - is an elementary process in which data crosses the boundary from outside to inside. This data may come from a data input screen or another application. The data may be used to maintain one or more internal logical files. The data can be either control information or business information. If the data is control information it does not have to update an internal logical file.

External Outputs (EO) - an elementary process in which derived data passes across the boundary from inside to outside. Additionally, an EO may update an ILF. The data creates reports or output files sent to other applications. These reports and files are created from one or more internal logical files and external interface file.  The following graphic represents on EO with 2 FTR's there is derived information (green) that has been derived from the ILF's

External Inquiry (EQ) - an elementary process with both input and output components that result in data retrieval from one or more internal logical files and external interface files. The input process does not update any Internal Logical Files, and the output side does not contain derived data. The graphic below represents an EQ with two ILF's and no derived data.

Internal Logical Files (ILF’s) - a user identifiable group of logically related data that resides entirely within the applications boundary and is maintained through external inputs.

External Interface Files (EIF’s) - a user identifiable group of logically related data that is used for reference purposes only. The data resides entirely outside the application and is maintained by another application. The external interface file is an internal logical file for another application.

**Value adjustment factors:**

1. Does the system require reliable backup and recovery?
2. Are specialized data communications required to transfer information to or from the application?
3. Are there distributed processing functions?
4. Is performance critical?
5. Will the system run in an existing, heavily utilized operational environment?
6. Does the system require on-line data entry?
7. Does the on-line data entry require the input transaction to be built over multiple screens or operations?
8. Are the internal logical files updated on-line?
9. Are the inputs, outputs, files, or inquiries complex?
10. Is the internal processing complex?
11. Is the code designed to be reusable?
12. Are conversion and installation included in the design?
13. Is the system designed for multiple installations in different organizations?
14. Is the application designed to facilitate change and for ease of use by the user?

**Solved example:**

The estimated number of FP is derived:

FPestimated = count-total 3 [0.65 + 0.01 3 S (Fi)]

FPestimated = 375

organizational average productivity = 6.5 FP/pm.

burdened labor rate = $8000 per month, the cost per FP is approximately $1230.

Based on the FP estimate and the historical productivity data, the total estimated project cost is $461,000 and the estimated effort is 58 person-months.

|  |  |
| --- | --- |
| **General System Characterstics (GSCs)** | **Degree of Influence (DI) 0-5** |
|  |  |
| Data Communications | \_\_\_\_\_1\_\_\_\_\_\_\_\_ |
| Distributed Data Processing | \_\_\_\_\_0\_\_\_\_\_\_\_\_ |
| Performace | \_\_\_\_\_3\_\_\_\_\_\_\_\_ |
| Heavily Used Configurations | \_\_\_\_\_2\_\_\_\_\_\_\_\_ |
| Transaction Rate | \_\_\_\_\_1\_\_\_\_\_\_\_\_ |
| Online Data Entry | \_\_\_\_\_0\_\_\_\_\_\_\_\_ |
| End User Efficency | \_\_\_\_\_3\_\_\_\_\_\_\_\_ |
| Online Update | \_\_\_\_\_0\_\_\_\_\_\_\_\_ |
| Complex Processing | \_\_\_\_\_1\_\_\_\_\_\_\_\_ |
| Reusability | \_\_\_\_\_3\_\_\_\_\_\_\_\_ |
| Installization Ease | \_\_\_\_\_2\_\_\_\_\_\_\_\_ |
| Operational Ease | \_\_\_\_\_1\_\_\_\_\_\_\_\_ |
| Multiple Sites | \_\_\_\_\_1\_\_\_\_\_\_\_\_ |
| Facilitate Change | \_\_\_\_\_2\_\_\_\_\_\_\_\_ |
| **Transaction Rate (TDI)** | \_\_\_\_\_20\_\_\_\_\_\_\_\_ |

**🡺FP=UFP\*VAF**

ilf - low

elf - low

ei - high

eq - low

eo - low

vaf - 0.85

ext ip = 5\*6

ext op = 5\*4

ext inq = 2\*3

ilf = 5\*7

eif = 1\*5

count total = 30+20+6+35+5 = 96

fp=upf\*vaf= 96\*0.85 = 81.6

Therefore, the calculated FPA is 81.6.

**Aim:** Design Test Cases for any two functionalities of your project

* Testing is the process of analyzing a system or system component to detect the differences between specified (required) and observed (existing) behavior.
* Activities involved in testing are:
  + Establish the test objectives
  + Design the test cases
  + Write the test cases
  + Test the test cases
  + Execute the tests
  + Evaluate the test results
  + Change the system

1. Select what has to be tested
   * Analysis: Completeness of requirements
   * Design: Cohesion
   * Implementation: Source code
2. Decide how the testing is done
   * Review or code inspection
   * Proofs (Design by Contract)
   * Black-box, white box,
   * Select integration testing strategy (big bang, bottom up, top down, sandwich)
3. Develop test cases
   * A test case is a set of test data or situations that will be used to exercise the unit (class, subsystem, system) being tested or about the attribute being measured

**Important testing documents are**

* Test plan
  + Focuses on managerial aspects of testing
  + Documents the scope, approach, resources and schedule of testing activities
  + Requirements and the components to be tested are identified in this document
* Test case specification
  + Writing **effective test cases** is a skill and that can be achieved by some experience and in-depth study of the application on which test cases are being written
* Test Incident Report
  + Each execution of each test is documented by test incident report
  + Actual results of the tests and differences from the expected output are recorded
* Test summary reports
  + It lists all the failures discovered during the tests that need to be investigated
  + Developers analyze and prioritize each failure
  + And plan for changes in the system.
  + These changes in turn can trigger new test cases and new test executions

Test Case Specification Template example:

**A.4 Task:**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Test case id** | **Test cases** | **Priority** | **Preconditions** | **Input test data** | **Steps to be executed** | **Expected results** | **Actual results** | **Pass/fail** | **Comments** |
| 1 | Test if user is able to login successfully. | A | User must be registered already | correct username,  correct password | 1)Enter input 2)click submit/login | User must successfully login to the web page |  |  |  |
| 2 | Test if unregistered users is not able to login to the site | A |  | incorrect username, incorrect password | 1)Enter input 2)click submit/login | Proper error must be displayed and prompt to enter login again | (note down the results you have observed) |  |  |

**B.1 Tasks given in PART A to be completed here:**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Test case id** | **Test cases** | **Priority** | **Preconditions** | **Input test data** | **Steps to be executed** | **Expected results** | **Actual results** | **Pass/fail** | **Comments** |
| 1 | Test if owner is able to login successfully. | A | User must be registered already | correct username,  correct password | 1)Enter input(correct )username and password on the respective fields 2)click submit/login | User must successfully login to the system | The page for the assigned account opens | pass |  |
| 2 | Test if unregistered users is not able to login to the site | A |  | incorrect username,incorrect password | 1)Enter input(incorrect )username and password on the respective fields 2)click submit/login | Proper error must be displayed and prompt to enter login again | Error message displayed | pass |  |
| 3 | Test if Manager is able to login successfully. | A | User must be registered already | correct username,  correct password | 1)Enter input(correct )username and password on the respective fields 2)click submit/login | User must successfully login to the system | No changes occur | Fail | Log shows:  Exception in thread "AWT-EventQueue-0" java.lang.NullPointerException |
| 4 | Test if Employee is able to login successfully. | A | User must be registered already | correct username,  correct password | 1)Enter input(correct )username and password on the respective fields 2)click submit/login | User must successfully login to the system | No chages occur | Fail | Log Shows:  Exception in thread "AWT-EventQueue-0" java.lang.NullPointerException |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Test case id** | **Test cases** | **Priority** | **Preconditions** | **Input test data** | **Steps to be executed** | **Expected results** | **Actual results** | **Pass/fail** | **Comments** |
| 1 | Test if user is able to enter details to db correctly | A | User must be logged in and have access to entry of information option | correct data | 1)Enter input 2)click submit | New row added | New row added | pass |  |
| 2 | Test if data entered in db is incorrect | A | User must be logged in and have access to entry of information option | incorrect data | 1)Enter input 2)click submit | Proper error must be displayed | Error message | pass |  |