Phases of S/W Development Processes

REQUIREMENTS ANALYSIS AND SPECIFICATION

- Experienced developers take considerable time to understand the exact requirements of the customer and to meticulously document those.
- They know that without a clear understanding of the problem and proper documentation of the same, it is impossible to develop a satisfactory solution

- The goal of the requirements analysis and specification phase is to clearly understand the customer requirements and to systematically organise the requirements into a document called the Software Requirements Specification (SRS) document.
- The SRS document is the final outcome of the requirements analysis and specification phase.
- The engineers who gather and analyse customer requirements and then write the requirements specification document are known as **system analysts.**

How is the SRS document validated

- Once the SRS document is ready, it is first review ed internally by the project team to ensure that it accurately captures all the user requirements, and that it is understandable, consistent, unambiguous, and complete.
- The SRS document is then given to the customer for review.
- After the customer has reviewed the SRS document and agrees to it, it forms the basis for all future development activities and also serves as a contract document between the customer and the development organisation.

Requirements Analysis and Specification phase activities ?

- Requirements gathering and analysis
- Requirements specification

Users of SRS Document

- Users, customers, and marketing personnel
- Software developers
- Test engineers
- User documentation writers
- Project managers
- Maintenance engineers

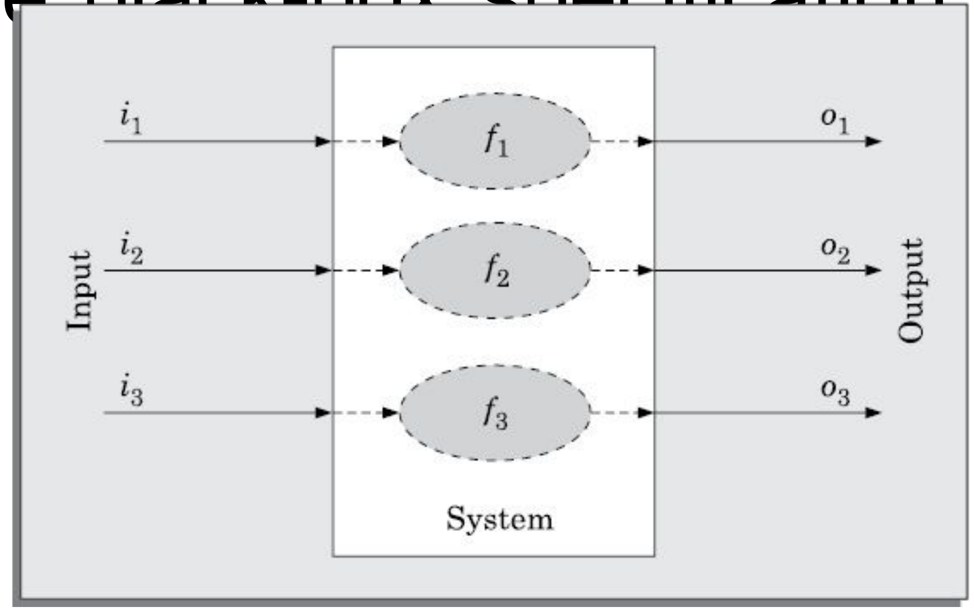
Why Spend Time and Resource to Develop an SRS Document?

- Basis for starting the software development work.
- Forms an agreement between the customers and the developers
- Reduces future reworks
- Provides a basis for estimating costs and schedules
- Provides a baseline for validation and verification
- Facilitates future extensions

Characteristics of a Good SRS Document

- IEEE Recommended Practice for Software Requirements Specifications[IEEE830] describes the content and qualities of a good software requirements specification (SRS).
- Some of the identified desirable qualities of an SRS document are the following:
- Concise
- Implementation-independent
- Traceable
- Modifiable
- Identification of response to undesired events
- Verifiable

- The SRS document should describe the system to be developed as a black box, and should specify only the externally visible behaviour of the system. For this reason, the SRS document is also called the black-box specification of the software being



The black-box view of a system as performing a set of functions.

Attributes of Bad SRS Documents

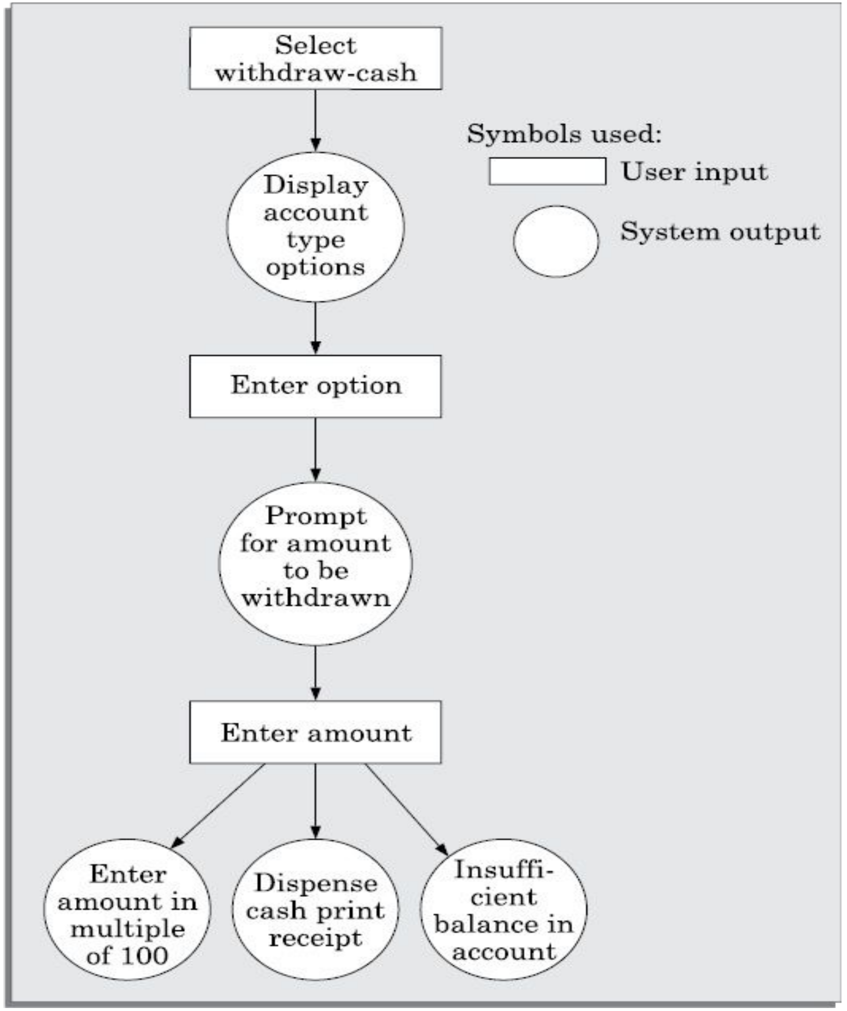
- The most damaging problems are incompleteness, ambiguity, and contradictions
- Over-specification
- Forward references
- Wishful thinking
- Noise

Important Categories of Customer Requirements

- As per the IEEE 830 guidelines, the important categories of user requirements are the following.
- Functional requirements
- Non-functional requirements
  - Design and implementation constraints
  - External interfaces required
  - Other non-functional requirements
- Goals of implementation

Functional requirements

- The functional requirements capture the functionalities required by the users from the system
- The functional requirements capture the functionalities required by the users from the system
- These functions can be considered similar to a mathematical function $f : I \rightarrow O$, meaning that a function transforms an element (ii) in the input domain (I) to a value (oi) in the output (O)
- To document the functional requirements of a system, it is necessary to identify the high-level functions of the systems by reading the informal documentation of the gathered requirements.
- The high-level functions would be split into smaller subrequirements.
- Each high-level function is an instance of use of the system (use case) by the user in some way.


Select withdraw-cash
Display account type options
Enter option
Prompt for amount to be withdrawn
Enter amount
Enter amount in multiple of 100
Dispense cash print receipt
Insuffi-cient balance in account
Symbols used:
User input
System output

User and system interactions in high-level functional requirement.

How to Document the Functional Requirements

- A function can be documented by identifying the state at which the data is to be input to the system, its input data domain, the output data domain, and the type of processing to be carried on the input data to obtain the output data.

Ex: Withdraw cash from ATM

- R.1: Withdraw cash
  - Description:The withdraw cash function first determines the type of account that the user has and the account number from which the user wishes to withdraw cash. It checks the balance to determine whether the requested amount is available in the account. If enough balance is available, it outputs the required cash, otherwise it generates an error Message.
- R.1.1: Select withdraw amount option
  - Input: “Withdraw amount” option selected Output: User prompted to enter the account type
- R.1.2: Select account type
  - Input : User selects option from savings/checking/deposit.
  - Output: Prompt to enter amount
- R.1.3: Get required amount
  - Input: Amount to be withdrawn in integer values greater than 100 and less than 10,000 in multiples of 100.
  - Output: The requested cash and printed transaction statement.
  - Processing: The amount is debited from the user’s account if sufficient balance is available, otherwise an error message displayed.

Example: Search book availability in library:

An initial informal description of a required functionality is usually given by the customer as a statement of purpose (SoP) based on which an later requirements gathering, the analyst understand the functionality. Ho wever, the functionalities of search book availability is intuitively obvious to any one who has used a library. So, we are not including an informal description of search book availability functionality here and in the following, we documents this functional requirement.

R.1: Search book

Description Once the user selects the search option, he would be asked to enter the keywords. The system would search the book in the book list based on the key words entered. After making the search, the system should output the details of all books whose title or author name match any of the key words entered. The book details to be displayed include: title, author name, publisher name, year of publication, ISBN number, catalog number, and the location in the library.

R.1.1: Select search option

Input: “Search” option

Output: User prompted to enter the key words

R.1.2: Search and display

Input: Key words

Output: Details of all books whose title or author name matches any of the key words entered by the user. The book details displayed would include— title of the book, author name, ISBN number, catalog number, yea of publication, number of copies available, and the location in the library.

Processing: Search the book list based on the key words:

R.2: Renew book

Description: When the "renew" option is selected, the user is asked to enter his membership number and password. After password validation, the list of the books borrowed by him are displayed. The user can renew any of his borrowed books by indicating them. A requested book cannot be renewed if it is reserved by another user. In this case, an error message would be displayed.

R.2.1: Select renew option

State: The user has logged in and the main menu has been displayed.

Input: "Renew" option selection.

Output: Prompt message to the user to enter his membership number and password.

R.2.2: Login

State: The renew option has been selected.

Input: Membership number and password.

Output: List of the books borrowed by the user is displayed, and user is prompted to select the books to be renewed, if the password is valid. If the password is invalid, the user is asked to re-enter the password.

Processing: Password validation, search the books issued to the user from the borrower's list and display.

Next function: R.2.3 if password is valid and R.2.2 if password is invalid.

R.2.3: Renew selected books

Input: User choice for books to be renewed out of the books borrowed by him.

Output: Confirmation of the books successfully renewed and apology message for the books that could not be renewed.

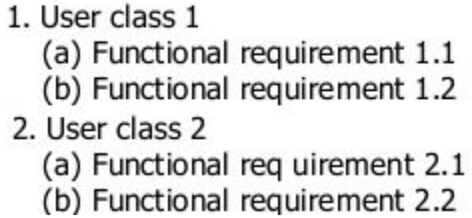
Processing: Check if any one has reserved any of the requested books. Renew the books selected by the user in the borrower's list, if no one has reserved those books.

- Specification of large software:
  - split the requirements into sections of related requirements.
  - functional requirements of a academic institute automation software can be split into sections such as accounts, academics, inventory, publications, etc.
  - trade house automation software functional req:
    - Customer management
    - Account management
    - Purchase management
    - Vendor management
    - Inventory management

Organisation of the SRS Document

- organisation of an SRS document as prescribed by the IEEE 830 standard[IEEE 830]
- the three basic issues that any SRS document should discuss are:
  - functional requirements,
  - non-functional requirements
  - guidelines for system implementation

- Introduction
  - ➔ Purpose
  - ➔ Project scope
  - ➔ Environmental characteristics
- Overall description of organisation of SRS document
  - ➔ Product perspective
  - ➔ Product features
  - ➔ User classes
  - ➔ Operating environment
  - ➔ Design and implementation constraints
  - ➔ User documentation
- Functional requirements for organisation of SRS docu



1. User class 1
   (a) Functional requirement 1.1
   (b) Functional requirement 1.2
2. User class 2
   (a) Functional req uirement 2.1
   (b) Functional requirement 2.2

- External interface requirements
  - ➔ User interfaces
  - ➔ Hardware interfaces
  - ➔ Software interfaces
  - ➔ Communications interfaces
- Other non-functional requirements for organisation of SRS document
  - ➔ Performance requirements
  - ➔ Safety requirements
  - ➔ Security requirements

Example 4.9 (Personal library software): It is proposed to develop a software that would be used by individuals to manage their personal collection of books. The following is an informal description of the requirements of this software as worked out by the marketing department. Develop the functional and non-functional requirements for the software. A person can have up to a few hundreds of books. The details of all the books such as name of the book, year of publication, date of purchase, price, and publisher would be entered by the owner. A book should be assigned a unique serial number by the computer. This number would be written by the owner using a pen on the inside page of the book. Only a registered friend can be lent a book. While registering a friend, the following data would have to be supplied—name of the friend, his address, land line number, and mobile number. Whenever a book issue request is given, the name of the friend to whom the book is to be issued and the unique id of the book is entered. At this, the various books outstanding against the borrower along with the date borrowed are displayed for information of the owner. If the owner wishes to go ahead with the issue of the book, then the date of issue, the title of the book, and the unique identification number of the book are stored. When a friend returns a book, the date of return is stored and the book is removed from his borrowing list. Upon query, the software should display the name, address, and telephone numbers of each friend against whom books are outstanding along with the titles of the outstanding books and the date on which those were issued. The software should allow the owner to update the details of a friend such as his address, phone, telephone number, etc. It should be possible for the owner to delete all the data pertaining

The records should be stored using a free (public domain) data base management system. The software should run on both Windows and Unix machines.

Whenever the owner of the library software borrows a book from his friends, would enter the details regarding the title of the book, and the date borrowed and the friend from whom he borrowed it. Similarly, the return details of books would be entered. The software should be able to display all the books borrowed from various friends upon request by the owner.

It should be possible for any one to query about the availability of a particular book through a web browser from any location. The owner should be able to query the total number of books in the personal library, and the total amount he has invested in his library. It should also be possible for him to view the number of books borrowed and returned by any (or all) friend(s) over any specified time.

Functional requirements

The software needs to support three categories of functionalities as described below:

1. Manage own books

1.1 Register book

Description: To register a book in the personal library, the details of a book, such as name, year of publication, date of purchase, price and publisher are entered. This is stored in the database and a unique serial number is generated.

Input: Book details

Output: Unique serial number

R.1.2: Issue book

Description: A friend can be issued book only if he is registered. The various books outstanding against him along with the date borrowed are first displayed.

R.1.2.1: Display outstanding books

Description: First a friend's name and the serial number of the book to be issued are entered. Then the books outstanding against the friend should be displayed.

Input: Friend name

Output: List of outstanding books along with the date on which each was borrowed.

R.1.2.2: Confirm issue book

If the owner confirms, then the book should be issued to him and the relevant records should be updated.

Input: Owner confirmation for book issue. Output: Confirmation of book issue.

R.1.3: Query outstanding books

Description: Details of friends who have books outstanding against their name is displayed.

Input: User selection

Output: The display includes the name, address and telephone numbers of each friend against whom books are outstanding along with the titles of the outstanding books and the date on which those were issued.

R.1.4: Query book

Description: Any user should be able to query a particular book from anywhere using a web browser.

Input: Name of the book.

Output: Availability of the book and whether the book is issued out.

R.1.5: Return book

Description: Upon return of a book by a friend, the date of return is stored and the book is removed from the borrowing list of the concerned friend.

Input: Name of the book.

Output: Confirmation message.

2. Manage friend details

R.2.1: Register friend

Description: A friend must be registered before he can be issued books. After the registration data is entered correctly, the data should be stored and a confirmation message should be displayed.

Input: Friend details including name of the friend, address, land line number and mobile number.

Output: Confirmation of registration status.

Description: When a friend's registration information changes, the same must be updated in the computer.

R.2.2.1: Display current details

Input: Friend name.

Output: Currently stored details.

R2.2.2: Update friend details

Input: Changes needed.

Output: Updated details with confirmation of the changes.

R.3.3: Delete a friend record

Description: Delete records of inactive members.

Input: Friend name.

Output: Confirmation message.

- Manage own books
  - Register books
  - Issue books
- Manage friend details
  - Register friend

3. Manage borrowed books

R.3.1: Register borrowed books

Description: The books borrowed by the user of the personal library are registered.

Input: Title of the book and the date borrowed.

Output: Confirmation of the registration status.

R.3.2: Deregister borrowed books

Description: A borrowed book is deregistered when it is returned.

Input: Book name.

Output: Confirmation of deregistration.

R.3.3: Display borrowed books

Description: The data about the books borrowed by the owner are displayed.

Input: User selection.

Output: List of books borrowed from other friends.

4. Manage statistics

R.4.1: Display book count

Description: The total number of books in the personal library should be displayed.

Input: User selection.

Output: Count of books.

R4.2: Display amount invested

Description: The total amount invested in the personal library is displayed.

Input: User selection.

Output: Total amount invested.

R.4.2: Display number of transactions *Description:* The total numbers of books issued and returned over a specific period by one (or all) friend(s) is displayed.

Input: Start of period and end of period.

Output: Total number of books issued and total number of books returned.

Non-functional requirements

N.1: Database: A data base management system that is available free of cost in the public domain should be used.

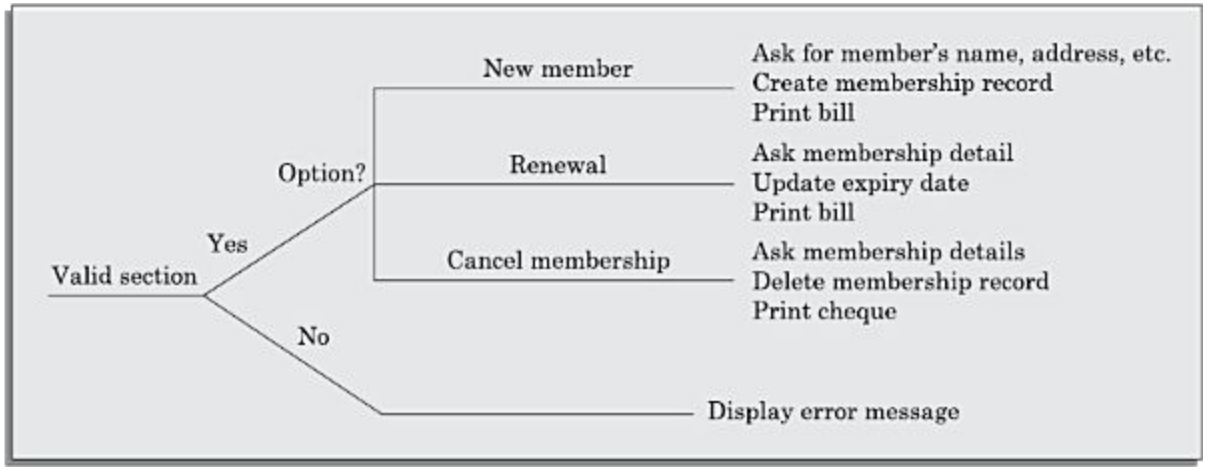
N.2: Platform: Both Windows and Unix versions of the software need to be developed. **N.3: Web-support:** It should be possible to invoke the query book functionality from any place by using a web browser.

Observation: Since there are many functional requirements, the requirements have been organised into four sections: Manage own books, manage friends, manage borrowed books, and manage statistics. Now each section has less than 7 functional requirements. This would not only enhance the readability of the document, but would also help in design.

Techniques for Representing Complex Logic

- A good SRS document should properly characterise the conditions under which different scenarios of interaction occur

Decision tree



Decision table

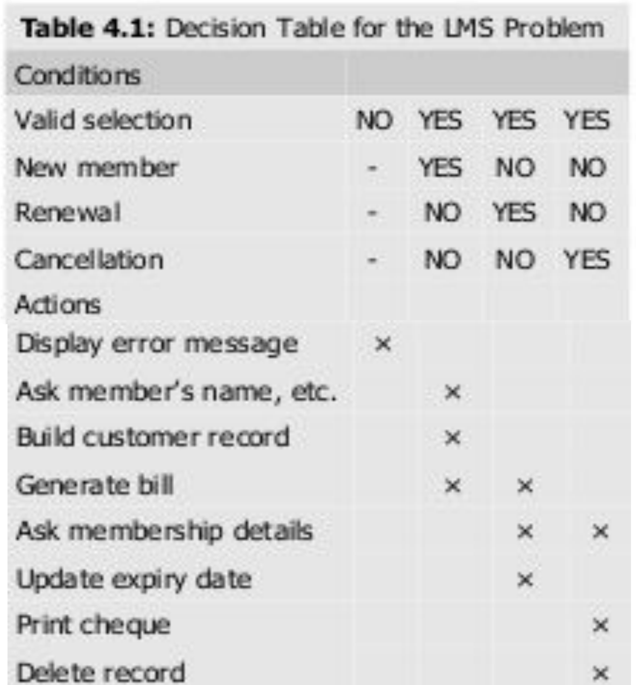
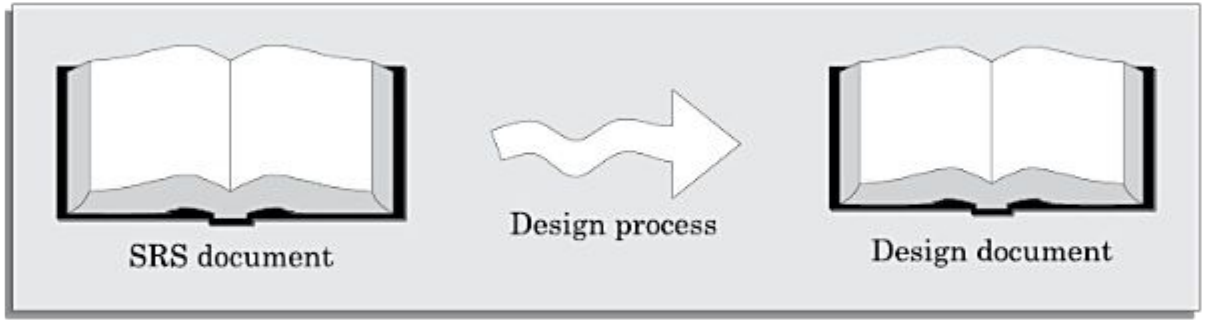


Table 4.1: Decision Table for the LMS Problem

Conditions				
Valid selection	NO	YES	YES	YES
New member	-	YES	NO	NO
Renewal	-	NO	YES	NO
Cancellation	-	NO	NO	YES
Actions				
Display error message	×			
Ask member's name, etc.		×		
Build customer record		×		
Generate bill		×	×	
Ask membership details			×	×
Update expiry date			×	
Print cheque				×
Delete record				×

Design

- The activities carried out during the design phase (called as design process) transform the SRS document into the design document.



Outcome of the Design Process

- Different modules required
- Control relationships among modules
- Interfaces among different modules
- Data structures of the individual modules
- Algorithms required to implement the individual modules

HOW TO CHARACTERIZE A GOOD SOFTWARE DESIGN?

- Correctness
- Understandability
- Efficiency
- Maintainability

COHESION AND COUPLING

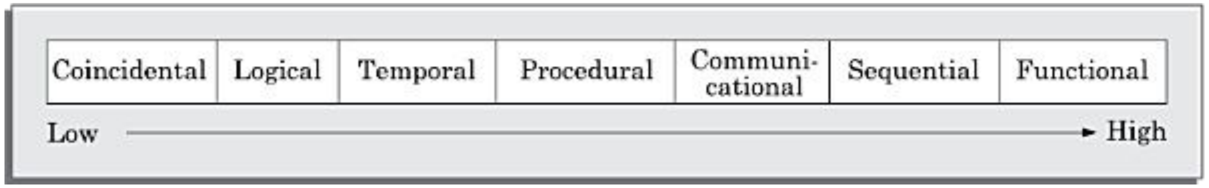
- Cohesion is a measure of the functional strength of a module.
- coupling between two modules is a measure of the degree of interaction (or interdependence) between the two modules.
- Good module decomposition is indicated through high cohesion of the individual modules and low coupling of the modules with each other.
- Two modules are said to be highly coupled, if either of the following two situations arise:
  - If the function calls between two modules involve passing large chunks of shared data, the modules are tightly coupled.
  - If the interactions occur through some shared data, then also we say that they are highly coupled.

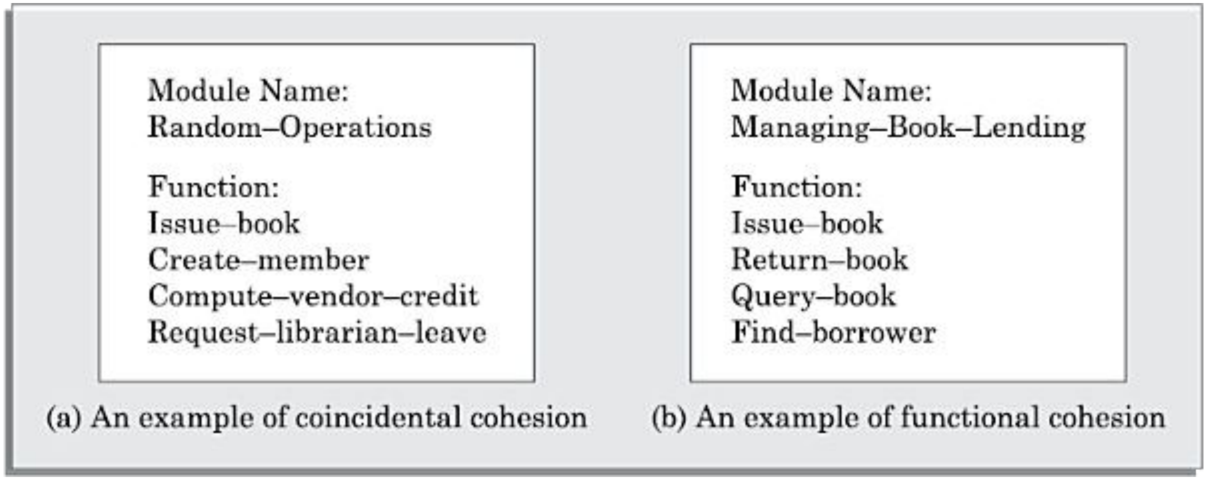
Cohesion

- When the functions of the module co-operate with each other for performing a single objective, then the module has good cohesion
- Functional independence
  - A module that is highly cohesive and also has low coupling with other modules is said to be functionally independent of the other modules.
  - Advantages:
    - Error isolation
    - Scope of reuse
    - Understandability

- Classification of Cohesiveness
  - Cohesiveness of a module is the degree to which the different functions of the module co-operate to work towards a single objective

Classes of Cohesion

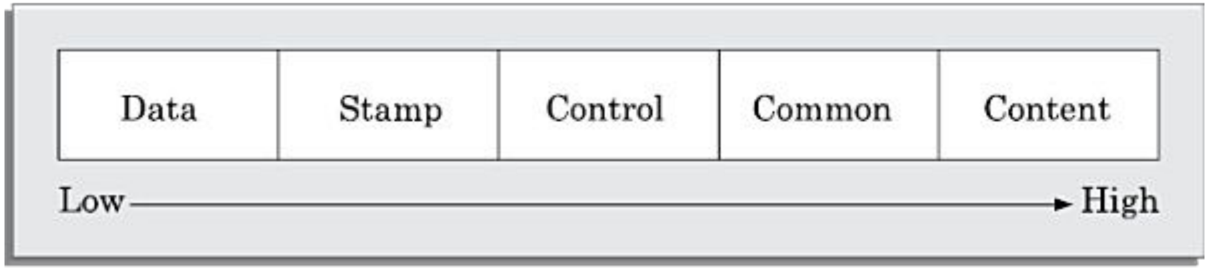




- **Coincidental cohesion:**
  - if the module performs a set of tasks that relate to each other very loosely, if at all.
- **Logical cohesion:**
  - if all elements of the module perform similar operations, such as error handling, data input, data output, etc.
- **Temporal cohesion:**
  - When a module contains functions that are related by the fact that these functions are executed in the same time span, then the module is said to possess temporal cohesion.
- **Procedural cohesion:**
  - if the set of functions of the module are executed one after the other, though these functions may work towards entirely different purposes and operate on very different data.
- **Communicational cohesion:**
  - if all functions of the module refer to or update the same data structure.
- **Sequential cohesion:**
  - if the different functions of the module execute in a sequence, and the output from one function is input to the next in the sequence.
- **Functional cohesion:**
  - if different functions of the module co-operate to complete a single task.

Classification of Coupling

- The degree of coupling between two modules depends on their interface complexity
- **Data coupling:** Two modules are data coupled, if they communicate using an elementary data item that is passed as a parameter between the two, e.g. an integer, a float, a character, etc.
- **Stamp coupling:** Two modules are stamp coupled, if they communicate using a composite data item such as a record in PASCAL or a structure in C
- **Control coupling:** Control coupling exists between two modules, if data from one module is used to direct the order of instruction execution in another.
- **Common coupling:** Two modules are common coupled, if they share some global data items.
- **Content coupling:** Content coupling exists between two modules, if they share code. That is, a jump from one module into the code of another module can occur.



FUNCTION-ORIENTED SOFTWARE DESIGN

- These techniques, to start with, view a system as a black-box that provides a set of services to the users of the software
- These services are also known as high level functions
- During the
- design process, these high-level functions are successively decomposed into more detailed functions.
- The term top-down decomposition i s often used to denote the successive decomposition of a set of high-level functions into more detailed functions.

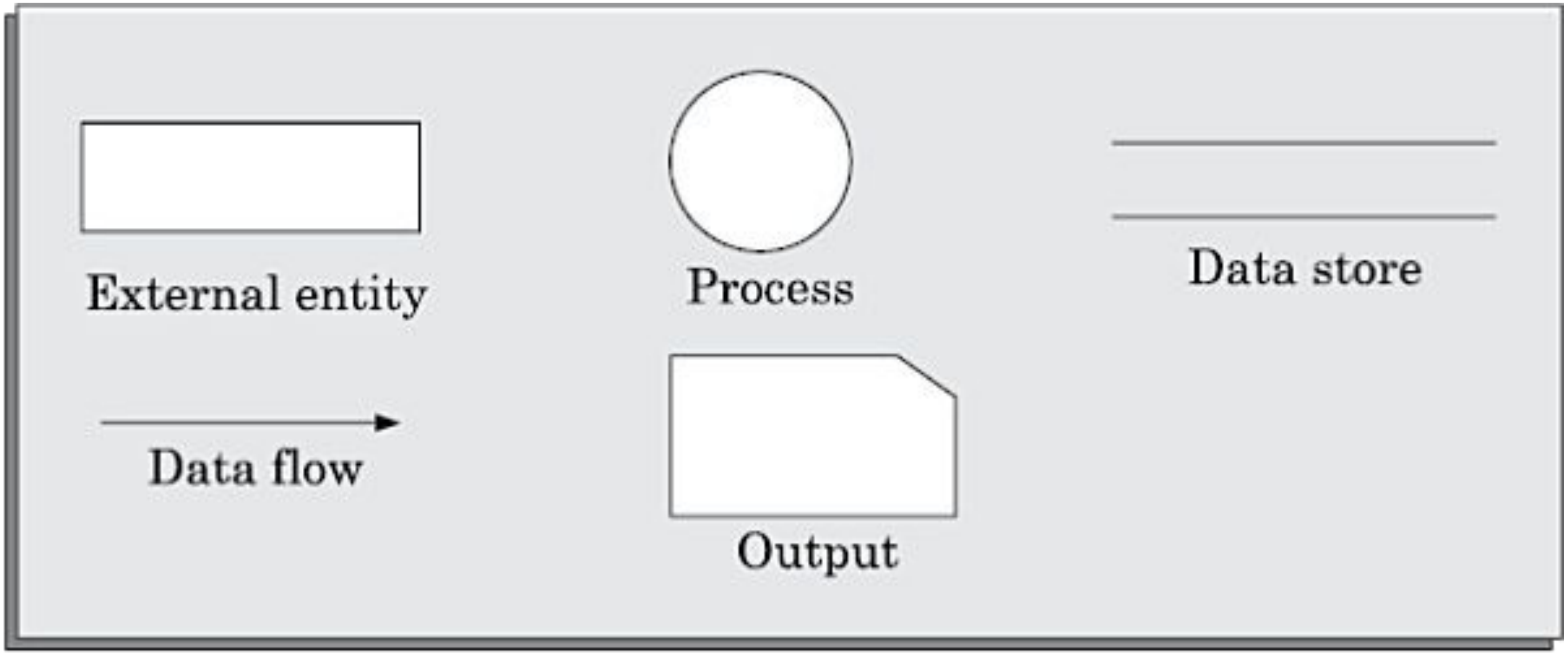
STRUCTURED ANALYSIS

- during structured analysis, the major processing tasks (high-level functions) of the system are analysed, and the data flow among these processing tasks are represented graphically.
- Significant contributions to the development of the structured analysis techniques have been made by Gane and Sarson [1979], and DeMarco and Yourdon [1978].
- The structured analysis technique is based on the following underlying principles:
  - Top-down decomposition approach.
  - Application of divide and conquer principle. Through this each high- level function is independently decomposed into detailed functions.
  - Graphical representation of the analysis results using data flow diagrams (DFDs).

Data Flow Diagrams(DFD)

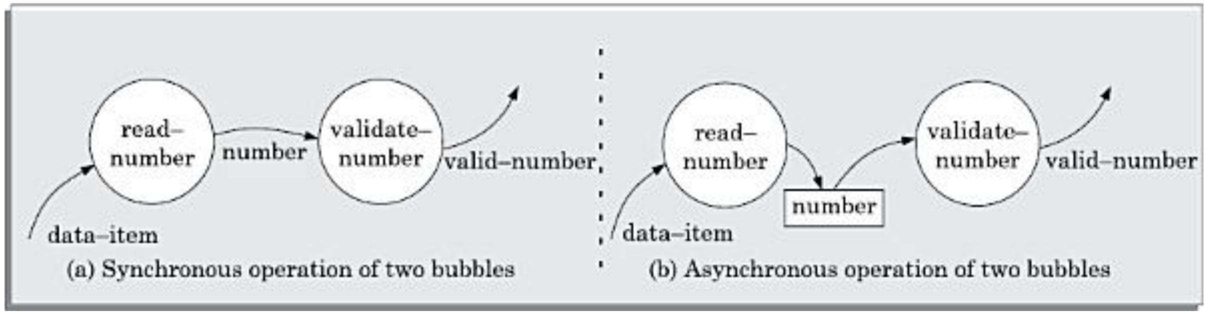
- The DFD (also known as the bubble chart) is a simple graphical formalism that can be used to represent a system in terms of the input data to the system, various processing carried out on those data, and the output data generated by the system.
- The main reason why the DFD technique is so popular is probably because of the fact that DFD is a very simple formalism— it is simple to understand and use.

Primitive symbols used for constructing DFDs



- **Function/Process symbol**: A function is represented using a circle. This symbol is called a process or a bubble. Bubbles are annotated with the names of the corresponding functions.
- **External entity symbol:**An external entity such as a librarian, a library member, etc. is represented by a rectangle.
- **Data flow symbol:**A directed arc (or an arrow) is used as a data flow symbol.
- **Data store symbol:**A data store is represented using two parallel lines. It represents a logical file. That is, a data store symbol can represent either a data structure or a physical file on disk.
- **Output symbol:**The output symbol is used when a hard copy is produced.

Synchronous and asynchronous operations

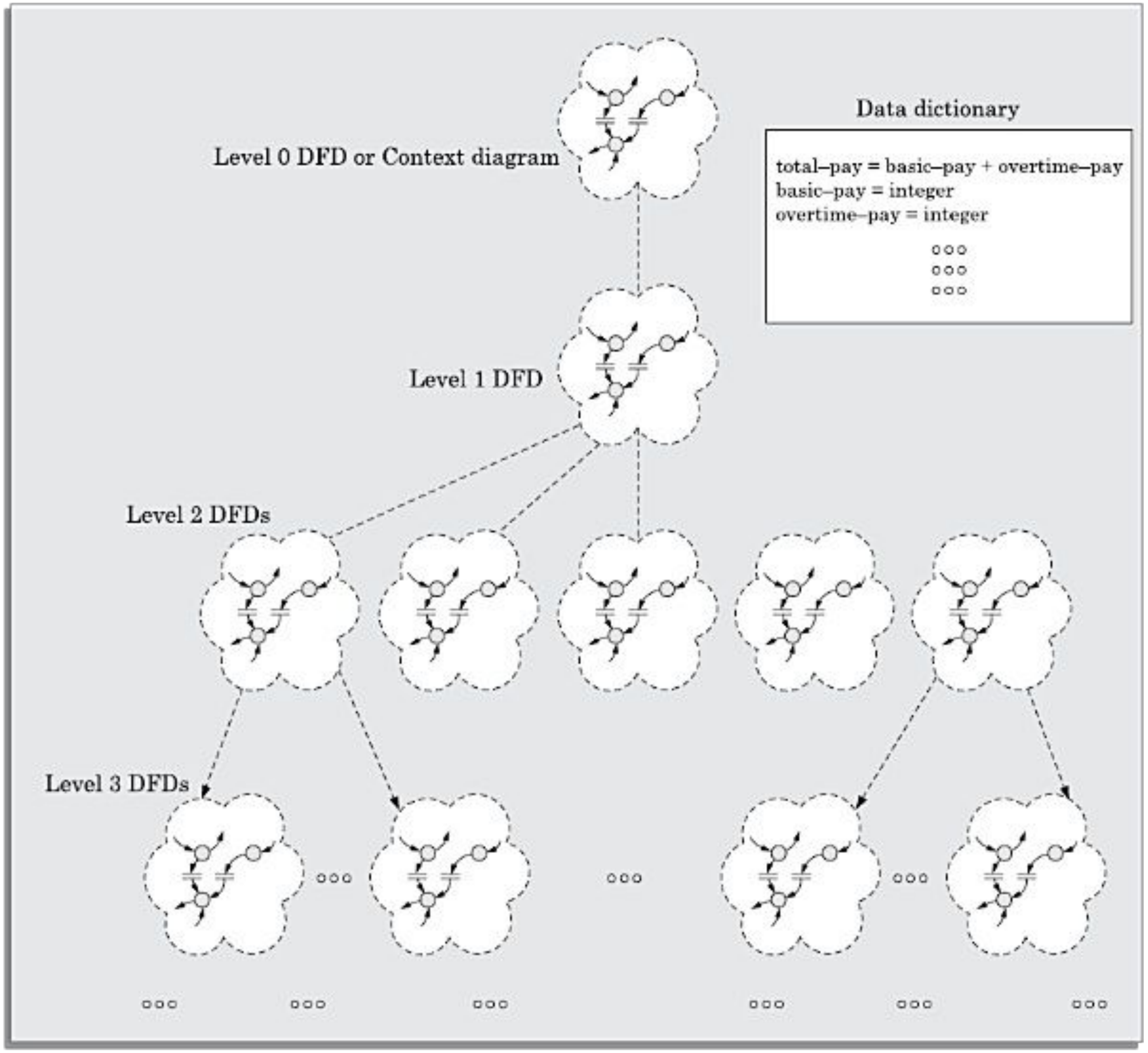


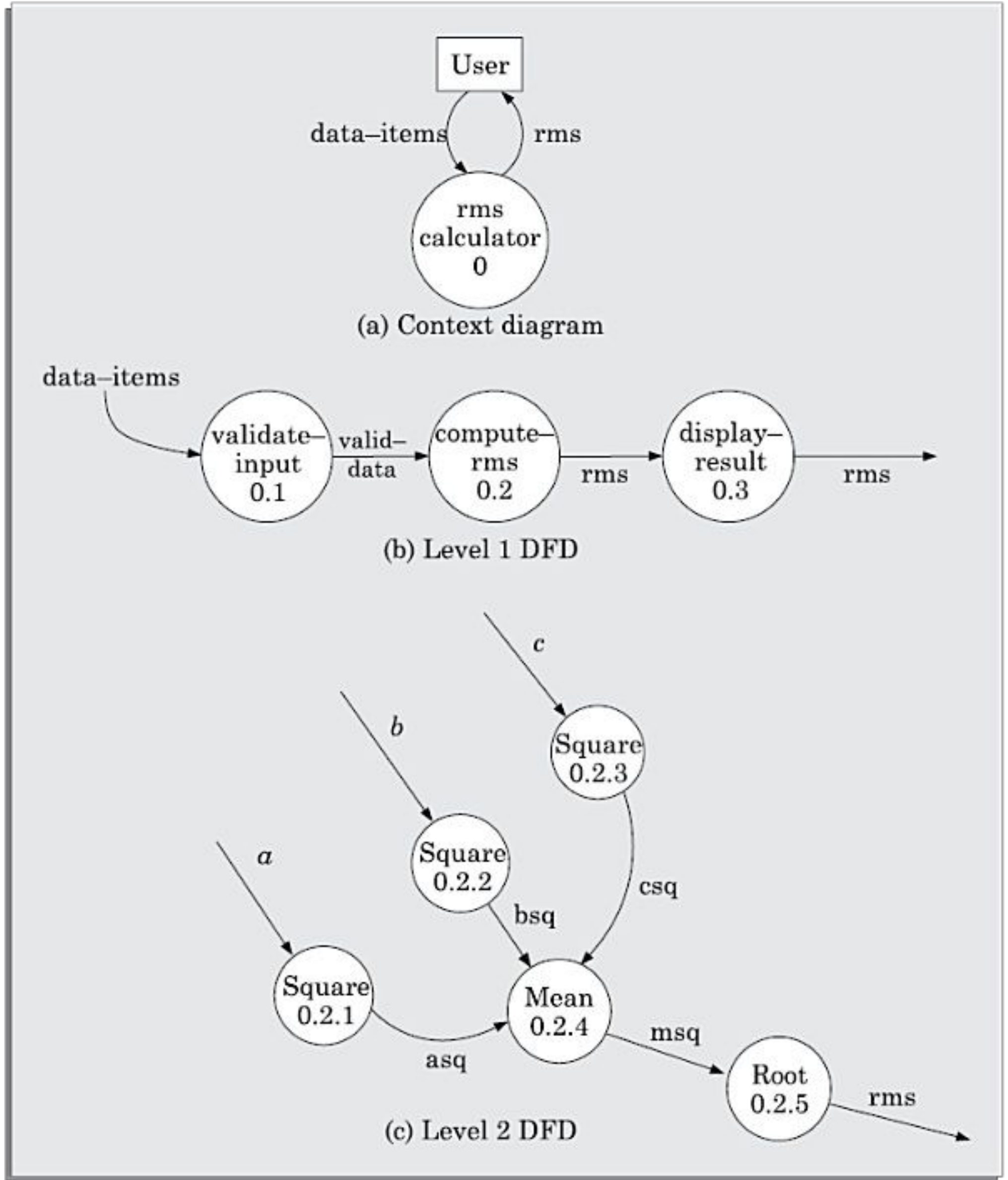
(a) Synchronous operation of two bubbles

(b) Asynchronous operation of two bubbles

Data dictionary

Every DFD model of a system must be accompanied by a data dictionary.
A
data dictionary lists all data items that appear in a DFD model.


Level 0 DFD or Context diagram
Level 1 DFD
Level 2 DFDs
Level 3 DFDs
Data dictionary
total–pay = basic–pay + overtime–pay
basic–pay = integer
overtime–pay = integer



(a) Context diagram

(b) Level 1 DFD

(c) Level 2 DFD

(RMS Calculating Software): A software system called RMS calculating software would read three integral numbers from the user in the range of –1000 and +1000 and would determine the root mean square (RMS) of the three input numbers and display it.

Data dictionary for the DFD model of Example 6.1

data-items: {integer}3

rms: float

valid-data:data-items

a: integer

b: integer

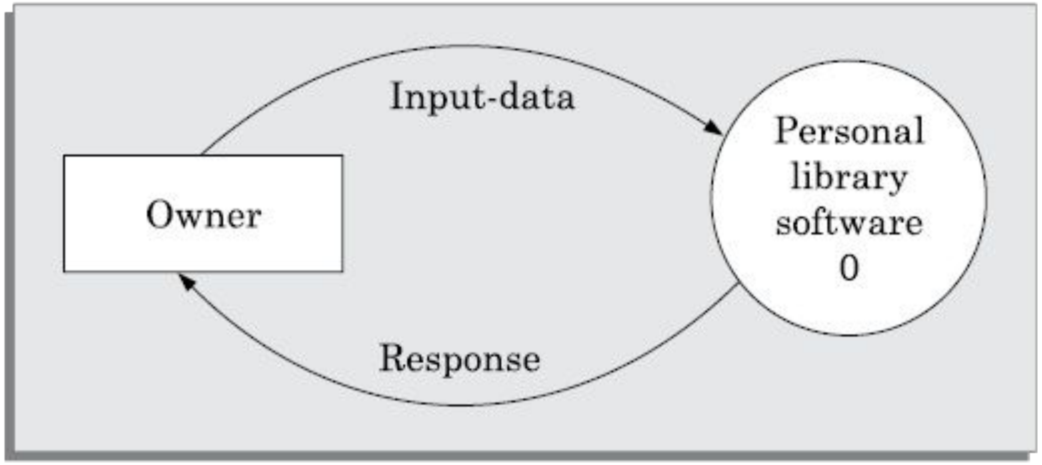
c: integer

asq: integer

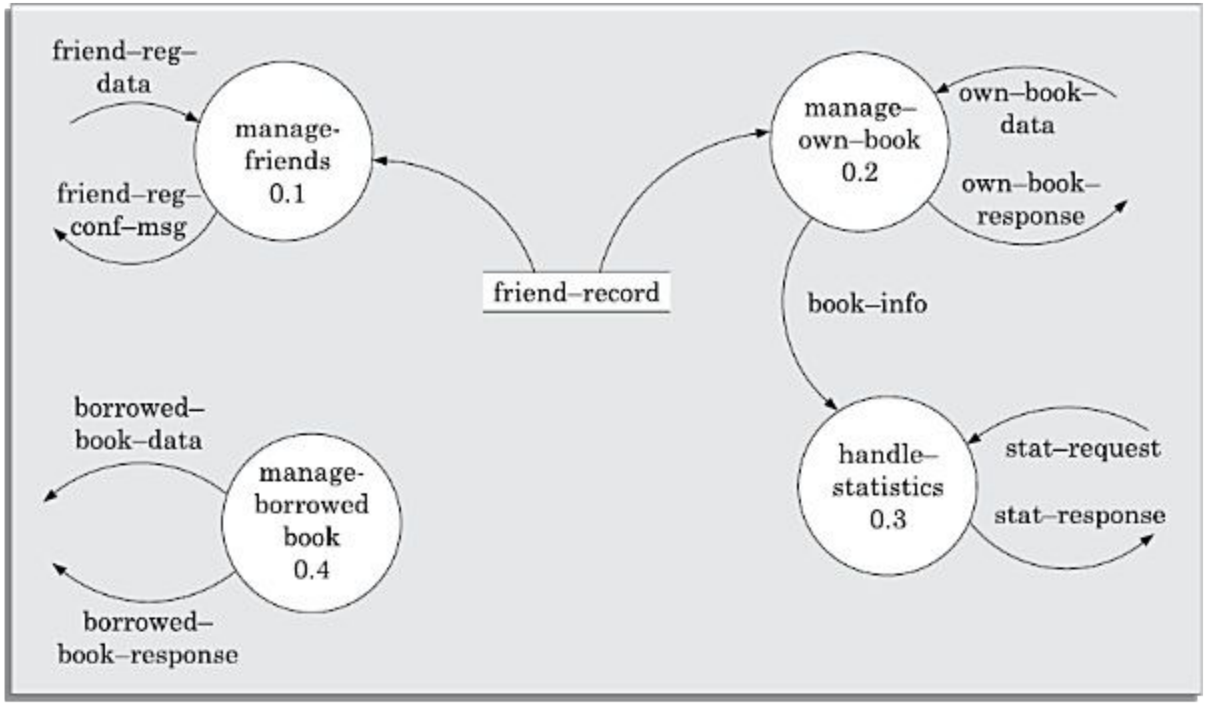
bsq: integer

csq: integer

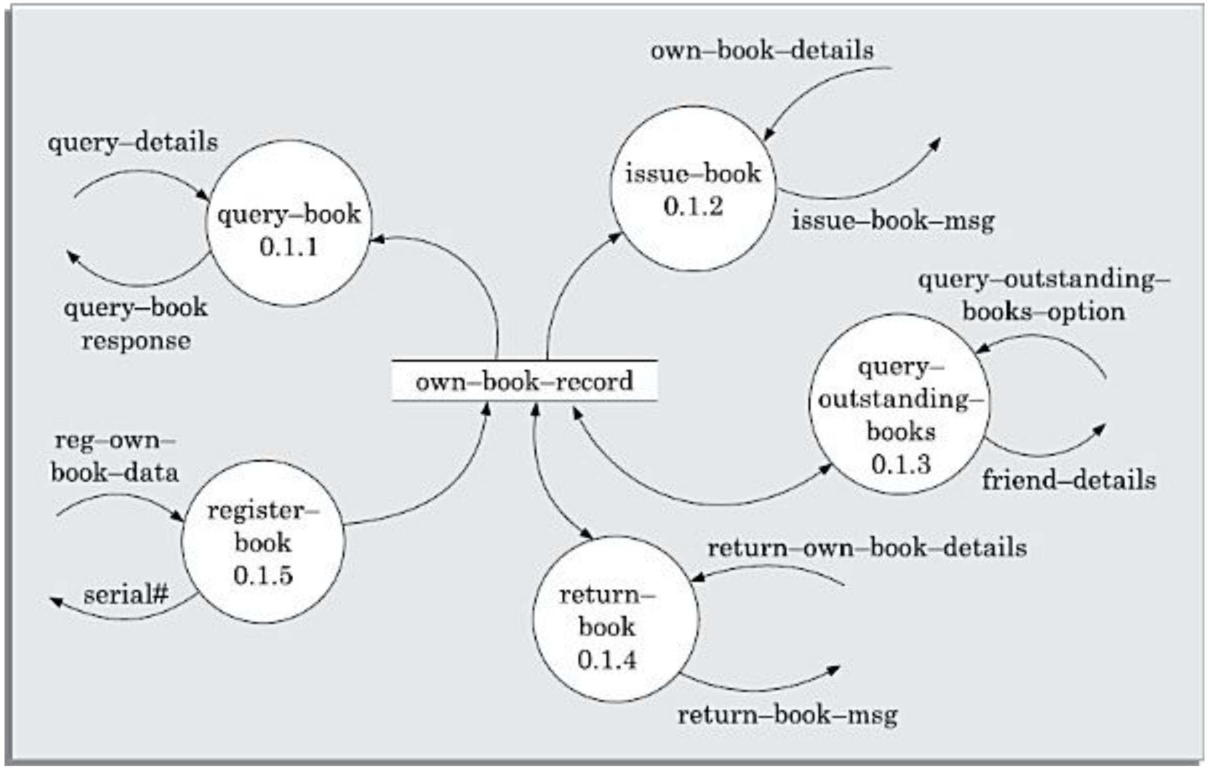
Personal Library Software



Context diagram


friend–reg–data
manage-friends 0.1
friend–reg–conf–msg
friend–record
manage–own–book 0.2
own–book–data
own–book–response
book–info
handle–statistics 0.3
stat–request
stat–response
borrowed–book–data
manage-borrowed book 0.4
borrowed–book–response

Level 1 DFD



own–book–details
query–details
issue–book 0.1.2
query–book 0.1.1
issue–book–msg
query–book response
query–outstanding–books–option
own–book–record
query–outstanding–books 0.1.3
reg–own–book–data
register–book 0.1.5
friend–details
return–own–book–details
serial#
return–book 0.1.4
return–book–msg

Level 2 DFD

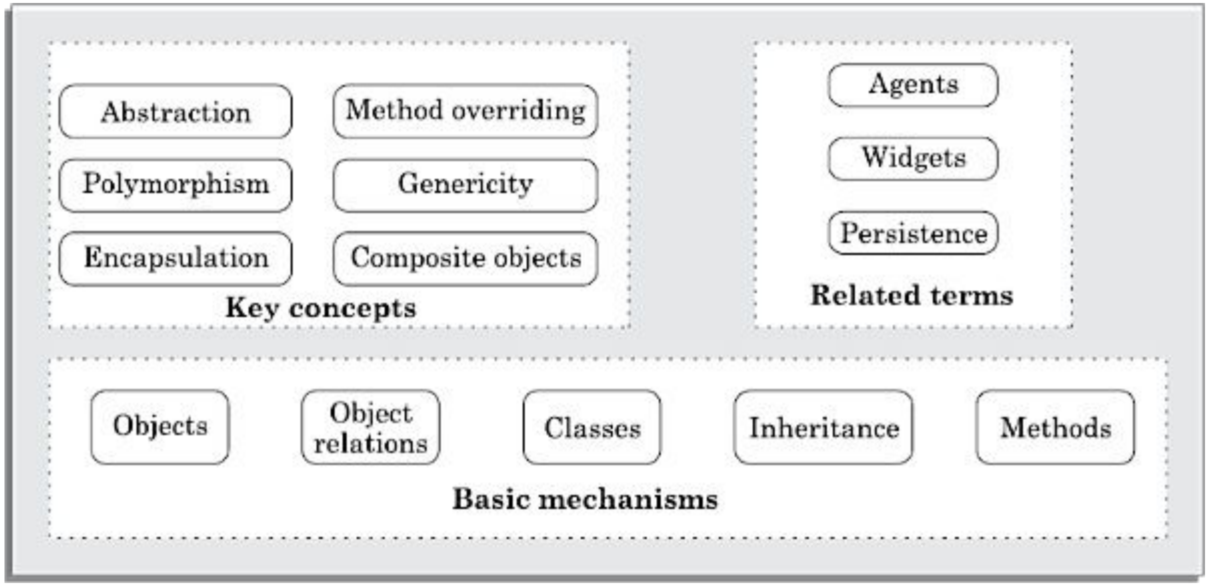
Data dictionary for the DFD model

input-data: friend-reg-data + own-book-data + stat-request +
borrowed-book-data
response: friend-reg-conf-msg + own-book-response + stat-response + borrowed-book-response
own-book-data: query-details + own-book-details + query-outstanding-books-option + return-own book-
details + reg-own-book-data
own-book-response: query-book-response + issue-book-msg + friend-details + return-book- msg + serial#.
borrowed-book-data: borrowed-book-details + book-return-details + display-books-option
borrowed-book-
response: reg-msg + unreg-msg + borrowed-books-list
friend-reg-data: name + address + landline# + mobile#
own-book-details: friend-reg-data + book-title + data-of-issue
return-own-book-details: book-title + date-of-return
friend-details: name + address + landline# + mobile# + book-list
borrowed-book-details: book-title + borrow-date
serial#: integer

Shortcomings of the DFD model

- a short label may not capture the entire functionality of a bubble
- Not-well defined control aspects are not defined by a DFD
- Decomposition: The method of carrying out decomposition to arrive at the successive levels and the ultimate level to which decomposition is carried out are highly subjective and depend on the choice and judgment of the analyst.
- Improper data flow diagram: T he da ta flow diagramming technique does not provide any specific guidance as to how exactly to decompose a given function into its subfunctions and we have to use subjective judgment to carry out decomposition.

OBJECT MODELLING USING UML



Class Relationships

- Inheritance

  B is a permanent part of A

  A is made up of Bs

  A is a permanent collection of Bs
- Association and link

  B is a part of A

  A contains B

  A is a collection of Bs
- Aggregation and composition

  A is a kind of B

  A is a specialisation of B

  A behaves like B
- Dependency

  A delegates to B

  A needs help from B

  A collaborates with B. Here collaborates with can be any of a large variety of collaborations that are possible among classes such as employs, credits, precedes, succeeds, etc.

Advantages of OOD

- Code and design reuse
- Increased productivity
- Ease of testing and maintenance
- Better code and design understandability enabling development of large programs

Disadvantages of OOD

- The principles of abstraction, data hiding, inheritance, etc. do incur run time overhead due to the additional code that gets generated on account of these features.
- Data is gets scattered across various objects in an object-oriented implementation. Therefore, the spatial locality of data becomes weak and this leads to higher cache miss ratios and consequently to larger memory access times. This finally shows up as increased program run time.

UNIFIED MODELLING LANGUAGE (UML)

- As the name itself implies, UML is a language for documenting models.
- As is the case with any other language, UML has its syntax (a set of basic symbols and sentence formation rules) and semantics (meanings of basic symbols and sentences).
- It provides a set of basic graphical notations (e.g. rectangles, lines, ellipses, etc.) that can be combined in certain ways to document the design and analysis results.

What is a model?

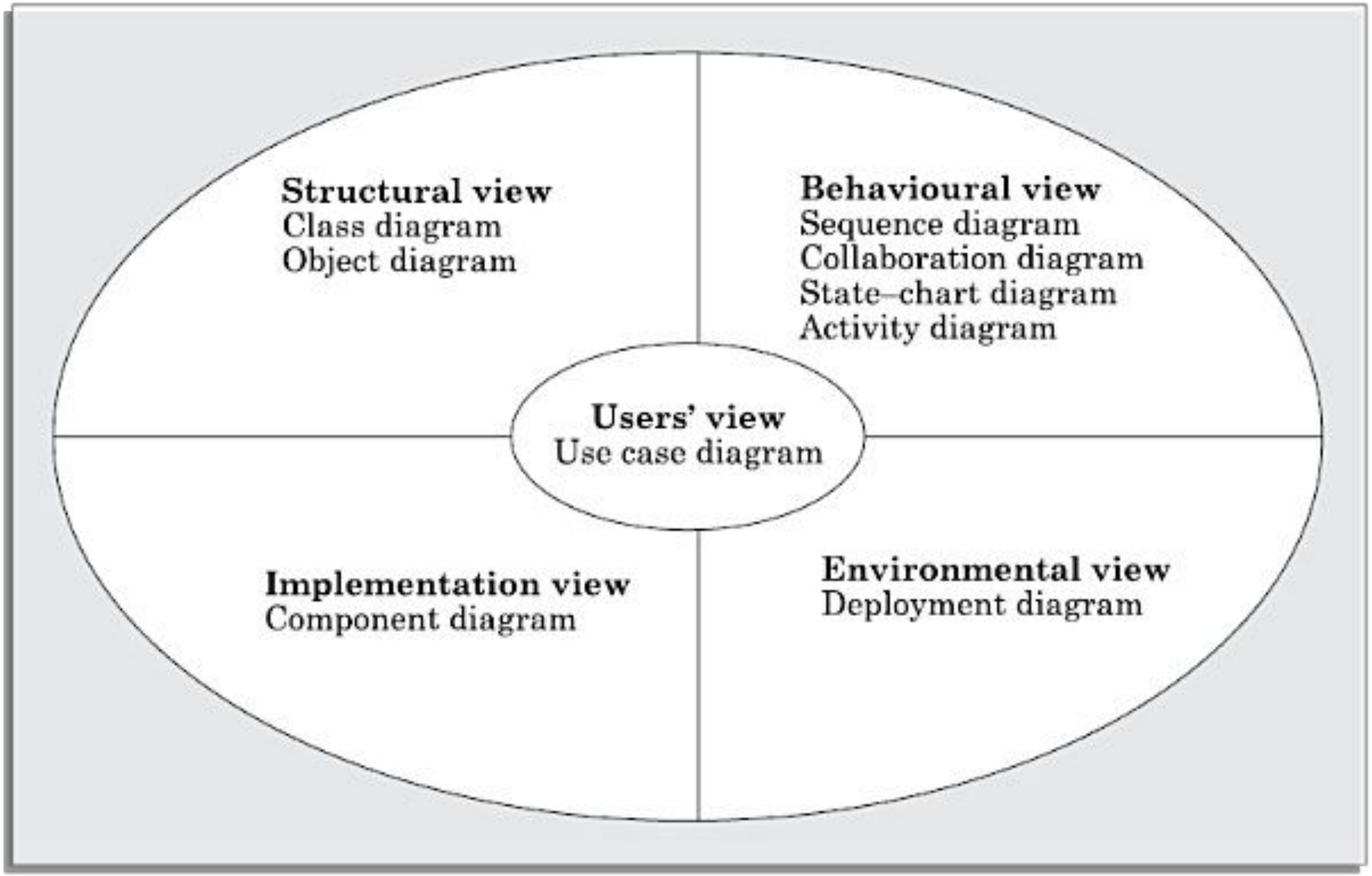
- A model is an abstraction of a real problem (or situation), and is constructed by leaving out unnecessary details.
- This reduces the problem complexity and makes it easy to understand the problem (or situation)

Why construct a model?

- it helps to manage the complexity in a problem and facilitates arriving at good solutions.
- To reduce the design costs.
- The initial model of a problem is called an analysis model.
- The analysis model of a problem can be refined into a design model using a design methodology.
- Models are used for the following purposes:
  - Analysis, Specification, Design, Coding, Visualisation and understanding of an implementation, Testing, etc.

UML DIAGRAMS

- If a single model is made to capture all the required perspectives, then it would be as complex as the original problem, and would be of little use.
- UML diagrams can capture the following views (models) of a system:
  - User’s view
  - Structural view
  - Behaviourial view
  - Implementation view
  - Environmental view


Structural view
Class diagram
Object diagram
Behavioural view
Sequence diagram
Collaboration diagram
State–chart diagram
Activity diagram
Users' view
Use case diagram
Implementation view
Component diagram
Environmental view
Deployment diagram

- Users’ view
  - The users’ view captures the view of the system in terms of the functionalities offered by the system to its users.
- Structural view
  - The structural view defines the structure of the problem (or the solution) in terms of the kinds of objects (classes) important to the understanding of the working of a system and to its implementation. It is also called as static model.
- Behaviourial view
  - The behaviourial view captures how objects interact with each other in time to realise the system behaviour. The system behaviour captures the time-dependent (dynamic) behaviour of the system. It therefore constitutes the dynamic model of the system.
- Implementation view
  - This view captures the important components of the system and their Interdependencies
- Environmental view
  - This view models how the different components are implemented on different pieces of hardware.

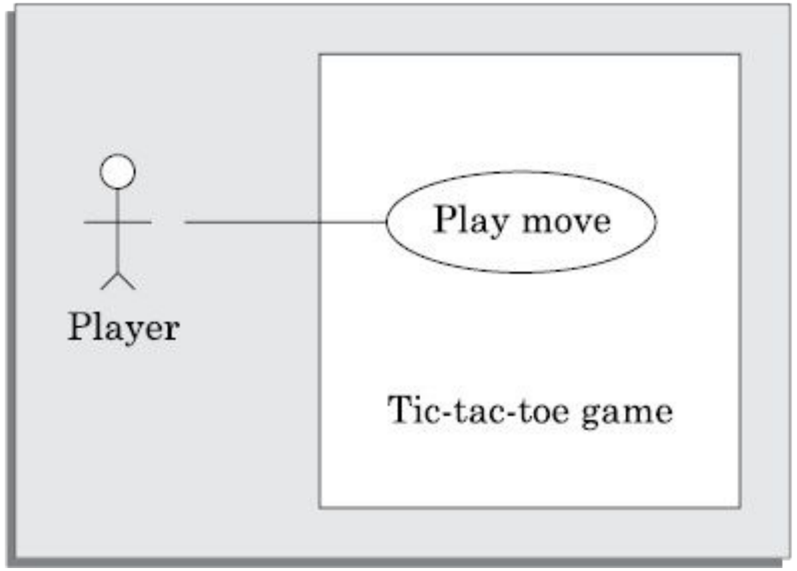
USE CASE MODEL

- The use cases represent the different ways in which a system can be used by the users.

Representation of Use Cases

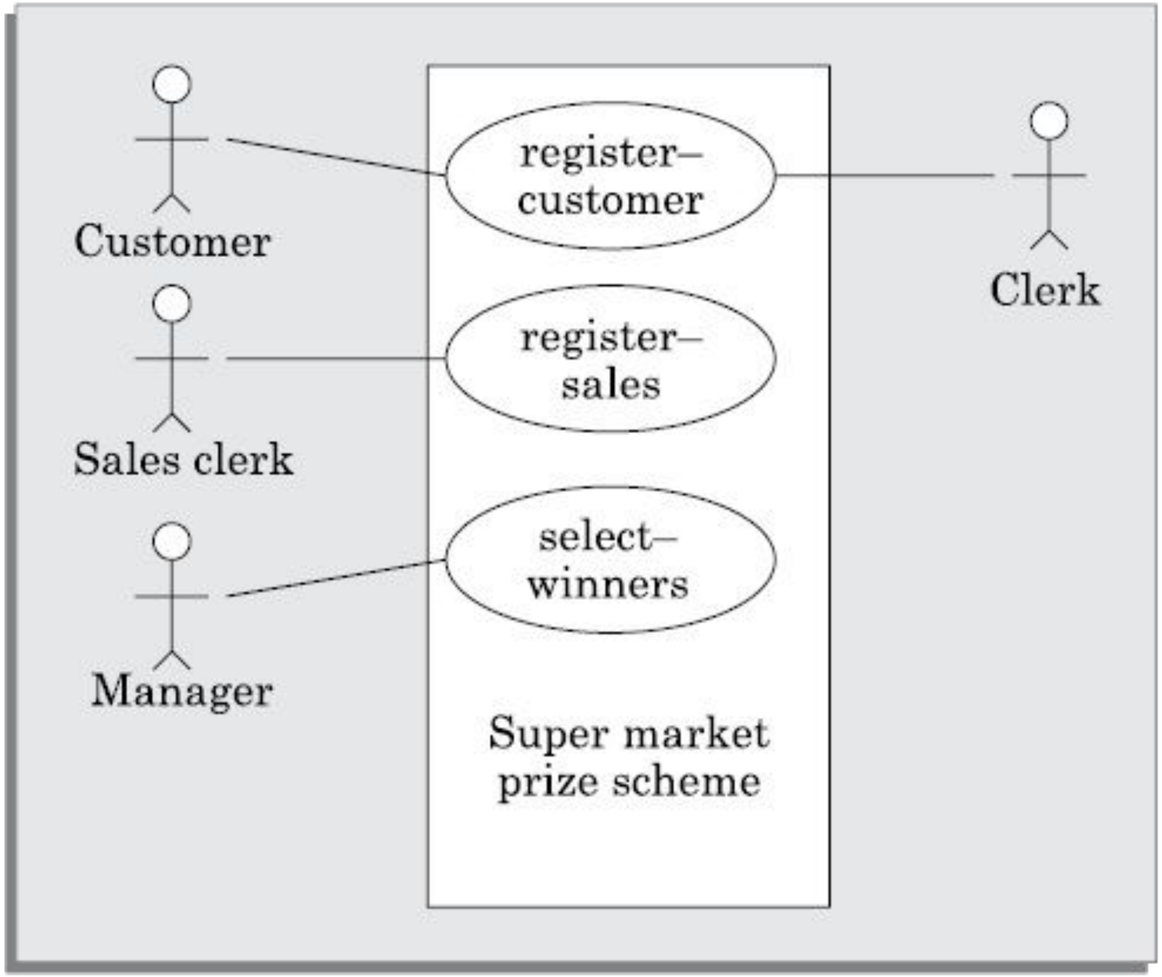
- A use case model can be documented by drawing a use case diagram and writing an accompanying text elaborating the drawing.
- In the use case diagram, each use case is represented by an ellipse with the name of the use case written inside the ellipse.
- All the ellipses (i.e. use cases) of a system are enclosed within a rectangle which represents the system boundary.
- The name of the system being modeled (e.g., library information system) appears inside the rectangle.
- Both human users and external systems can be represented by stick person icons.
- When a stick person icon represents an external system, it is annotated by the stereotype <<external system>>.
- You can draw a rectangle around the use cases, called the system boundary box, to indicates the scope of your system. Anything within the box represents functionality that is in scope and anything outside the box is not.
- However, drawing the system boundary is optional.

- Text description
- Contact persons
- Actors
- Pre-condition
- Post-condition
- Non-functional requirement’s
- Exceptions, error situations
- Sample dialogs
- Specific user interface requirements
- Document references



Exampe

- Supermarket Prize Scheme: A super market needs to develop a software that would help it to automate a scheme that it plans to introduce to encourage regular customers. In this scheme, a customer would have first register by supplying his/her residence address, telephone number, and the driving license number. Each customer who registers for this scheme is assigned a unique customer number (CN) by the computer. A customer can present his CN to the check out staff when he makes any purchase. In this case, the value of his purchase is credited against his CN. At the end of each year, the supermarket intends to award surprise gifts to 10 customers who make the highest total purchase over the year. Also, it intends to award a 22 caret gold coin to every customer whose purchase exceeded Rs. 10,000. The entries against the CN are reset on the last day of every year after the prize winners' lists are generated.


Customer
Sales clerk
Manager
register–customer
register–sales
select–winners
Super market prize scheme
Clerk

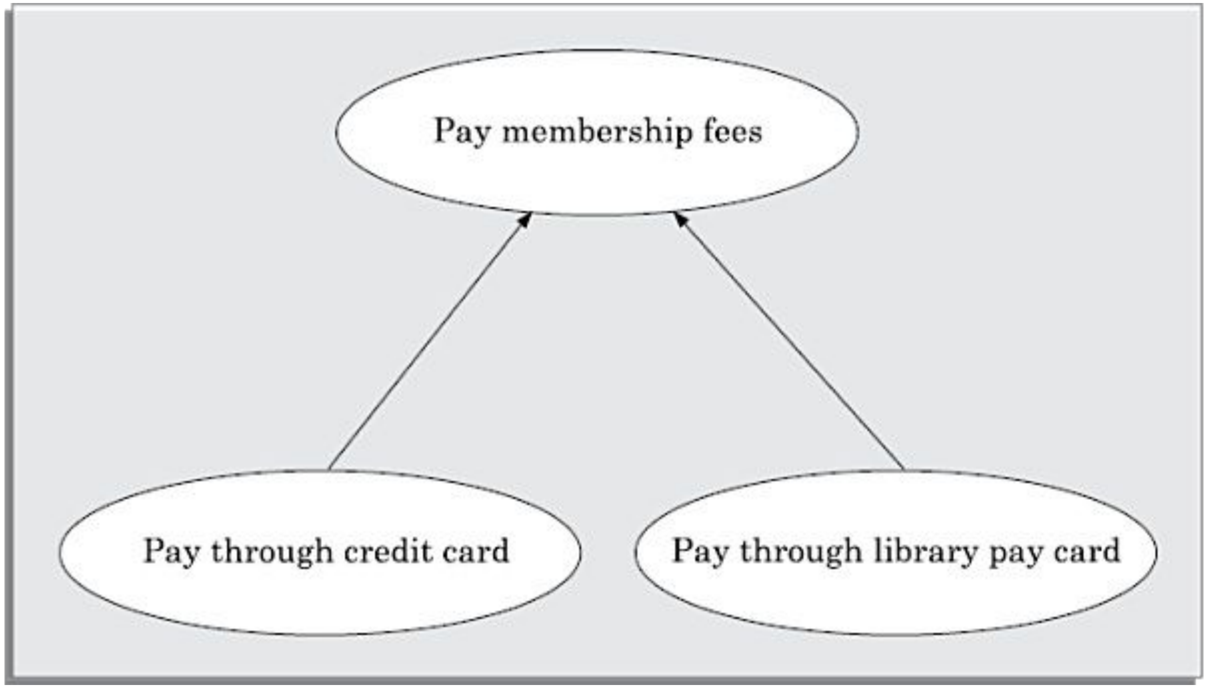
- Text description
  - U1: register-customer: Using this use case, the customer can register himself by providing the necessary details.
- Scenario 1: Mainline sequence
  - 1. Customer: select register customer option
  - 2 . System: display prompt to enter name, address, and telephone number.
  - 3. Customer: enter the necessary values
  - 4 : System: display the generated id and the message that the customer has successfully been registered.
- Scenario 2: At step 4 of mainline sequence
  - 4 : System: displays the message that the customer has already registered.
- Scenario 3: At step 4 of mainline sequence
  - 4 : System: displays message that some input information have not been entered. The system displays a prompt to enter the missing values.
- U2: register-sales: Using this use case, the clerk can register the details of the purchase made by a customer.
- Scenario 1: Mainline sequence
  - 1. Clerk: selects the register sales option.
  - 2. System: displays prompt to enter the purchase details and the id of the customer.
  - 3. Clerk: enters the required details.

- U3: select-winners. Using this use case, the manager can generate the winner list.
- Scenario 2: Mainline sequence
  - 1. Manager: selects the select-winner option.
  - 2 . System: displays the gold coin and the surprise gift winner list.

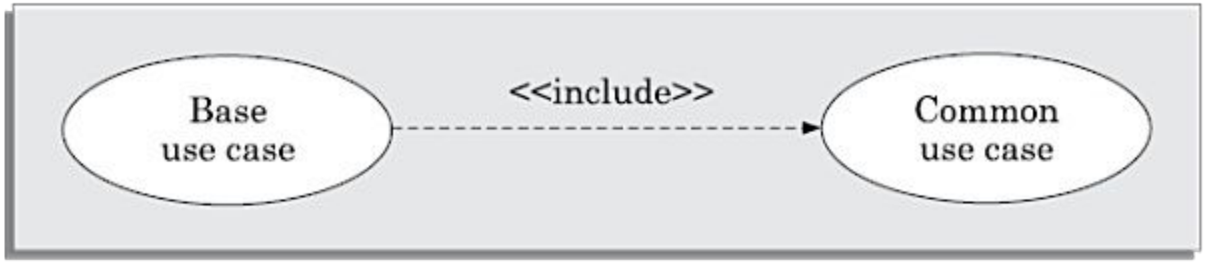
- Why Develop the Use Case Diagram?
- How to Identify the Use Cases of a System?
- Essential Use Case versus Real Use Case

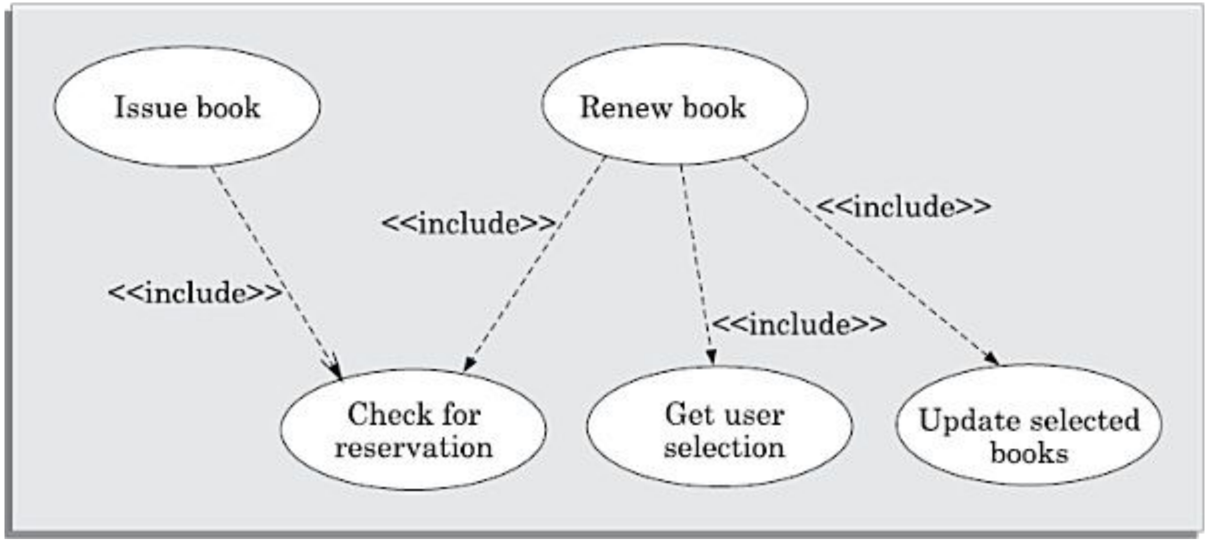
Factoring of Commonality among Use Cases

- Generalisation

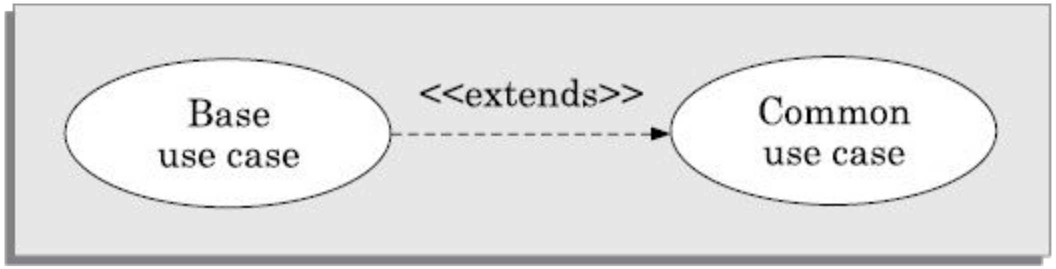


Includes

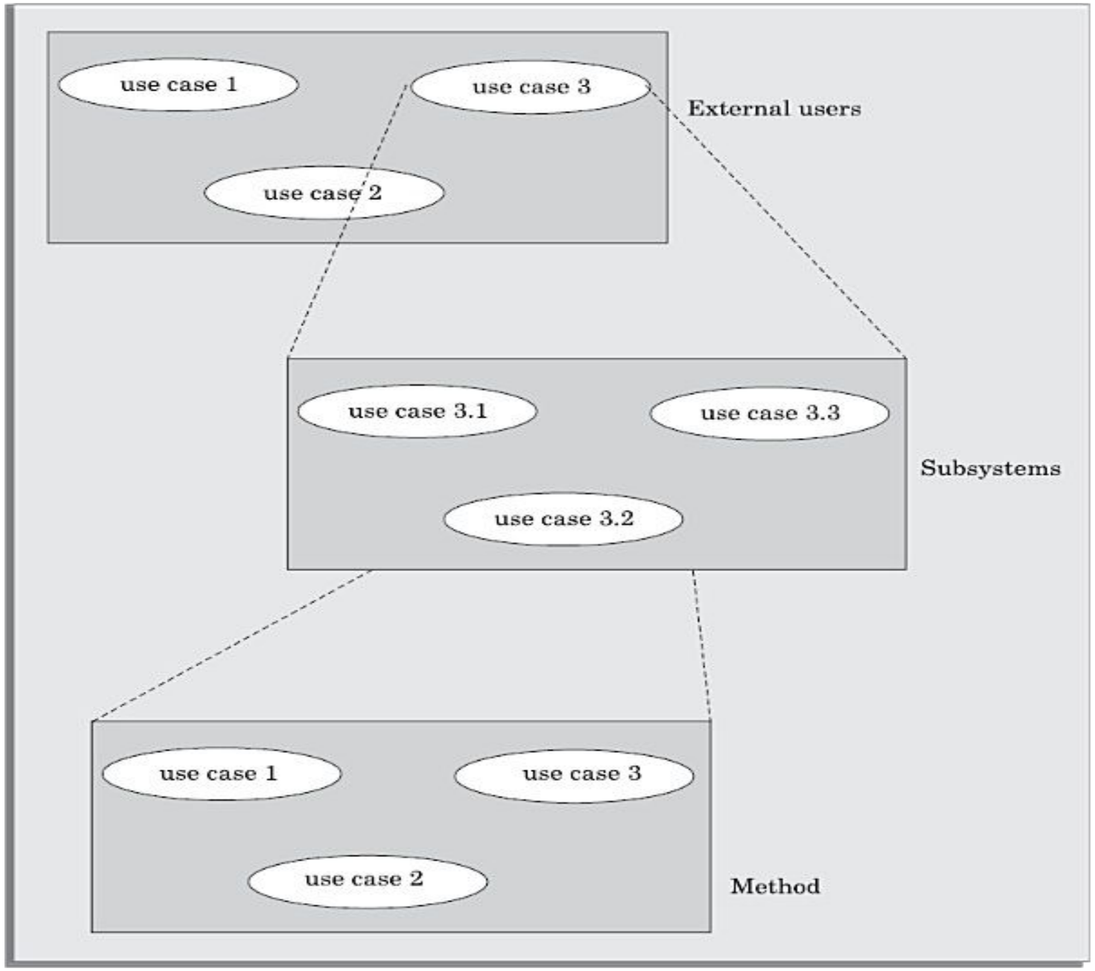

Base use case
<<include>>
Common use case


Issue book
Renew book
<<include>>
<<include>>
<<include>>
<<include>>
<<include>>
Check for reservation
Get user selection
Update selected books

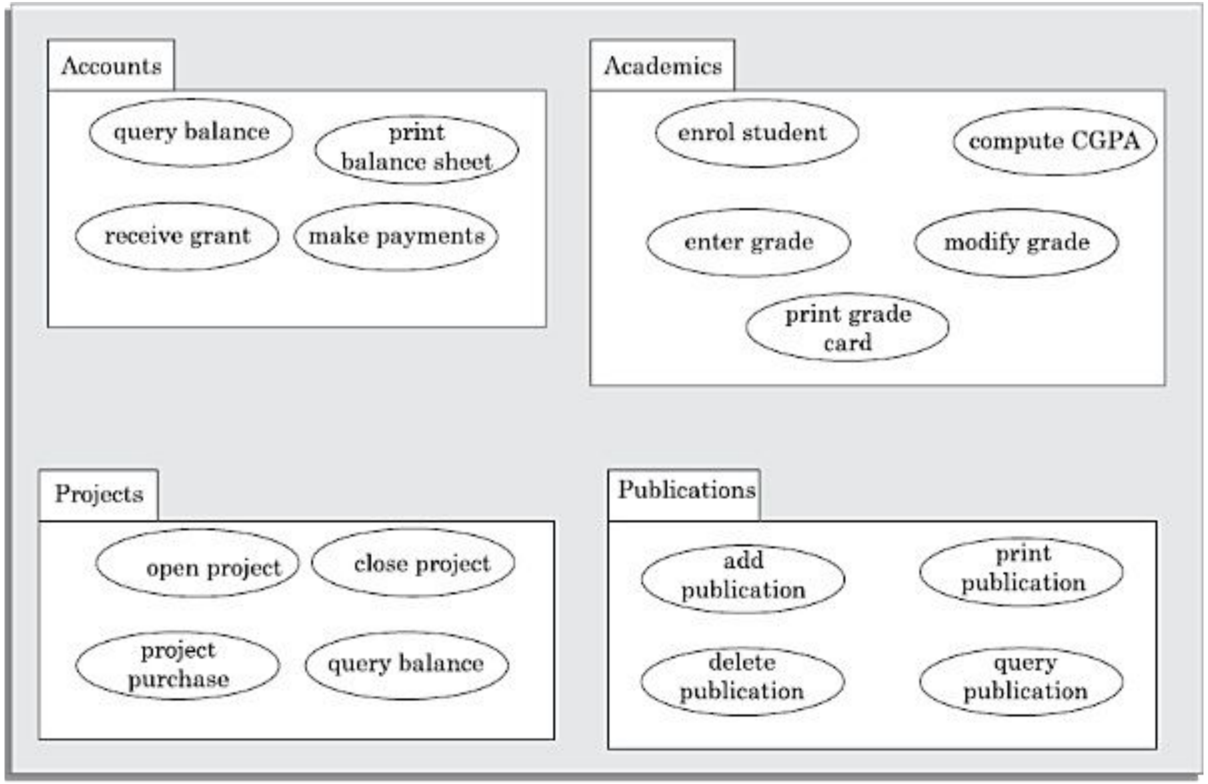
Extends



Organisation



USE CASE PACKAGING



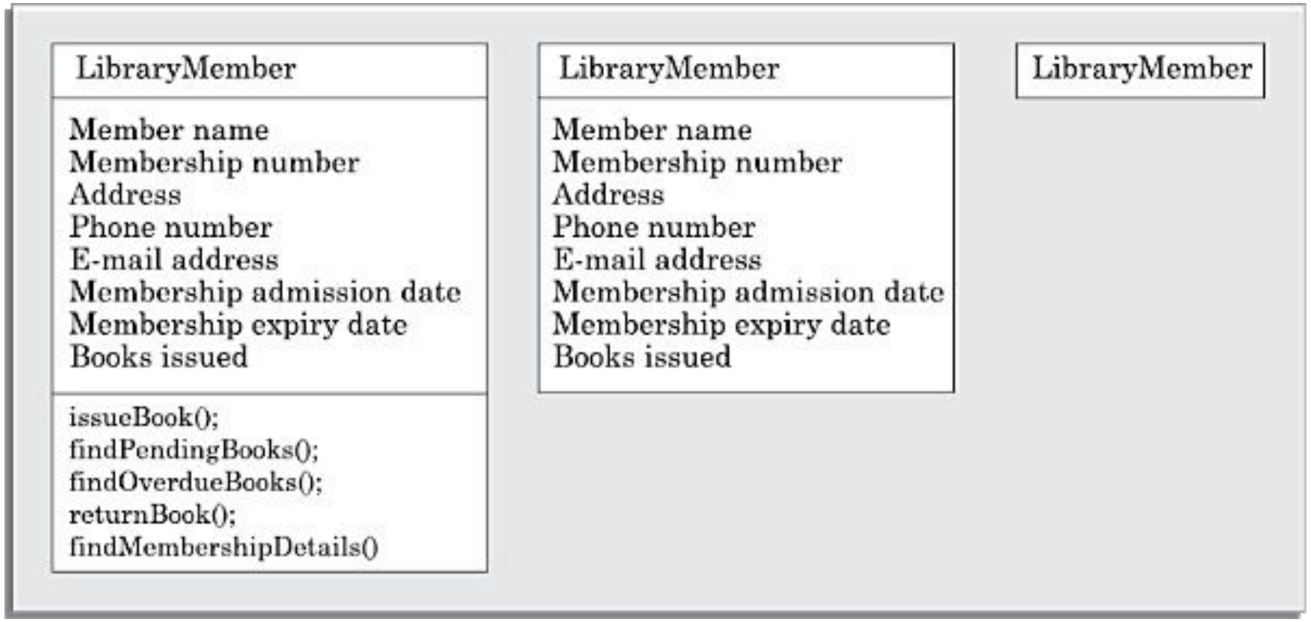
CLASS DIAGRAMS

- A class diagram describes the static structure of a system.
- It shows how a system is structured rather than how it behaves.
- The static structure of a system comprises a number of class diagrams and their dependencies.
- The main constituents of a class diagram are classes and their relationships—generalisation, aggregation, association, and various kinds of dependencies.

Classes

- The classes represent entities with common features, i.e., attributes and operations.
- Classes are represented as solid outline rectangles with compartments.
- Classes have a mandatory name compartment where the name is written centered in boldface.
- The class name is usually written using mixed case convention and begins with an uppercase (e.g. LibraryMember).
- Object names on the other hand, are written using a mixed case convention, but starts with a small case letter (e.g., studentMember).
- Class names are usually chosen to be singularnouns.

- Classes have optional attributes and operations compartments. A class may appear on several diagrams.
- Its attributes and operations are suppressed on all but one diagram.



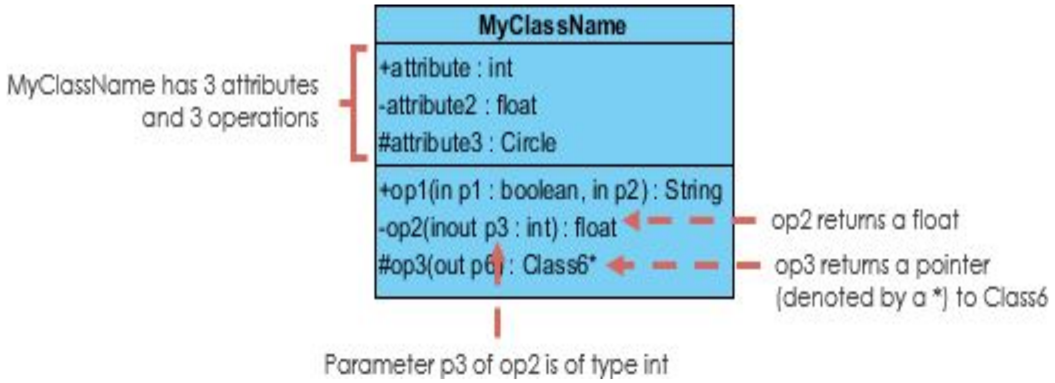
Different representations of the LibraryMember class

Attributes

- An attribute is a named property of a class. It represents the kind of data that an object might contain.
- Attributes are listed with their names, and may optionally contain specification of their type (that is, their class, e.g., Int, Book, Employee, etc.), an initial value, and constraints.
- Attribute names are written left-justified using plain type letters, and the names should begin with a lower case letter.
- Attribute names may be followed by square brackets containing a multiplicity expression, e.g. sensorStatus[10].
- The type of an attribute is written by following the attribute name with a colon and the type name, (e.g., sensorStatus[1]:Int).
- The attribute name may be followed by an initialisation expression e.g. sensorStatus[1]:Int=0

Operations

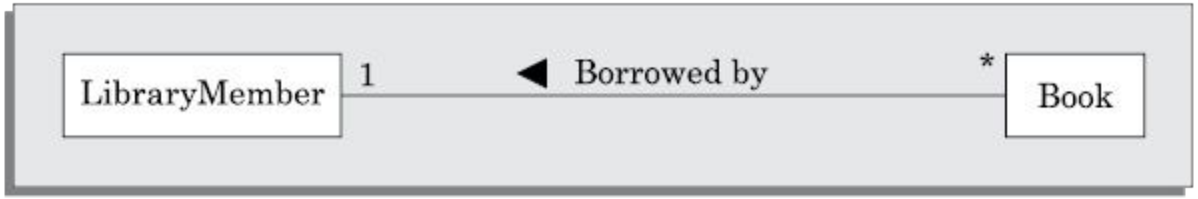
- Operations are shown in the third partition. They are services the class provides.
- The return type of a method is shown after the colon at the end of the method signature.
- The return type of method parameters are shown after the colon following the parameter name.
- Operations map onto class methods in code


MyClassName
+attribute : int
-attribute2 : float
#attribute3 : Circle
+op1(in p1 : boolean, in p2) : String
-op2(inout p3 : int) : float
#op3(out p6) : Class6*
MyClassName has 3 attributes and 3 operations
op2 returns a float
op3 returns a pointer (denoted by a *) to Class6
Parameter p3 of op2 is of type int

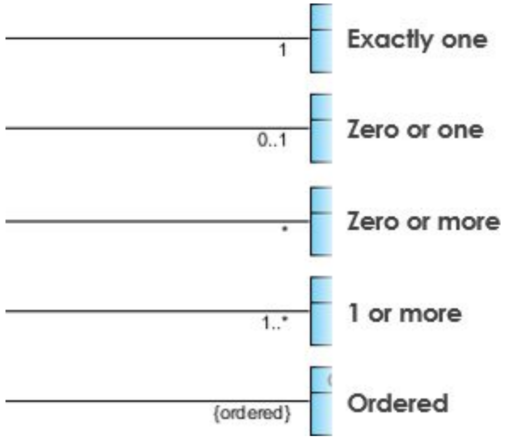
Association

- Associations are relationships between classes in a UML Class Diagram.
- They are represented by a solid line between classes.
- Associations are typically named using a verb or verb phrase which reflects the real world problem domain.

multiplicity/Cordinality

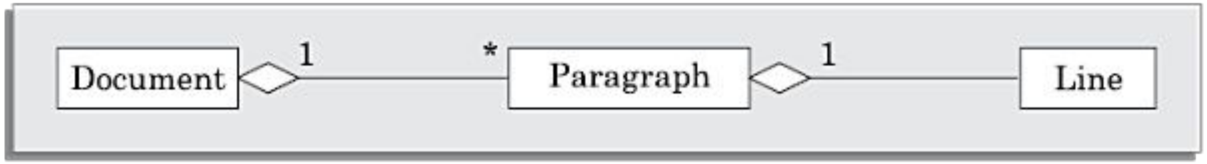


Association between two classes



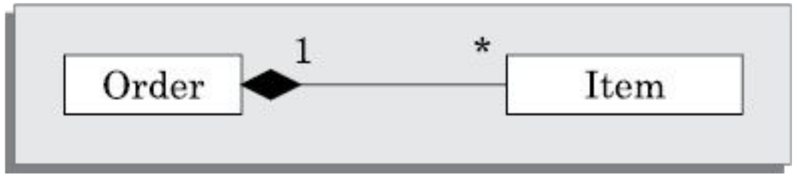
Aggregation

- Aggregation is a special type of association relation where the involved classes are not only associated to each other, but a whole-part relationship exists between them.
- The aggregation relationship cannot be reflexive (i.e. recursive). That is, an object cannot contain objects of the same class as itself.
- Also, the aggregation relation is not symmetric. That is, two classes A and B cannot contain instances of each other.
- However, the aggregation relationship can be transitive. In this case, aggregation may consist of an arbitrary number of levels.



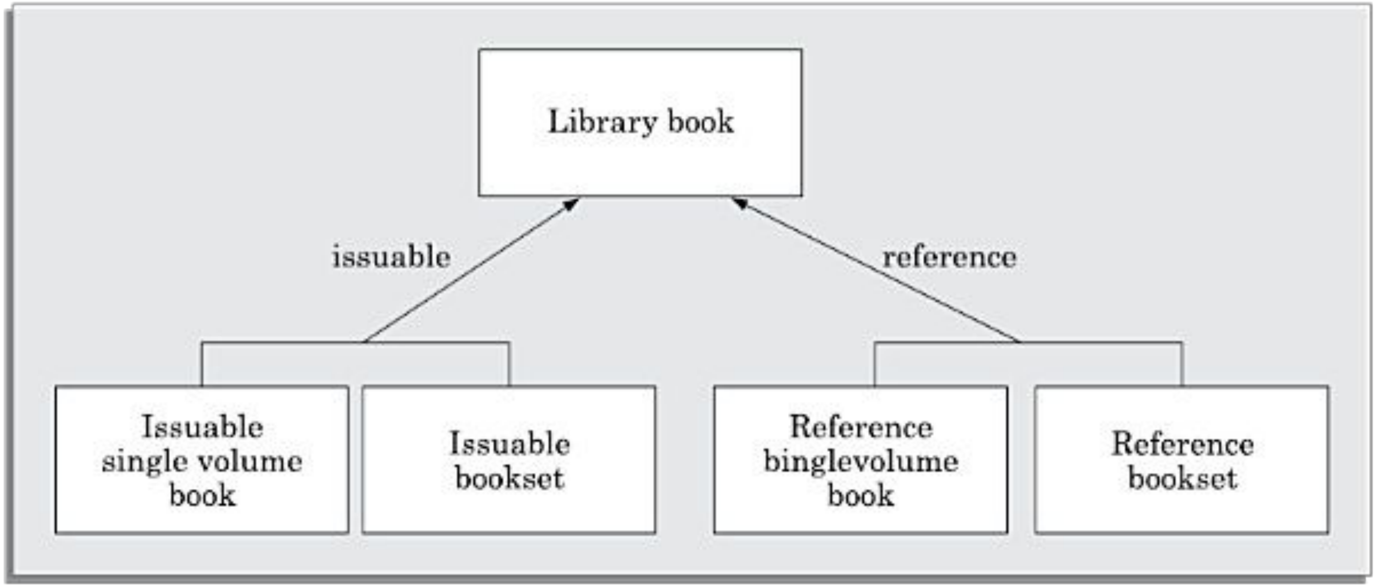
Composition

- Composition is a stricter form of aggregation, in which the parts are existence-dependent on the whole.
- This means that the life of the parts cannot exist outside the whole.
- In other words, the lifeline of the whole and the part are identical.
- When the whole is created, the parts are created and when the whole is destroyed, the parts are destroyed.

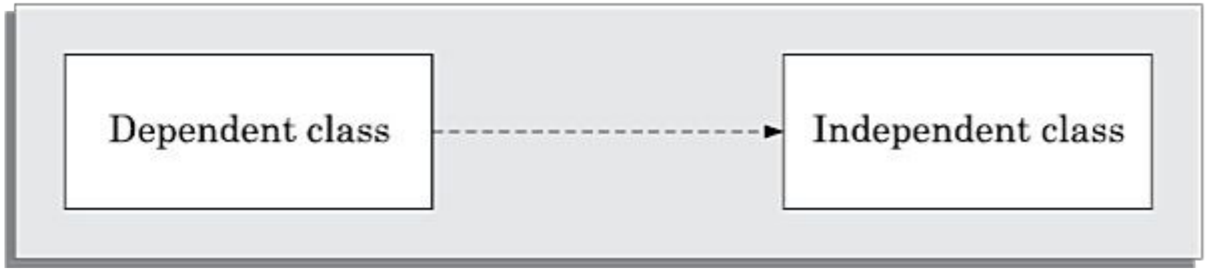


Inheritance

- The inheritance relationship is represented by means of an empty arrow pointing from the subclass to the superclass.



Dependency

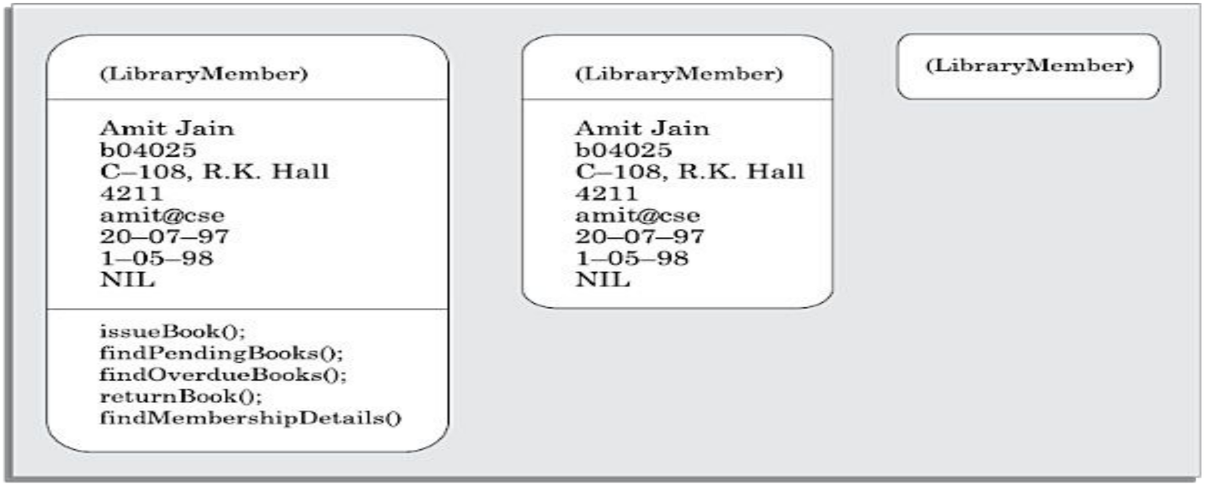


Constraint

A constraint describes a condition or an integrity rule. Constraints are typically used to describe the permissible set of values of an attribute, to specify the pre- and post-conditions for operations, to define certain ordering of items, etc. For example, to denote that the books in a library are sorted on ISBN number we can annotate the book class with the constraint {sorted}

Object diagrams

- Object diagrams shows the snapshot of the objects in a system at a point in time.
- Since it shows instances of classes, rather than the classes themselves, it is often called as an instance diagram.

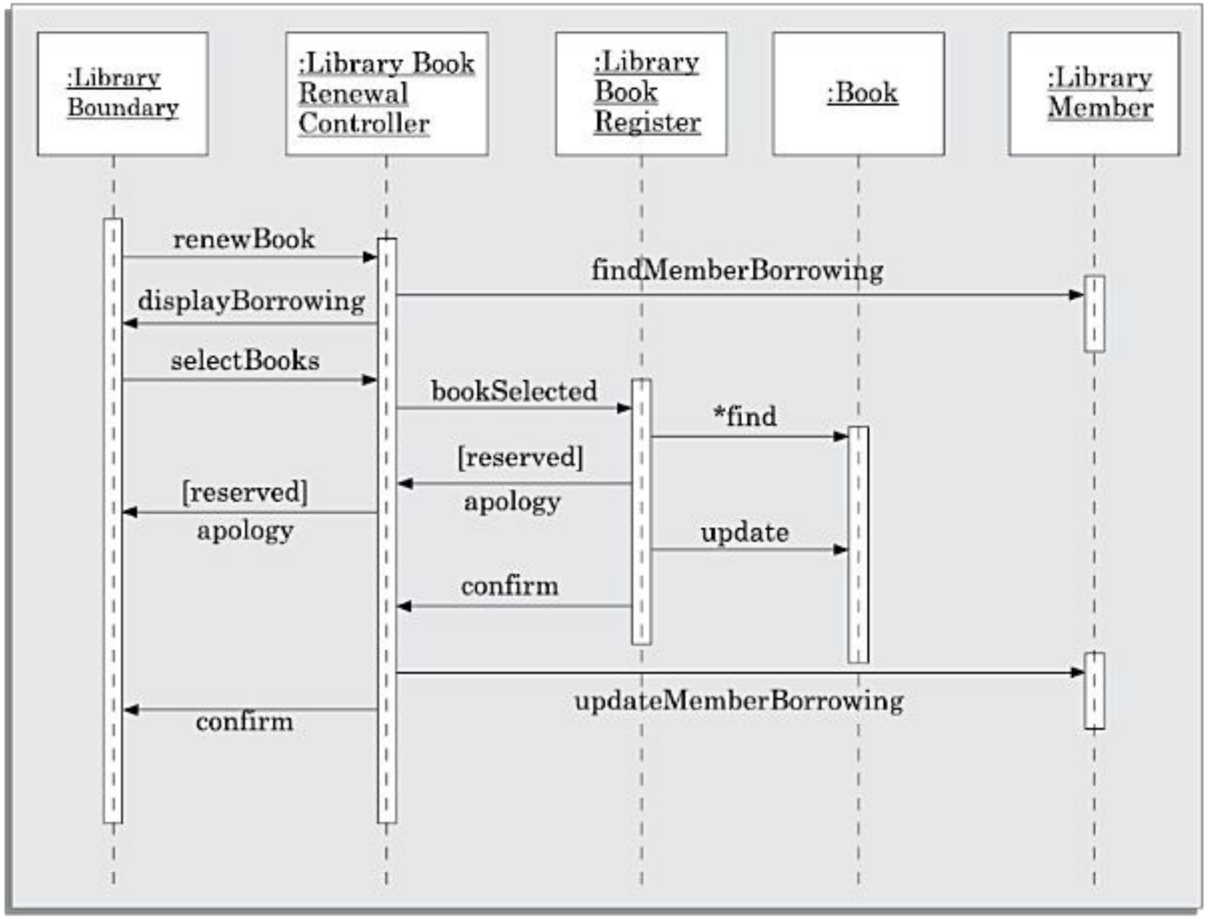


INTERACTION DIAGRAMS

- When a user invokes one of the functions supported by a system, the required behaviour is realised through the interaction of several objects in the system.
- There are two kinds of interaction diagrams—sequence diagrams and collaboration diagrams.
- These two diagrams are equivalent in the sense that any one diagram can be derived automatically from the other.

Sequence diagram

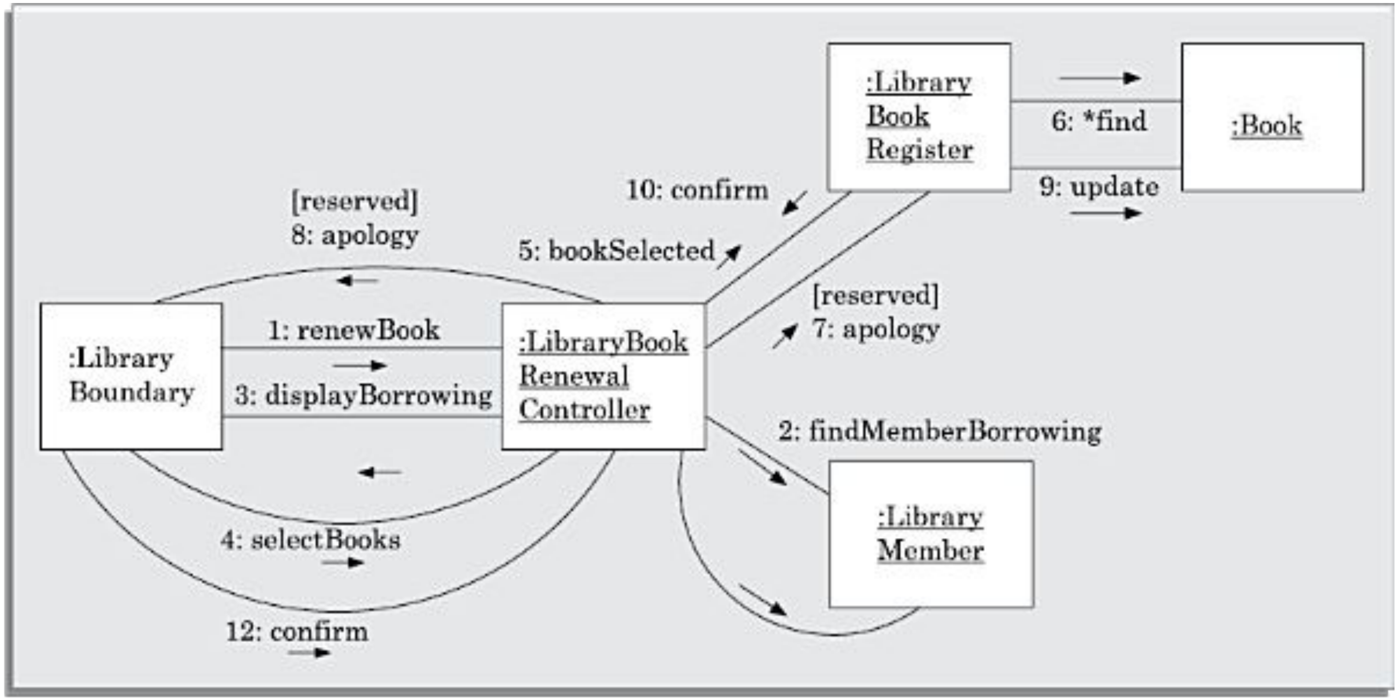
- A sequence diagram shows interaction among objects as a two dimensional chart.
- The chart is read from top to bottom.
- The objects participating in the interaction are shown at the top of the chart as boxes attached to a vertical dashed line.
- Inside the box the name of the object is written with a colon separating it from the name of the class and both the name of the object and the class are underlined.



Sequence diagram for the renew book use case

Collaboration diagram

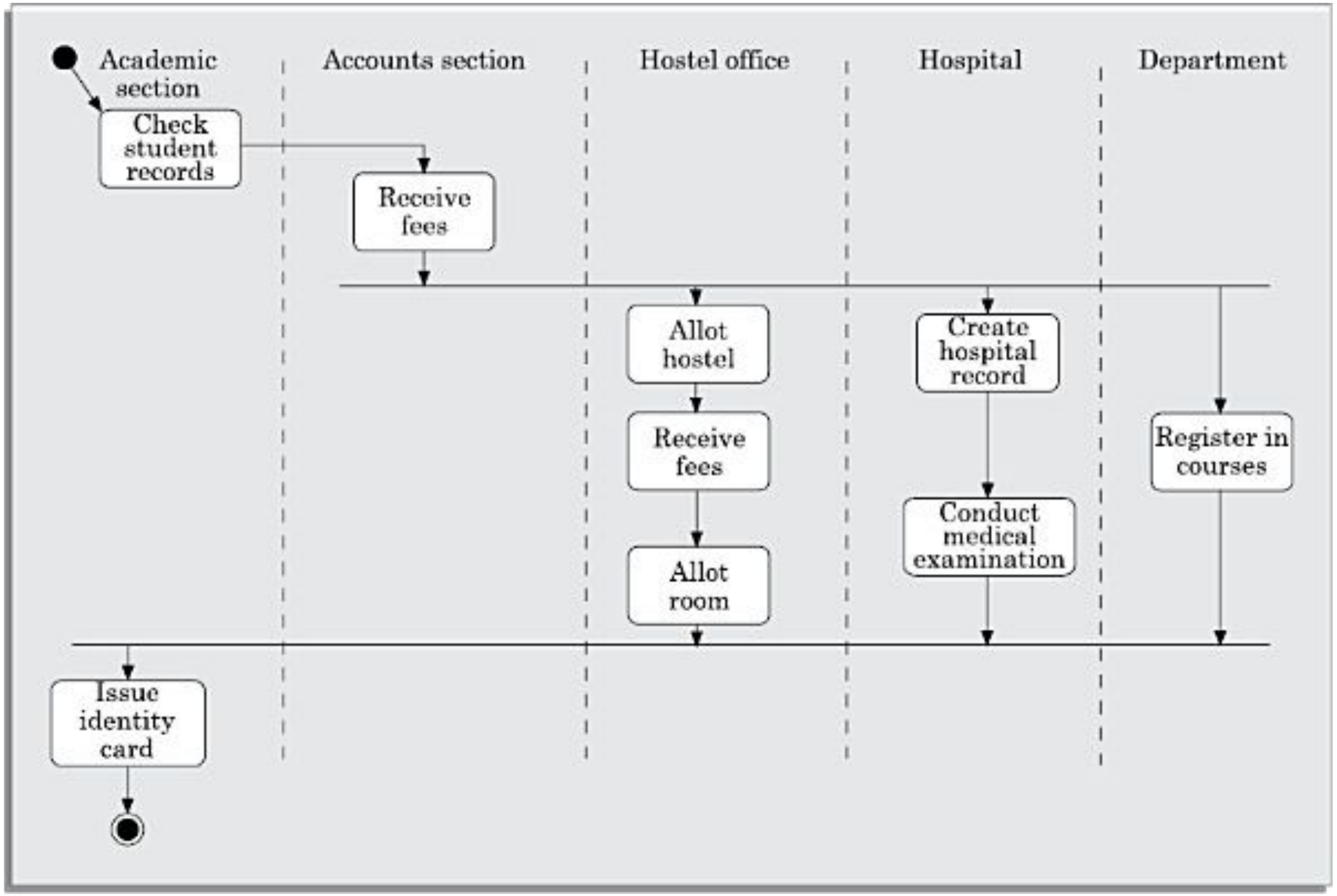
- A collaboration diagram shows both structural and behavioural aspects.
- This is unlike a sequence diagram which shows only the behavioural aspects.
- The structural aspect of a collaboration diagram consists of objects and links among them indicating association.
- In this diagram, each object is also called a collaborator.
- The behavioural aspect is described by the set of messages exchanged among the different collaborators.



Collaboration diagram for the renew book use case.

ACTIVITY DIAGRAM

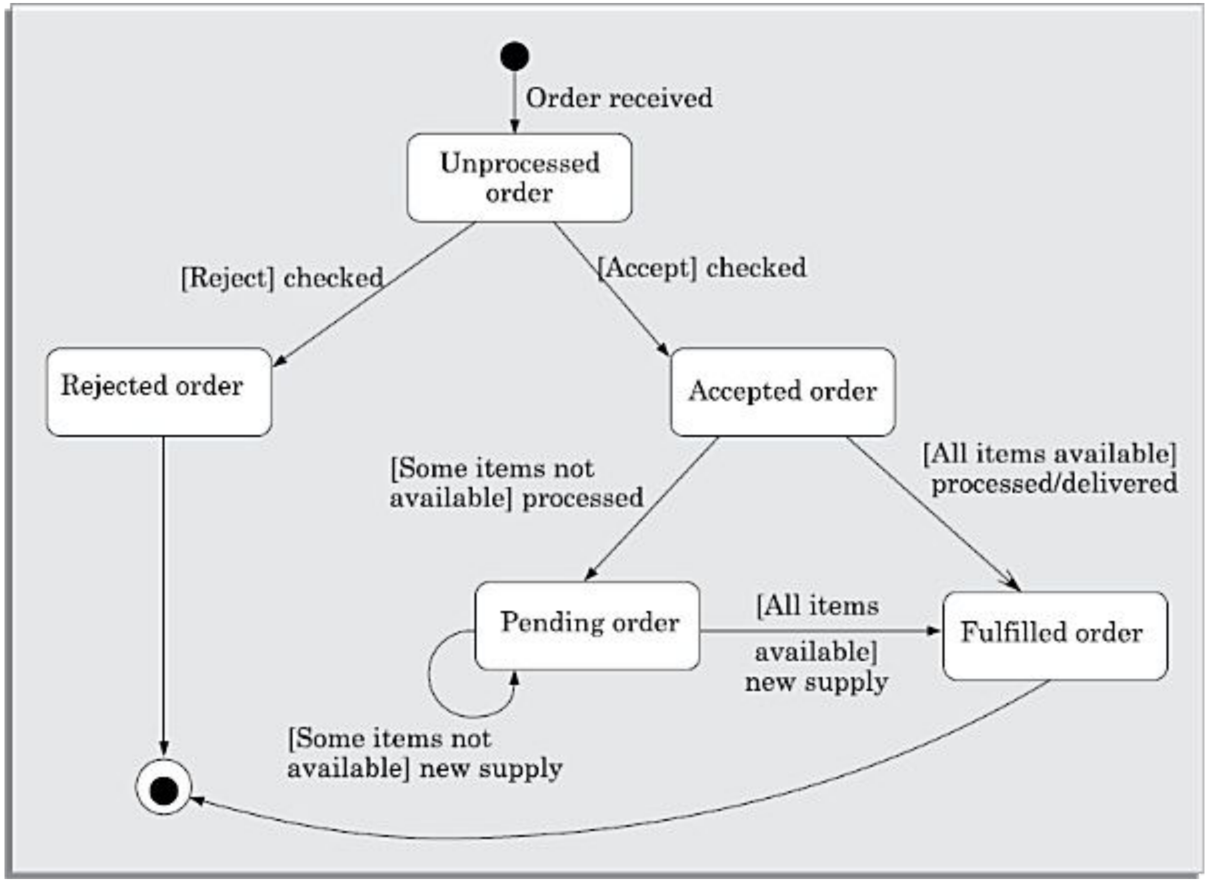
- The activity diagram focuses on representing various activities or chunks of processing and their sequence of activation. The activities in general may not correspond to the methods of classes.
- An activity is a state with an internal action and one or more outgoing transitions which automatically follow the termination of the internal activity.
- If an activity has more than one outgoing transitions, then exact situation under which each is executed must be identified through appropriate conditions.



Activity diagram for student admission procedure

STATE CHART DIAGRAM

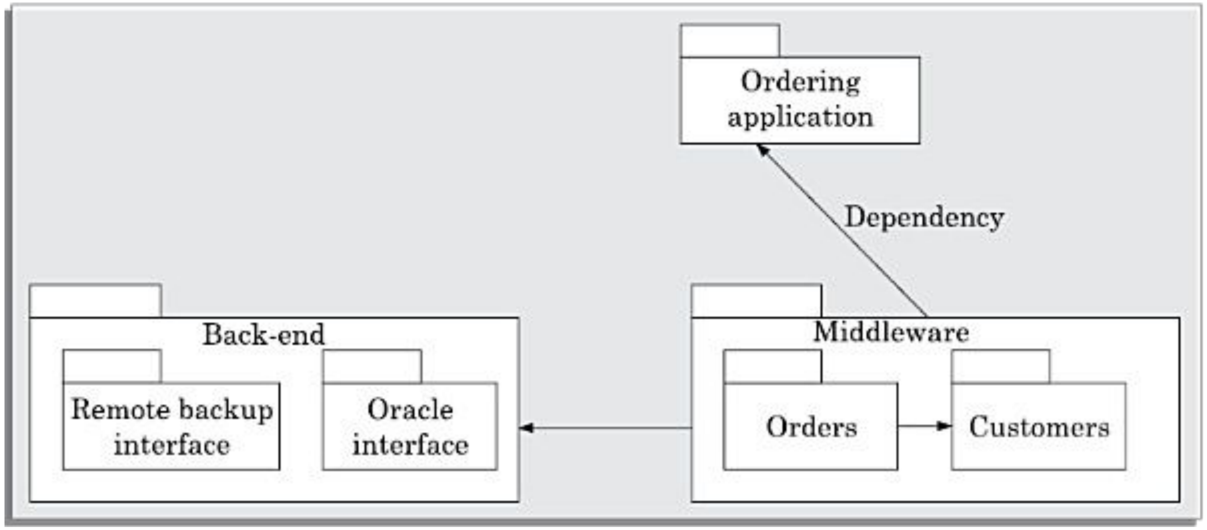
- A state chart diagram is normally used to model how the state of an object changes in its life time.
- State chart diagrams are good at describing how the behaviour of an object changes across several use case executions.
- Basic elements of a state chart:
  - **Initial state:** This represented as a filled circle.
  - **Final state**: This is represented by a filled circle inside a larger circle.
  - **State**: These are represented by rectangles with rounded corners.
  - **Transition**: A transition is shown as an arrow between two states.


Order received
Unprocessed order
[Reject] checked
[Accept] checked
Rejected order
Accepted order
[Some items not available] processed
[All items available] processed/delivered
Pending order
[All items available] new supply
Fulfilled order
[Some items not available] new supply

State chart diagram for an order object.

Package diagram

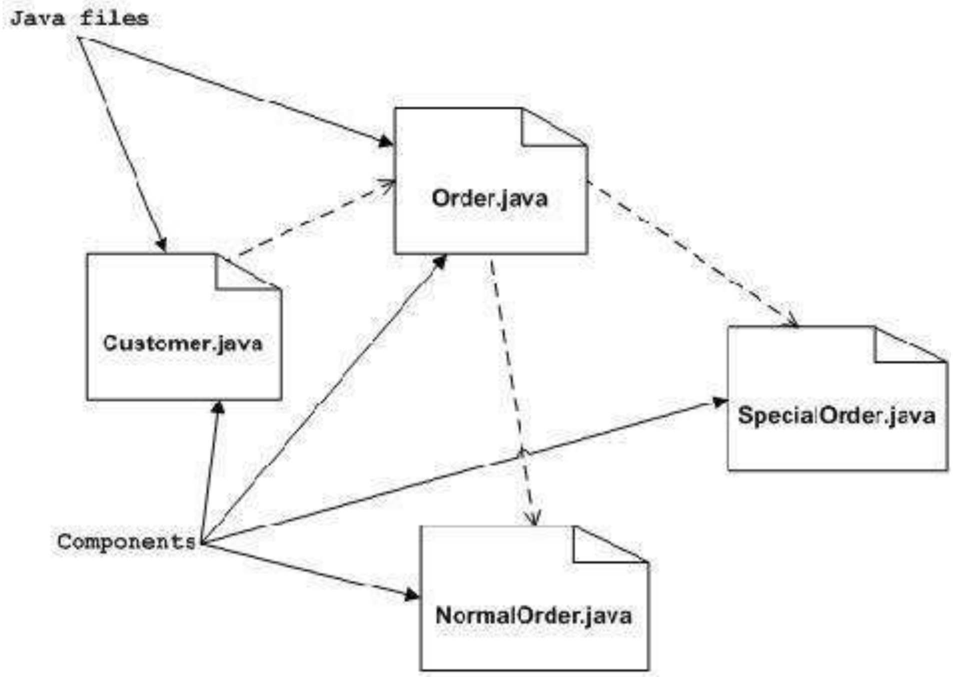
- A package is a grouping of several classes. In fact, a package diagram can be used to group any UML artifacts.
- These are very useful to document organisation of source files for large projects that have a large number ofprogram files.



Component diagram

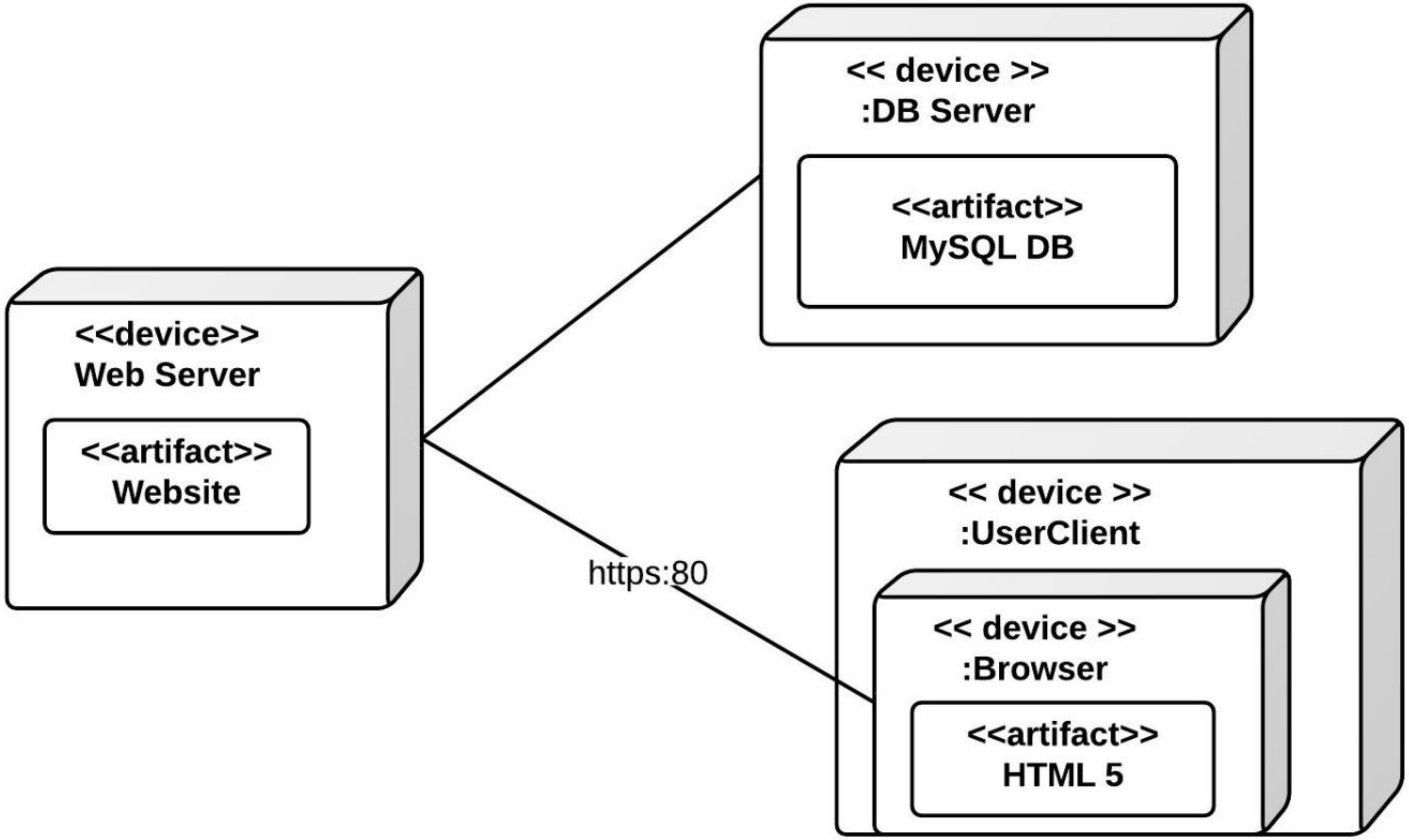
- A component represents a piece of software that can be independently purchased, upgraded, and integrated into an existing software.
- A component diagram can be used to represent the physical structure of an implementation in terms of the various components of the system.
- A component diagram is typically used to achieve the following purposes:
  - Organise source code to be able to construct executable releases.
  - Specify dependencies among different components.
- A package diagram can be used to provide a high-level view of each component in terms the different classes it contains.

Component diagram of an order management system



Deployment diagram

- The deployment diagram shows the environmental view of a system.
- That is, it captures the environment in which the software solution is implemented.
- In other words, a deployment diagram shows how a software system will be physically deployed in the hardware environment.
- That is, which component will execute on which hardware component and how they will they communicate with each other.
- Since the diagram models the run time architecture of an application, this diagram can be very useful to the system’s operation staff.


<< device >>
:DB Server
<<artifact>>
MySQL DB
<<device>>
Web Server
<<artifact>>
Website
https:80
<< device >>
:UserClient
<< device >>
:Browser
<<artifact>>
HTML 5

CODING AND TESTING

- Coding is undertaken once the design phase is complete and the design documents have been successfully reviewed.
- After all the modules of a system have been coded and unit tested, the integration and system testing phase is undertaken.

CODING

- The objective of the coding phase is to transform the design of a system into code in a high-level language, and then to unit test this code.

Coding Standards and Guidelines

- Good software development organisations usually develop their own coding standards and guidelines depending on what suits their organisation best and based on the specific types of software they develop.

Representative coding standards

- Rules for limiting the use of globals:
- Standard headers for different modules the following is an example of header format that is being used in some companies:
  - Name of the module.
  - Date on which the module was created.
  - Author’s name.
  - Modification history.
  - Synopsis of the module. This is a small writeup about what the module does.
  - Different functions supported in the module, along with their input/output parameters.
  - Global variables accessed/modified by the module.
- Naming conventions for global variables, local variables, and constant identifiers:
- Conventions regarding error return values and exception handling mechanisms

Representative coding guidelines

- Do not use a coding style that is too clever or too difficult to understand
- Avoid obscure side effects
- Do not use an identifier for multiple purposes
- Code should be well-documented
- Length of any function should not exceed 10 source lines
- Do not use GO TO statements:

CODE REVIEW

- Testing is an effective defect removal mechanism. However, testing is applicable to only executable code.
- Review is a very effective technique to remove defects from source code.
- In fact, review has been acknowledged to be more cost-effective in removing defects as compared to testing.
- two types of reviews are carried out on the code of a module:
  - Code inspection.
  - Code walkthrough.

Code Walkthrough:

- Code walkthrough is an informal code analysis technique.
- The main objective of code walkthrough is to discover the algorithmic and logical errors in the code.
- Guidelines for code walktrough:
  - The team performing code walkthrough should not be either too big or too small. Ideally, it should consist of between three to seven members.
  - Discussions should focus on discovery of errors and avoid deliberations on how to fix the discovered errors.
  - In order to foster co-operation and to avoid the feeling among the engineers that they are being watched and evaluated in the code walkthrough meetings, managers should not attend the walkthroughMeetings.

Code Inspection

- The principal aim of code inspection is to check for the presence of some common types of errors that usually creep into code due to programmer mistakes and oversights and to check whether coding standards have been adhered to.
- Following is a list of some classical programming errors which can be checked during code inspection:
  - Use of uninitialised variables.
  - Jumps into loops.
  - Non-terminating loops.
  - Incompatible assignments.
  - Array indices out of bounds.
  - Improper storage allocation and deallocation.
  - Mismatch between actual and formal parameter in procedure calls.
  - Use of incorrect logical operators or incorrect precedence among operators.
  - Improper modification of loop variables.
  - Comparison of equality of floating point values.
  - Dangling reference caused when the referenced memory has not been allocated

Clean Room Testing

- This type of testing relies heavily on walkthroughs, inspection, and formal verification.
- The programmers are not allowed to test any of their code by executing the code other than doing some syntax testing using a compiler.
- It is interesting to note that the term cleanroom was first coined at IBM by drawing analogy to the semiconductor fabrication units where defects are avoided by manufacturing in an ultra-clean atmosphere.
- The main problem with this approach is that testing effort is increased as walkthroughs, inspection, and verification are time consuming for detecting all simple errors.

SOFTWARE DOCUMENTATION

- When a software is developed, in addition to the executable files and the source code, several kinds of documents such as users' manual, software requirements specification (SRS) document, design document, test document, installation manual, etc., are dev eloped as part of the software engineering process.
- Good documents
- are helpful in the following ways:
  - enhances understandability of code , in turn reduces the effort and time required for maintenance.
  - help the users to understand and effectively use the system.
  - effectively tackle the manpower turnover problem.
  - helps the manager to effectively track the progress of the project.
- Different types of software documents can broadly be classified into the following:
  - Internal documentation: These are provided in the source code itself.
  - External documentation: These are the supporting documents such as SRS document, installation document, user manual, design document, and test document.

Internal Documentation

- Internal documentation is the code comprehension features provided in the source code itself. Internal do cumentation can be provided in the code in several forms.
- The important types of internal documentation are the following:
  - ➔ Comments embedded in the source code.
  - ➔ Use of meaningful variable names.
  - ➔ Module and function headers.
  - ➔ Code indentation.
  - ➔ Code structuring (i.e., code decomposed into modules and functions).
  - ➔ Use of enumerated types.
  - ➔ Use of constant identifiers.
  - ➔ Use of user-defined data types.

External Documentation

- External documentation is provided through various types of supporting documents such as users' manual, software requirements specification document, design document, test document, etc.
- A systematic software development style ensures that all these documents are of good quality and are produced in an orderly fashion.

Gunning's fog index

Gunning's fog index (developed by Robert Gunning in 1952) is a
metric
that has been designed to measure the readability of a document.

$$\text{fog}(D) = 0.4 \times \left(\frac{\text{words}}{\text{sentences}}\right) + \text{per cent of words having 3 or more syllables}$$

DEBUGGING

- After a failure has been detected, it is necessary to first identify the program statement(s) that are in error and are responsible for the failure, the error can then be fixed.
- Debugging Approaches:
  - Brute force method
  - Backtracking
  - Cause elimination method
  - Program slicing

Brute force method

- This is the least efficient method. In this approach, print statements are inserted throughout the program to print the intermediate values with the hope that some of the printed values will help to identify the statement in error.
- This approach becomes more systematic with the use of a symbolic debugger (also called a source code debugger), because values of different variables can be easily checked and break points and watch points can be easily set to test the values of variables effortlessly.
- Single stepping using a symbolic debugger is another form of this approach, where the developer mentally computes the expected result after every source instruction and checks whether the same is computed by single stepping through the program.

Backtracking

- In this approach, starting from the statement at which an error symptom has been observed, the source code is traced backwards until the error is discovered.
- Unfortunately, as the number of source lines to be traced back increases, the number of potential backward paths increases and may become unmanageably large for complex programs, limiting the use of this approach.

Cause elimination method

- In this approach, once a failure is observed, the symptoms of the failure (i.e., certain variable is having a negative value though it should be positive, etc.) are noted.
- Based on the failure symptoms, the causes which could possibly have contributed to the symptom is developed and tests are conducted to eliminate each.
- A related technique of identification of the error from the error symptom is the software fault tree analysis.

Program slicing

- This technique is similar to back tracking.
- In the backtracking approach, one often has to examine a large number of statements. However, the search space is reduced by defining slices.
- A slice of a program for a particular variable and at a particular statement is the set of source lines preceding this statement that can influence the value of that variable.
- Program slicing makes use of the fact that an error in the value of a variable can be caused by the statements on which it is data dependent.

Testing

- The aim of program testing is to identify all defects in a program.
- However, in practice, even after satisfactory completion of the testing phase, it is not possible to guarantee that a program is error free.
- This is because the input data domain of most programs is very large, and it is not practical to test the program exhaustively with respect to each value that the input can assume.

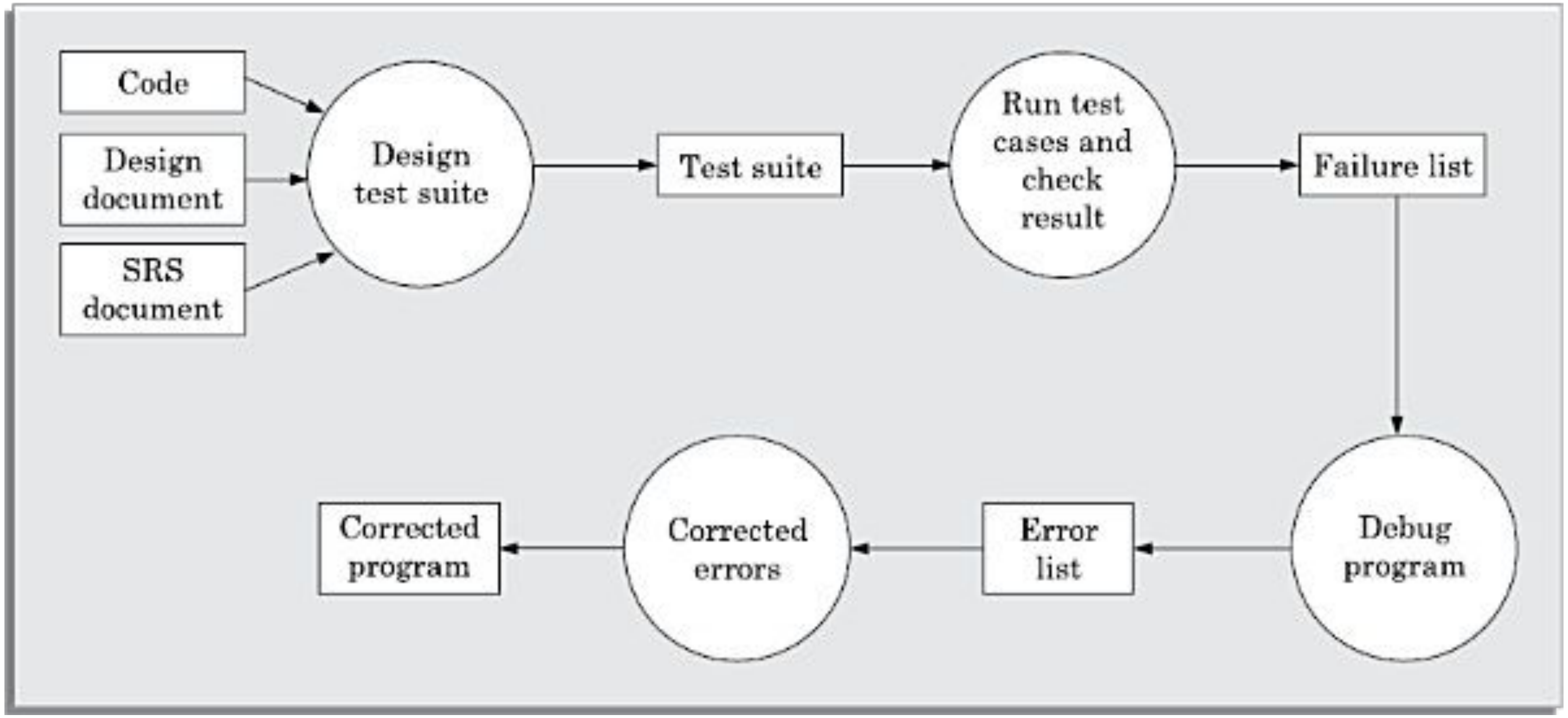
Terminologies

- A **mistake** is essentially any programmer action that later shows up as an incorrect result during program execution.
- An **error** is the result of a mistake committed by a developer in any of the development activities.
- A **failure** of a program essentially denotes an incorrect behaviour exhibited by the program during its execution.
- A **test case** is a triplet [I , S, R], where I is the data input to the program under test, S is the state of the program at which the data is to be input, and R is the result expected to be produced by the program.
- A **test scenario** is an abstract test case in the sense that it only identifies the aspects of the program that are to be tested without identifying the input, state, or output. A test case can be said to be an implementation of a test scenario.

- A **test script** is an encoding of a test case as a short program. Test scripts are developed for automated execution of the test cases.
- A test case is said to be a **positive test case** if it is designed to test whether the software correctly performs a required functionality.
- A test case is said to be **negative test case**, if it is designed to test whether the software carries out something, that is not required of the system.
- A **test suite** is the set of all test that have been designed by a tester to test a given program.
- **Testability** of a requirement denotes the extent to which it is possible to determine whether an implementation of the requirement conforms to it in both functionality and performance.

A **failure mode** of a software denotes an observable way in which it can fail. In other words, all failures that have similar observable symptoms, constitute a failure mode. As an example of the failure modes of a software, consider a railway ticket booking software that has three failure modes—failing to book an available seat, incorrect seat booking (e.g., booking an already booked seat), and system crash.

Equivalent faults denote two or more bugs that result in the system failing in the same failure mode. As an example of equivalent faults, consider the following two faults in C language—division by zero and illegal memory access errors. These two are equivalent faults, since each of these leads to a program crash.



Testing process.

Test Case Design

- There are essentially two main approaches to systematically design test cases:
  - Black-box approach
  - White-box (or glass-box) approach
- In the black-box approach, test cases are designed using only the functional specification of the software.
- That is, test cases are designed solely based on an analysis of the input/out behaviour (that is, functional behaviour) and does not require any knowledge of the internal structure of a program.
- For this reason, black-box testing is also known as functional testing.
- On the other hand, designing white-box test cases requires a thorough knowledge of
- the internal structure of a program, and therefore white-box testing is also called structural testing.
- Black- box test cases are designed solely based on the input-output behaviour of a program. In contrast, white-box test cases are based on an analysis of the code.
- a program has to be tested using the test cases designed by both the approaches, and one testing using one approach does not substitute testing using the other.

- Exhaustive testing of almost any non-trivial system is impractical due to the fact that the domain of input data values to most practical software systems is either extremely large or infinite.
- Therefore, we must design an optional test suite that is of reasonable size and can uncover as many errors existing in the system as possible.
- But larger test suite does not always detects more error. For example lets consider the following code:

  -
    ```
    if (x>y)
        max = x;
    else
        max = x;
    ```

- For the above code segment, consider the following test suites
  - Test suite 1: **{ (x=3,y=2); (x=2,y=3) }**
  - Test suite 2: **{ (x=3,y=2); (x=4,y=3); (x=5,y=1) }**
- Test suite 1 can detect the error.
- Test suite 2 can not detect the error despite of being large.
- Therefore, systematic approaches should be followed to design an optimal test suite. In an optimal test suite, each test case is designed to detect different errors.

BLACK-BOX TESTING

- In black-box testing, test cases are designed from an examination of the input/output values only and no knowledge of design or code is required.
- The following are the two main approaches available to design black box test cases:
  - Equivalence class partitioning
  - Boundary value analysis

Equivalence Class Partitioning

In the equivalence class partitioning approach, the domain of input values to the program under test is partitioned into a set of equivalence classes. The partitioning is done such that for every input data belonging to the same equivalence class, the program behaves similarly.

The following are two general guidelines for designing the equivalence classes:

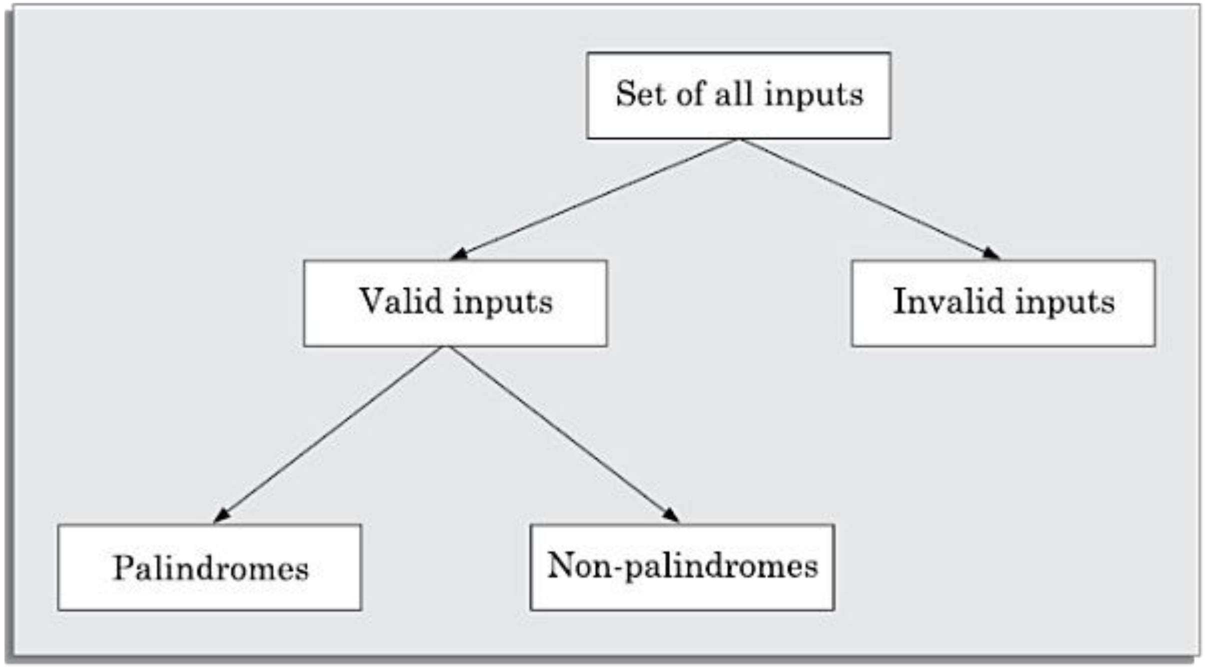
1. If the input data values to a system can be specified by a range of values, then one valid and two invalid equivalence classes need to be defined. For example, if the equivalence class is the set of integers in the range 1 to 10 (i.e., [1,10]), then the invalid equivalence classes are $[-\infty,0]$, $[11,+\infty]$.
2. If the input data assumes values from a set of discrete members of some domain, then one equivalence class for the valid input values and another equivalence class for the invalid input values should be defined. For example, if the valid equivalence classes are {A,B,C}, then the invalid equivalence class is U-{A,B,C}, where U is the universe of possible input values.

Example 1: A software can compute the square root of an input integer which can assume values in the range of 0 to 5000. Design 3 Equivalence Class Partitioning test cases.
Answer: There are three equivalence classes:

- The set of negative integers.
- the set of integers in the range of 0 and 5000 and
- the integers larger than 5000.
- Therefore, the test cases must include representatives for each of the three equivalence classes and possible 3 test sets can be:
- Test suite 1: {-5,500,6000}, Test suite 2: {-15,900,6020} and Test suite 3: {-3,4999,5100}

- **Example 2:** Design the black-box test suite for the following program. The program computes the intersection point of two straight lines and displays the result. It reads two integer pairs (m1, c1) and (m2, c2) defining the two straight lines of the form y=mx + c.
- **Answer :** The equivalence classes are the following:
  - Parallel lines (m1=m2, c1≠c2)
  - Intersecting lines (m1≠m2)
  - Coincident lines (m1=m2, c1=c2)
  - Now, selecting one representative value from each equivalence class, the test suit {(2, 2) (2, 5)}, {(5, 5) (7, 7)}, {(10, 10) (10, 10)} are obtained.

Example: Design equivalence class partitioning test suite for a function that reads a character string of size less than five characters and displays whether it is a palindrome.
Answer: The equivalence classes are palindromes, non-palindromes, and invalid inputs. Now, selecting one representative value from each equivalence class, we have the required test suite: {abc,aba,abcdef}.



Boundary value analysis

- Programmers often fail to properly address the special processing required by the input values that lie at the boundary of the different equivalence classes. For example, programmers may improperly use < instead of <=, or conversely <= for <, etc.
- Boundary value analysis-based test suite design involves designing test cases using the values at the boundaries of different equivalence classes.
- **Example:** For a function that computes the square root of integer values in the range of 0 and 5000, the test cases must include the following values: {0,-1,5000,5001}.

WHITE-BOX TESTING

- WHITE BOX TESTING (also known as Clear Box Testing, Open Box Testing, Glass Box Testing, Transparent Box Testing, Code-Based Testing or Structural Testing) is a software testing method in which the internal structure/design/implementation of the item being tested is known to the tester.
- White-box testing is an important type of unit testing.
- A white-box testing strategy can either be coverage-based or fault-based.

Fault-based testing

A fault-based testing strategy targets to detect certain types of faults. These faults that a test strategy focuses on constitutes the fault model of the strategy. An example of a fault-based strategy is mutation testing.

Coverage-based testing

A coverage-based testing strategy attempts to execute (or cover) certain elements of a program. Popular examples of coverage-based testing strategies are:

- statement coverage
- branch coverage
- multiple condition coverage
- path coverage-based testing.

Testing criterion for coverage-based testing

The set of specific program elements that a testing strategy targets to execute is called the testing criterion of the strategy.

Stronger versus weaker testing

One white-box testing strategy is said to be stronger than another strategy, if all types of errors detected by the first testing strategy is also detected by the second testing strategy, and the second testing strategy additionally detects some more types of errors. If a stronger testing has been performed, then a weaker testing need not be carried out.

Mutation Testing

- The idea behind mutation testing is to make a few arbitrary changes to a program at a time.
- Each time the program is changed, it is called a mutated program and the change effected is called a mutant.
- A mutation operator makes specific changes to a program.
- If a mutant does not introduce any error in the program, then the original program and the mutated program are called equivalent programs.
- A mutated program is tested against the original test suite of the program.
- If there exists at least one test case in the test suite for which a mutated program yields an incorrect result, then the mutant is said to be dead, since the error introduced by the mutation operator has successfully been detected by the test suite.
- If a mutant remains alive even after all the test cases have been exhausted, the test suite is enhanced to kill the mutant.

Statement Coverage

- The principal idea governing the statement coverage strategy is that unless a statement is executed, there is no way to determine whether an error exists in that statement.

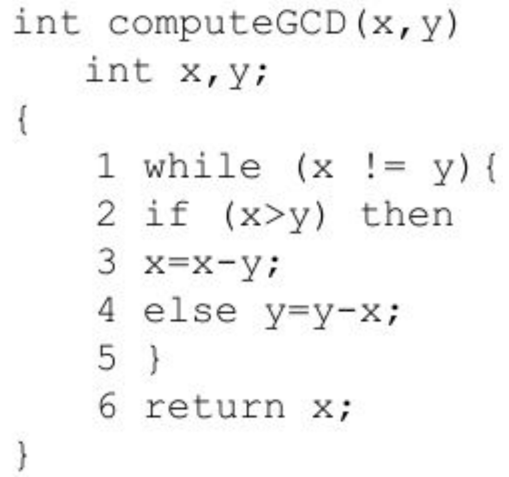
Design statement coverage-based test suite for the following Euclid’s GCD computation program:

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

Answer: To design the test cases for the statement coverage, the conditional expression of the while statement needs to be made true and the conditional expression of the if statement needs to be made both true and false. By choosing the test set {(x = 3, y = 3), (x = 4, y = 3), (x = 3, y = 4)}, all statements of the program would be executed at least once.

Branch Coverage

- A test suite satisfies branch coverage, if it makes each branch condition in the program to assume true and false values in turn.
- In other words, for branch coverage each branch in the CFG representation of the program must be taken at least once, when the test suite is executed.
- Branch testing is also known as edge testing, since in this testing scheme, each edge of a program's control flow graph is traversed at least once.
- Branch testing would guarantee statement coverage since every statement must belong to some branch
- statement coverage does not ensure branch coverage
  - Ex: if(x>2) x+=1;
  - test suite{5} would achieve statement coverage. However, it does not achieve branch coverage, since the condition (x > 2) is not made false by any test case in the suite.



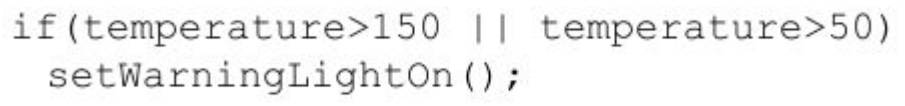
```
int computeGCD(x,y)
   int x,y;
{
    while (x != y){
    if (x>y) then
    x=x-y;
    else y=y-x;
    }
    return x;
}
```

The test suite {(x = 3, y = 3), (x = 3, y = 2), (x = 4, y = 3), (x = 3, y = 4)} achieves branch coverage.

Multiple Condition Coverage

- test cases are designed to make each component of a composite conditional expression to assume both true and false values
- For example, consider the composite conditional expression ((c1 .and.c2).or.c3). A test suite would achieve MC coverage, if all the component conditions c1, c2 and c3 are each made to assume both true and false values.
- For a composite conditional expression of n components, 2n test cases are required for multiple condition coverage. Thus, for multiple condition coverage, the number of test cases increases exponentially with the number of component conditions. Therefore, multiple condition coverage-based testing technique is practical only if n (the number of conditions) is small.

Consider the following C program segment:



```
if(temperature>150 || temperature>50)
  setWarningLightOn();
```

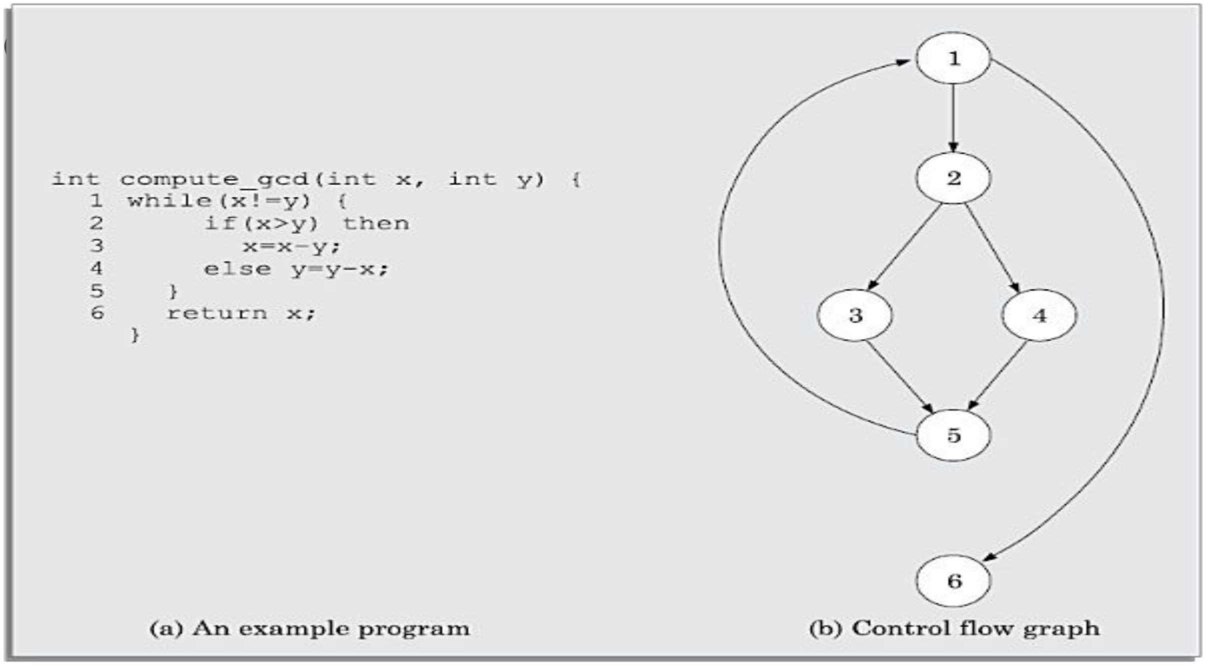
The program segment has a bug in the second component condition, it should have been temperature<50.
The test suite {temperature=160, temperature=40} achieves branch coverage.
But, it is not able to check that setWarningLightOn(); should not be called for temperature values within 150 and 50.

Path Coverage

- A test suite achieves path coverage if it exeutes each linearly independent paths (o r basis paths) at least once. A linearly independent path can be defined in terms of the control flow graph (CFG) of a program.

Control flow graph (CFG)

A control flow graph describes the sequence in which the different instructions of a
program get



```
int compute_gcd(int x, int y) {
  1  while(x!=y) {
  2      if(x>y) then
  3         x=x-y;
  4      else y=y-x;
  5    }
  6    return x;
     }
```

(a) An example program

(b) Control flow graph

McCabe’s Cyclomatic Complexity Metric

McCabe’s cyclomatic complexity defines an upper bound on the number of independent paths in a program.

Method 1: Given a control flow graph G of a program, the cyclomatic complexity V(G) can be computed as:
V(G) = E – N + 2
where, N is the number of nodes of the control flow graph and E is the number of edges in the control flow graph.

Method 2: An alternate way of computing the cyclomatic complexity of a program is based on a visual inspection of the control flow graph is as follows —
In this method, the cyclomatic complexity V (G) for a graph G is given by the following expression:
V(G) = Total number of non-overlapping bounded areas + 1

Method 3: The cyclomatic complexity of a program can also be easily computed by computing the number of decision and loop statements of the program. If N is the number of decision and loop statements of a program, then the McCabe’s metric is equal to N + 1.

Steps to carry out path coverage-based testing

- 1. Draw control flow graph for the program.
- 2. Determine the McCabe’s metric V(G).
- 3. Determine the cyclomatic complexity. This gives the minimum number of test cases required to achieve path coverage.
- 4. repeat
  - Test using a randomly designed set of test cases.
  - Perform dynamic analysis to check the path coverage achieved.
  - until at least 90 per cent path coverage is achieved.

INTEGRATION TESTING

- The objective of integration testing is to check whether the different modules of a program interface with each other properly.
- Any one (or a mixture) of the following approaches can be used to develop the test plan:
  - Big-bang approach to integration testing
  - Top-down approach to integration testing
  - Bottom-up approach to integration testing
  - Mixed (also called sandwiched) approach to integration testing

Big-bang approach to integration testing

- In this approach, all the modules making up a system are integrated in a single step.
- In simple words, all the unit tested modules of the system are simply linked together and tested.
- However, this technique can meaningfully be used only for very small systems.
- The main problem with this approach is that once a failure has been detected during integration testing, it is very difficult to localise the error as the error may potentially lie in any of the modules.
- Therefore, debugging errors reported during big-bang integration testing are very expensive to fix.
- As a result, big-bang integration testing is almost never used for large programs.

Bottom-up approach to integration testing

Large software products are often made up of several subsystems. A subsystem might consist of many modules which communicate among each other through well-defined interfaces. In bottom-up integration testing, first the modules for the each subsystem are integrated. Thus, the subsystems can be integrated separately and independently.

Top-down approach to integration testing

- Top-down integration testing starts with the root module in the structure chart and one or two subordinate modules of the root module.
- After the top-level ‘skeleton’ has been tested, the modules that are at the
- immediately lower layer of the ‘skeleton’ are combined with it and Tested.
- An advantage of top-down integration testing is that it requires writing only stubs, and stubs are simpler to write compared to drivers.
- A disadvantage of the top-down integration testing approach is that in the absence of lower-level routines, it becomes difficult to exercise the top-level routines in the desired manner since the lower level routines usually perform input/output (I/O) operations.

Mixed approach to integration testing

- The mixed (also called sandwiched) integration testing follows a combination of top-down and bottom-up testing approaches.
- In top- down approach, testing can start only after the top-level modules have been coded and unit tested.
- Similarly, bottom-up testing can start only after the bottom level modules are ready.
- The mixed approach overcomes this shortcoming of the top-down and bottom-up approaches. In the mixed testing approach, testing can start as and when modules become available after unit testing.

Smoke Testing

- Smoke testing is carried out before initiating system testing to ensure that system testing would be meaningful, or whether many parts of the software would fail.
- The idea behind smoke testing is that if the integrated program cannot pass even the basic tests, it is not ready for a vigorous testing.
- For smoke testing, a few test cases are designed to check whether the basic functionalities are working.
- For example, for a library automation system, the smoke tests may check whether books can be created and deleted, whether member records can be created and deleted, and whether books can be loaned and returned

SOFTWARE MAINTENANCE

- Software maintenance denotes any changes made to a software product after it has been delivered to the customer.
- Maintenance is inevitable for almost any kind of product.
- Software products need maintenance to correct errors, enhance features, port to new platforms, etc.

Types of Software Maintenance

Corrective: Corrective maintenance of a software product is necessary either to rectify the bugs observed while the system is in use.

Adaptive: A software product might need maintenance when the customers need the product to run on new platforms, on new operating systems, or when they need the product to interface with new hardware or software.

Perfective: A software product needs maintenance to support the new features that users want it to support, to change different functionalities of the system according to customer demands, or to enhance the performance of the system.

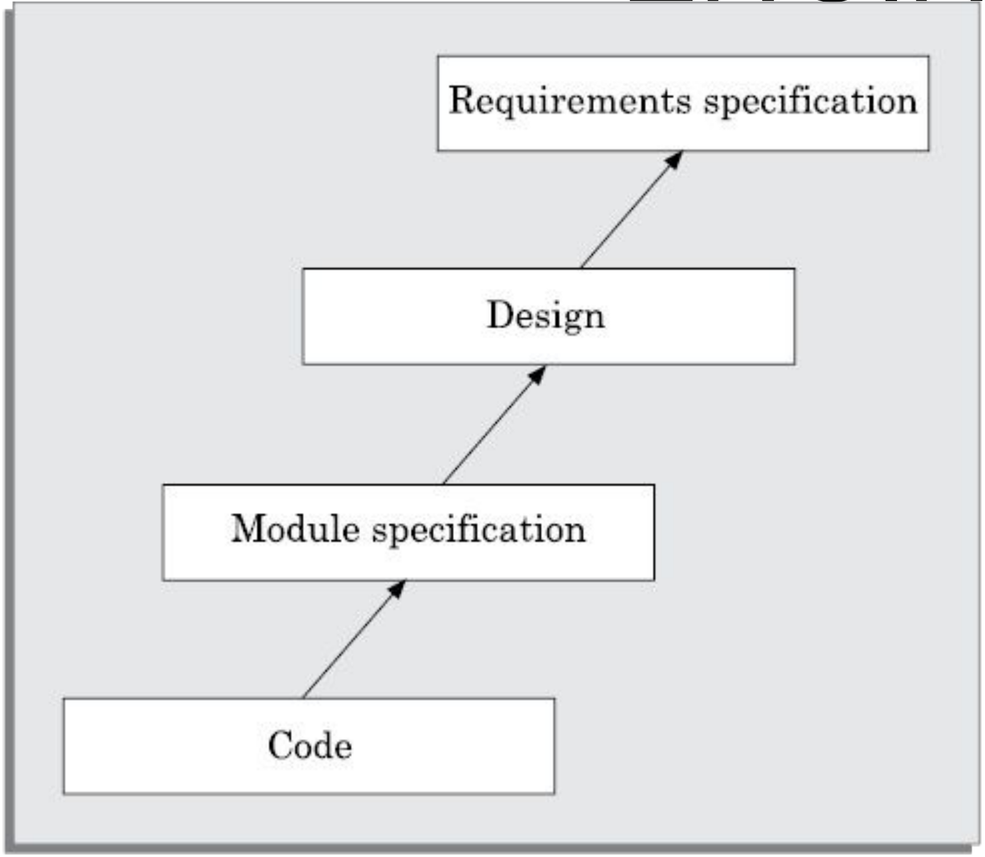
Characteristics of Software Evolution

Lehman’s first law: A software product must change continually or become progressively less useful.

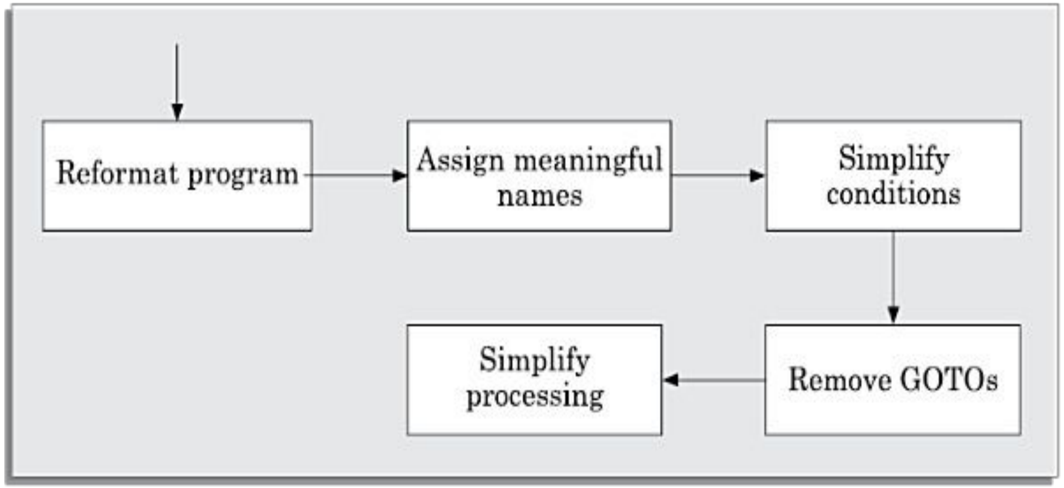
Lehman’s second law: The structure of a program tends to degrade as more and more maintenance is carried out on it.

Lehman’s third law: Over a program’s lifetime, its rate of development is approximately constant

SOFTWARE REVERSE ENGINEERING



A process model for reverse engineering

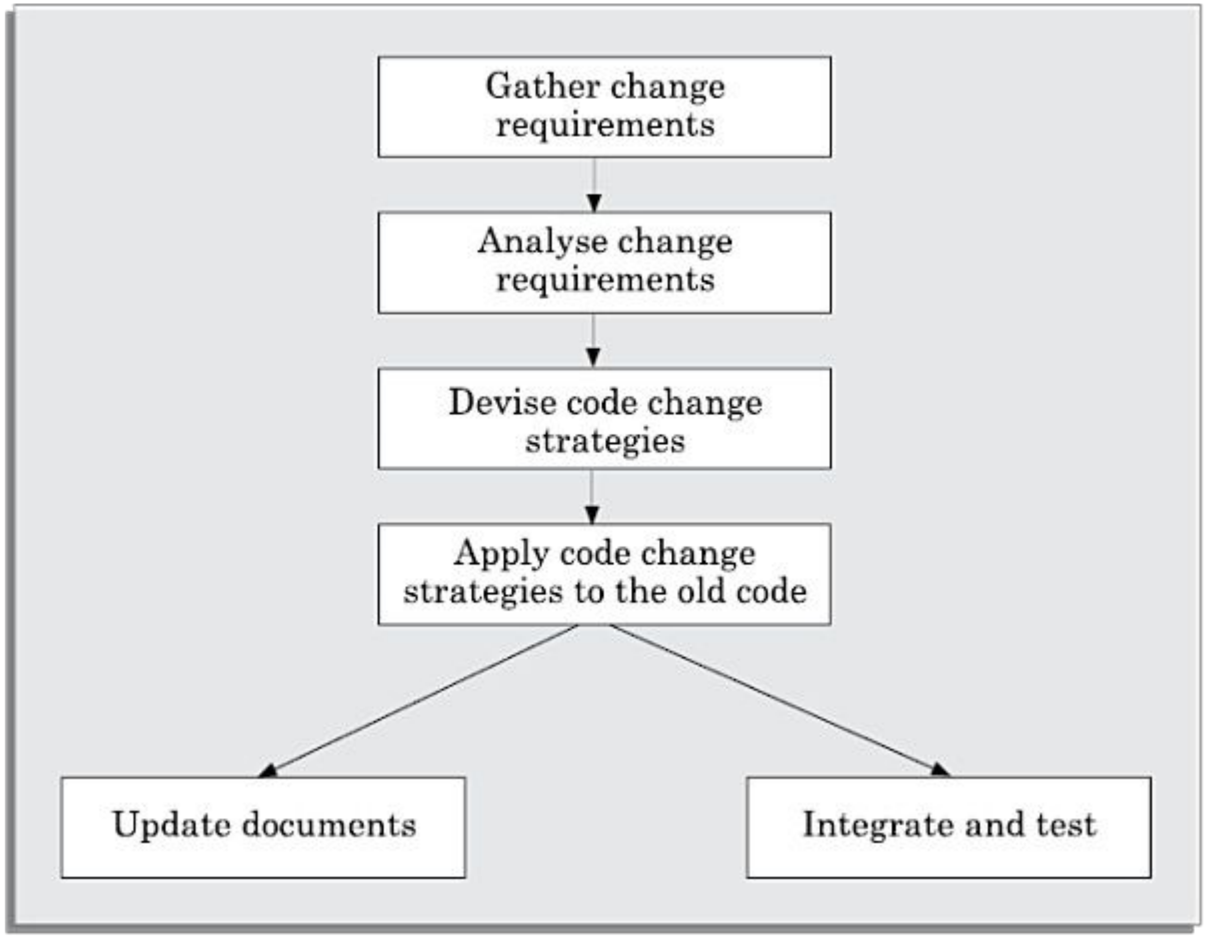


Cosmetic changes carried out before reverse engineering.

SOFTWARE MAINTENANCE PROCESS MODELS

- The activities involved in a software maintenance project are not unique and depend on several factors such as:
- (i) the extent of modification to the product required,
- (ii) the resources available to the maintenance team,
- (iii) the conditions of the existing product (e.g., how structured it is, how well documented it is, etc.),
- (iv) the expected project risks, etc.
- There are two software maintenance process models are available:
  - First Model
  - Second Model

First model

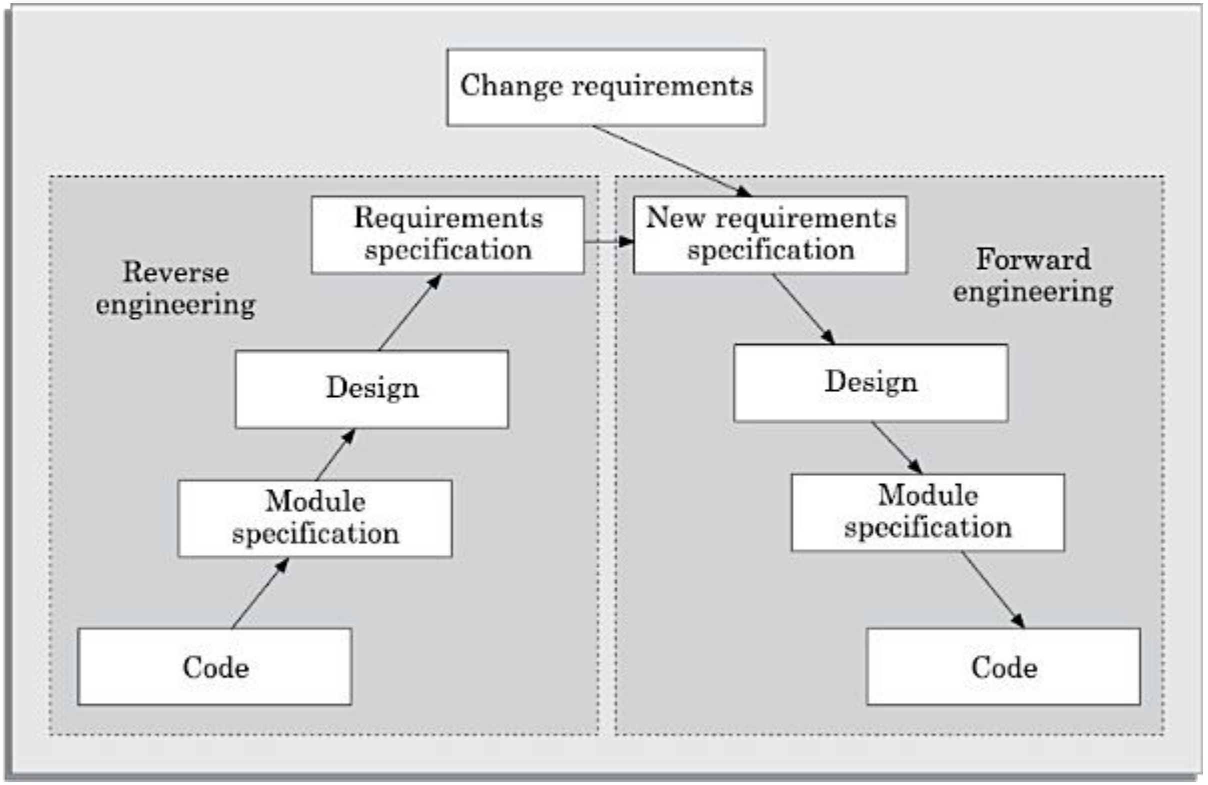

Gather change requirements
Analyse change requirements
Devise code change strategies
Apply code change strategies to the old code
Update documents
Integrate and test

The first model is preferred for projects involving small reworks where the code is changed directly and the changes are reflected in the relevant documents later

- In this approach, the project starts by gathering the requirements for changes.
- The requirements are next analysed to formulate the strategies to be adopted for code change.
- At this stage, the association of at least a few members of the original development team goes a long way in reducing the cycle time, especially for projects involving unstructured and inadequately documented code.
- The availability of a working old system to the maintenance engineers at the maintenance site greatly facilitates the task of the maintenance team as they get a good insight into the working of the old system and also can compare the working of their modified system with the old system.
- Also, debugging of the re- engineered system becomes easier as the program traces of both the systems can be compared to localise the bugs.

Second model

The second model is preferred for projects where the amount of rework required is significant. This approach can be represented by a reverse engineering cycle followed by a forward engineering cycle.



- The reverse engineering cycle is required for legacy products.
- During the reverse engineering, the old code is analysed (abstracted) to extract the module specifications.
- The module specifications are then analysed to produce the design.
- The design is analysed (abstracted) to produce the original requirements specification.
- The change requests are then applied to this requirements specification to arrive at the new requirements specification.
- At this point a forward engineering is carried out to produce the new code.
- At the design, module specification, and coding a substantial reuse is made from the reverse engineered products.
- An important advantage of this approach is that it produces a more structured design compared to what the original product had, produces good documentation, and very often results in increased efficiency.
- The efficiency improvements are brought about by a more efficient design.
- However, this approach is more costly than the first approach.

ESTIMATION OF MAINTENANCE COST

annual change traffic (ACT)

$$\mathrm{ACT} = \frac{\mathrm{KLOC}_{\mathrm{added}} + \mathrm{KLOC}_{\mathrm{deleted}}}{\mathrm{KLOC}_{\mathrm{total}}}$$

Maintenance cost = ACT × Development cost