**Identifying the Requirements from Problem Statements**

**Theory**

Objectives:

After completing this experiment you will be able to:

* Identify ambiguities, inconsistencies and incompleteness from a requirements specification
* Identify and state functional requirements
* Identify and state non-functional requirements

Requirements:

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.

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

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.
* **Consistency:** To illustrate this, consider the automation of a nuclear plant. Suppose one of the clients say that it the radiation level inside the plant exceeds R1, all reactors should be shut down. 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.

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.

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:

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.
* Identify the cases where an end user gets some meaningful work done by using the system.
* 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.
* Any high level requirement identified could have different sub-requirements.

Preparing Software Requirements Specifications:

Once all possible FRs and non-FRs have been identified, which are complete, consistent, and non-ambiguous, the Software Requirements Specification (SRS) is to be prepared. The SRS is prepared by the service provider, and verified by its client. This document serves as a legal agreement between the client and the service provider. Once the concerned system has been developed and deployed, and a proposed feature was not found to be present in the system, the client can point this out from the SRS. Also, if after delivery, the client says a new feature is required, which was not mentioned in the SRS, the service provider can again point to the SRS. The scope of the current experiment, however, doesn't cover writing a SRS.

**CASE STUDY**

**A Library Information System for SE VLabs Institute**

The SE VLabs Institute has been recently setup to provide state-of-the-art research facilities in the field of Software Engineering. Apart from research scholars (students) and professors, it also includes quite a large number of employees who work on different projects undertaken by the institution.

As the size and capacity of the institute is increasing with the time, it has been proposed to develop a Library Information System (LIS) for the benefit of students and employees of the institute. The librarian, who has administrative privileges and complete control over the system, can enter a new record into the system when a new book has been purchased, or remove a record in case any book is taken off the shelf. Any non-member is free to use this system to browse/search books online. However, issuing or returning books is restricted to valid users (members) of LIS only.

The final deliverable would a web application (using the recent HTML 5), which should run only within the institute LAN. Although this reduces security risk of the software to a large extent, care should be taken no confidential information (eg., passwords) is stored in plain text.

**Identification of functional requirements**

The above problem statement gives a brief description of the proposed system. From the above, even without doing any deep analysis, we might easily identify some of the basic functionality of the system:

* **New user registration:** Any member of the institute who wishes to avail the facilities of the library has to register himself with the Library Information System. On successful registration, a user ID and password would be provided to the member. He has to use this credentials for any future transaction in LIS.
* **Search book:** Any member of LIS can avail this facility to check whether any particular book is present in the institute's library. A book could be searched by its:
  + Title
  + Authors name
  + Publisher's name
* **User login:** A registered user of LIS can login to the system by providing his employee ID and password as set by him while registering. After successful login, "Home" page for the user is shown from where he can access the different functionalities of LIS: search book, issue book, return book, reissue book. Any employee ID not registered with LIS cannot access the "Home" page -- a login failure message would be shown to him, and the login dialog would appear again. In case the user fails to answer the security question correctly, his LIS account would be blocked. He needs to contact with the administrator to make it active again.
* **Issue book:** Any member of LIS can issue a book against his account provided that:
  + The book is available in the library i.e. could be found by searching for it in LIS
  + No other member has currently issued the book
  + Current user has not issued the maximum number of books that can

If the above conditions are met, the book is issued to the member.  
Note that this FR would remain **incomplete** if the "maximum number of books that can be issued to a member" is not defined. We assume that this number has been set to four for students and research scholars, and to ten for professors.  
Once a book has been successfully issued, the user account is updated to reflect the same.

* **Return book:** A book is issued for a finite time, which we assume to be a period of 20 days. That is, a book once issued should be returned within the next 20 days by the corresponding member of LIS. After successful return of a book, the user account is updated to reflect the same.
* **Reissue book:** Any member who has issued a book might find that his requirement is not over by 20 days. In that case, he might choose to reissue the book, and get the permission to keep it for another 20 days.

In a similar way we can list other functionality offered by the system as well. The "New User Registration" would, however, be available to non-members. Moreover, an already registered user shouldn't be allowed to register himself once again.

Having identified the (major) functional requirements, we assign an identifier to each of them for future reference and verification. Following table shows the list:

| Table 01: Identifier and priority for software requirements | | |
| --- | --- | --- |
| **#** | **Requirement** | **Priority** |
| R1 | New user registration | High |
| R2 | User Login | High |
| R3 | Search book | High |
| R4 | Issue book | High |
| R5 | Return book | High |
| R6 | Reissue book | Low |

**Identification of non-functional requirements**

Having talked about functional requirements, let's try to identify a few non-functional requirements.

* **Performance Requirements:**
  + This system should remain accessible 24x7
  + At least 50 users should be able to access the system altogether at any given time
* **Security Requirements:**
  + This system should be accessible only within the institute LAN
  + The database of LIS should not store any password in plain text -- a hashed value has to be stored
* **Software Quality Attributes**
* **Database Requirements**
* **Design Constraints:**
  + The LIS has to be developed as a web application, which should work with Firefox 5, Internet Explorer 8, Google Chrome 12, Opera 10
  + The system should be developed using HTML 5

Once all the functional and non-functional requirements have been identified, they are documented formally in SRS, which then serves as a legal agreement.

**Estimation of Project Metrics**

**Theory**

Objectives

**After completing this experiment you will be able to:**

* Categorize projects using COCOMO, and estimate effort and development time required for a project
* Estimate the program complexity and effort required to recreate it using Halstead's metrics

Project Estimation Techniques:

A software project is not just about writing a few hundred lines of source code to achieve a particular objective. 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.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?

In this experiment we will focus on two methods for estimating project metrics: COCOMO and Halstead's method.

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. 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:  
*Effort = a \* (KDSI)b PMTdev = 2.5 \* (Effort)c Months*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:

| **Software project** | ***a*** | ***b*** | ***c*** |
| --- | --- | --- | --- |
| Organic | 2.4 | 1.05 | 0.38 |
| Semi-detached | 3.0 | 1.12 | 0.35 |
| Embedded | 3.6 | 1.20 | 0.32 |
|  |  |  |  |

**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 .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)**.

| Cost drivers for Intermediate COCOMO | | | | | | |
| --- | --- | --- | --- | --- | --- | --- |
| **Cost Drivers** | **Ratings** | | | | | |
| **Very Low** | **Low** | **Nominal** | **High** | **Very High** | **Extra High** |
| **Product attributes** |  |  |  |  |  |  |
| Required software reliability | 0.75 | 0.88 | 1.00 | 1.15 | 1.40 |  |
| Size of application database |  | 0.94 | 1.00 | 1.08 | 1.16 |  |
| Complexity of the product | 0.70 | 0.85 | 1.00 | 1.15 | 1.30 | 1.65 |
| **Hardware attributes** |  |  |  |  |  |  |
| Run-time performance constraints |  |  | 1.00 | 1.11 | 1.30 | 1.66 |
| Memory constraints |  |  | 1.00 | 1.06 | 1.21 | 1.56 |
| Volatility of the virtual machine environment |  | 0.87 | 1.00 | 1.15 | 1.30 |  |
| Required turnabout time |  | 0.87 | 1.00 | 1.07 | 1.15 |  |
| **Personnel attributes** |  |  |  |  |  |  |
| Analyst capability | 1.46 | 1.19 | 1.00 | 0.86 | 0.71 |  |
| Applications experience | 1.29 | 1.13 | 1.00 | 0.91 | 0.82 |  |
| Software engineer capability | 1.42 | 1.17 | 1.00 | 0.86 | 0.70 |  |
| Virtual machine experience | 1.21 | 1.10 | 1.00 | 0.90 |  |  |
| Programming language experience | 1.14 | 1.07 | 1.00 | 0.95 |  |  |
| **Project attributes** |  |  |  |  |  |  |
| Application of software engineering methods | 1.24 | 1.10 | 1.00 | 0.91 | 0.82 |  |
| Use of software tools | 1.24 | 1.10 | 1.00 | 0.91 | 0.83 |  |
| Required development schedule | 1.23 | 1.08 | 1.00 | 1.04 | 1.10 |  |

EAF is used to refine the estimates obtained by basic COCOMO as follows:*Effort|corrected = Effort \* EAFTdev|corrected = 2.5 \* (Effort|corrected) c*

**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.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. 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.

Advantages of COCOMO:

COCOMO is a simple model, and should help one to understand the concept of project metrics estimation.

Halstead's Complexity Metrics

Halstead took a linguistic approach to determine the complexity of a program. According to him, a computer program consists of a collection of different operands and operators. THalstead's metrics are computed based on the operators and operands used in a computer program. Any given program has the following four parameters:

* + **n1**: Number of unique operators used in the program
  + **n2**: Number of unique operands used in the program
  + **N1**: Total number of operators used in the program
  + **N2**: Total number of operands used in the program

Using the above parameters one compute the following metrics:

* + **Program Length**: N = N1 + N2
  + **Program Vocabulary**: n = n1 + n2
  + **Volume**: V = N \* lg n
  + **Difficulty**: D = (n1 \* N2) / (2 \* n2)
  + **Effort**: E = D \* V
  + **Time to Implement**: T = E / 18 (in seconds)

The program volume V is the minimum number of bits needed to encode the program. It represents the size of the program while taking into account the programming language.  
The difficulty metric indicates how difficult a program is to write or understand.

**CASE STUDY**

**A Library Information System for SE VLabs Institute**

The SE VLabs Institute has been recently setup to provide state-of-the-art research facilities in the field of Software Engineering. Apart from research scholars (students) and professors, it also includes quite a large number of employees who work on different projects undertaken by the institution.

As the size and capacity of the institute is increasing with the time, it has been proposed to develop a Library Information System (LIS) for the benefit of students and employees of the institute. LIS will enable the members to borrow a book (or return it) with ease while sitting at his desk/chamber. The system also enables a member to extend the date of his borrowing if no other booking for that particular book has been made. For the library staff, this system aids them to easily handle day-to-day book transactions. The librarian, who has administrative privileges and complete control over the system, can enter a new record into the system when a new book has been purchased, or remove a record in case any book is taken off the shelf. Any non-member is free to use this system to browse/search books online. However, issuing or returning books is restricted to valid users (members) of LIS only.

The final deliverable would a web application (using the recent HTML 5), which should run only within the institute LAN. Although this reduces security risk of the software to a large extent, care should be taken no confidential information (eg., passwords) is stored in plain text.

The SE VLabs Institute has a IT management team of it's own. This team has been given the task to execute the Library Information System project. The team consists of a few experts from industry, and a batch of highly qualified engineers experienced with design and implementation of information systems.

Using COCOMO and based on the team size (small) and experience (high), the concerned project could be categorized as "organic". The experts, based on their prior experience, suggested that the project size could roughly be around 10 KLOC. This would serve as the basis for estimation of different project parameters using basic COCOMO, as shown below:

Effort = a \* (KLOC)b PM

Tdev = 2.5 \* (Effort)c Months

For organic category of project the values of a, b, c are 2.4, 1.05, 0.38 respectively. So, the projected effort required for this project becomes

Effort = 2.4 \* (10)1.05 PM

= 27 PM (approx)

So, around 27 person-months are required to complete this project. With this calculated value for effort we can also approximate the development time required:

Tdev = 2.5 \* (27)0.38 Months

= 8.7 Months (approx)

So, the project is supposed to be complete by nine months. However, estimations using basic COCOMO are largely idealistic. Let us refine them using intermediate COCOMO. Before doing so we determine the Effort Adjustment Factor (EAF) by assigning approprite weight to each of the following attributes.

| **Cost Drivers** | **Ratings** | | | | | |
| --- | --- | --- | --- | --- | --- | --- |
| **Very Low** | **Low** | **Nominal** | **High** | **Very High** | **Extra High** |
| **Product attributes** |  |  |  |  |  |  |
| Required software reliability | 0.75 | 0.88 | 1.00 | 1.15 | 1.40 |  |
| Size of application database |  | 0.94 | 1.00 | 1.08 | 1.16 |  |
| Complexity of the product | 0.70 | 0.85 | 1.00 | 1.15 | 1.30 | 1.65 |
| **Hardware attributes** |  |  |  |  |  |  |
| Run-time performance constraints |  |  | 1.00 | 1.11 | 1.30 | 1.66 |
| Memory constraints |  |  | 1.00 | 1.06 | 1.21 | 1.56 |
| Volatility of the virtual machine environment |  | 0.87 | 1.00 | 1.15 | 1.30 |  |
| Required turnabout time |  | 0.87 | 1.00 | 1.07 | 1.15 |  |
| **Personnel attributes** |  |  |  |  |  |  |
| Analyst capability | 1.46 | 1.19 | 1.00 | 0.86 | 0.71 |  |
| Applications experience | 1.29 | 1.13 | 1.00 | 0.91 | 0.82 |  |
| Software engineer capability | 1.42 | 1.17 | 1.00 | 0.86 | 0.70 |  |
| Virtual machine experience | 1.21 | 1.10 | 1.00 | 0.90 |  |  |
| Programming language experience | 1.14 | 1.07 | 1.00 | 0.95 |  |  |
| **Project attributes** |  |  |  |  |  |  |
| Application of software engineering methods | 1.24 | 1.10 | 1.00 | 0.91 | 0.82 |  |
| Use of software tools | 1.24 | 1.10 | 1.00 | 0.91 | 0.83 |  |
| Required development schedule | 1.23 | 1.08 | 1.00 | 1.04 | 1.10 |  |

The cells with yellow backgrounds highlight our choice of weight for each of the cost drivers. EAF is determined by multiplying all the chosen weights. So, we get

EAF = 0.53 (approx)

Using this EAF value we refine our estimates from basic COCOMO as shown below

Effort|corrected = Effort \* EAF

= 27 \* 0.53

= 15 PM (approx)

Tdev|corrected = 2.5 \* (Effort|corrected)c

= 2.5 \* (15)0.38

= 7 months (approx)

After refining our estimates it seems that seven months would likely be sufficient for completion of this project. This is still a rough estimate since we have not taken the underlying components of the software into consideration. Complete COCOMO model considers such parameters to give a more realistic estimate.

**Modeling UML Use Case Diagrams and Capturing Use Case Scenarios**

**Theory**

Objectives

**After completing this experiment you will be able to:**

* How to identify different actors and use cases from a given problem statement
* How to associate use cases with different types of relationships
* How to draw a use-case diagram

Use case diagrams:

Use case diagrams belong to the category of behavioural diagram of UML diagrams. Use case diagrams aim to present a graphical overview of the functionality provided by the system. It consists of a set of actions (referred to as use cases) that the concerned system can perform, one or more actors, and dependencies among them.

**Actor:** An actor can be defined as an object or set of objects, external to the system, which interacts with the system to get some meaningful work done. Actors could be human, devices, or even other systems.

Actors can be classified as below:

* **Primary actor**: They are principal users of the system, who fulfill their goal by availing some service from the system.
* **Supporting actor**: They render some kind of service to the system. "Bank representatives", who replenishes the stock of cash, is such an example. It may be noted that replenishing stock of cash in an ATM is not the prime functionality of an ATM.

In a use case diagram primary actors are usually drawn on the top left side of the diagram.

**Use Case**

A use case is simply a functionality provided by a system.

Continuing with the example of the ATM, *withdraw cash* is a functionality that the ATM provides. Therefore, this is a use case. Other possible use cases includes, *check balance*, *change PIN*, and so on.

Use cases include both successful and unsuccessful scenarios of user interactions with the system.

**Subject**

Subject is simply  the system under consideration. Use cases apply to a subject. For example, an ATM is a subject, having multiple use cases, and multiple actors interact with it. However, one should be careful of external systems interacting with the subject as actors.

Graphical Representation

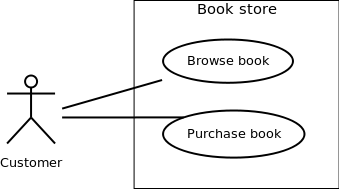
An actor is represented by a stick figure and name of the actor is written below it. A use case is depicted by an ellipse and name of the use case is written inside it. The subject is shown by drawing a rectangle. Label for the system could be put inside it. Use cases are drawn inside the rectangle, and actors are drawn outside the rectangle, as shown in figure - 01.

Figure - 01: A use case diagram for a book store

Association between Actors and Use Cases:

A use case is triggered by an actor. Actors and use cases are connected through binary associations indicating that the two communicates through message passing.

An actor must be associated with at least one use case. Similarly, a given use case must be associated with at least one actor. Association among the actors are usually not shown. However, one can depict the class hierarchy among actors.

Use Case Relationships

Three types of relationships exist among use cases:

* Include relationship
* Extend relationship
* Use case generalization

**Include Relationship**

Include relationships are used to depict common behaviour that are shared by multiple use cases. This could be considered analogous to writing functions in a program in order to avoid repetition of writing the same code. Such a function would be called from different points within the program.

**Notation:** Include relationship is depicted by a dashed arrow with a «include» stereotype from the including use case to the included use case.

**Extend Relationship :** Use case extensions are used used to depict any variation to an existing use case. They are used to the specify the changes required when any assumption made by the existing use case becomes false.

**Generalization Relationship:** Generalization relationship are used to represent the inheritance between use cases. A derived use case specializes some functionality it has already inherited from the base use case.

Identifying Use cases:

Once the primary and secondary actors have been identified, we have to find out their goals i.e. what are the functionality they can obtain from the system. Any use case name should start with a verb like, "Check balance".

**CASE STUDY**

**A Library Information System for SE VLabs Institute**

The SE VLabs Institute has been recently setup to provide state-of-the-art research facilities in the field of Software Engineering. Apart from research scholars (students) and professors, it also includes quite a large number of employees who work on different projects undertaken by the institution.

As the size and capacity of the institute is increasing with the time, it has been proposed to develop a Library Information System (LIS) for the benefit of students and employees of the institute. LIS will enable the members to borrow a book (or return it) with ease while sitting at his desk/chamber. The system also enables a member to extend the date of his borrowing if no other booking for that particular book has been made. Any non-member is free to use this system to browse/search books online. However, issuing or returning books is restricted to valid users (members) of LIS only.

The final deliverable would a web application (using the recent HTML 5), which should run only within the institute LAN. Although this reduces security risk of the software to a large extent, care should be taken no confidential information (eg., passwords) is stored in plain text.

From the given problem statement we can identify a list of actors and use cases as shown in tables 1 & 2 respectively. We assign an identifier to each use case, which we would be using to map from the software requirements identified earlier.

| Table 1: List of actors | |
| --- | --- |
| **Actor** | **Description** |
| Member | Can avail LIS facilities; could be student, professor, researcher |
| Non-member | Need to register to avail LIS facilities |
| Librarian | Update inventory and other administrative tasks |
| Library staff | Handle day-to-day activities with the LIS |

| Table 2: List of use cases | | |
| --- | --- | --- |
| **#** | **Use Case** | **Description** |
| UC1 | Register | Allows to register with the LIS and create an account for all transactions |
| UC2 | User login | LIS authenticates a member to let him avail the facilities |
| UC3 | Search book | A member can can search for a book |
| UC4 | Issue book | Allows a member to issue a specified book against his account |
| UC5 | Return book | To return a book, which has been issued earlier by a member |
| UC6 | Reissue book | To reissue a book |
| UC7 | User logout | User logs out from the system |

Before presenting the details of individual use cases, let us do a mapping from requirements specifications to use cases. A list of functional requirements can be found in the . For each such requirements, we identify the use case(s) that helps to achieve the requirement. This mapping is shown in table 3. Please note that we would be mapping only *functional requirements* into use cases. A method to deal with non-functional requirements could be found in .

| Table 3: Mapping functional requirements to use cases | | |
| --- | --- | --- |
| **FR #** | **FR Description** | **Use Case(s)** |
| R1 | New user registration | UC1 |
| R2 | User login | UC2 |
| R3 | Search book | UC3 |
| R4 | Issue book | UC4 |
| R5 | Return book | UC5 |
| R6 | Reissue book | UC6 |

Now let us deal with the inner details of a few use cases and the actors with whom they are associated. Table 4 shows the details of the "User login" use case using a template presented in table 1 in .

|  |  |
| --- | --- |
| Table 4: UC2 -- User login | |
| **Use Case** | UC2. User login |
| **Description** | Allows a member to login to the system using his user ID and password |
| **Assumptions** |  |
| **Actors** | * Member |
| **Steps** | 1. User types in user ID 2. User types in password 3. User clicks on the 'Login' button 4. IF successful THEN show home page ELSE display error |
| **Variations** |  |
| **Non-functional** |  |
| **Issues** |  |

The above use case lets an already registered member of the LIS to login to the system and possible use it's various features. if login credentials are incorrect, an error message is displayed to him. Figure 1 shows its pictorial representation.

A picture containing text, diagram, screenshot, line

Description automatically generated  
 Figure 1: Use case diagram showing "New user registration" use case

The above figure also depicts extension of a use case. "Answer security question" is not a use case by itself, and is not invoked in a "normal" flow. However, if the user fails to answer the security question correctly, his account is temporarily blocked. Details of the concerned use case extension is shown in table 5.

|  |  |
| --- | --- |
| Table 5: Extension for use case New user registration | |
| Use Case Extension | Answer security question **extends** UC2. User login |
| Description | Deals with the condition when a user has three consecutive login failures, and he attempts to login again |
| Steps | **3a.** IF consecutive failure count is 3 THEN invoke "Answer security question" |

The details of the "Issue book" use case is shown in table 6.

|  |  |
| --- | --- |
| Table 6: UC5 -- Issue book | |
| Use Case | UC5. Issue book |
| Description | Allows a member to issue a specified book against his account |
| Assumptions | 1. User is logged in 2. The book is available 3. User's account has not exceeded the limit of maximum books that can be issued |
| Actors | * Member (primary) * Library staff |
| Steps | 1. User logs in 2. User searches for a book 3. User clicks on "Issue" button to issue the book 4. User's account is updated 5. Library staff delivers the book |
| Variations |  |
| Non-functional |  |
| Issues |  |

In order to issue a book, the availability of the book has to be checked. Also, the system needs to verify whether another book could be issued to the current user. These have been represented by the "generalization" relationship in figure 2. A diagram of a library information system

Description automatically generated with medium confidence  
 Figure 2: Use case diagram showing "Issue book" use case

**E-R Modeling from the Problem Statements**

**THOERY**

Objectives

After completing this experiment you will be able to:

* Identify entity sets, their attributes, and various relationships
* Represent the data model through ER diagram

Entity Relationship Model:

Entity-Relationship model is used to represent a logical design of a database to be created. In ER model, real world objects (or concepts) are abstracted as entities, and different possible associations among them are modeled as relationships.

Entity Set and Relationship Set:

An entity set is a collection of all similar entities. For example, "Student" is an entity set that abstracts all students. Ram, John are specific entities belonging to this set. Similarly, a "Relationship" set is a set of similar relationships.

Attributes of Entity:

Attributes are the characteristics describing any entity belonging to an entity set. Any entity in a set can be described by zero or more attributes.

Keys

One or more attribute(s) of an entity set can be used to define the following keys:

* **Super key:** One or more attributes, which when taken together, helps to uniquely identify an entity in an entity set.
* **Candidate key:** It is a minimal subset of a super key. In other words, a super key might contain extraneous attributes, which do not help in identifying an object uniquely. When such attributes are removed, the key formed so is called a candidate key.
* **Primary key:** A database might have more than one candidate key. Any candidate key chosen for a particular implementation of the database is called a primary key.
* **Prime attribute:** Any attribute taking part in a super key

Weak Entity:

An entity set is said to be weak if it is dependent upon another entity set. A weak entity can't be uniquely identified only by it's attributes. Existence of a family is entirely dependent on the concerned employee. So, it is meaningful only with reference to employee.

Entity Generalization and Specialization:

Once we have identified the entity sets, we might find some similarities among them. For example, multiple person interacts with a banking system. Most of them are customers, and rest employees or other service providers. Here, customers, employees are persons, but with certain specializations.

Mapping Cardinalities:

One of the main tasks of ER modeling is to associate different entity sets. Let's consider two entity sets E1 and E2 associated by a relationship set R. Based on the number of entities in E1 and E2 are associated with, we can have the following four type of mappings:

* **One to one:** An entity in E1 is related to at most a single entity in E2, and vice versa
* **One to many:** An entity in E1 could be related to zero or more entities in E2. Any entity in E2 could be related to at most a single entity in E1.
* **Many to one:** Zero or more number of entities in E1 could be associated to a single entity in E2. However, an entity in E2 could be related to at most one entity in E1.
* **Many to many:** Any number of entities could be related to any number of entities in E2, including zero, and vice versa.

ER Diagram:

From a given problem statement we identify the possible entity sets, their attributes, and relationships among different entity sets. Once we have these information, we represent them pictorially, called an entity-relationship (ER) diagram.

Graphical Notations for ER Diagram

| **Term** | **Notation** | **Remarks** |
| --- | --- | --- |
| Entity set | Entity | Name of the set is written inside the rectangle |
| Attribute | Attribute | Name of the attribute is written inside the ellipse |
| Entity with attributes | Entity with attributes | Roll is the primary key; denoted with an underline |
| Weak entity set | Weak entity |  |
| Relationship set | Relationship | Name of the relationship is written inside the diamond |
| Related enity sets | Entity relationship |  |
| Relationship cardinality | Relationship cardinality | A person can own zero or more cars but no two persons can own the same car |
| Relationship with weak entity set | Weak entity relationship |  |

**CASE STUDY**

**A Library Information System for SE VLabs Institute**

The SE VLabs Institute has been recently setup to provide state-of-the-art research facilities in the field of Software Engineering. Apart from research scholars (students) and professors, it also includes quite a large number of employees who work on different projects undertaken by the institution.

As the size and capacity of the institute is increasing with the time, it has been proposed to develop a Library Information System (LIS) for the benefit of students and employees of the institute. LIS will enable the members to borrow a book (or return it) with ease while sitting at his desk/chamber. The system also enables a member to extend the date of his borrowing if no other booking for that particular book has been made. For the library staff, this system aids them to easily handle day-to-day book transactions. The librarian, who has administrative privileges and complete control over the system, can enter a new record into the system when a new book has been purchased, or remove a record in case any book is taken off the shelf. Any non-member is free to use this system to browse/search books online. However, issuing or returning books is restricted to valid users (members) of LIS only.

The final deliverable would a web application (using the recent HTML 5), which should run only within the institute LAN. Although this reduces security risk of the software to a large extent, care should be taken no confidential information (eg., passwords) is stored in plain text.

A robust database backend is essential for a high-quality information system. Database schema should be efficiently modeled, refined, and normalized. In this section we would develop a simple ER model for the Library Information System.

The first step towards ER modeling is to identify the set of relevant entities from the given problem statement. The two primary, and obvious, entity sets in this context are "Member" and "Book". The entity set "Member" represents all students, professors, or employees who have registered themselves with the LIS. While registering with the LIS one has to furnish a lot of personal and professional information. This typically includes name (well, that is trivial), employee ID (roll # for students), email address, phone #, age, date of joining in this institute. These characteristics are the attributes of the entities belonging to the entity set "Member".

It is essential for an entity to have one or more attributes that help us to distinguish it from another entity. 'Name' can't help that -- two persons could have exactly the same name. However, ('Name', 'Phone #') combination seems to be okay. No two persons can have the same phone number. 'Employee ID', 'Email address' are other potential candidates. Here, 'Employee ID', 'Email address' and ('Name', 'Phone #') are super keys

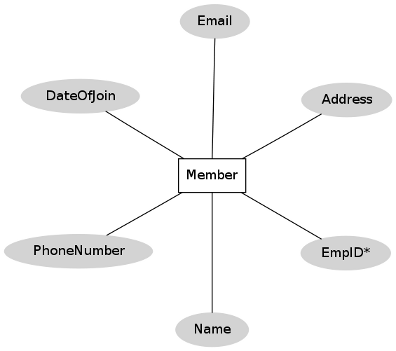
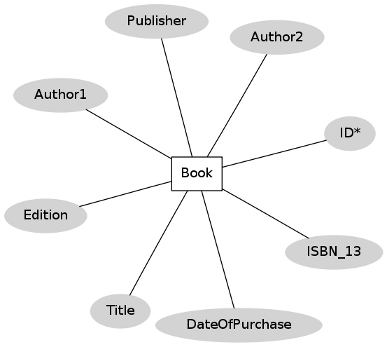


Figure 1: "Member" entity set

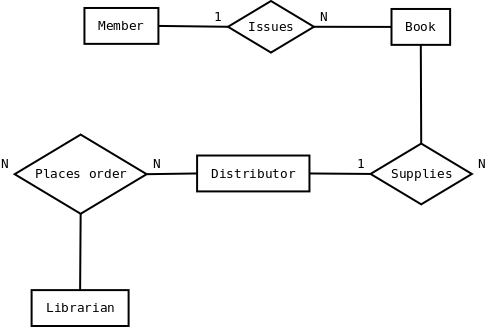
Let us now focus on the "Book" entity set. Typical attributes of a book are it's title, name of author(s), publisher, date of publication, edition, language, ISBN-10, ISBN-13, price (of course!), date of purchase. For instance, several books could have the same title. Again, ISBN numbers for a book are specific to it's edition -- it can't distinguish between two books of the same edition. One might be tempted to use a combination of ('Title', 'Authors') as a primary key. This has some shortcomings. It is advisable not to use texts as a PK. Moreover, the number of authors that a book could have is not fixed, although it is a small, finite number. The rules of normalization (not covered here) would dictate to have a separate field for each author like 'Author1', 'Author2', and so on.

  
Figure 2: "Book" entity set

One point to note here is that a book is likely to have multiple copies in the library. Therefore, one might wish to have a '# of copies' attribute for the "Book" entity set. The approach that we have taken is to uniquely identify each book even though they are copies of the same title.

To buy any new book an order is to be placed to the distributor. This task is done by the librarian. Therefore, "Librarian" and "Distributor" are two other entities playing roles in this system.

Having identified the key entities, we could now relate them with each other. Let us consider the entity sets "Member" and "Book". A member can issue books. In fact, he can issue multiple books up to a finite number say, NThis relationship between "Member" and "Book" entity sets is pictorially depicted in figure 3.

  
Figure 3: Relationships among different entity sets

**Identifying Domain Classes from the Problem Statements**

**THEORY**

Objectives

**After completing this experiment you will be able to:**

* Understand the concept of domain classes
* Identify a list of potential domain classes from a given problem statement

Domain Class:

In Object Oriented paradigm Domain Object Model has become subject of interest for its excellent problem comprehending capabilities towards the goal of designing a good software system. Domain Model, as a conceptual model gives proper understanding of problem description through its highly effective component – the Domain Classes. Domain classes are the abstraction of key entities, concepts or ideas presented in the problem statement .

Below we discuss some techniques that can be used to identify the domain classes.

Traditional Techniques for Identification of Classes.

**Grammatical Approach Using Nouns**

This object identification technique was proposed by Russell J. Abbot, and Grady Booch made the technique popular . This technique involves grammatical analysis of the problem statement to identify list of potential classes. The logical steps are:

1. Obtain the user requirements (problem statement) as a simple, descriptive English text. This basically corresponds to the use-case diagram for the problem statement.
2. Identify and mark the nouns, pronouns and noun phrases from the above problem statements.
3. List of potential classes is obtained based on the category of the nouns (details given later). For example, nouns that direct refer to any person, place, or entity in general, correspond to different objects. And so does singular proper nouns. On the other hand, plural nouns and common nouns are candidates that usually map into classes.

**Advantages**

This is one of the simplest approaches that could be easily understood and applied by a larger section of the user base. The problem statement does not necessarily be in English, but in any other language.

**Disadvantages**

The problem statement always may not help towards correct identification of a class. At times it could give us redundant classes. At times the problem statement may use abbreviations for large systems or concepts, and therefore, the identified class may actually point to an aggregate of classes. In other words, it may not find all the objects.

Using Generalization:

In this approach, all potential objects are classified into different groups based on some common behaviour. Classes are derived from these groups.

Using Subclasses :

Here, instead of identifying objects one goes for identification of classes based on some similar characteristics. These are the specialized classes. Common characteristics are taken from them to form the higher level generalized classes.

Steps to Identify Domain Classes from Problem Statement

We now present the steps to identify domain classes from a given problem statement. This approach is mostly based on the “Grammatical approach using nouns” discussed above, with some insights from .

1. Make a list of potential objects by finding out the nouns and noun phrases from narrative problem statement
2. Apply subject matter expertise (or domain knowledge) to identify additional classes
3. Filter out the redundant or irrelevant classes
4. Classify all potential objects based on categories.

| **Categories** | **Explanation** |
| --- | --- |
| People | Humans who carry out some function |
| Places | Areas set aside for people or things |
| Things | Physical objects |
| Organizations | Collection of people, resources, facilities and capabilities having a defined mission |
| Concepts | Principles or Ideas not tangible |
| Events | Things that happen (usually at a given date and time), or as a steps in an ordered sequence |

1. Group the objects based on similar attributes. While grouping we should remember that
   1. Different nouns (or noun phrases) can actually refer to the same thing (examples: house, home, abode)
   2. Same nouns (or noun phrases) could refer to different things or concepts (example: I go to school every day / This school of thought agrees with the theory)
2. Give related names to each group to generate the final list of top level classes
3. Iterate over to refine the list of classes

**CASE STUDY**

**A Library Information System for SE VLabs Institute**

The SE VLabs Institute has been recently setup to provide state-of-the-art research facilities in the field of Software Engineering. Apart from research scholars (students) and professors, it also includes quite a large number of employees who work on different projects undertaken by the institution.

As the size and capacity of the institute is increasing with the time, it has been proposed to develop a Library Information System (LIS) for the benefit of students and employees of the institute. LIS will enable the members to borrow a book (or return it) with ease while sitting at his desk/chamber. The system also enables a member to extend the date of his borrowing if no other booking for that particular book has been made. However, issuing or returning books is restricted to valid users (members) of LIS only.

The final deliverable would a web application (using the recent HTML 5), which should run only within the institute LAN. Although this reduces security risk of the software to a large extent, care should be taken no confidential information (eg., passwords) is stored in plain text.

From the given problem statement we can identify the following nouns and noun phrases:

* The SE VLabs Institute
* Software Engineering
* Research scholars
* Students
* Professors
* Employees
* Projects
* Institution
* Library Information System
* Members
* Book
* Desk
* Chamber
* LAN
* Software
* Information
* Passwords

Let us put the above into different categories.

**People**

* Research scholars
* Students
* Professors
* Employees
* Members
* Library staff
* Librarian
* Non-member

**Places**

* Chamber

**Things**

* Projects
* Book
* Desk
* System
* Shelf
* LAN

**Organizations**

* The SE VLabs Institute
* Institution

**Concepts**

* Software Engineering
* Library Information System
* Record
* Web application
* Software
* Information
* Password

**Events**

* Transactions

The nouns and noun phrases in the problem statement gives us a list of 25 potential classes. However, all of them may not be relevant. If we filter these entries, we might find that the follwong set of classes directly relate to the business activities of LIS:

* Member
* Book
* Transaction (of books)
* Librarian
* Employee

Although not explicitly mentioned in the problem statement, based on knowledge in related area one may point out few other potential classes:

* Book Inventory
* Distributor
* Order
* Order Line Item
* Payment
* Invoice

Among the classes listed above, 'Member', 'Librarian', 'Employee' share some common characteristics. For instance, everyone has a name, each has got an unique ID in the institution. The above identified conceptual classes pave the way for modeling of design and implementation classes.

**Statechart and Activity Modeling**

**THEORY**

Objectives

**After completing this experiment you will be able to:**

* Identify the distinct states a system have
* Identify the events causing transitions from one state to another
* Represent the above information pictorially using simple states
* Identify activities representing basic units of work, and represent their flow

Statechart Diagrams:

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.

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

Building Blocks of a Statechart Diagram:

**State**

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-01, an initial state is represented by a circle filled with black. An intermediate state is depicted by a rectangle with rounded corners. A final state is represented by a unfilled circle with an inner black-filled circle.

State

Figure-01: 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-02 shows the syntax for representing a typical (intermediate) state

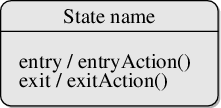


Figure-02: A typical state in a statechart diagram.

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

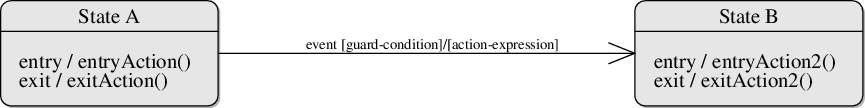
* **Event** is the what is causing the concerned transition (mandatory) -- Written in past tense
* **Guard-condition** is (are) precondition(s), which must be true for the transition to happen [optional]
* **Action-expression** indicate action(s) to be performed as a result of the transition [optional]

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.

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

Guidelines for drawing Statechart Diagrams:

Following steps could be followed, as suggested in [[i]](http://vlabs.iitkgp.ernet.in/isad/isad/6/references/) 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

Activity Diagrams:

Activity diagrams fall under the category of behavioural diagrams in Unified Modeling Language. It is a high level diagram used to visually represent the flow of control in a system. It has similarities with traditional flow charts.

Activity diagrams, however, cannot depict the message passing among related objects. As such, it can't be directly translated into code. These kind of diagrams are suitable for confirming the logic to be implemented with the business users. These diagrams are typically used when the business logic is complex. In simple scenarios it can be avoided entirely .

Components of an Activity Diagram

**Activity**

An activity denotes a particular action taken in the logical flow of control. This could simply be invocation of a mathematical function, alter an object's properties and so on . An activity is represented with a rounded rectangle, as shown in table-01. A label inside the rectangle identifies the corresponding activity.

There are two special type of activity nodes: initial and final. They are represented with a filled circle, and a filled in circle with a border respectively (table-01). There could be multiple initial nodes, which means that invoking that particular activity diagram would initiate multiple flows.

A final node represents the end point of all activities. Like an initial node, there could be multiple final nodes.

**Flow:** A flow is represented with a directed arrow. This is used to depict transfer of control from one activity to another, or to other types of components, as we will see below. A flow is often accompanied with a label, called the guard condition, indicating the necessary condition for the transition to happen.

**Decision:** A decision node, represented with a diamond, is a point where a single flow enters and two or more flows leave. The control flow can follow only one of the outgoing paths. The outgoing edges often have guard conditions indicating true-false or if-then-else conditions. Alternately, a note can be attached to the decision node indicating the condition to be tested.

**Merge :** This is represented with a diamond shape, with two or more flows entering, and a single flow leaving out. A merge node represents the point where at least a single control should reach before further processing could continue.

**Fork :** Fork is a point where parallel activities begin. For example, when a student has been registered with a college, he can in parallel apply for student ID card and library card. A fork is graphically depicted with a black bar, with a single flow entering and multiple flows leaving out.

**Join:** A join is depicted with a black bar, with multiple input flows, but a single output flow. Physically it represents the synchronization of all concurrent activities. Unlike a merge, in case of a join all of the incoming controls **must be completed** before any further progress could be made.

**Note:** UML allows attaching a note to different components of a diagram to present some textual information. The information could simply be a comment or may be some constraint. A note can be attached to a decision point, for example, to indicate the branching criteria.

**Partition**

Different components of an activity diagram can be logically grouped into different areas, called partitions or swimlanes. They often correspond to different units of an organization or different actors. The drawing area can be partitioned into multiple compartments using vertical (or horizontal) parallel lines. Partitions in an activity diagram are not mandatory.

The following table shows commonly used components with a typical activity diagram.

| **Component** | **Graphical Notation** |
| --- | --- |
| Activity | Activity |
| Flow | Flow |
| Decision | Decision |
| Merge | Merge |
| Fork | Fork |
| Join | Join |
| Note | Note |

Table-01: Typical components used in an activity diagram

**CASE STUDY**

**A Library Information System for SE VLabs Institute**

The SE VLabs Institute has been recently setup to provide state-of-the-art research facilities in the field of Software Engineering. Apart from research scholars (students) and professors, it also includes quite a large number of employees who work on different projects undertaken by the institution.

As the size and capacity of the institute is increasing with the time, it has been proposed to develop a Library Information System (LIS) for the benefit of students and employees of the institute. LIS will enable the members to borrow a book (or return it) with ease while sitting at his desk/chamber. The librarian, who has administrative privileges and complete control over the system, can enter a new record into the system when a new book has been purchased, or remove a record in case any book is taken off the shelf. Any non-member is free to use this system to browse/search books online.

From the given problem we can identify at least four different functionality offered by the system:

* Register a new member
* Issue book
* Reissue book
* Update inventory

To begin with, let's consider the activity diagram for user registration, as shown in figure - 01.

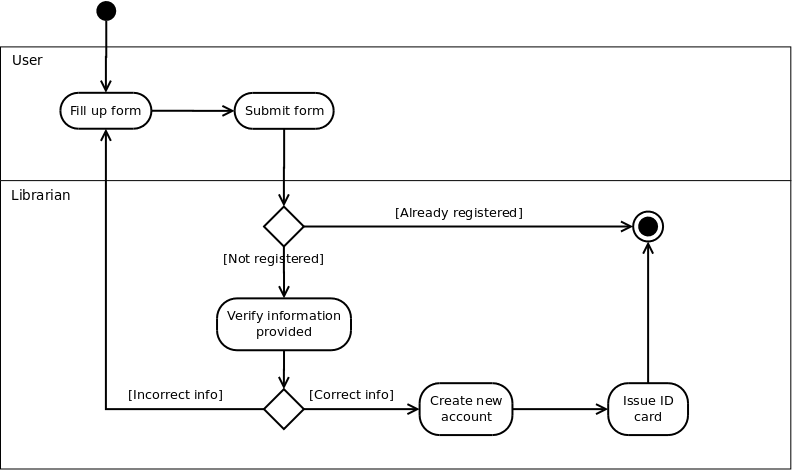


Figure-01: Activity diagram for new user registration

A new user fills up the registration form for library membership (either online or in paper), and submits to the librarian. Of course, an already registered user can't create another account for himself (or, herself). For users' who don't have an account already and have submitted their registration forms, the librarian verifies the information provided. If all information have been provided correctly, librarian goes on with creating a new account for the user. Once a new account has been created for the user, he (she) is being issued an ID card, which is to be provided for any future transaction in the library.

Note that in the above diagram two swim lanes haven been shown indicated by the labels *User* and *Librarian*. The activities have been placed in swim lanes that correspond to the relevant role.

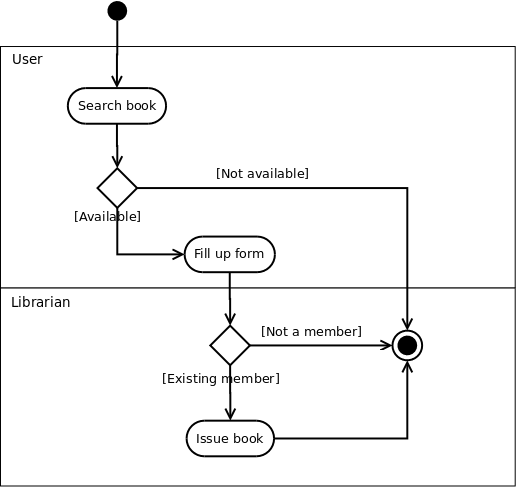
One of the major events that occur in any library is issue of books to it's members. 

Figure-02: Activity diagram for issuing books

Now let's focus on figure-03, which shows the typical workflow of inventory update by the librarian. Note that since these are the tasks performed only by the librarian (and no one else plays a role), we skip the swim lanes.

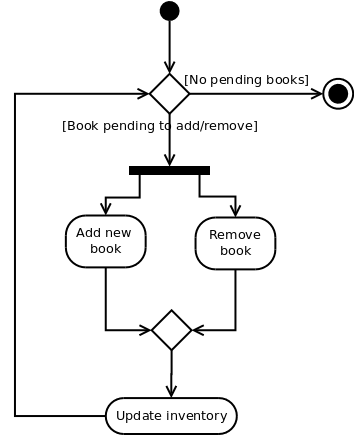


Figure-03: Activity diagram for updating inventory

Addition of new books and removing records of books taken off from the shelves could be done parallely. This means, one doesn't have to complete the task of addition of all new books before doing any removal. Merging of these two activities and the subsequent *Update inventory* activity indicates that it is not required to complete all addition and removals before proceeding to update the database.

Finally, the workflow terminates when all addition and removal tasks have been completed.

**Modeling UML Class Diagrams and Sequence diagrams**

**THEORY**

Objectives

**After completing this experiment you will be able to:**

* Graphically represent a class, and associations among different classes
* Identify the logical sequence of activities undergoing in a system, and represent them pictorially

Structural and Behavioral aspects:

Developing a software system in object oriented approach is very much dependent on understanding the problem. Some aspects and the respective models are used to describe problems and in context of those aspects the respective models give a clear idea regarding the problem to a designer. For developer, structural and behavioral aspects are two key aspects to see through a problem to design a solution for the same.

Class diagram: It is a graphical representation for describing a system in context of its static construction.

Elements in class diagram: Class diagram contains the system classes with its data members, operations and relationships between classes.

Class:

A set of objects containing similar data members and member functions is described by a class. In UML syntax, class is identified by solid outline rectangle with three compartments which contain

* **Class name :** A class is uniquely identified in a system by its name. A textual string is taken as class name. It lies in the first compartment in class rectangle.
* **Attributes:** Property shared by all instances of a class. It lies in the second compartment in class rectangle.
* **Operations :**An execution of an action can be performed for any object of a class. It lies in the last compartment in class rectangle.
* **Generalization/Specialization :** It describes how one class is derived from another class. Derived class inherits the properties of its parent class.

Geometric\_Shapes is the class that describes how many sides a particular shape has. Triangle, Quadrilateral and Pentagon are the classes that inherit the property of the Geometric\_Shapes class. So the relations among these classes are generalization. Now Equilateral\_Triangle, Isosceles\_Triangle and Scalene\_Triangle, all these three classes inherit the properties of Triangle class as each one of them has three sides. So, these are specialization of Triangle class.

Relationships:

Existing relationships in a system describe legitimate connections between the classes in that system.

* **Association :**It is an instance level relationship that allows exchanging messages among the objects of both ends of association. A simple straight line connecting two class boxes represent an association. We can give a name to association and also at the both end we may indicate role names and multiplicity of the adjacent classes. Association may be uni-directional.
* **Aggregation :**It is a special form of association which describes a part-whole relationship between a pair of classes. It means, in a relationship, when a class holds some instances of related class, then that relationship can be designed as an aggregation.
* Composition : It is a strong from of aggregation which describes that whole is completely owns its part. Life cycle of the part depends on the whole.
* **Multiplicity :** It describes how many numbers of instances of one class is related to the number of instances of another class in an association.

**Notation for different types of multiplicity:**

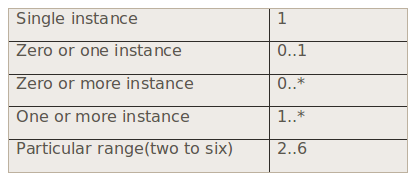


Figure-06:

**Example**

One vehicle may have two or more wheels

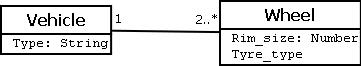


Figure-07:

Sequence diagram:

It represents the behavioral aspects of a system. Sequence diagram shows the interactions between the objects by means of passing messages from one object to another with respect to time in a system.

Elements in sequence diagram:

Sequence diagram contains the objects of a system and their life-line bar and the messages passing between them.

Object:

Objects appear at the top portion of sequence diagram. Object is shown in a rectangle box. Name of object precedes a colon ‘:’ and the class name, from which the object is instantiated. The whole string is underlined and appears in a rectangle box. Also, we may use only class name or only instance name.

Objects which are created at the time of execution of use case and are involved in message passing , are appear in diagram, at the point of their creation.

Life-line bar:

A down-ward vertical line from object-box is shown as the life-line of the object. A rectangle bar on life-line indicates that it is active at that point of time.

Messages:

Messages are shown as an arrow from the life-line of sender object to the life-line of receiver object and labeled with the message name. Chronological order of the messages passing throughout the objects’ life-line show the sequence in which they occur . There may exist some different types of messages :

* **Synchronous messages:**Receiver start processing the message after receiving it and sender needs to wait until it is made. A straight arrow with close and fill arrow-head from sender life-line bar to receiver end, represent a synchronous message.
* **Asynchronous messages:**For asynchronous message sender needs not to wait for the receiver to process the message. A function call that creates thread can be represented as an asynchronous message in sequence diagram.
* **Return message:**For a function call when we need to return a value to the object, from which it was called, then we use return message. But, it is optional, and we are using it when we are going to model our system in much detail. A dashed arrow with open arrow-head from sender life-line bar to receiver end, represent that message.
* **Response message:**One object can send a message to self. We use this message when we need to show the interaction between the same object.

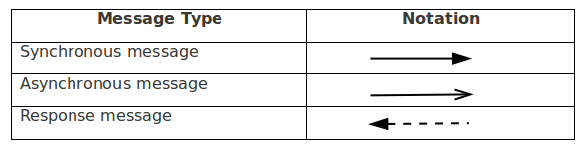


Figure-08

**CASE STUDY**

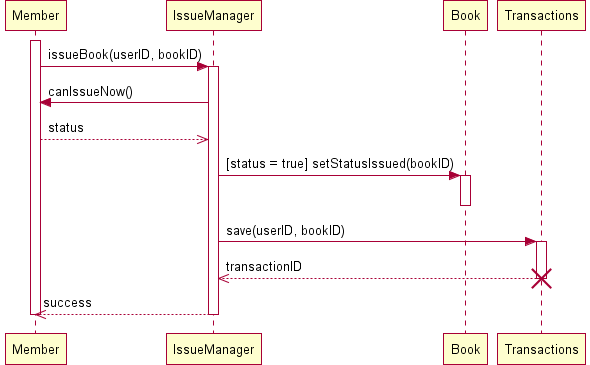
**A Library Information System for SE VLabs Institute**

The SE VLabs Institute has been recently setup to provide state-of-the-art research facilities in the field of Software Engineering. Apart from research scholars (students) and professors, it also includes quite a large number of employees who work on different projects undertaken by the institution.

As the size and capacity of the institute is increasing with the time, it has been proposed to develop a Library Information System (LIS) for the benefit of students and employees of the institute. LIS will enable the members to borrow a book (or return it) with ease while sitting at his desk/chamber. The librarian, who has administrative privileges and complete control over the system, can enter a new record into the system when a new book has been purchased, or remove a record in case any book is taken off the shelf. Any non-member is free to use this system to browse/search books online. However, issuing or returning books is restricted to valid users (members) of LIS only.

The final deliverable would a web application (using the recent HTML 5), which should run only within the institute LAN. Although this reduces security risk of the software to a large extent, care should be taken no confidential information (eg., passwords) is stored in plain text.

Let us consider the "Issue Book" use case and represent the involved steps in a sequence diagram as shown in figure 1. An user makes a request to issue a book against his account. This is shown by the "issueBook(bookID)" call from "Member" to "IssueManager" objects. At this point the system checks whether that particular user can issue another book (based on the maximum number of books that he can issue) by invoking the "canIssue()" method on the "Member". As a result of this call, a response ("status") is sent back to the "IssueManager" class. If the "status" is "true" (as indicated in the note), status of the concerned book is set to "issued". A new transaction is saved corresponding to the current issue of book by the user. Finally, a success message is sent back to "Member" indicating that the book was successfully issued.

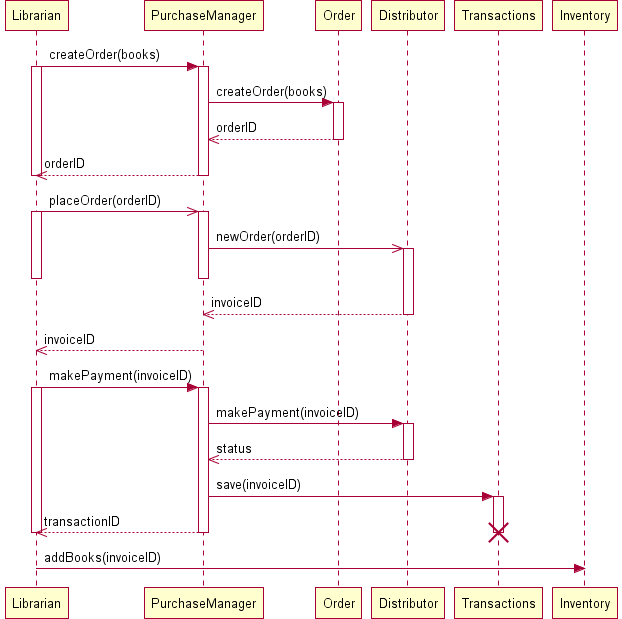
  
Figure 1: Sequence diagram for "Issue Book"

Few points could be noted here. Notes can be used almost anywhere within an UML diagram for whatever purpose. In figure 1 we use a note to specify the condition when status of a book is set to 'issued'. UML 1.0 had used guard conditions to specify such kind of Boolean logic. However, if the number of IF-THEN-ELSE conditions in a sequence diagram becomes high, the diagram gets complicated. In such cases one can draw multiple sequence diagrams for alternate conditions.

One key component in figure 1 is the "IssueManager" class. This class doesn't represent the actual Library Information System (LIS). Rather, this is a part of LIS -- a specific module to handle issuing of books to the members.

Also, note that the life cycle of the "Transactions" has been shown as self-destroyed. To understand this, consider how a transaction is actually implemented in code. Thereafter, the transaction object is not required to be in memory.

Figure 2 shows the order of steps involved in the process of purchasing of a new book. In this case also, "PurchaseManager" is a part of LIS, which manages all books that are being purchased. The activation bars indicate the different instances when a particular object is active in their corresponding life cycles.

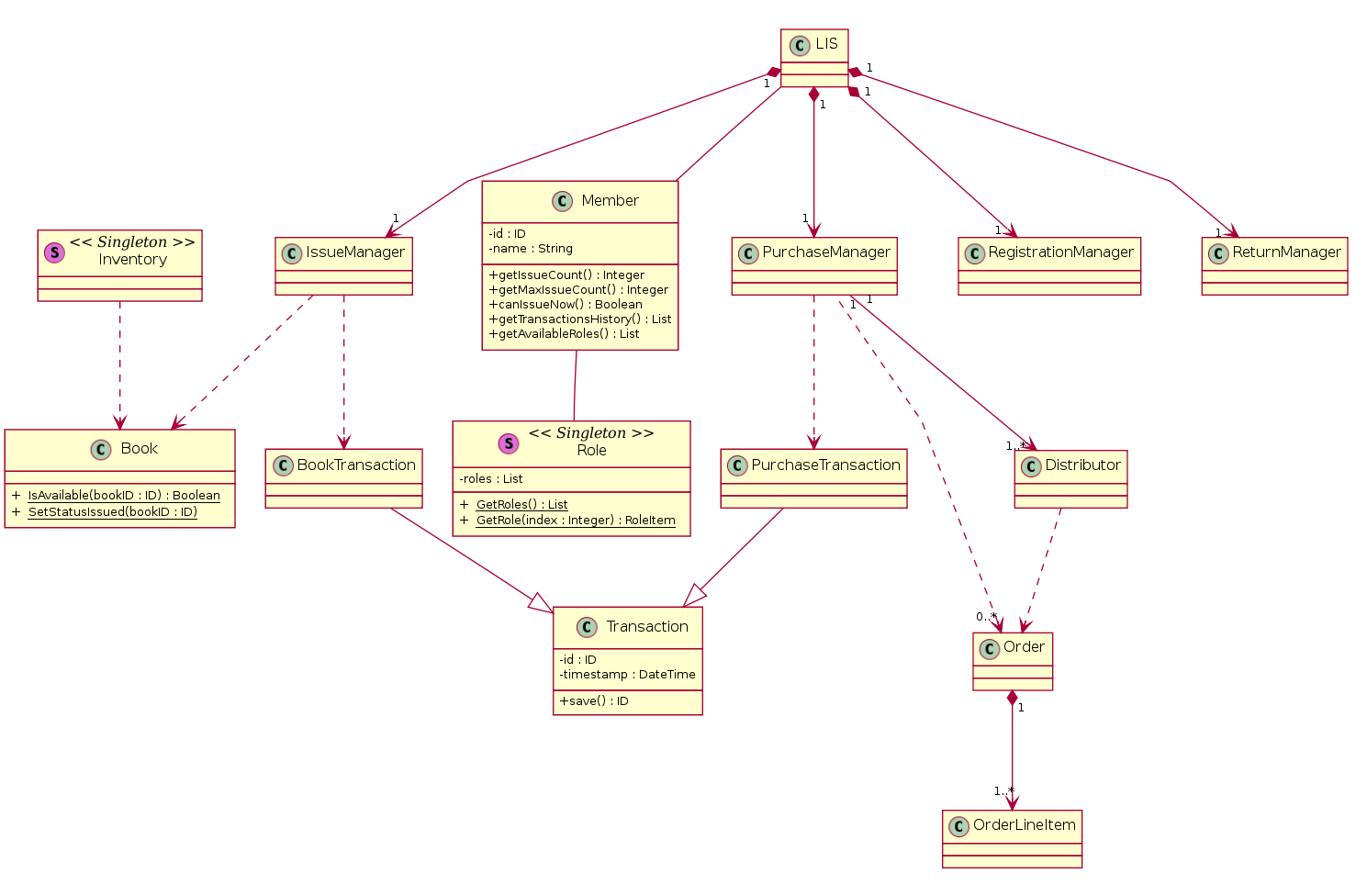
  
Figure 2: Sequence diagram for "Purchase Books"

One may have doubts over the inclusion of "Distributor" class. "Distributor" is not a constituent of the LIS; however, it interacts with LIS. Whenever the librarian places a new order to XYZ, the order is being sent electronically to XYZ, processed (possibly with a delay), a corresponding invoice is generated, and sent back to LIS. "placeOrder(orderID)" has been indicated as asynchronous calls since the calling object can continue with other tasks. Once the ordered books have been received, the librarian opts to make payment for his orders, which, too, could happen electronically through Net Banking. Technology has, indeed, made a huge progress!

Classes are the fundamental components of any object oriented design and development. Unless individual class, it's attributes and associated operations have been modeled well, a lot of suffereing could await during the development phase. One builds a model, analyze it's efficiency, and refines it thereafter, if required. Therefore, an analyst, designer, or developer doesn't have the tight constraints to create a perfect art at one go.

Based on conceptual modeling and domain knowledge we already had identified a list of classes. We present them here once again:

* Member
* Book
* Transaction (of books)
* Librarian
* Order
* Order Line Item
* Payment
* Invoice

Let's focus on the "Member", "Librarian" and "Employee" classes. The "Employee" class could be considered as a parent class, some of whose properties are inherited by the "Member" class. Again, "Librarian" is just a special type of "Member" with certain extra privileges. However, it may be noted here that LIS in no way would be interested to know about employees who are not members of LIS. Moreover, to distinguish between a normal member and a librarian, one could define a set of roles, and assign them appropriately to the members. This approach provides a flexible approach to manage users.. A list could be maintained in the "Member" class to indicate which roles are associated with a particular instance of it.  
Figure 3: A simplified class diagram for LIS

The "LIS" class consists of several modules: "RegistrationManager", "IssueManager" "ReturnManager", and "PurchaseManager". Their "composition" relationship with "LIS" indicates that any of these individual modules wouldn't exist without the existence of "LIS". The "IssueManager" class is responsible for issue and reissue of books while considering the two-times reissue constraint placed on a book.

The relation between "IssueManager" class and "Book" class is shown as "weak dependency". This is due to the reason that the "IssueManager" class do not require a "Book" as it's member variable. Rather, when an user has issued a book, the concerned method in "IssueManager" just needs to update the status of the corresponding book. The relationship between "PurchaseManager" and "Distributor" is, however, not a weak dependency. The "PurchaseManager" class has a member variable of type "Distributor", which keeps track of the distributor selected for the current purchase.

With the classes so identified, the code for issue book could look as follows:

public ID IssueBook(ID userID, ID bookID) {

Member user = Member.GetMember(userID);

ID transactionID = null;

if ( user.canIssueNow() && Book.IsAvailable(bookID) ) {

Book.SetStatusIssued(bookID);

user.incrementIssueCount(bookID);

BookTransaction transaction = new BookTransaction(userID, bookID);

transaction.save();

transactionID = transaction.getID();

}

return transactionID;

}

The code for reissuing a book to an user could look like the following.

public ID ReissueBook(ID userID, ID bookID) {

Member user = Member.GetMember(userID);

ID transactionID = null;

if ( user.canIssueNow() && Book.IsAvailable(bookID) ) {

Integer count = user.getReissueCountFor(bookID); // # of times this books has been reissued after it's recent issue by the user

if ( count < REISSUE\_LIMIT ) {

user.incrementReissueCount(bookID);

BookTransaction transaction = new BookTransaction(userID, bookID);

transaction.save();

transactionID = transaction.getID();

}

}

return transactionID;

}

**Modeling Data Flow Diagrams**

**THEORY**

Objectives

**After completing this experiment you will be able to:**

* Identify external entities and functionalities of any system
* Identify the flow of data across the system
* Represent the flow with Data Flow Diagrams

Data Flow 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.

Graphical notations for Data Flow Diagram

| **Term** | **Notation** | **Remarks** |
| --- | --- | --- |
| External entity | External entity | Name of the external entity is written inside the rectangle |
| Process | Process | Name of the process is written inside the circle |
| Data store | Data store | A left-right open rectangle is denoted as data store; name of the data store is written inside the shape |
| Data flow | Data flow | Data flow is represented by a directed arc with its data name |

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.. 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.

**Numbering of processes :**If process ‘p’ in context diagram is split into 3 processes ‘p1’, ‘p2’and ‘p3’ in next level then these are labeled as 0.1, 0.2 and 0.3 in level 1 respectively. Let the process ‘p3’ is again split into three processes ‘p31’, ‘p32’ and ‘p33’ in level 2, so, these are labeled as 0.3.1, 0.3.2 and 0.3.3 respectively and so on.

**Balancing DFD:**The data that flow into the process and the data that flow out to the process need to be match when the process is split into in the next level. This is known as balancing a DFD.

**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.

**CASE STUDY**

**A Library Information System for SE VLabs Institute**

The SE VLabs Institute has been recently setup to provide state-of-the-art research facilities in the field of Software Engineering. Apart from research scholars (students) and professors, it also includes quite a large number of employees who work on different projects undertaken by the institution.

As the size and capacity of the institute is increasing with the time, it has been proposed to develop a Library Information System (LIS) for the benefit of students and employees of the institute. LIS will enable the members to borrow a book (or return it) with ease while sitting at his desk/chamber. The librarian, who has administrative privileges and complete control over the system, can enter a new record into the system when a new book has been purchased, or remove a record in case any book is taken off the shelf. Any non-member is free to use this system to browse/search books online. However, issuing or returning books is restricted to valid users (members) of LIS only.

The final deliverable would a web application, which should run only within the institute LAN. Although this reduces security risk of the software to a large extent, care should be taken no confidential information (eg., passwords) is stored in plain text.

Figure 1 shows the context-level DFD for LIS. The entire system is represented with a single circle (process). Two database are used to keep track of member information and details of books in the library.

Otherwise, the user can continue with his operation. Note that a DFD does not show conditional flows. It can only summarize the information flowing in and out of the system.

The data flow with the label "Requested book details" identify the information that the user has to provide in order to issue a book. LIS checks with the books database whether the given book is available. After a book has been issued, the transaction details is provided to the member.

A picture containing diagram, text, technical drawing, plan

Description automatically generated

Figure 1: Context-level DFD for Library Information System

The level-1 DFD is shown in figure 2. Here, we split the top-level view of the system into multiple logical components. Each process has a name, and a dotted-decimal number in the form 1.x.

A picture containing diagram, text, screenshot, circle

Description automatically generated

Figure 2: Level 1 DFD for Library Information System

Comparing figures 1 and 2 one might observe that the information flow in and out of LIS has been preserved. We observe in figure 2 that the sub-processes themselves exchange information among themselves. These information flows would be, in turn, preserved if we decompose the system into a level 2 DFD.

**Estimation of Test Coverage Metrics and Structural Complexity**

**THOERY**

Objectives

**After completing this experiment you will be able to:**

* Identify basic blocks in a program module, and draw it's control flow graph (CFG)
* Identify the linearly independent paths from a CFG
* Determine Cyclomatic complexity of a module in a program

Control Flow Graph

A control flow graph (CFG) is a directed graph where the nodes represent different instructions of a program, and the edges define the sequence of execution of such instructions. Figure 1 shows a small snippet of code (compute the square of an integer) along with it's CFG. A directed edge from node #1 to node #2 in figure 1 implies that after execution of the first statement, the control of execution is transferred to the second instruction.

int x = 10, x\_2 = 0;

x\_2 = x \* x;

return x\_2;

A simple CFG

Figure 1: A simple program and it's CFG

A program, however, doesn't always consist of only sequential statements. There could be branching and looping involved in it as well. Figure 2 shows how a CFG would look like if there are sequential, selection and iteration kind of statements in order.

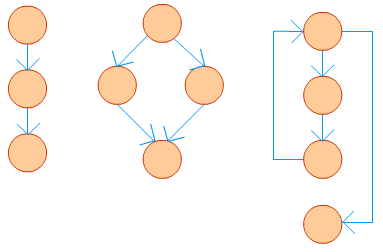


Figure 2: CFG for different types of statements

A real life application seldom could be written in a few lines. In fact, it might consist of thousand of lines. A CFG for such a program is likely to become very large, and it would contain mostly straight-line connections. To simplify such a graph different sequential statements could be grouped together to form a *basic block*. A **basic block** is a maximal sequence of program instructions I1, I2, ..., In such that for any two adjacent instructions Ik and Ik+1, the following holds true:

* Ik is executed immediately before Ik+1
* Ik+1 is executed immediately after Ik

The size of a CFG could be reduced by representing each basic block with a node. To illustrate this, let's consider the following example.

sum = 0;

i = 1;

while (i ≤ n) {

sum += i;

++i;

}

printf("%d", sum);

if (sum > 0) {

printf("Positive");

}

The CFG with basic blocks is shown for the above code in figure 3.

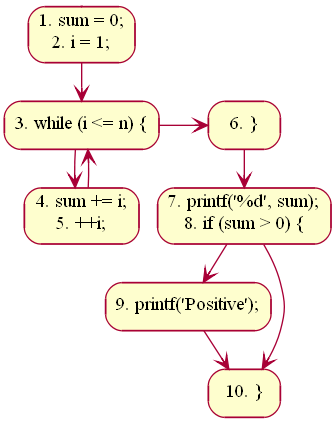


Figure 3: Basic blocks in a CFG.

The first statement of a basic block is termed as **leader**. Any node *x* in a CFG is said to dominate another node *y* (written as *x dom y*) if all possible execution paths that goes through node *y* must pass through node *x*. The node *x* is said to be a **dominator** . In the above example, line #s 1, 3, 4, 6, 7, 9, 10 are leaders. The node containing lines 7, 8 dominate the node containing line # 10.

If any block (or sub-graph) in a CFG is not connected with the sub-graph containing the entry block, that signifies the concerned block contains code, which is unreachable while the program is executed. Such unreachable code can be safely removed from the program. To illustrate this, let's consider a modified version of our previous code:

sum = 0;

i = 1;

while (i ≤ n) {

sum += i;

++i;

}

return sum;

if (sum < 0) {

return 0;

}

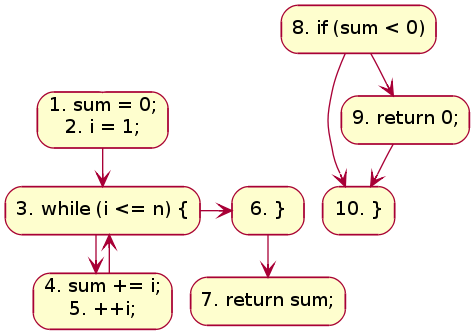


Figure 4: CFG with unreachable blocks

**Terminologies**

**Path**  
A path in a CFG is a sequence of nodes and edges that starts from the initial node (or entry block) and ends at the terminal node. The CFG of a program could have more than one terminal nodes.

**Linearly Independent Path**  
A linearly independent path is any path in the CFG of a program such that it includes at least one new edge not present in any other linearly independent path. A set of linearly independent paths give a clear picture of all possible paths that a program can take during it's execution. Therefore, path-coverage testing of a program would suffice by considering only the linearly independent paths.  
In figure 3 we can find four linearly independent paths:

1 - 3 - 6 - (7, 8) - 10

1 - 3 - 6 - (7, 8) - 9 - 10

1 - 3 - (4, 5) - 6 - (7, 8) - 10

1 - 3 - (4, 5) - 6 - (7, 8) - 9 - 10

McCabe's Cyclomatic Complexity

McCabe had applied graph-theoretic analysis to determine the complexity of a program module . Cyclomatic complexity metric, as proposed by McCabe, provides an upper bound for the number of linearly independent paths that could exist through a given program module. Thus, if Cyclomatic complexity of any program module is 7, there could be up to seven linearly independent paths in the module. For a complete testing, each of those possible paths should be tested.

**Computing Cyclomatic Complexity**

Let *G* be a a given CFG. Let *E* denote the number of edges, and *N* denote the number of nodes. Let *V(G)* denote the Cyclomatic complexity for the CFG. *V(G)* can be obtained in either of the following three ways:

* **Method #1:***V(G) = E - N + 2*
* **Method #2:** *V(G)* could be directly computed by a visual inspection of the CFG:*V(G) = Total number of bounded areas + 1*It may be noted here that structured programming would always lead to a planar CFG.
* **Method #3:** If LN be the total number of loops and decision statements in a program, then*V(G) = LN + 1*

In case of object-oriented programming, the above equations apply to methods of a class . Also, the value of V(G) so obtained is incremented by 1 considering the entry point of the method. Once the complexities of individual modules of a program are known, complexity of the program (or class) could be determined by :*V(G) = SUM( V(Gi) ) - COUNT( V(Gi) ) + 1*where *COUNT( V(Gi) )* gives the total number of procedures (methods) in the program (class).

**Optimum Value of Cyclomatic Complexity**

A set of threshold values for Cyclomatic complexity has been presented in , which we reproduce below.

| **V(G)** | **Module Category** | **Risk** |
| --- | --- | --- |
| 1-10 | Simple | Low |
| 11-20 | More complex | Moderate |
| 21-50 | Complex | High |
| > 50 | Unstable | Very high |

It has been suggested that the Cyclomatic complexity of any module should not exceed 10. Doing so would make a module difficult to understand for humans. Note that, a high value of V(G) is possible for a given module if it contains multiple *cases* in C like *switch-case* statements. McCabe had exempted such modules from the limit of V(G) as 10 .

**Merits**

McCabe's Cyclomatic complexity has certain advantages:

* Independent of programming language
* Helps in risk analysis during development or maintenance phase
* Gives an idea about the maximum number of test cases to be executed (hence, the required effort) for a given module

**Demerits**

Cyclomatic complexity doesn't reflect on cohesion and coupling of modules.

McCabe's Cyclomatic complexity was originally proposed for procedural languages. One may look in  to get an idea of how the complexity calculation could be modified for object-oriented languages. In fact, one may also wish to make use of Chidamber-Kemerer metrics (or any other similar metric), which has been designed for object-oriented programming.

**CASE STUDY**

**A Library Information System for SE VLabs Institute**

The SE VLabs Institute has been recently setup to provide state-of-the-art research facilities in the field of Software Engineering. Apart from research scholars (students) and professors, it also includes quite a large number of employees who work on different projects undertaken by the institution.

As the size and capacity of the institute is increasing with the time, it has been proposed to develop a Library Information System (LIS) for the benefit of students and employees of the institute. LIS will enable the members to borrow a book (or return it) with ease while sitting at his desk/chamber.. The librarian, who has administrative privileges and complete control over the system, can enter a new record into the system when a new book has been purchased, or remove a record in case any book is taken off the shelf. Any non-member is free to use this system to browse/search books online. However, issuing or returning books is restricted to valid users (members) of LIS only.

The final deliverable would a web application (using the recent HTML 5), Although this reduces security risk of the software to a large extent, care should be taken no confidential information (eg., passwords) is stored in plain text.

Let us determine the Cyclomatic complexity for the "ReissueBook" method as shown below:

public ID ReissueBook(ID userID, ID bookID) {

Member user = Member.GetMember(userID);

ID transactionID = null;

if ( user.canIssueNow() && Book.IsAvailable(bookID) ) {

Integer count = user.getReissueCountFor(bookID); // # of times this books has been reissued after it's recent issue by the user

if ( count < REISSUE\_LIMIT ) {

user.incrementReissueCount(bookID);

BookTransaction transaction = new BookTransaction(userID, bookID);

transaction.save();

transactionID = transaction.getID();

}

}

return transactionID;

}

The Control Flow Graph for the above module is shown in figure 1. The CFG has six nodes and seven edges. So, the Cyclomatic complexity is V(G) = 7 - 6 + 2 = 3. It can be verified with the other two formulae as well: # of regions + 1 = 2 + 1 = 3. Also, # of decision points = 2. So, V(G) = 2 + 1 = 3. However, as mentioned in the theory section, for methods of classes we add an extra 1 to the V(G). So, the Cyclomatic complexity of this method becomes 4, which is good.

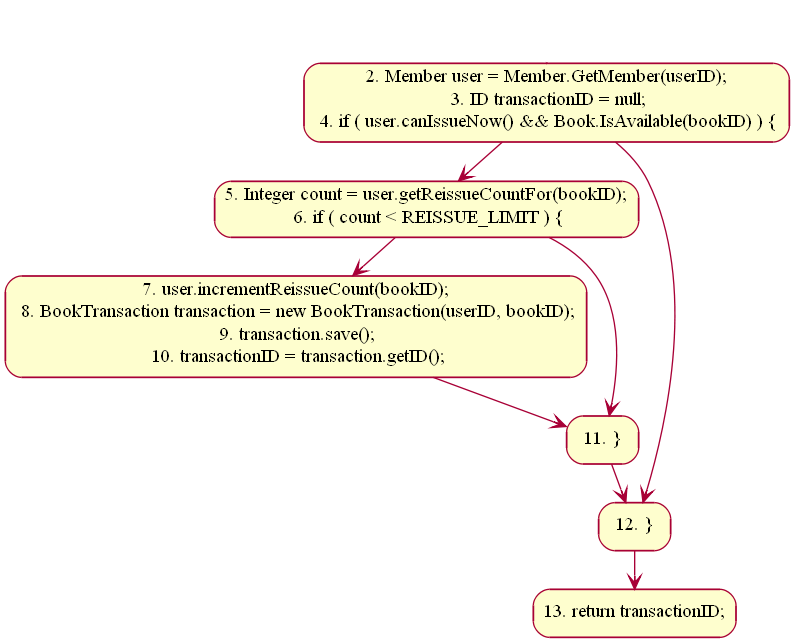


Figure 1. CFG for "ReissueBook" method

Note that in line # 3 two decisions have been short-circuited. Taking this into account, V(G) for the module would become 5, which is OK. This implies that the method could have upto five linearly independent paths. By looking at figure 1 we can easily identify three such paths. However, as mentioned that line # 3 consists of two decision points, that results in another "implicit" path. Based on these, we can design four test cases that would result in Boolean values for this sequence { user.canIssueNow, Book.IsAvailable, count < REISSUE\_LIMIT }. The four such cases are shown below:

* { true, true, true } : Output should be a valid ID
* { false, true, true } : Output would be null
* { true, false, true } : Output would be null
* { true, true, false } : Output would be null

Now let us focus on the "IssueManager" class. For simplicity, let's assume it has only two methods: IssueBook and ReissueBook, as shown below.

public Class IssueManager {

public ID IssueBook(ID userID, ID bookID) {

Member user = Member.GetMember(userID);

ID transactionID = null;

if ( user.canIssueNow() && Book.IsAvailable(bookID) ) {

Book.SetStatusIssued(bookID);

user.incrementIssueCount(bookID);

BookTransaction transaction = new BookTransaction(userID, bookID);

transaction.save();

transactionID = transaction.getID();

}

return transactionID;

}

public ID ReissueBook(ID userID, ID bookID) {

Member user = Member.GetMember(userID);

ID transactionID = null;

if ( user.canIssueNow() && Book.IsAvailable(bookID) ) {

Integer count = user.getReissueCountFor(bookID); // # of times this books has been reissued after it's recent issue by the user

if ( count < REISSUE\_LIMIT ) {

user.incrementReissueCount(bookID);

BookTransaction transaction = new BookTransaction(userID, bookID);

transaction.save();

transactionID = transaction.getID();

}

}

return transactionID;

}

}

"IssueBook" has two decision points (*if* and *&&*). So, V(GIssueBook) = (2 + 1) + 1 = 4. We have already determined V(GReissueBook) to be 5. So, the total Cyclomatic complexity of this class (having two methods) becomes*V(G) = (4 + 5) - 2 + 1 = 8*

**Designing Test Suites**

**THEORY**

Objectives

**After completing this experiment you will be able to:**

* Learn about different techniques of testing a software
* Design unit test cases to verify the functionality and locate bugs, if any

Software Testing

Testing software is an important part of the development life cycle of a software. It is an expensive activity. Hence, appropriate testing methods are necessary for ensuring the reliability of a program. According to the ANSI/IEEE 1059 standard, the definition of testing is the process of analyzing a software item, to detect the differences between existing and required conditions i.e. defects/errors/bugs and to evaluate the features of the software item.

The purpose of testing is to verify and validate a software and to find the defects present in a software. The purpose of finding those problems is to get them fixed.

* **Verification**is the checking or we can say the testing of software for consistency and conformance by evaluating the results against pre-specified requirements.
* **Validation** looks at the systems correctness, i.e. the process of checking that what has been specified is what the user actually wanted.
* **Defect** is a variance between the expected and actual result. The defect’s ultimate source may be traced to a fault introduced in the specification, design, or development (coding) phases.

Standards for Software Test Documentation

IEEE 829-1998 is known as the 829 Standard for Software Test Documentation. It is an IEEE standard that specifies the form of a set of documents for use in software testing [[i]](http://vlabs.iitkgp.ernet.in/isad/isad/10/references/). There are other different standards discussed below.

* IEEE 1008, a standard for unit testing
* IEEE 1012, a standard for Software Verification and Validation
* IEEE 1028, a standard for software inspections
* IEEE 830, a guide for developing system requirements specifications
* IEEE 730, a standard for software quality assurance plans
* IEEE 1061, a standard for software quality metrics and methodology
* IEEE 12207, a standard for software life cycle processes and life cycle data
* BS 7925-1, a vocabulary of terms used in software testing
* BS 7925-2, a standard for software component testing

Testing Frameworks

Following are the different testing frameworks:

* jUnit - for Java unit test
* Selenium - is a suite of tools for automating web applications for software testing purposes, plugin for Firefox
* HP QC - is the HP Web-based test management tool. It familiarizes with the process of defining releases, specifying requirements, planning tests, executing tests, tracking defects, alerting on changes, and analyzing results. It also shows how to customize project
* IBM Rational - Rational software has a solution to support business sector for designing, implementing and testing software

Need for Software Testing

There are many reasons for why we should test software, such as:

* Software testing identifies the software faults. The removal of faults helps reduce the number of system failures. Reducing failures improves the reliability and the quality of the systems.
* Software testing can also improves the other system qualities such as maintainability, usability, and testability.
* In order to meet the different legal requirements.
* In order to meet industry specific standards such as the Aerospace, Missile and Railway Signaling standards.

Test Cases and Test Suite

A test case describes an input descriptions and an expected output descriptions. Input are of two types: preconditions (circumstances that hold prior to test case execution) and the actual inputs that are identified by some testing methods. The set of test cases is called a test suite. We may have a test suite of all possible test cases.

Types of Software Testing

Testing is done in every stage of software development life cycle, but the testing done at each level of software development is different in nature and has different objectives. There are different types of testing, such as stress testing, volume testing, configuration testing, compatibility testing, recovery testing, maintenance testing, documentation testing, and usability testing. Software testing are mainlyof following types

1. **Unit Testing**
2. **Integration Testing**
3. **System Testing**

**Unit Testing**

Unit testing is done at the lowest level. It tests the basic unit of software, that is the smallest testable piece of software. The individual component or unit of a program are tested in unit testing. Unit testing are of two types.

* **Black box testing**: This is also known as **functional testing** , where the test cases are designed based on input output values only. There are many types of Black Box Testing but following are the prominent ones.

- **Equivalence class partitioning**: In this approach, the domain of input values to a program is divided into a set of equivalence classes.

- **Boundary value analysis**: In this approach, while designing the test cases, the values at boundaries of different equivalence classes are taken into consideration.

**White box testing**: It is also known as **structural testing**. In this testing, test cases are designed on the basis of examination of the code.This testing is performed based on the knowledge of how the system is implemented. It includes analyzing data flow, control flow, information flow, coding practices, exception and error handling within the system, to test the intended and unintended software behavior. Though white box testing can be performed any time in the life cycle after the code is developed, but it is a good practice to perform white box testing during the unit testing phase.

**Integration Testing**

Integration testing is performed when two or more tested units are combined into a larger structure. The main objective of this testing is to check whether the different modules of a program interface with each other properly or not. This testing is mainly of two types:

* **Top-down approach**
* **Bottom-up approach**

In bottom-up approach, each subsystem is tested separately and then the full system is tested.. After the top-level ‘skeleton’ has been tested, the immediately subroutines of the ‘skeleton’ are combined with it and tested.

**System Testing**

System testing tends to affirm the end-to-end quality of the entire system. System testing is often based on the functional / requirement specification of the system. Non-functional quality attributes, such as reliability, security, and maintainability are also checked. There are three types of system testing

* **Alpha testing** is done by the developers who develop the software. This testing is also done by the client or an outsider with the presence of developer or we can say tester.
* **Beta testing**is done by very few number of end users before the delivery, where the change requests are fixed, if the user gives any feedback or reports any type of defect.
* **User Acceptance testing**is also another level of the system testing process where the system is tested for acceptability. This test evaluates the system's compliance with the client requirements and assess whether it is acceptable for software delivery

An error correction may introduce new errors. Regression testing does not belong to either unit testing, integration testing, or system testing, instead, it is a separate dimension to these three forms of testing.

**Regression Testing**

The purpose of regression testing is to ensure that bug fixes and new functionality introduced in a software do not adversely affect the unmodified parts of the program. Regression testing is an important activity at both testing and maintenance phases. To this end, regression testing is to retest the software using the test cases selected from the original test suite.

**Example**

Write a program to calculate the square of a number in the range 1-100

#include <stdio.h>

int

main()

{

int n, res;

printf("Enter a number: ");

scanf("%d", &n);

if (n >= 1 && n <= 100)

{

res = n \* n;

printf("\n Square of %d is %d\n", n, res);

}

else if (n<= 0 || n > 100)

printf("Beyond the range");

return 0;

}

Output

Inputs Outputs

I1 : -2 O1 : Beyond the range

I2 : 0 O2 : Beyond the range

I3 : 1 O3 : Square of 1 is 1

I4 : 100 O4 : Square of 100 is 10000

I5 : 101 O5 : Beyond the range

I6 : 4 O6 : Square of 4 is 16

I7 : 62 O7 : Square of 62 is 3844

Test Cases

T1 : {I1 ,O1}

T2 : {I2 ,O2}

T3 : {I3, O3}

T4 : {I4, O4}

T5 : {I5, O5}

T6 : {I6, O6}

T7 : {I7, O7}

Some Remarks

A prevalent misconception among the beginners is that one should be concerned with testing only after coding ends. Testing is, in fact, not a phase towards the end. It is rather a continuous process. The Software Requirements Specification (SRS) document captures all features to be expected from the system. The requirements so identified here should serve as a basis towards preparation of the test cases. However, testing a software is not only about proving that it works correctly. Successful testing should also point out the bugs present in the system, if any.

**CASE STUDY**

**A Library Information System for SE VLabs Institute**

The SE VLabs Institute has been recently setup to provide state-of-the-art research facilities in the field of Software Engineering. Apart from research scholars (students) and professors, it also includes quite a large number of employees who work on different projects undertaken by the institution.

As the size and capacity of the institute is increasing with the time, it has been proposed to develop a Library Information System (LIS) for the benefit of students and employees of the institute. LIS will enable the members to borrow a book (or return it) with ease while sitting at his desk/chamber. The librarian, who has administrative privileges and complete control over the system, can enter a new record into the system when a new book has been purchased, or remove a record in case any book is taken off the shelf. Any non-member is free to use this system to browse/search books online. However, issuing or returning books is restricted to valid users (members) of LIS only.

The final deliverable would a web application (using the recent HTML 5), which should run only within the institute LAN. Although this reduces security risk of the software to a large extent, care should be taken no confidential information (eg., passwords) is stored in plain text.

As already discussed under the theory section, test case preparation could begin right after requirements identification stage. It is desirable (and advisable) to create a Requirements Traceability Matrix (RTM) showing a mapping from individual requirement to test case(s). A simplified form of the RTM is shown in table 1 (the numbers shown in this table are arbitrary; not specific to LIS).

| Table 1: A simplified mapping from requirements to test cases | |
| --- | --- |
| **Requirement #** | **Test Case #** |
| R1 | TC1 |
| R2 | TC2, TC3, TC4 |
| R3 | TC5 |
| R4 | TC6 |

Table 1 states which test case should help us to verify that a specified feature has been implemented and working correctly. For instance, if test case # TC6 fails, that would indicate requirement # R4 has not fully realized yet.

To be specific to our problem, let us see how we can design test cases to verify the "User Login" feature. The simplest scenario is when both user name and password have been typed in correctly. The outcome will be that the user could then avail all features of LIS. However, there could be multiple unsuccessful conditions:

* User ID is wrong
* Password is wrong
* User ID & password are wrong
* Wrong password given twice consecutively
* Wrong password given thrice consecutively, and security question answered correctly
* Wrong password given thrice consecutively, and security question answered incorrectly

We would create test case for the above stated login scenarios. These test cases together would constitute a test suite to verify the concerned requirement. Table 2 shows the details of this test suite.

In a similar way, test suites corresponding to other user requirements could be created as well. A good test plan can reduce the burden of testing team by specifying what exactly they should focus on.