PRAGNANAM BRAHMA
ANITS
ANIL NEERUKONDA INSTITUTE OF TECHNOLOGY & SCIENCES
Autonomous status accorded by UGC and Andhra University
Approved by AICTE, Permanently Affiliated to Andhra University
Accredited by NBA (IT,CSE,EEE,ECE, and Mech) & accredited by NAAC with "A" Grade
Sangivalasa - 531162, Bheemunipatnam (Mandal), Visakhapatnam (Dist.)
Phone: 08933 - 225083, 225084, 226131, Fax: 08933-226395
Email: principal@anits.edu.in
COLLEGE CODE - ANIL

DEPARTMENT OF COMPUTER SCIENCES & ENGINEERING

Vision

Our vision is to emerge as a world class Computer Science and Engineering department through excellent teaching and strong research environment that responds swiftly to the challenges of changing computer science technology and addresses technological needs of the stakeholders.

Mission

To enable our students to master the fundamental principles of computing and to develop in them the skills needed to solve practical problems using contemporary computer-based technologies and practices to cultivate a community of professionals who will serve the public as resources on state-of- the-art computing science and information technology.

Course outcomes

Student will be able to

1. Analyze the case study and apply the UML notations.

2. Estimate the project metrics using COCOMO.

3. Calculate the complexity using McCabe's Cyclomatic Complexity.

4. Compare and contrast testing techniques.

PROGRAM OUTCOMES (POs)

Graduate Attribute1: Engineering Knowledge

PO-A: An ability to apply the knowledge of basic engineering sciences, humanities, core engineering and computing concept in modeling and designing computer based systems.

Graduate Attribute2: Problem Analysis

PO-B: An ability to identify, analyze the problems in different domains and define the requirements appropriate to the solution.

Graduate Attribute3: Design/Development of Solution

PO-C: An ability to design, implement &amp; test a computer based system, component or process that meet functional constraints such as public health and safety, cultural, societal and environmental considerations.

Graduate Attribute4: Conduct Investigations of Complex Problems

PO-D: An ability to apply computing knowledge to conduct experiments and solve complex problems, to analyze and interpret the results obtained within specified timeframe and financial constraints consistently.

Graduate Attribute5: Modern Tool Usage

PO-E: An ability to apply or create modern techniques and tools to solve engineering problems that demonstrate cognition of limitations involved in design choices.

Graduate Attribute6: The Engineer and Society

PO-F: An ability to apply contextual reason and assess the local and global impact of professional engineering practices on individuals, organizations and society.

Graduate Attribute7: Environment and Sustainability

PO-G: An ability to assess the impact of engineering practices on societal and environmental sustainability.

Graduate Attribute8: Ethics

PO-H Ability to apply professional ethical practices and transform into good responsible citizens with social concern.

Graduate Attribute9: Individual and Team Work

PO-I: Acquire capacity to understand and solve problems pertaining to various fields of engineering and be able to function effectively as an individual and as a member or leader in a team.

Graduate Attribute10: Communication

PO-J: An ability to communicate effectively with range of audiences in both oral and written forms through technical papers, seminars, presentations, assignments, project reports etc.

Graduate Attribute11: Project Management and Finance

PO-K: An ability to apply the knowledge of engineering, management and financial principles to develop and critically assess projects and their outcomes in multidisciplinary areas.

Graduate Attribute12: Life-long Learning

PO-L: An ability to recognize the need and prepare oneself for lifelong self learning to be abreast with rapidly changing technology.

PROGRAM SPECIFIC OUTCOMES (PSOs)

1. Programming and software Development skills: Ability to acquire programming efficiency to analyze, design and develop optimal solutions, apply standard practices in software project development to deliver quality software product.

2. Computer Science Specific Skills: Ability to formulate, simulate and use knowledge in various domains like data engineering, image processing and information and network security, Artificial intelligence etc., and provide solutions to new ideas and innovations.

ANIL NEERUKONDA INSTITUTE OF TECHNOLOGY AND SCIENCES

A Laboratory Manual

For

SOFTWARE ENGINEERING LAB/MINI PROJECT LAB (CSE 327)

¾ B.Tech- 2nd Semester

PRAGNANAM BRAHMA

ANITS

Prepared by

1. **Mr. P Krishnanjaneyulu, Asst.Prof.**
2. **Mrs. SVSS Lakshmi, Asst.Prof.**
3. **Mrs. Anita T, Asst.Prof.**

DEPARTMENT OF COMPUTER SCIENCE &ENGINEERING

LIST OF EXPERIMENTS

SI.No	Name of the Experiment	CO
1	Identifying Requirements from Problem Statements: 1 week Requirements, Characteristics of Requirements, Categorization of Requirements, Functional Requirements, Identifying Functional Requirements, and Preparing Software Requirements Specifications.	1
2	Estimation of Project Metrics: Project Estimation Techniques, COCOMO, Basic COCOMO Model, Intermediate COCOMO Model, Complete COCOMO Model, Advantages of COCOMO, Drawbacks of COCOMO, Halstead's Complexity Metrics.	2
3	Modeling UML Use Case Diagrams and Capturing Use Case Scenarios: 1 week Use case diagrams, Actor, Use Case, Subject, Graphical Representation, Association between Actors and Use Cases, Use Case Relationships, Include Relationship, Extend Relationship, Generalization Relationship, Identifying Actors, Identifying Use cases, Guidelines for drawing Use Case diagrams.	1

4	Identifying Domain Classes from the Problem Statements: 1 week Domain Class, Traditional Techniques for Identification of Classes, Grammatical Approach Using Nouns, Advantages, Disadvantages, Using Generalization, Using Subclasses, Steps to Identify Domain Classes from Problem Statement, Advanced Concepts.	1
5	Statechart and Activity Modeling: 1 week Statechart Diagrams, Building Blocks of a Statechart Diagram, State, Transition, Action, Guidelines for drawing Statechart Diagrams, Activity Diagrams, Components of an Activity Diagram, Activity, Flow, Decision, Merge, Fork, Join, Note, Partition, A Simple Example, Guidelines for drawing an Activity Diagram.	1
6	Modeling UML Class Diagrams and Sequence Diagrams: 1 week Structural and Behavioral Aspects, Class diagram, Class, Relationships, Sequence diagram, Elements in sequence diagram, Object, Life-line bar, Messages.	1
7	Modeling Data Flow Diagrams: 1 week Data Flow Diagram, Graphical notations for Data Flow Diagram, Symbols used in DFD, Context diagram and leveling DFD.	1
8	Estimation of Test Coverage Metrics and Structural Complexity: 1 week Control Flow Graph, Terminologies, McCabe's Cyclomatic Complexity, Computing Cyclomatic Complexity, Optimum Value of Cyclomatic Complexity, Merits, Demerits.	3
9	Designing Test Suites: 1 week Software Testing, Standards for Software Test Documentation, Testing Frameworks, Need for Software Testing, Test Cases and Test Suite, Types of Software Testing, Unit Testing, Integration Testing, System Testing, Example, Some Remarks.	4

LIST OF INDUSTRY RELEVANT SKILLS

- Software engineering is the study of and practice of engineering to build, design, develop, maintain, and retire software. There are different areas of software engineering and it serves many functions throughout the application lifecycle.

- Effective software engineering requires software engineers to be educated about good software engineering best practices, disciplined and cognizant of how your company develops software, the operation it will fulfill, and how it will be maintained.

- Software engineering is a new era as CIOs and Digital Leaders now understand the importance of software engineering and the impact – both good and bad – it can have on your bottom line.

- Vendors, IT staff, and even departments outside of IT need to be aware that software engineering is increasing in its impact – it is affecting almost all aspects of your daily business.

- Software engineering is important because specific software is needed in almost every industry, in every business, and for every function. It becomes more important as time goes on – if something breaks within your application portfolio, a quick, efficient, and effective fix needs to happen as soon as possible.

GENERAL INSTRUCTIONS

Follow are the steps to be followed in general to perform the experiments in Software Engineering Lab/Mini Project Lab Read the theory about the experiment.

1. View the simulation provided for a chosen, related problem.
2. Take the self evaluation to judge your understanding (optional, but recommended).
3. Solve the given list of exercises.
4. Experiment Specific Instructions.

Following are the instructions specifically for this experiment

1. From the given problem statement, try to figure out if there's any inconsistency with the requirement specification.
2. Also, try to determine what are the functional and non-functional requirements are.
3. Select the check boxes accordingly, and then click on the 'Submit' button.

SOFTWARE ENGINEERING LAB/MINI PROJECT LAB	
Course Code: CSE327	**Credits : 2**

Instruction : 3 Periods/Week	**Sessional Marks : 50**
End Exam : 3 Hours	**End Exam Marks : 50**

RUBRICS

Key Performance Criteria(KPC)	4-Very Good	3-Good	2-Fair	1-Need to improve
Understanding the Scenario/Project/Problem/Query(2)	Understand the problem/scenario/project /query exceptionally well with feasibility study.(2)	Understand the problem/scenario/project/query Properly (2)	Understand the problem/scenario/project/query at basic level.(1)	Understand the problem/scenario/project/query partially.(1)
Analyze the Scenario/Project/Problem/Query and design (3)	Proper analysis and marking of all the notations, relations in the design with a neat diagrammatic representation(3)	Proper analysis and marking of all the notations, relations in the design with Understandable Diagrammatic representation(3)	Improper analysis and Ambiguous marking of all the notations, relations in the design (2)	Incomplete analysis without proper diagrammatic representation.(1)
Correctness of the solution(4)	The solution produces appropriate results for all the inputs tested along with test cases(4)	The solution produces correct results for most of the inputs(3)	The solution results the correct answers for some inputs and results wrong answers for some cases(3)	Solution does not produce the appropriate results for the given inputs(2)
Courtesy , Ethics, team work (based on physical observation)(3)	While performing the project/experiment, the student is in proper dress code, always respectful of others, mindful of safety, and leaves the area clean.(3)	While performing the project/experiment, the student is in proper dress code, many times respectful of others, many times mindful of safety, and leaves the area clean only after being reminded.(3)	While performing the project/experiment , the student is in partial dress code, sometimes respectful of others, sometimes mindful of safety, and leaves the area clean only after being reminded.(2)	While performing the project/experiment, the student is not in proper dress code, not respectful of others, not mindful of safety, and leaves the area messy even after being reminded.(1)
Presentation of Case Study/Mini Project(3)	flow of presentation, visible representation of the case study/mini project with excellent communication(3)	flow of presentation, visible representation of the case study/mini project with good	flow of presentation, visible representation of the case study/mini project	flow of presentation, visible representation of the case study/mini project

		communication(2)	communicated moderately (2)	communicated Not up to the mark(1)
viva/contents of observation/record/Documentation(5)	Able to answer more than 75 % of the questions(5)	able to answer 51 % to 75% of the questions(4)	Able to answer 50% of the questions(3)	Unable to answer any question related to the given problem (2)
Preparation/Submission of Documentation/ppt/record (5)	Submitted on time, presented / communicated diagrams, tables, writing style etc. excellently(5)	Submitted on time, presented / communicated neatly the diagrams, tables, writing style etc. (4)	Submitted on time, presented / communicated diagrams, tables, writing style, etc. not so neatly but in a readable manner(3)	Submitted, presented / communicated diagrams, tables, writing style, etc. with ambiguity/not in a readable manner.(2)

Exercise Questions

Experiment 1

1. When is feasibility study done?

○ After requirements specifications have been finalized

○ During the period when requirements specifications are prepared

○ Before the final requirements specifications are done

○ Could be done at eny time

2. A good requirement specification is one, which is

○ Consistent

○ Complete

○ Unambiguous

○ All of the above

3. Requirement specification is done

○ After requirements are determined

○ Before requirements are determined

○ Simultaneously with requirements determination

○ Independent of requirements determination

4. Functional requirements of a system are related to

○ Using the system (by users) to get some meaningful work done

○ How the system functions under different constraints

○ Whether they adhere to the organization policies

5. SRS refers to

○ Software Requirements Specification

○ System Resources Statement

○ Statement of Reliability of System

○ Standard Requirements Statement

6. The main objective behind preparing a SRS is to

○ Let client and developers agree that they understand each other

○ Formally note down the requirements

○ Estimate the cost of development

○ To judge whether the project could be undertaken

Experiment 2:

1. According to the COCOMO model, a project can be categorized into

○ 3 types

○ 5 types

○ 5 types

○ No such categorization

2. In Intermediate COCOMO model, Effort Adjustment Factor (EAF) is derived from the effort multipliers by

○ Adding them

○ Multiplying them

○ Taking their weighted average

○ Considering their maximum

3. Project metrics are estimated during which phase?

○ Feasibility study

○ Planning

○ Design

○ Development

4. According to Halsetad's metrics, program length is given by the

○ Sum of total number of operators and operands

○ Sum of number of unique operators and operands

○ Total number of operators

○ Total number of operands

5. Complete COCOMO considers a software as a

○ Homogeneous system

○ Heterogeneous system

6. Consider you are developing a web application, which would make use of a lot of web services provided by Facebook, Google, Flickr. Would it be wise to make estimates for this project using COCOMO?

○ Yes, of course

○ Not at all

Experiment 3:

1. What does a use case diagram represent?

○ A set of actions

○ Time sequence of statements executed

○ How to use a particular module

○ Dont know

2. In a use case diagram, relationships between different actors are normally shown

○ True

○ False

3. Generalization relationship exists between two use cases when

○ A use case derives from a base use case

○ A use case derives from a base use case and specializes some of its inherited functionality

○ A use case includes functionality of another use case

○ No two use cases can be related

4. A college has an online Library Management System. Who's the primary actor here?

○ Librarian

○ Director of the college

○ Teacher

○ Student

5. Use cases can be used for testing, which includes

○ Validation

○ Verification

○ Both

○ None

Experiment 4:

1. Domain Object Model is used to

○ Build concept

○ Build the databases

○ Construct the logical operations

2. Classes contain the same type objects – here the same type means

○ Coming from the same sources

○ They are of same data type

○ They are using the same attributes and methods

3. List of potential objects can be made from a narrative problem statement by marking the

○ Verbs from the problem statement

○ Adjective from the problem statement

○ Nouns from the problem statement

○ None of the above

4. All the potential objects we get by using noun method must be into the scope of solution space

○ True

○ False

5. What does merging of parallel activities indicate?

○ At least one of the acitivities should be completed before doing the next activity

○ At most one activity could be left over before proceeding with the next activity

○ All the parallel activities must be completed before performing the next activity

○ None of the above.

Experiment 5:

1. What does an entry action of a state indicate?

○ Action performed after the system moves into the given state

○ Action performed before system moves into the given state

○ An optional action performed when system moves into the given state

○ None of the above

2. What does the guard condition depicted over the transition between any two states indicate?

○ A condition that must be true for the transition to happen

○ A condition that must be false for the transition to occur

○ An indicator that this transition should not happen

○ An event that might happen as result of the transition

3. A state can contain one or more sub-state(s) within it

○ True

○ False

4. What does forking of several activities from a synchronization point indicate?

○ All those activities should get executed one after another

○ The activities can be performed in parallel

○ One or more activities could be skipped

5. A decision point in an activity diagram is a control where

○ Multiple parallel activities merge

○ Decision is taken whether a transition should happen

○ A condition is checked and decided which activity should be performed next

○ There is no such control

Experiment 6:

1. A class is a

○ Blueprint

○ Specific instance of an object

○ Category of user requirement

○ None of the above

2. In class diagrams, a class is represented with a

○ Rectangle

○ Human stick figure

○ Ellipse

○ Diamond

3. From a class diagram it is evident that

○ All classes work in isolation

○ Each class is related with every other class

○ Most of the classes are related

○ Class diagram show object interactions

4. Inheritance among classes are represented by a

○ Solid line from the extending to the extended class

○ Line with an unfilled arrow head from the extending to the extended class

○ Line with a filled diamond from the extending to the extended class

○ Dotted line with extend stereotype from the extending to the extended class

5. Private members (or methods) in a class are indicated with a

○ Hash (#) sign

○ Minus (-) sign

○ Plus (+) sign

○ Tilde (~) sign

6. What does a sequence diagram represent?

○ Workflow in the system

○ How classes are related to each other

○ Sequence of events flow to achieve a target

○ Sequence of activities to be performed before moving to next state

7. In UML 2.0 a synchronous message is represented with a

○ Solid arrow with filled arrowhead

○ Solid arrow with empty arrowhead

○ Dotted arrow with filled arrowhead

○ Dotted arrow with empty arrowhead

8. An object can send a synchronous message and multiple asynchronous message in parallel

○ True

○ False

9. In context of Web, which of the following represent an asynchronous message passing?

○ HTTP GET request

○ HTTP POST request

○ AJAX calls

○ All of the above

Experiment 7:

1. A DFD represents

○ Flow of control

○ Flow of data

○ Both the above

○ Neither one

2. Which is not a component of a DFD?

○ Data flow

○ Decision

○ Process

○ Data store

3. How many processes are present in level-0 or context diagram?

○ 0

○ 1

○ 2

○ 3

4. External entities can appear in a DFD

○ At any level

○ Only at level-0

○ Only at level-1

○ Either at level-0 or at level-1

5. Data flow in a DFD is not possible in between

○ Two processes

○ Data store and process

○ External entity and data store

○ Process and external entity.

Experiment 8:

1. What does CFG of a program describe?

○ Sequence of function calls

○ Sequence of statements executed

○ Contents of the stack

● There's nothing called CFG!

2. A set of paths are said to be linearly independent if

○ Each of them is distinct

○ No paths have a common node

○ No two paths have a common node

○ All the paths are pairwise distinct

3. According to McCabe's Cyclomatic complexity, V(G) = E - N + 2. Here, N is

○ No. of statements in the program

○ No. of unique operators used

○ No. of nodes in the CFG

○ No. of edges in the CFG

Experiment 9:

1. Software testing is the process of

○ Demonstrating that errors are not present

○ Establishing confidence that a program does what it is supposed to do

○ Executing a program to show that it is working as per specifications

○ Executing a program with the intent of finding errors

2. Alpha testing is done by

○ Customer

○ Tester

○ Developer

○ All of the above

3. Test suite is

○ Set of test cases

○ Set of inputs

○ Set of outputs

○ None of the above

4. Regression testing is primarily related to

○ Functional testing

○ Data flow testing

○ Development testing

○ Maintenance testing

5. Acceptance testing is done by

○ Developers

○ Customers

○ Testers

○ All of the above

6. Testing the software is basically

○ Verification

○ Validation

○ Verification and Validation

○ None of the above

7. Functionality of a software is tested by

○ White box testing

○ Black box testing

○ Regression testing

○ None of the above

8. Top down approach is used for

○ Development

○ Identification of faults

○ Validation

○ Functional testing

9. Testing of software with actual data and in the actual environment is called

○ Alpha testing

○ Beta testing

○ Regression testing

○ None of the above

10. During the development phase, which of the following testing approach is not adopted

○ Unit testing

○ Bottom up testing

○ Integration testing

○ Acceptance testing

11. Beta testing is carried out by

○ Users

○ Developers

○ Testers

○ All of the above

12. Equivalence class partitioning is related to

○ Structural testing

○ Black box testing

○ Mutation testing

○ All of the above

13. During the software validation:

○ Process is checked

○ Product is checked

○ Developers' performance is evaluated

○ The customer checks the product

14. Software mistakes during coding are known as:

○ Failures

○ Ddefects

○ Bbugs

○ Errors

15. Which is not a functional testing technique ?

○ Boundary value analysis

○ Decission table

○ Regression testing

○ None of the above

Experiment 1

Identifying Requirements from Problem Statements: Requirements, Characteristics of Requirements, Categorization of Requirements, Functional Requirements, Identifying Functional Requirements, Preparing Software Requirements Specifications.

1.1. Introduction

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

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

1.2. 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

Time Required

Around **3.00** hours

1.3. Requirements

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

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

1.4. Characteristics of Requirements

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

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

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

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

1.5. Categorization of Requirements

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

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

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

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

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

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

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

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

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

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

1.6. Functional Requirements

Identifying Functional Requirements

- Given a problem statement, the functional requirements could be identified by focusing on the following points:
- Identify the high level functional requirements simply from the conceptual understanding of the problem. For example, a Library Management System, apart from anything else, should be able to issue and return books.
- Identify the cases where an end user gets some meaningful work done by using the system. For example, in a digital library a user might use the "Search Book" functionality to obtain information about the books of his interest.
- If we consider the system as a black box, there would be some inputs to it, and some output in return. This black box defines the functionalities of the system. For example, to search for a book, user gives title of the book as input and gets the book details and location as the output.
- Any high level requirement identified could have different sub-requirements. For example, "Issue Book" module could behave differently for different class of users, or for a particular user who has issued the book thrice consecutively.

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. IEEE provides a template [iv], also available here, which could be used for this purpose. 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 Software Engineering.

The Library Information System (LIS) is 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.

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. This same thing happens when any registered user types in his password wrong. However, if incorrect password has been provided for three time consecutively, the security question for the user (specified while registering) with an input box to answer it are also shown. If the user can answer the security question correctly, a new password would be sent to his email address. 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. However, a member can reissue any book at most twice, after which he has to return it. Once a book has been successfully reissued, the user account is updated to reflect the information.

In a similar way we can list other functionality offered by the system as well. However, certain features might not be evident directly from the problem system, but which, nevertheless, are required. One such functionality is "User Verification". The LIS should be able to judge between a registered and non-registered member. Most of the functionality would be available to a registered member. 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.

Assessment Week 1 on SRS Preparation

1. When is feasibility study done?

○ After requirements specifications have been finalized

○ During the period when requirements specifications are prepared

○ Before the final requirements specifications are done

○ Could be done at eny time

2. A good requirement specification is one, which is

○ Consistent

○ Complete

○ Unambiguous

○ All of the above

3. Requirement specification is done

○ After requirements are determined

○ Before requirements are determined

○ Simultaneously with requirements determination

○ Independent of requirements determination

4. Functional requirements of a system are related to

○ Using the system (by users) to get some meaningful work done

○ How the system functions under different constraints

○ Whether they adhere to the organization policies

5. SRS refers to

○ Software Requirements Specification

○ System Resources Statement

○ Statement of Reliability of System

○ Standard Requirements Statement

6. The main objective behind preparing a SRS is to

○ Let client and developers agree that they understand each other

○ Formally note down the requirements

○ Estimate the cost of development

○ To judge whether the project could be undertaken

Steps for conducting the experiment

General Instructions

Follow are the steps to be followed in general to perform the experiments in **Software Engineering Lab**

- Read the theory about the experiment.
- View the simulation provided for a chosen, related problem.
- Take the self evaluation to judge your understanding (optional, but recommended).
- Solve the given list of exercises.
- Experiment Specific Instructions.

Following are the instructions specifically for this experiment:

- From the given problem statement, try to figure out if there's any inconsistency with the requirement specification.
- Also, try to determine what are the functional and non-functional requirements are.
- Select the check boxes accordingly, and then click on the 'Submit' button.

Following books and websites have been consulted for this experiment.
You are suggested to go through them for further details.

Bibliography

- Requirements Engineering: A Good Practice Guide, Ian Sommerville, Pete Sawyer, Wiley India Pvt Ltd, 2009
- Fundamentals of Software Engineering, Rajib Mall, Prentice-Hall of India, 3rd Edition, 2009
- Webliography
- Lecture on "System Analysis and Design", NPTEL.
- When Telepathy Won't Do: Requirements Engineering Key Practices.
- Requirements Analysis: Process of requirements gathering and requirement definition.
- IEEE Recommended Practice for Software Requirements Specifications.
- Requirements Trace-ability and Use Cases.

Experiment 2

Estimation of Project Metrics: Project Estimation Techniques, COCOMO, Basic COCOMO Model, Intermediate COCOMO Model, Complete COCOMO Model, Advantages of COCOMO, Drawbacks of COCOMO, Halstead's Complexity Metrics.

2.1. Introduction

After gathering the entire requirements specific to software project usually we need to think about different solution strategy for the project. Expert business analysts are analyzing their benefits and as well as their shortcomings by means of cost, time and resources require to develop it.

In this experiment, we will learn how to estimate cost, effort and duration for a software project, and then select one solution approach which will be found suitable to fulfill the organizational goal.

2.2. 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

Time Required

Around 3.00 hours

Project Estimation Techniques

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

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

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

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

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

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. These product classes typically correspond to application, utility and system programs, respectively. Data processing programs could be considered as application programs. Compilers, linkers, are examples of utility programs. Operating systems, real-time system programs are examples of system programs. One could easily apprehend that it would take much more time and effort to develop an OS than an attendance management system.

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

Organic: A development project is said to be of organic type, if

- The project deals with developing a well understood application
- The development team is small
- The team members have prior experience in working with similar types of projects

Semidetached: A development project can be categorized as semidetached type, if

- The team consists of some experienced as well as inexperienced staff
- Team members may have some experience on the type of system to be developed

Embedded: Embedded type of development project are those, which

- Aims to develop a software strongly related to machine hardware
- Team size is usually large

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

Basic COCOMO Model

The basic COCOMO model helps to obtain a rough estimate of the project parameters. It estimates effort and time required for development in the following way: Effort = a * (KDSI)b PMTdev = 2.5 * (Effort)c Monthswhere

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. Thus, any project that makes use of modern programming practices would have lower estimates in terms of effort and cost. Each of the 15 such attributes can be rated on a six-point scale ranging from "very low" to "extra high" in their relative order of importance. Each attribute has an effort multiplier fixed as per the rating. The product of effort multipliers of all the 15 attributes gives the Effort Adjustment Factor (EAF).

Cost drivers for INtermediate COCOMO (Source: http://en.wikipedia.org/wiki/COCOMO)

	Ratings					
Cost Drivers	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	

Cost drivers for INtermediate COCOMO (Source: http://en.wikipedia.org/wiki/COCOMO)

	Ratings					
Cost Drivers	Very Low	Low	Nominal	High	Very High	Extra High
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. (One might not even develop all the sub-systems -- just use the available services). The complete COCOMO model takes these factors into account to provide a far more accurate estimate of project metrics.

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

Advantages of COCOMO

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

Drawbacks of COCOMO

This is not a proper measure of a program’s size. Indeed, estimating the size of a software is a difficult task, and any slight miscalculation could cause a large deviation in subsequent project estimates. Moreover, COCOMO was proposed in 1981 keeping the waterfall model of project life cycle in mind. It fails to address other popular approaches like prototype, incremental, spiral, agile models. Moreover, in present day a software project may not necessarily consist of coding of every bit of functionality. Rather, existing software components are often used and glued together towards the development of a new software. COCOMO is not suitable in such cases.

Using Basic COCOMO model to estimate project parameters

Use the simulator on the right hand side to understand how project type and size affects the different parameters estimated.

Quick glance at the formulae:

Effort: a * (Size)b person-month

Time for development: 2.5 * (Effort)c month

Drag the slider to change the project size. Note: select the nearest discrete value corresponding to the actual size.

Top of Form

Project Type	a	b	c
Organic ▾	2.4	1.05	0.38
Project size (in KLOC)	2		
Effort (in PM)	4.97		
Tdev (in month)	4.6		

Project Type	a	b	c
# of developers	2		

Bottom of Form

As evident from the simulation parameters, size of a semi-detached project is larger than that of an organic project, and size of an embedded project is larger than that of a semi-detached, and thereby affecting factors like effort and development time.

Case Study

A Library Information System

Library Information System provides 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 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. It is planned that the current project will be undertaken by a small team consisting of one expert and few engineers. Actual team composition would be determined in a later stage.

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.

	Ratings					
Cost Drivers	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		

	Ratings					
Cost Drivers	Very Low	Low	Nominal	High	Very High	Extra High
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.

Assessment for Exp. 2

1. According to the COCOMO model, a project can be categorized into

○ 3 types

○ 5 types

○ 5 types

○ No such categorization

2. In Intermediate COCOMO model, Effort Adjustment Factor (EAF) is derived from the effort multipliers by

○ Adding them

○ Multiplying them

○ Taking their weighted average

○ Considering their maximum

3. Project metrics are estimated during which phase?

○ Feasibility study

○ Planning

○ Design

○ Development

4. According to Halsetad's metrics, program length is given by the

○ Sum of total number of operators and operands

○ Sum of number of unique operators and operands

○ Total number of operators

○ Total number of operands

5. Complete COCOMO considers a software as a

○ Homogeneous system

○ Heterogeneous system

6. Consider you are developing a web application, which would make use of a lot of web services provided by Facebook, Google, Flickr. Would it be wise to make estimates for this project using COCOMO?

○ Yes, of course

○ Not at all

Steps for conducting the experiment

General Instructions

Follow are the steps to be followed in general to perform the experiments in Software Engineering Lab.

- Read the theory about the experiment
- View the simulation provided for a chosen, related problem
- Take the self evaluation to judge your understanding (optional, but recommended)
- Solve the given list of exercises
- Experiment Specific Instructions

Following are the instructions specifically for this experiment:

- The left hand side of the 'Exercises' page will present the problem
- The right hand side of the page asks to evaluate certain parameters
- The values of the parameters are to be typed in in the adjoining text boxes
- If all the values entered are correct, then the solution is correct. Otherwise user will be indicated where the error has occurred.

Following books and websites have been consulted for this experiment. You are suggested to go through them for further details.

Bibliography

- Fundamentals of Software Engineering, Rajib Mall, Prentice-Hall of India, 3rd Edition, 2009
- Software Engineering, Ian Sommerville, Addison Wesley Longman, 9th Edition, March 2010

Webliography

- COCOMO (Constructive Cost Model), Seminar on Software Cost Estimation WS 2002 / 2003, presented by Nancy Merlo – Schett.
- Software Engineering, National Program on Technology Enhanced Learning.
- Halstead Metrics, Verifysoft Technology.

Experiment 3

Modeling UML Use Case Diagrams and Capturing Use Case Scenarios: Use case diagrams, Actor, Use Case, Subject, Graphical Representation, Association between Actors and Use Cases, Use Case Relationships, Include Relationship, Extend Relationship, Generalization Relationship, Identifying Actors, Identifying Use cases, Guidelines for drawing Use Case diagrams.

Introduction

Use case diagram is a platform that can provide a common understanding for the end-users, developers and the domain experts. It is used to capture the basic functionality i.e. use cases, and the users of those available functionality, i.e. actors, from a given problem statement.

In this experiment, we will learn how use cases and actors can be captured and how different use cases are related in a system.

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

Time required around 3.00 hours

Use case diagrams

Use case diagrams belong to the category of behavioral 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.

For example, consider the case where a customer *withdraws cash* from an ATM. Here, customer is a human actor.

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. For example, a customer uses an ATM to withdraw cash when he needs it. A customer is the primary actor here.

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 include, ***check balance, change PIN,*** **and so on.**

Use cases include both successful and unsuccessful scenarios of user interactions with the system. For example, authentication of a customer by the ATM would fail if he enters wrong PIN. In such case, an error message is displayed on the screen of the ATM.

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.

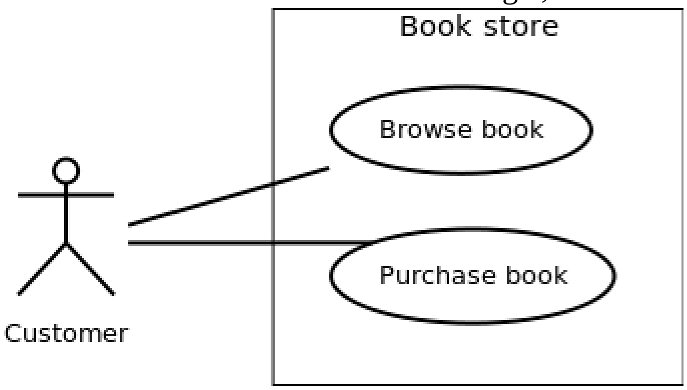


Figure - 01: A use case diagram for a book store

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. Associations 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.

Example

For example, consider an email application. A user can send a new mail, reply to an email he has received, or forward an email. However, in each of these three cases, the user must be logged in to perform those actions. Thus, we could have a *login* use case, which is included by *compose mail*, *reply*, and *forward email* use cases. The relationship is shown in figure - 02.

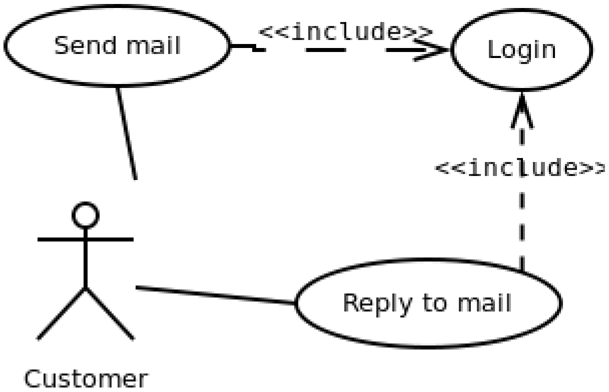


Figure - 02: Include relationship between use cases

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 [iv, v].

Example

Let's consider an online bookstore. The system allows an authenticated user to buy selected book(s). While the order is being placed, the system also allows specifying any special shipping instructions [vii], for example, call the customer before delivery. This *Shipping Instructions* step is optional, and not a part of the main *Place Order* use case. Figure - 03 depicts such relationship.

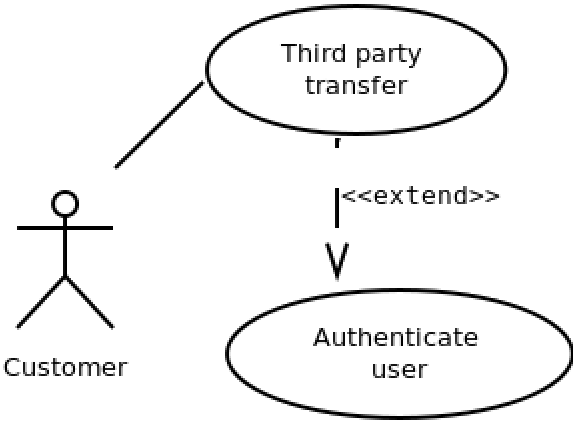


Figure - 03: Extend relationship between use cases

Notation

Extend relationship is depicted by a dashed arrow with a «extend» stereotype from the extending use case to the extended use case.

Generalization Relationship

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

Example

To illustrate this, consider a graphical application that allows users to draw polygons. We could have a use case *draw polygon*. Now, rectangle is a particular instance of polygon having four sides at right angles to each other. So, the use case *draw rectangle* inherits the properties of the use case *draw polygon* and overrides it's drawing method. This is an example of generalization relationship. Similarly, a generalization relationship exists between *draw rectangle* and *draw square* use cases. The relationship has been illustrated in figure - 04.

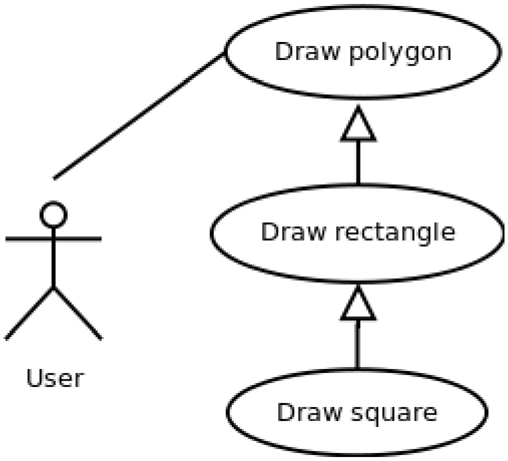


Figure - 04: Generalization relationship among use cases

Notation

Generalization relationship is depicted by a solid arrow from the specialized (derived) use case to the more generalized (base) use case.

Identifying Actors

- Given a problem statement, the actors could be identified by asking the following questions
- Who gets most of the benefits from the system? (The answer would lead to the identification of the primary actor).
- Who keeps the system working? (This will help to identify a list of potential users)
- What other software / hardware does the system interact with?
- Any interface (interaction) between the concerned system and any other system?

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".

Guidelines for drawing Use Case diagrams

Following general guidelines could be kept in mind while trying to draw a use case diagram

- Determine the system boundary
- Ensure that individual actors have well-defined purpose
- Use cases identified should let some meaningful work done by the actors
- Associate the actors and use cases -- there shouldn't be any actor or use case floating without any connection
- Use include relationship to encapsulate common behaviour among use cases , if any

Test Case:

From the following problem statement, identify the actors and use cases

An automated teller machine (ATM) lets customers to withdraw cash anytime from anywhere without requiring involvement of any banking clerk or representative. Customer must insert his ATM card into he machine and authenticate himself by typing in his personal identification number (PIN). He cannot avail any of the facilities if the PIN entered is wrong. Authenticated customers can also change their PIN. They can deposit cash to their account with the bank. Also they can transfer funds to any other account. The ATM also provides options to the user to pay electricity or phone bill. Everyday morning the stock of cash in the ATM machine is replenished by a representative from the bank. Also, if the machine stops working, then it is fixed by a maintenance guy.

For a given problem at first we identify the actors. This can be by following the steps as mentioned in the theory section. Once actors have been identified, we they make use of the system and thus, identify the use cases.

Case Study

1: A Library Information System for Institute

The 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.

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 table 1. For each such requirement, 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	User types in user ID User types in password User clicks on the 'Login' button 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 its various features. If the user provides a correct pair of (<user_id>, <password>) then he can access his home page. However, if login credentials are incorrect, an error message is displayed to him. Figure 1 shows its pictorial representation.

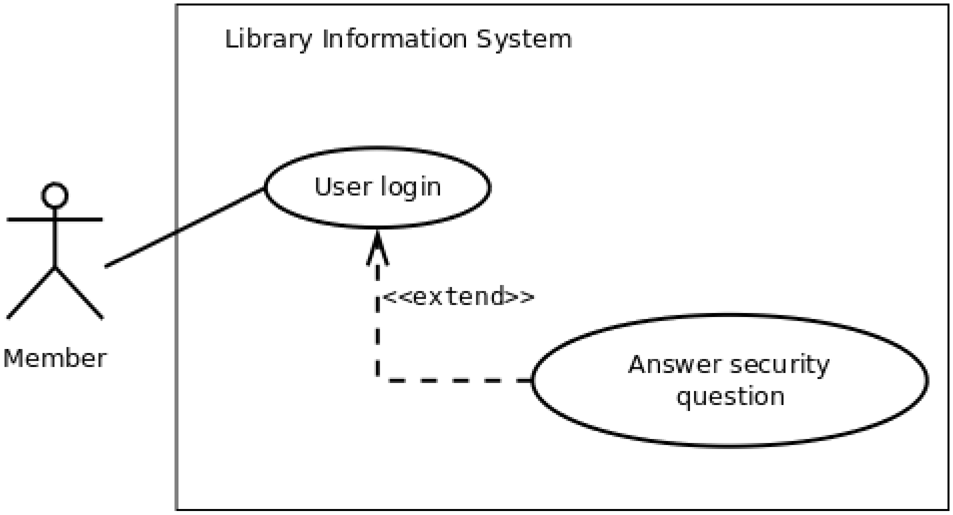


Figure 1: Use case diagram showing "New user registration" use case

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, when a member is trying to login, and provides incorrect (<user_id>, <password>) for three consecutive times, he is asked the security question that was set during registration. If user can answer the question correctly, the password is send to his email address. However, if the user fails to answer the security question correctly, his account is temporarily blocked. A detail 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 detail 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	User is logged in The book is available User's account has not exceeded the limit of maximum books that can be issued
Actors	Member (primary) Library staff
Steps	User logs in User searches for a book User clicks on "Issue" button to issue the book User's account is updated 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 are shown in figure 2 by the «include» relationship among the use cases. The maximum # of books that can be issued to a user depends on whether he is a student or a professor. So, "Verify issue count" is a general use case, which has been specialized by "Verify student issue count" and "Verify professor issue count" use cases. These have been represented by the "generalization" relationship in figure 2.

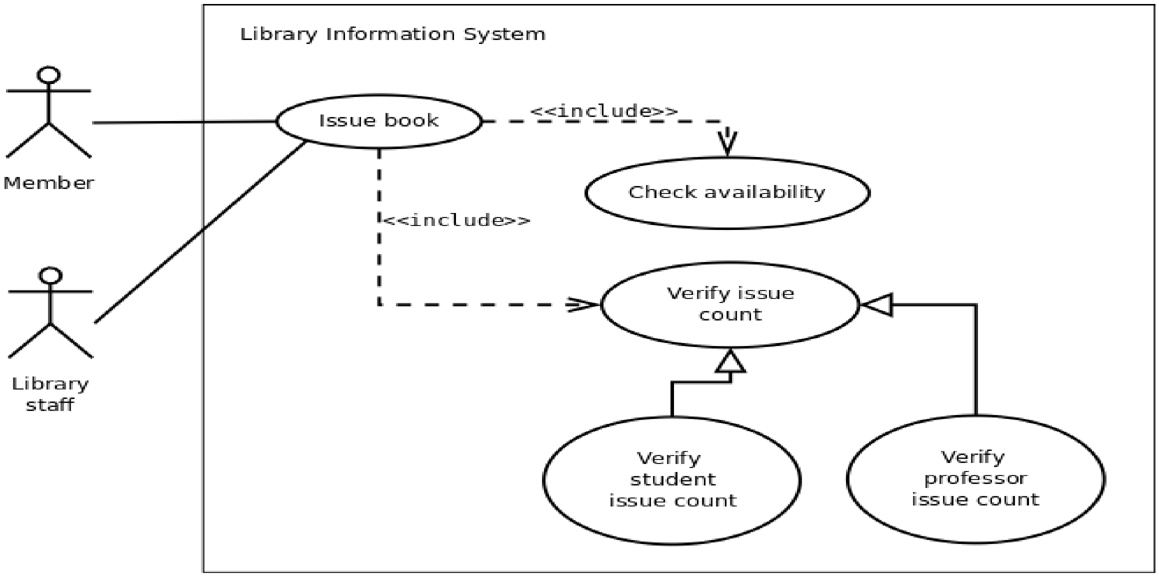


Figure 2: Use case diagram showing "Issue book" use case

In the above scenario "Member" is the primary actor who triggers the "Issue book" use case. "Library staff" is a secondary actor here.

Practical / Viva Questions

1. What does a use case diagram represent?

○ A set of actions

○ Time sequence of statements executed

○ How to use a particular module

○ Dont know

2. In a use case diagram, relationships between different actors are normally shown

○ True

○ False

3. Generalization relationship exists between two use cases when

○ A use case derives from a base use case

○ A use case derives from a base use case and specializes some of its inherited functionality

○ A use case includes functionality of another use case

○ No two use cases can be related

4. A college has an online Library Management System. Who's the primary actor here?

○ Librarian

○ Director of the college

○ Teacher

○ Student

5. Use cases can be used for testing, which includes

○ Validation

○ Verification

○ Both

○ None

Steps for conducting the experiment

General Instructions:

Follow are the steps to be followed in general to perform the experiments in Software Engineering Lab.

- Read the theory about the experiment
- View the simulation provided for a chosen, related problem
- Take the self evaluation to judge your understanding (optional, but recommended)
- Solve the given list of exercises
- Experiment Specific Instructions

Following are the instructions specifically for this experiment:

- Identify an actor, and write its name in the left textbox of **'Table #1'**. Click the adjacent **'Add'** button to add this actor. Repeat this for all the possible actors.
- Identify a use case, and write its name in the left textbox of **'Table #2'**. Click the adjacent **'Add'** button to add this use case. Repeat this for all the possible use cases.
- If you want to delete any actor or use case, go to the **'Table #4'**, remove the actor or use case.
- **'Table #3'** lets you define relationships between any two components. Select the first actor/use case from the first dropdown list, the second from the third drop down list. Select one relationship from the second dropdown list. If you want to put any label to the relation, write the text for the label to the adjacent textbox. Click on the **'Add'** button at the right side to add this relation. Repeat this for all the possible relations.
- Relationships so defined will be displayed in the **'Table #5'**. Here, you have the option to remove a wrongly defined relationship.

- Give the name of actors and use cases in **'Table #1'** and **'Table #2'** respectively by using alphabets, numerics and white-space only.
- Write the text of the label for relationships in **'Table #3'** using alphabets, numerics and white-space only.

Following books and websites have been consulted for this experiment. You are suggested to go through them for further details.

Bibliography

- Object-Oriented Modeling and Design with UML, Michael Blaha, James Rumbaugh, Prentice-Hall of India, 2nd Edition
- Object-Oriented Analysis and Design using UML, Mahesh P. Matha, Prentice-Hall of India,

Webliography

- Use Case Diagrams
- Use case diagram -- Wikipedia
- Unified Modeling Language, Superstructure, V2.1.2
- "Functional Requirements and Use Cases", Ruth Malan and Dana Bredemeyer, Bredemeyer Consulting
- "A Use Case Template: draft for discussion", Derek Coleman, Hewlett-Packard Software Initiative
- Extend relationships
- Requirements Trace-ability and Use Cases
- UML Use Case Diagrams: Tips and FAQ

Experiment 4

Identifying Domain Classes from the Problem Statements: Domain Class, Traditional Techniques for Identification of Classes, Grammatical Approach Using Nouns, Advantages, Disadvantages, Using Generalization, Using Subclasses, Steps to Identify Domain Classes from Problem Statement, Advanced Concepts.

Introduction

Same types of objects are typically implemented by class in object oriented programming. As the structural unit of the system can be represented through the classes, so, it is very important to identify the classes before start implementing all the logical flows of the system.

In this experiment we will learn how to identify the classes from a given problem statement.

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.
- Time Required Around 3.00 hours.

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. As stated in, domain classes are used for representing business activities during the analysis phase.

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:

- Obtain the user requirements (problem statement) as a simple, descriptive English text. This basically corresponds to the use-case diagram for the problem statement.

- Identify and mark the nouns, pronouns and noun phrases from the above problem statements.
- 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.
- Make a list of potential objects by finding out the nouns and noun phrases from narrative problem statement
- Apply subject matter expertise (or domain knowledge) to identify additional classes
- Filter out the redundant or irrelevant classes
- 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

- Group the objects based on similar attributes. While grouping we should remember that
- Different nouns (or noun phrases) can actually refer to the same thing (examples: house, home, abode).
- 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)
- Give related names to each group to generate the final list of top level classes
- Iterate over to refine the list of classes

Advanced Concepts

Identification of domain classes might not be a simple task for novices. It requires expertise and domain knowledge to identify business classes from plain English text. The concepts presented here have been kept simple in order to make a student familiarize with the subject. A lot of work has been done in this area, and various techniques have been proposed to identify domain classes. Interested readers may look at the following paper for an advanced treatment on this subject matter.

From the following problem statement, identify the domain classes

An automated teller machine (ATM) lets customers to withdraw cash anytime from anywhere without requiring involvement of any banking clerk or representative. Customer must insert his ATM card into the machine and authenticate himself by typing in his personal identification number (PIN). He cannot avail any of the facilities if the PIN entered is wrong. Authenticated customers can also change their PIN. They can deposit cash to their account with the bank. Also they can transfer funds to any other account. The ATM also provides options to the user to pay electricity or phone bill. Everyday morning the stock of cash in the ATM machine is replenished by a representative from the bank. Also, if the machine stops working, then it is fixed by a maintenance guy.

Case Study

A Library Information System for Institute

The 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.

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

- The Institute
- Software Engineering
- Research scholars
- Students
- Professors
- Employees
- Projects
- Institution
- Library Information System
- Members
- Book
- Desk
- Chamber
- System
- Library staff
- Librarian
- Transactions
- Record
- Shelf
- Non-member
- Web application
- 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 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. For example, 'Chamber' is not something related to the Library Information System. And so are 'Projects', 'Desk', 'Shelf'. In a similar way, 'Software

Engineering', 'Web application', 'Software' doesn't seem to be potential classes in this context. 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. In fact, 'Librarian' and 'Member' are some specialized category of the class 'Employee'. (This considers a student is also an "employee".) The above identified conceptual classes pave the way for modeling of design and implementation classes.

Viva Questions

1. Domain Object Model is used to

○ Build concept

○ Build the databases

○ Construct the logical operations

2. Classes contain the same type objects – here the same type means

○ Coming from the same sources

○ They are of same data type

○ They are using the same attributes and methods.

3. List of potential objects can be made from a narrative problem statement by marking the

○ Verbs from the problem statement

○ Adjective from the problem statement

○ Nouns from the problem statement

○ None of the above

4. All the potential objects we get by using noun method must be into the scope of solution space

○ True

○ False

5. What does merging of parallel activities indicate?

○ At least one of the acitivities should be completed before doing the next activity

○ At most one activity could be left over before proceeding with the next activity

○ All the parallel activities must be completed before performing the next activity

○ None of the above

Steps for conducting the experiment

General Instructions

Follow are the steps to be followed in general to perform the experiments in Software Engineering Lab.

- Read the theory about the experiment
- View the simulation provided for a chosen, related problem
- Take the self evaluation to judge your understanding (optional, but recommended)
- Solve the given list of exercises

Experiment Specific Instructions

Following are the instructions specifically for this experiment:

- Identify a noun or noun phrase, and write it in the left textbox of **Table #1**. Click the adjacent **'Add'** button to add this noun or noun phrase. Repeat this until all the nouns and noun phrases are taken from the problem statement.
- If you want to delete any noun phrase, you can do it from the left list of **Table #2**.
- In **Table #2**, we will categorize the nouns which are taken as Potential objects in our solution.
- All the potential objects under their selected category are shown in **Table #3**.
- You may add your own attribute from **Table #4** to the attribute list of **Table #4** if it is required.
- If you want to remove/modify an attribute what you have added, you can do it from the attribute list of **Table #4**.
- **Table #5** shows all the objects with its related attributes.

- If you want to remove/modify any attribute/object shown in **Table #6**, you can do it easily from the same table and can re-insert the elements accordingly.
- **Table #7** is to define the classes based on common attributes it may have. Give a class name in the middle textbox. Click the adjacent **'Add'** button.
- **Table #8** shows all the classes based on some common attributes which are taken from **Table #7**.
- If you want to remove/modify any attribute/class shown in **Table #8**, you can do it easily from the same table and can re-insert the elements accordingly.
- You are allowed to decide some classes on basis of your knowledge gaining through your work with the related problem domain.
- **Table #9** shown you all the domain classes for the given problem statement.

Note:

- Give the name of attributes and classes by using alphabets numeric and whitespace only.

Exercise Problem:

Identify the domain classes from the following problem statement

The latest cab services agency in the city has approached you to develop a Cab Management System for them. Following is the information they have given to implement the system.

Mr. Bose is the boss of this agency. Cabs are solely owned by the agency. They hire drivers to drive the cabs. Most of the cabs are without AC. However, a few comes with AC.

The agency provides service from 8 AM to 8 PM. Presently the service is limited only within Kolkata. Whenever any passenger books a cab, an available cab is allocated for him. A booking receipt is given to the passenger. He is then dropped to his home, office, or wherever he wants to go. In case the place is in too interior, the passenger is dropped at the nearest landmark.

Payments are made to the drivers by cheque drawn at the local branch of At Your Risk Bank. All kind of finances required for the business are dealt with this bank.

Recently Mr. Roy, neighbour of Mr. Bose, has given a proposal to book one of the cab in the morning everyday to drop his son to school, and drop him back to home later. Few other persons in the locality have also found the plan a good one. Hence, Mr. Bose is planning to introduce this "Drop to school" plan also very soon.

Learning Objectives:

- Identifying potential classes (and their attributes) from a given problem statement.
- Use expert knowledge on the subject matter to identify other relevant classes.

Following books and websites have been consulted for this experiment. You are suggested to go through them for further details.

Bibliography

- UML and C++ A Practical Guide To Object-Oriented Development, Richard C. Lee, William M. Tepfenhart, Prentice-Hall of India, 2nd Edition, 2005.
- Fundamentals of Software Engineering, Rajib Mall, Prentice-Hall of India, 3rd Edition, 2009.

Webliography

- Domain Analysis
- Domain Analysis Using Textual Analysis Approach
- Domain model
- Business Modeling – The Domain Model

Experiment 5

Statechart and Activity Modeling: Statechart Diagrams, Building Blocks of a Statechart Diagram, State, Transition, Action, Guidelines for drawing Statechart Diagrams, Activity Diagrams, Components of an Activity Diagram, Activity, Flow, Decision, Merge, Fork, Join, Note, Partition, A Simple Example, Guidelines for drawing an Activity Diagram.

Introduction

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

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

Objectives

After completing this experiment you will be able to:

- Identify the distinct states a system has.
- 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.
- Time Required: Around 3.00 hours.

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.

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

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

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 remains 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.

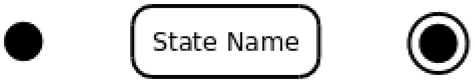


Figure-01: Representation of initial, intermediate, and final states of a statechart diagram. Intermediate states usually have two compartments, separated by a horizontal line, called the name compartment and internal transitions compartment.

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 remains in this state or until the action expression results in a completed computation.
- **Include**: Indicates invocation of a sub-machine.

Any other action label identifies 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

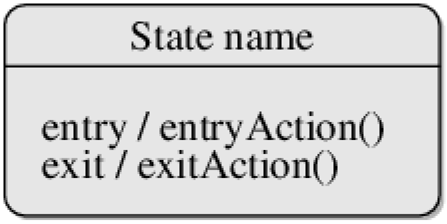


Figure-02: A typical state in a statechart diagram.

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 [iii]
- **Guard-condition** is (are) precondition(s), which must be true for the transition to happen [optional]
- **Action-expression** indicate action(s) to be performed as a result of the transition [optional]

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

Action

Actions represent 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:

- Exit actions of the present state
- Actions specified for the transition
- Entry actions of the next state.

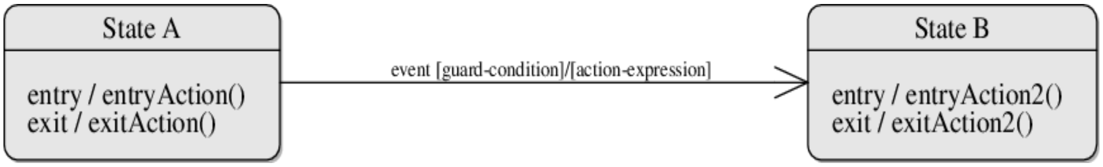


Figure-03: A statechart diagram showing transition from state A to B

Guidelines for drawing Statechart Diagrams

Following steps could be followed, as suggested 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. However, it is more powerful than a simple flow chart since it can represent various other concepts like concurrent activities, their joining.

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

Below we describe the building blocks 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. 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 types of activity nodes: initial and final. They are represented with a filled circle, and a filled in circle with a border respectively (table-01). Initial node represents the starting point of a flow in an activity diagram. There could be multiple initial nodes, which mean 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. Any transition reaching a final node would stop all activities.

Flow

A flow (also termed as edge, or transition) 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. The syntax to depict it is [guard condition].

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. However, they can be omitted in obvious cases. The input edge could also have guard 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. For example, a sales order is closed only when the customer has receive the product, **and** the sales company has received it's payment.

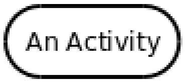
Note

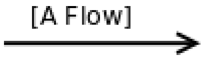
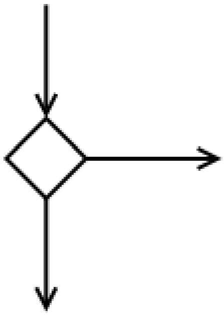
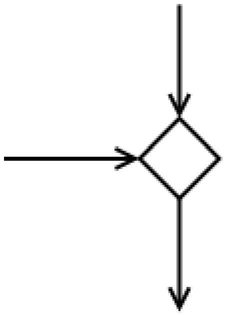
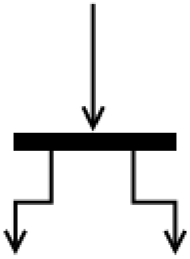
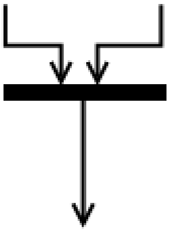
UML allows attaching a note to different components of a diagram to present some textual information. The information could simply be a comment or may be some constraint. A note can be attached to a decision point, for example, to indicate the branching criteria.

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	

Component	Graphical Notation
Flow	
Decision	
Merge	
Fork	
Join	

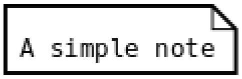
Component	Graphical Notation
Note	

Table-01: Typical components used in an activity diagram

Apart from the above stated components, there are few other components as well (representing events, sending of signals, nested activity diagrams), which won't be discussed here. **A Simple Example**

Figure-04 shows a simple activity diagram with two activities. The figure depicts two stages of a form submission. At first a form is filled up with relevant and correct information. Once it is verified that there is no error in the form, it is then submitted. The two other symbols shown in the figure are the initial node (dark filled circle), and final node (outer hollow circle with inner filled circle). It may be noted that there could be zero or more final node(s) in an activity diagram.

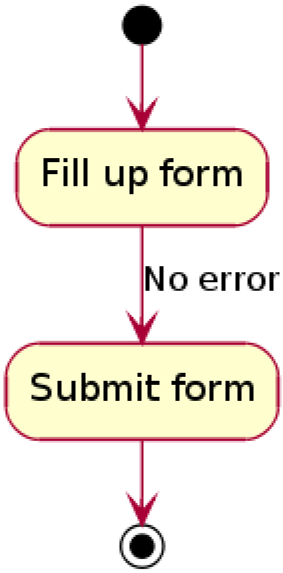


Figure-04: A simple activity diagram.

Guidelines for drawing an Activity Diagram

The following general guidelines could be followed to pictorially represent a complex logic.

- Identify tiny pieces of work being performed by the system
- Identify the next logical activity that should be performed
- Think about all those conditions that should be made, and all those constraints that should be satisfied, before one can move to the next activity

Put non-trivial guard conditions on the edges to avoid confusion

Case Study

A Library Information System for Institute

The 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.

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.

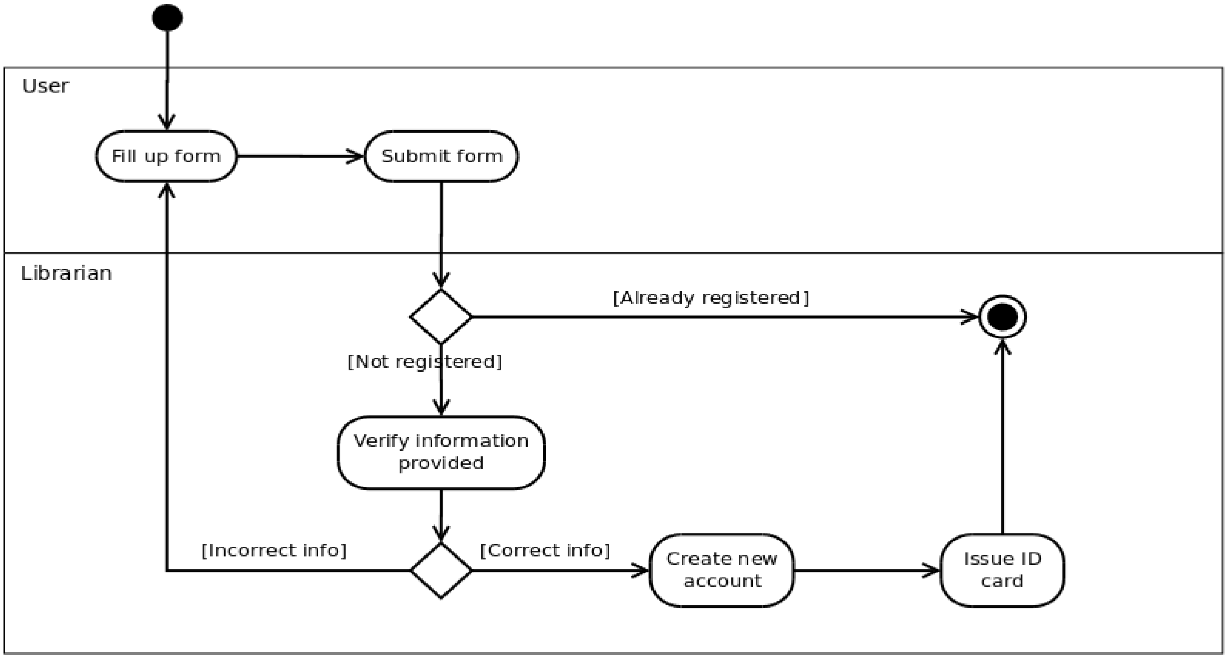


Figure-01: Activity diagram for new user registration

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, possibly against the central database used by the institution. If all information have been provided correctly, librarian goes on with creating a new account for the user. Otherwise, the user is asked to provide all and correct information in his (her) registration form. 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 tries to depict the workflow involved while issuing books.

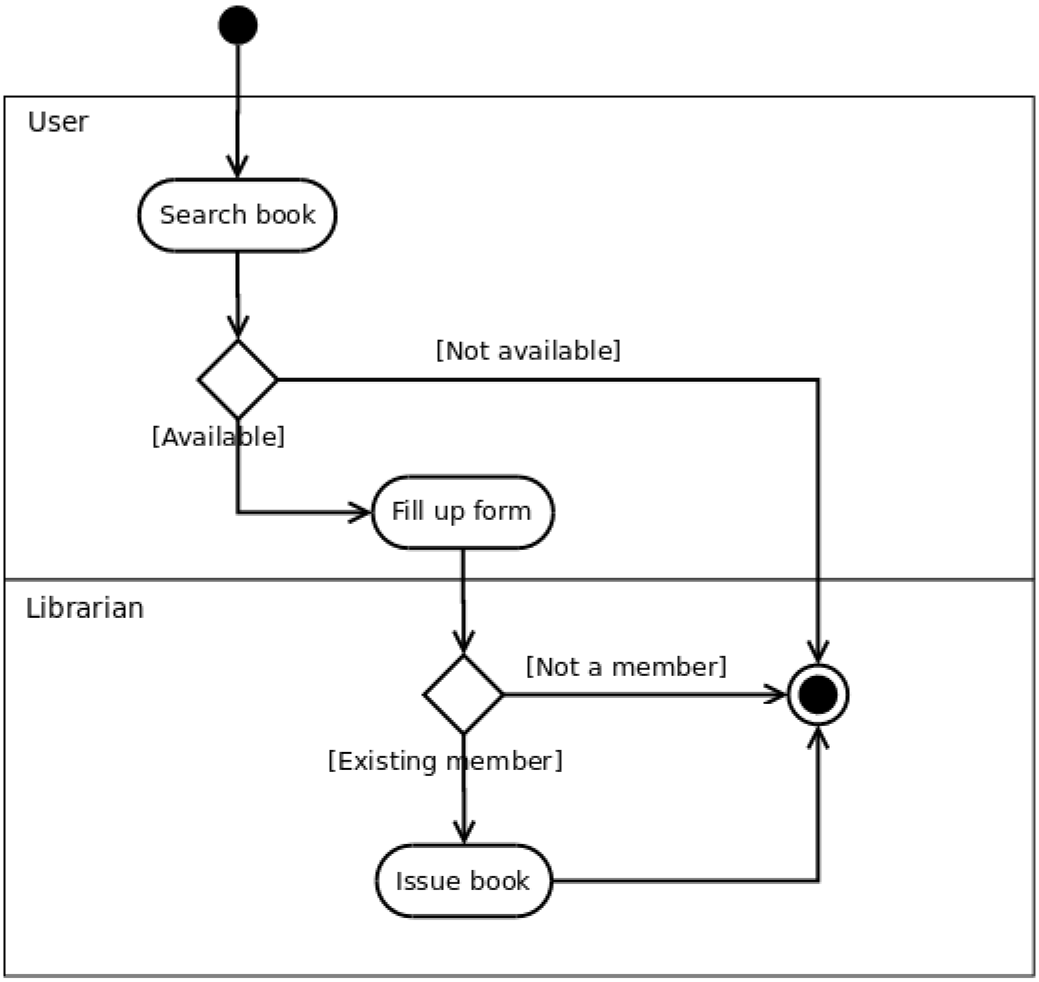


Figure-02: Activity diagram for issuing books

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.

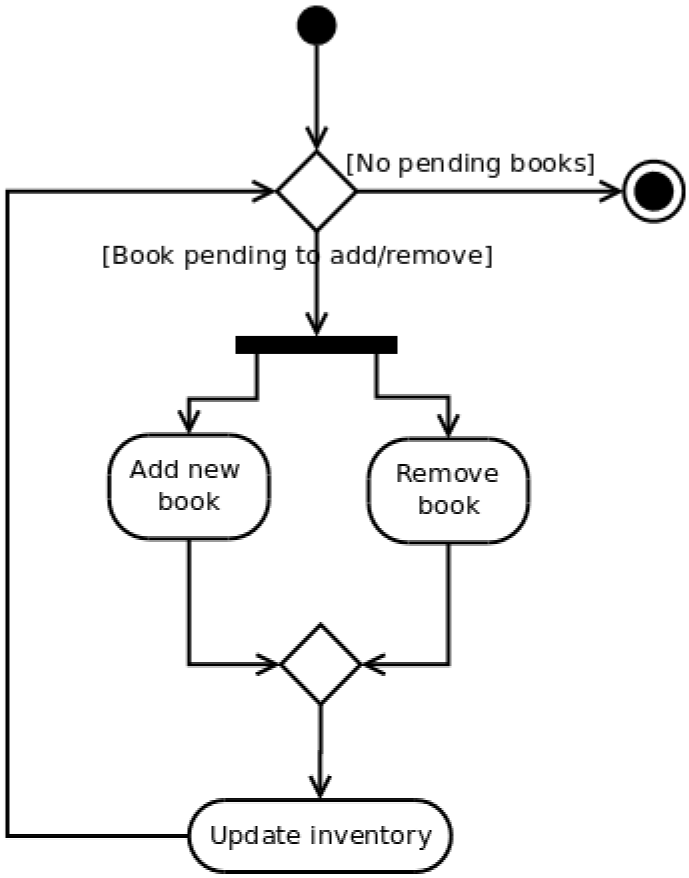


Figure-03: Activity diagram for updating inventory

Addition of new books and removing records of books taken off from the shelves could be done parallel. 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. That is, a few books could be added, and then update the database, then again continues with the tasks.

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

Viva Questions

1. What does an entry action of a state indicate?

○ Action performed after the system moves into the given state

○ Action performed before system moves into the given state

○ An optional action performed when system moves into the given state.

2. What does the guard condition depicted over the transition between any two states indicate?

○ A condition that must be true for the transition to happen

○ A condition that must be false for the transition to occur

○ An indicator that this transition should not happen

○ An event that might happen as result of the transition

3. A state can contain one or more sub-state(s) within it

○ True

○ False

4. What does forking of several activities from a synchronization point indicate?

○ All those activities should get executed one after another

○ The activities can be performed in parallel

○ One or more activities could be skipped

5. A decision point in an activity diagram is a control where

○ Multiple parallel activities merge

○ Decision is taken whether a transition should happen

○ A condition is checked and decided which activity should be performed next

○ There is no such control.

General Instructions

Follow are the steps to be followed in general to perform the experiments in Software Engineering Lab.

- Read the theory about the experiment.
- View the simulation provided for a chosen, related problem.
- Take the self evaluation to judge your understanding (optional, but recommended).
- Solve the given list of exercises.

Case study

A Library Information System for Institute

The 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.

Think about these points:

- What are the different states of case study?
- What activities are performed in each state?
- What action does make the move from one state to another?

Learning Objectives:

- Identifying different states of a system.
- Identifying activities performed in each state.

Webliography

- UML 2 State Machine Diagrams.
- Modeling Behavior with UML Interactions and Statecharts.
- UML 2 State Machine Diagramming Guidelines.
- Statechart Diagram.
- Activity Diagram.
- UML Activity Diagram.
- UML 2 Activity Diagrams.
- Activity Diagrams.

Experiment 6

Modeling UML Class Diagrams and Sequence Diagrams: Structural and Behavioral Aspects, Class diagram, Class, Relationships, Sequence diagram, Elements in sequence diagram, Object, Life-line bar, Messages.

Introduction

Classes are the structural units in object oriented system design approach, so it is essential to know all the relationships that exist between the classes, in a system. All objects in a system are also interacting to each other by means of passing messages from one object to another. Sequence diagram shows these interactions with time ordering of the messages.

- In this Experiment, we will learn about the representation of class diagram and sequence diagram. We also learn about different relationships that exist among the classes, in a system.
- From the experiment of sequence diagram, we will learn about different types of messages passing in between the objects and time ordering of those messages, in a system.

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.

Time Required: Around 3.00 hours.

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.

Example

To build a structural model for an Educational Organization, ‘Course’ can be treated as a class which contains attributes ‘courseName’ & ‘courseID’ with the operations ‘addCourse()’ & ‘removeCourse()’ allowed to be performed for any object to that class.

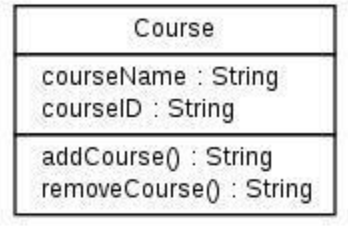


Figure-01: Class Representation.

Generalization/Specialization

It describes how one class is derived from another class. Derived class inherits the properties of its parent class.

Example

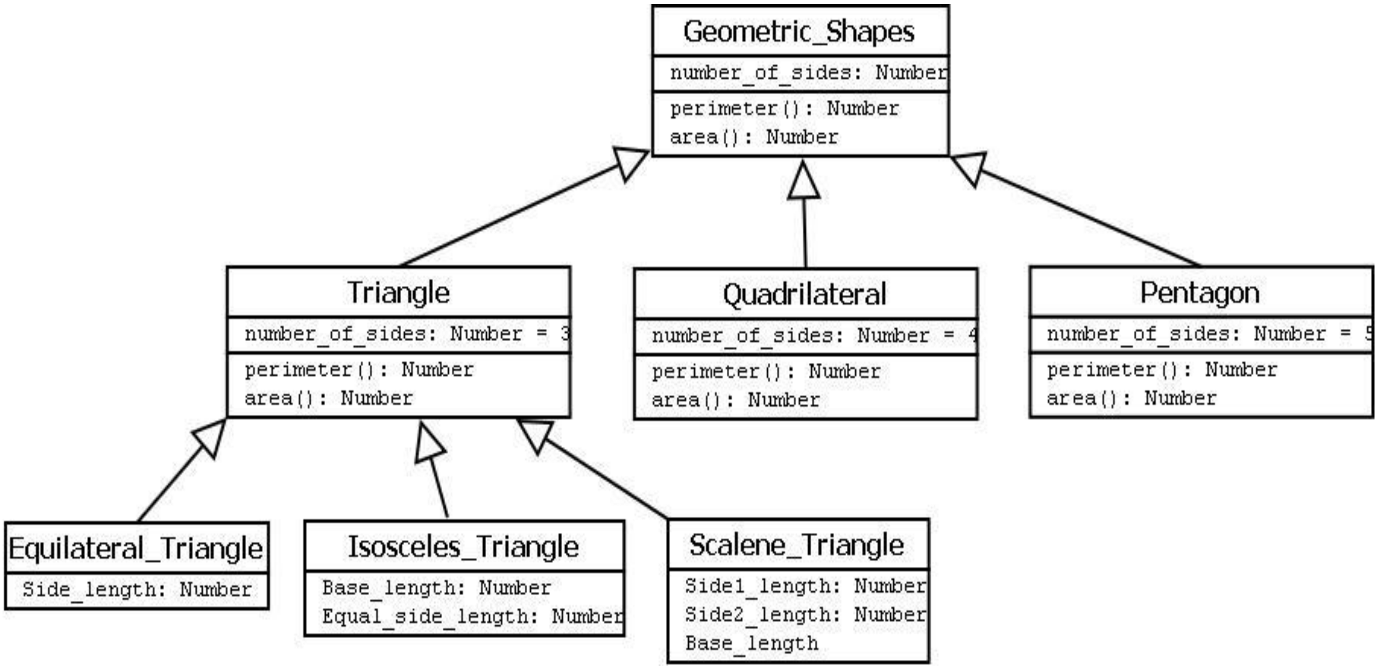


Figure-02:

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.

Example

In structure model for a system of an organization an employee (instance of ‘Employee’ class) is always assigned to a particular department (instance of ‘Department’ class) and the association can be shown by a line connecting the respective classes.

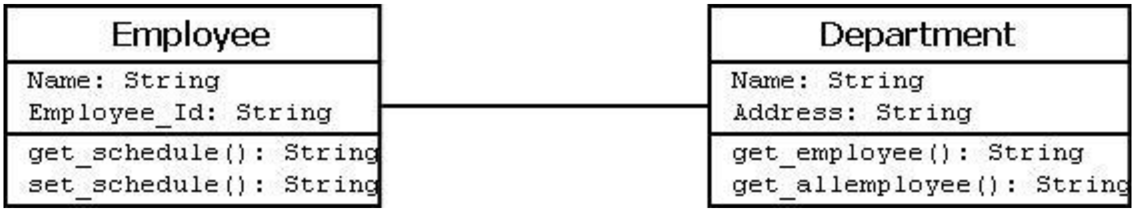


Figure-03:

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.

Example

For a supermarket in a city, each branch runs some of the departments they have. So, the relation among the classes ‘Branch’ and ‘Department’ can be designed as an aggregation. In UML, it can be shown as in the fig. below

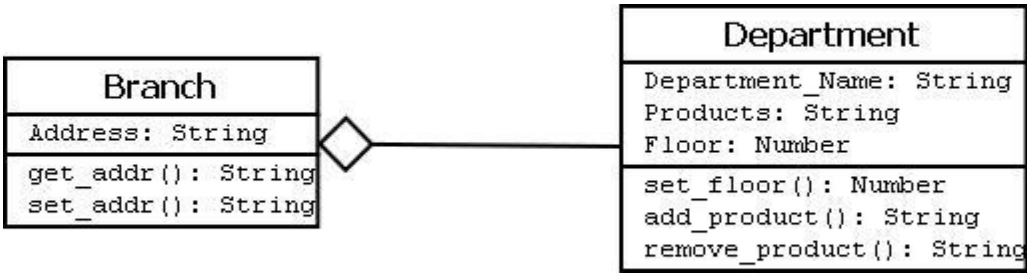


Figure-04:

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.

Example

Let consider a shopping mall has several branches in different locations in a city. The existence of branches completely depends on the shopping mall as if it is not exist any branch of it will no longer exists in the city. This relation can be described as composition and can be shown as below

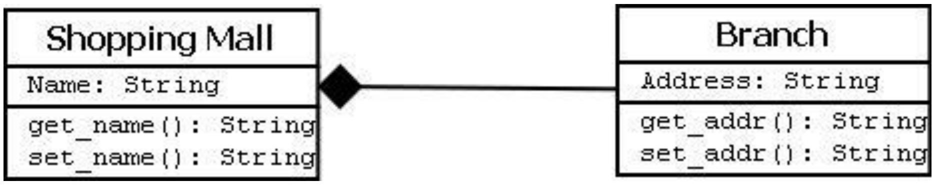
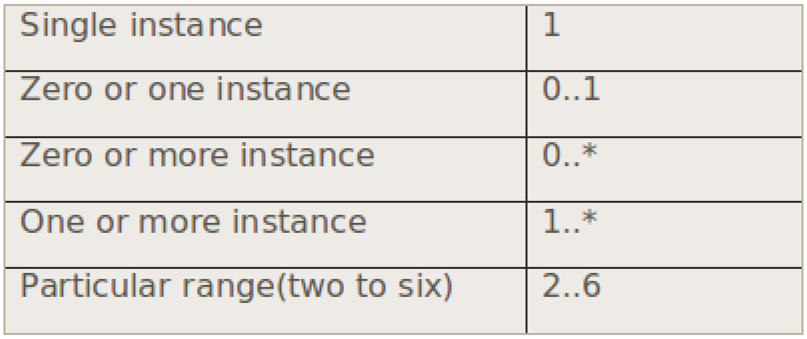


Figure-05:

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:



Single instance	1
Zero or one instance	0..1
Zero or more instance	0..*
One or more instance	1..*
Particular range(two to six)	2..6

Figure-06:

Example

One vehicle may have two or more wheels

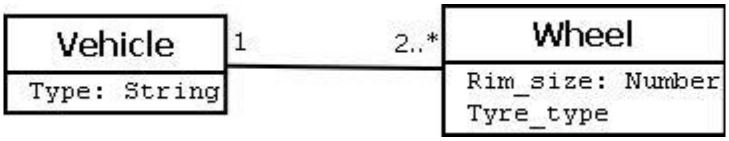


Figure-07:

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

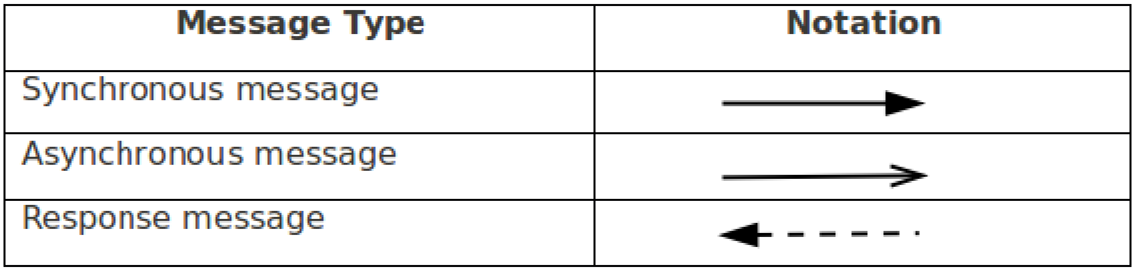
Messages are shown as an arrow from the life-line of sender object to the life-line of receiver object and labeled with the message name. Chronological order of the messages passing throughout the objects' life-line show the sequence in which they occur. There may exist some different types of messages

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. A straight arrow with open arrow-head from sender life-line bar to receiver end, represent an asynchronous message.

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.



Message Type	Notation
Synchronous message	——▶
Asynchronous message	——>
Response message	◀- - - - -

Figure-08

Case Study

A Library Information System 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.

Let us consider the "Issue Book" use case and represent the involved steps in a sequence diagram as shown in figure 1. We assume that the book to be issued is available. 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.

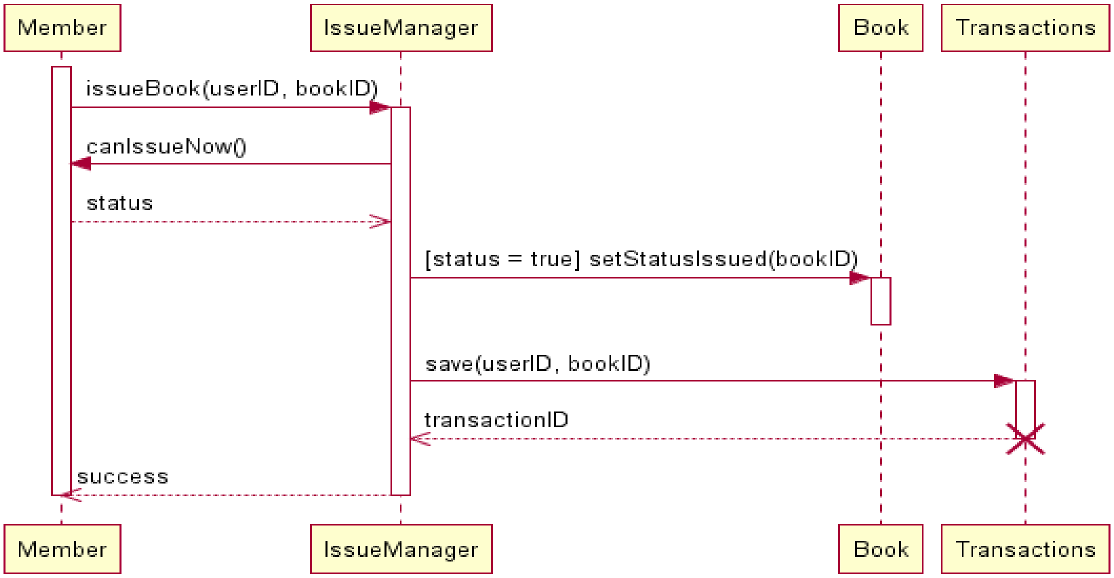


Figure 1: Sequence diagram for "Issue Book"

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. UML 2.0 provide components to specify the alternate scenarios within a sequence diagram (not discussed here). One can definitely make use of these components. 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. One creates an object from "Transactions" class, fills it up with all necessary information, and then saves the transaction. 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.

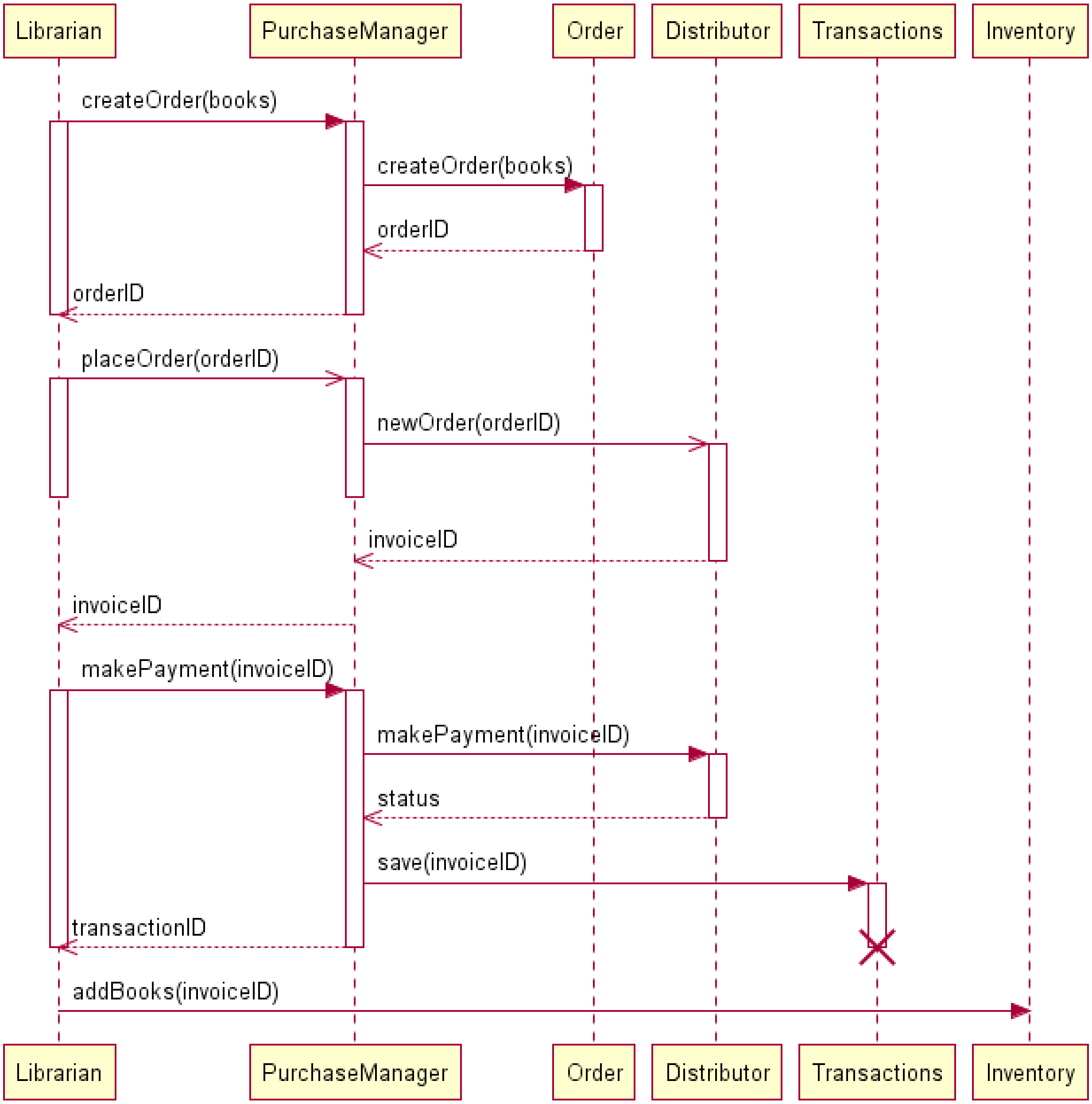


Figure 2: Sequence diagram for "Purchase Books"

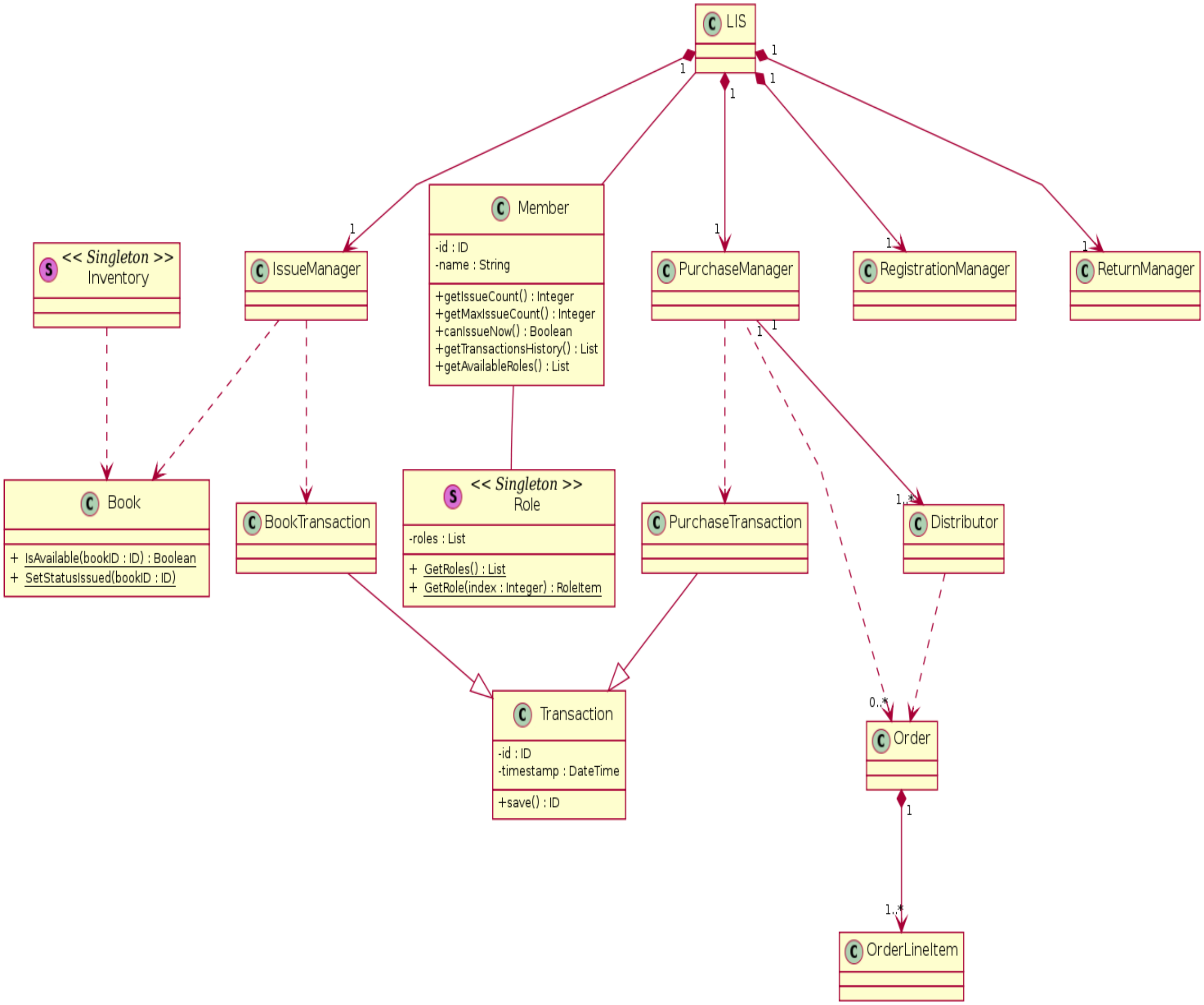
Classes are the fundamental components of any object oriented design and development. Unless individual class, it's attributes and associated operations have been modeled well, a lot of suffereing could await during the development phase. However, unlike waterfall model, the life cycle in object oriented development is iterative. 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
- Employee
- Book Inventory
- Distributor
- 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. For example, if the librarian goes on a leave, another member could be assigned the librarian role temporarily. Therefore, we decide to have a single "Member" class, whose instances could have one or more roles. This is shown in figure 3 with the "association" relationship between "Member" and "Role" classes. The "Role" class could consist of a list of available roles. A list could be maintained in the "Member" class to indicate which roles are associated with a particular instance of it.



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. No instance of "Book" needs to be created. The arrow from "IssueManager" to "Book" indicates that only the former knows about the "Book" class. 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.

Practical/ Viva Questions

1. A class is a

○ Blueprint

○ Specific instance of an object

○ Category of user requirement

○ None of the above

2. In class diagrams, a class is represented with a

○ Rectangle

○ Human stick figure

○ Ellipse

○ Diamond

3. From a class diagram it is evident that

○ All classes work in isolation

○ Each class is related with every other class

○ Most of the classes are related

○ Class diagram show object interactions

4. Inheritance among classes are represented by a

○ Solid line from the extending to the extended class

○ Line with an unfilled arrow head from the extending to the extended class

○ Line with a filled diamond from the extending to the extended class

○ Dotted line with extend stereotype from the extending to the extended class

5. Private members (or methods) in a class are indicated with a

○ Hash (#) sign

○ Minus (-) sign

○ Plus (+) sign

○ Tilde (~) sign

6. What does a sequence diagram represent?

○ Workflow in the system

○ How classes are related to each other

○ Sequence of events flow to achieve a target

○ Sequence of activities to be performed before moving to next state

7. In UML 2.0 a synchronous message is represented with a

○ Solid arrow with filled arrowhead

○ Solid arrow with empty arrowhead

○ Dotted arrow with filled arrowhead

○ Dotted arrow with empty arrowhead

8. An object can send a synchronous message and multiple asynchronous message in parallel

○ True

○ False

Following books and websites have been consulted for this experiment. You are suggested to go through them for further details.

Bibliography

- Fundamentals of Software Engineering, Rajib Mall, Prentice-Hall of India, 3rd Edition, 2009.
- THE UNIFIED MODELING LANGUAGE USER GUIDE, GRADY BOOCH, JAMES RUMBAUGH, IVAR JACOBSON, ADDISON-WESLEY, Low Priced Edition, 2000.

Webliography

- Class diagram
- UML Tutorial: Part 1 -- Class Diagrams.
- Synchronous messages The UML.
- Asynchronous message, UML sequence diagram.

Experiment 7

Modeling Data Flow Diagrams: Data Flow Diagram, Graphical notations for Data Flow Diagram, Symbols used in DFD, Context diagram and leveling DFD.

Introduction

Information Systems (IS) help in managing and updating the vast business-related information. Before designing such an IS, it is helpful to identify the various stakeholders, and the information that they would be exchanging with the system. An IS, however, is a large software comprised of several modules, which, in turn, share the process the available data. These data are often stored in databases for further references. A Data Flow Diagram (DFD) is used to pictorially represent the functionalities of the ISs by focusing on the sources and destinations of the data flowing in the system.

Objectives

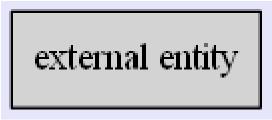
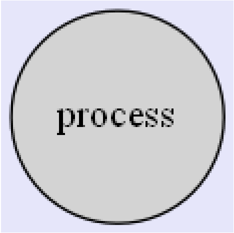
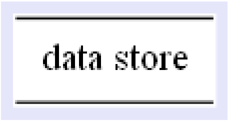
After completing this experiment you will be able to:

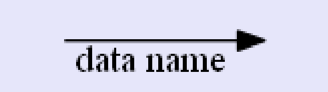
- Identify external entities and functionalities of any system.
- Identify the flow of data across the system.
- Represent the flow with Data Flow Diagrams.
- Time Required: Around 3.00 hours.

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		Name of the external entity is written inside the rectangle
Process		Name of the process is written inside the circle
Data store		A left-right open rectangle is denoted as data store; name of the data store is written inside the shape

Term	Notation	Remarks
Data flow		Data flow is represented by a directed arc with its data name

Explanation of Symbols used in DFD

1. **Process**: Processes are represented by circle. The name of the process is written into the circle. The name of the process is usually given in such a way that represents the functionality of the process. More detailed functionalities can be shown in the next Level if it is required. Usually it is better to keep the number of processes less than 7. If we see that the number of processes becomes more than 7 then we should combine some the processes to a single one to reduce the number of processes and further decompose it to the next level.
2. **External entity**: External entities are only appearing 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.
3. **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.
4. **Data flow**: Data flows are shown as a directed edge between two components of a Data Flow Diagram. Data can flow from external entity to process, data store to process, in between two processes and vice-versa.

Context diagram and leveling DFD

We start with a broad overview of a system represented in level 0 diagram. It is known as context diagram of the system. The entire system is shown as single process and also the interactions of external entities with the system are represented in context diagram. Further we split the process in next levels into several numbers of processes to represent the detailed functionalities performed by the system. Data stores may appear in higher level DFDs.

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 :

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

A DFD provides an easy mechanism to identify the flow of data in any information system. The adjoining simulation illustrates the generic steps to draw a DFD, and some thumb rules to keep in mind while drawing it.

Case Study
A Library Information System for Institute

The 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, 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). The external entities interacting with this system are members of LIS, library staff, librarian, and non-members of LIS. Two databases are used to keep track of member information and details of books in the library.

Let us focus on the external entity, Member. In order to issue or return books a member has to login to the system. The data flow labeled with “Login credentials” indicates the step when a member authenticates himself by providing required information (user ID, password). The system in turn verifies the user credentials using information stored in the member’s database. If all information are not provided correctly, the user is shown a login failure message. 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” identifies 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 are provided to the member.

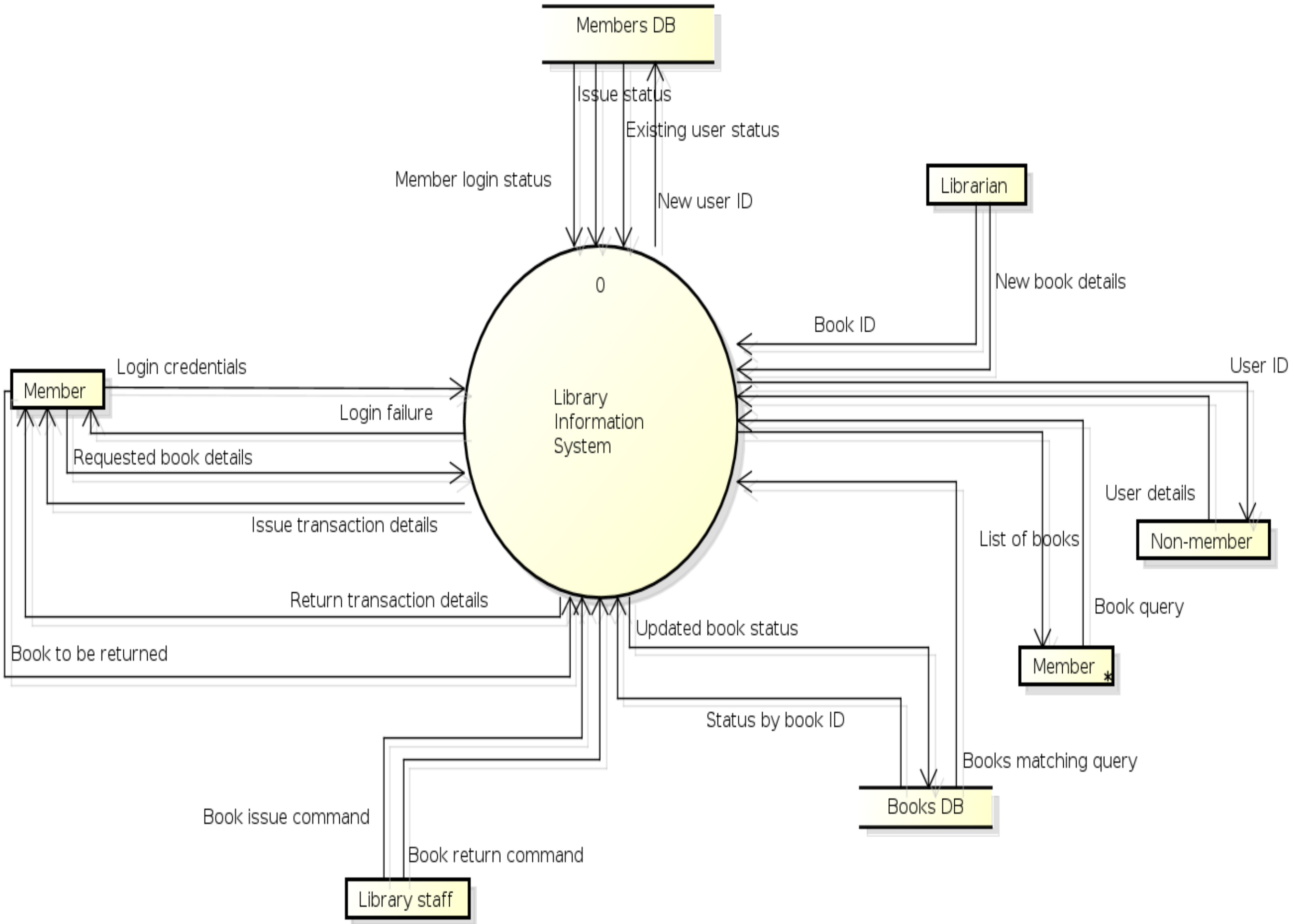


Figure 1: Context-level DFD for Library Information System

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. For example, the process "Issue book" has the number 1.2, which indicates that in the level 1 DFD the concerned process is numbered 2. Other processes are numbered in a similar way.

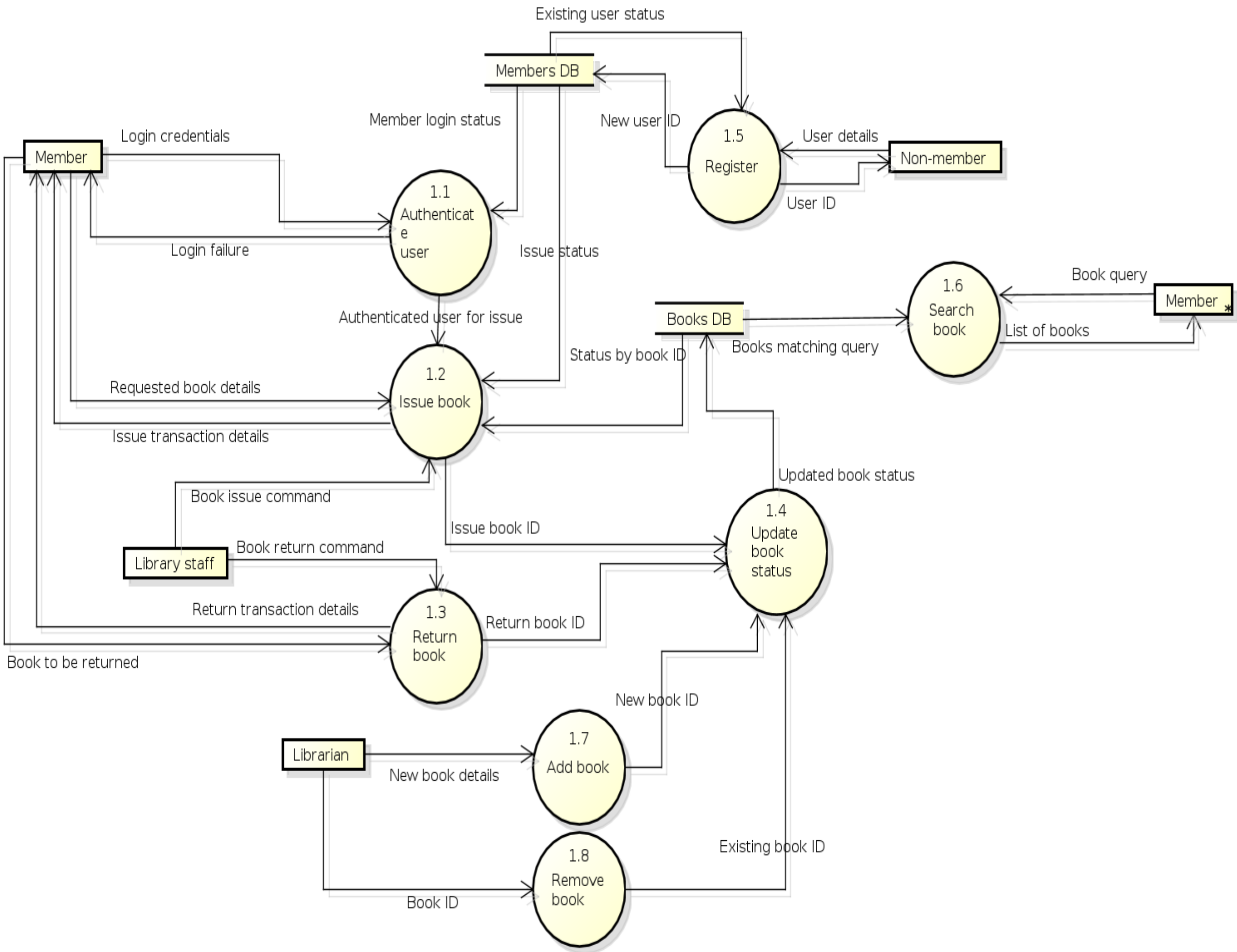


Figure 2: Level 1 DFD for Library Information System

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.

Finally, in order to eliminate intersecting lines and make the DFD complex, the Member external entity has been duplicated in figure 2. This is indicated by a * mark near the right-bottom corner of the entity box.

Practical / Viva Questions

1. A DFD represents

○ Flow of control

○ Flow of data

○ Both the above

○ Neither one

2. Which is not a component of a DFD?

○ Data flow

○ Decision

○ Process

○ Data store

3. How many processes are present in level-0 or context diagram?

○ 0

○ 1

○ 2

○ 3

4. External entities can appear in a DFD

○ At any level

○ Only at level-0

○ Only at level-1

○ Either at level-0 or at level-1

5. Data flow in a DFD is not possible in between

○ Two processes

○ Data store and process

○ External entity and data store

○ Process and external entity

Exercise Problem

Draw a context-level DFD to depict the typical user authentication process used by any system. An user gives two inputs -- user name and password.

Following books and websites have been consulted for this experiment.

You are suggested to go through them for further details.

Bibliography

- Software Engineering, Ian Sommerville, Addison Wesley Longman, 9th Edition, March 2010.
- Fundamentals of Software Engineering, Rajib Mall, Prentice-Hall of India, 3rd Edition, 2009.

Webliography

- NPTEL course material - System Analysis and Design.
- Software Engineering Virtual Lab - Simulation.
- Software Engineering Virtual Lab - Case Study.

Experiment 8

Estimation of Test Coverage Metrics and Structural Complexity: Control Flow Graph, Terminologies, McCabe's Cyclomatic Complexity, Computing Cyclomatic Complexity, Optimum Value of Cyclomatic Complexity, Merits, Demerits.

Introduction

A visual representation of flow of control within a program may help the developer to perform static analysis of his code. One could break down his program into multiple basic blocks, and connect them with directed edges to draw a Control Flow Graph (CFG). A CFG of a program helps in identifying how complex a program is. It also helps to estimate the maximum number of test cases one might require to test the code.

In this experiment, we will learn about basic blocks and how to draw a CFG using them. We would look into paths and linearly independent paths in context of a CFG. Finally, we would learn about McCabe's cyclomatic complexity, and classify a given program based on that.

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.
- Time Required: Around **3.00** hours.

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. For simplicity, each node in the CFG has been labeled with the line numbers of the program containing the instructions. 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;
```

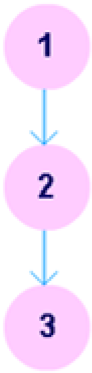


Figure 1: A simple program and it's CFG

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.

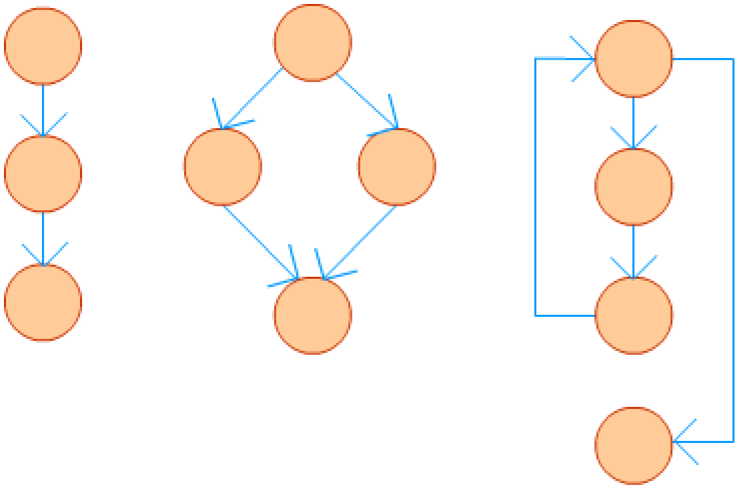


Figure 2: CFG for different types of statements

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 [ii, iii] 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.

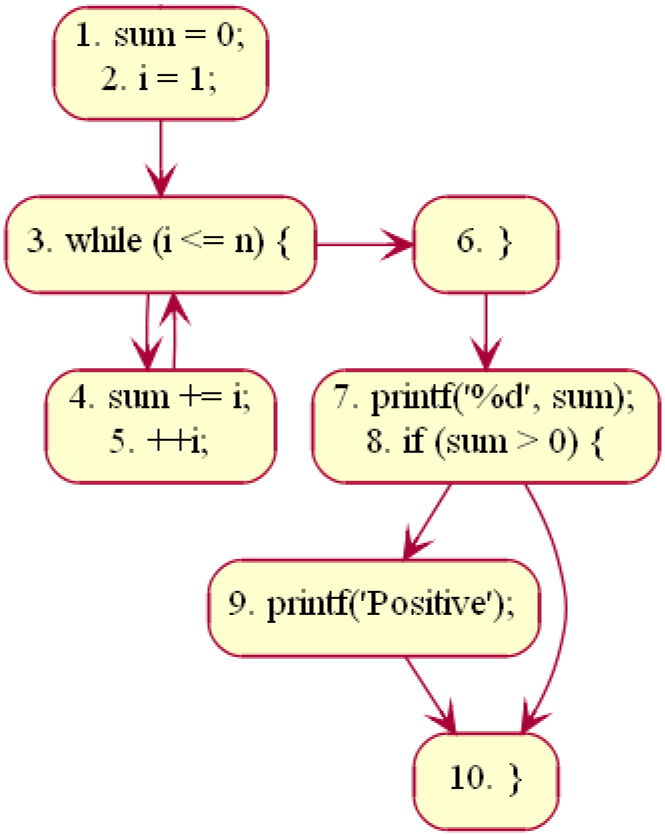


Figure 3: Basic blocks in a CFG.

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. The block containing line #s 1, 2 is said to be the entry block; the block containing line # 10 is said to be the exit block.

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 shows the corresponding CFG. The sub-graph containing line #s 8, 9, 10 is disconnected from the graph containing the entry block. The code in the disconnected sub-graph would never get executed, and, therefore, could be discarded.

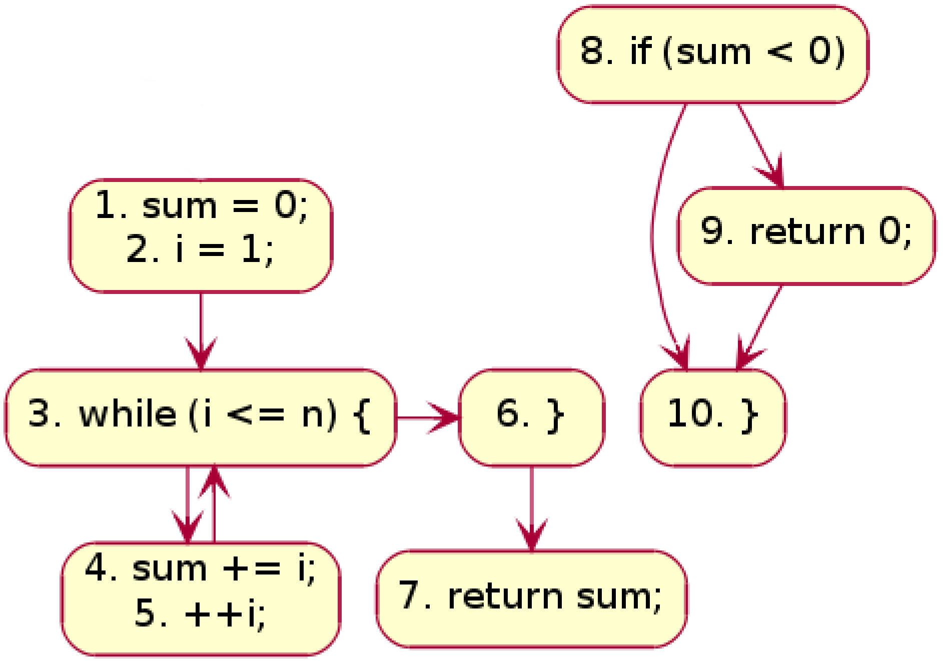


Figure 4: CFG with unreachable blocks

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 node.

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

Note that 1 - 3 - (4, 5) - 3 - (4, 5) - 6 - (7, 8) - 10, for instance, won't qualify as a linearly independent path because there is no new edge not already present in any of the above four linearly independent paths.

McCabe's Cyclomatic Complexity

McCabe had applied graph-theoretic analysis to determine the complexity of a program module [vi]. 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. Complexity of a module increases as the number of such paths in the module increase. 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 + 1It 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, thenV(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. A quick summary of how different types of statements affect V(G). 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)) + 1where ***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. If any module is found to have Cyclomatic complexity greater than 10, the module should be considered for redesign. 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.

Example: Code for GCD computation by Euclid's method

```
while (x != y)
{
    if (x > y)
        x = x - y;
    else
        y = y - x;
}
return x;
```

Determining McCabe's Cyclomatic Complexity

Method #1

N = No. of nodes = 7

E = No. of edges = 8

V(G) = E - N + 2 = 8 - 7 + 2 = 3.

Method #2

V(G) = Total no. of non overlapping areas + 1 = 2 + 1 = 3.

Method #3

V(G) = Total no. of decision statements and loops + 1 = 1 + 1 + 1 = 3.

Exercise Problem:

Identification of basic blocks from a program and determining it's cyclomatic complexity

```
#include <stdio.h>
Int main(int argc, char **argv)
{
        int i;
        int sum;
        int n=10;
        sum = 0;
```

```
          for (i = 1; i <= n; i++)
           sum += i;
           printf("Sum of first %d natural numbers is: %d\n", n,
           sum);
      return 0;
  }
```

Any sequence of instructions in a program could be represented in terms of basic blocks, and a CFG could be drawn using those basic blocks. For the given C program:

- Identify the basic blocks and verify whether your representation matches with the output produced after compiling your program.
- Draw a Control Flow Graph (CFG) using these basic blocks. Again, verify how the CFG generated after compilation relates to the basic blocks identified by the compiler.
- Calculate McCabe's complexity from the CFG so obtained.
- Note that gcc translates the high-level program into an intermediate representation using GIMPLE. So, the CFG generated from your code would not show the actual instructions.

Learning Objectives:

- Identify the basic basic blocks for a given program.
- Draw a CFG using the basic blocks.
- Determination of McCabe's complexity from a CFG.

Following books and websites have been consulted for this experiment. **You are suggested to go through them for further details.**

Bibliography

1. Software Engineering: A Practioner's Approach, Roger S. Pressman, McGraw Hills, 7th Edition, 2009.
2. Software Engineering, Ian Sommerville, Addison Wesley Longman, 9th Edition, March 2010.
3. Fundamentals of Software Engineering, Rajib Mall, Prentice-Hall of India, 3rd Edition, 2009.
4. Just Enough Software Test Automation, Daniel J. Mosley, Bruce A. Posey, Prentice Hall, July 2002.

Webliography

- Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric.
- Control flow graphs and loop optimizations - Purdue Engineering.
- Control Flow Analysis.
- Basic Blocks and Flow Graphs.
- Measuring Structures (Complexity).

Experiment 9

Designing Test Suites: 1 week Software Testing, Standards for Software Test Documentation, Testing Frameworks, Need for Software Testing, Test Cases and Test Suite, Types of Software Testing, Unit Testing, Integration Testing, System Testing, Example, Some Remarks.

Introduction

Development of a new software, like any other product, remains incomplete until it subjected to exhaustive tests. The primary objective of testing is not to verify that all desired features have been implemented correctly. However, it also includes verification of the software behavior in case of "bad inputs".

In this experiment we discuss in brief about different types of testing, and provide mechanisms to have hands-on experience on unit testing.

Objectives

After completing this experiment you will be able to:

1. Learn about different techniques of testing a software
2. Design unit test cases to verify the functionality and locate bugs, if any
3. Time Required: Around 3.00 hours

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 software and to find the defects present in 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]. 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 1044, a standard for the classification of software anomalies

- IEEE 1044-1, a guide to the classification of software anomalies
- 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

1. **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 condition that the last few years of the 20th century systems had to be shown to be free from the ‘millennium bug’.
- 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
- **Unit Testing**
- **Integration Testing**
- **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. e.g. Consider a software program that computes whether an integer number is even or not that is in the range of 0 to 10. Determine the equivalence class test suite. There are three equivalence classes for this program. - The set of negative integer - The integers in the range 0 to 10 - The integer larger than 10.

- **Boundary value analysis:** In this approach, while designing the test cases, the values at boundaries of different equivalence classes are taken into consideration. e.g. In the above given example as in equivalence class partitioning, a boundary values based test suite is { 0, -1, 10, 11 }.

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. White box testing can be performed to validate whether code implementation follows intended design, to validate implemented security functionality, and to uncover exploitable vulnerabilities.This testing requires access to the source code. 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. But the top-down integration testing starts with the main routine and one or two subordinate routines in the system.

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. Therefore, after every round of error-fixing, another testing is carried out, i.e. called regression testing. 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 [2]. Regression testing is an important activity at both testing and maintenance phases. When a piece of software is modified, it is necessary to ensure that the quality of the software is preserved. 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 efforts for testing should begin in the form of preparation of test cases after the requirements have been finalized. 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. Test cases should be designed in such a way that all target features could be verified. 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 Institute

The 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.

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. Note that it is possible that a particular requirement might need multiple test cases to verify whether it has been implemented correctly.

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
- 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.

Table 2: A test suite to verify the "User Login" feature

#	**TS1**
Title	**Verify "User Login" functionality**
Description	**To test the different scenarios that might arise while an user is trying to login**

#	Summary	Dependency	Pre-condition	Post-condition	Execution Steps	Expected Output
TC1	Verify that user already registered with the LIS is able to login with correct user ID and password		Employee ID *149405* is a registered user of LIS; user's password is *this_is_password*	User is logged in	1. Type in employee ID as *149405* 2. Type in password *this_is_password* 3. Click on the 'Login' button	"Home" page for the user is displayed
TC2	Verify that an unregistered user of LIS is unable to login		Employee ID *149405xx* is not a registered user of LIS	User is not logged in	1. Type in employee ID as *149405xx* 2. Type in password *whatever* 3. Click on the 'Login' button	The "Login" dialog is shown with a *"Login failed! Check your user ID and password"* message
TC3	Verify that user already registered with the LIS		Employee ID *149405* is a registered user of LIS; user's	User is not logged in	1. Type in employee ID as *149405* 2. Type in password *whatever*	The "Login" dialog is shown with a *"Login failed! Check your user ID and*

Table 2: A test suite to verify the "User Login" feature

#	**TS1**
Title	**Verify "User Login" functionality**
Description	**To test the different scenarios that might arise while an user is trying to login**

#	Summary	Dependency	Pre-condition	Post-condition	Execution Steps	Expected Output
	is unable to login with incorrect password		password is *this_is_password*		3. Click on the 'Login' button	*password"* message
TC4	Verify that user already registered with the LIS is unable to login with incorrect password given twice consecutively	TC3	This test case is executed after execution of TC3 before executing any other test case	User is not logged in	1. Type in employee ID as *149405* 2. Type in password *whatever2* 3. Click on the 'Login' button	The "Login" dialog is shown with a *"Login failed! Check your user ID and password"* message
TC5	Verify that user already registered with the LIS is unable to login with incorrect password given thrice consecutively	TC4	This test case is executed after execution of TC4 before executing any other test case	User is not logged in	1. Type in employee ID as *149405* 2. Type in password *whatever3* 3. Click on the 'Login' button	The "Login" dialog is shown with a *"Login failed! Check your user ID and password"* message; the security question and input box for the answer are displayed
TC6	Verify that a registered user can login after three consecutive failures by correctly answering the security question	TC5	This test case is executed after execution of TC6 before executing any other test case. Answer to the security question is *my_answer*.	Email sent containing new password. The email is expected to be received within 2 minute.	1. Type in the answer as *my_answer* 2. Click on the 'Email Password' button	Login dialog is displayed; an email containing the new password is received
TC7	Verify that a		Execute the test	User	1. Type in the answer	The message *"Your*

Table 2: A test suite to verify the "User Login" feature

#	TS1
Title	**Verify "User Login" functionality**
Description	**To test the different scenarios that might arise while an user is trying to login**

#	Summary	Dependency	Pre-condition	Post-condition	Execution Steps	Expected Output
	registered user's account is blocked after three consecutive failures and answering the security question incorrectly		cases TC3, TC4, and TC5 once again (in order) before executing this test case	account has been blocked	as *not_my_answer* 2. Click on the 'Email Password' button	*account has been blocked! Please contact the administrator."* appears

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.

Practical / Viva Questions

1. Software testing is the process of

○ Demonstrating that errors are not present

○ Establishing confidence that a program does what it is supposed to do

○ Executing a program to show that it is working as per specifications

○ Executing a program with the intent of finding errors

2. Alpha testing is done by

○ Customer

○ Tester

○ Developer

○ All of the above

3. Test suite is

○ Set of test cases

○ Set of inputs

○ Set of outputs

○ None of the above

4. Regression testing is primarily related to

○ Functional testing

○ Data flow testing

○ Development testing

○ Maintenance testing

5. Acceptance testing is done by

○ Developers

○ Customers

○ Testers

○ All of the above

6. Testing the software is basically

○ Verification

○ Validation

○ Verification and Validation

○ None of the above

7. Functionality of a software is tested by

○ White box testing

○ Black box testing

○ Regression testing

○ None of the above

8. Top down approach is used for

○ Development

○ Identification of faults

○ Validation

○ Functional testing

9. Testing of software with actual data and in the actual environment is called

○ Alpha testing

○ Beta testing

○ Regression testing

○ None of the above

10. During the development phase, which of the following testing approach is not adopted

○ Unit testing

○ Bottom up testing

○ Integration testing

○ Acceptance testing

11. Beta testing is carried out by

○ Users

○ Developers

○ Testers

○ All of the above

12. Equivalence class partitioning is related to

○ Structural testing

○ Black box testing

○ Mutation testing

○ All of the above

13. During the software validation:

○ Process is checked

○ Product is checked

○ Developers' performance is evaluated

○ The customer checks the product

14. Software mistakes during coding are known as:

○ Failures

○ Ddefects

○ Bbugs

○ Errors

15. Which is not a functional testing technique ?

○ Boundary value analysis

○ Decission table

○ Regression testing

○ None of the above

Steps for conducting the experiment

General Instructions

Follow are the steps to be followed in general to perform the experiments in Software Engineering Lab.

- Read the theory about the experiment
- View the simulation provided for a chosen, related problem
- Take the self evaluation to judge your understanding (optional, but recommended)

- Solve the given list of exercises

Experiment Specific Instructions

Following are the instructions specifically for this experiment:

1. Type in the code (in JavaScript) to be tested in the text area below the header **"Code"**
2. Once the code is ready, click on the **"Create test suite"** just below the code area. A small dialog box will appear just below the button.
3. Add a title and summary for the test suite to be created. (Both are optional.) When done, click on the **"Add**" button. If this test suite is not supposed to be added, click on the **"Cancel**" link.
4. After clicking on the **"Add**" button from the previous step, a dialog will display the new test suite. Every test suite is identified with a unique ID: an (auto-incrementing) integer prefixed with "TS". The first test suite will have the ID **"TS0** ", and so on.
5. An already added test suite could be removed by clicking on the **"Remove**" link
6. Once a test suite has been created, click on the **"Add test cases**" button to add the test cases individually
7. After clicking on the **"Add test cases**" button a spreadsheet-type dialog will appear just below the button. The spreadsheet has six columns:
   - **Summary:** A brief description of the test case (mandatory)
   - **Script:** A (JavaScript) function to be called for execution (mandatory)
   - **Expected Output:** The value the above function call is expected to return (mandatory)
   - **Actual Output:** The value actually obtained after making the function call. This column would be populated automatically after the test suite is executed.
   - **Manual Testing:** Certain test cases could not be checked automatically. For example, a testing framework may not verify that an HTML element X is not overlapping with another HTML element Y. In such cases, manual intervention is required. To specify that a test case would be executed manually, select the **"Yes"** check box under this column.
   - **Status:** Indicates the status of a test case after it is executed. Possible values are:
     - **No Run:** The test suite has not been executed yet or the test case has been set for **Manual Testing**)
     - **Pass:** The concerned test case's expected and actual values are same
     - **Fail:** The concerned test case's expected and actual values are NOT same
   - The number of columns in the spreadsheet is fixed (six). The number of initial rows is 10. New rows could, however, be added by pressing **Enter** while the last column of the last row is being selected.
8. **NOTE:** Summary of a test case IS mandatory for every test case in the test suite. If a particular test case has no summary, the concerned test case, and all other subsequent test cases, would be ignored!
9. Once test cases have been written, click on the **"Execute test suite"** button to execute the test cases for the concerned test suite. Please note that this may not execute all the test cases in the test suite if the constraints as mentioned in the previous step are not met. Also, any test case set for **Manual testing** would be skipped.
10. Results of execution of the test suite would appear just below the **"Execute test suite"** button. If all the test cases of the test suite have passed (i.e. values

under **Expected Output** column is same as under **Actual Output** column for each row), the background colour of the results dialog would be green. Otherwise, it would be red.

Design a test suite for the following problem

The Absolute Beginners Inc. seems to have been fascinated by your work. Recently they have entrusted you with a task of writing a web-based mathematical software (using JavaScript). As part of this software, your team mate has written a small module, which computes area of simple geometric shapes. A portion of the module is shown below.

```
function square(side)
{
        return side * side
}
function rectangle(side1, side2)
{
        return side1 * side1;
}
function circle(radius)
{
        return Math.PI * radius * radius;
}
function right_triangle(base, height)
{
        return 1 / 2 * base * height;
}
```

Prepare a test suite that will

- Verify each of the above mentioned individual function is working correctly

Your task essentially is to verify whether each of the above function is returning correct values for given inputs. For example, a rectangle with length 10 unit and breadth 5 unit will have an area of 50 sq. unit. This can be verified from the output of the function call

```
1  rectangle(10, 5);
```

However, testing also attempts to point out possible bugs in the software. How would the above code behave for a call

```
1  rectangle(10, -5);
```

Modify the code to address this defect.

- In each function, return -1 if any given dimension is negative.
- Modify the test suite such that it reflects desired performance for both correct and incorrect input(s).
- The code has another bug -- how would you identify it from testing results? Fix the bug and test it again.

Learning Objectives:

- Get familiarized with unit testing.
- Verify implementation of functional requirements by writing test cases.
- Analyze results of testing to ascertain the current state of a project.

Following books and websites have been consulted for this experiment. You are suggested to go through them for further details.

Bibliography

- Fundamentals of Software Engineering, Rajib Mall, Prentice-Hall of India, 3rd Edition, 2009.
- Software Engineering: A Practioner's Approach, Roger S. Pressman, McGraw Hills, 7th Edition, 2009.

Webliography

- Standards for software test documentation.
- IBM Rational Software information Center.
- Developing Test Plans.