Object Oriented Design

Object oriented design is the result of focusing attention not on the function performed by the program, but instead on the data that are to do manipulated by the program.

It is mainly the process of using an object methodology to design a computing system or application.

Object Oriented Design begins with an examination of the real world "things" that are part of the problem to be solved. These things (which we will call objects) are characterized individually in terms of their attributes and behavior.

GLA UNIVERSITY MATHURA
Recognised by UGC Under Section 2(f)
Accredited with A Grade by NAAC

Basic Concepts of OOD

Object Oriented Design is not dependent on any specific implementation language. Problems are modeled using objects.

Objects have

Behaviour (they do things)

State (which changes when they do things)

GLA UNIVERSITY MATHURA
Recognised by UGC Under Section 2(f)
Accredited with A Grade by NAAC

Terminologies related to OOD

Object

The word “Object” is used very frequently and conveys different meaning in different circumstances. Here, meaning is an entity able to save a state (information) and which offers a number of operations (behavior) to either examine or affect this state. An object is characterized by number of operations and a state which remembers the effect of these operations

.

Messages

Objects communicate by message passing. Messages consist of the identity of the target object, the name of the requested operation and any other operation needed to perform the function. Message are often implemented as procedure or function calls.

Abstraction

In object oriented design, complexity is managed using abstraction. Abstraction is the elimination of the irrelevant and the amplification of the essentials.

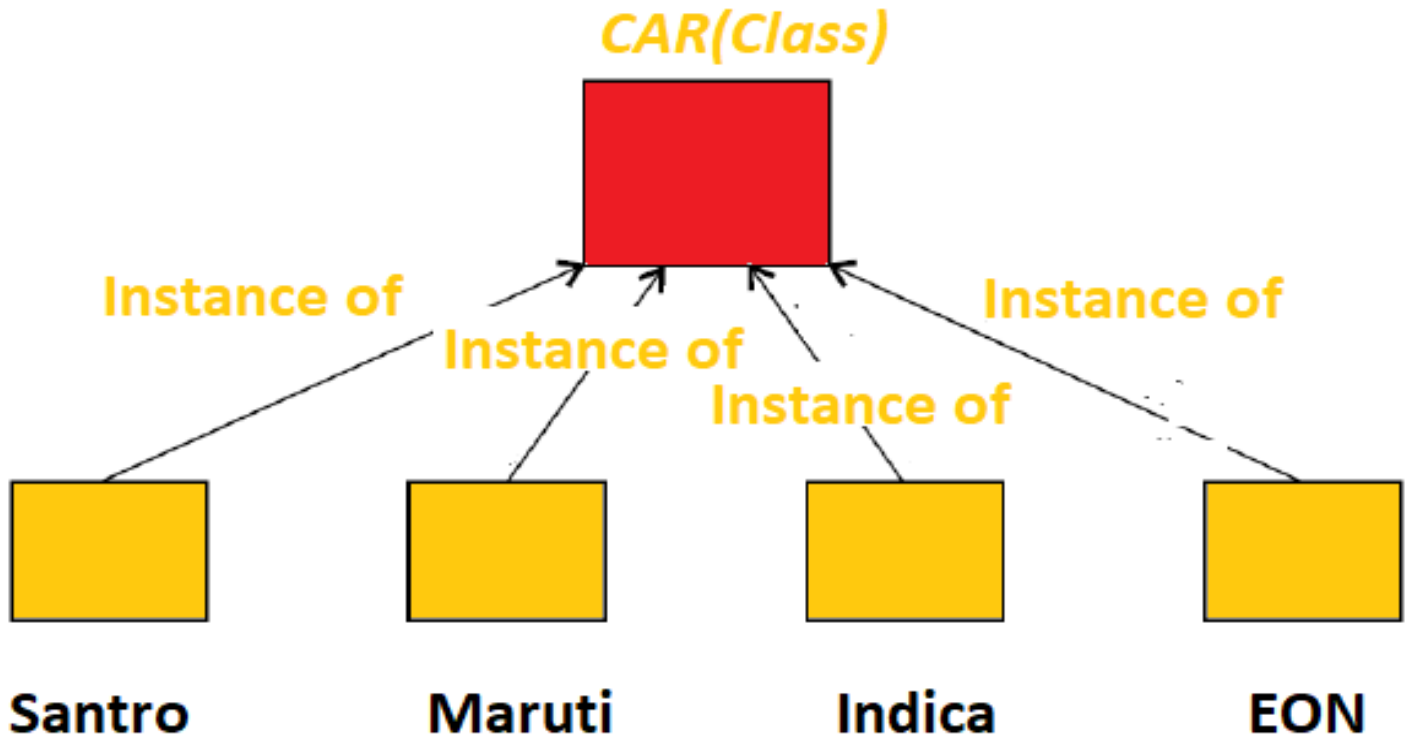
GLA UNIVERSITY MATHURA
Recognised by UGC Under Section 2(f)
Accredited with A Grade by NAAC

Class

In any system, there shall be number of objects. Some of the objects may have common characteristics and we can group the objects according to these characteristics. This type of grouping is known as a class. Hence, a class is a set of objects that share a common structure and a common behaviour.

GLA UNIVERSITY MATHURA
Recognised by UGC Under Section 2(f)
Accredited with A Grade by NAAC

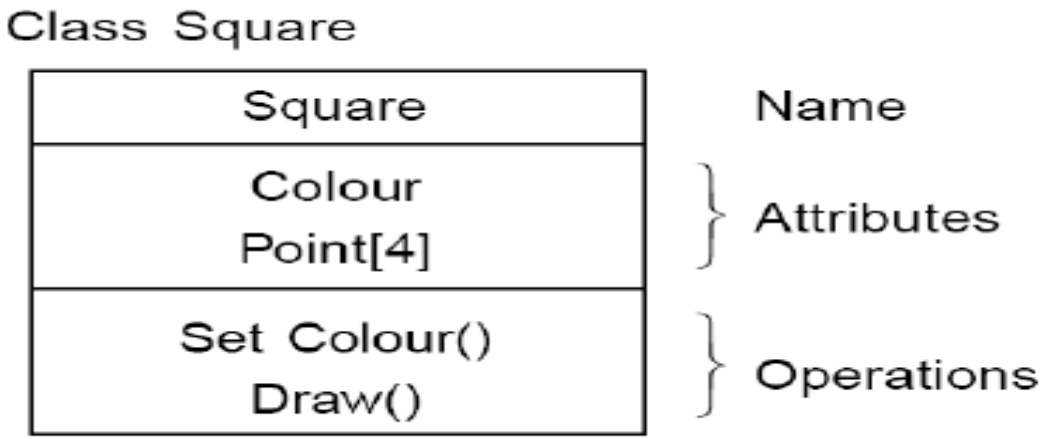
Ex. We may define a class “car” and each object that represent a car becomes an instance of this class. In this class “car”, Indica, Santro, Maruti, Indigo are instances of this class as shown in fig



GLA UNIVERSITY MATHURA
Recognised by UGC Under Section 2(f)
Accredited with A Grade by NAAC

Attributes

An attributes is a data value held by the objects in a class. The square class has two attributes: a colour and array of points. Each attributes has a value for each object instance



GLA UNIVERSITY MATHURA
Recognised by UGC Under Section 2(f)
Accredited with A Grade by NAAC

Operations

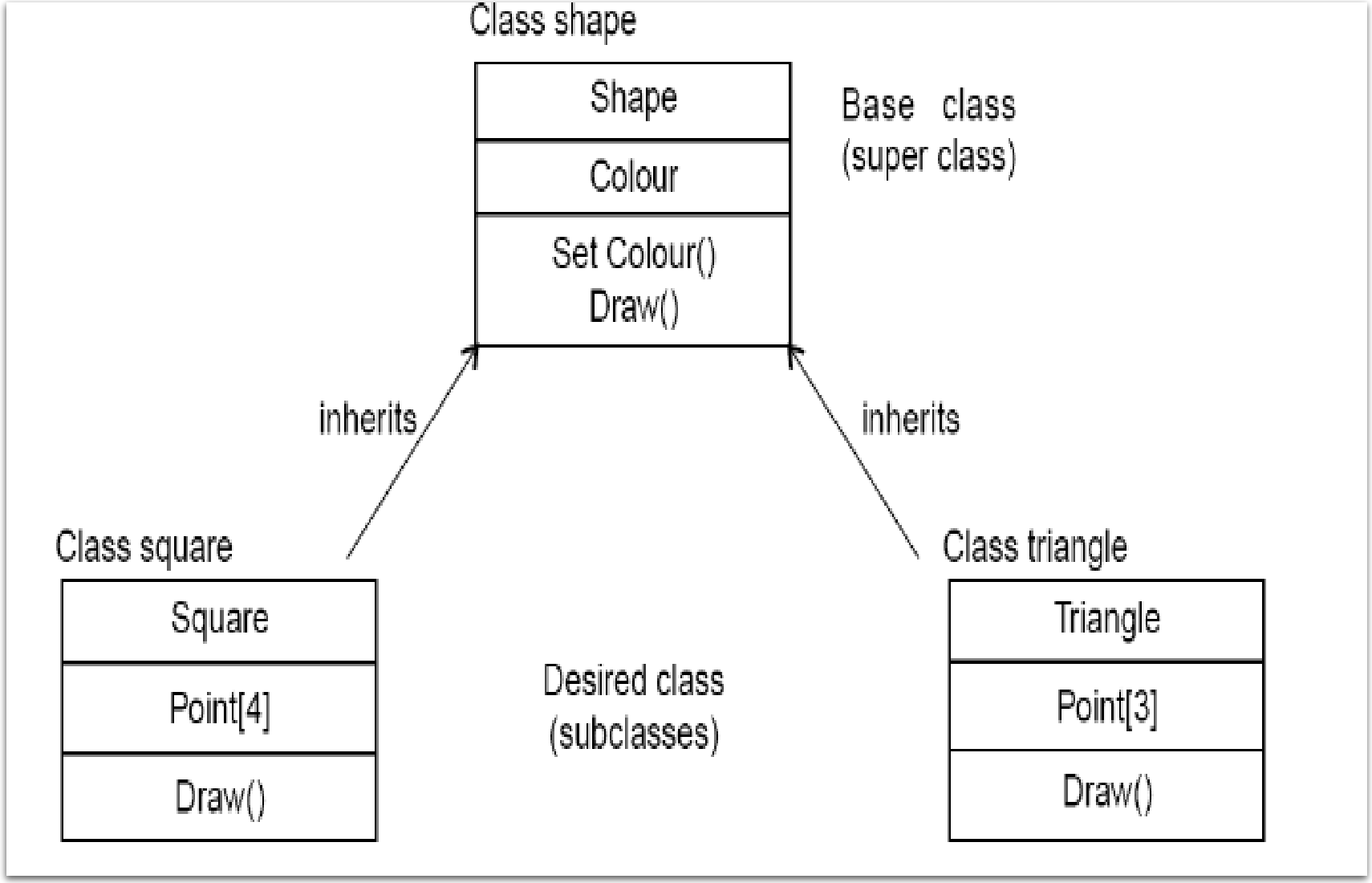
An operation is a function or transformation that may be applied to or by objects in a class.

In the square class, we have two operations: set colour() and draw(). All objects in a class share the same operations. An object “knows” its class, and hence the right implementation of the operation.

GLA UNIVERSITY MATHURA
Recognised by UGC Under Section 2(f)
Accredited with A Grade by NAAC

Inheritance

OOD allows similar classes to stack up in a hierarchical manner where the lower or sub-classes can import, implement, and re-use allowed variables and functions from their immediate super classes. This property of OOD is called an inheritance. This makes it easier to define a specific class and to create generalized classes from specific ones.



Source: Software Engineering by KK Agarwal &Yogesh Singh

GLA UNIVERSITY MATHURA
Recognised by UGC Under Section 2(f)
Accredited with A Grade by NAAC

Polymorphism

OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned the same name. This is known as polymorphism, which allows a single interface is performing functions for different types.

Encapsulation (Information Hiding)

Encapsulation is also commonly referred to as “Information Hiding”. It consists of the separation of the external aspects of an object from the internal implementation details of the object.

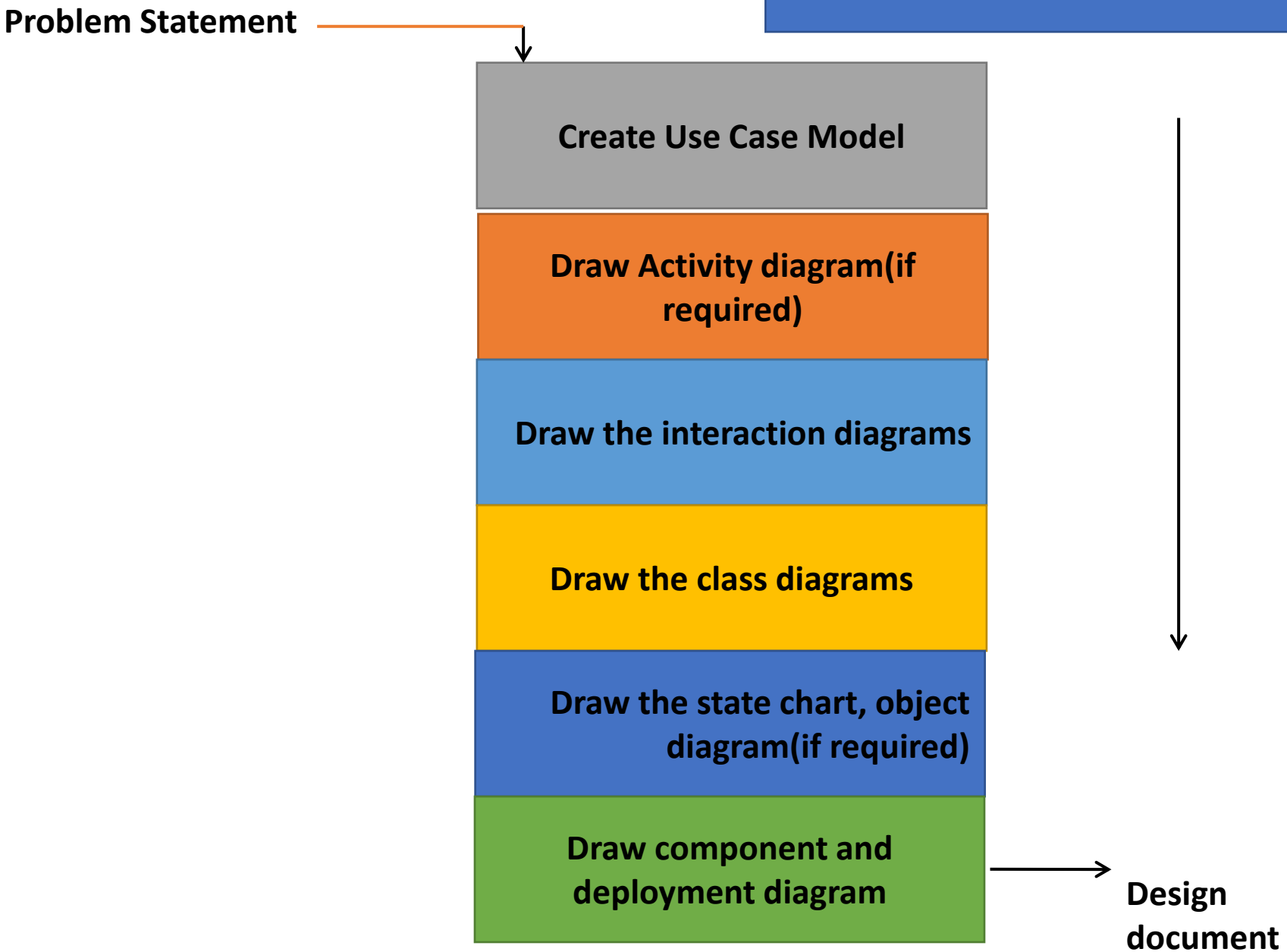
GLA UNIVERSITY MATHURA
Recognised by UGC Under Section 2(f)
Accredited with A Grade by NAAC

Hierarchy

Hierarchy involves organizing something according to some particular order or rank. It is another mechanism for reducing the complexity of software by being able to treat and express sub-types in a generic way.

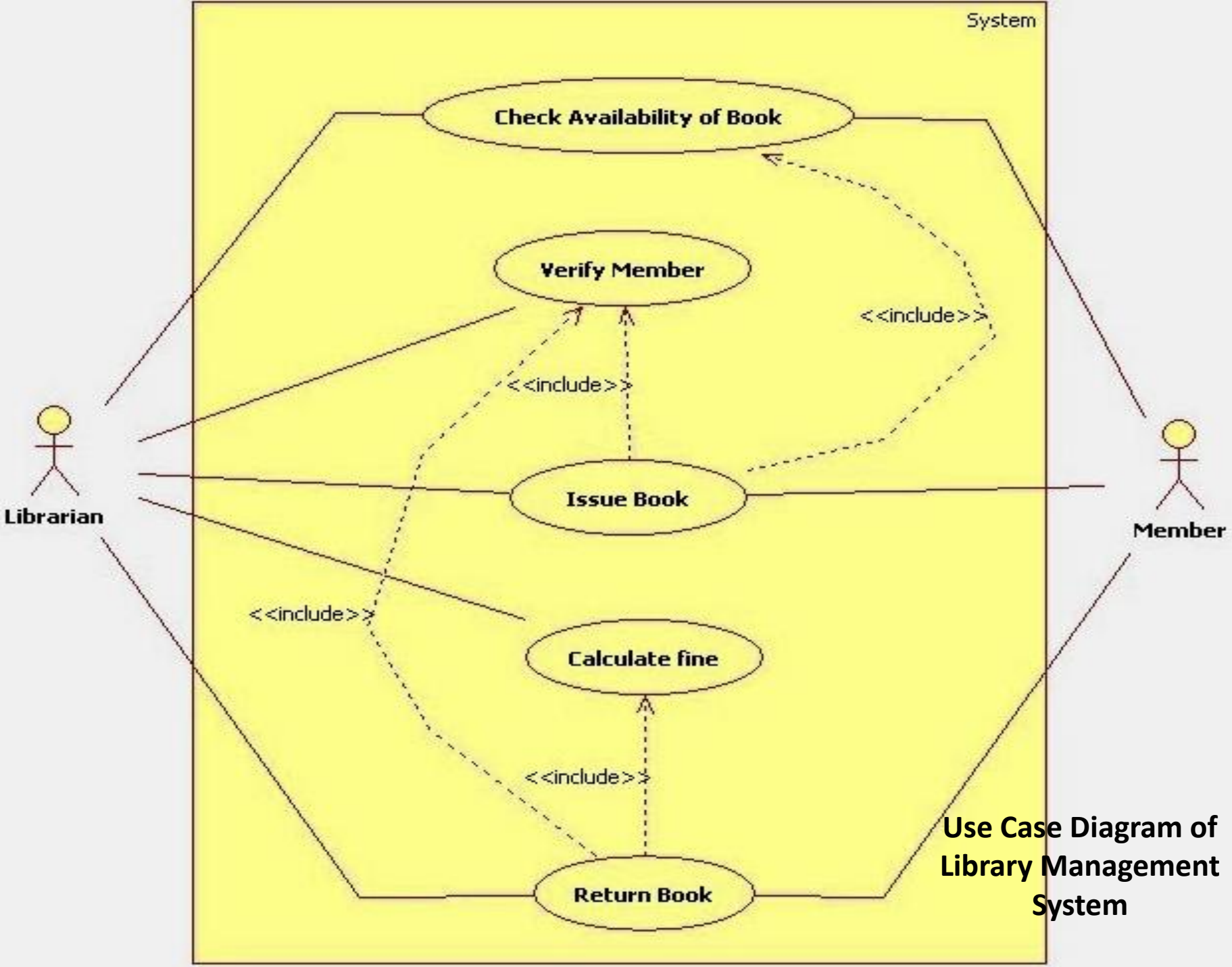
GLA UNIVERSITY MATHURA
Accredited with A Grade by NAAC

Steps for analysis & design of object oriented system



Create use case model

First step is to identify the actors interacting with the system. We should then write the use case and draw the use case diagram.



Use Case Diagram of Library Management System

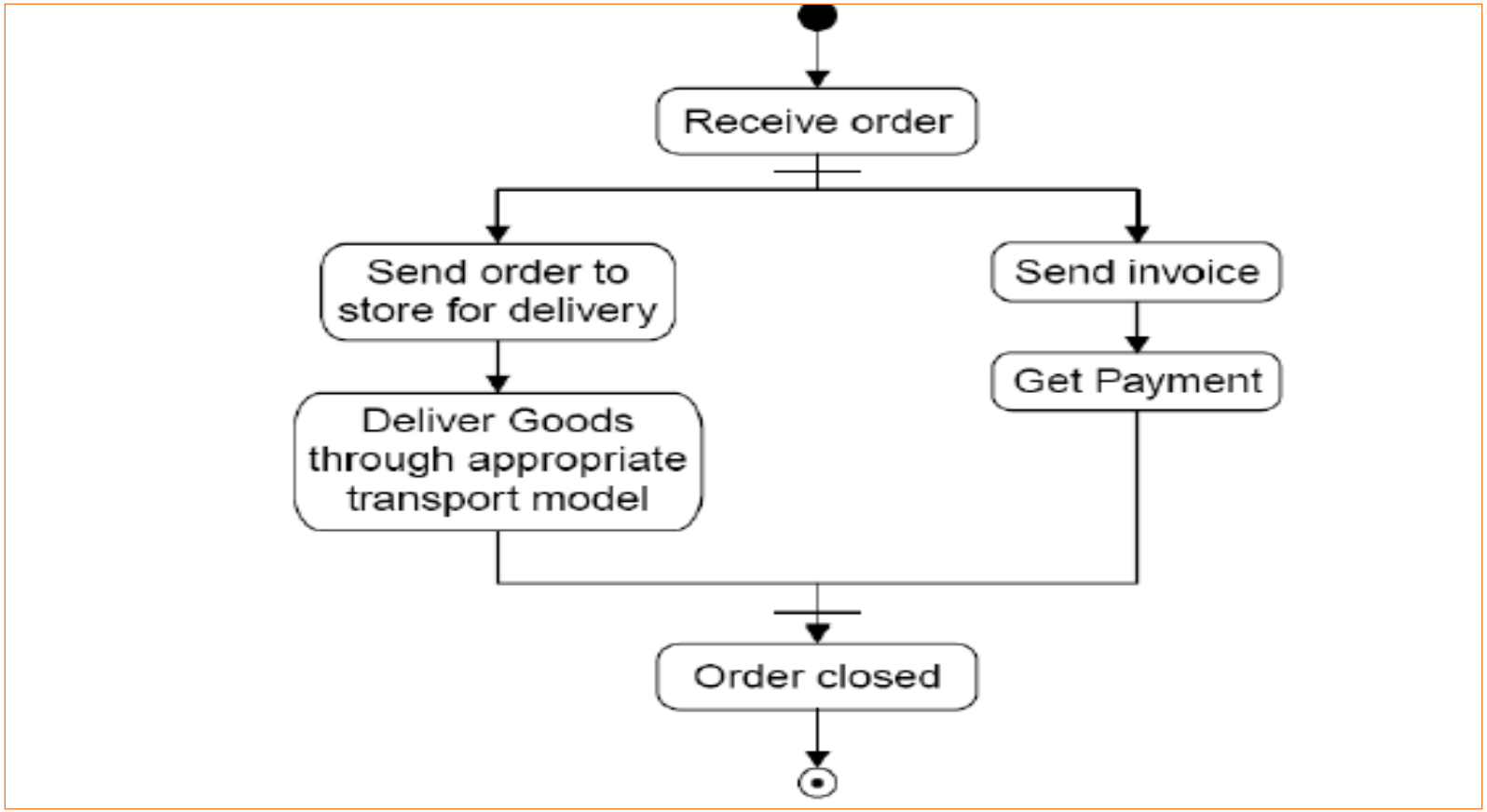
GLA UNIVERSITY
MATHURA
Recognised by UGC Under Section 2(f)
Accredited with A Grade by NAAC

Draw activity diagram (If required)

Activity Diagram illustrate the dynamic nature of a system by modeling the flow of control form activity to activity. An activity represents an operation on some class in the system that results in a change in the state of the system.

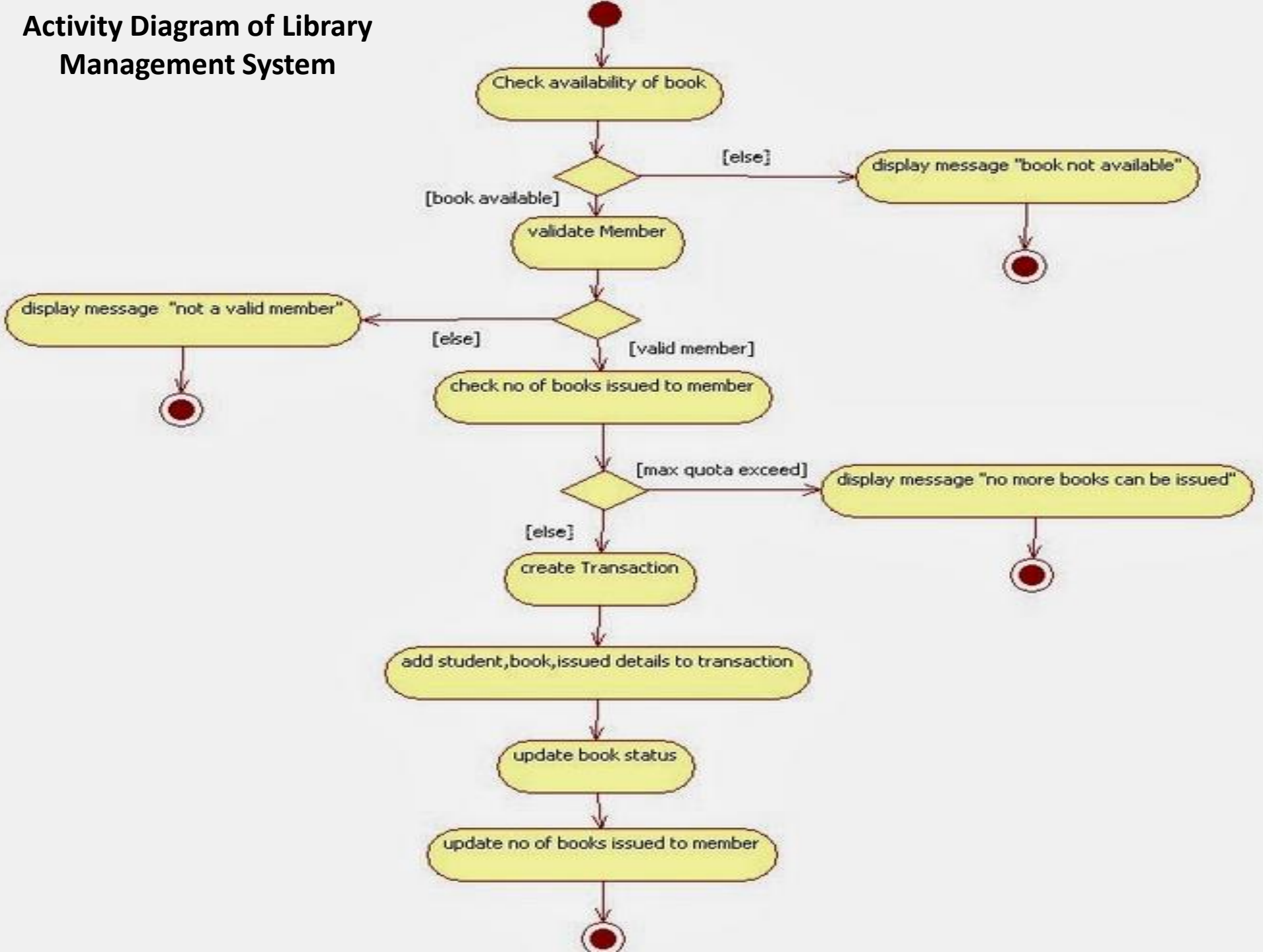
GLA UNIVERSITY MATHURA
Recognised by UGC Under Section 2(f)
Accredited with A Grade by NAAC

Activity diagram processing an order to deliver some goods.



Source: Software Engineering by KK Agarwal &Yogesh Singh

Activity Diagram of Library Management System



Draw the interaction diagram

An interaction diagram shows an interaction, consisting of a set of objects and their relationship, including the messages that may be dispatched among them. Interaction diagrams address the dynamic view of a system

Steps to draws interaction diagrams are as under

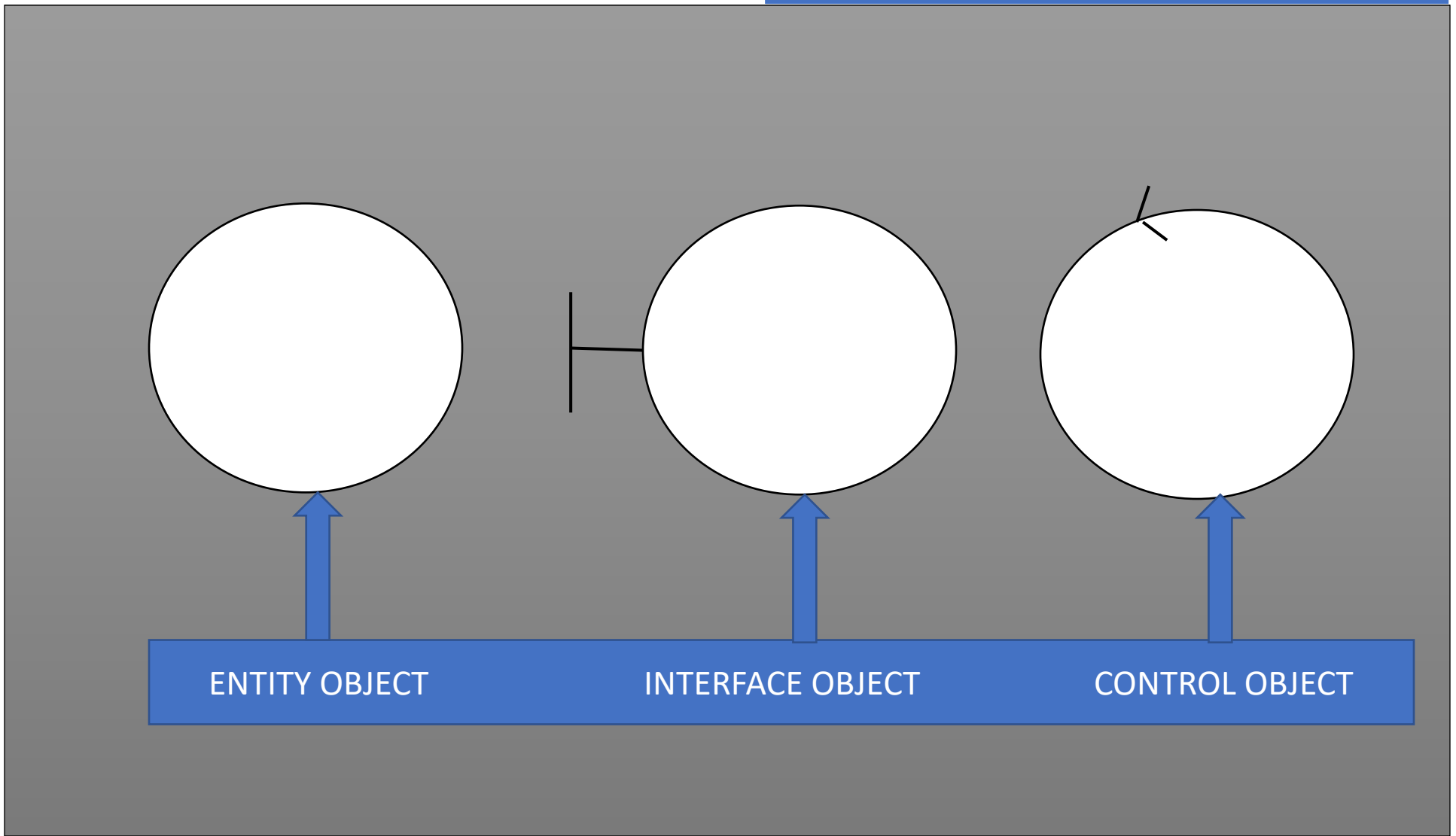
a)Firstly, we should identify that the objects with respects to every use case.

b) We draw the sequence diagrams for every use case.

d) We draw the collaboration diagrams for every use case.

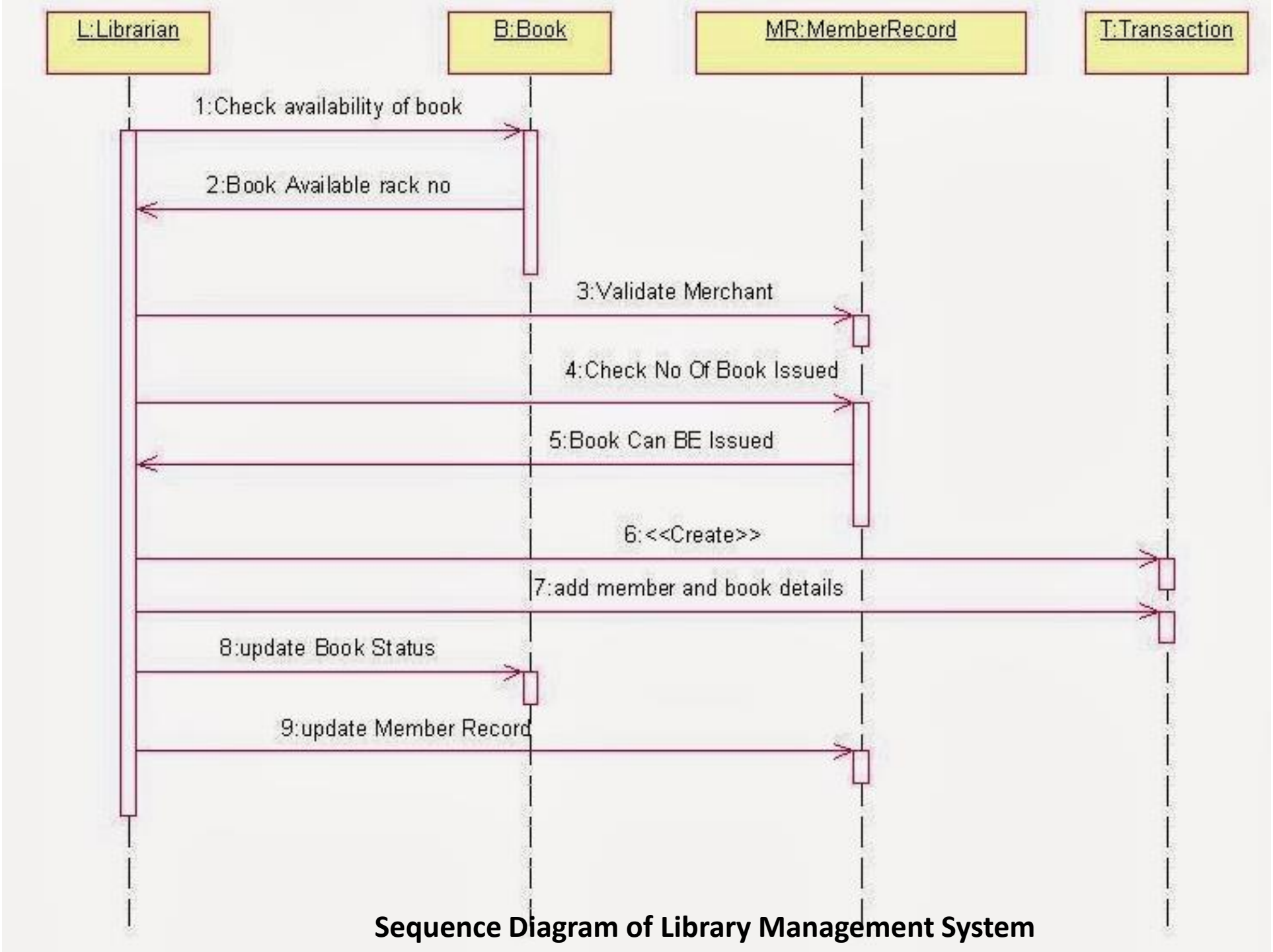
GLA UNIVERSITY MATHURA
Recognised by UGC Under Section 2(f)
Accredited with A Grade by NAAC

The object types used in this analysis model are entity objects, interface objects and control objects



Sequence Diagram

- Sequence diagram descries interaction among classes in terms of an exchange of message over time. Sequence diagram demonstrate the behavior of objects in a use case by describing the object and messages they pass. A sequence diagram depicts the sequence of actions that occurs in system. The invocation of methods in each object and the order in which they captured in a sequence diagram. This makes sequence diagram very useful.



Sequence Diagram of Library Management System

GLA UNIVERSITY MATHURA
Recognised by UGC Under Section 2(f)
Accredited with A Grade by NAAC

Draw the class diagram

The class diagram shows the relationship amongst classes. There are four types of relationships in class diagrams.

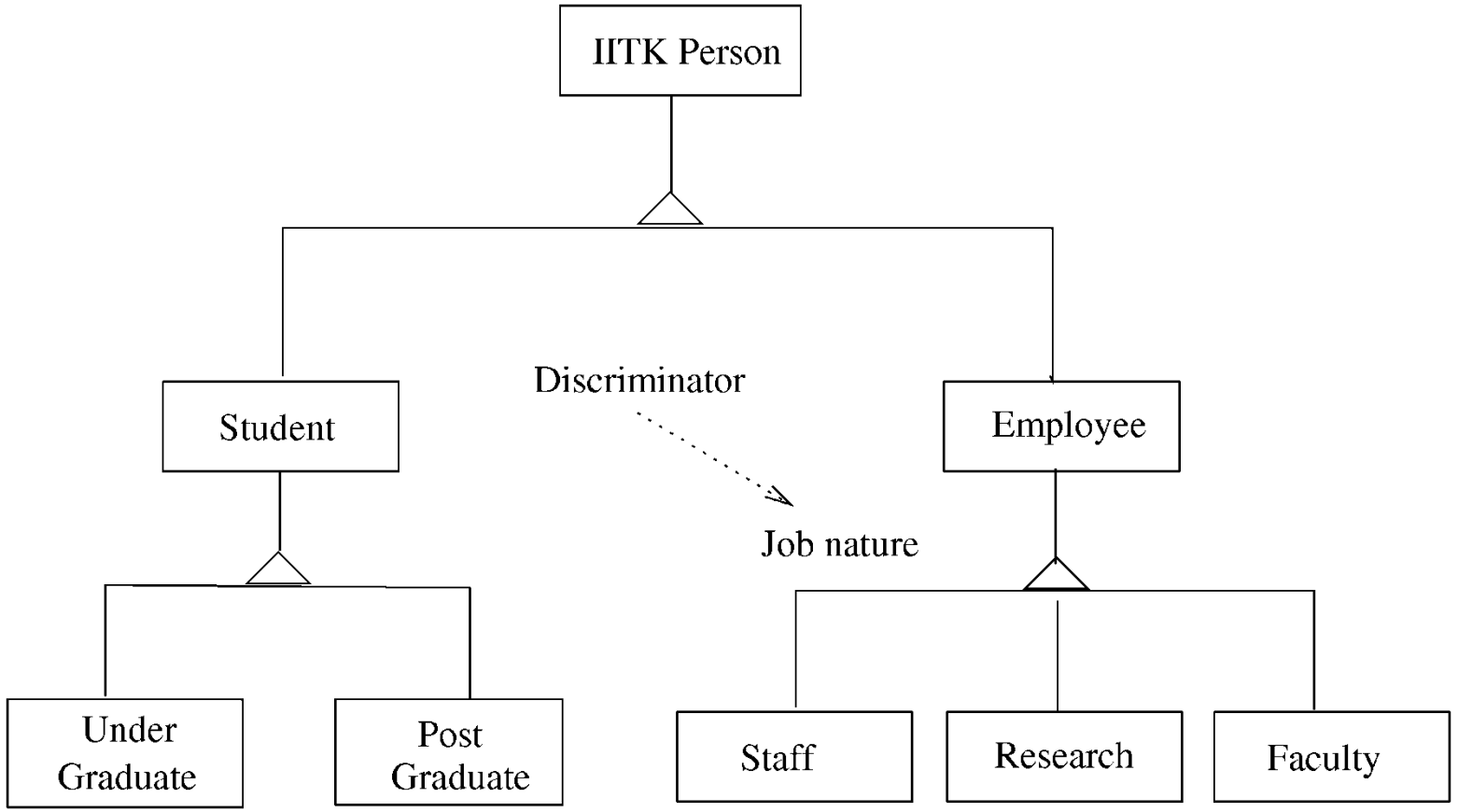
a) Association are semantic connection between classes. When an association connects two classes, each class can send messages to the other in a sequence or a collaboration diagram. Associations can be bi-directional or unidirectional.

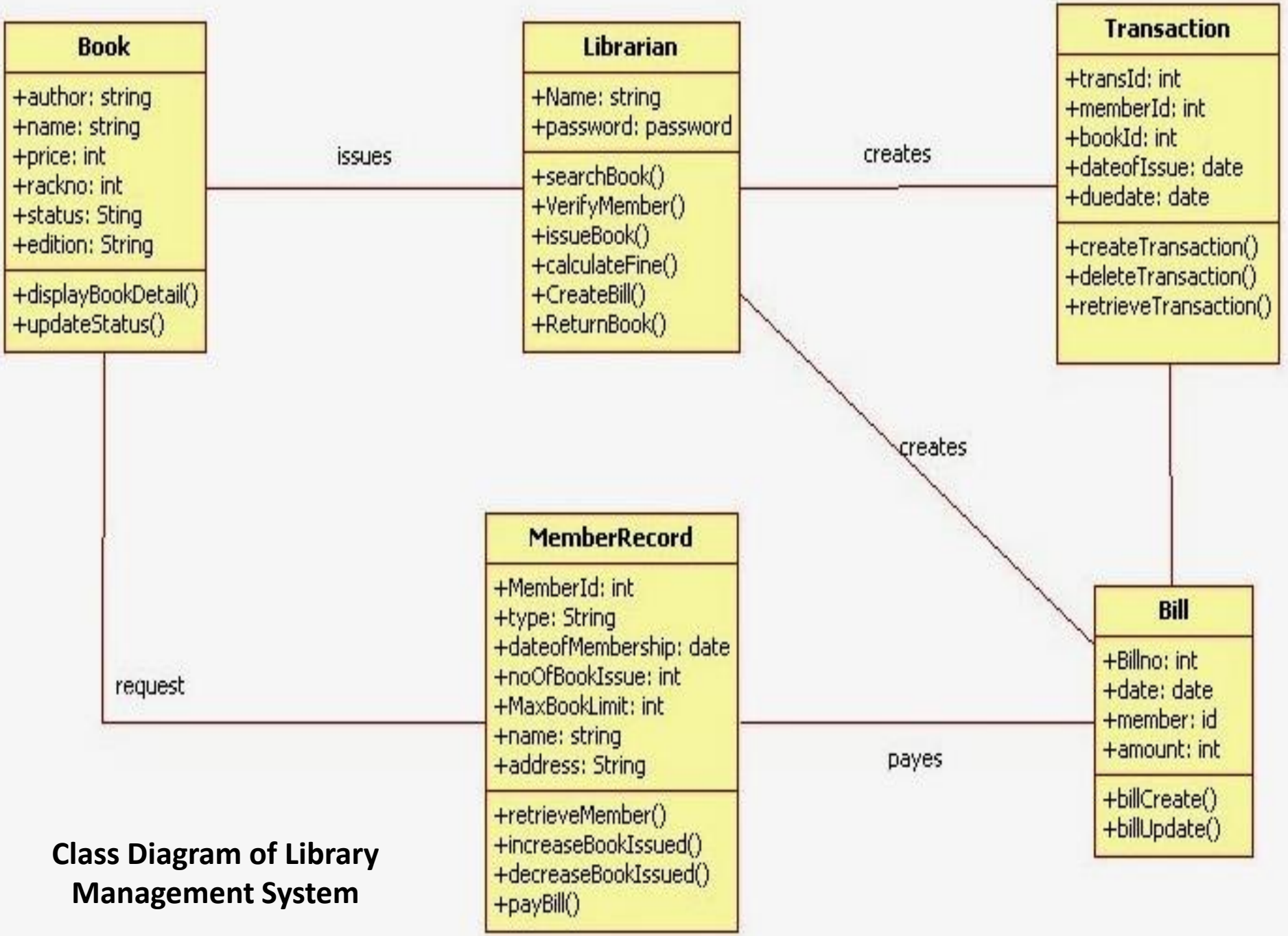
b) **Dependencies** connect two classes. Dependencies are always unidirectional and show that one class, depends on the definitions in another class.

c) **Aggregations** are stronger form of association. An aggregation is a relationship between a whole and its parts.

d) **Generalizations** are used to show an inheritance relationship between two classes.

Example – class hierarchy



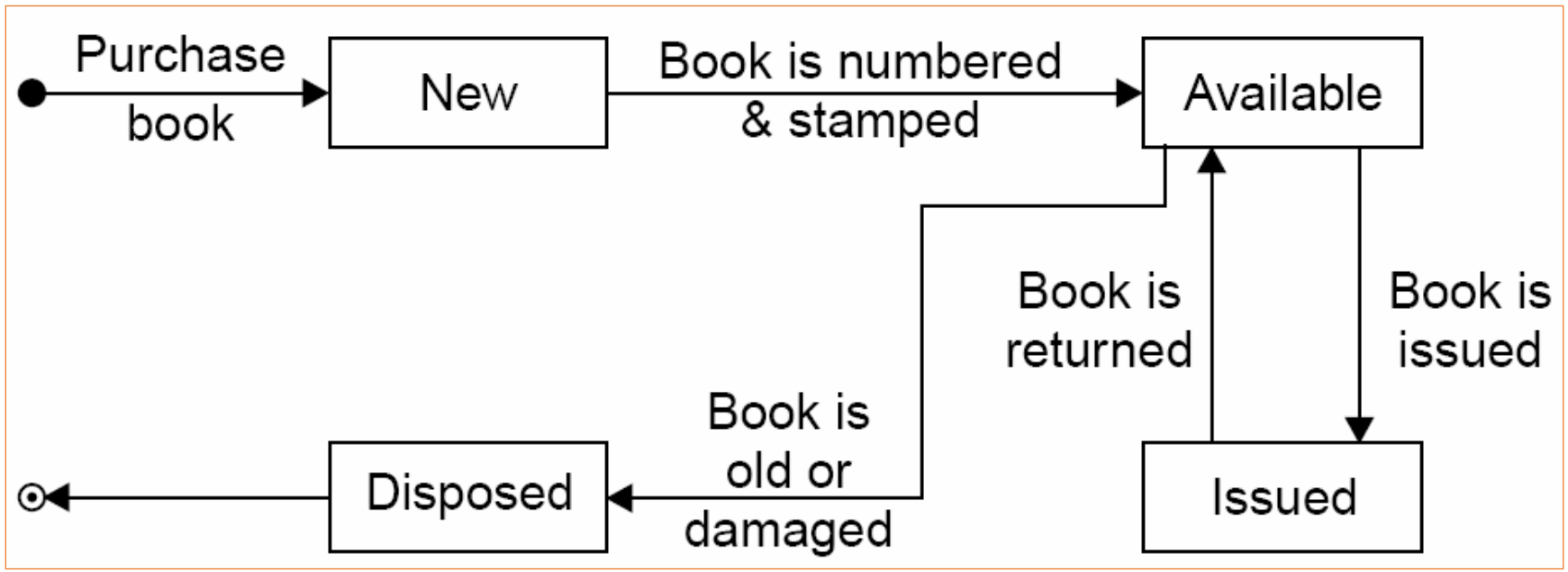


Class Diagram of Library Management System

GLA UNIVERSITY MATHURA
Recognised by UGC Under Section 2(f)
Accredited with A Grade by NAAC

Design of state chart diagrams

A state chart diagram is used to show the state space of a given class, the event that cause a transition from one state to another, and the action that result from a state change. A state transition diagram for a “book” in the library system is given



Source: Software Engineering by KK Agarwal &Yogesh Singh

GLA UNIVERSITY MATHURA
Recognised by UGC Under Section 2(f)
Accredited with A Grade by NAAC

Draw component and development diagram

Component diagrams address the static implementation view of a system they are related to class diagrams in that a component typically maps to one or more classes, interfaces or collaboration.

GLA UNIVERSITY MATHURA
Recognised by UGC Under Section 2(f)
Accredited with A Grade by NAAC

Modeling Language (UML)

Unified Modeling Language (UML) is a general purpose modeling language. The main aim of UML is to define a standard way to **visualize** the way a system has been designed. It is quite similar to blueprints used in other fields of engineering.

Unified Modeling Language (UML) and Modeling

- UML is a graphical notation useful for OO analysis and design
- Allows representing various aspects of the system
- Various notations are used to build different models for the system
- OOD methodologies use UML to represent the models they create

Modeling

- Modeling is used in many disciplines – architecture, aircraft building, …
- A model is a simplification of reality
- A good model includes those statements that have broad effect and omits minor statements
- A model of a system is not the system!

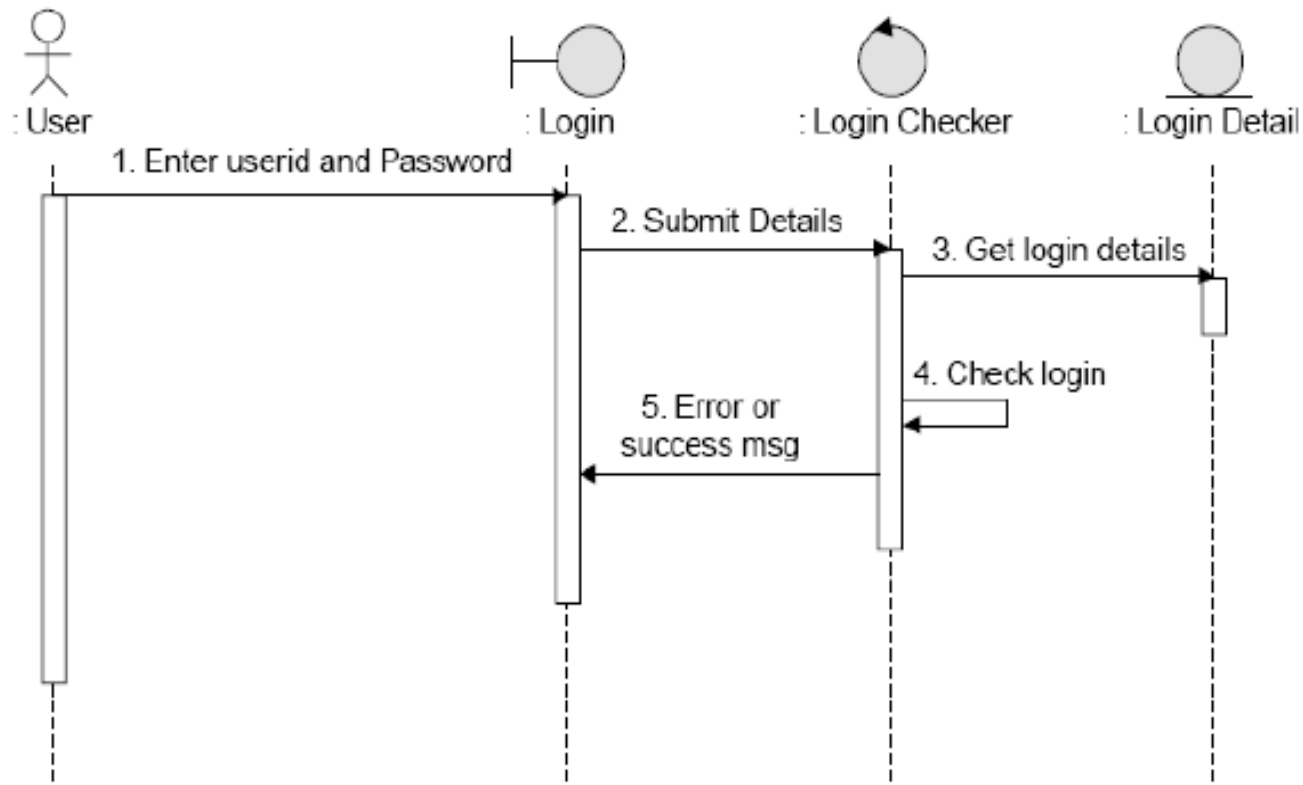
Why build models?

- Models help us visualize a system
- Help specify the system structure
- Gives us a template that can guide the construction
- Document the decisions taken and their rationale

Views in an UML

- A use case view
- A design view
- A process view
- Implementation view
- Deployment view

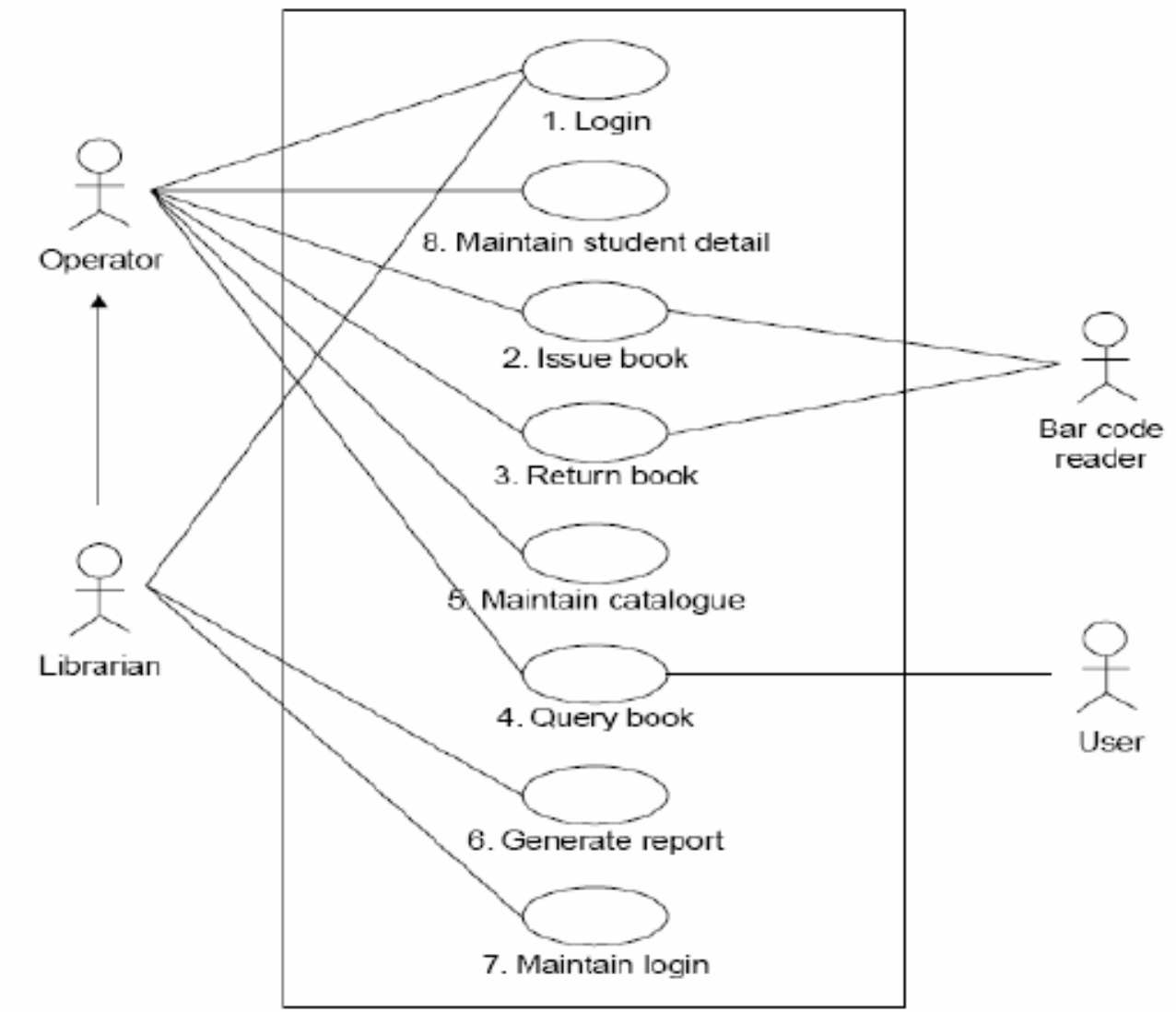
- We will focus primarily on models for design – class diagram, interaction diagram, etc.



Sequence diagram—Login

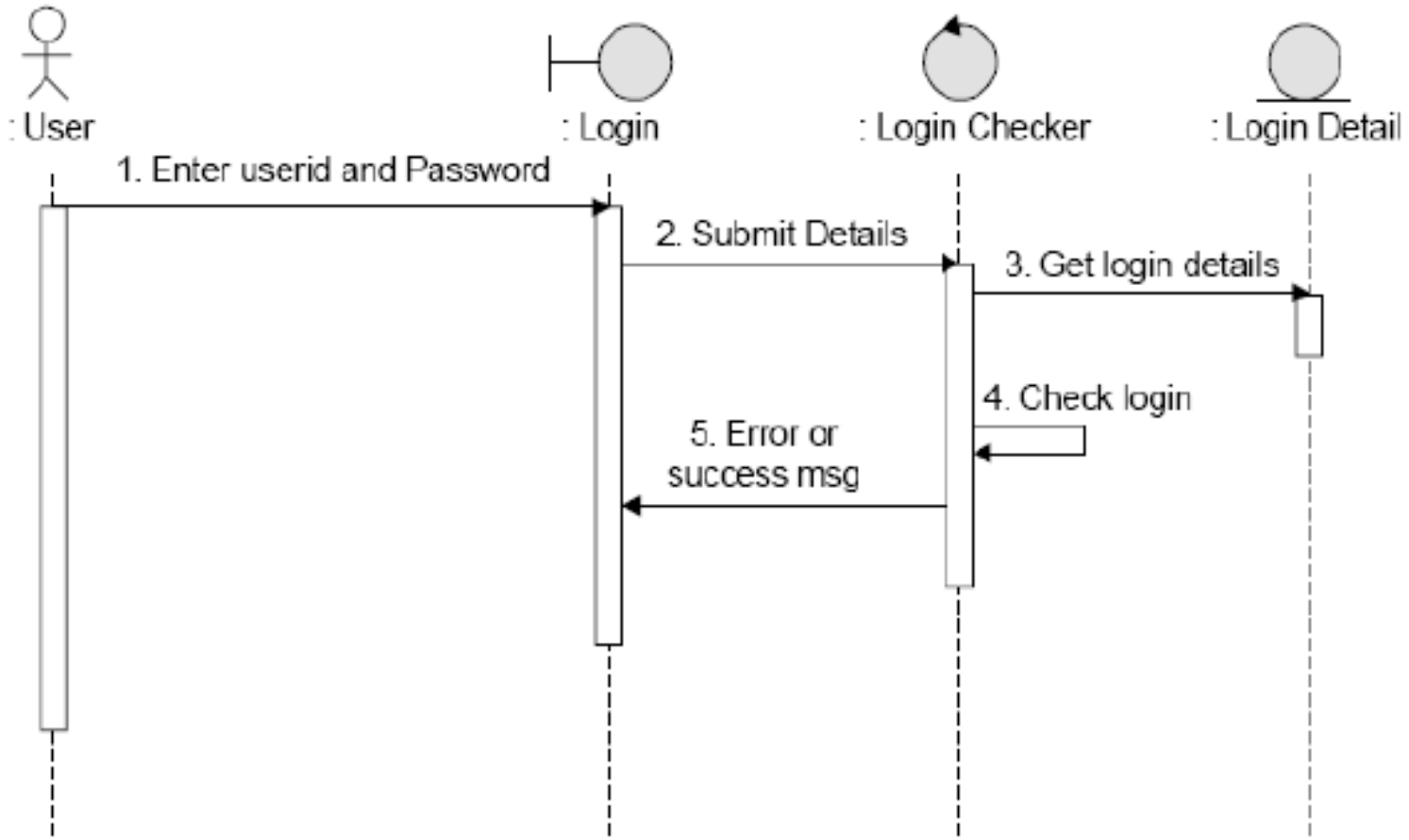
Source: Software Engineering by KK Agarwal &Yogesh Singh

Software Design



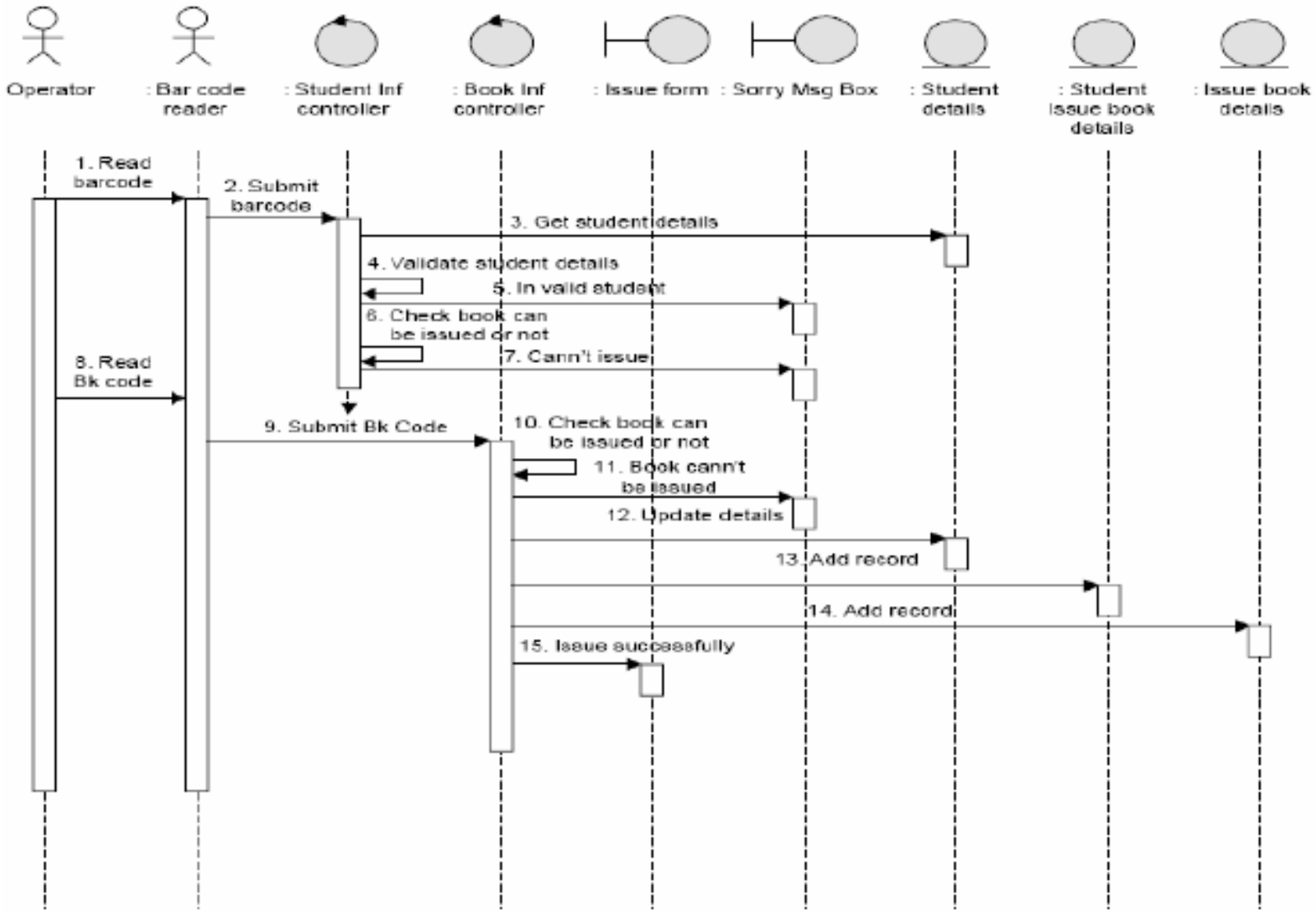
Use case diagram for library management system

Software Design



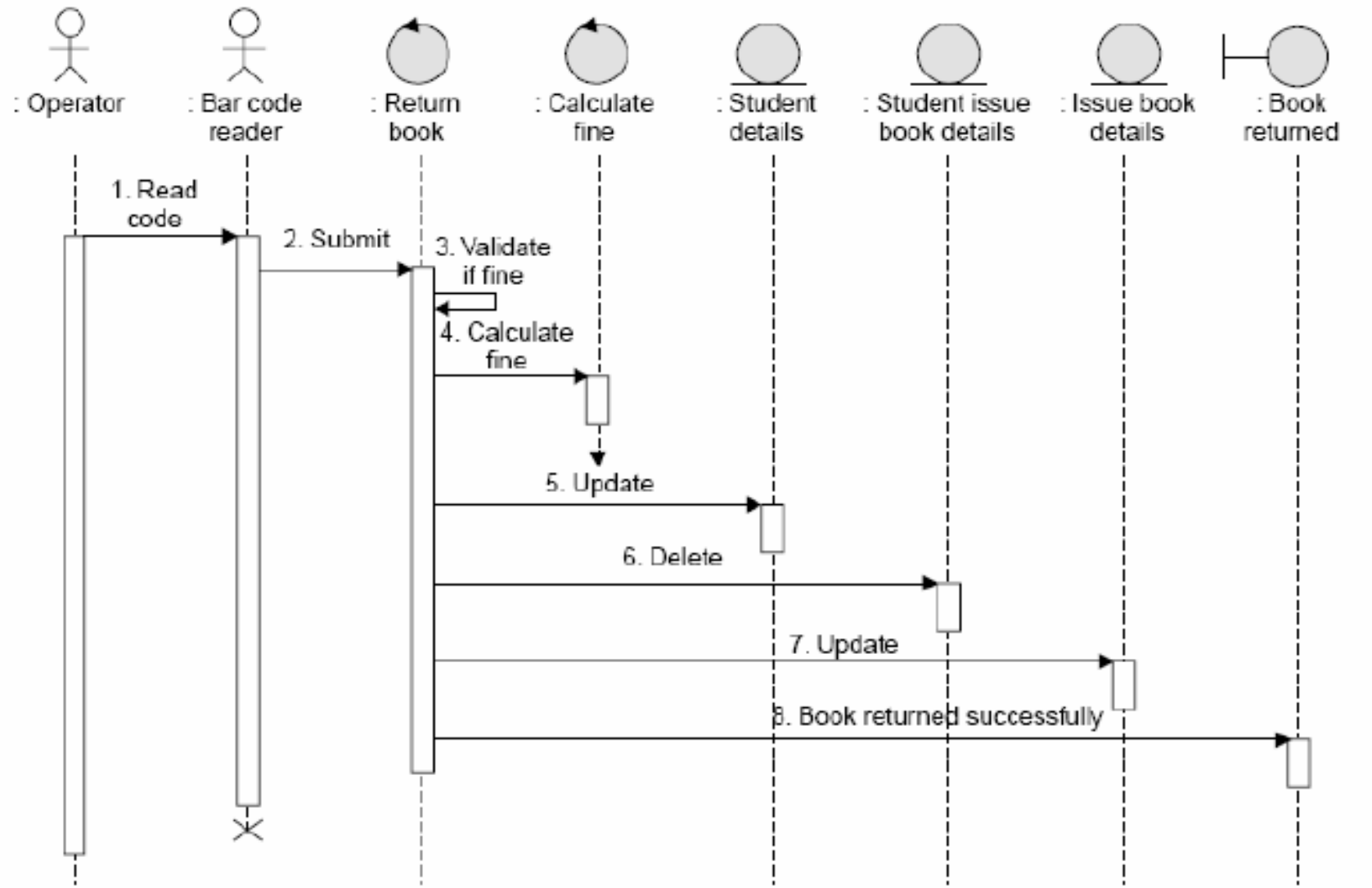
Sequence diagram—Login

Software Design

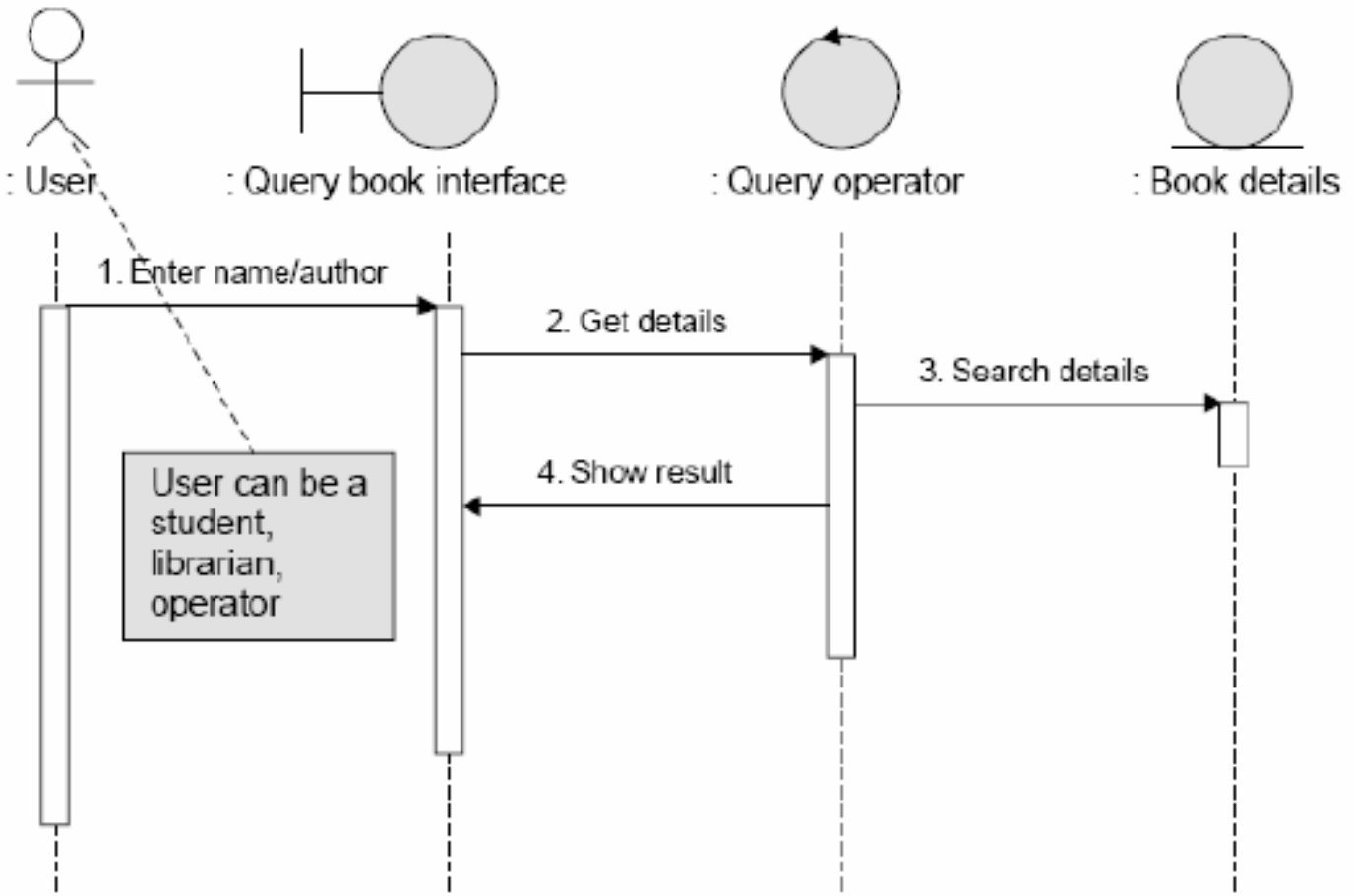


Sequence diagram—issue book

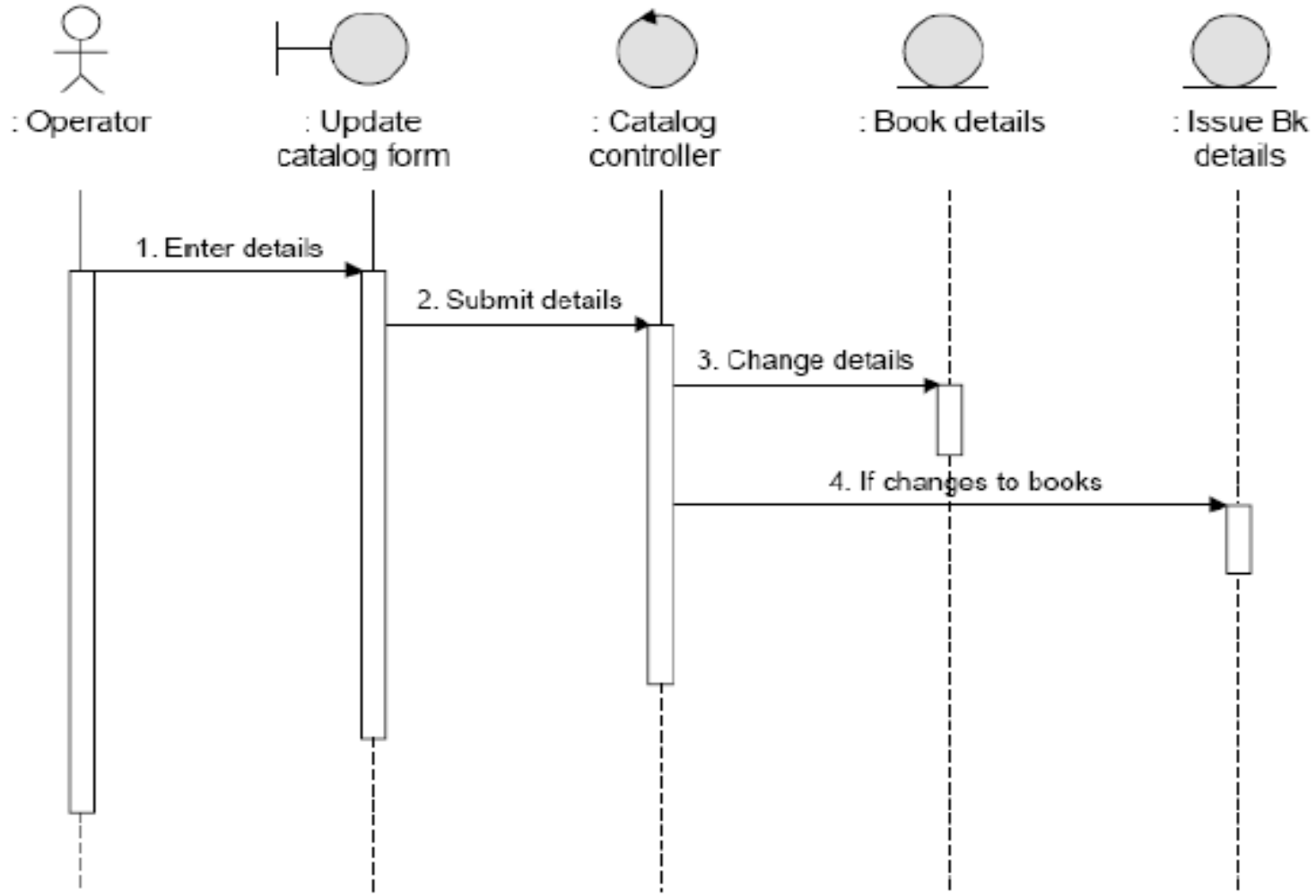
Software Design



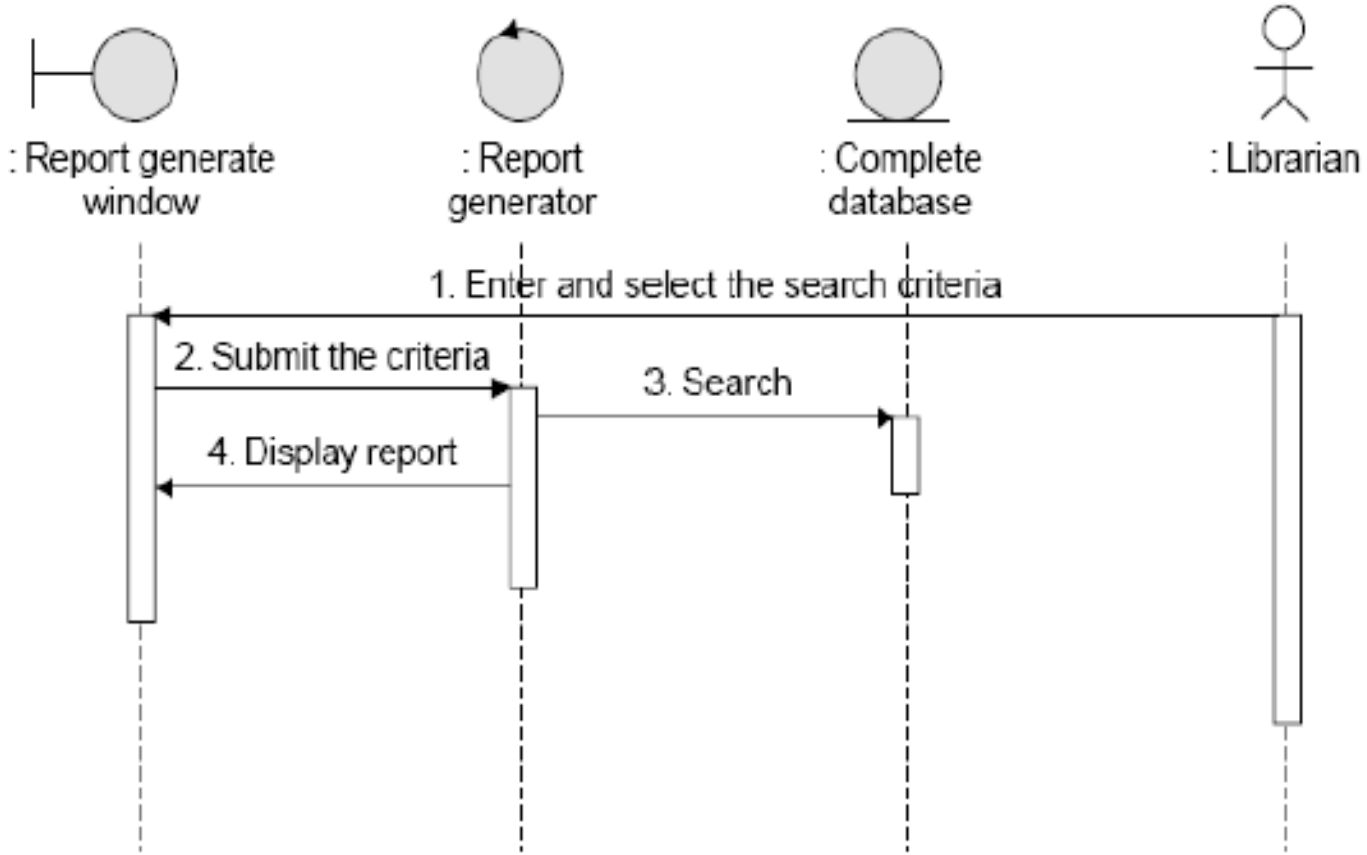
Sequence diagram—return book



Sequence diagram—query book

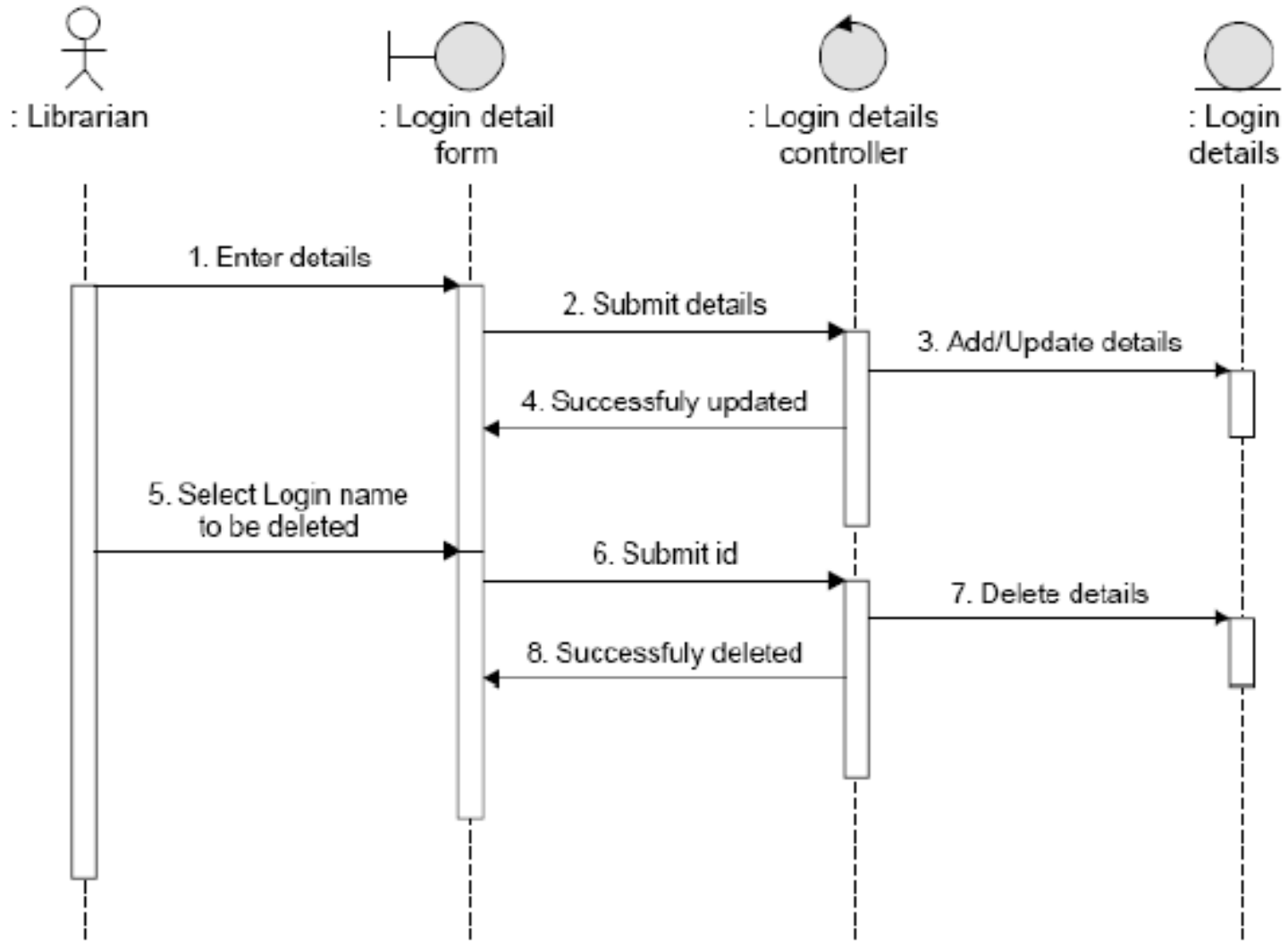


Sequence diagram—maintain catalog



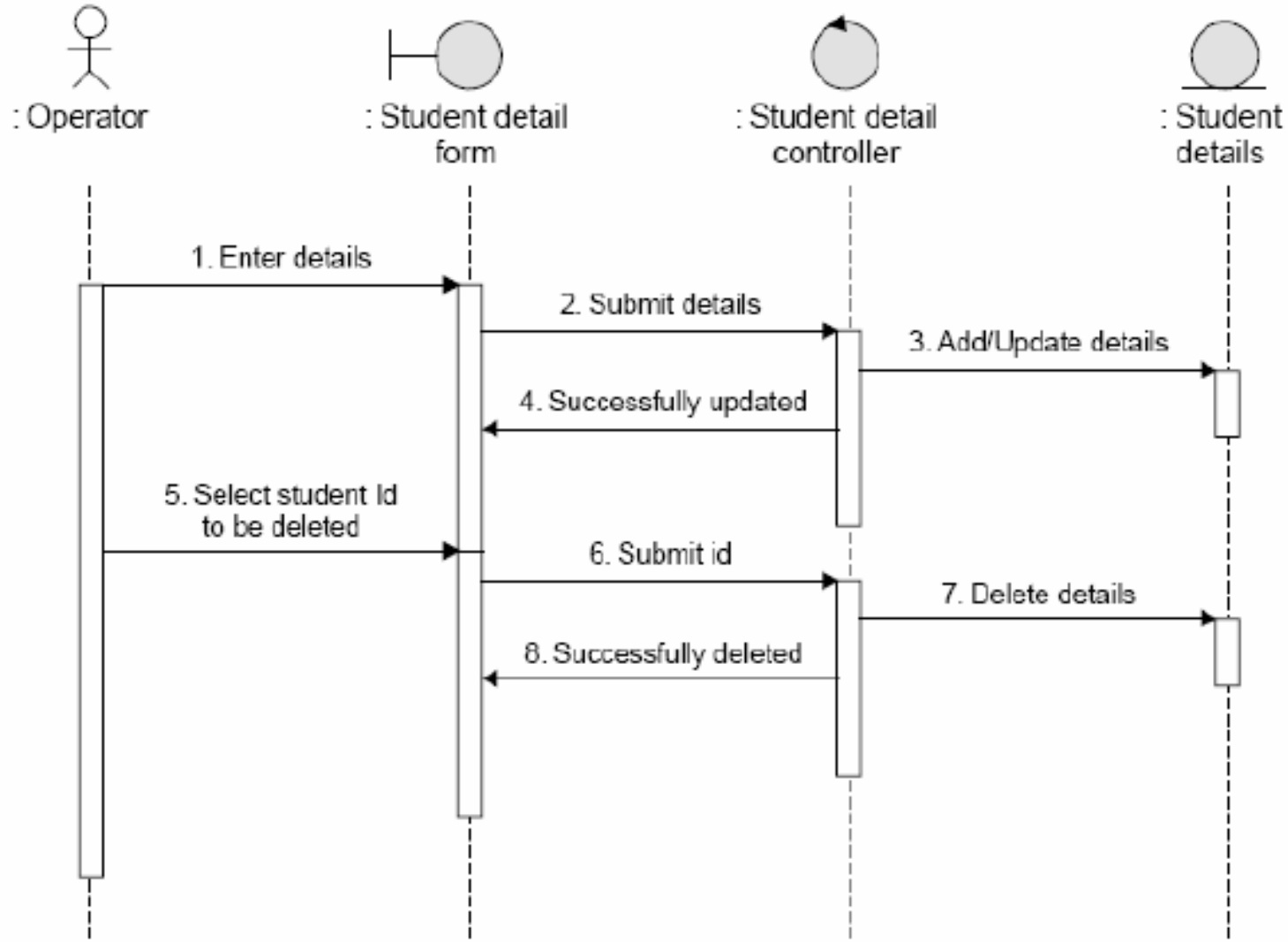
Sequence diagram—generate reports

Software Design



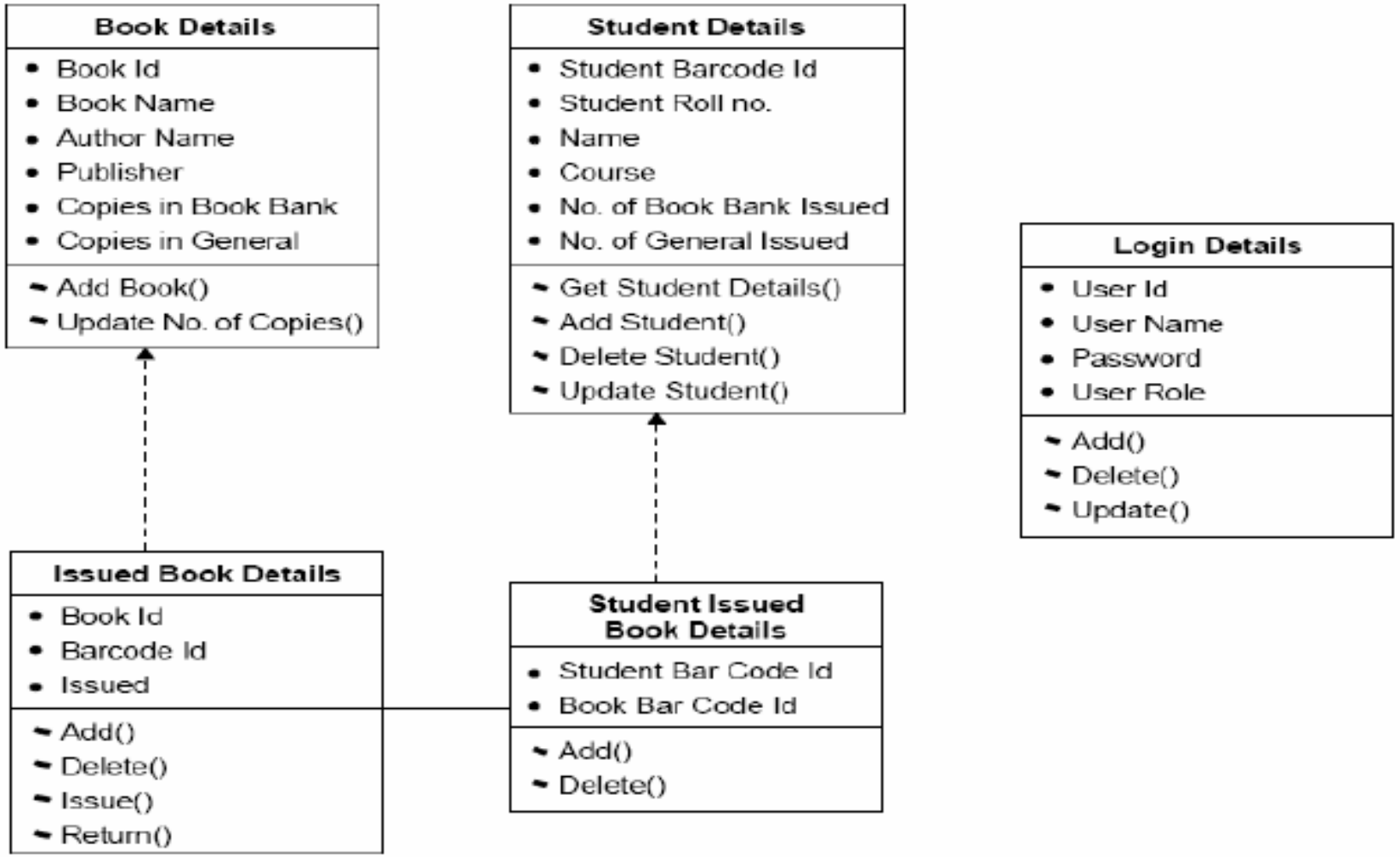
Sequence diagram—maintain login

Software Design



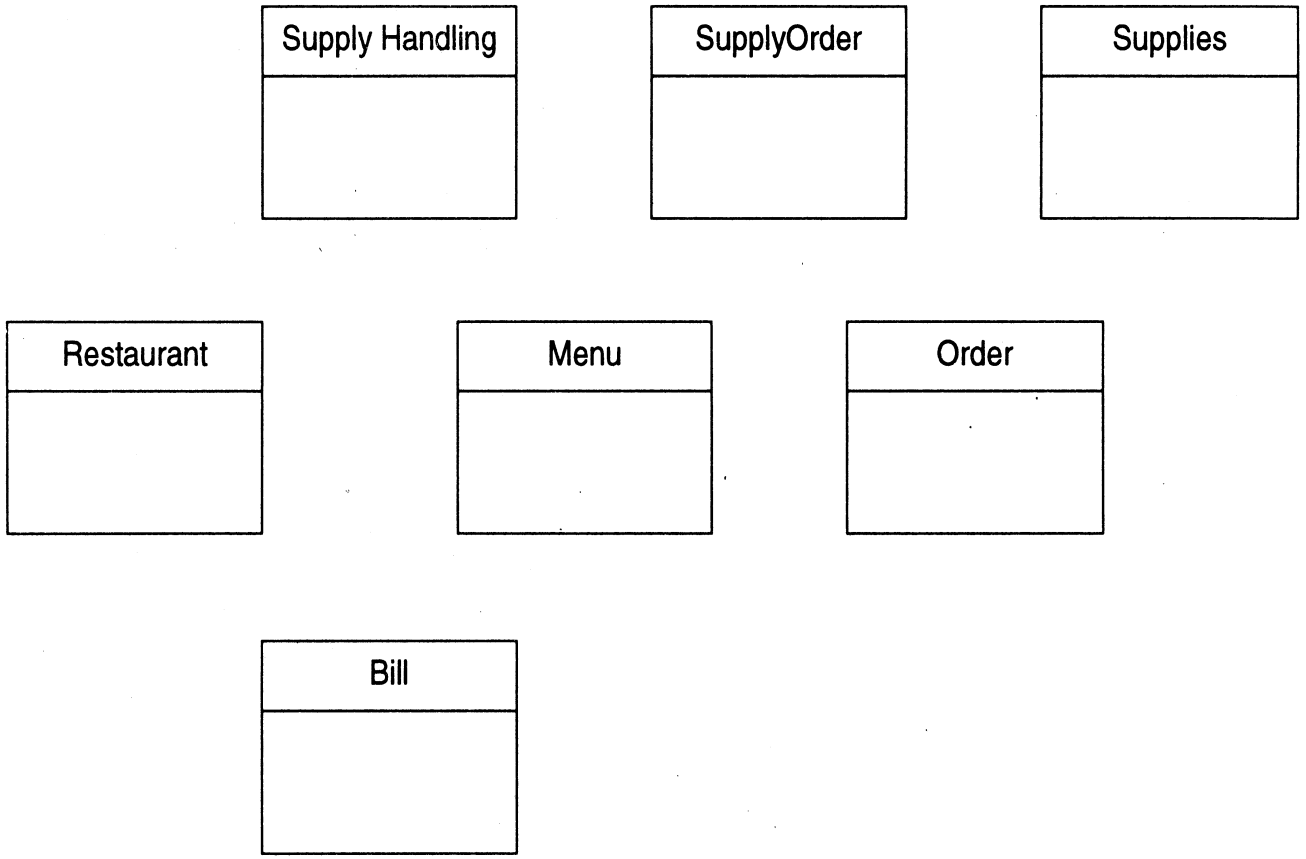
Sequence diagram—maintain student details

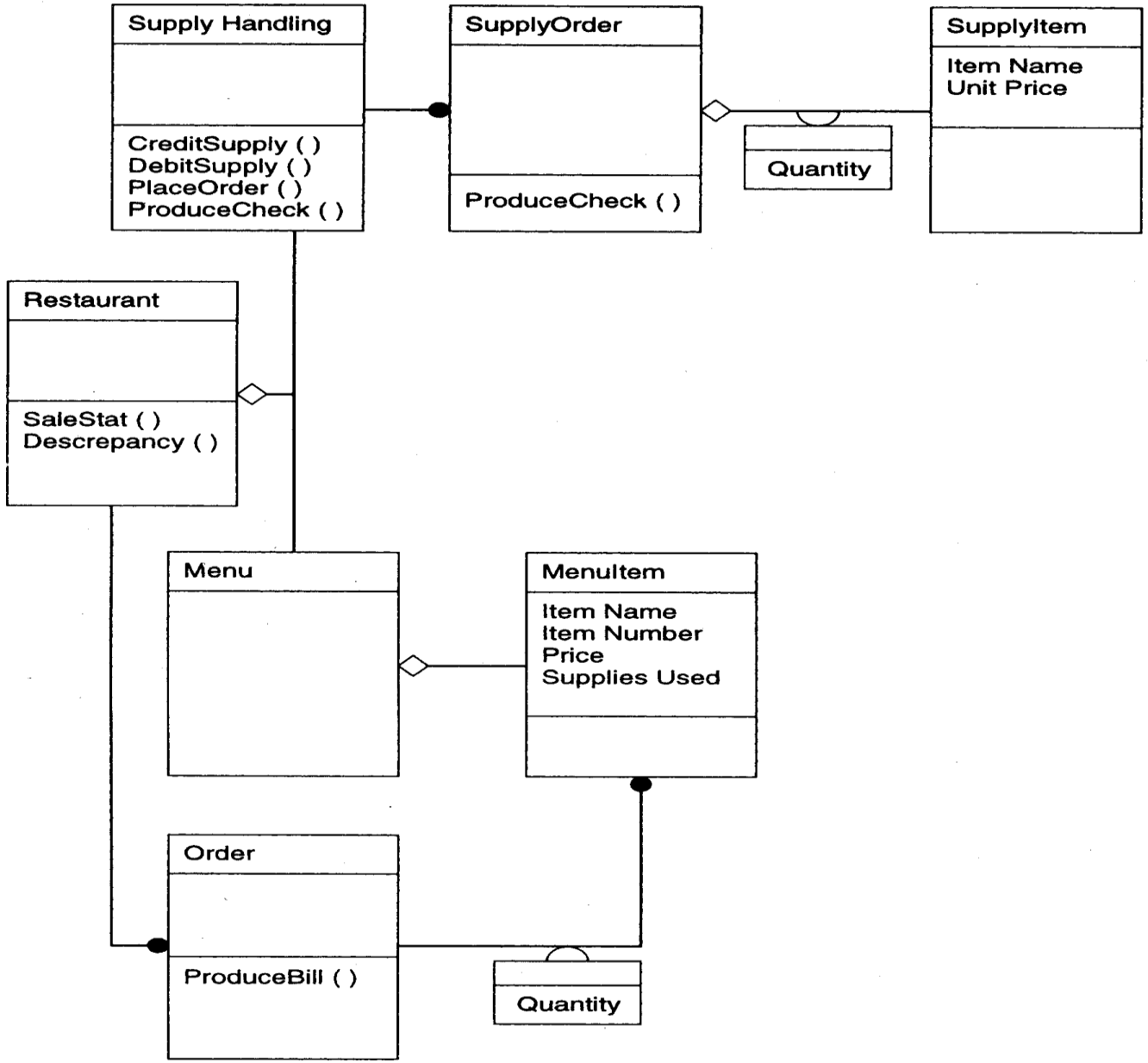
Class diagram of entity classes



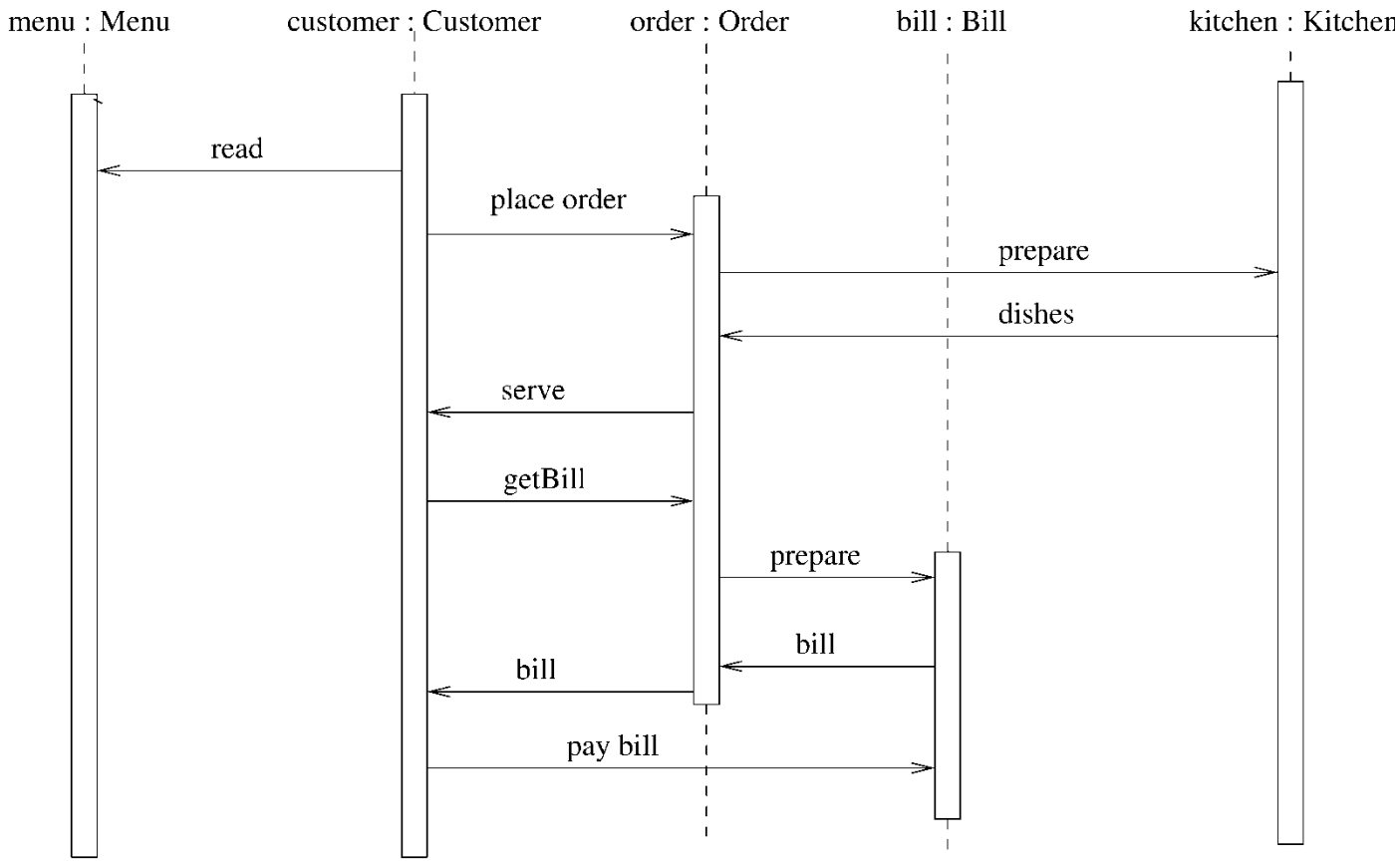
Class diagram of entity classes

Restaurant example: Initial classes




Supply Handling
CreditSupply ( )
DebitSupply ( )
PlaceOrder ( )
ProduceCheck ( )
SupplyOrder
ProduceCheck ( )
Quantity
SupplyItem
Item Name
Unit Price
Restaurant
SaleStat ( )
Descrepancy ( )
Menu
MenuItem
Item Name
Item Number
Price
Supplies Used
Order
ProduceBill ( )
Quantity

Restaurant example: a seq diag



GLA UNIVERSITY MATHURA
Recognised by UGC Under Section 2(f)
Accredited with A Grade by NAAC

Coding

The coding is the process of transforming the design of a system into a computer language format. This coding phase of software development is concerned with software translating design specification into the source code. It is necessary to write source code & internal documentation so that conformance of the code to its specification can be easily verified.

Coding is done by the coder or programmers who are independent people than the designer. The goal is not to reduce the effort and cost of the coding phase, but to cut to the cost of a later stage. The cost of testing and maintenance can be significantly reduced with efficient coding.

Source: https://www.javatpoint.com/

Goals of Coding

To translate the design of system into a computer language format: The coding is the process of transforming the design of a system into a computer language format, which can be executed by a computer and that perform tasks as specified by the design of operation during the design phase.

To reduce the cost of later phases: The cost of testing and maintenance can be significantly reduced with efficient coding.

Making the program more readable: Program should be easy to read and understand. It increases code understanding having readability and understandability as a clear objective of the coding activity can itself help in producing more maintainable software.

Source: https://www.javatpoint.com/

Verification

It is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.

Validation

It is the process of evaluating a system or component during or at the end of development process to determine whether it satisfies the specified requirements .

Testing= Verification +Validation

GLA UNIVERSITY MATHURA
Recognised by UGC Under Section 2(f)
Accredited with A Grade by NAAC

Code Inspection

Code Inspection is the most formal type of review, which is a kind of static testing to avoid the defect multiplication at a later stage.

The main purpose of code inspection is to find defects and it can also spot any process improvement if any.

An inspection report lists the findings, which include metrics that can be used to aid improvements to the process as well as correcting defects in the document under review.

Preparation before the meeting is essential, which includes reading of any source documents to ensure consistency.

Inspections are often led by a trained moderator, who is not the author of the code.

The inspection process is the most formal type of review based on rules and checklists and makes use of entry and exit criteria.

It usually involves peer examination of the code and each one has a defined set of roles.

After the meeting, a formal follow-up process is used to ensure that corrective actio

Source: https://www.tutorialspoint.com/

Metrics

A metric is a quantitative measure of the degree to which a system, component or process possesses a given attribute.

GLA UNIVERSITY MATHURA
Recognised by UGC Under Section 2(f)
Accredited with A Grade by NAAC

Use of Software Metrics

1.How to measure the size of a software?
2. How much will it cost to develop a software?
3. How many bugs can we expect?
4. When can we stop testing?
5. When can we release the software?
6. What is the complexity of a module?
7. What is the module strength and coupling?
8. What is the reliability at the time of release?
9. Which test technique is more effective?
10. Are we testing hard or are we testing smart?
11. Do we have a strong program or a week test suite?

Different Definitions of Metrics

Pressman explained as "A measure provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of the product or process".

Fenton defined measurement as " it is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules".

Software Metrics

Software metrics can be defined as “The continuous application of measurement based techniques to the software development process and its products to supply meaningful and timely management information, together with the use of those techniques to improve that process and its products”.

Categories of Metrics

1:Product metrics: Describe the characteristics of the product such as size, complexity, design features, performance, efficiency, reliability, portability, etc.

2:Process metrics: Describe the effectiveness and quality of the processes that produce the software product.

Examples are:

Effort required in the process

Time to produce the product

Effectiveness of defect removal during development

Number of defects found during testing

Maturity of the process

GLA UNIVERSITY MATHURA
Recognised by UGC Under Section 2(f)
Accredited with A Grade by NAAC

Continue…

3.Project metrics: Describe the project characteristics and execution.

Examples are :

Number of software developers

Staffing pattern over the life cycle of the software

Cost and schedule

Productivity

GLA UNIVERSITY MATHURA
Recognised by UGC Under Section 2(f)
Accredited with A Grade by NAAC

Token Count

Halstead Vocabulary

The size of the vocabulary of a program, which consists of the number of unique tokens used to build a program is defined as:

number of unique tokens used to build a program is defined as:

$\eta = \eta 1 + \eta 2$

η : vocabulary of a program

η1 : number of unique operators

η2: number of unique operands

GLA UNIVERSITY MATHURA
Recognised by UGC Under Section 2(f)
Accredited with A Grade by NAAC

Length of the program

Halstead Program Length

The length of the program in the terms of the total number of tokens used is:

N = N1+N2

N : program length

N1 : total occurrences of operators

N2 : total occurrences of operands

Estimated Program Length

$\hat{N} = \eta 1 \log_2 \eta 1 + \eta 2 \log_2 \eta 2$

Volume

$V = N * \log_2 \eta$

The unit of measurement of volume is the common unit for size “bits”. It is the actual size of a program if a uniform binary encoding for the vocabulary is used.

Potential Volume

$V^* = (2+\eta_2^*) \log_2 (2+\eta_2^*)$

unique input and output parameters are represented by **η_2^***

Program Level

$L = V^* / V$

The value of L ranges between zero and one,

with L=1 representing a program written at the highest possible level (i.e., with minimum size).

Program Difficulty

$$D = 1 / L$$

As the volume of an implementation of a program increases, the program level decreases and the difficulty increases. Thus, programming practices such as redundant usage of operands, or the failure to use higher-level control constructs will tend to increase the volume as well as the difficulty.

Effort

$$E = V / L = D * V$$

The unit of measurement of E is elementary mental discriminations.

Potential Volume=$V^*=(2+\eta_2^*)\log_2(2+\eta_2^*)$

Estimated Program level

$\hat{L}=2\eta_2/(\eta 1 N2)$

Estimated Program Difficulty

$\hat{D}=1/\hat{L}$

Effort and Time

E(Effort)

$E = V / L^{\wedge} = V * D^{\wedge}$

T(Time)

E / β

Normally beta is set to 18

Language Level

$$\lambda = L \times V^* = {}^{2}L_V$$

GLA UNIVERSITY MATHURA
Recognised by UGC Under Section 2(f)
Accredited with A Grade by NAAC

Counting rules for C language

1.Comments are not considered.

2. The identifier and function declarations are not considered.

3. All the variables and constants are considered operands.

4. Global variables used in different modules of the same program are counted as multiple occurrences of the same variable.

5.Local variables with the same name in different functions are counted as unique operands.

6. Functions calls are considered as operators.

7. All looping statements e.g., do {…} while ( ), while ( ) {…}, for ( ) {…}, all control statements e.g., if ( ) {…}, if ( ) {…} else {…}, etc. are considered as operators.

8. In control construct switch ( ) {case:…}, switch as well as all the case statements are considered as operators.

The reserve words like return, default, continue, break, sizeof, etc., are considered as operators.

10. All the brackets, commas, and terminators are considered as operators.

11.GOTO is counted as an operator and the label is counted as an operand.

12. The unary and binary occurrence of “+” and “-” are dealt separately. Similarly “*” (multiplication operator) are dealt with separately.

13.In the array variables such as “array-name [index]” “arrayname” and “index” are considered as operands and [ ] is considered as operator.

14. In the structure variables such as “struct-name, member-name” or “struct-name -> member-name”, struct-name, member-name are taken as operands and ‘.’, ‘->’ are taken as operators. Some names of member elements in different structure variables are counted as unique operands.

15. All the hash directive are ignored.

Pr

GLA UNIVERSITY MATHURA
Recognised by UGC Under Section 2(f)
Accredited with A Grade by NAAC

Consider the sorting program as mentioned below. List out the operators and operands and also calculate the values of software science measures like

η, N, V, E, λ

operators	occurrences	operands	occurrences
int	4	sort	1
()	5	x	7
,	4	n	3
[]	7	i	8
if	2	j	7
<	2	save	3
;	11	im1	3
for	2	2	2
=	6	1	3
–	1	0	1
<=	2	–	–
++	2	–	–
return	2	–	–
{}	3	–	–
n1=14	N1=53	n2=10	N2=38

List of Operands & Operators

Operators	Occurrence	Operands	Occurrence
Int	4	SORT	1
()	5	x	7
,	4	n	3
[]	7	i	8
If	2	j	7
<	2	save	3
;	11	im1	3
for	2	2	2
=	6	1	3
-	1	0	1
<=	2	-	-
++	2	-	-
return	2	-	-
{}	3	-	-

$\eta_1 = 14$ $N_1 = 53$ $\eta_2 = 10$ $N_2 = 38$

Program Length: N=N1+N2=53+38=91

Vocabulary of the program: η1+ η2=14+10=24

Volume: V=N X $\log_2 \eta 2 = 91 \times \log_2 24 = 417$ bits

The estimated program length $\hat{N}$

$\eta 1 \log_2 \eta 1 + \eta 2 \log_2 \eta 2$

$14 \log_2 14 + 10 \log_2 10 = 86.45$

Potential Volume=$V^* = (2+\eta_2^*) \log_2 (2+\eta_2^*)$

unique input and output parameters are represented by η_2^*

$\eta_2^* = 3$ so $V^* = (2+3) \log_2 (2+3) = 11.6$

Program Level

L = V* / V=11.6/417=0.027

Program Difficulty

D = 1 / L =1/0.027=37.03

Estimated program level $\hat{L} = \frac{2}{\eta_1} \times \frac{\eta_2}{N_2} = \frac{2}{14} \times \frac{10}{38}$

$= 0.038$

Lal alstead

Language	Language Level	Variance
PL/1	1.53	0.92
ALGOL	1.21	0.74
FORTRAN	1.14	0.81
CDC Assembly	0.88	0.42
PASCAL	2.54	----
APL	2.42	----
C	0.857	0.445

Software Testing

- What is Testing?

Many people understand many definitions of testing :

1. Testing is the process of demonstrating that errors are not present.
2. The purpose of testing is to show that a program performs its intended functions correctly.
3. Testing is the process of establishing confidence that a program does what it is supposed to do.

These definitions are incorrect.

Software Testing

A more appropriate definition is:

"*Testing is the process of executing a program with the intent of finding errors.*"

Software Testing

- Why should We Test ?

Although software testing is itself an expensive activity, yet launching of software without testing may lead to cost potentially much higher than that of testing, specially in systems where human safety is involved.

In the software life cycle the earlier the errors are discovered and removed, the lower is the cost of their removal.

Software Testing

- Who should Do the Testing ?
  - Testing requires the developers to find errors from their software.
  - It is difficult for software developer to point out errors from own creations.
  - Many organisations have made a distinction between development and testing phase by making different people responsible for each phase.

Some Terminologies

- **Error, Mistake, Bug, Fault and Failure**

People make **errors**. A good synonym is **mistake**. This may be a syntax error or misunderstanding of specifications. Sometimes, there are logical errors.

When developers make mistakes while coding, we call these mistakes "**bugs**".

A **fault** is the representation of an error, where representation is the mode of expression, such as narrative text, data flow diagrams, ER diagrams, source code etc. Defect is a good synonym for fault.

A **failure** occurs when a fault executes. A particular fault may cause different failures, depending on how it has been exercised.

Software Testing

➢ Test, Test Case and Test Suite

Test and **Test case** terms are used interchangeably. In practice, both are same and are treated as synonyms. Test case describes an input description and an expected output description.

Test Case ID	
Section-I (Before Execution)	Section-II (After Execution)
Purpose :	Execution History:
Pre condition: (If any)	Result:
Inputs:	If fails, any possible reason (Optional);
Expected Outputs:	Any other observation:
Post conditions:	Any suggestion:
Written by:	Run by:
Date:	Date:

Fig. 2: Test case template

The set of test cases is called a **test suite**. Hence any combination of test cases may generate a test suite.

Software Testing

- Verification and Validation

Verification is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.

Validation is the process of evaluating a system or component during or at the end of development process to determine whether it satisfies the specified requirements .

Testing= Verification+Validation

Software Testing

➢ Alpha, Beta and Acceptance Testing

The term **Acceptance Testing** is used when the software is developed for a specific customer. A series of tests are conducted to enable the customer to validate all requirements. These tests are conducted by the end user / customer and may range from adhoc tests to well planned systematic series of tests.

The terms alpha and beta testing are used when the software is developed as a product for anonymous customers.

Alpha Tests are conducted at the developer's site by some potential customers. These tests are conducted in a controlled environment. Alpha testing may be started when formal testing process is near completion.

Beta Tests are conducted by the customers / end users at their sites. Unlike alpha testing, developer is not present here. Beta testing is conducted in a real environment that cannot be controlled by the developer.

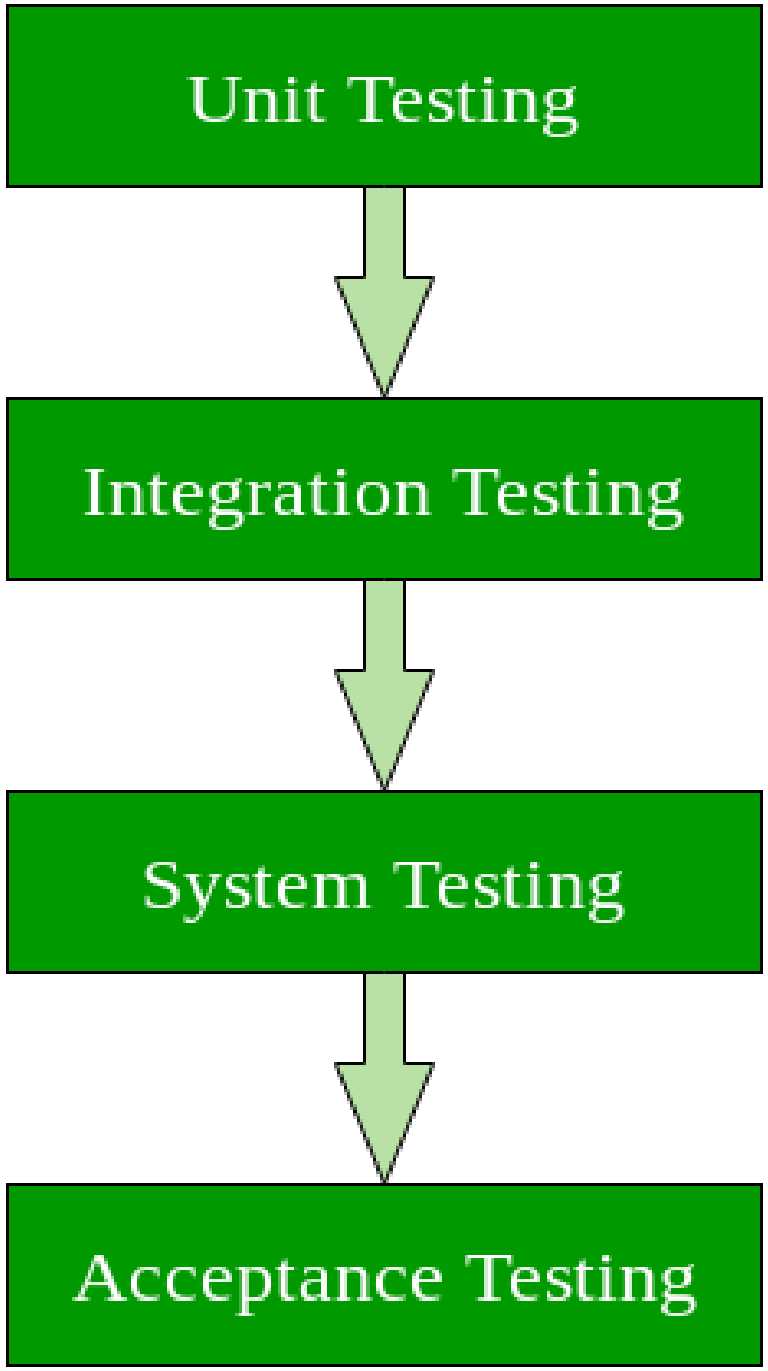
What are different types of software testing?

1. **Manual Testing:** Manual testing includes testing software manually, i.e., without using any automated tool or any script. In this type, the tester takes over the role of an end-user and tests the software to identify any unexpected behavior or bug. There are different stages for manual testing such as unit testing, integration testing, system testing, and user acceptance testing.

2.**Automation Testing:** Automation testing, which is also known as Test Automation, is when the tester writes scripts and uses another software to test the product. This process involves the automation of a manual process. Automation Testing is used to re-run the test scenarios that were performed manually, quickly, and repeatedly.

What are different levels of software testing?

1. **Unit Testing:** A level of the software testing process where individual units/components of a software/system are tested. The purpose is to validate that each unit of the software performs as designed.
2. **Integration Testing:** A level of the software testing process where individual units are combined and tested as a group. The purpose of this level of testing is to expose faults in the interaction between integrated units.
3. **System Testing:** A level of the software testing process where a complete, integrated system/software is tested. The purpose of this test is to evaluate the system's compliance with the specified requirements.
4. **Acceptance Testing:** A level of the software testing process where a system is tested for acceptability. The purpose of this test is to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery.


Unit Testing
Integration Testing
System Testing
Acceptance Testing

Software techniques can be majorly classified into two categories

1. **Black Box Testing:** The technique of testing in which the tester doesn't have access to the source code of the software and is conducted at the software interface without concern with the internal logical structure of the software is known as black-box testing.

2. **White-Box Testing:** The technique of testing in which the tester is aware of the internal workings of the product, has access to its source code, and is conducted by making sure that all internal operations are performed according to the specifications is known as white box testing.

Black Box Testing	White Box Testing
Internal workings of an application are not required.	Knowledge of the internal workings is a must.
Also known as closed box/data-driven testing.	Also known as clear box/structural testing.
End users, testers, and developers.	Normally done by testers and developers.
This can only be done by a trial and error method.	Data domains and internal boundaries can be better tested.

Types of Black Box Testing

There are many types of Black Box Testing but the following are the prominent ones –

Functional testing – This black box testing type is related to the functional requirements of a system; it is done by software testers.
Non-functional testing – This type of black box testing is not related to testing of specific functionality, but non-functional requirements such as performance, scalability, usability.
Regression testing – Regression Testing is done after code fixes, upgrades or any other system maintenance to check the new code has not affected the existing code.

Techniques Used in Black Box Testing

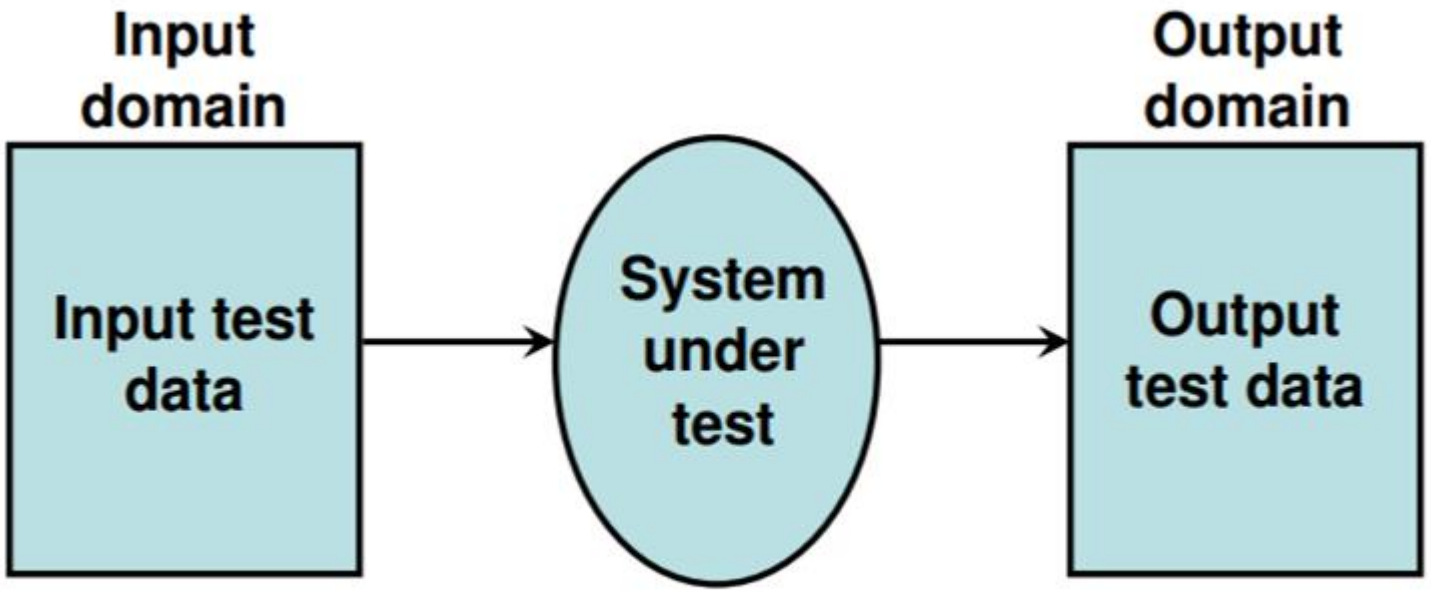
Decision Table Technique	Decision Table Technique is a systematic approach where various input combinations and their respective system behavior are captured in a tabular form. It is appropriate for the functions that have a logical relationship between two and more than two inputs.
Boundary Value Technique	Boundary Value Technique is used to test boundary values, boundary values are those that contain the upper and lower limit of a variable. It tests, while entering boundary value whether the software is producing correct output or not.
State Transition Technique	State Transition Technique is used to capture the behavior of the software application when different input values are given to the same function. This applies to those types of applications that provide the specific number of attempts to access the application.
All-pair Testing Technique	All-pair testing Technique is used to test all the possible discrete combinations of values. This combinational method is used for testing the application that uses checkbox input, radio button input, list box, text box, etc.

Techniques Used in Black Box Testing

Cause-Effect Technique	Cause-Effect Technique underlines the relationship between a given result and all the factors affecting the result.It is based on a collection of requirements.
Equivalence Partitioning Technique	Equivalence partitioning is a technique of software testing in which input data divided into partitions of valid and invalid values, and it is mandatory that all partitions must exhibit the same behavior.
Error Guessing Technique	Error guessing is a technique in which there is no specific method for identifying the error. It is based on the experience of the test analyst, where the tester uses the experience to guess the problematic areas of the software.
Use Case Technique	Use case Technique used to identify the test cases from the beginning to the end of the system as per the usage of the system. By using this technique, the test team creates a test scenario that can exercise the entire software based on the functionality of each function from start to end.

Functional testing

➔Functional Testing- testing based on functionality of the program rather than internal structure of the code (Black box testing)



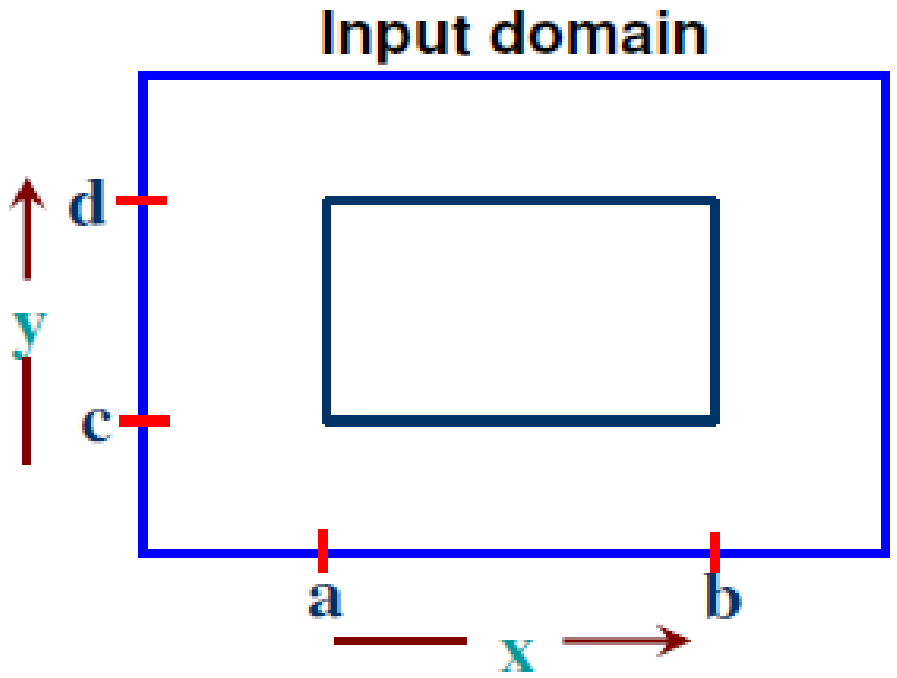
Black box testing

Boundary Value Analysis

Consider a program with two input variables x and y. These input variables have specified boundaries as:

$$a \leq x \leq b$$

$$c \leq y \leq d$$



Input domain for program having two input variables

Boundary value analysis is the best known functional testing technique.

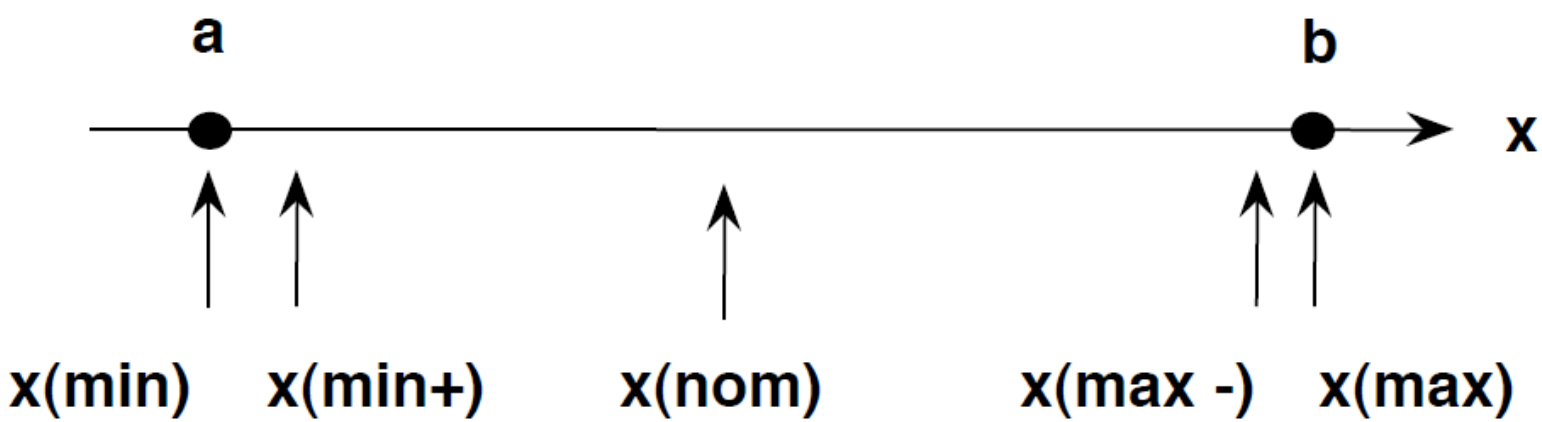
The objective of functional testing is to use knowledge of the functional nature of a program to identify test cases.

Historically, functional testing has focused on the input domain, but it is a good supplement to consider test cases based on the range as well.

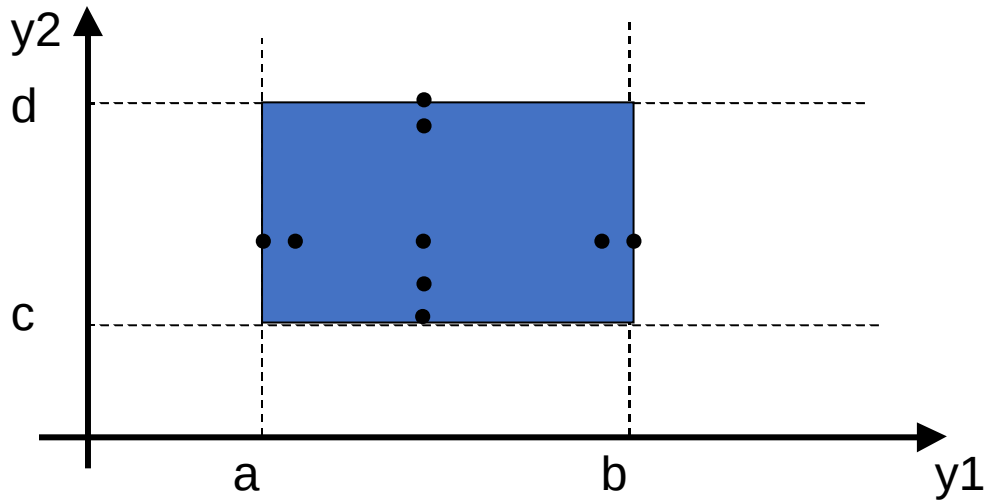
Boundary value analysis focuses on the boundary of the input space to identify test cases.

Value Selection in Boundary Value Analysis

- The basic idea in boundary value analysis is to select input variable values at their:
  - Minimum
  - Just above the minimum
  - A nominal value
  - Just below the maximum
  - Maximum

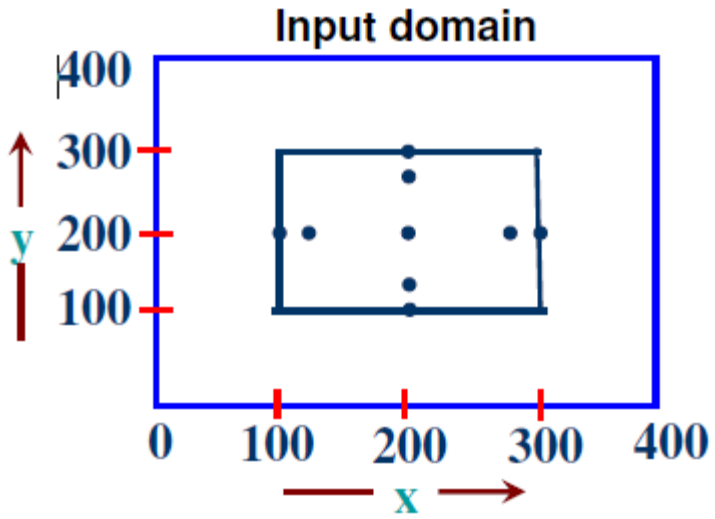

a
b
x
x(min)
x(min+)
x(nom)
x(max -)
x(max)

Boundary Value Analysis for Program



The Boundary Value Analysis

- The boundary value analysis test cases for our program with two inputs
- Variables (x and y) that may have any value from 100 to 300 are:
- (200,100),(200,101), (200,200), (200,299), (200,300), (100,200), (101,200), (299,200) and
- (300,200).
- This input domain is shown in Fig.. Each dot represent a test case
- and inner rectangle is the domain of legitimate inputs. Thus, for a program of n
- variables, boundary value analysis yield **4n + 1** test cases.



Input domain of two variables x and y with boundaries [100,300] each

Software Testing

Problem 1

Consider a program for the determination of the nature of roots of a quadratic equation. Its input is a triple of positive integers (say a,b,c) and values may be from interval [0,100]. The program output may have one of the following words.

[Not a quadratic equation; Real roots; Imaginary roots; Equal roots]

Design the boundary value test cases.

Software Testing

Solution

Quadratic equation will be of type:

$$ax^2+bx+c=0$$

Roots are real if $(b^2-4ac)>0$

Roots are imaginary if $(b^2-4ac)<0$

Roots are equal if $(b^2-4ac)=0$

Equation is not quadratic if $a=0$

The boundary value test cases are :

Test Case	*a*	*b*	*c*	*Expected output*
1	0	50	50	Not Quadratic
2	1	50	50	Real Roots
3	50	50	50	Imaginary Roots
4	99	50	50	Imaginary Roots
5	100	50	50	Imaginary Roots
6	50	0	50	Imaginary Roots
7	50	1	50	Imaginary Roots
8	50	99	50	Imaginary Roots
9	50	100	50	Equal Roots
10	50	50	0	Real Roots
11	50	50	1	Real Roots
12	50	50	99	Imaginary Roots
13	50	50	100	Imaginary Roots

Software Testing

Eample-2

Consider a program for determining the Previous date. Its input is a triple of day, month and year with the values in the range

$1 \leq month \leq 12$

$1 \leq day \leq 31$

$1900 \leq year \leq 2025$

The possible outputs would be Previous date or invalid input date. Design the boundary value test cases.

Software Testing

Solution

The Previous date program takes a date as input and checks it for validity. If valid, it returns the previous date as its output.

With single fault assumption theory, 4n+1 test cases can be designed and which are equal to 13.

Software Testing

The boundary value test cases are:

Test Case	Month	Day	Year	Expected output
1	6	15	1900	14 June, 1900
2	6	15	1901	14 June, 1901
3	6	15	1962	14 June, 1962
4	6	15	2024	14 June, 2024
5	6	15	2025	14 June, 2025
6	6	1	1962	31 May, 1962
7	6	2	1962	1 June, 1962
8	6	30	1962	29 June, 1962
9	6	31	1962	Invalid date
10	1	15	1962	14 January, 1962
11	2	15	1962	14 February, 1962
12	11	15	1962	14 November, 1962
13	12	15	1962	14 December, 1962

Software Testing

Example 3

Consider a simple program to classify a triangle. Its inputs is a triple of positive integers (say x, y, z) and the date type for input parameters ensures that these will be integers greater than 0 and less than or equal to 100. The program output may be one of the following words:

[Scalene; Isosceles; Equilateral; Not a triangle]

Design the boundary value test cases.

Solution

The boundary value test cases are shown below:

Test case	x	y	z	Expected Output
1	50	50	1	Isosceles
2	50	50	2	Isosceles
3	50	50	50	Equilateral
4	50	50	99	Isosceles
5	50	50	100	Not a triangle
6	50	1	50	Isosceles
7	50	2	50	Isosceles
8	50	99	50	Isosceles
9	50	100	50	Not a triangle
10	1	50	50	Isosceles
11	2	50	50	Isosceles
12	99	50	50	Isosceles
13	100	50	50	Not a triangle

Robustness Testing

Robustness testing

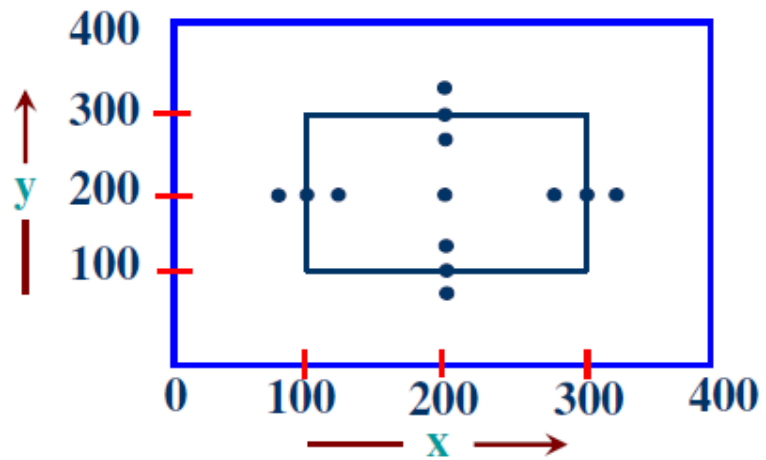
It is nothing but the extension of boundary value analysis. Here, we would like to see, what happens when the extreme values are exceeded with a value slightly greater than the maximum, and a value slightly less than minimum. It means, we want to go outside the legitimate boundary of input domain. This extended form of boundary value analysis is called robustness testing

There are four additional test cases which are outside the legitimate input domain. Hence total test cases in robustness testing are 6n+1, where n is the number of input variables. So, 13 test cases are:

(200,99), (200,100), (200,101), (200,200), (200,299), (200,300)

(200,301), (99,200), (100,200), (101,200), (299,200), (300,200), (301,200)

Robustness Testing



Robustness test cases for two variables x and y with range [100,300] each

Software Testing

Eample-2

Consider a program for determining the Previous date. Its input is a triple of day, month and year with the values in the range

$1 \leq month \leq 12$

$1 \leq day \leq 31$

$1900 \leq year \leq 2025$

The possible outputs would be Previous date or invalid input date.

Design robust testing.

Solution

Robust test cases are $6n+1$. Hence total 19 robust test cases are designed and are given on next slide.

Test case	Month	Day	Year	Expected Output
1	6	15	1899	Invalid date (outside range)
2	6	15	1900	14 June, 1900
3	6	15	1901	14 June, 1901
4	6	15	1962	14 June, 1962
5	6	15	2024	14 June, 2024
6	6	15	2025	14 June, 2025
7	6	15	2026	Invalid date (outside range)
8	6	0	1962	Invalid date
9	6	1	1962	31 May, 1962
10	6	2	1962	1 June, 1962
11	6	30	1962	29 June, 1962
12	6	31	1962	Invalid date
13	6	32	1962	Invalid date
14	0	15	1962	Invalid date
15	1	15	1962	14 January, 1962
16	2	15	1962	14 February, 1962
17	11	15	1962	14 November, 1962
18	12	15	1962	14 December, 1962
19	13	15	1962	Invalid date

Software Testing

Example

Consider the program for the determination of nature of roots of a quadratic equation Design the Robust test case

Software Testing

Solution

Quadratic equation will be of type:

$$ax^2+bx+c=0$$

Roots are real if $(b^2-4ac)>0$

Roots are imaginary if $(b^2-4ac)<0$

Roots are equal if $(b^2-4ac)=0$

Equation is not quadratic if $a=0$

Software Testing

Solution

Robust test cases are 6n+1. Hence, in 3 variable input cases total number of test cases are 19 as given on next slide:

Software Testing

Test case	a	b	c	Expected Output
1	-1	50	50	Invalid input`
2	0	50	50	Not quadratic equation
3	1	50	50	Real roots
4	50	50	50	Imaginary roots
5	99	50	50	Imaginary roots
6	100	50	50	Imaginary roots
7	101	50	50	Invalid input
8	50	-1	50	Invalid input
9	50	0	50	Imaginary roots
10	50	1	50	Imaginary roots
11	50	99	50	Imaginary roots
12	50	100	50	Equal roots
13	50	101	50	Invalid input
14	50	50	-1	Invalid input
15	50	50	0	Real roots
16	50	50	1	Real roots
17	50	50	99	Imaginary roots
18	50	50	100	Imaginary roots
19	50	50	101	Invalid input

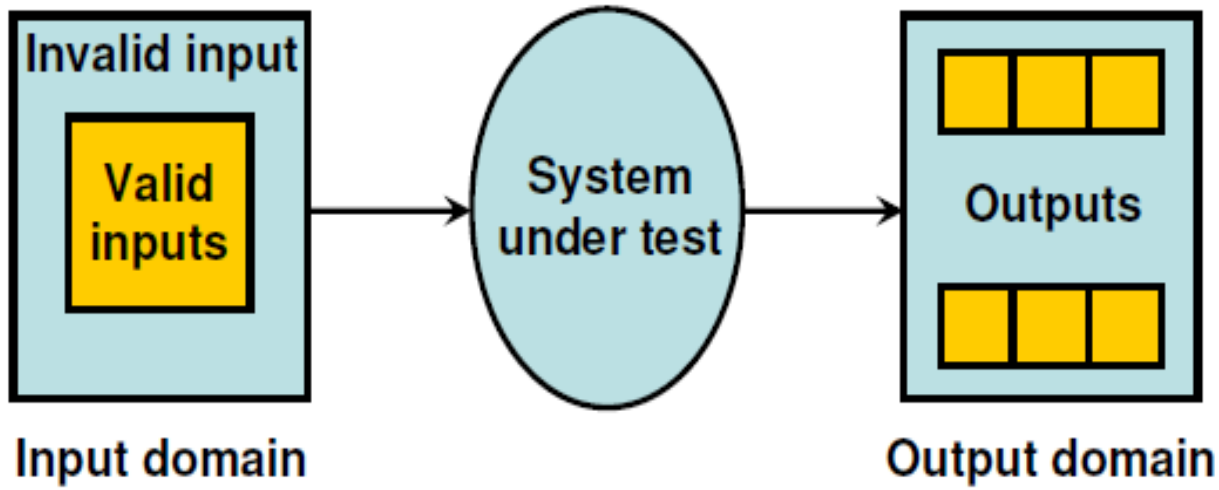
Equivalence Class Testing

In this method, input domain of a program is partitioned into a finite number of equivalence classes such that one can reasonably assume, but not be absolutely sure, that the test of a representative value of each class is equivalent to a test of any other value.

Two steps are required to implementing this method:

1. The equivalence classes are identified by taking each input condition and partitioning it into valid and invalid classes. For example, if an input condition specifies a range of values from 1 to 999, we identify one valid equivalence class [1<item<999]; and two invalid equivalence classes [item<1] and [item>999].

2. Generate the test cases using the equivalence classes identified in the previous step. This is performed by writing test cases covering all the valid equivalence classes. Then a test case is written for each invalid equivalence class so that no test contains more than one invalid class. This is to ensure that no two invalid classes mask each other.



Most of the time, equivalence class testing defines classes of the input domain. However, equivalence classes should also be defined for output domain. Hence, we should design equivalence classes based on input and output domain.

Software Testing

Consider the program for determining the previous date in a calendar
Identify the equivalence class test cases for output & input domains.

$1 \leq month \leq 12$

$1 \leq day \leq 31$

$1900 \leq year \leq 2025$

The possible outputs would be Previous date or invalid input date.

Solution

Output domain equivalence class are:

O_1={<D,M,Y>: Previous date if all are valid inputs}

O_1={<D,M,Y>: Invalid date if any input makes the date invalid}

Test case	*M*	*D*	*Y*	*Expected output*
1	6	15	1962	14 June, 1962
2	6	31	1962	Invalid date

We may have another set of test cases which are based on input domain.

I_1={month: $1 \leq m \leq 12$}

I_2={month: $m < 1$}

I_3={month: $m > 12$}

I_4={day: $1 \leq D \leq 31$}

I_5={day: $D < 1$}

I_6={day: $D > 31$}

I_7={year: $1900 \leq Y \leq 2025$}

I_8={year: $Y < 1900$}

I_9={year: $Y > 2025$}

Inputs domain test cases are :

Test Case	*M*	*D*	*Y*	*Expected output*
1	6	15	1962	14 June, 1962
2	-1	15	1962	Invalid input
3	13	15	1962	invalid input
4	6	15	1962	14 June, 1962
5	6	-1	1962	invalid input
6	6	32	1962	invalid input
7	6	15	1962	14 June, 1962
8	6	15	1899	invalid input (Value out of range)
9	6	15	2026	invalid input (Value out of range)

Consider a simple program to classify a triangle. Its inputs is a triple of positive integers (say x, y, z) and the date type for input parameters ensures that these will be integers greater than 0 and less than or equal to 100. The program output may be one of the following words:

[Scalene; Isosceles; Equilateral; Not a triangle]

Identify the equivalence class test cases for output and input domain.

Solution

Output domain equivalence classes are:

O_1={<*x*,*y*,*z*>: Equilateral triangle with sides *x*,*y*,*z*}
O_1={<*x*,*y*,*z*>: Isosceles triangle with sides *x*,*y*,*z*}
O_1={<*x*,*y*,*z*>: Scalene triangle with sides *x*,*y*,*z*}
O_1={<*x*,*y*,*z*>: Not a triangle with sides *x*,*y*,*z*}

The test cases are:

Test case	*x*	*y*	*z*	*Expected Output*
1	50	50	50	Equilateral
2	50	50	99	Isosceles
3	100	99	50	Scalene
4	50	100	50	Not a triangle

Input domain based classes are:

$I_1 = \{x : x < 1\}$

$I_2 = \{x : x > 100\}$

$I_3 = \{x : 1 \leq x \leq 100\}$

$I_4 = \{y : y < 1\}$

$I_5 = \{y : y > 100\}$

$I_6 = \{y : 1 \leq y \leq 100\}$

$I_7 = \{z : z < 1\}$

$I_8 = \{z : z > 100\}$

$I_9 = \{z : 1 \leq z \leq 100\}$

Some inputs domain test cases can be obtained using the relationship amongst x, y and z.

$I_{10}=\{< x,y,z >: x = y = z\}$

$I_{11}=\{< x,y,z >: x = y, x \neq z\}$

$I_{12}=\{< x,y,z >: x = z, x \neq y\}$

$I_{13}=\{< x,y,z >: y = z, x \neq y\}$

$I_{14}=\{< x,y,z >: x \neq y, x \neq z, y \neq z\}$

$I_{15}=\{< x,y,z >: x = y + z\}$

$I_{16}=\{< x,y,z >: x > y + z\}$

$I_{17}=\{< x,y,z >: y = x + z\}$

$I_{18}=\{< x,y,z >: y > x + z\}$

$I_{19}=\{< x,y,z >: z = x + y\}$

$I_{20}=\{< x,y,z >: z > x + y\}$

Test cases derived from input domain are:

Test case	x	y	z	Expected Output
1	0	50	50	Invalid input
2	101	50	50	Invalid input
3	50	50	50	Equilateral
4	50	0	50	Invalid input
5	50	101	50	Invalid input
6	50	50	50	Equilateral
7	50	50	0	Invalid input
8	50	50	101	Invalid input
9	50	50	50	Equilateral
10	60	60	60	Equilateral
11	50	50	60	Isosceles
12	50	60	50	Isosceles
13	60	50	50	Isosceles

(Contd.)...

Test case	*x*	*y*	*z*	*Expected Output*
14	100	99	50	Scalene
15	100	50	50	Not a triangle
16	100	50	25	Not a triangle
17	50	100	50	Not a triangle
18	50	100	25	Not a triangle
19	50	50	100	Not a triangle
20	25	50	100	Not a triangle

Software Testing

Problem 1

Consider a program for the determination of the nature of roots of a quadratic equation. Its input is a triple of positive integers (say a,b,c) and values may be from interval [0,100]. The program output may have one of the following words.

[Not a quadratic equation; Real roots; Imaginary roots; Equal roots]

Design Equivalence test case

Solution

Solution

Output domain equivalence class test cases can be identified as follows:

O_1={<a,b,c>:Not a quadratic equation if a = 0}

O_1={<a,b,c>:Real roots if $(b^2-4ac)>0$}

O_1={<a,b,c>:Imaginary roots if $(b^2-4ac)<0$}

O_1={<a,b,c>:Equal roots if $(b^2-4ac)=0$}`

The number of test cases can be derived form above relations and shown below:

Test case	*a*	*b*	*c*	*Expected output*
1	0	50	50	Not a quadratic equation
2	1	50	50	Real roots
3	50	50	50	Imaginary roots
4	50	100	50	Equal roots

Software Testing

We may have another set of test cases based on input domain.

$I_1 = \{a: a = 0\}$

$I_2 = \{a: a < 0\}$

$I_3 = \{a: 1 \leq a \leq 100\}$

$I_4 = \{a: a > 100\}$

$I_5 = \{b: 0 \leq b \leq 100\}$

$I_6 = \{b: b < 0\}$

$I_7 = \{b: b > 100\}$

$I_8 = \{c: 0 \leq c \leq 100\}$

$I_9 = \{c: c < 0\}$

$I_{10} = \{c: c > 100\}$

Test Case	a	b	c	Expected output
1	0	50	50	Not a quadratic equation
2	-1	50	50	Invalid input
3	50	50	50	Imaginary Roots
4	101	50	50	invalid input
5	50	50	50	Imaginary Roots
6	50	-1	50	invalid input
7	50	101	50	invalid input
8	50	50	50	Imaginary Roots
9	50	50	-1	invalid input
10	50	50	101	invalid input

Here test cases 5 and 8 are redundant test cases. If we choose any value other than nominal, we may not have redundant test cases. Hence total test cases are 10+4=14 for this problem.

Worst Case Testing

Software Testing

Worst-case testing

If we reject "single fault" assumption theory of reliability and may like to see what happens when more than one variable has an extreme value. In electronic circuits analysis, this is called "worst case analysis". It is more thorough in the sense that boundary value test cases are a proper subset of worst case test cases. It requires more effort. Worst case testing for a function of n variables generate 5^n test cases as opposed to $4n+1$ test cases for boundary value analysis. Our two variables example will have $5^2=25$ test cases

Worst Case Analysis

Software Testing

Table 1: Worst cases test inputs for two variables example

Test case number	Inputs		Test case number	Inputs	
	x	*y*		*x*	*y*
1	100	100	14	200	299
2	100	101	15	200	300
3	100	200	16	299	100
4	100	299	17	299	101
5	100	300	18	299	200
6	101	100	19	299	299
7	101	101	20	299	300
8	101	200	21	300	100
9	101	299	22	300	101
10	101	300	23	300	200
11	200	100	24	300	299
12	200	101	25	300	300
13	200	200	--		

Software Testing

Example

Consider the program for the determination of nature of roots of a quadratic equation Design the Robust test case and worst test cases for this program.

Software Testing

Solution

Robust test cases are 6n+1. Hence, in 3 variable input cases total number of test cases are 19 as given on next slide:

Software Testing

Test case	a	b	c	Expected Output
1	-1	50	50	Invalid input`
2	0	50	50	Not quadratic equation
3	1	50	50	Real roots
4	50	50	50	Imaginary roots
5	99	50	50	Imaginary roots
6	100	50	50	Imaginary roots
7	101	50	50	Invalid input
8	50	-1	50	Invalid input
9	50	0	50	Imaginary roots
10	50	1	50	Imaginary roots
11	50	99	50	Imaginary roots
12	50	100	50	Equal roots
13	50	101	50	Invalid input
14	50	50	-1	Invalid input
15	50	50	0	Real roots
16	50	50	1	Real roots
17	50	50	99	Imaginary roots
18	50	50	100	Imaginary roots
19	50	50	101	Invalid input

Software Testing

In case of worst test case total test cases are 5^n. Hence, 125 test cases will be generated in worst test cases. The worst test cases are given below:

Test Case	a	b	c	Expected output
1	0	0	0	Not Quadratic
2	0	0	1	Not Quadratic
3	0	0	50	Not Quadratic
4	0	0	99	Not Quadratic
5	0	0	100	Not Quadratic
6	0	1	0	Not Quadratic
7	0	1	1	Not Quadratic
8	0	1	50	Not Quadratic
9	0	1	99	Not Quadratic
10	0	1	100	Not Quadratic
11	0	50	0	Not Quadratic
12	0	50	1	Not Quadratic
13	0	50	50	Not Quadratic
14	0	50	99	Not Quadratic

(Contd.)...

Software Testing

Test Case	A	b	c	Expected output
15	0	50	100	Not Quadratic
16	0	99	0	Not Quadratic
17	0	99	1	Not Quadratic
18	0	99	50	Not Quadratic
19	0	99	99	Not Quadratic
20	0	99	100	Not Quadratic
21	0	100	0	Not Quadratic
22	0	100	1	Not Quadratic
23	0	100	50	Not Quadratic
24	0	100	99	Not Quadratic
25	0	100	100	Not Quadratic
26	1	0	0	Equal Roots
27	1	0	1	Imaginary
28	1	0	50	Imaginary
29	1	0	99	Imaginary
30	1	0	100	Imaginary
31	1	1	0	Real Roots

(Contd.)...

Software Testing

Test Case	A	b	C	Expected output
32	1	1	1	Imaginary
33	1	1	50	Imaginary
34	1	1	99	Imaginary
35	1	1	100	Imaginary
36	1	50	0	Real Roots
37	1	50	1	Real Roots
38	1	50	50	Real Roots
39	1	50	99	Real Roots
40	1	50	100	Real Roots
41	1	99	0	Real Roots
42	1	99	1	Real Roots
43	1	99	50	Real Roots
44`	1	99	99	Real Roots
45	1	99	100	Real Roots
46	1	100	0	Real Roots
47	1	100	1	Real Roots
48	1	100	50	Real Roots

(Contd.)...

Software Testing

Test Case	A	b	C	Expected output
49	1	100	99	Real Roots
50	1	100	100	Real Roots
51	50	0	0	Equal Roots
52	50	0	1	Imaginary
53	50	0	50	Imaginary
54	50	0	99	Imaginary
55	50	0	100	Imaginary
56	50	1	0	Real Roots
57	50	1	1	Imaginary
58	50	1	50	Imaginary
59	50	1	99	Imaginary
60	50	1	100	Imaginary
61	50	50	0	Real Roots
62	50	50	1	Real Roots
63	50	50	50	Imaginary
64	50	50	99	Imaginary
65	50	50	100	Imaginary

(Contd.)...

Software Testing

Test Case	A	b	C	Expected output
66	50	99	0	Real Roots
67	50	99	1	Real Roots
68	50	99	50	Imaginary
69	50	99	99	Imaginary
70	50	99	100	Imaginary
71	50	100	0	Real Roots
72	50	100	1	Real Roots
73	50	100	50	Equal Roots
74	50	100	99	Imaginary
75	50	100	100	Imaginary
76	99	0	0	Equal Roots
77	99	0	1	Imaginary
78	99	0	50	Imaginary
79	99	0	99	Imaginary
80	99	0	100	Imaginary
81	99	1	0	Real Roots
82	99	1	1	Imaginary

(Contd.)...

Software Testing

Test Case	A	b	C	Expected output
83	99	1	50	Imaginary
84	99	1	99	Imaginary
85	99	1	100	Imaginary
86	99	50	0	Real Roots
87	99	50	1	Real Roots
88	99	50	50	Imaginary
89	99	50	99	Imaginary
90	99	50	100	Imaginary
91	99	99	0	Real Roots
92	99	99	1	Real Roots
93	99	99	50	Imaginary Roots
94	99	99	99	Imaginary
95	99	99	100	Imaginary
96	99	100	0	Real Roots
97	99	100	1	Real Roots
98	99	100	50	Imaginary
99	99	100	99	Imaginary
100	99	100	100	Imaginary

(Contd.)...

Software Testing

Test Case	A	b	C	Expected output
101	100	0	0	Equal Roots
102	100	0	1	Imaginary
103	100	0	50	Imaginary
104	100	0	99	Imaginary
105	100	0	100	Imaginary
106	100	1	0	Real Roots
107	100	1	1	Imaginary
108	100	1	50	Imaginary
109	100	1	99	Imaginary
110	100	1	100	Imaginary
111	100	50	0	Real Roots
112	100	50	1	Real Roots
113	100	50	50	Imaginary
114	100	50	99	Imaginary
115	100	50	100	Imaginary
116	100	99	0	Real Roots
117	100	99	1	Real Roots
118	100	99	50	Imaginary

(Contd.)...

Software Testing

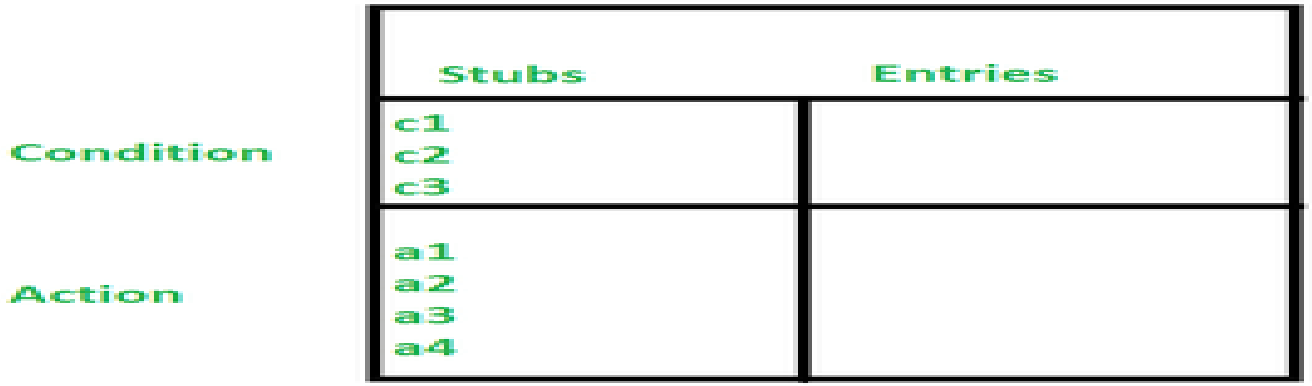
Test Case	A	b	C	Expected output
119	100	99	99	Imaginary
120	100	99	100	Imaginary
121	100	100	0	Real Roots
122	100	100	1	Real Roots
123	100	100	50	Imaginary
124	100	100	99	Imaginary
125	100	100	100	Imaginary

Decision Table Testing

- Decision table technique is one of the widely used case design techniques for black box testing. This is a systematic approach where various input combinations and their respective system behavior are captured in a tabular form.
- That's why it is also known as a cause-effect table. This technique is used to pick the test cases in a systematic manner; it saves the testing time and gives good coverage to the testing area of the software application.
- Decision table technique is appropriate for the functions that have a logical relationship between two and more than two inputs.

This technique is related to the correct combination of inputs and determines the result of various combinations of input. To design the test cases by decision table technique, we need to consider conditions as input and actions as output.

Parts of Decision Tables



1.Condition Stubs : The conditions are listed in this first upper left part of the decision table that is used to determine a particular action or set of actions.
2.Action Stubs : All the possible actions are given in the first lower left portion (i.e, below condition stub) of the decision table.
3.Condition Entries : In the condition entry, the values are inputted in the upper right portion of the decision table. In the condition entries part of the table, there are multiple rows and columns which are known as Rule.
4.Action Entries : In the action entry, every entry has some associated action or set of actions in the lower right portion of the decision table and these values are called outputs.

Types of Decision Tables :

The decision tables are categorized into two types and these are given below:
1.Limited Entry : In the limited entry decision tables, the condition entries are restricted to binary values.

2.Extended Entry : In the extended entry decision table, the condition entries have more than two values. The decision tables use multiple conditions where a condition may have many possibilities instead of only ‘true’ and ‘false’ are known as extended entry decision tables.

Example

- Most of us use an email account, and when you want to use an email account, for this you need to enter the email and its associated password.

- If both email and password are correctly matched, the user will be directed to the email account's homepage; otherwise, it will come back to the login page with an error message specified with "Incorrect Email" or "Incorrect Password."

Example

Email (condition1)	T	T	F	F
Password (condition2)	T	F	T	F
Expected Result (Action)	Account Page	Incorrect password	Incorrect email	Incorrect email

Number of possible conditions = 2^ Number of Values of the second condition
Number of possible conditions =2^2 = 4

Decision table testing

Example (ATM Decision table)

	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5
Conditions					
User Inserts Valid Card	F	T	T	T	T
User Enters Valid PIN	-	F	F	T	T
Three Invalid PINs attempted	-	F	T	F	F
Sufficient balance for the request	-	-	-	F	T
Actions					
Reject Card	Y	N	N	N	N
Prompt to Reenter PIN	N	Y	N	N	N
Eat the Card	N	N	Y	N	N
Dispense Requested Cash	N	N	N	N	Y

Software Testing

Example

Consider the triangle program Identify the test cases using the decision table

Software Testing

Conditions C_1 : x < y + z ?	F	T	T	T	T	T	T	T	T	T	T
C_2 : y < x + z ?	–	F	T	T	T	T	T	T	T	T	T
C_3 : z < x + y ?	–	–	F	T	T	T	T	T	T	T	T
C_4 : x = y ?	–	–	–	T	T	T	T	F	F	F	F
C_5 : x = z ?	–	–	–	T	T	F	F	T	T	F	F
C_6 : y = z ?	–	–	–	T	F	T	F	T	F	T	F
a_1 : Not a triangle	X	X	X								
a_2 : Scalene											X
a_3 : Isosceles							X		X	X	
a_4 : Equilateral				X							
a_5 : Impossible					X	X		X			

Solution

There are eleven functional test cases, three to fail triangle property, three impossible cases, one each to get equilateral, scalene triangle cases, and three to get on isosceles triangle. The test cases are given

Test case	*x*	*y*	*z*	*Expected Output*
1	4	1	2	Not a triangle
2	1	4	2	Not a triangle
3	1	2	4	Not a triangle
4	5	5	5	Equilateral
5	?	?	?	Impossible
6	?	?	?	Impossible
7	2	2	3	Isosceles
8	?	?	?	Impossible
9	2	3	2	Isosceles
10	3	2	2	Isosceles
11	3	4	5	Scalene

Structural Testing

Structural testing, also known as glass box testing or white box testing is an approach where the tests are derived from the knowledge of the software's structure or internal implementation.

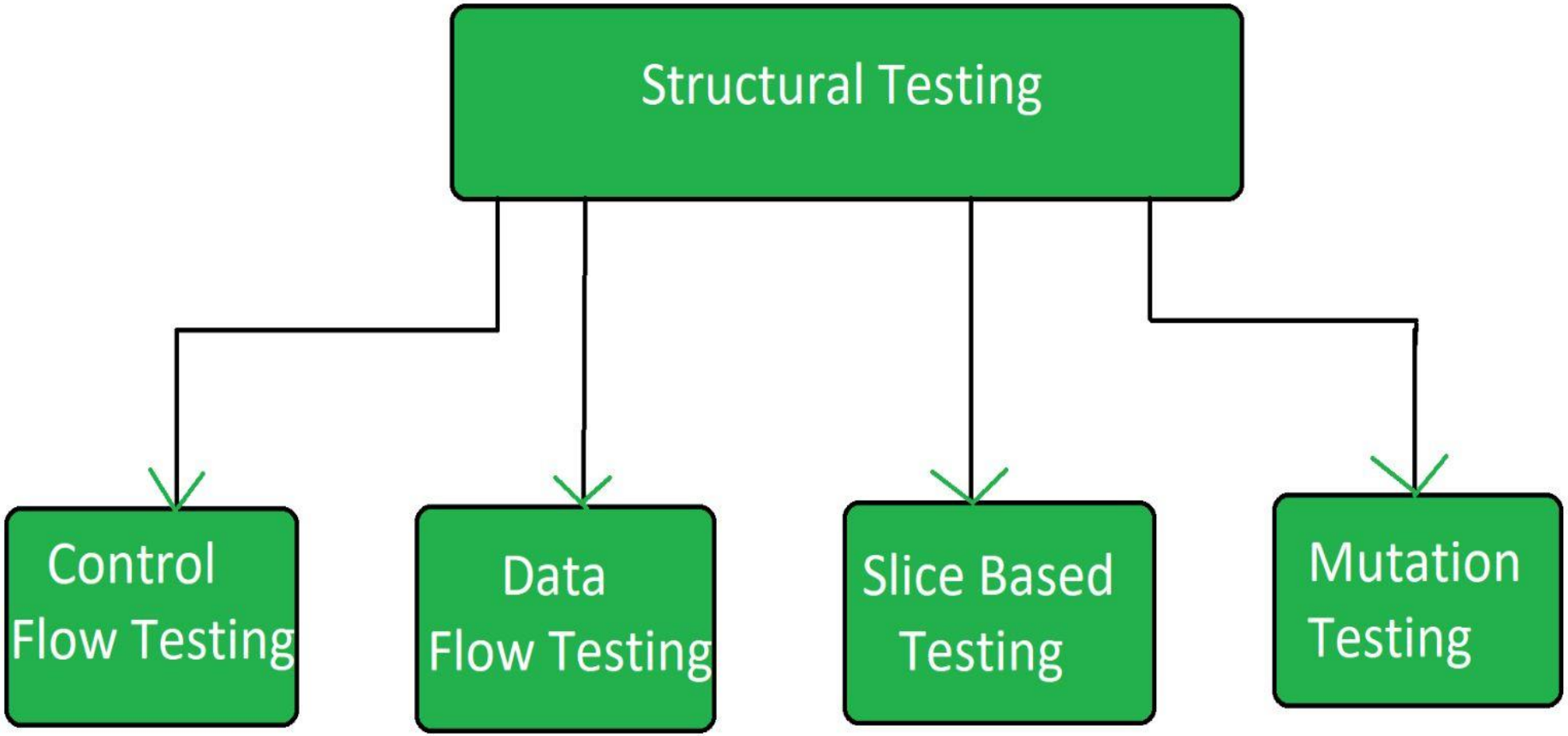
The other names of structural testing includes clear box testing, open box testing, logic driven testing or path driven testing.

Structural Testing Techniques:

Statement Coverage - This technique is aimed at exercising all programming statements with minimal tests.
Branch Coverage - This technique is running a series of tests to ensure that all branches are tested at least once.
Path Coverage - This technique corresponds to testing all possible paths which means that each statement and branch are covered.

Types of Structural Testing:

There are 4 types of Structural Testing:



Control Flow Testing:
Control flow testing is a type of structural testing that uses the programs's control flow as a model. The entire code, design and structure of the software have to be known for this type of testing.

Data Flow Testing:
It uses the control flow graph to explore the unreasonable things that can happen to data.
The detection of data flow anomalies are based on the associations between values and variables.

Slice Based Testing:
It was originally proposed by Weiser and Gallagher for the software maintenance. It is useful for software debugging, software maintenance, program understanding and quantification of functional cohesion. It divides the program into different slices and tests that slice which can majorly affect the entire software.

Mutation Testing:
Mutation Testing is a type of Software Testing that is performed to design new software tests and also evaluate the quality of already existing software tests. Mutation testing is related to modification a program in small ways. It focuses to help the tester develop effective tests or locate weaknesses in the test data used for the program.

Advantages of Structural Testing:

- It provides thorough testing of the software.
- It helps in finding out defects at an early stage.
- It helps in elimination of dead code.
- It is not time consuming as it is mostly automated.

Disadvantages of Structural Testing:

- It requires knowledge of the code to perform test.
- It requires training in the tool used for testing.
- Sometimes it is expensive.

Structural Testing Tools:

- JBehave
- Cucumber
- Junit
- Cfix

Path Testing

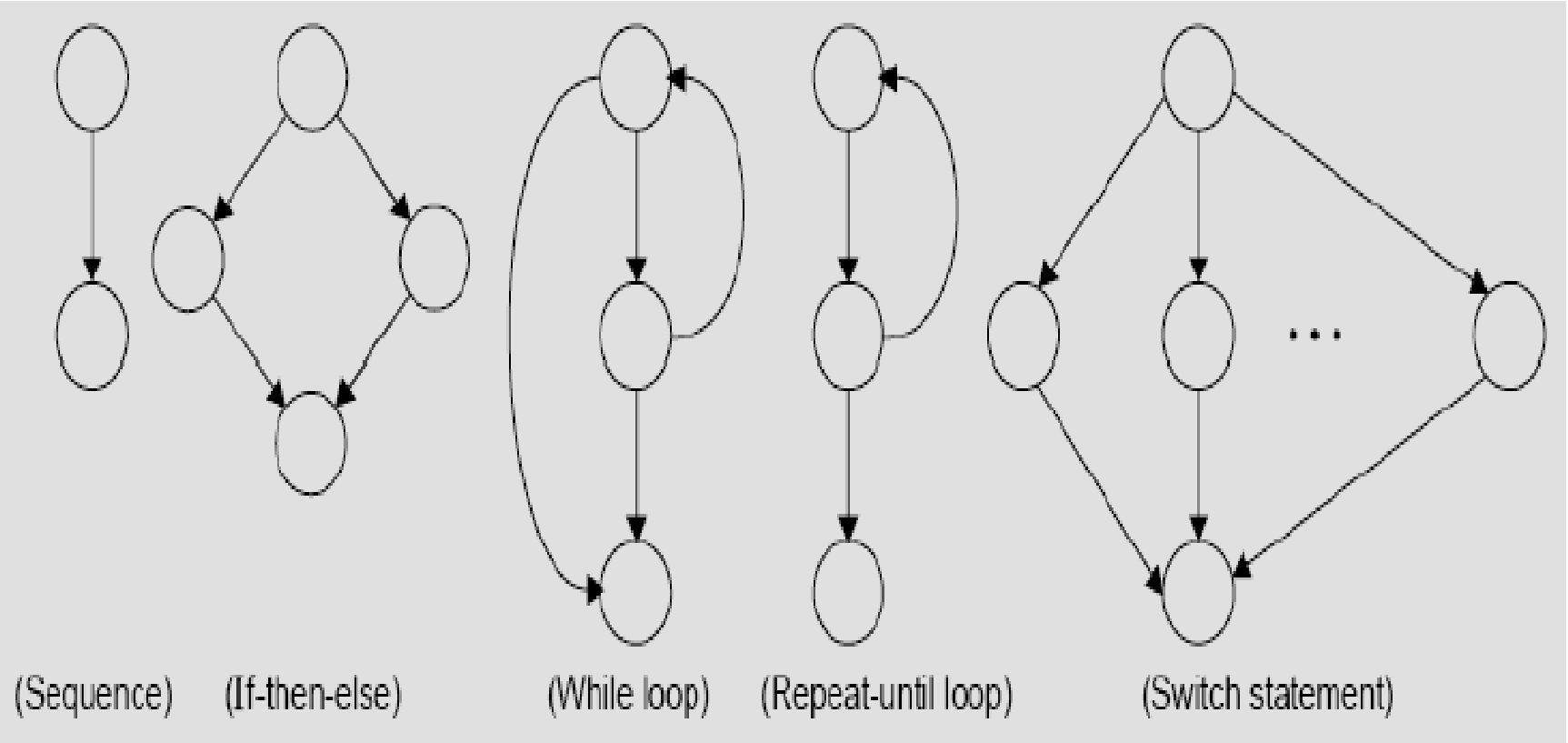
Path testing is the name given to a group of test techniques based on judiciously selecting a set of test paths through the program. If the set of paths is properly chosen, then it means that we have achieved some measure of test thoroughness.

This type of testing involves:

1. generating a set of paths that will cover every branch in the program.
2. finding a set of test cases that will execute every path in the set of program paths.

Flow Graph

The control flow of a program can be analysed using a graphical representation known as flow graph. The flow graph is a directed graph in which nodes are either entire statements or fragments of a statement, and edges represents flow of control.



Software Testing

Example

Consider the problem for the determination of the nature of roots of a quadratic equation. Its input a triple of positive integers (say a,b,c) and value may be from interval [0,100].

The output may have one of the following words:

[Not a quadratic equation; real roots; Imaginary roots; Equal roots]

Draw the flow graph and DD path graph. Also find independent paths from the DD Path graph.

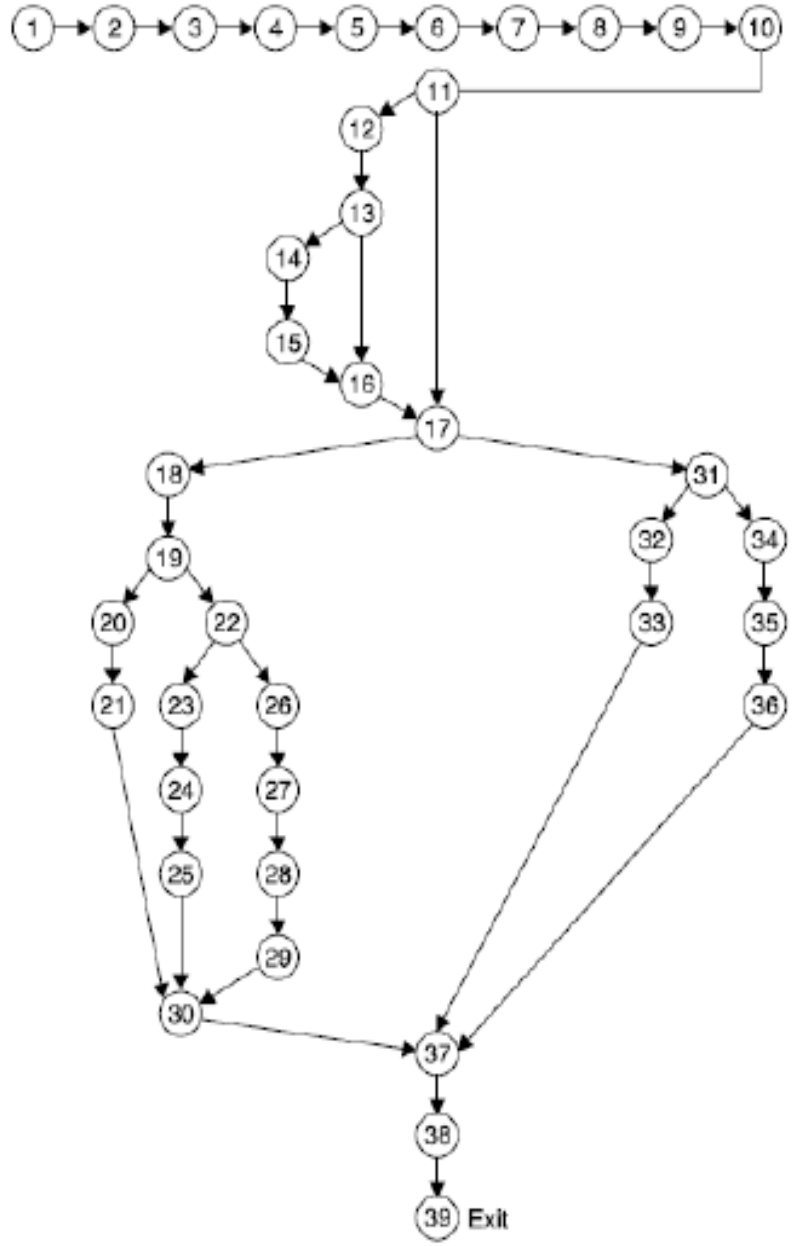
Software Testing

```
#include <conio.h>
#include <math.h>
1       int main()
2     {
3       int a,b,c,validInput=0,d;
4       double D;
5       printf("Enter the 'a' value: ");
6       scanf("%d",&a);
7       printf("Enter the 'b' value: ");
8       scanf("%d",&b);
9       printf("Enter the 'c' value: ");
10      scanf("%d",&c);
11      if ((a >= 0) && (a <= 100) && (b >= 0) && (b <= 100) && (c >= 0)
          && (c <= 100)) {
12        validInput = 1;
13        if (a == 0) {
14          validInput = -1;
15        }
16      }
17      if (validInput==1) {
18        d = b*b - 4*a*c;
19        if (d == 0) {
20          printf("The roots are equal and are r1 = r2 = %f\n",
                  -b/(2*(float) a));
```

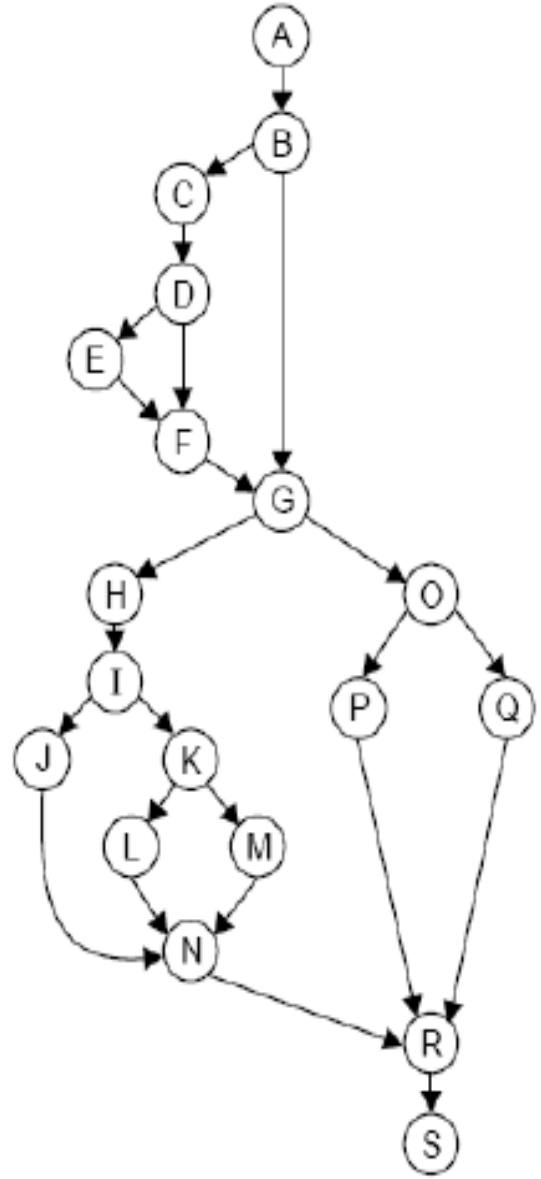
Software Testing

```
21        }
22        else if ( d > 0 ) {
23          D=sqrt(d);
24          printf("The roots are real and are r1 = %f and r2 = %f\n",
                    (-b-D)/(2* a), (-b+D)/(2* a));
25        }
26        else {
27          D=sqrt(-d)/(2*a);
28          printf("The roots are imaginary and are r1 = (%f,%f) and
                    r2 = (%f,%f)\n", -b/(2.0*a),D,-b/(2.0*a),-D);
29        }
30      }
31      else if (validInput == -1) {
32        printf("The vlaues do not constitute a Quadratic equation.");
33      }
34      else {
35        printf("The inputs belong to invalid range.");
36      }
37      getche();
38      return 1;
39    }
```

Solution


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
Exit

Software Testing



The mapping table for DD path graph is:

Flow graph nodes	DD Path graph corresponding node	Remarks
1 to 10	A	Sequential nodes
11	B	Decision node
12	C	Intermediate node
13	D	Decision node
14,15	E	Sequential node
16	F	Two edges are combined here
17	G	Two edges are combined and decision node
18	H	Intermediate node
19	I	Decision node
20,21	J	Sequential node
22	K	Decision node
23,24,25	L	Sequential node

Software Testing

Flow graph nodes	DD Path graph corresponding node	Remarks
26,27,28,29	M	Sequential nodes
30	N	Three edges are combined
31	O	Decision node
32,33	P	Sequential node
34,35,36	Q	Sequential node
37	R	Three edges are combined here
38,39	S	Sequential nodes with exit node

Independent paths are:

(i) ABGOQRS

(ii) ABGOPRS

(iii) ABCDFGOQRS

(iv) ABCDEFGOPRS

(v) ABGHIJNRS

(vi) ABGHIKLNRS

(vi) ABGHIKMNRS

Cyclomatic Complexity

Cyclomatic complexity of a code section is the quantitative measure of the number of linearly independent paths in it. It is a software metric used to indicate the complexity of a program. It is computed using the Control Flow Graph of the program.

V(G) = E – N + 2P

where,

E = the number of edges in the control flow graph

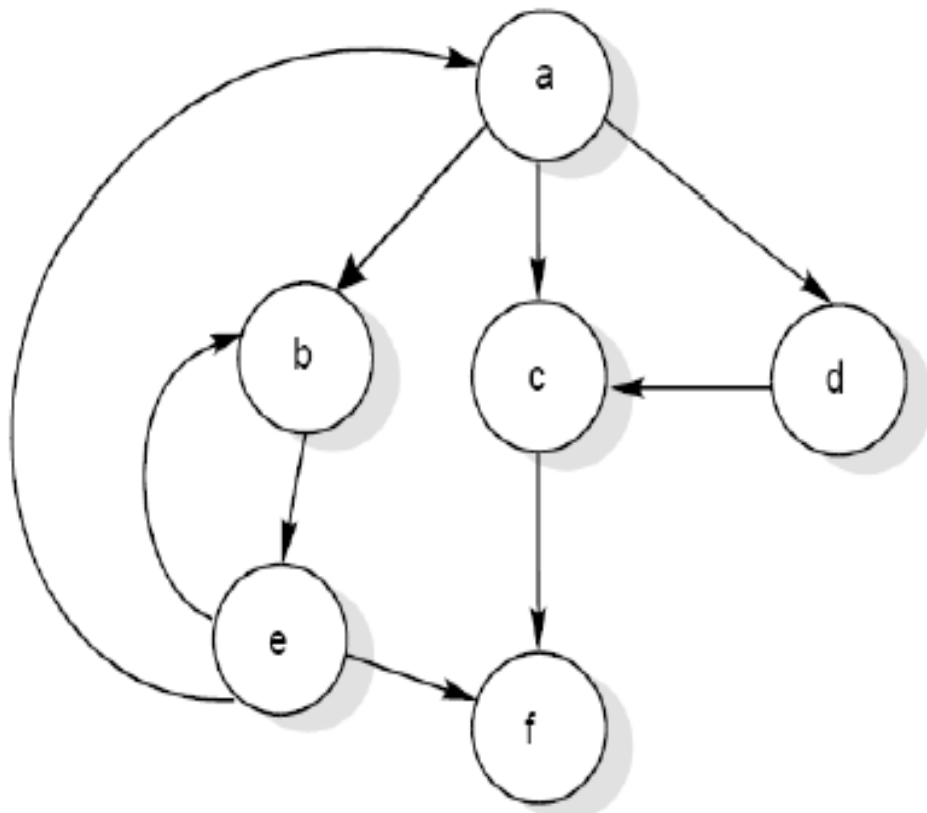
N = the number of nodes in the control flow graph

P = the number of connected components

Cyclomatic Complexity

McCabe's cyclomatic metric V(G) = $e - n + 2P$.

For example, a flow graph shown in in Fig. 21 with entry node '*a*' and exit node '*f*.



Software Testing

The value of cyclomatic complexity can be calculated as :

$$V(G) = 9 - 6 + 2 = 5$$

Here $e = 9$, $n = 6$ and $P = 1$

There will be five independent paths for the flow graph illustrated in Fig. 21.

Path 1 : *a c f*

Path 2 : *a b e f*

Path 3 : *a d c f*

Path 4 : *a b e a c f* **or** *a b e a b e f*

Path 5 : *a b e b e f*

Software Testing

Several properties of cyclomatic complexity are stated below:

1. V(G) ≥1
2. V (G) is the maximum number of independent paths in graph G.
3. Inserting & deleting functional statements to G does not affect V(G).
4. G has only one path if and only if V(G)=1.
5. Inserting a new row in G increases V(G) by unity.
6. V(G) depends only on the decision structure of G.

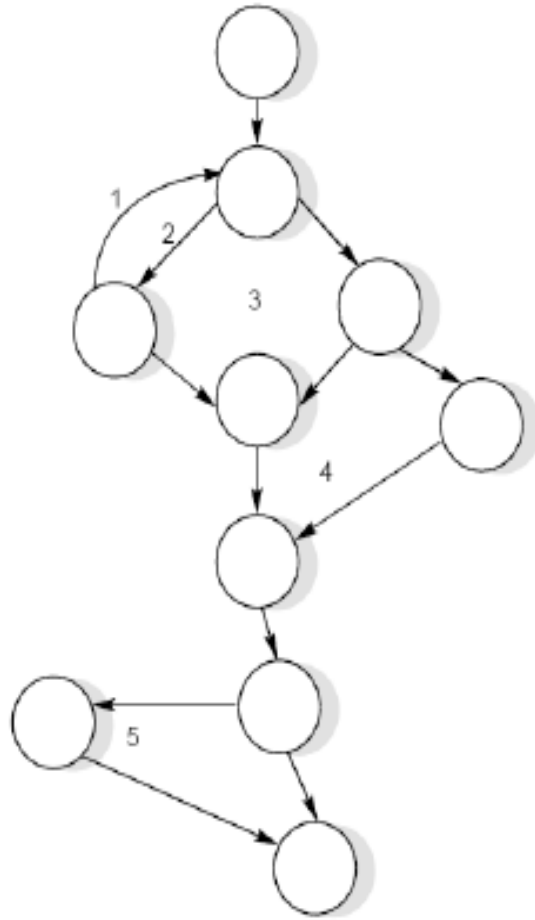
Software Testing

Two alternate methods are available for the complexity calculations.

Cyclomatic complexity is equal to the number of regions of the flow graph.

Example

Consider a flow graph given and calculate the cyclomatic complexity by all three methods.



Solution

Cyclomatic complexity can be calculated by any of the three methods.

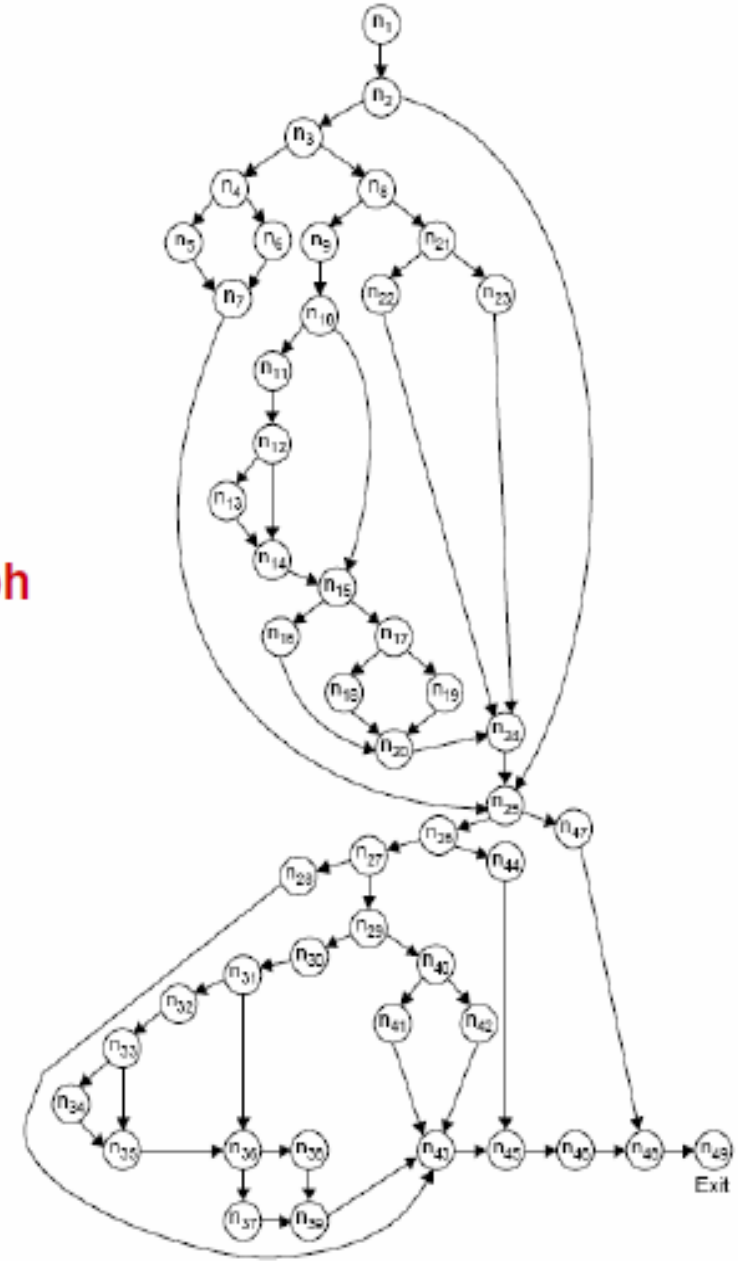
1. $V(G) = e - n + 2P$

$= 13 - 10 + 2 = 5$

Example

Consider the previous date program with DD path graph given.
Find cyclomatic complexity.

DD path graph of previous date problem



Solution

Number of edges (e) = 65

Number of nodes (n) =49

$$V(G) = e - n + 2P = 65 - 49 + 2 = 18$$

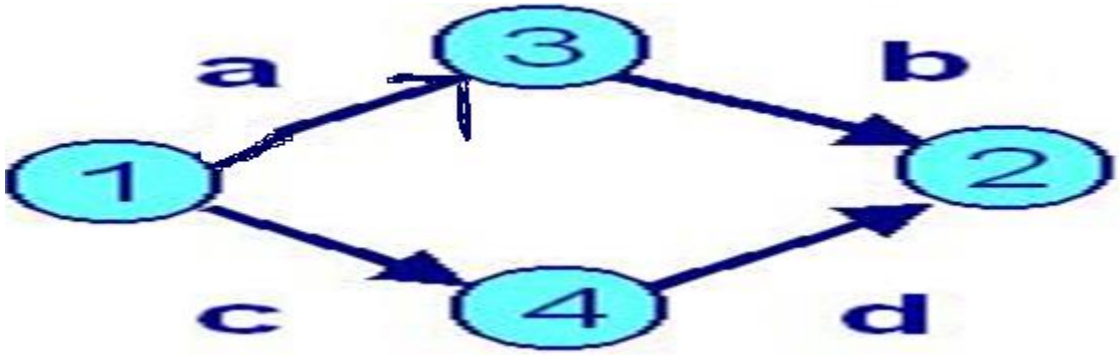
V(G) = Number of regions = 18

Graph Matrix

- In graph matrix based testing, we convert Our flow graph into a square matrix with one row and one column for every node in the graph.
- If the size of graph increases, it becomes difficult to do path tracing manually.

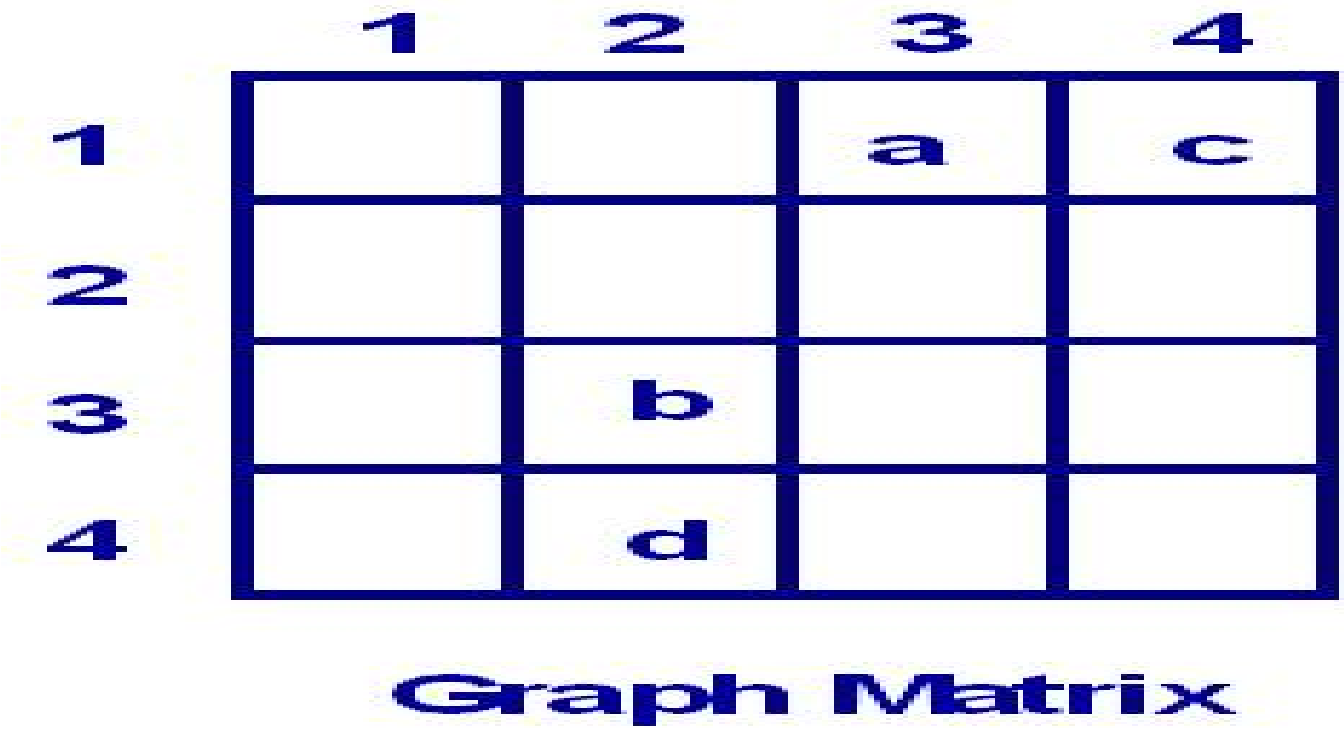
Process of constructing the Square Matrix leading to computation of Cyclomatic Complexity goes like this:

- **Step 1:** Start from the following basic Flow Graph as an example of an input

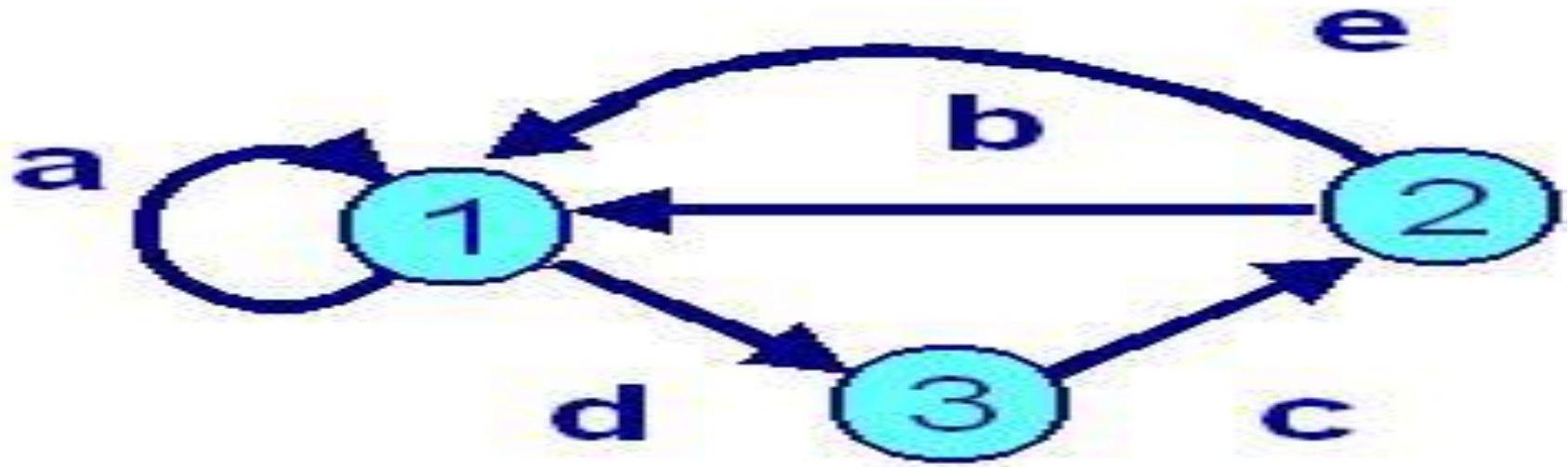


Flow Graph

Step 2: Construct its corresponding



Graph Matrix



Flow Graph

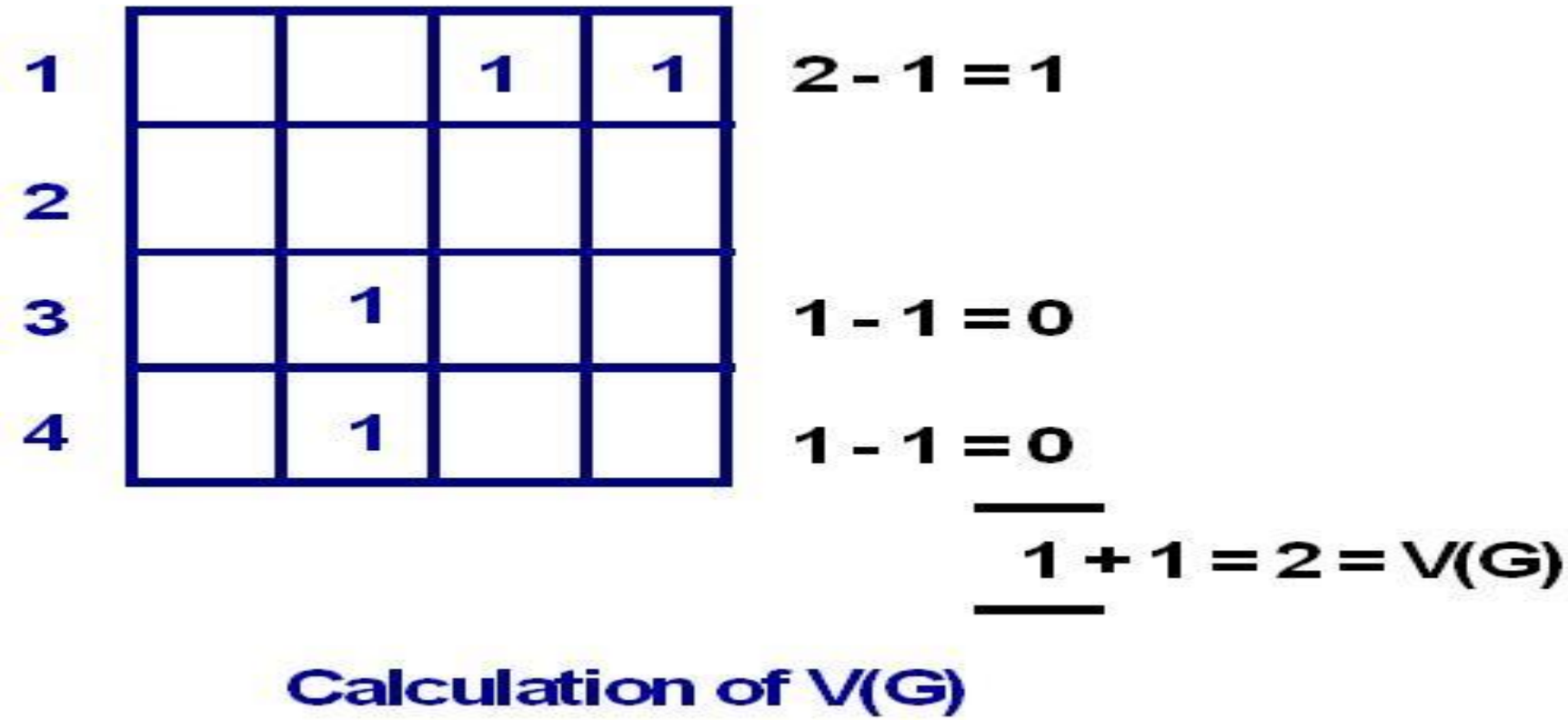
- It may be noted that if there are several links between two nodes then ” + ” sign denotes a parallel link.
- This is, however, not very useful. Hence we assign a weight to each entry of the graph matrix.
- We use ‘1’ to denote that the edge is present and ‘0’ to indicate its absence. Such a matrix is known as a connection matrix.

For above figure connection matrix can be shown as under.

	1	2	3	4
1			1	1
2				
3		1		
4		1		

Connection Matrix

Step 5: Compute it's V(G) & draw the matrix again for the same

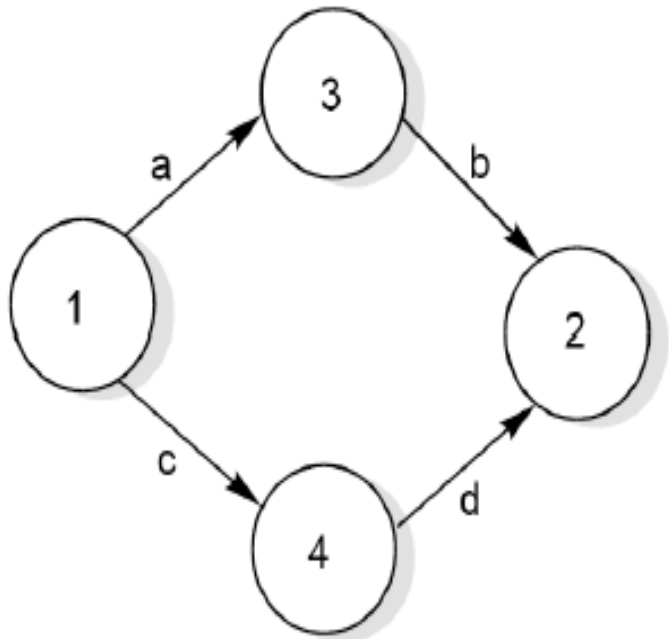


Calculation of V(G)

i.e., we sum each row of the above matrix. Then, we subtract “1” from each row. We, then, add this result or column of result and add “1” to it.
This gives us V(G).
For above matrix we get V(G) = 2.

Graph Matrices

A graph matrix is a square matrix with one row and one column for every node in the graph. The size of the matrix (i.e., the number of rows and columns) is equal to the number of nodes in the flow graph. Some examples of graphs and associated matrices are shown in fig.



Flow graph

	1	2	3	4
1			a	c
2				
3		b		
4		d		

Graph Matrix

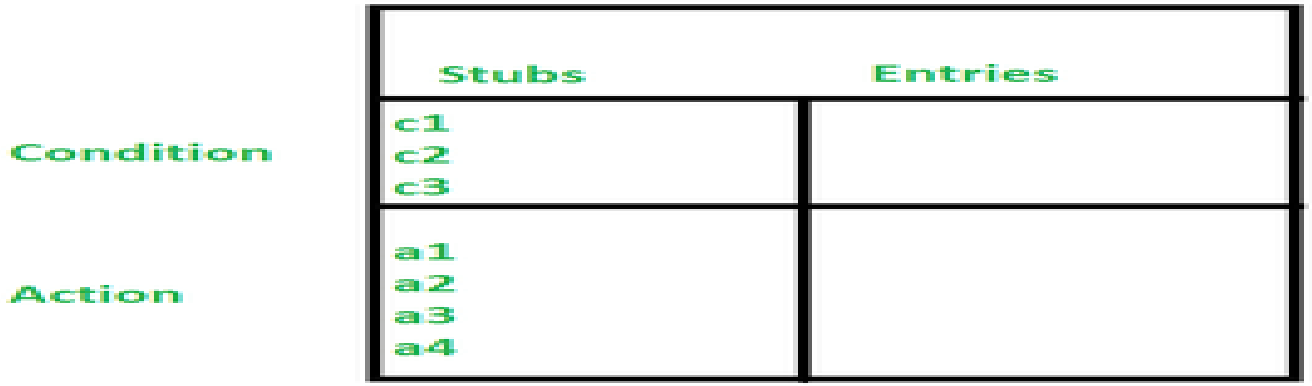
Flow graph and graph matrices

Decision Table Testing

- Decision table technique is one of the widely used case design techniques for black box testing. This is a systematic approach where various input combinations and their respective system behavior are captured in a tabular form.
- That's why it is also known as a cause-effect table. This technique is used to pick the test cases in a systematic manner; it saves the testing time and gives good coverage to the testing area of the software application.
- Decision table technique is appropriate for the functions that have a logical relationship between two and more than two inputs.

This technique is related to the correct combination of inputs and determines the result of various combinations of input. To design the test cases by decision table technique, we need to consider conditions as input and actions as output.

Parts of Decision Tables



1.Condition Stubs : The conditions are listed in this first upper left part of the decision table that is used to determine a particular action or set of actions.
2.Action Stubs : All the possible actions are given in the first lower left portion (i.e, below condition stub) of the decision table.
3.Condition Entries : In the condition entry, the values are inputted in the upper right portion of the decision table. In the condition entries part of the table, there are multiple rows and columns which are known as Rule.
4.Action Entries : In the action entry, every entry has some associated action or set of actions in the lower right portion of the decision table and these values are called outputs.

Types of Decision Tables :

The decision tables are categorized into two types and these are given below:
1.Limited Entry : In the limited entry decision tables, the condition entries are restricted to binary values.

2.Extended Entry : In the extended entry decision table, the condition entries have more than two values. The decision tables use multiple conditions where a condition may have many possibilities instead of only ‘true’ and ‘false’ are known as extended entry decision tables.

Example

- Most of us use an email account, and when you want to use an email account, for this you need to enter the email and its associated password.

- If both email and password are correctly matched, the user will be directed to the email account's homepage; otherwise, it will come back to the login page with an error message specified with "Incorrect Email" or "Incorrect Password."

Example

Email (condition1)	T	T	F	F
Password (condition2)	T	F	T	F
Expected Result (Action)	Account Page	Incorrect password	Incorrect email	Incorrect email

Decision table testing

Example (ATM Decision table)

	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5
Conditions					
User Inserts Valid Card	F	T	T	T	T
User Enters Valid PIN	-	F	F	T	T
Three Invalid PINs attempted	-	F	T	F	F
Sufficient balance for the request	-	-	-	F	T
Actions					
Reject Card	Y	N	N	N	N
Prompt to Reenter PIN	N	Y	N	N	N
Eat the Card	N	N	Y	N	N
Dispense Requested Cash	N	N	N	N	Y

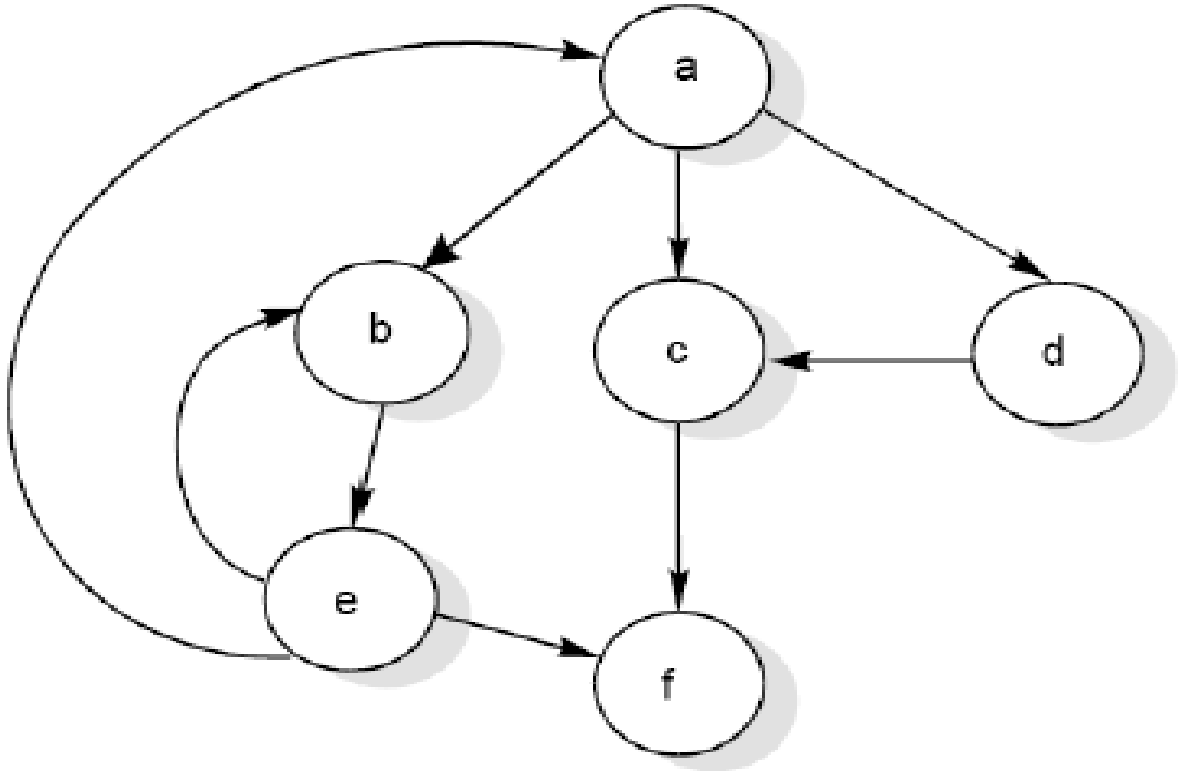
Test case design

C_1:x,y,z are sides of a triangle?	N	Y							
C_2:x = y?	--	Y				N			
C_3:x = z?	--	Y		N		Y		N	
C_4:y = z?	--	Y	N	Y	N	Y	N	Y	N
a_1: Not a triangle	X								
a_2: Scalene									X
a_3: Isosceles					X		X	X	
a_4: Equilateral		X							
a_5: Impossible			X	X		X			

Cyclomatic Complexity

McCabe's cyclomatic metric V(G) = $e - n + 2P$.

For example, a flow graph shown in in Fig. 21 with entry node '*a*' and exit node '*f*.



Software Testing

The value of cyclomatic complexity can be calculated as :

$$V(G) = 9 - 6 + 2 = 5$$

Here $e = 9$, $n = 6$ and $P = 1$

There will be five independent paths for the flow graph illustrated in Fig. 21.

Path 1 : *a c f*

Path 2 : *a b e f*

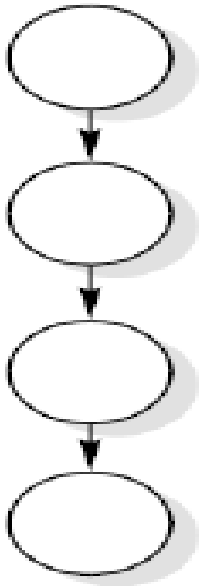
Path 3 : *a d c f*

Path 4 : *a b e a c f* **or** *a b e a b e f*

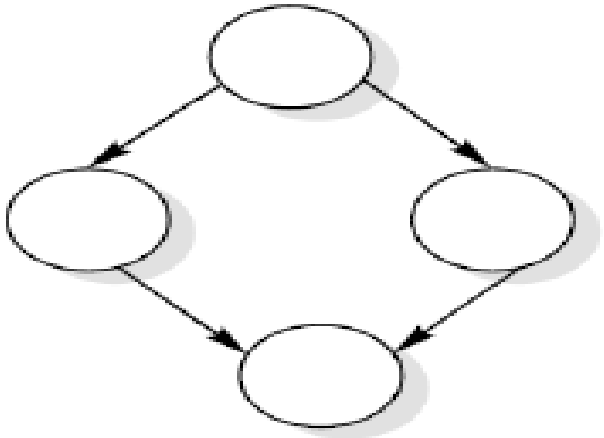
Path 5 : *a b e b e f*

Software Testing

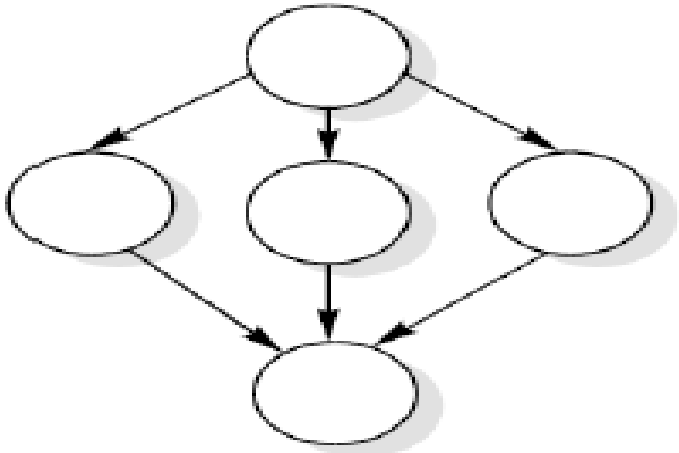
The role of P in the complexity calculation V(G)=e-n+2P is required to be understood correctly. We define a flow graph with unique entry and exit nodes, all nodes reachable from the entry, and exit reachable from all nodes. This definition would result in all flow graphs having only one connected component. One could, however, imagine a main program M and two called subroutines A and B having a flow graph shown in Fig. 22.



M:



A:



B:

Software Testing

Let us denote the total graph above with 3 connected components as

$$V(M \cup A \cup B) = e - n + 2P$$

$$= 13\text{-}13\text{+}2^{*}3$$

$$= 6$$

This method with P ≠ 1 can be used to calculate the complexity of a collection of programs, particularly a hierarchical nest of subroutines.

Software Testing

Two alternate methods are available for the complexity calculations.

1. Cyclomatic complexity V(G) of a flow graph G is equal to the number of predicate (decision) nodes plus one.

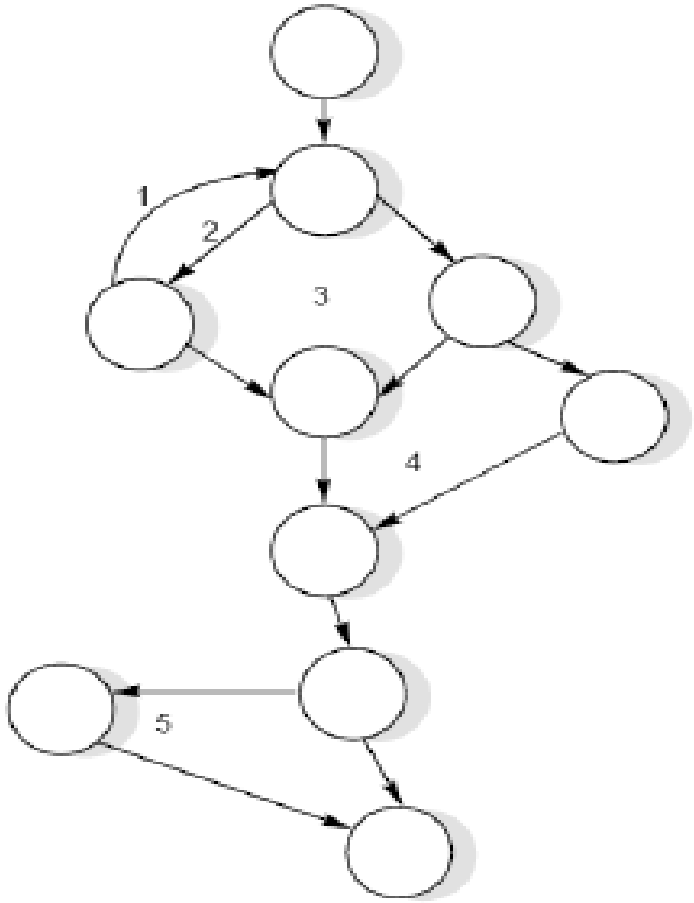
   $V(G)=\prod+1$

   Where $\prod$ is the number of predicate nodes contained in the flow graph G.

2. Cyclomatic complexity is equal to the number of regions of the flow graph.

Software Testing

Consider a flow graph given in Fig. and calculate the cyclomatic complexity by all three methods.



Solution

Cyclomatic complexity can be calculated by any of the three methods.

1. V(G) $= e - n + 2P$

   $= 13 - 10 + 2 = 5$

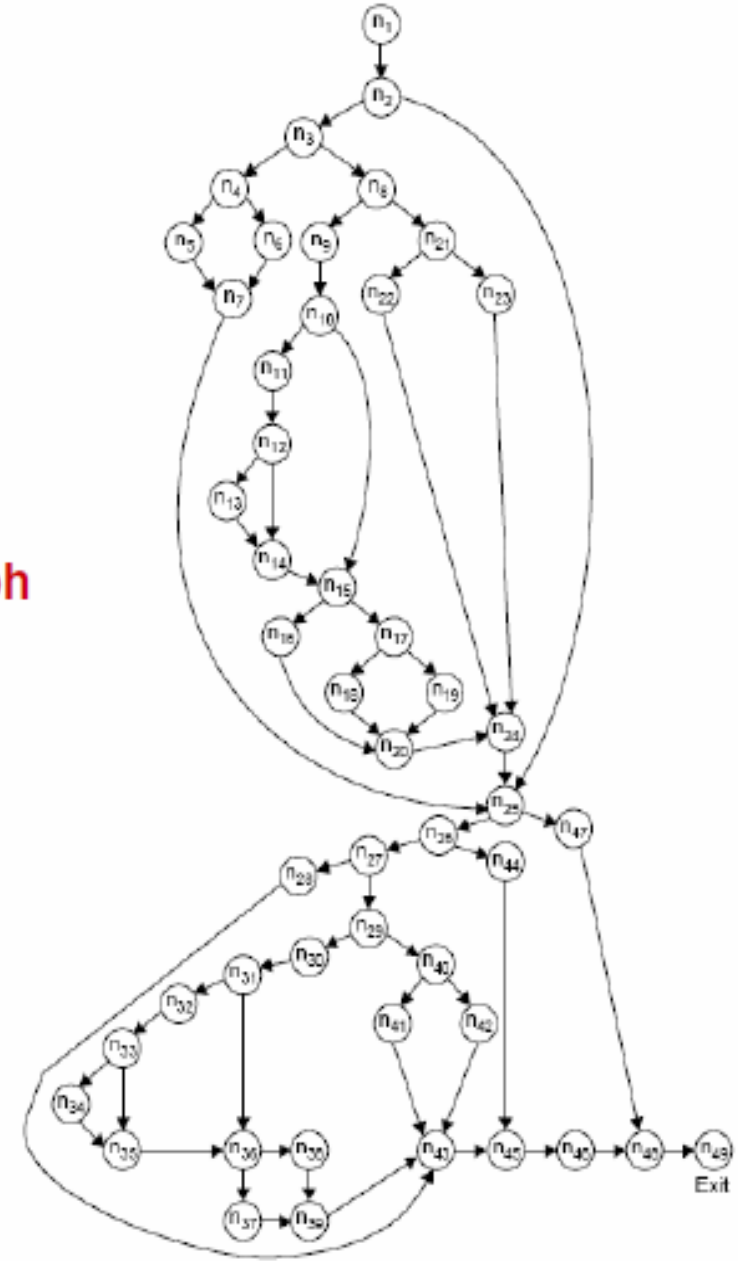
2. V(G) $= \pi + 1$

   $= 4 + 1 = 5$

3. V(G) = number of regions

   $= 5$

DD path graph of previous date problem



Solution

Number of edges (e) = 65

Number of nodes (n) =49

(i) $V(G) = e - n + 2P = 65 - 49 + 2 = 18$

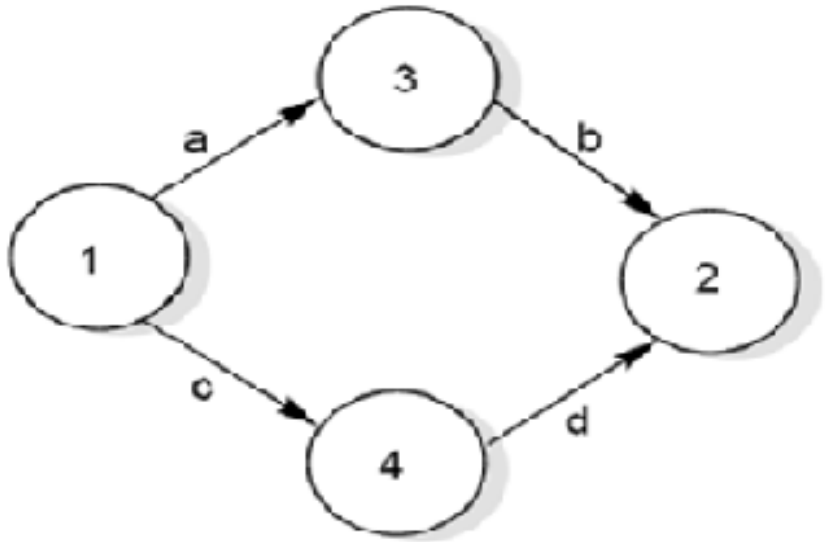
(ii) $V(G) = \pi + 1 = 17 + 1 = 18$

(iii) V(G) = Number of regions = 18

The cyclomatic complexity is 18.

Graph Matrices

A graph matrix is a square matrix with one row and one column for every node in the graph. The size of the matrix (i.e., the number of rows and columns) is equal to the number of nodes in the flow graph. Some examples of graphs and associated matrices are shown in fig. :



Flow graph

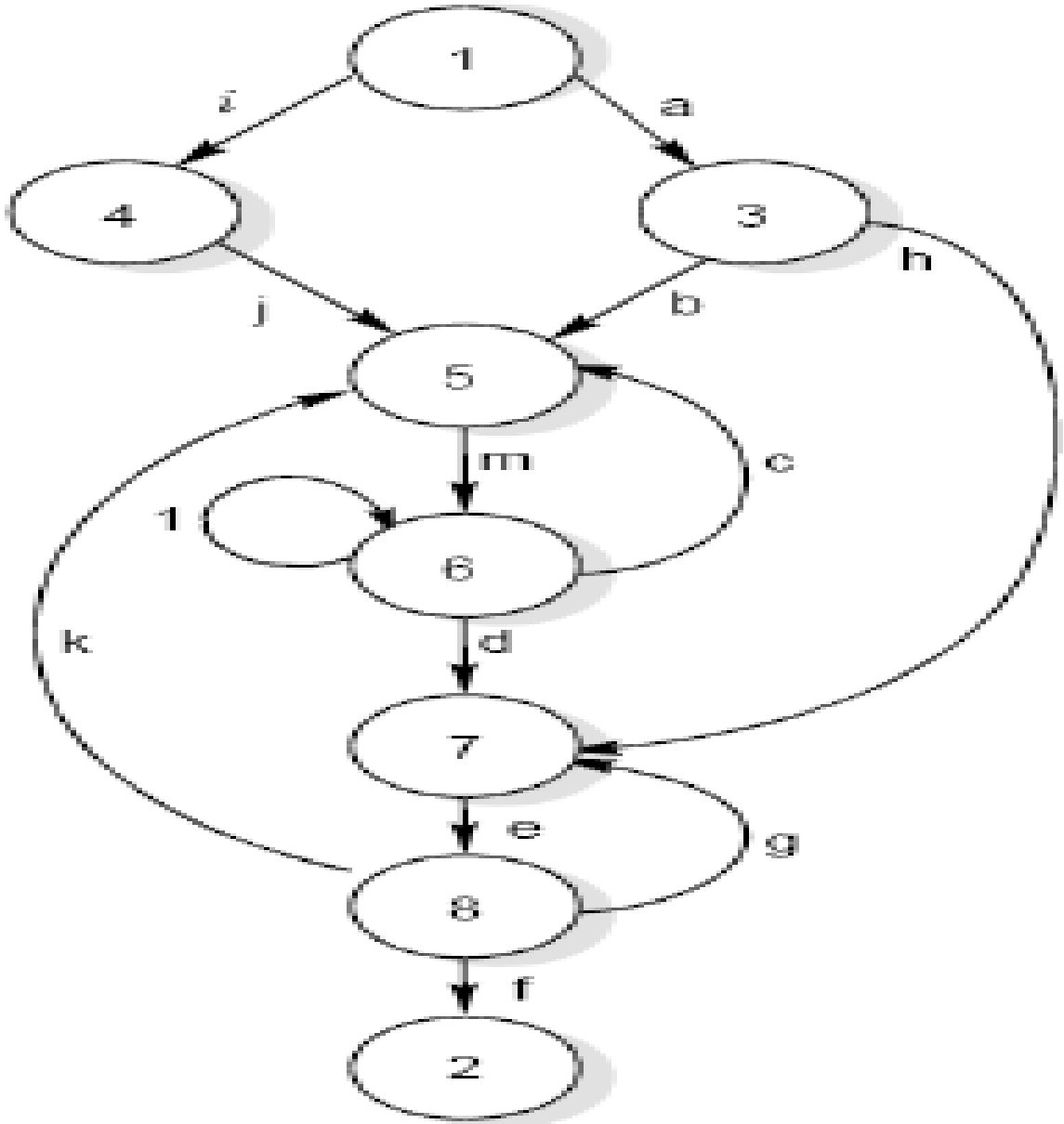
(a)

	1	2	3	4
1			a	c
2				
3		b		
4		d		

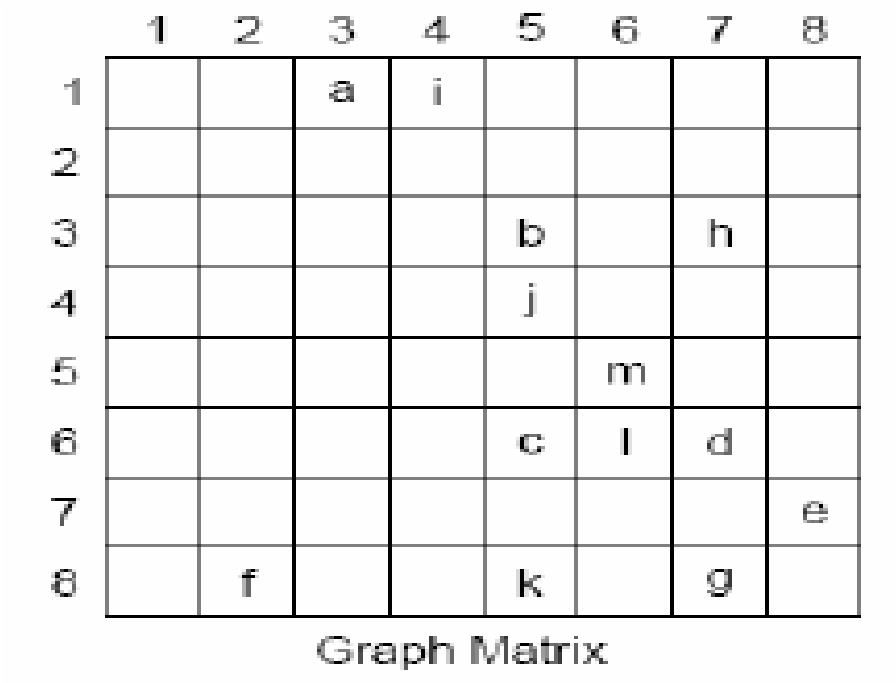
Graph Matrix

	1	2	3	4
1		ab + cd		
2				
3				
4				

$[A]^2$



Flow graph



Graph Matrix

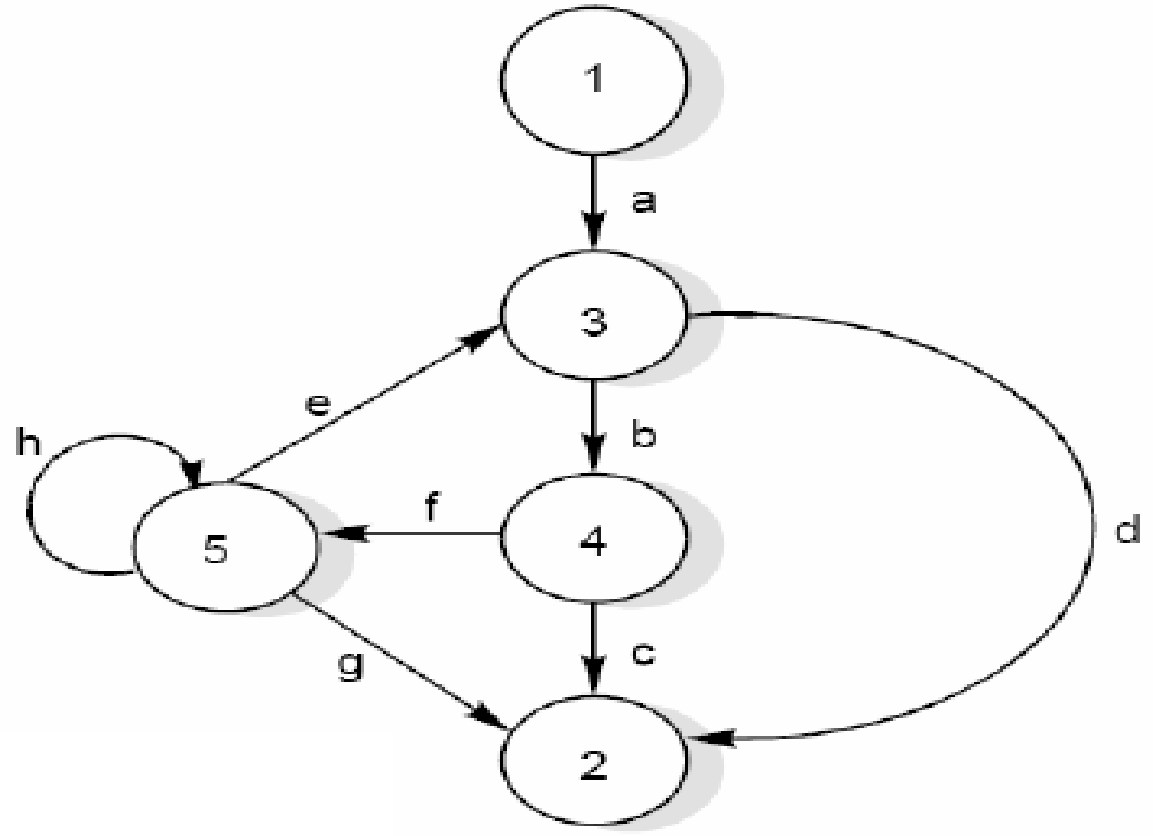
Software Testing

	1	2	3	4	5	6	7	8	Connections
1			1	1					2 – 1 = 1
2									
3					1		1		2 – 1 = 1
4					1				1 – 1 = 0
5						1			1 – 1 = 0
6					1	1	1		3 – 1 = 2
7								1	1 – 1 = 0
8		1			1		1		3 – 1 = 2
									6 + 1 = 7

Software Testing

Example

Consider the flow graph shown in the Fig. and draw the graph & connection matrices. Find out cyclomatic complexity and two / three link paths from a node to any other node.



Software Testing

Solution

The graph & connection matrices are given below :

	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		h

Graph Matrix (A)

	1	2	3	4	5	Connections
1			1			1 – 1 = 0
2						
3		1		1		2 – 1 = 1
4		1			1	2 – 1 = 1
5		1	1		1	3 – 1 = 2
						4 + 1 = 5

Connection Matrix

To find two link paths, we have to generate a square of graph matrix [A] and for three link paths, a cube of matrix [A] is required.

Software Testing

	1	2	3	4	5
1		ad		ab	
2					
3		bc			bf
4		fg	fe		fh
5		ed + hg	he	eb	h^2

$[A^2]$

	1	2	3	4	5
1		abc			afb
2					
3		bfg	bfe		bfh
4		fed + fhg	fhe	feb	fh^2
5		ebc + hed + h^2g	h^2e	heb	ebf + h^3

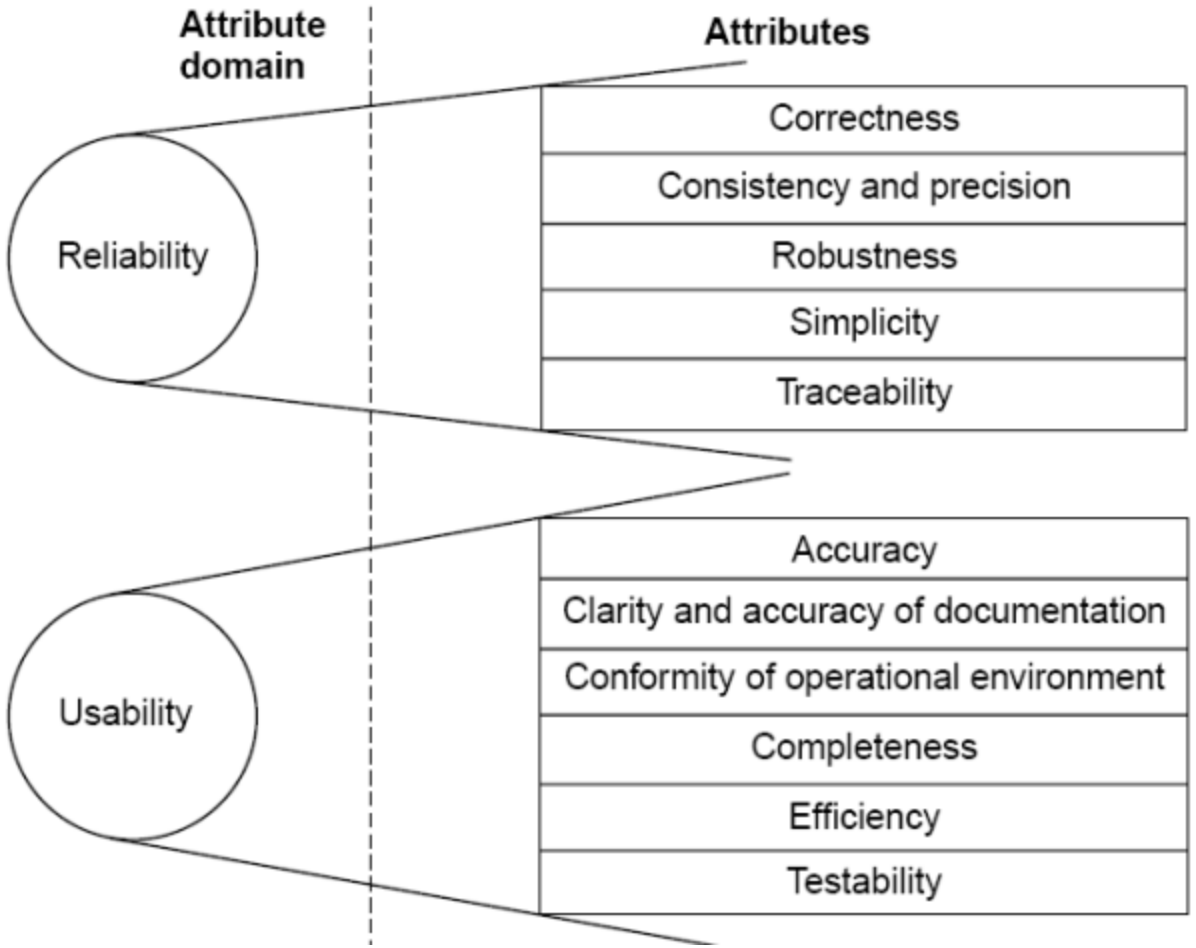
$[A^3]$

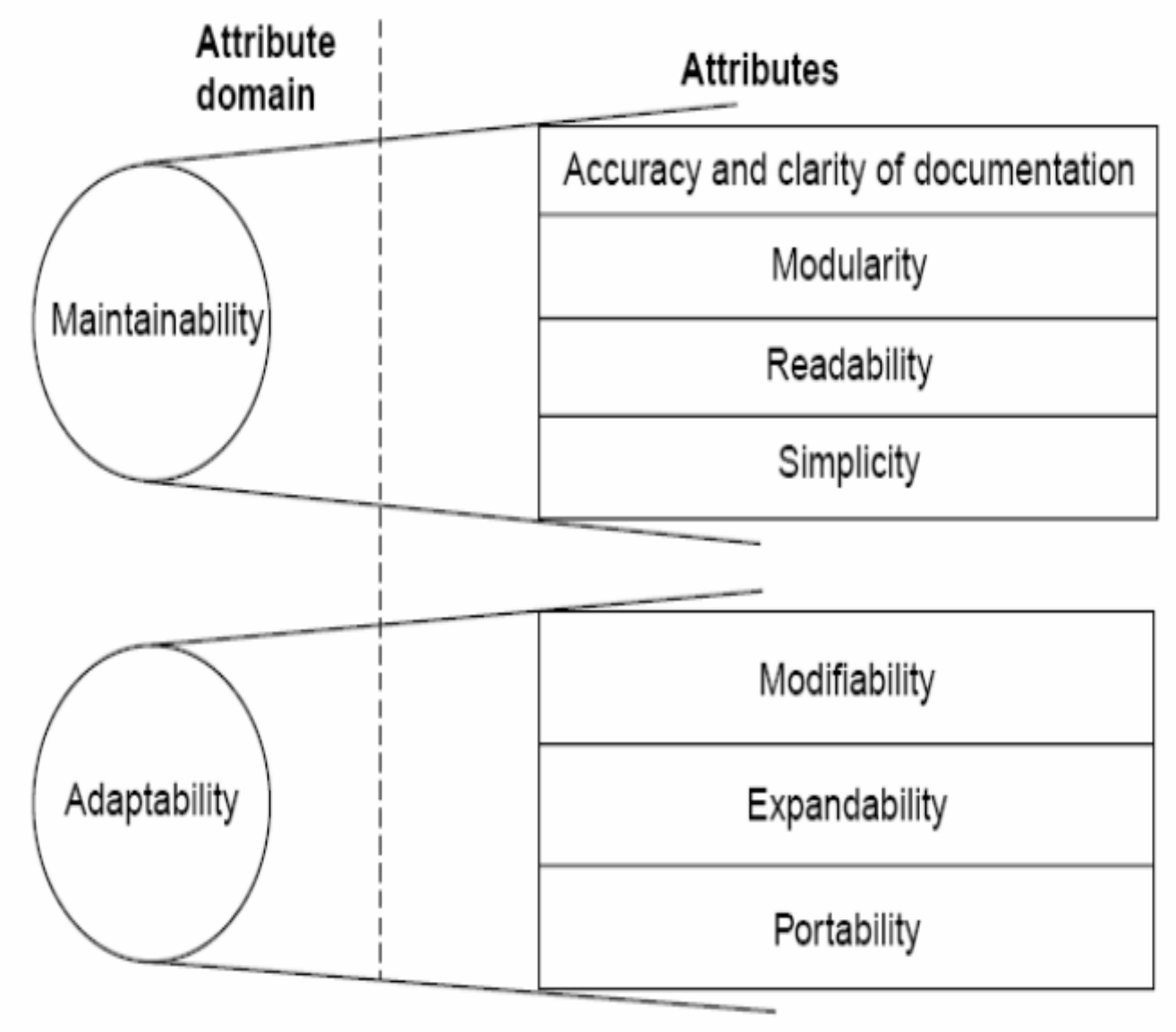
Software Quality

Different people understand different meanings of quality like:

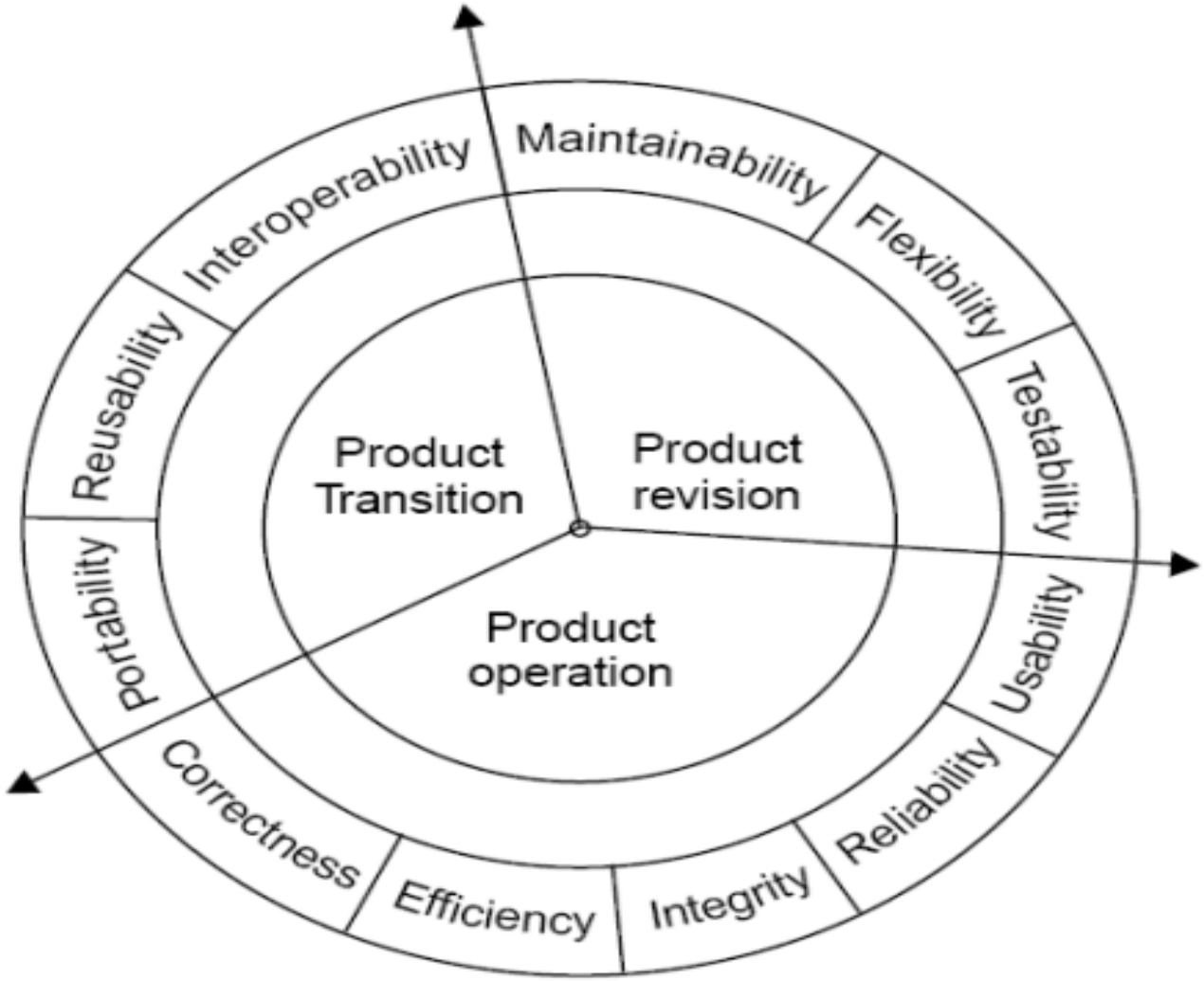
- conformance to requirements
- fitness for the purpose
- level of satisfaction

Software quality attributes

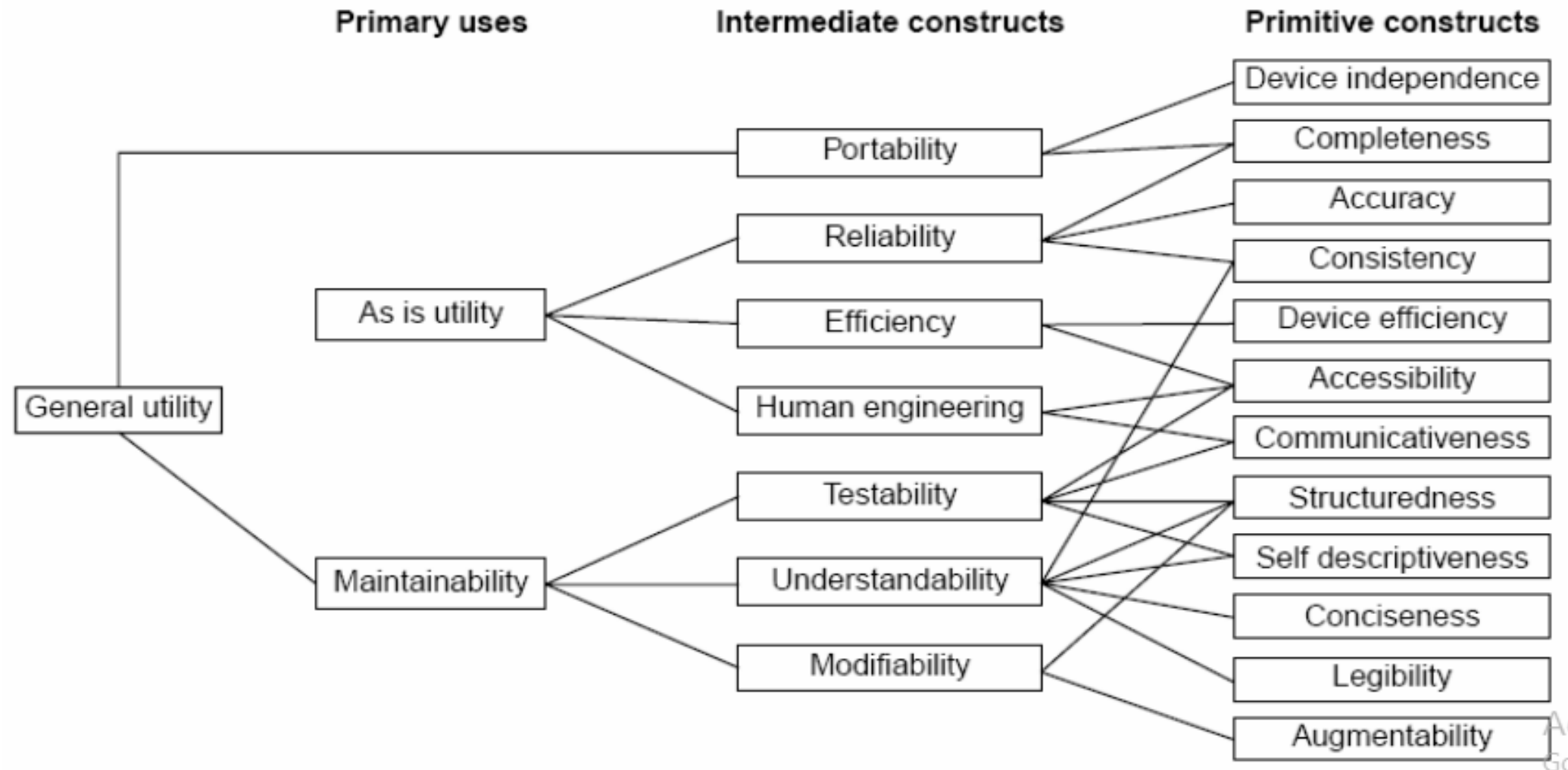



Attribute domain
Attributes
Maintainability
Accuracy and clarity of documentation
Modularity
Readability
Simplicity
Adaptability
Modifiability
Expandability
Portability

- McCall Software Quality Model



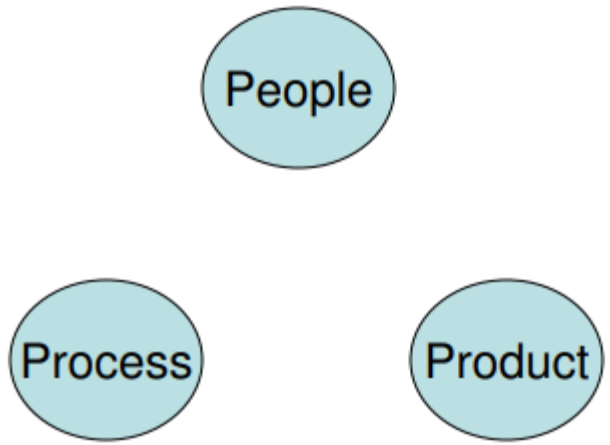
- # Boehm Software Quality Model



Software Certification

To whom should we target

- People
- Process
- Product



We have seen many certified developers (Microsoft certified, Cisco certified, JAVA certified), certified processes (like ISO or CMM) and certified products.

Types of Certification

- People
  - Industry specific
- Process
  - Industry specific
- Product
  - For the customer directly and helps to select a particular product

Certification of Processes

The most popular process certification approaches are:

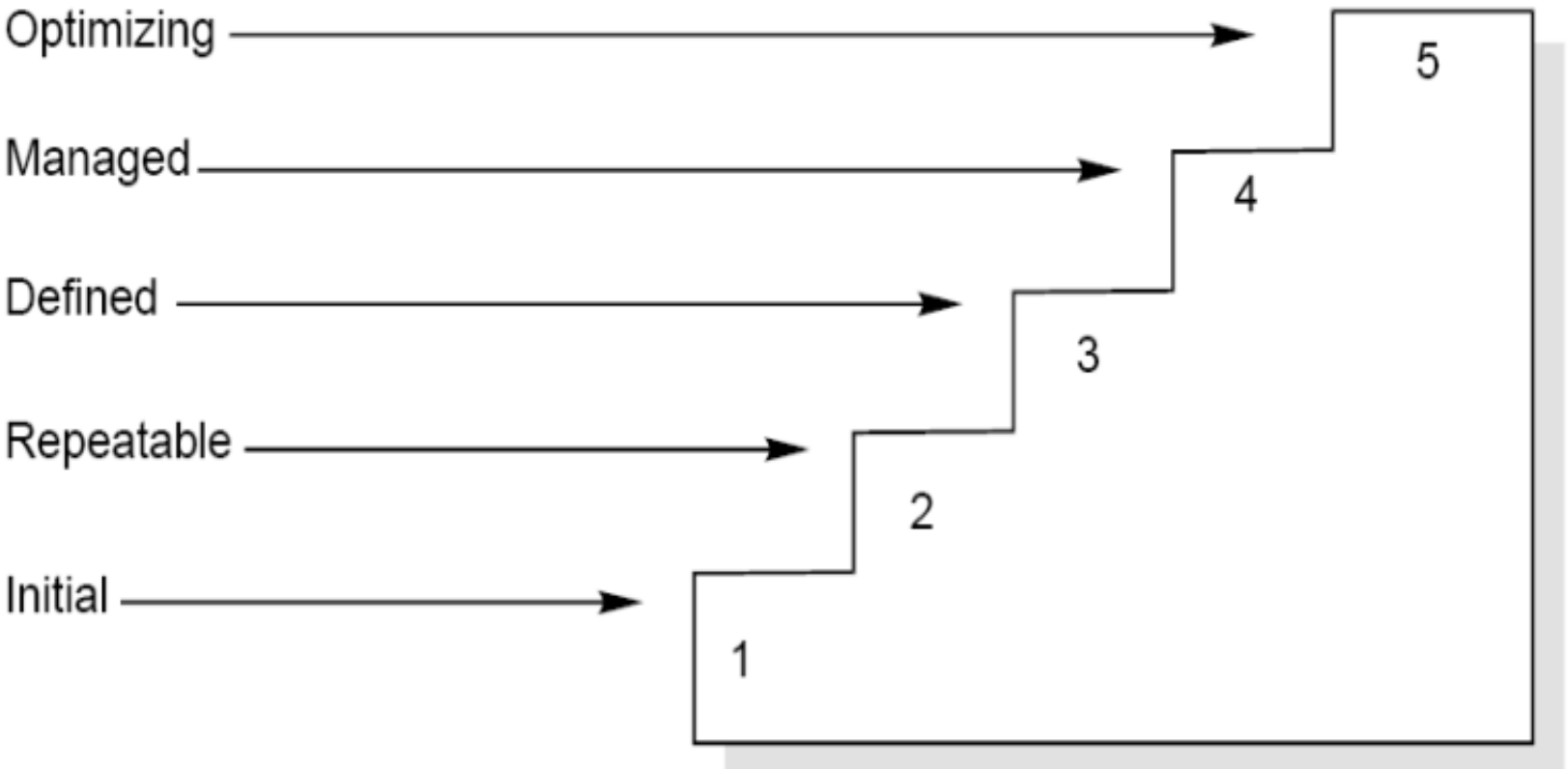
- ISO 9000
- SEI-CMM

One should always be suspicious about the quality of end product, however, certification reduces the possibility of poor quality products.

Any type of process certification helps to produce good quality and stable software product.

- Capability Maturity Model

It is a strategy for improving the software process, irrespective of the actual life cycle model used.



Maturity Levels:

- ✓ Initial (Maturity Level 1)
- ✓ Repeatable (Maturity Level 2)
- ✓ Defined (Maturity Level 3)
- ✓ Managed (Maturity Level 4)
- ✓ Optimizing (Maturity Level 5)

Maturity Level	Characterization
Initial	Adhoc Process
Repeatable	Basic Project Management
Defined	Process Definition
Managed	Process Measurement
Optimizing	Process Control

Software Maintenance

What is Software Maintenance?

Software Maintenance is a very broad activity that includes error corrections, enhancements of capabilities, deletion of obsolete capabilities, and optimization.

Categories of Maintenance

- Corrective maintenance

This refer to modifications initiated by defects in the software.

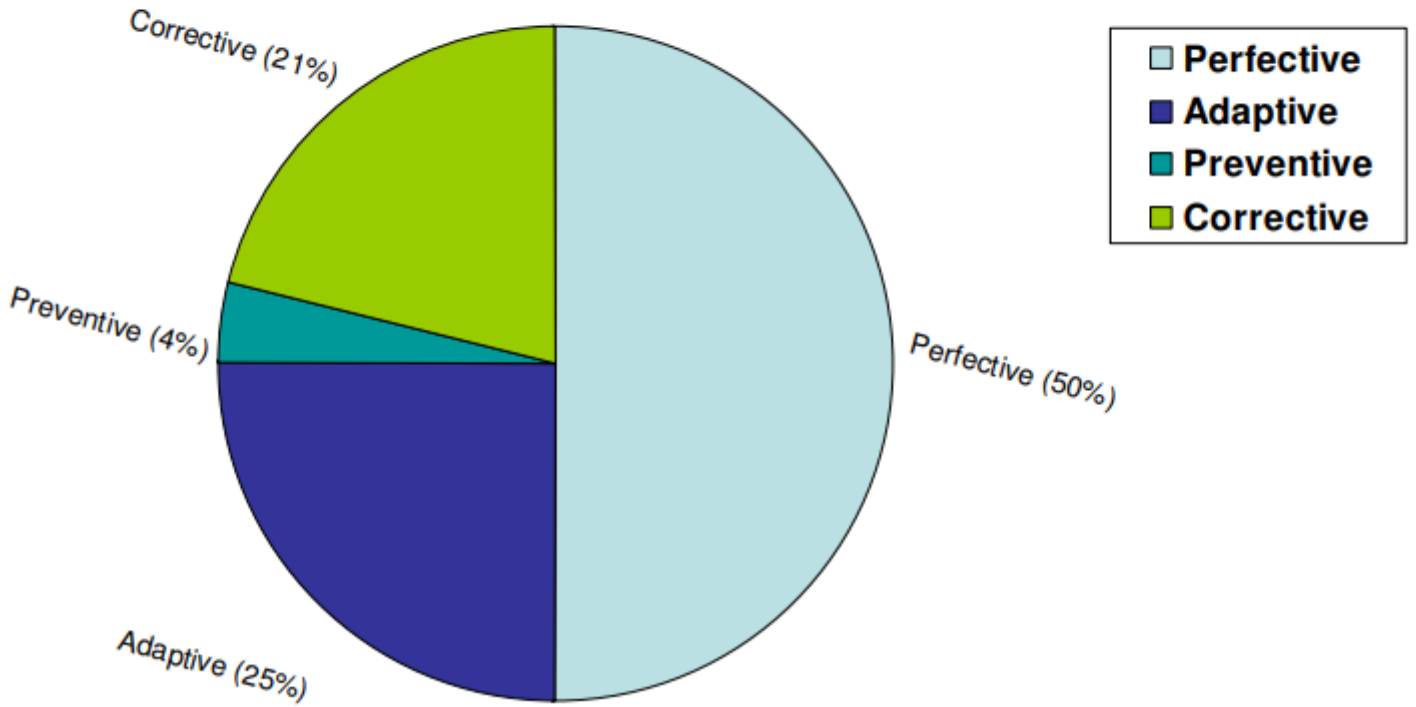
- Adaptive maintenance

It includes modifying the software to match changes in the ever changing environment.

- Perfective maintenance

It means improving processing efficiency or performance, or restructuring the software to improve changeability. This may include enhancement of existing system functionality, improvement in computational efficiency etc.

Software Maintenance



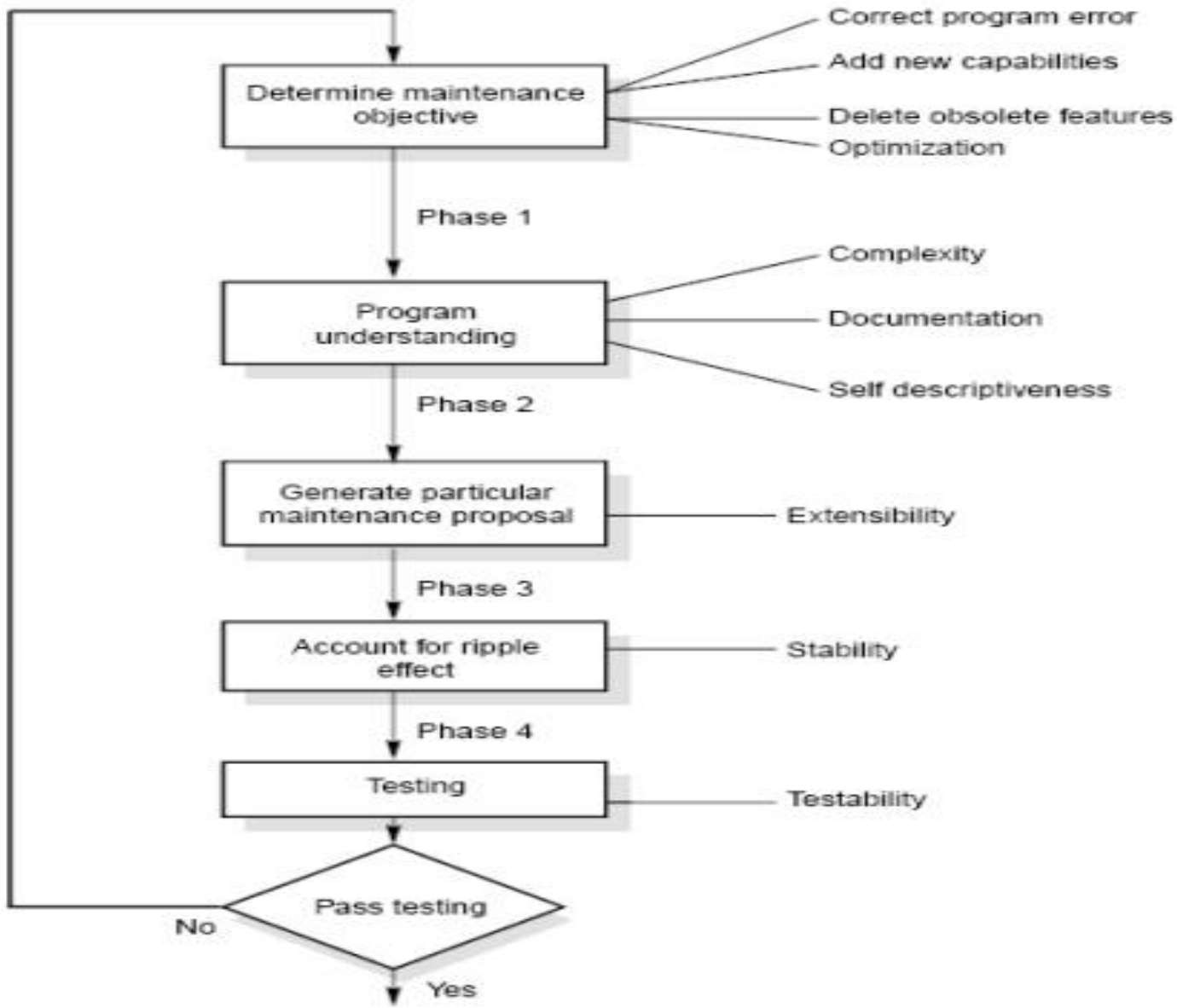
Problems During Maintenance

- Often the program is written by another person or group of persons.
- Often the program is changed by person who did not understand it clearly.
- Program listings are not structured.
- High staff turnover.
- Information gap.
- Systems are not designed for change.

Potential Solutions to Maintenance Problems

- Budget and effort reallocation
- Complete replacement of the system
- Maintenance of existing system

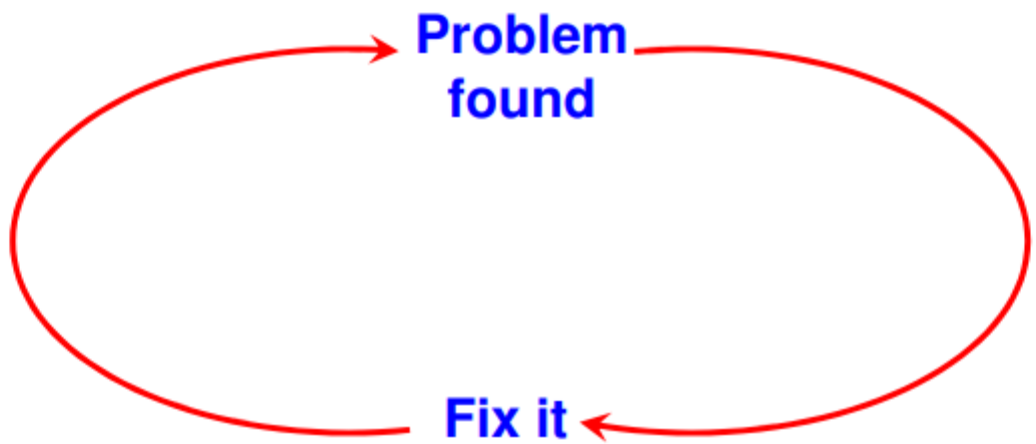
The Maintenance Process



Maintenance Models

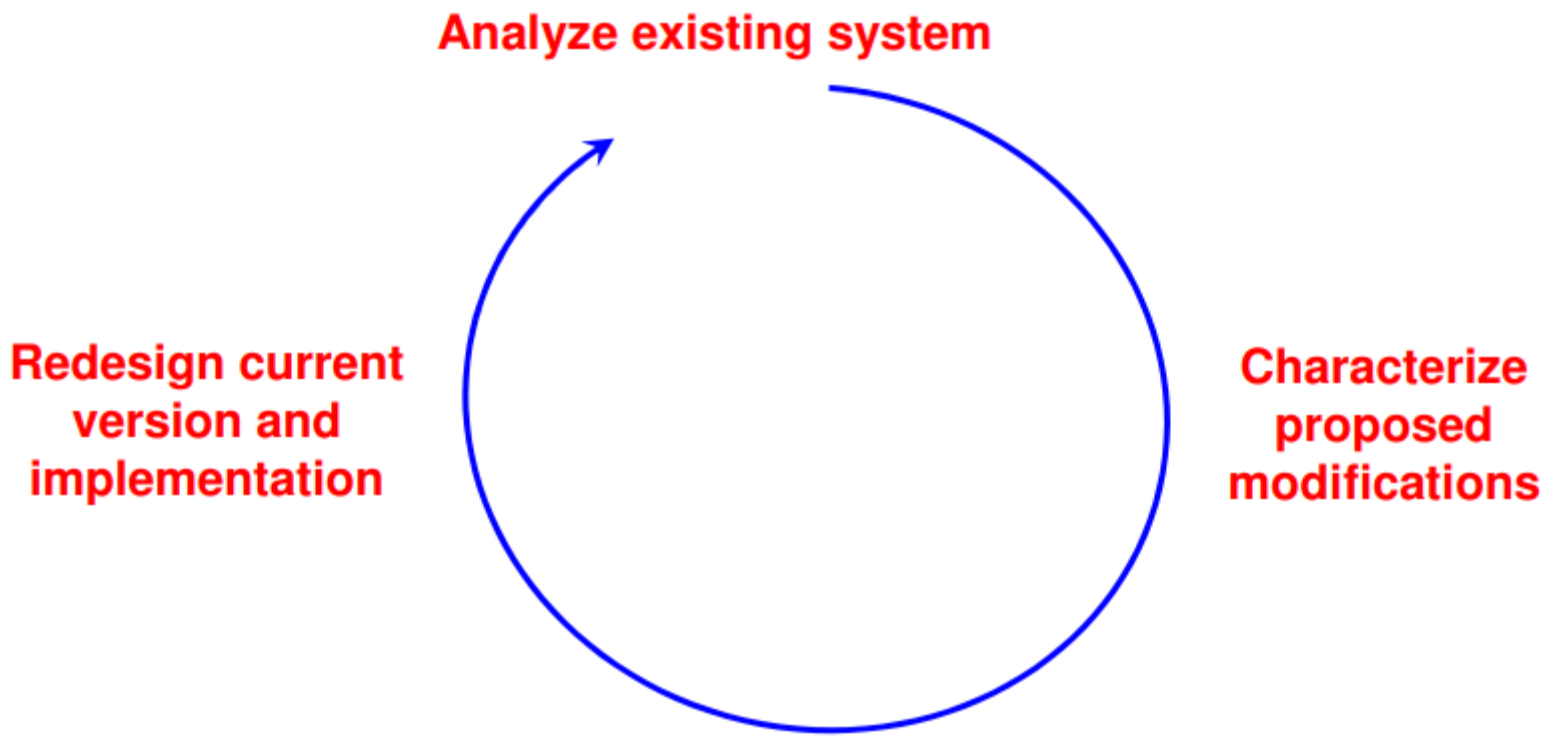
- Quick-fix Model

This is basically an adhoc approach to maintaining software. It is a fire fighting approach, waiting for the problem to occur and then trying to fix it as quickly as possible.



Software Maintenance

- Iterative Enhancement Model
  - Analysis
  - Characterization of proposed modifications
  - Redesign and implementation

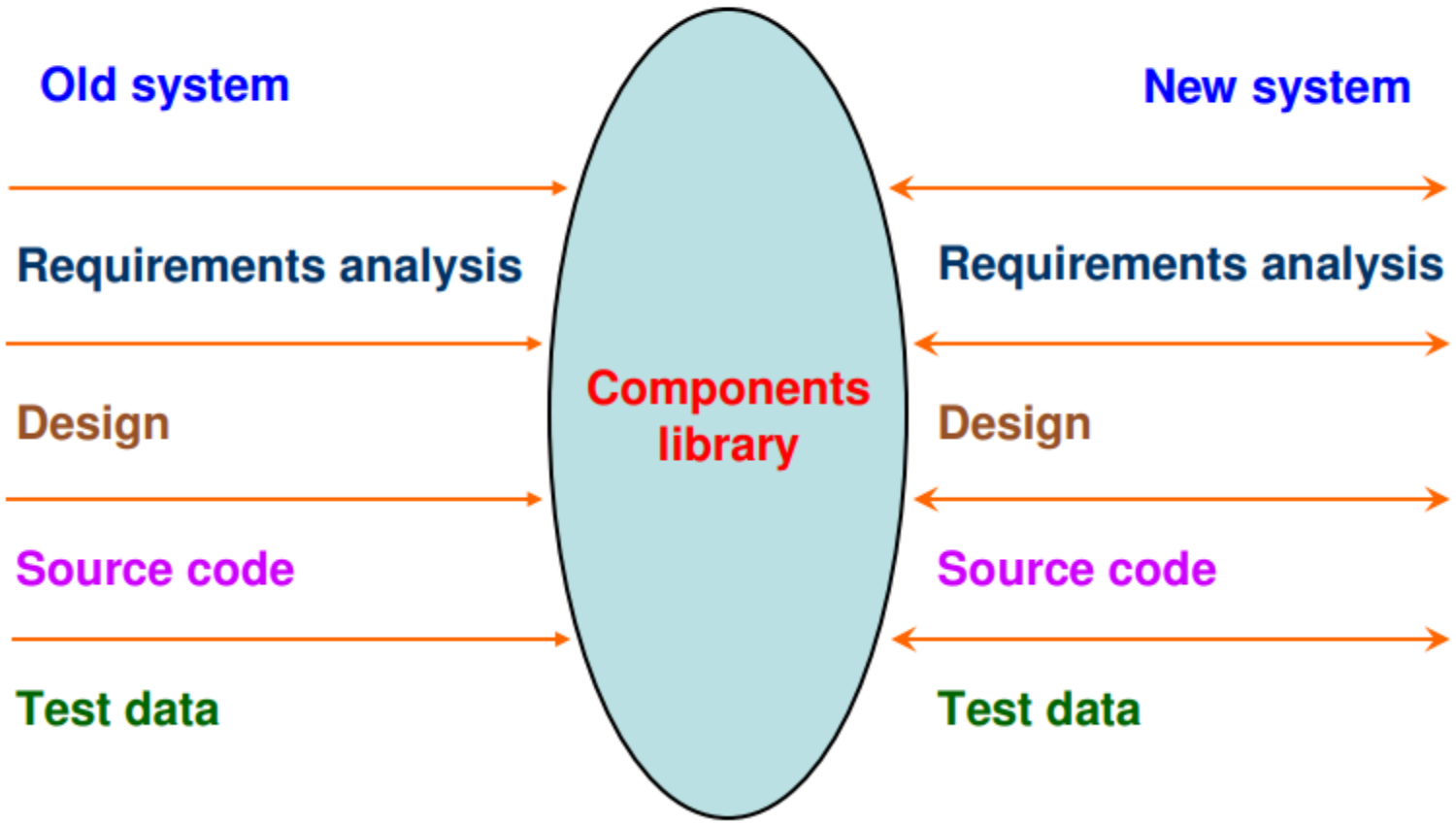

Analyze existing system
Characterize proposed modifications
Redesign current version and implementation

- Reuse Oriented Model

The reuse model has four main steps:

1. Identification of the parts of the old system that are candidates for reuse.
2. Understanding these system parts.
3. Modification of the old system parts appropriate to the new requirements.
4. Integration of the modified parts into the new system.

Software Maintenance



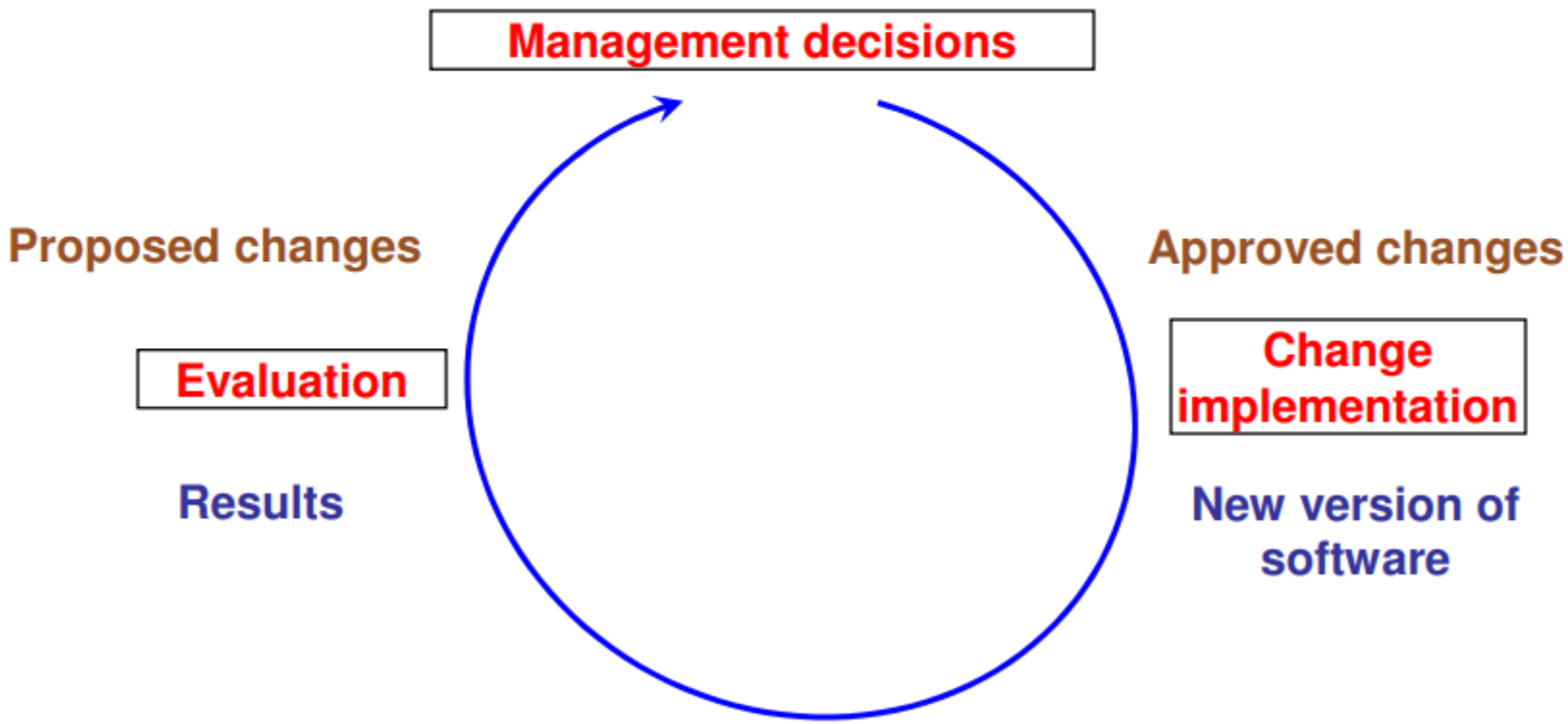
Software Maintenance

- Boehm's Model

Boehm proposed a model for the maintenance process based upon the economic models and principles.

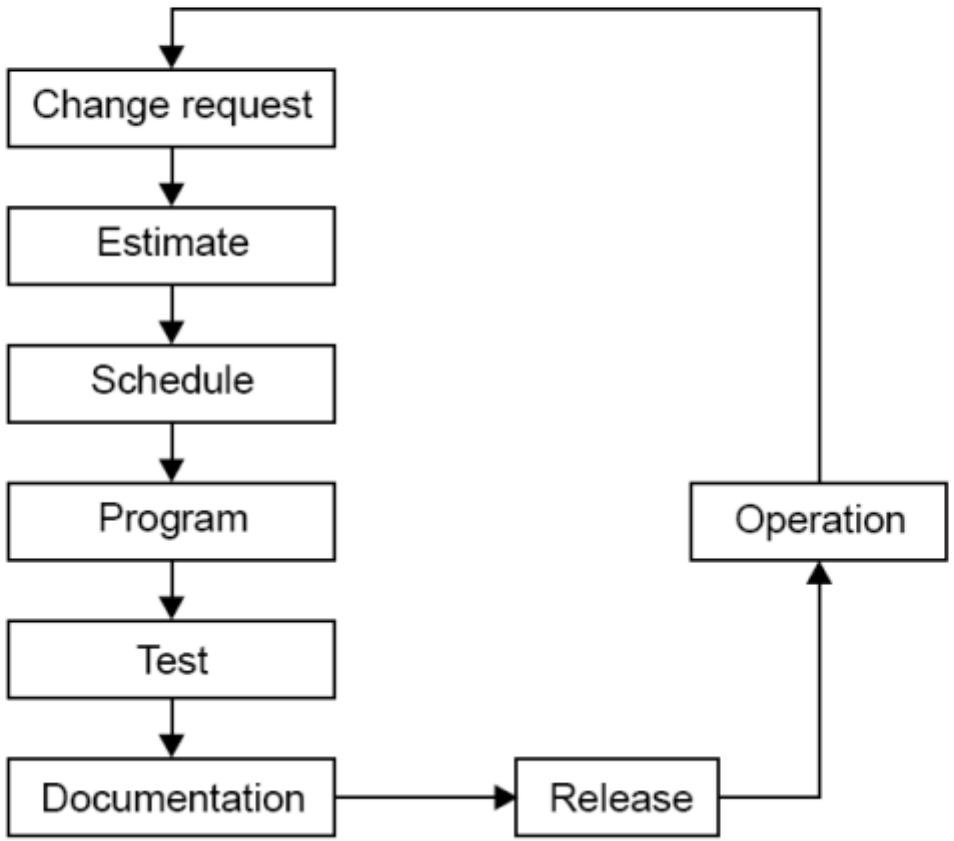
Boehm represent the maintenance process as a closed loop cycle.

Software Maintenance



- ## Taute Maintenance Model

It is a typical maintenance model and has eight phases in cycle fashion. The phases are shown in Fig. 7



Software Maintenance

Phases :

1. Change request phase
2. Estimate phase
3. Schedule phase
4. Programming phase
5. Test phase
6. Documentation phase
7. Release phase

Software Maintenance

- Belady and Lehman Model

$$M = P + Ke^{(c-d)}$$

where

M : Total effort expended

P : Productive effort that involves analysis, design, coding, testing and evaluation.

K : An empirically determined constant.

c : Complexity measure due to lack of good design and documentation.

d : Degree to which maintenance team is familiar with the software.

Software Maintenance

Example

The development effort for a software project is 500 person months. The empirically determined constant (K) is 0.3. The complexity of the code is quite high and is equal to 8. Calculate the total effort expended (M) if

(i) maintenance team has good level of understanding of the project (d=0.9)

(ii) maintenance team has poor understanding of the project (d=0.1)

Solution

Development effort (P) = 500 PM

K = 0.3

C = 8

(i) maintenance team has good level of understanding of the project (d=0.9)

$$M = P + Ke^{(c-d)}$$

$$= 500 + 0.3e^{(8-0.9)}$$

$$= 500 + 363.59 = 863.59 \text{ PM}$$

(ii) maintenance team has poor understanding of the project (d=0.1)

$$M = P + Ke^{(c-d)}$$

$$= 500 + 0.3e^{(8-0.1)}$$

$$= 500 + 809.18 = 1309.18 \text{ PM}$$

- Boehm Model

Boehm used a quantity called Annual Change Traffic (ACT).

"The fraction of a software product's source instructions which undergo change during a year either through addition, deletion or modification".

$$ACT = \frac{KLOC_{added} + KLOC_{deleted}}{KLOC_{total}}$$

AME = ACT x SDE

Where, **SDE :** Software development effort in person months

ACT : Annual change Traffic

EAF : Effort Adjustment Factor

AME = ACT * SDE * EAF

Example

Annual Change Traffic (ACT) for a software system is 15% per year. The development effort is 600 PMs. Compute estimate for Annual Maintenance Effort (AME). If life time of the project is 10 years, what is the total effort of the project ?

Solution

The development effort = 600 PM

Annual Change Traffic (ACT) = 15%

Total duration for which effort is to be calculated = 10 years

The maintenance effort is a fraction of development effort and is assumed to be constant.

$$AME = ACT \times SDE$$

$$= 0.15 \times 600 = 90 \text{ PM}$$

Maintenance effort for 10 years = 10 x 90 = [redacted] 900PM

Total effort = 600 + 900 = 1500 PM

- ISO 9000

The SEI capability maturity model initiative is an attempt to improve software quality by improving the process by which software is developed.

ISO-9000 series of standards is a set of document dealing with quality systems that can be used for quality assurance purposes. ISO-9000 series is not just software standard. It is a series of five related standards that are applicable to a wide variety of industrial activities, including design/ development, production, installation, and servicing. Within the ISO 9000 Series, standard ISO 9001 for quality system is the standard that is most applicable to software development.

Software Reverse Engineering

It is a process of recovering the design, requirement specifications and functions of a product from an analysis of its code.

It builds a program database and generates information from this.

The purpose of reverse engineering is to facilitate the maintenance work by improving the understandability of a system and to produce the necessary documents for a legacy system.

- **Reverse Engineering Goals:**

- Cope with Complexity.
- Recover lost information.
- Detect side effects.
- Synthesise higher abstraction.
- Facilitate Reuse.

Steps of Software Reverse Engineering:

1.Collection Information:
This step focuses on collecting all possible information (i.e., source design documents etc.) about the software.
2.Examining the information:
The information collected in step-1 as studied so as to get familiar with the system.
3.Extracting the structure:
This step concerns with identification of program structure in the form of structure chart where each node corresponds to some routine.
4.Recording the functionality:
During this step processing details of each module of the structure, charts are recorded using structured language like decision table, etc.

1.Recording data flow:

From the information exA tracted in step-3 and step-4, set of data flow diagrams are derived to show the flow of data among the processes.

2.Recording control flow:

High level control structure of the software is recorded.

3.Review extracted design:

Design document extracted is reviewed several times to ensure consistency and correctness. It also ensures that the design represents the program.

4.Generate documentation:

Finally, in this step, the complete documentation including SRS, design document, history, overview, etc. are recorded for future use.

Reverse Engineering Tools:

Reverse engineering if done manually would consume lot of time and human labour and hence must be supported by automated tools. Some of tools are given below:

- **CIAO and CIA:** A graphical navigator for software and web repositories along with a collection of Reverse Engineering tools.
- **Rigi:** A visual software understanding tool.
- **Bunch:** A software clustering/modularization tool.
- **GEN++:** An application generator to support development of analysis tools for the C++ language.
- **PBS:** Software Bookshelf tools for extracting and visualizing the architecture of programs.

Software Re-engineering

It is a process of software development which is done to improve the maintainability of a software system. Re-engineering is the examination and alteration of a system to reconstitute it in a new form. This process encompasses a combination of sub-processes like reverse engineering, forward engineering, reconstructing etc.

Objectives of Re-engineering:

- To describe a cost-effective option for system evolution.
- To describe the activities involved in the software maintenance process.
- To distinguish between software and data re-engineering and to explain the problems of data re-engineering.

Steps involved in Re-engineering:

1. Inventory Analysis
2. Document Reconstruction
3. Reverse Engineering
4. Code Reconstruction
5. Data Reconstruction
6. Forward Engineering