

SOFTWARE ENGINEERING

What is Software?

Software is more than just a program code. A program is an executable code, which serves some computational purpose. Software is considered to be a collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called **software product**.

What is Engineering?

Engineering is all about developing products, using well-defined, scientific principles and methods.

Software Engineering?

***Software engineering* is an engineering branch associated with the development of software product using well-defined scientific principles, methods and procedures. The outcome of software engineering is an efficient and reliable software product.**

IEEE defines software engineering as:

The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software.

Why Software Engineering?

- ❑ Without using software engineering principles it would be difficult to develop large programs.
- ❑ Complexity and difficulty levels of the programs increase exponentially with their sizes.
- ❑ Software engineering principles use two important techniques to reduce problem complexity:
 - *Abstraction*
 - *Decomposition*

Abstraction

- A problem can be simplified by **omitting irrelevant details.**
- Consider only those aspects of the problem that are relevant for certain purpose and suppress other aspects that are not relevant for the given purpose.
- Once the simpler problem is solved, then the omitted details can be taken into consideration to solve the next lower level abstraction, and so on.
- A powerful way of reducing the complexity of the problem.

Decomposition

- a complex problem is divided into several smaller problems and then the smaller problems are solved one by one.
- any random decomposition of a problem into smaller parts will not help.
- The problem has to be decomposed such that each component of the decomposed problem can be solved independently and then the solution of the different components can be combined to get the full solution.
- A good decomposition of a problem should minimize interactions among various components.
- If the different subcomponents are interrelated, then the different components cannot be solved separately and the desired reduction in complexity will not be realized.

Need of Software Engineering

- **Large software** - It is easier to build a wall than to a house or building, likewise, as the size of software become large engineering has to step to give it a scientific process.
- **Scalability**- If the software process were not based on scientific and engineering concepts, it would be easier to re-create new software than to scale an existing one.
- **Cost**- As hardware industry has shown its skills and huge manufacturing has lower down the price of computer and electronic hardware. But the cost of software remains high if proper process is not adapted.
- **Dynamic Nature**- The always growing and adapting nature of software hugely depends upon the environment in which the user works. If the nature of software is always changing, new enhancements need to be done in the existing one. This is where software engineering plays a good role.
- **Quality Management**- Better process of software development provides better and quality software product.

CHARACTERISTICS OF GOOD SOFTWARE

A software product can be judged by what it offers and how well it can be used.

This software must satisfy on the following grounds:

- Operational
- Transitional
- Maintenance

Operational

This tells us how well software works in operations. It can be measured on:

- **Budget**
- **Usability**
- **Efficiency**
- **Correctness**
- **Functionality**
- **Dependability**
- **Security**
- **Safety**

Transitional

This aspect is important when the software is moved from one platform to another:

- Portability
- Interoperability
- Reusability
- Adaptability

Maintenance

This aspect briefs about how well a software has the capabilities to maintain itself in the everchanging environment:

- **Modularity**
- **Maintainability**
- **Flexibility**
- **Scalability**

SOFTWARE DEVELOPMENT LIFE CYCLE

Life Cycle Model

- A Software life cycle model (also called process model) is a descriptive and diagrammatic representation of the software life cycle.
- It represents all the activities required to make a software product transit through its life cycle phases.
- It captures the order in which these activities are to be undertaken.
- It maps the different activities performed on a software product from its inception to retirement.
- Different life cycle models may map the basic development activities to phases in different ways.
- No matter which life cycle model is followed, the basic activities are included in all life cycle models though the activities may be carried out in different orders in different life cycle models.

Need for A Life Cycle Model

- To develop of a software product in a systematic and disciplined manner.
- To make a clear understanding among team members about when and what to do. Otherwise it would lead to chaos and project failure.
- A software life cycle model defines entry and exit criteria for every phase. A phase can start only if its phase-entry criteria have been satisfied.
- Without software life cycle model the entry and exit criteria for a phase cannot be recognized.
- Without software life cycle models it becomes very difficult for software project managers to monitor the progress of the project.

Different Life Cycle Models

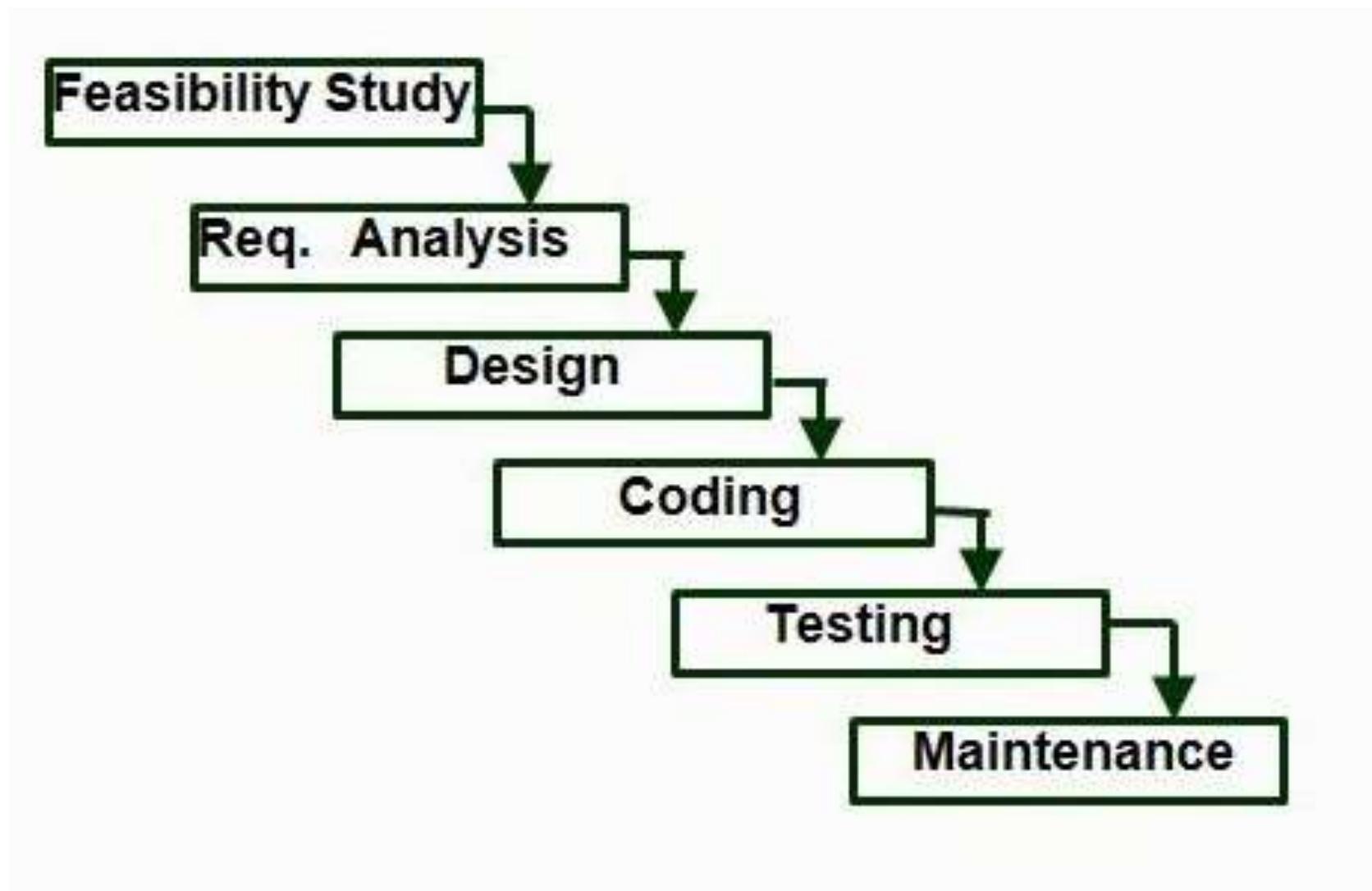
- Classical Waterfall Model
- Iterative Waterfall Model
- Prototyping Model
- Evolutionary Model
- Spiral Model

Classical Waterfall Model

sposto

The classical waterfall model is intuitively the most obvious way to develop software. Though the classical waterfall model is elegant and intuitively obvious, it is not a practical model in the sense that it cannot be used in actual software development projects. Thus, this model can be considered to be a *theoretical way of developing software*. But all other life cycle models are essentially derived from the classical waterfall model. So, in order to be able to appreciate other life cycle models, it is necessary to learn the classical waterfall model.

Classical Waterfall Model



Classical Waterfall Model

- Feasibility Study**
- Requirements Analysis and Specification**
- Design**
- Coding and Unit Testing**
- Integration and System Testing**
- Maintenance**

Feasibility Study

- The main aim of feasibility study is to determine whether it would be financially and technically feasible to develop the product.
- At first project managers or team leaders try to have a rough understanding of what is required to be done by visiting the client side. They study different input data to the system and output data to be produced by the system. They study what kind of processing is needed to be done on these data and they look at the various constraints on the behavior of the system.
- After they have an overall understanding of the problem they investigate the different solutions that are possible. Then they examine each of the solutions in terms of what kind of resources required, what would be the cost of development and what would be the development time for each solution.
- Based on this analysis they pick the best solution and determine whether the solution is feasible financially and technically. They check whether the customer budget would meet the cost of the product and whether they have sufficient technical expertise in the area of development.

Requirements Analysis and Specification

The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document them properly. This phase consists of two distinct activities, namely

- Requirements gathering and analysis
- Requirements specification

The requirements analysis activity is begun by collecting all relevant data regarding the product to be developed from the users of the product and from the customer through interviews and discussions.

Requirements Analysis and Specification

The data collected from a group of users usually contain several contradictions and ambiguities.

Identify all ambiguities and contradictions in the requirements and resolve them through further discussions with the customer.

After all ambiguities, inconsistencies, and incompleteness have been resolved and all the requirements properly understood, the requirements specification activity can start.

During this activity, the user requirements are systematically organized into a Software Requirements Specification (SRS) document. The customer requirements identified during the requirements gathering and analysis activity are organized into a SRS document. The important components of this document are functional requirements, the nonfunctional requirements, and the goals of implementation.

Design

Transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language.

In technical terms, during the design phase the software architecture is derived from the SRS document.

Two distinctly different approaches are available:

- The traditional design approach
- The object-oriented design approach.

Traditional Design Approach

Traditional design consists of two different activities; first a structured analysis of the requirements specification is carried out where the detailed structure of the problem is examined. This is followed by a structured design activity. During structured design, the results of structured analysis are transformed into the software design.

Object Oriented Design Approach

In this technique, various objects that occur in the problem domain and the solution domain are first identified, and the different relationships that exist among these objects are identified. The object structure is further refined to obtain the detailed design.

Coding and Unit Testing

The purpose of the coding phase (sometimes called the **implementation** phase) of software development is to translate the software design into source code. Each component of the design is implemented as a program module. The end-product of this phase is a set of program modules that have been individually tested. During this phase, each module is unit tested to determine the correct working of all the individual modules. It involves testing each module in isolation as this is the most efficient way to debug the errors identified at this stage.

Integration and System Testing

milon

Integration of different modules is undertaken once they have been coded and unit tested.

All the modules are integrated in a planned manner.

Integration is normally carried out incrementally over a number of steps.

During each integration step, the partially integrated system is tested and a set of previously planned modules are added to it.

Finally, when all the modules have been successfully integrated and tested, system testing is carried out.

Integration and System Testing

System testing usually consists of three different kinds of testing activities:

- **α – testing:** It is the system testing performed by the development team.
- **β –testing:** It is the system testing performed by a friendly set of customers.
- **Acceptance testing:** It is the system testing performed by the customer himself after the product delivery to determine whether to accept or reject the delivered product.

Maintenance

Maintenance of a typical software product requires much more than the effort necessary to develop the product itself.

Maintenance involves performing any one or more of the following three kinds of activities:

- Correcting errors that were not discovered during the product development phase.
This is called corrective maintenance.
- Improving the implementation of the system, and enhancing the functionalities of the system according to the customer's requirements. This is called perfective maintenance.
- Porting the software to work in a new environment. For example, porting may be required to get the software to work on a new computer platform or with a new operating system. This is called adaptive maintenance.

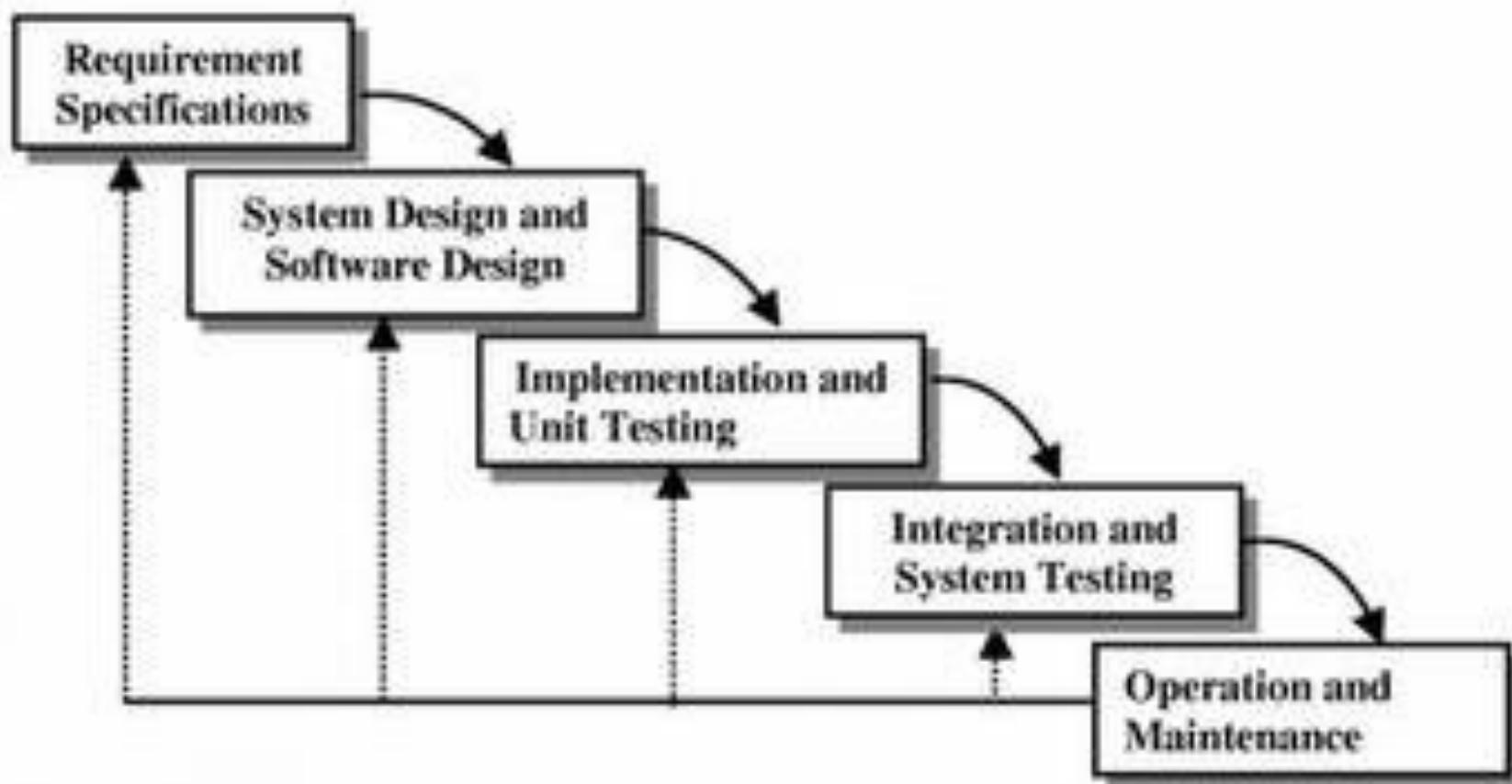
Shortcomings of The Classical Waterfall Model

The classical waterfall model is an **idealistic one** since it assumes that no development error is ever committed by the engineers during any of the life cycle phases. However, in practical development environments, the engineers do commit a large number of errors in almost every phase of the life cycle. The source of the defects can be many: oversight, wrong assumptions, use of inappropriate technology, communication gap among the project engineers, etc. These defects usually get detected much later in the life cycle.

Software Development Life Cycle Model

Iterative Waterfall Model

To overcome the major shortcomings of the classical waterfall model, we come up with the iterative waterfall model.



Iterative Waterfall Model

- **Feedback paths are provided for error correction as & when detected later in a phase.** Though errors are inevitable, but it is desirable to detect them in the same phase in which they occur. If so, this can reduce the effort to correct the bug.
- The advantage of this model is that there is a working model of the system at a very early stage of development which makes it easier to find functional or design flaws. Finding issues at an early stage of development enables to take corrective measures in a limited budget.
- The disadvantage with this SDLC model is that it is applicable only to large and bulky software development projects. This is because it is hard to break a small software system into further small serviceable increments/modules.

Prototyping Model

- ❑ A prototype is a toy implementation of the system.
- ❑ A prototype usually exhibits limited functional capabilities, low reliability, and inefficient performance compared to the actual software.
- ❑ A prototype is usually built using several shortcuts. The shortcuts might involve using inefficient, inaccurate, or dummy functions. The shortcut implementation of a function. It may produce the desired results by using a table look-up instead of performing the actual computations.
- ❑ A prototype usually turns out to be a very crude version of the actual system.
- ❑ A prototype of the actual product is preferred in situations such as:
 - User requirements are not complete
 - Technical issues are not clear

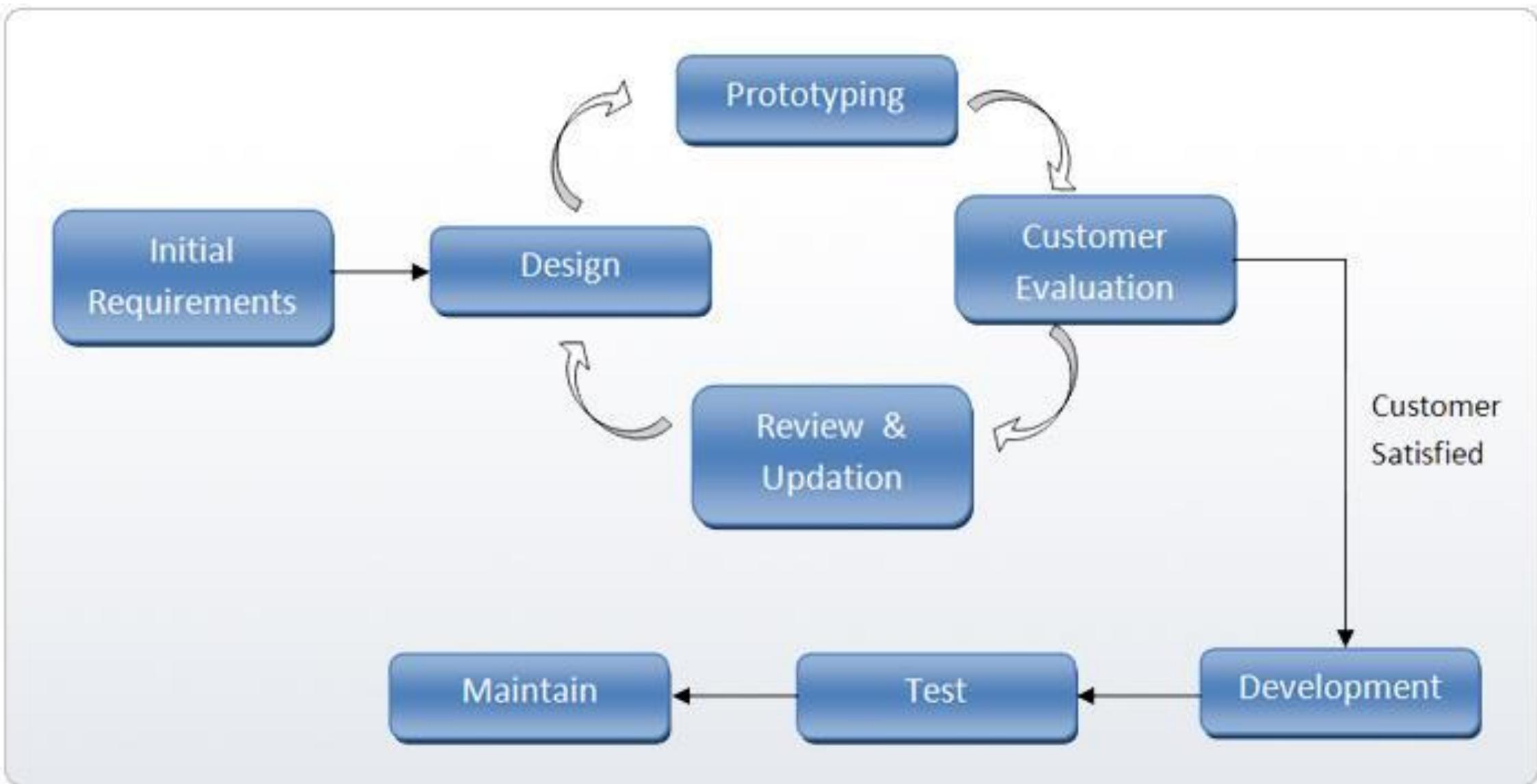
Prototyping Model

□ Need for a prototype in software development

There are several uses of a prototype. An important purpose is to illustrate the input data formats, messages, reports, and the interactive dialogues to the customer. This is a valuable mechanism for gaining better understanding of the customer's needs:

- How the screens might look like
- How the user interface would behave
- How the system would produce outputs

Prototyping Model



Prototyping Model

A reason for developing a prototype is that it is impossible to get the perfect product in the first attempt. Many researchers and engineers advocate that if anyone want to develop a good product he/she must plan to throw away the first version. The experience gained in developing the prototype can be used to develop the final product.

A prototyping model can be used when technical solutions are unclear to the development team. A developed prototype can help engineers to critically examine the technical issues associated with the product development. Often, major design decisions depend on issues like the response time of a hardware controller, or the efficiency of a sorting algorithm, etc. In such circumstances, a prototype may be the best or the only way to resolve the technical issues.

Evolutionary Model

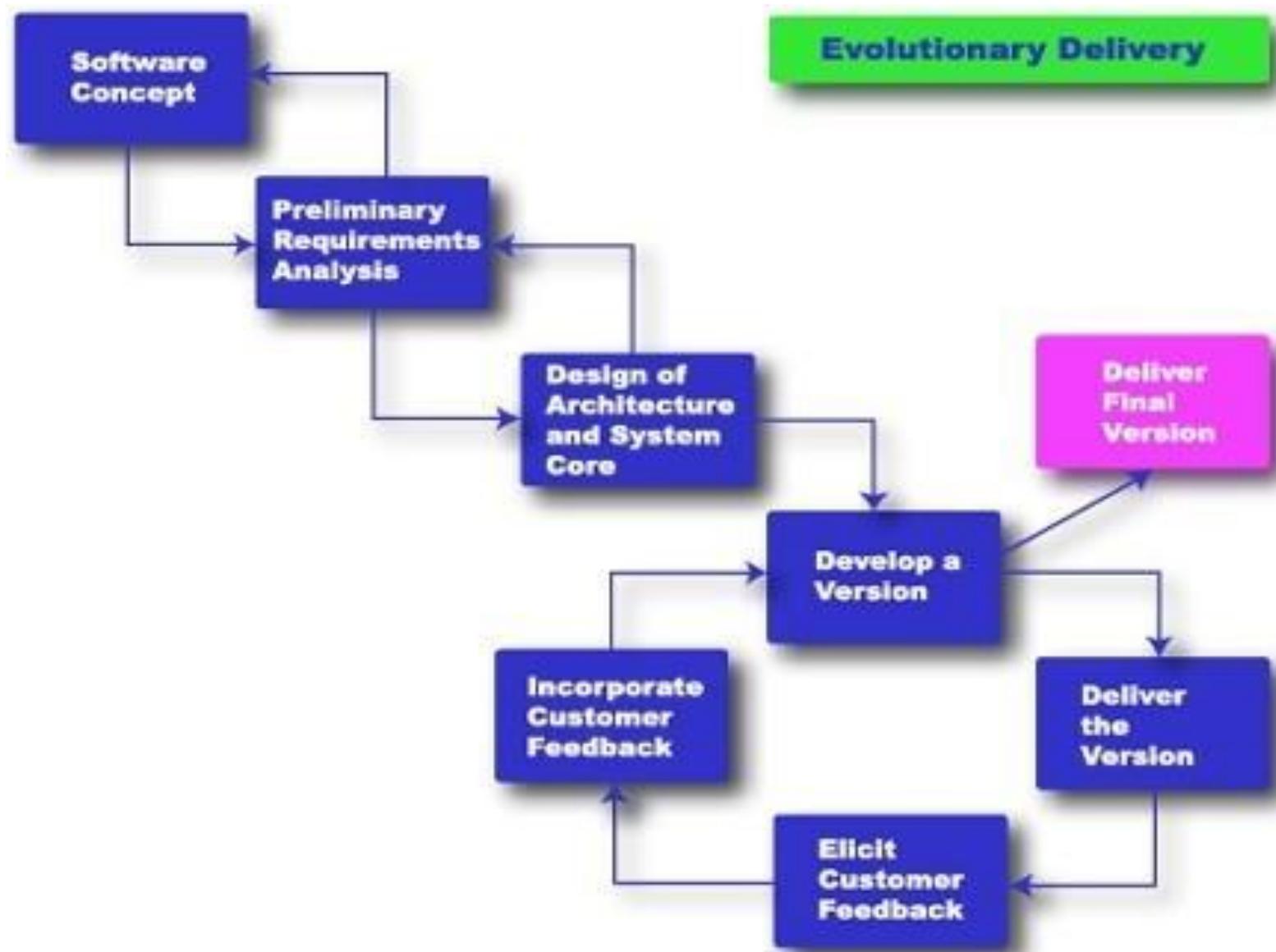
It is also called *successive versions model* or *incremental model*. At first, a simple working model is built. Subsequently it undergoes functional improvements & we keep on adding new functions till the desired system is built.

Applications:

Large projects where you can easily find modules for incremental implementation. Often used when the customer wants to start using the core features rather than waiting for the full software.

Also used in object oriented software development because the system can be easily portioned into units in terms of objects.

Evolutionary Model



Evolutionary Model

Advantages:

- User gets a chance to experiment partially developed system.**
- Reduce the error because the core modules get tested thoroughly.**

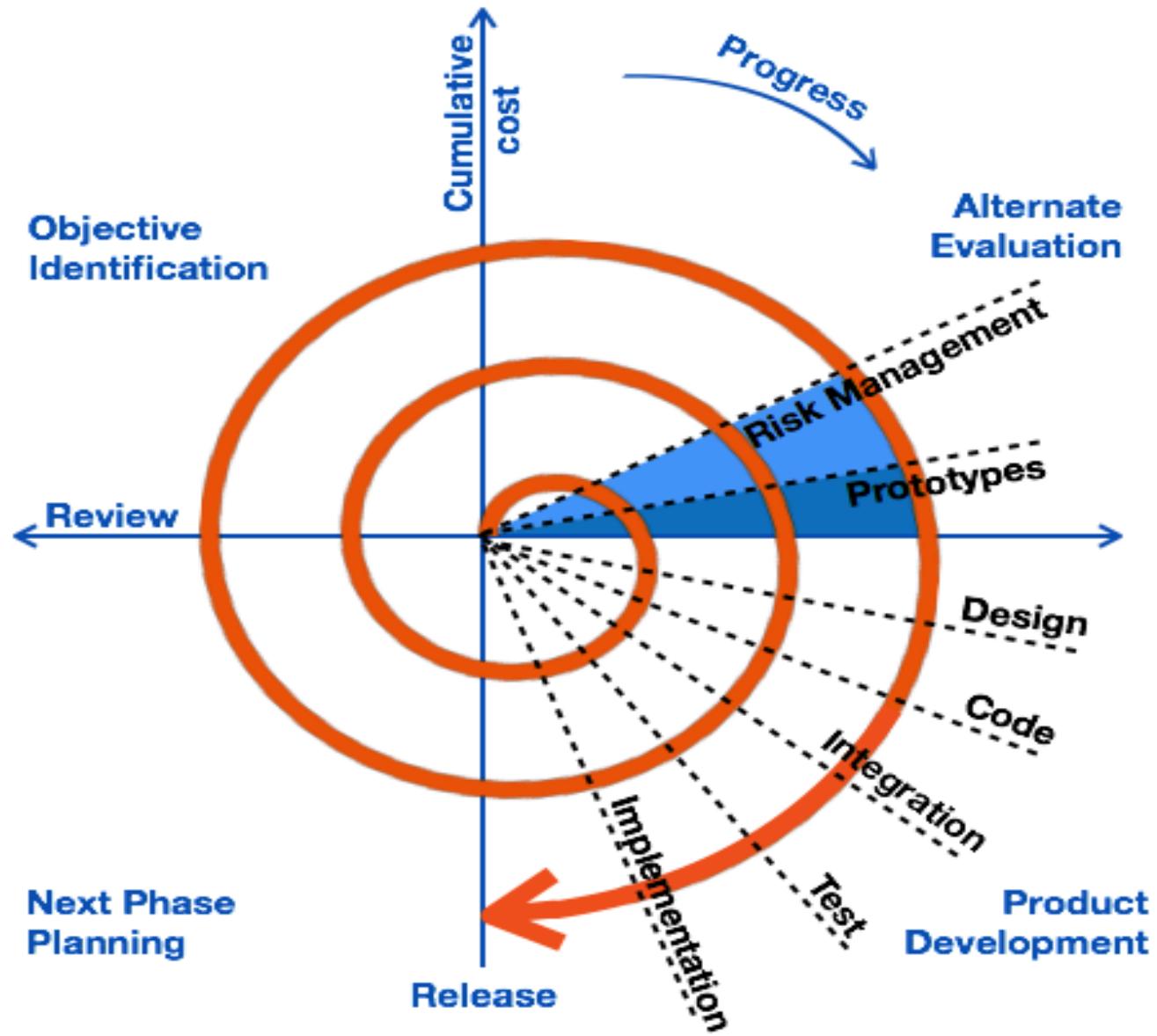
Disadvantages:

- It is difficult to divide the problem into several versions that would be acceptable to the customer which can be incrementally implemented & delivered.**

Spiral Model

- ❑The diagrammatic representation of the Spiral model appears like a spiral with many loops.**
- ❑The exact number of loops in the spiral is not fixed.**
- ❑Each loop of the spiral represents a phase of the software process. For example, the innermost loop might be concerned with feasibility study, the next loop with requirements specification, the next one with design, and so on.**
- ❑Each phase in this model is split into four sectors (or quadrants).**

Spiral Model



Spiral Model

First quadrant (Objective Setting)

- During the first quadrant, it is needed to identify the objectives of the phase.
- Examine the risks associated with these objectives.

Second Quadrant (Risk Assessment and Reduction)

- A detailed analysis is carried out for each identified project risk.
- Steps are taken to reduce the risks. For example, if there is a risk that the requirements are inappropriate, a prototype system may be developed.

Spiral Model

Third Quadrant (Development and Validation)

- **Develop and validate the next level of the product after resolving the identified risks.**

Fourth Quadrant (Review and Planning)

- **Review the results achieved so far with the customer and plan the next iteration around the spiral.**
- **Progressively more complete version of the software gets built with each iteration around the spiral.**

Spiral Model

Circumstances to use spiral model

The spiral model is called a meta model since it encompasses all other life cycle models. Risk handling is inherently built into this model. The spiral model is suitable for development of technically challenging software products that are prone to several kinds of risks. However, this model is much more complex than the other models – this is probably a factor deterring its use in ordinary projects.

RAD Model

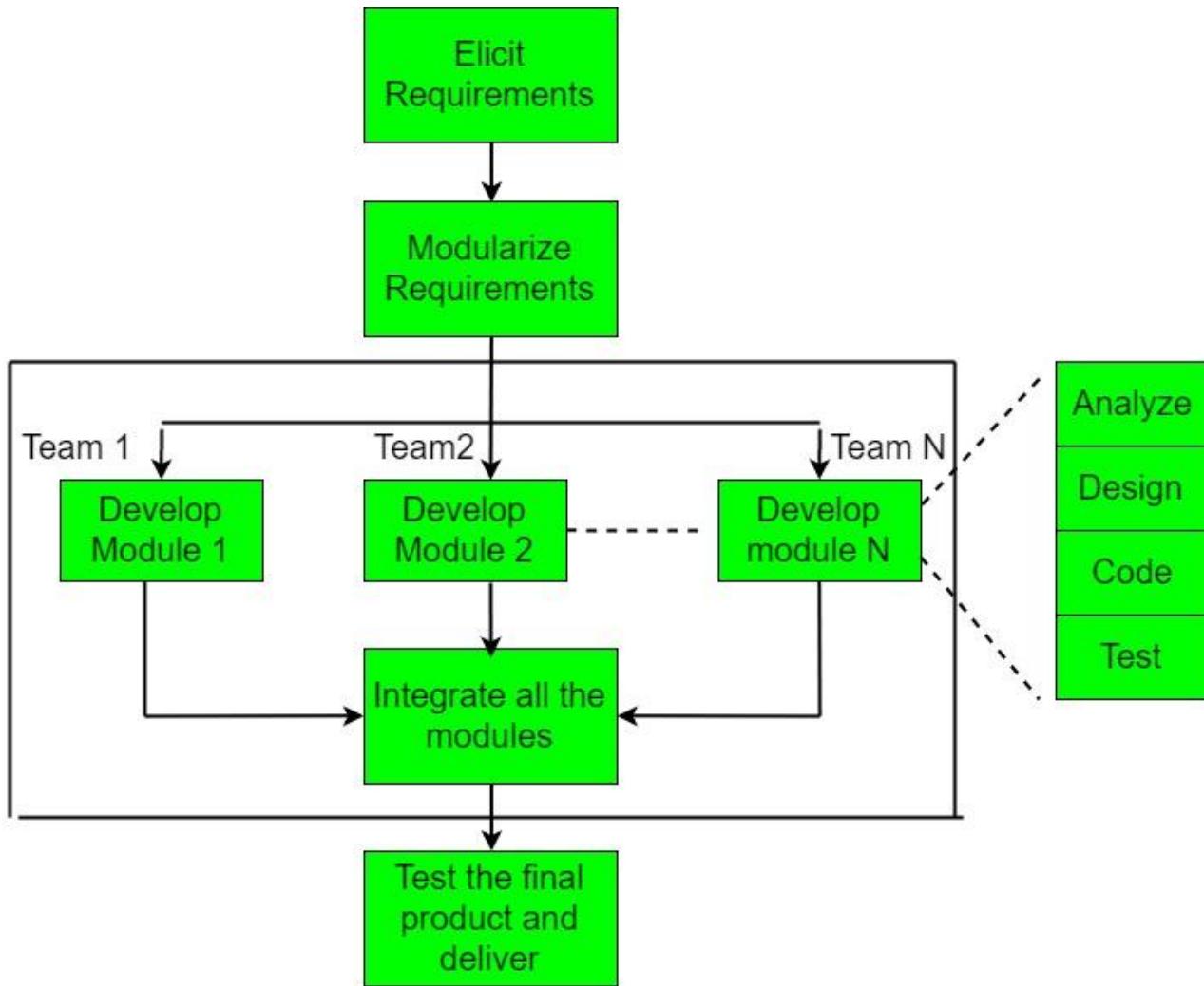
RAD Model

The Rapid Application Development Model was first proposed by IBM in the 1980s. The RAD model is a type of incremental process model in which there is an extremely short development cycle. When the requirements are fully understood and the component-based construction approach is adopted then the RAD model is used.

he critical feature of this model is the use of powerful development tools and techniques. A software project can be implemented using this model if the project can be broken down into small modules wherein each module can be assigned independently to separate teams. Multiple teams work on developing the software system using the RAD model parallelly.

Another striking feature of this model is a short period i.e. the time frame for delivery(time-box) is generally 60-90 days.

RAD Model



RAD Model

When to use the RAD Model?

- Well-understood Requirements:** When project requirements are stable and transparent, RAD is appropriate.
- Time-sensitive Projects:** Suitable for projects that need to be developed and delivered quickly due to tight deadlines.
- Small to Medium-Sized Projects:** Better suited for smaller initiatives requiring a controllable number of team members.
- High User Involvement:** Fits where ongoing input and interaction from users are essential.
- Innovation and Creativity:** Helpful for tasks requiring creative inquiry and innovation.
- Prototyping:** It is necessary when developing and improving prototypes is a key component of the development process.
- Low technological Complexity:** Suitable for tasks using comparatively straightforward technological specifications.

RAD Model

Objectives of Rapid Application Development Model (RAD)

Speedy Development

Accelerating the software development process is RAD's main goal. RAD prioritizes rapid prototyping and iterations to produce a working system as soon as possible.

Adaptability and Flexibility

RAD places a strong emphasis on adapting quickly to changing needs. Due to the model's flexibility, stakeholders can modify and improve the system in response to changing requirements and user input.

Stakeholder Participation

Throughout the development cycle, RAD promotes end users and stakeholders' active participation.

RAD Model

Objectives of Rapid Application Development Model (RAD)

Improved Interaction

Development teams and stakeholders may collaborate and communicate more effectively. Frequent communication and feedback loops guarantee that all project participants are in agreement, which lowers the possibility of misunderstandings.

Improved Quality via Prototyping

Prototypes enable early system component testing and visualization in Rapid Application Development (RAD).

Customer Satisfaction

Through rapid delivery of functioning prototypes and user involvement throughout the development process, Rapid Application Development (RAD) enhances the probability of customer satisfaction with the final product.

RAD Model

Advantages of Rapid Application Development Model (RAD)

- **The use of reusable components helps to reduce the cycle time of the project.**
- **Feedback from the customer is available at the initial stages.**
- **Reduced costs as fewer developers are required.**
- **The use of powerful development tools results in better quality products in comparatively shorter periods.**
- **The progress and development of the project can be measured through the various stages.**
- **It is easier to accommodate changing requirements due to the short iteration time spans.**
- **Productivity may be quickly boosted with a lower number of employees.**

RAD Model

Disadvantages of Rapid application development model (RAD)

- **The use of powerful and efficient tools requires highly skilled professionals.**
- **The absence of reusable components can lead to the failure of the project.**
- **The team leader must work closely with the developers and customers to close the project on time.**
- **The systems which cannot be modularized suitably cannot use this model.**
- **Customer involvement is required throughout the life cycle.**
- **It is not meant for small-scale projects as in such cases, the cost of using automated tools and techniques may exceed the entire budget of the project.**
- **Not every application can be used with RAD.**

Agile Model

- ❑ Agile Software Development is a software development methodology that values **flexibility, collaboration, and customer satisfaction.**
- ❑ It is based on the Agile Manifesto, a set of principles for software development that prioritize individuals and interactions, working software, customer collaboration, and responding to change.
- ❑ Agile Software Development is an **iterative and incremental approach** to software development that emphasizes the importance of delivering a working product quickly and frequently.
- ❑ It involves close collaboration between the development team and the customer to ensure that the product meets their needs and expectations.

Agile Model

Four Core Values of Agile Software Development

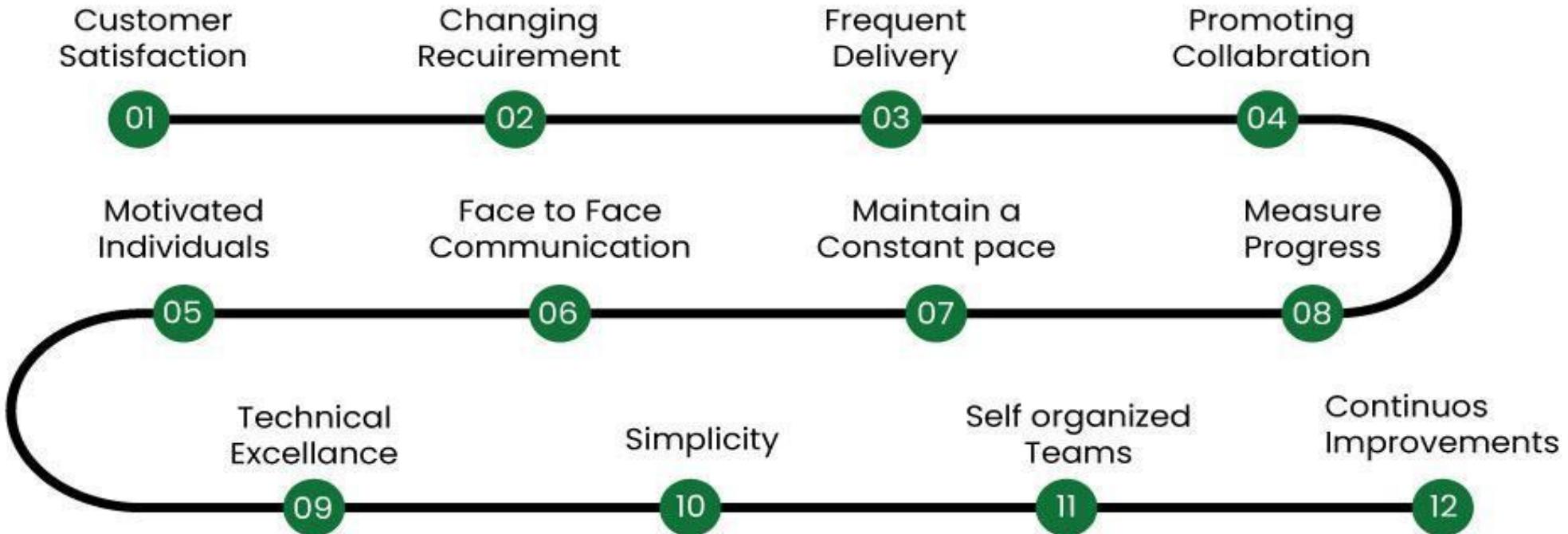
- **Individuals and Interactions over Processes and Tools**
- **Working Software over Comprehensive Documentation**
- **Customer Collaboration over Contract Negotiation**
- **Responding to Change over Following a Plan**



Agile Model

Twelve Principles of Agile Software Development

The Agile Manifesto is based on four values and twelve principles that form the basis, for methodologies.



Agile Model

These principles include:

- 1. Ensuring customer satisfaction through the early delivery of software.**
- 2. Being open to changing requirements in the stages of the development.**
- 3. Frequently delivering working software with a main focus on preference for timeframes.**
- 4. Promoting collaboration between business stakeholders and developers as an element.**
- 5. Structuring the projects around individuals. Providing them with the necessary environment and support.**
- 6. Prioritizing face to face communication whenever needed.**
- 7. Considering working software as the measure of the progress.**
- 8. Fostering development by allowing teams to maintain a pace indefinitely.**
- 9. Placing attention on excellence and good design practices.**
- 10. Recognizing the simplicity as crucial factor aiming to maximize productivity by minimizing the work.**
- 11. Encouraging self organizing teams as the approach to design and build systems.**
- 12. Regularly reflecting on how to enhance effectiveness and to make adjustments accordingly.**

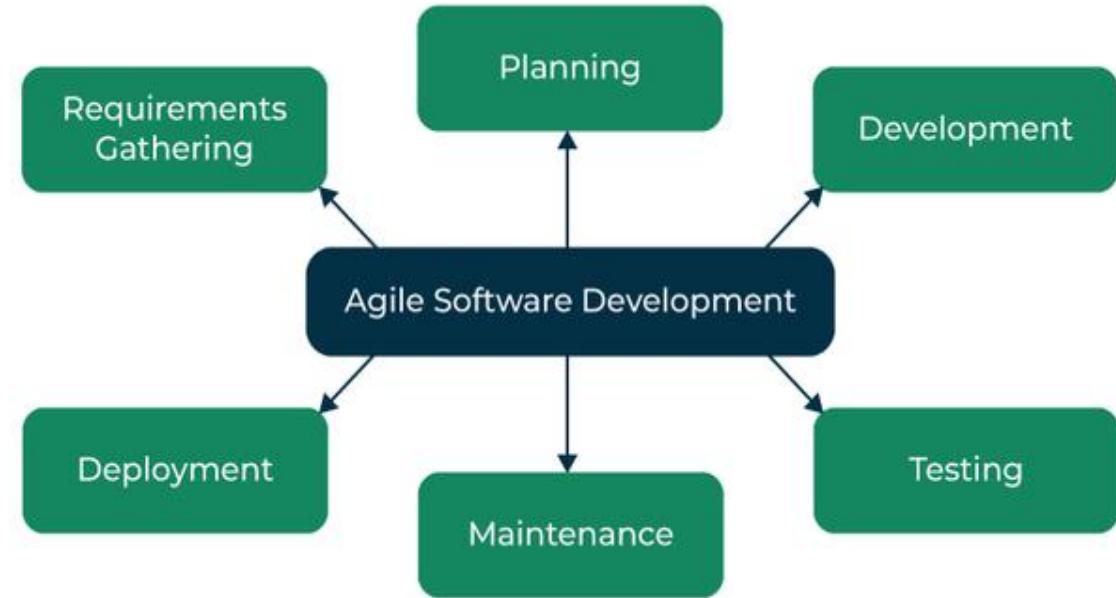
Agile Model

The Agile Software Development Process

Requirements Gathering: The customer's requirements for the software are gathered and prioritized.

Planning: The development team creates a plan for delivering the software, including the features that will be delivered in each iteration.

Development: The development team works to build the software, using frequent and rapid iterations.



Testing: The software is thoroughly tested to ensure that it meets the customer's requirements and is of high quality.

Deployment: The software is deployed and put into use.

Maintenance: The software is maintained to ensure that it continues to meet the customer's needs and expectations.

Agile Model

Advantages Agile Software Development

- Deployment of software is quicker and thus helps in increasing the trust of the customer.
- Can better adapt to rapidly changing requirements and respond faster.
- Helps in getting immediate feedback which can be used to improve the software in the next increment.
- People – Not Process. People and interactions are given a higher priority than processes and tools.
- Continuous attention to technical excellence and good design.
- Agile Software Development Methodology emphasize collaboration and communication among team members, stakeholders, and customers. This leads to improved understanding, better alignment, and increased buy-in from everyone involved.

Agile Model

Advantages Agile Software Development

- Agile methodologies are designed to be flexible and adaptable, making it easier to respond to changes in requirements, priorities, or market conditions. This allows teams to quickly adjust their approach and stay focused on delivering value.
- Agile methodologies place a strong emphasis on testing, quality assurance, and continuous improvement. This helps to ensure that software is delivered with high quality and reliability, reducing the risk of defects or issues that can impact the user experience.
- Agile methodologies prioritize customer satisfaction and focus on delivering value to the customer. By involving customers throughout the development process, teams can ensure that the software meets their needs and expectations.
- Agile methodologies promote a collaborative, supportive, and positive work environment. This can lead to increased team morale, motivation, and engagement, which can in turn lead to better productivity, higher quality work, and improved outcomes.

Requirements Analysis and Specifications

Before we start to develop our software, it becomes quite essential for us to understand and document the exact requirement of the customer. Experienced members of the development team carry out this job. They are called as **system analysts**.

The analyst starts *requirements gathering and analysis* activity by collecting all information from the customer which could be used to develop the requirements of the system. He then analyzes the collected information to obtain a clear and thorough understanding of the product to be developed, with a view to remove all ambiguities and inconsistencies from the initial customer perception of the problem.

Requirements Analysis and Specifications

The W/H Questions:

The following basic questions pertaining to the project should be clearly understood by the analyst in order to obtain a good grasp of the problem:

- What is the problem?
- Why is it important to solve the problem?
- What are the possible solutions to the problem?
- What exactly are the data input to the system and what exactly are the data output by the system?
- What are the likely complexities that might arise while solving the problem?
- If there are external software or hardware with which the developed software has to interface, then what exactly would the data interchange formats with the external system be?

Requirements Analysis and Specifications

Parts of a SRS document

The important parts of SRS document are:

- Functional requirements of the system
- Non-functional requirements of the system, and
- Goals of implementation

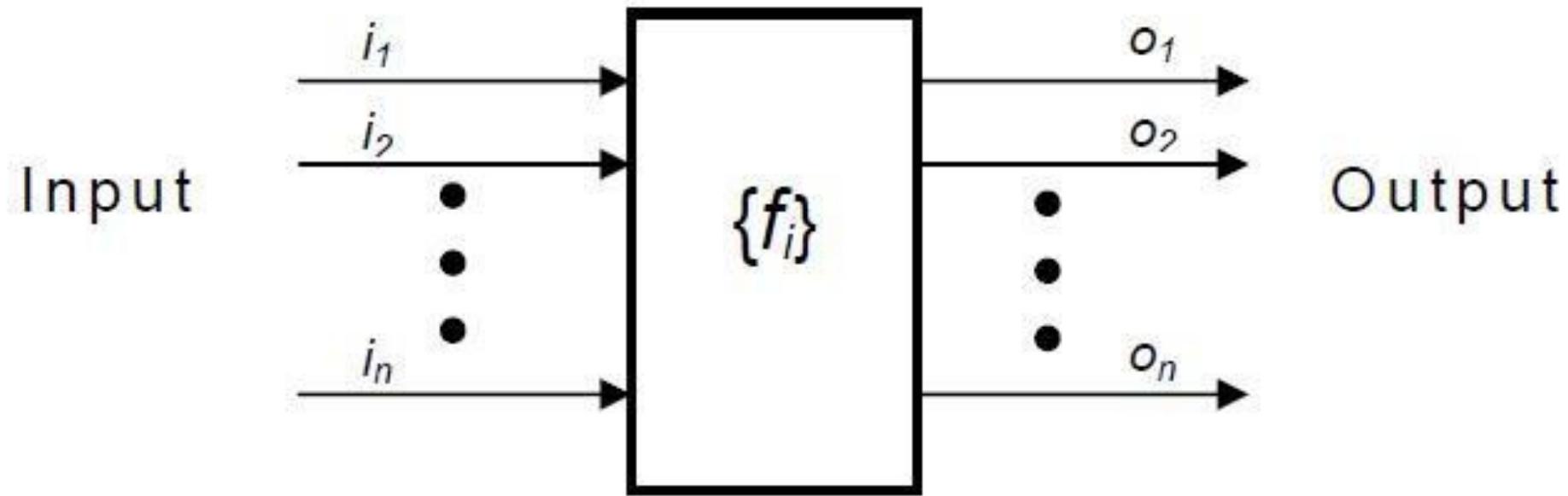
Requirements Analysis and Specifications

Functional requirements

The functional requirements part discusses the functionalities required from the system. The system is considered to perform a set of high-level functions $\{f_i\}$. The functional view of the system is shown in figure next slide. Each function f_i of the system can be considered as a transformation of a set of input data (i) to the corresponding set of output data (o_i). The user can get some meaningful piece of work done using a high-level function.

Requirements Analysis and Specifications

A system performing a set of functions



Requirements Analysis and Specifications

Nonfunctional requirements

Nonfunctional requirements deal with the characteristics of the system which cannot be expressed as functions - such as the maintainability of the system, portability of the system, usability of the system, etc.

Goals of implementation

The goals of implementation part documents some general suggestions regarding development. These suggestions guide trade-off among design goals. The goals of implementation section might document issues such as revisions to the system functionalities that may be required in the future, new devices to be supported in the future, reusability issues, etc. These are the items which the developers might keep in their mind during development so that the developed system may meet some aspects that are not required immediately.

Requirements Analysis and Specifications

Identifying functional requirements from a problem description

The high-level functional requirements often need to be identified either from an informal problem description document or from a conceptual understanding of the problem. Each high level requirement characterizes a way of system usage by some user to perform some meaningful piece of work. There can be many types of users of a system and their requirements from the system may be very different. So, it is often useful to identify the different types of users who might use the system and then try to identify the requirements from each user's perspective.

Requirements Analysis and Specifications

Example: -

Consider the case of the library system, where –

F1: Search Book function

Input: an author's name

Output: details of the author's books and the location of these books in the library

The function Search Book (F1) takes the author's name and transforms it into book details.

Functional requirements actually describe a set of high-level requirements, where each high-level requirement takes some data from the user and provides some data to the user as an output. Also each high-level requirement might consist of several other functions.

Requirements Analysis and Specifications

Documenting functional requirements

For documenting the functional requirements, we need to specify the set of functionalities supported by the system. A function can be specified 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. Let us first try to document the withdraw-cash function of an ATM (Automated Teller Machine) system. The withdraw-cash is a high-level requirement. It has several sub-requirements corresponding to the different user interactions. These different interaction sequences capture the different scenarios.

Requirements Analysis and Specifications

Example: - Withdraw Cash from ATM

R1: 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.

R1.1 select withdraw amount option

Input: “withdraw amount” option

Output: user prompted to enter the account type

Requirements Analysis and Specifications

R1.2: select account type

Input: user option

Output: prompt to enter amount

R1.3: 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

Requirements Analysis and Specifications

Properties of a good SRS document

The important properties of a good SRS document are the following:

Concise: The SRS document should be concise and at the same time unambiguous, consistent, and complete. Verbose and irrelevant descriptions reduce readability and also increase error possibilities.

Structured: It should be well-structured. A well-structured document is easy to understand and modify. In practice, the SRS document undergoes several revisions to cope up with the customer requirements. Often, the customer requirements evolve over a period of time. Therefore, in order to make the modifications to the SRS document easy, it is important to make the document well-structured.

Requirements Analysis and Specifications

Properties of a good SRS document

Black-box view: It should only specify what the system should do and refrain from stating how to do these. This means that the SRS document should specify the external behavior of the system and not discuss the implementation issues. The SRS document should view the system to be developed as black box, and should specify the externally visible behavior of the system. For this reason, the SRS document is also called the black-box specification of a system.

Conceptual integrity: It should show conceptual integrity so that the reader can easily understand it.

Requirements Analysis and Specifications

Properties of a good SRS document

Response to undesired events: It should characterize acceptable responses to undesired events. These are called system response to exceptional conditions.

Verifiable: All requirements of the system as documented in the SRS document should be verifiable. This means that it should be possible to determine whether or not requirements have been met in an implementation.

Requirements Analysis and Specifications

Problems without a SRS document

The important problems that an organization would face if it does not develop a SRS document are as follows:

- Without developing the SRS document, the system would not be implemented according to customer needs.
- Software developers would not know whether what they are developing is what exactly required by the customer.
- Without SRS document, it will be very much difficult for the maintenance engineers to understand the functionality of the system.
- It will be very much difficult for user document writers to write the users' manuals properly without understanding the SRS document.

Requirements Analysis and Specifications

Problems with an unstructured specification

- It would be very much difficult to understand that document.
- It would be very much difficult to modify that document.
- Conceptual integrity in that document would not be shown.
- The SRS document might be unambiguous and inconsistent.

Decision Tree Decision Table

Decision Tree

A decision tree gives a graphic view of the processing logic involved in decision making and the corresponding actions taken. The edges of a decision tree represent conditions and the leaf nodes represent the actions to be performed depending on the outcome of testing the condition.

Example: -

Consider Library Membership Automation Software (LMS) where it should support the following three options:

- **New member**
- **Renewal**
- **Cancel membership**

Decision Tree

New member option Decision:

Decision: When the 'new member' option is selected, the software asks details about the member like the member's name, address, phone number etc.

Action: If proper information is entered then a membership record for the member is created and a bill is printed for the annual membership charge plus the security deposit payable.

Renewal option Decision:

Decision: If the 'renewal' option is chosen, the LMS asks for the member's name and his membership number to check whether he is a valid member or not.

Action: If the membership is valid then membership expiry date is updated and the annual membership bill is printed, otherwise an error message is displayed.

Decision Tree

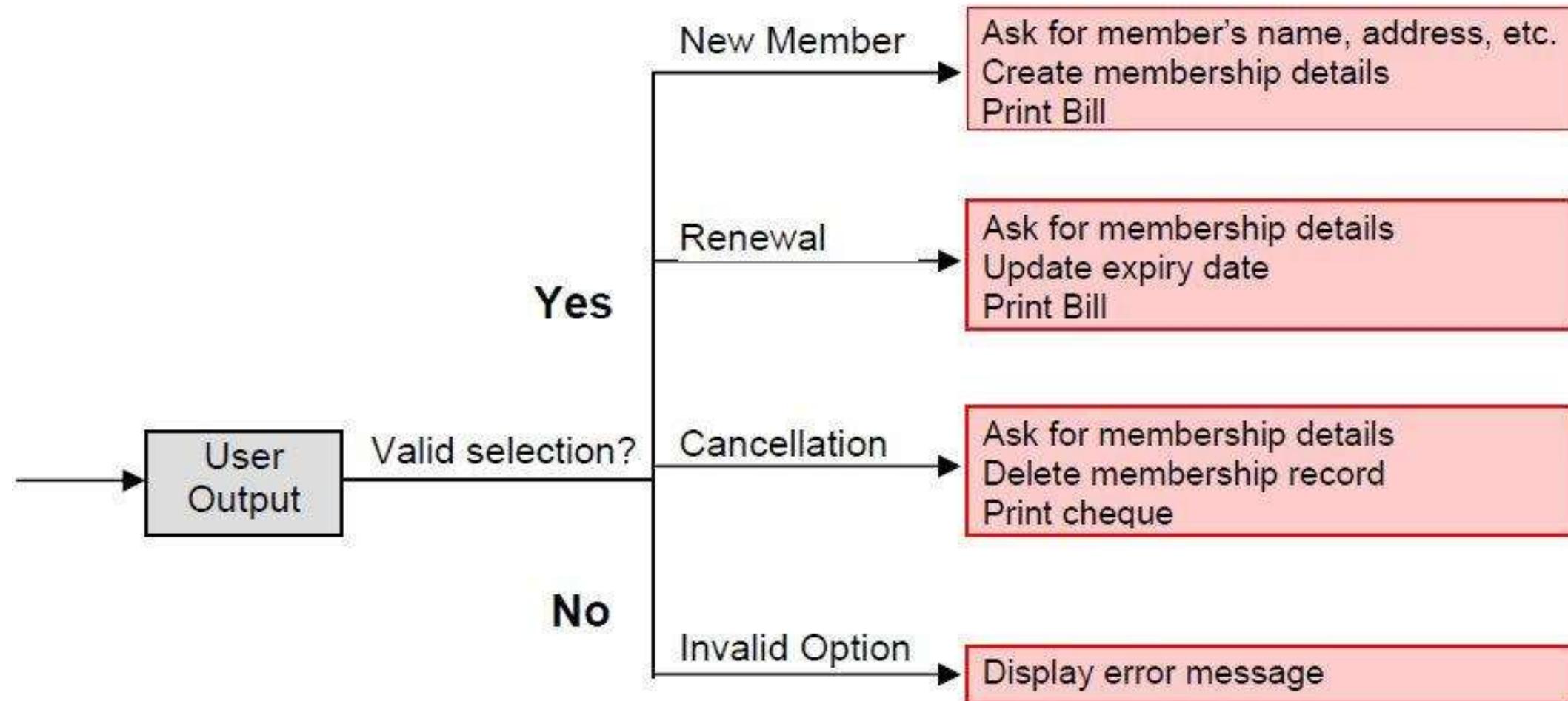
Cancel membership option :

Decision: If the 'cancel membership' option is selected, then the software asks for member's name and his membership number.

Action: The membership is cancelled, a cheque for the balance amount due to the member is printed and finally the membership record is deleted from the database.

Decision Tree

Decision Tree of LMS



Decision Table

A decision table is used to represent the complex processing logic in a tabular or a matrix form. The upper rows of the table specify the variables or conditions to be evaluated. The lower rows of the table specify the actions to be taken when the corresponding conditions are satisfied. A column in a table is called a *rule*. A rule implies that if a condition is true, then the corresponding action is to be executed.

Example:

Consider the previously discussed LMS example. The following decision table shows how to represent the LMS problem in a tabular form. Here the table is divided into two parts, the upper part shows the conditions and the lower part shows what actions are taken. Each column of the table is a rule.

Decision Table

Decision table for LMS

Conditions

Valid selection	No	Yes	Yes	Yes
New member	-	Yes	No	No
Renewal	-	No	Yes	No
Cancellation	-	No	No	Yes

Actions

Display error message	X	-	-	-
Ask member's details	-	X	-	-
Build customer record	-	X	-	-
Generate bill	-	X	X	-
Ask member's name & membership number	-	-	X	X
Update expiry date	-	-	X	-
Print cheque	-	-	-	X
Delete record	-	-	-	X

Decision Table

From the above table we can easily understand that, if the valid selection condition is false then the action taken for this condition is 'display error message'. Similarly, the actions taken for other conditions can be inferred from the table.

SOFTWARE DESIGN

Software Design

- Software design is a process to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.
- For assessing user requirements, an SRS (Software Requirement Specification) document is created. Whereas for coding and implementation, there is a need of more specific and detailed requirements in software terms. The output of this process can directly be used into implementation in programming languages.
- Software design is the first step in SDLC (Software Design Life Cycle), which moves the concentration from problem domain to solution domain. It tries to specify how to fulfill the requirements mentioned in SRS.

Software Design

Software Design Levels

Architectural Design - The architectural design is the highest abstract version of the system. It identifies the software as a system with many components interacting with each other. At this level, the designers get the idea of proposed solution domain.

High-level Design- The high-level design breaks the ‘single entity-multiple component’ concept of architectural design into less-abstracted view of sub-systems and modules and depicts their interaction with each other. High-level design focuses on how the system along with all of its components can be implemented in forms of modules. It recognizes modular structure of each sub-system and their relation and interaction among each other.

Software Design

Software Design Levels

Detailed Design- Detailed design deals with the implementation part of what is seen as a system and its sub-systems in the previous two designs. It is more detailed towards modules and their implementations. It defines logical structure of each module and their interfaces to communicate with other modules.

Software Design

Modularization

- Modularization is a technique to divide a software system into multiple discrete and independent modules, which are expected to be capable of carrying out task(s) independently. These modules may work as basic constructs for the entire software. Designers tend to design modules such that they can be executed and/or compiled separately and independently.
- Modular design unintentionally follows the rules of ‘divide and conquer’ problem-solving strategy. This is because there are many other benefits attached with the modular design of a software.

Software Design

Concurrency

- Back in time, all softwares were meant to be executed sequentially. By sequential execution we mean that the coded instruction will be executed one after another implying only one portion of program being activated at any given time. Say, a software has multiple modules, then only one of all the modules can be found active at any time of execution.
- In software design, concurrency is implemented by splitting the software into multiple independent units of execution, like modules and executing them in parallel. In other words, concurrency provides capability to the software to execute more than one part of code in parallel to each other.
- It is necessary for the programmers and designers to recognize those modules, which can be made parallel execution.

Software Design

Coupling and Cohesion

When a software program is modularized, its tasks are divided into several modules based on some characteristics. As we know, modules are set of instructions put together in order to achieve some tasks. They are though, considered as single entity but may refer to each other to work together. There are measures by which the quality of a design of modules and their interaction among them can be measured. These measures are called coupling and cohesion.

Software Design

Cohesion

Cohesion is a measure that defines the degree of intra-dependability within elements of a module. The greater the cohesion, the better is the program design.

There are seven types of cohesion, namely –

Co-incidental cohesion - It is unplanned and random cohesion, which might be the result of breaking the program into smaller modules for the sake of modularization. Because it is unplanned, it may serve confusion to the programmers and is generally not-accepted.

Logical cohesion - When logically categorized elements are put together into a module, it is called logical cohesion.

Temporal Cohesion - When elements of module are organized such that they are processed at a similar point in time, it is called temporal cohesion.

Procedural cohesion - When elements of module are grouped together, which are executed sequentially in order to perform a task, it is called procedural cohesion.

Software Design

Cohesion

Communicational cohesion - When elements of module are grouped together, which are executed sequentially and work on same data (information), it is called communicational cohesion.

Sequential cohesion - When elements of module are grouped because the output of one element serves as input to another and so on, it is called sequential cohesion.

Functional cohesion - It is considered to be the highest degree of cohesion, and it is highly expected. Elements of module in functional cohesion are grouped because they all contribute to a single well-defined function. It can also be reused.

Software Design

Coupling

Coupling is a measure that defines the level of inter-dependability among modules of a program. It tells at what level the modules interfere and interact with each other. The lower the coupling, the better the program.

There are five levels of coupling, namely -

Content coupling - When a module can directly access or modify or refer to the content of another module, it is called content level coupling.

Common coupling- When multiple modules have read and write access to some global data, it is called common or global coupling.

Software Design

Coupling

Control coupling- Two modules are called control-coupled if one of them decides the function of the other module or changes its flow of execution.

Stamp coupling- When multiple modules share common data structure and work on different part of it, it is called stamp coupling.

Data coupling- Data coupling is when two modules interact with each other by means of passing data (as parameter). If a module passes data structure as parameter, then the receiving module should use all its components.

Ideally, no coupling is considered to be the best.

Software Design

Design Verification

- The output of software design process is design documentation, pseudo codes, detailed logic diagrams, process diagrams, and detailed description of all functional or non-functional requirements.
- The next phase, which is the implementation of software, depends on all outputs mentioned above.
- It is then becomes necessary to verify the output before proceeding to the next phase. The earlier any mistake is detected, the better it is or it might not be detected until testing of the product.
- By structured verification approach, reviewers can detect defects that might be caused by overlooking some conditions. A good design review is important for good software design, accuracy and quality.

Software Design Strategies

Software Design Strategies

Software design is a process to conceptualize the software requirements into software implementation. Software design takes the user requirements as challenges and tries to find optimum solution. While the software is being conceptualized, a plan is chalked out to find the best possible design for implementing the Intended solution.

There are multiple variants of software design.

Structured Design

- Structured design is a conceptualization of problem into several well-organized elements of solution. It is basically concerned with the solution design. Benefit of structured design is, it gives better understanding of how the problem is being solved. Structured design also makes it simpler for designer to concentrate on the problem more accurately.
- Structured design is mostly based on ‘divide and conquer’ strategy where a problem is broken into several small problems and each small problem is individually solved until the whole problem is solved.
- The small pieces of problem are solved by means of solution modules. Structured design emphasis that these modules be well organized in order to achieve precise solution.

Structured Design

These modules are arranged in hierarchy. They communicate with each other. A good structured design always follows some rules for communication among multiple modules, namely -

- **Cohesion** - grouping of all functionally related elements.
- **Coupling** - communication between different modules.

A good structured design has *high* cohesion and *low* coupling arrangements.

Function Oriented Design

- In function-oriented design, the system is comprised of many smaller sub-systems known as functions. These functions are capable of performing significant task in the system. The system is considered as top view of all functions.
- Function oriented design inherits some properties of structured design where divide and conquer methodology is used.
- This design mechanism divides the whole system into smaller functions, which provides means of abstraction by concealing the information and their operation. These functional modules can share information among themselves by means of information passing and using information available globally.
- Another characteristic of functions is that when a program calls a function, the function changes the state of the program, which sometimes is not acceptable by other modules. Function oriented design works well where the system state does not matter and program/functions work on input rather than on a state.

Function Oriented Design

Design Process

- The entire system is logically broken down into smaller units known as functions on the basis of their operation in the system.
- The whole system is seen as how data flows in the system by means of data flow diagram.
- DFD depicts how functions change the data and state of entire system.
- Each function is then described at large.

Object Oriented Design

Object oriented design works around the entities and their characteristics instead of functions involved in the software system. This design strategy focuses on entities and its characteristics. The whole concept of software solution revolves around the engaged entities.

The important concepts of Object Oriented Design:

- **Objects** - All entities involved in the solution design are known as objects. For example, person, banks, company and customers are treated as objects. Every entity has some attributes associated to it and has some methods to perform on the attributes.
- **Classes** - A class is a generalized description of an object. An object is an instance of a class. Class defines all the attributes, which an object can have and methods, which defines the functionality of the object.

Object Oriented Design

In the solution design, attributes are stored as variables and functionalities are defined by means of methods or procedures. Some features of Object Oriented Designs are-

- **Encapsulation** - In OOD, the attributes (data variables) and methods (operation on the data) are bundled together is called encapsulation. Encapsulation not only bundles important information of an object together, but also restricts access of the data and methods from the outside world. This is called information hiding.
- **Inheritance** - OOD allows similar classes to stack up in hierarchical manner where the lower or sub-classes can import, implement and re-use allowed variables and methods from their immediate super classes. This property of OOD is known as inheritance. This makes it easier to define specific class and to create generalized classes from specific ones.
- **Polymorphism** - OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned same name. This is called polymorphism, which allows a single interface performing tasks for different types. Depending upon how the function is invoked, respective portion of the code gets executed.

Object Oriented Design

Design Process

Software design process can be perceived as series of well-defined steps. Though it varies according to design approach (function oriented or object oriented), yet It may have the following steps involved:

- A solution design is created from requirement or previous used system and/or system sequence diagram.
- Objects are identified and grouped into classes on behalf of similarity in attribute characteristics.
- Class hierarchy and relation among them are defined.
- Application framework is defined.

Software Design Approaches

A **system** is composed of more than one sub-systems and it contains a number of components. Further, these sub-systems and components may have their own set of sub-system and components and creates hierarchical structure in the system.

There are two generic approaches for software designing:

Top down Design

- Top-down design takes the whole software system as one entity and then decomposes it to achieve more than one sub-system or component based on some characteristics. Each subsystem or component is then treated as a system and decomposed further. This process keeps on running until the lowest level of system in the top-down hierarchy is achieved.
- Top-down design starts with a generalized model of system and keeps on defining the more specific part of it. When all components are composed the whole system comes into existence.
- Top-down design is more suitable when the software solution needs to be designed from scratch and specific details are unknown.

Software Design Approaches

Bottom Up Design

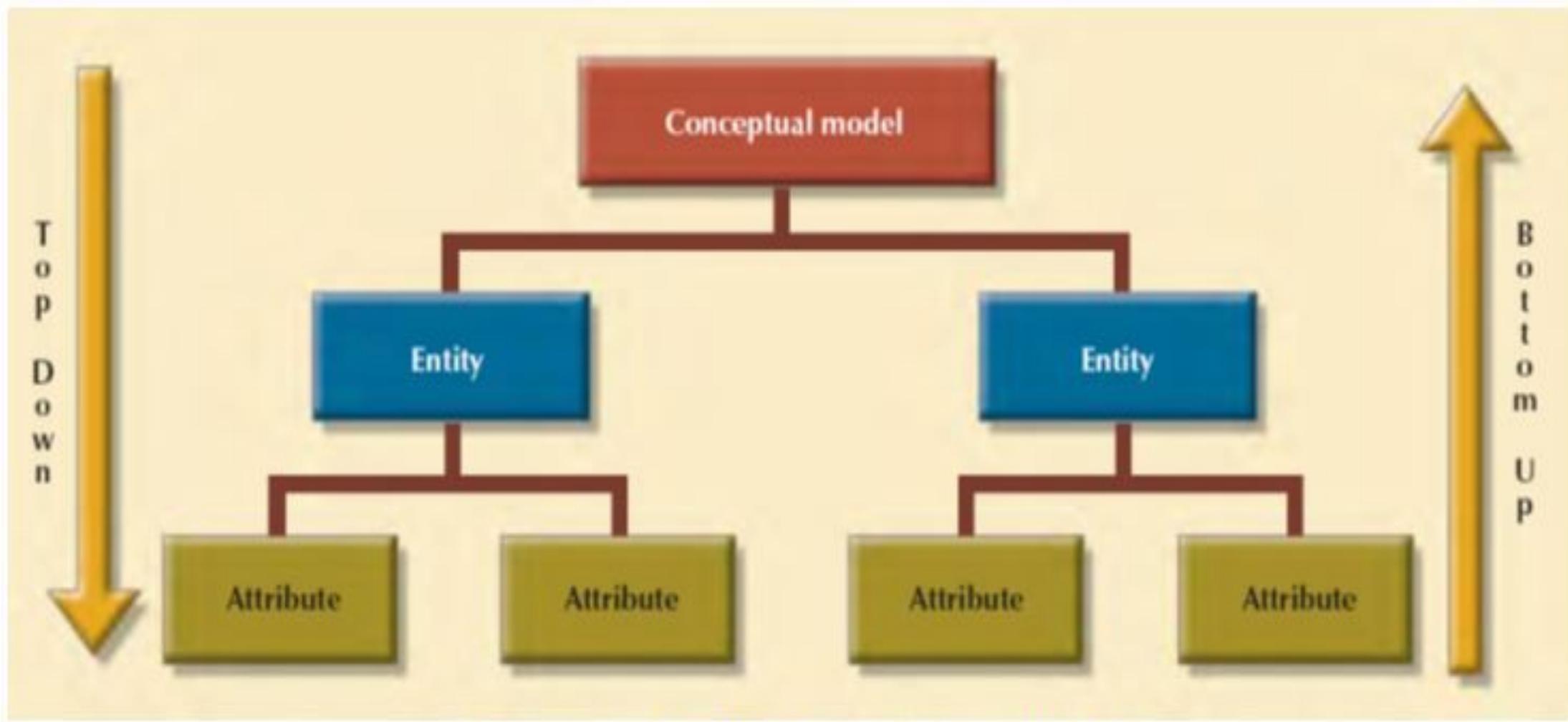
- The bottom up design model starts with most specific and basic components. It proceeds with composing higher level of components by using basic or lower level components. It keeps creating higher level components until the desired system is not evolved as one single component. With each higher level, the amount of abstraction is increased.
- Bottom-up strategy is more suitable when a system needs to be created from some existing system, where the basic primitives can be used in the newer system.
- Both, top-down and bottom-up approaches are not practical individually. Instead, a good combination of both is used.

Software Design Approaches

Bottom Up Design

- The bottom up design model starts with most specific and basic components. It proceeds with composing higher level of components by using basic or lower level components. It keeps creating higher level components until the desired system is not evolved as one single component. With each higher level, the amount of abstraction is increased.
- Bottom-up strategy is more suitable when a system needs to be created from some existing system, where the basic primitives can be used in the newer system.
- Both, top-down and bottom-up approaches are not practical individually. Instead, a good combination of both is used.

Top down Design VS. Bottom Up Design



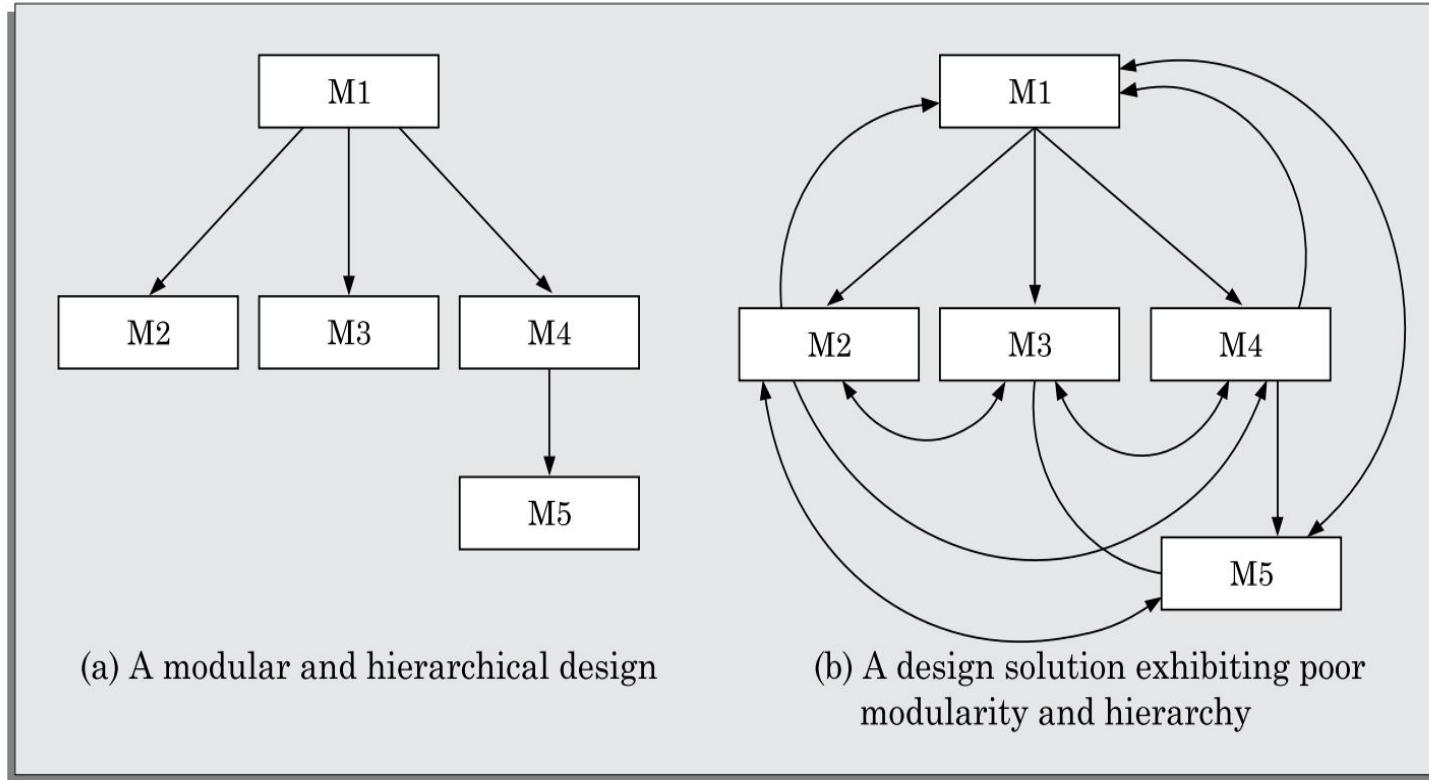
Analysis vs. Design

- Analysis and design activities differ in goal and scope.
- The analysis results are generic and does not consider implementation or the issues associated with specific platforms.
- The analysis model is usually documented using some graphical formalism.
- In case of the function-oriented approach, the analysis model would be documented using *data flow diagrams* (DFDs), whereas the design would be documented using structure chart.
- For object-oriented approach, both the design model and the analysis model will be documented using *unified modelling language* (UML).
- The analysis model would normally be very difficult to implement using a programming language.
- The design model is obtained from the analysis model through transformations over a series of steps.

Comparison of Modularity

- How can we compare the modularity of two alternate design solutions?
- From an inspection of the module structure, it is at least possible to intuitively form an idea as to which design is more modular.
- For example, consider two alternate design solutions to a problem that are represented in Figure given next, in which the modules M_1 , M_2 , etc. have been drawn as rectangles. The invocation of a module by another module has been shown as an arrow.
- It can easily be seen that the design solution of the Figure-(a) would be easier to understand since the interactions among the different modules is low. But, can we quantitatively measure the modularity of a design solution? Unless we are able to quantitatively measure the modularity of a design solution, it will be hard to say which design solution is more modular than another.
- Unfortunately, there are no quantitative metrics available yet to directly measure the modularity of a design. However, we can quantitatively characterize the modularity of a design solution based on the cohesion and coupling existing in the design.

Comparison of Modularity



A software design with high cohesion and low coupling among modules is the effective problem decomposition. Such a design would lead to increased productivity during program development by bringing down the perceived problem complexity.

Layered design

- A layered design is one in which when the call relations among different modules are represented graphically, it would result in a tree-like diagram with clear layering.
- In a layered design solution, the modules are arranged in a hierarchy of layers. A module can only invoke functions of the modules in the layer immediately below it.
- The higher layer modules can be considered to be similar to managers that invoke (order) the lower layer modules to get certain tasks done.
- A layered design can be considered to be implementing *control abstraction*, since a module at a lower layer is unaware of (about how to call) the higher layer modules.
- When a failure is detected while executing a module, it is obvious that the modules below it can possibly be the source of the error. This greatly simplifies debugging since one would need to concentrate only on a few modules to detect the error.

LAYERED ARRANGEMENT OF MODULES

- The *control hierarchy* represents the organisation of program components in terms of their call relationships.
- The control hierarchy of a design is determined by the order in which different modules call each other.
- Many different types of notations have been used to represent the control hierarchy. The most common notation is a treelike diagram known as a *structure chart*.
- In a layered design solution, the modules are arranged into several layers based on their call relationships.
- A module is allowed to call only the modules that are at a lower layer. That is, a module should not call a module that is either at a higher layer or even in the same layer.

LAYERED ARRANGEMENT OF MODULES

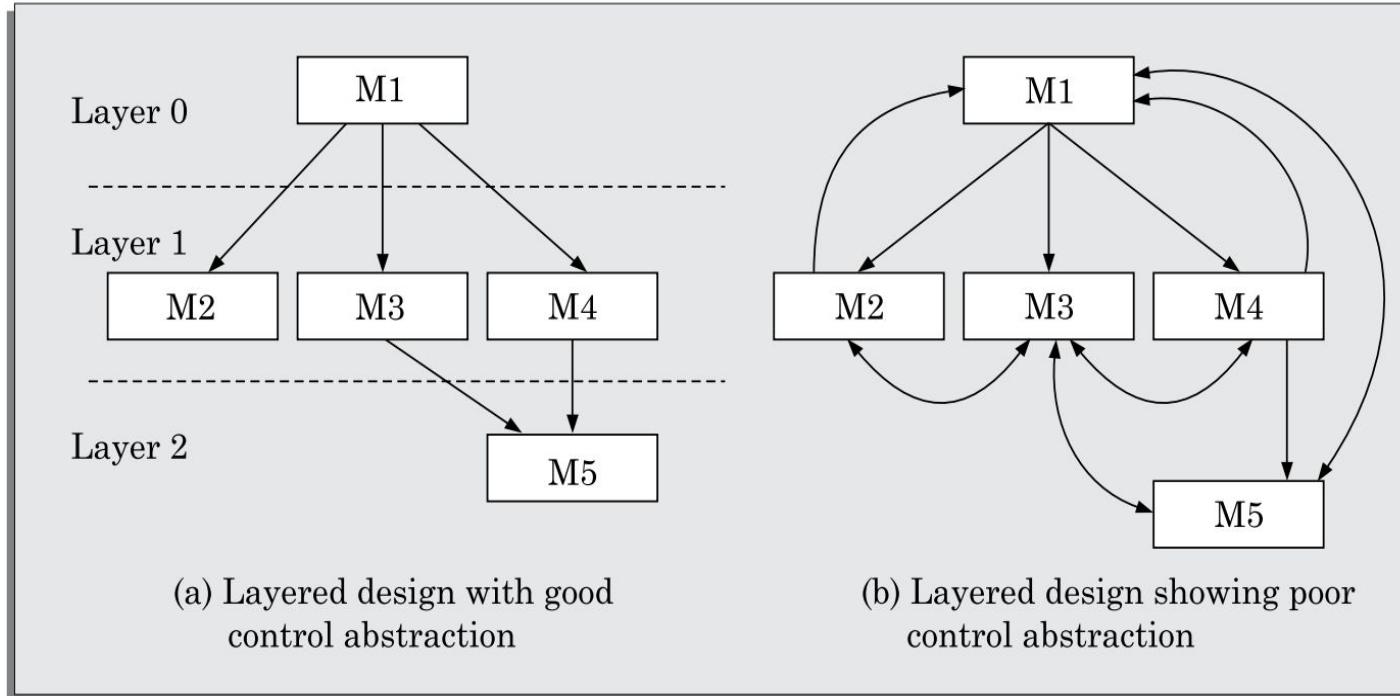


Figure (a) shows a layered design, whereas Figure (b) shows a design that is not layered. The design solution shown in Figure (b), is actually not layered since all the modules can be considered to be in the same layer.

LAYERED ARRANGEMENT OF MODULES

- In a layered design, the top-most module in the hierarchy can be considered as a manager that only invokes the services of the lower level module to discharge its responsibility.
- The modules at the intermediate layers offer services to their higher layer by invoking the services of the lower layer modules and also by doing some work themselves to a limited extent.
- The modules at the lowest layer are the worker modules. These do not invoke services of any module and entirely carry out their responsibilities by themselves.

LAYERED ARRANGEMENT OF MODULES

Terminologies associated with a layered design:

- **Superordinate and subordinate modules:** In a control hierarchy, a module that controls another module is said to be *superordinate* to it. Conversely, a module controlled by another module is said to be *subordinate* to the controller.
- **Visibility:** A module B is said to be visible to another module A, if A directly calls B. Thus, only the immediately lower layer modules are said to be visible to a module.
- **Control abstraction:** In a layered design, a module should only invoke the functions of the modules that are in the layer immediately below it. In other words, the modules at the higher layers, should not be visible (that is, abstracted out) to the modules at the lower layers. This is referred to as *control abstraction*.
- **Depth and width:** Depth and width of a control hierarchy provide an indication of the number of layers and the overall span of control respectively. For the design of Figure 5.6(a), the depth is 3 and width is also 3.

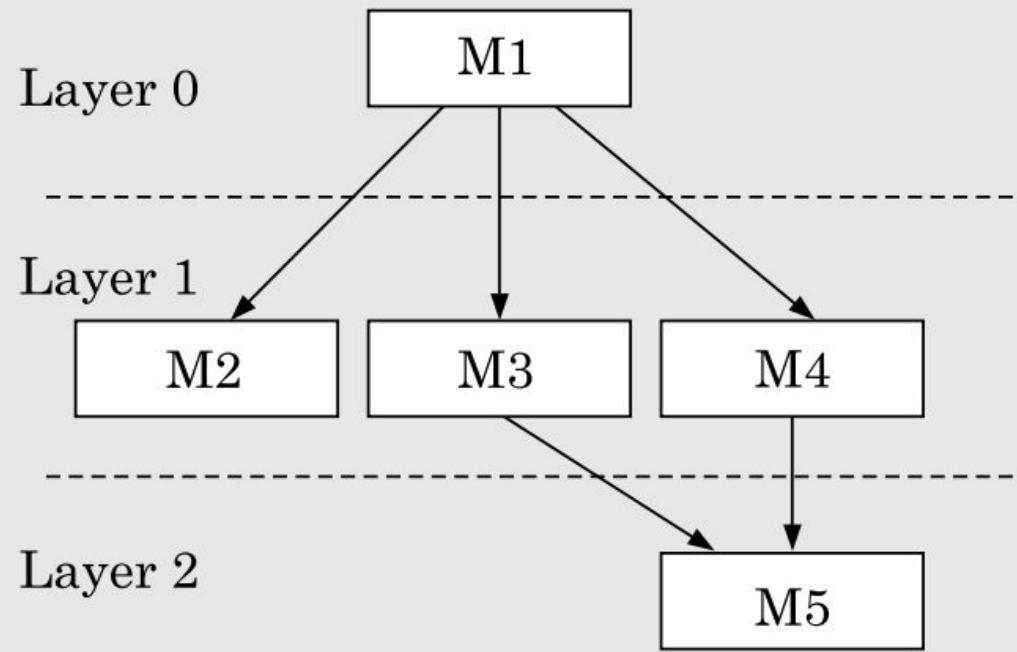
LAYERED ARRANGEMENT OF MODULES

Terminologies associated with a layered design:

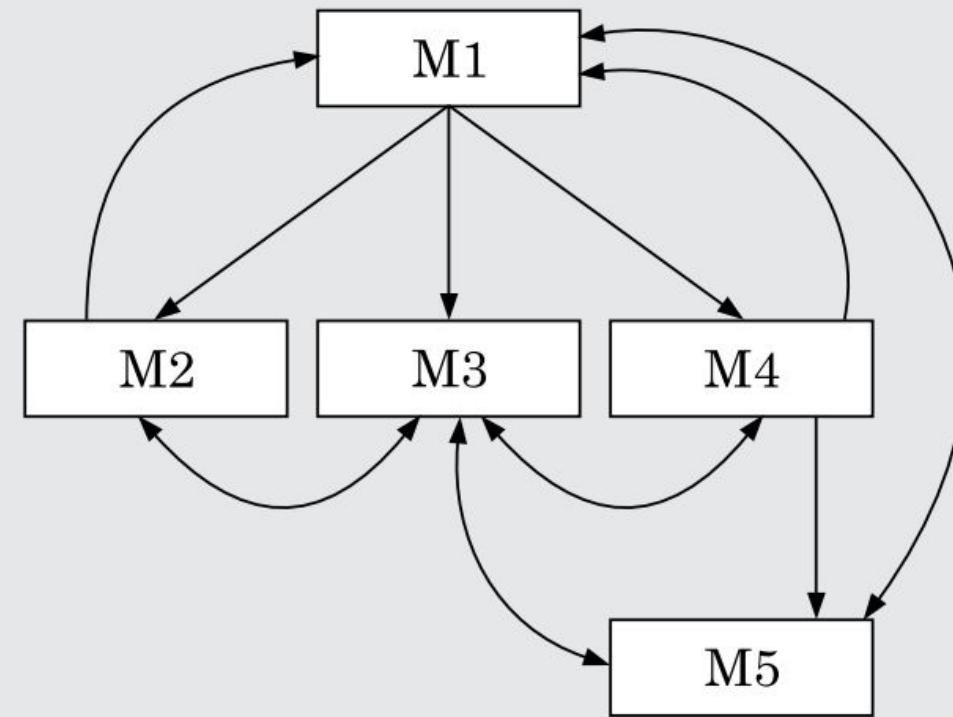
- **Fan-out:** Fan-out is a measure of the number of modules that are directly controlled by a given module. In Figure (a), the fan-out of the module M1 is 3. A design in which the modules have very high fan-out numbers is not a good design. The reason for this is that a very high fan-out is an indication that the module lacks cohesion. A module having a large fan-out (greater than 7) is likely to implement several different functions and not just a single cohesive function.
- **Fan-in:** Fan-in indicates the number of modules that directly invoke a given module. High fan-in represents code reuse and is in general, desirable in a good design. In Figure (a), the fan-in of the module M1 is 0, that of M2 is 1, and that of M5 is 2.

LAYERED ARRANGEMENT OF MODULES

Terminologies associated with a layered design:



(a) Layered design with good control abstraction



(b) Layered design showing poor control abstraction

Functional Independence

By the terms *Functional independence*, we mean that a module performs a single task and needs very little interaction with other modules.

Functional independence is a key to any good design primarily due to the following advantages it offers:

Error isolation: Whenever an error exists in a module, functional independence reduces the chances of the error propagating to the other modules. The reason behind this is that if a module is functionally independent, its interaction with other modules is low. Therefore, an error existing in the module is very unlikely to affect the functioning of other modules.

Further, once a failure is detected, error isolation makes it very easy to locate the error. On the other hand, when a module is not functionally independent, once a failure is detected in a functionality provided by the module, the error can be potentially in any of the large number of modules and propagated to the functioning of the module.

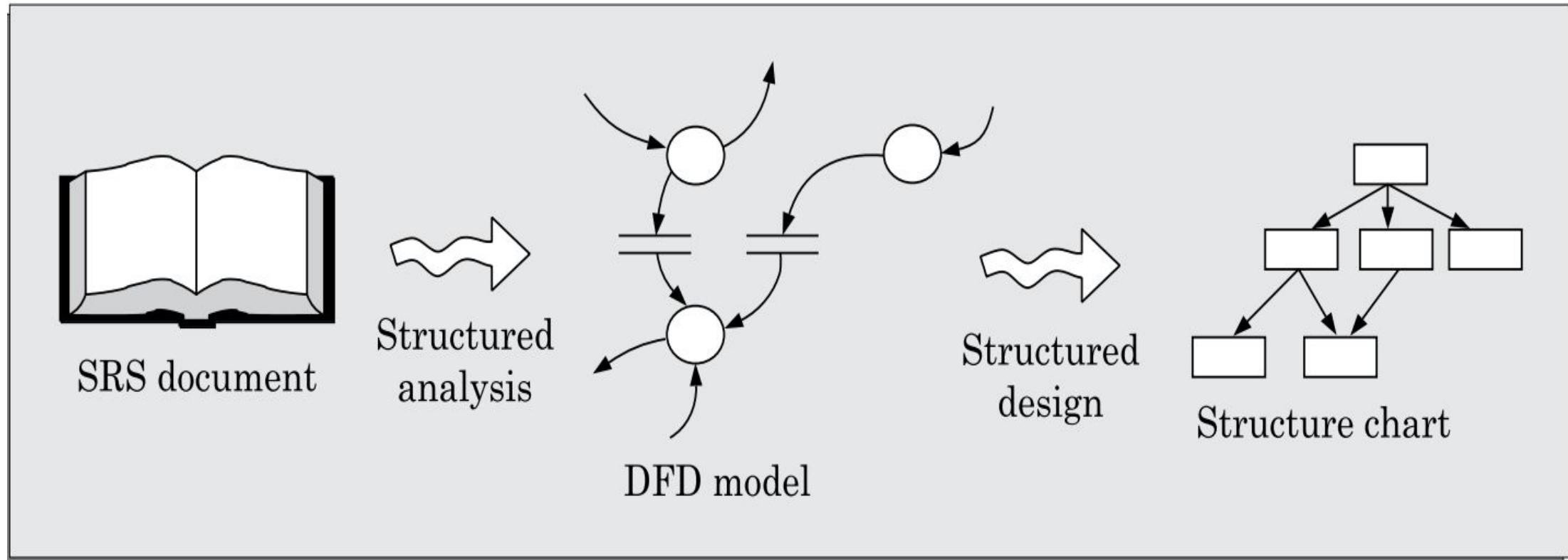
Scope of reuse: Reuse of a module for the development of other applications becomes easier. The reasons for this is as follows. A functionally independent module performs some well-defined and precise task and the interfaces of the module with other modules are very few and simple. A functionally independent module can therefore be easily taken out and reused in a different program. On the other hand, if a module interacts with several other modules or the functions of a module perform very different tasks, then it would be difficult to reuse it.

Function-Oriented Software Design

Function-Oriented Software Design

- Function-oriented design techniques were proposed nearly four decades ago.
- These techniques still very popular and are currently being used in many are at the present time software development projects.
- 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 provided by a software to its users are also known as the high-level functions supported by the software. During the design process, these high-level functions are successively decomposed into more detailed functions.
- After top-down decomposition has been carried out, the different identified functions are mapped to modules and a module structure is created.

SA/SD Design Methodology



- During structured analysis, the SRS document is transformed into a *data flow diagram* (DFD) model.
- During structured design, the DFD model is transformed into a structure chart.

Structured Analysis

- The structured analysis activity transforms the SRS document into a graphic model called the DFD model.
- During structured analysis, the major processing tasks (**high-level functions**) of the system are analyzed, and the data flow among these processing tasks are represented graphically.
- 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).

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.
- DFD representation of a problem is very easy to construct.
- It is a very powerful tool to tackle the complexity of industry standard problems.
- DFD model only represents the data flow aspects and does not show the sequence of execution of the different functions and the conditions based on which a function may or may not be executed.
- It completely ignores aspects such as control flow, the specific algorithms used by the functions, etc. In the DFD terminology, each function is called a process or a *bubble*.
- There is a prominent difference between DFD and Flowchart. The flowchart depicts flow of control in program modules. DFDs depict flow of data in the system at various levels. DFD does not contain any control or branch elements.

DFD

Symbols used for constructing DFDs

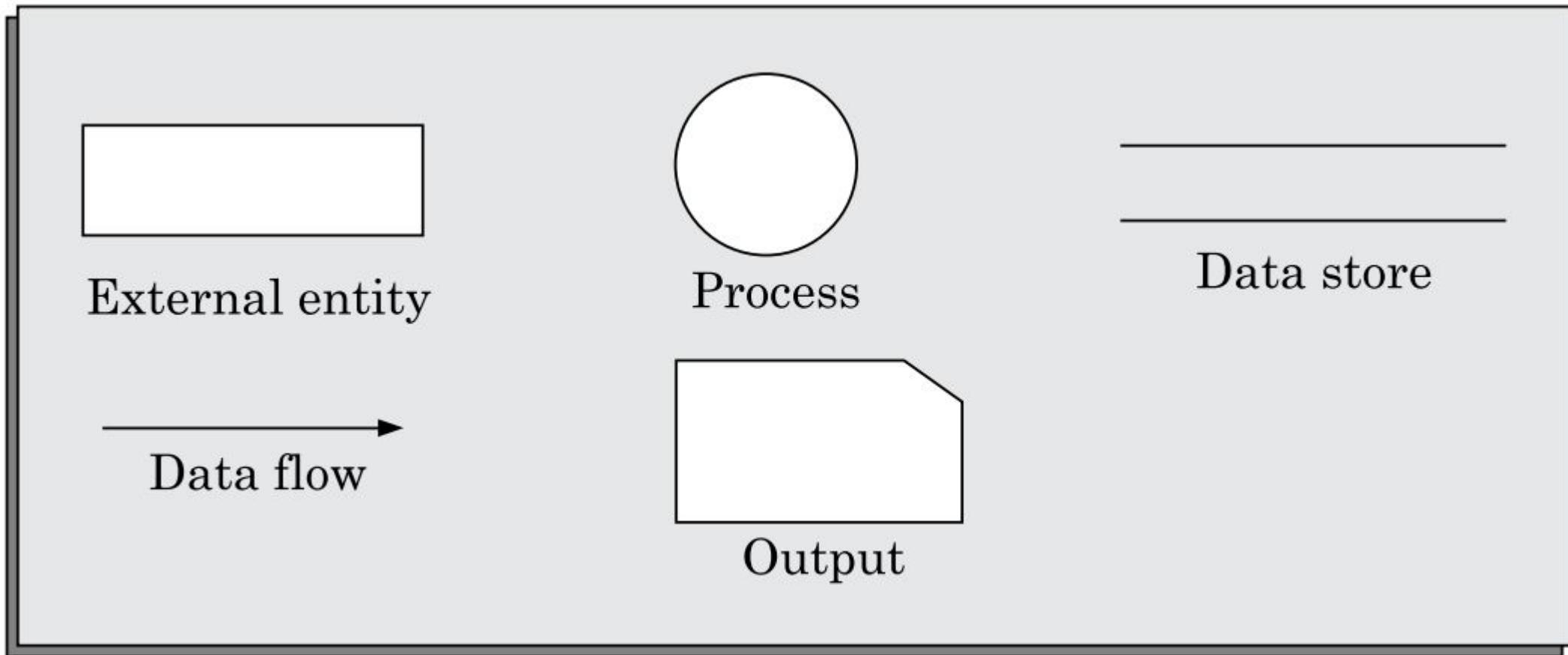
Function symbol: A function is represented using a circle. This symbol is called a *process* or a *bubble*. Bubbles are annotated with the names of the corresponding functions.

External entity symbol: An external entity is represented by a rectangle. The external entities are essentially those physical entities external to the software system which interact with the system by inputting data to the system or by consuming the data produced by the system. In addition to the human users, the external entity symbols can be used to represent external hardware and software such as another application software that would interact with the software being modelled.

Data flow symbol: A directed arc (or an arrow) is used as a data flow symbol. A data flow symbol represents the data flow occurring between two processes or between an external entity and a process in the direction of the data flow arrow. Data flow symbols are usually annotated with the corresponding data names.

DFD

Symbols used for constructing DFDs



DFD

Data store symbol: A data store is represented using two parallel lines. A data store symbol can represent either a data structure or a physical file on disk. Each data store is connected to a process by means of a data flow symbol. The direction of the data flow arrow shows whether data is being read from or written into a data store. An arrow flowing in or out of a data store implicitly represents the entire data of the data store and hence arrows connecting to a data store need not be annotated with the name of the corresponding data items.

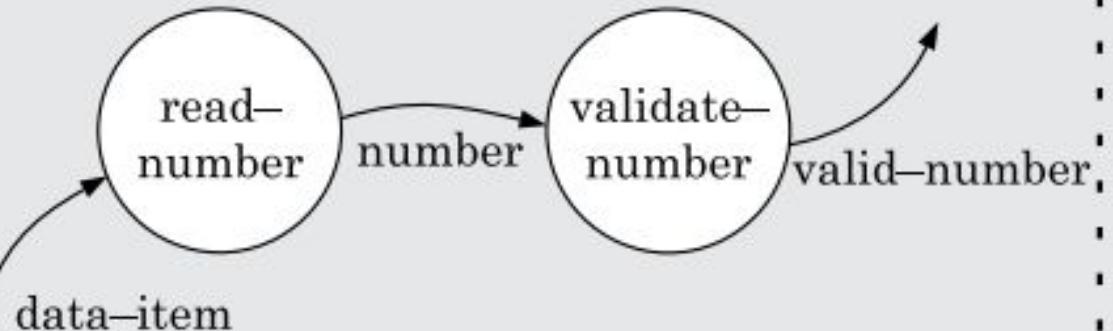
Output symbol: The output symbol is as shown in figure. The output symbol is used when a hard copy is produced.

Synchronous VS. Asynchronous Operations

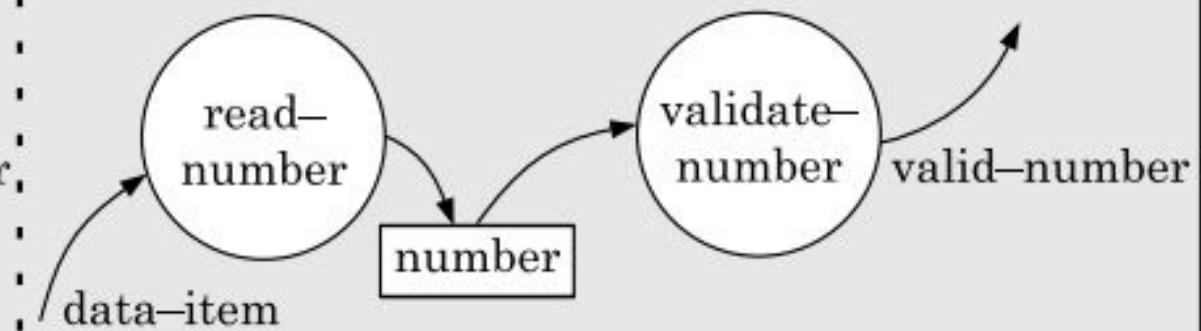
Synchronous: If two bubbles are directly connected by a data flow arrow, then they are synchronous. This means that they operate at the same speed. An example of such an arrangement is shown in Figure-(a). Here, the validate-number bubble can start processing only after the read-number bubble has supplied data to it; and the read-number bubble has to wait until the validate-number bubble has consumed its data.

Asynchronous: if two bubbles are connected through a data store, as in Figure-(b) then the speed of operation of the bubbles are independent. This statement can be explained using the following reasoning. The data produced by a producer bubble gets stored in the data store. It is therefore possible that the producer bubble stores several pieces of data items, even before the consumer bubble consumes any of them.

Synchronous VS. Asynchronous Operations



(a) Synchronous operation of two bubbles



(b) Asynchronous operation of two bubbles

Figure-(a).

Figure-(b).

DFD

Importance of DFD

DFD is a **very simple formalism** – it is simple to understand and use.

Starting with a set of high-level functions that a system performs, a DFD model hierarchically represents various simple to understand sub-functions.

Human mind is such that it can easily understand any hierarchical model of a system – because in a hierarchical model, starting with a very simple and abstract model of a system, different details of the system are slowly introduced through different hierarchies.

DFD is an elegant modeling technique that turns out to be useful not only to represent the results of structured analysis of a software problem, but also for several other applications such as showing the flow of documents or items in an organization.

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. The data items listed include all data flows and the contents of all data stores appearing on all the DFDs in a DFD model. However, a single data dictionary should capture all the data appearing in all the DFDs constituting the DFD model of a system.

Importance of Data Dictionary

- A data dictionary provides a standard terminology for all relevant data for use by the developers working in a project. A consistent vocabulary for data items is very important, since in large projects different developers of the project have a tendency to use different terms to refer to the same data.
- The data dictionary helps the developers to determine the definition of different data structures in terms of their component elements while implementing the design.
- The data dictionary helps to perform impact analysis. That is, it is possible to determine the effect of some data on various processing activities and *vice versa*.

Data Definition

Composite data items can be defined in terms of primitive data items using the following data definition operators.

+: denotes composition of two data items, e.g. **a+b** represents data **a** and **b**.

[,]: represents selection, i.e. any one of the data items listed in the brackets can occur. For example, **[a,b]** represents either **a** occurs or **b** occurs.

(): the contents inside the bracket represent optional data which may or may not appear. e.g. **a+(b)** represents either **a** occurs or **a+b** occurs.

{}: represents iterative data definition, e.g. **{name}5** represents five **name** data. **{name}*:** represents zero or more instances of **name** data.

=: represents equivalence, e.g. **a=b+c** means that **a** is composite data item comprising both **b** and **c**.

/* */: Anything appearing within **/*** and ***/** is considered as a comment.

Constructing DFD Model of a System

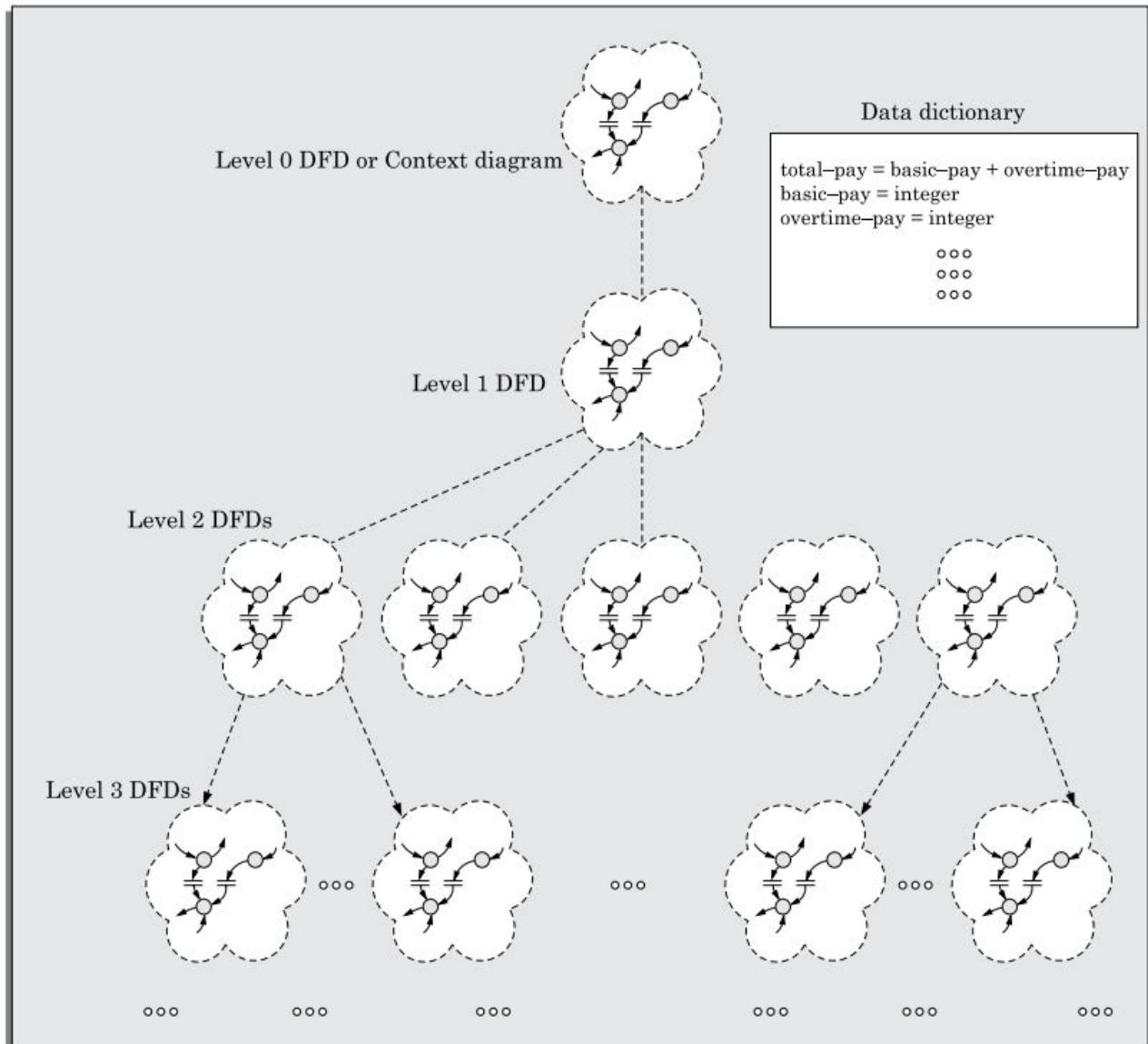
Construction of DFD

- A DFD model of a system graphically represents how each input data is transformed to its corresponding output data through a hierarchy of DFDs.
- The DFD model of a system is constructed by using a hierarchy of DFDs as shown in the figure given next.
- The top level DFD is called the level 0 DFD or the context diagram. This is the most abstract (simplest) representation of the system (highest level). It is the easiest to draw and understand.
- At each successive lower level DFDs, more and more details are gradually introduced.
- To develop a higher-level DFD model, processes are decomposed into their subprocesses and the data flow among these subprocesses are identified.

dharabahi

Construction of DFD

- To develop the data flow model of a system, first the most abstract representation (highest level) of the problem is to be worked out.
- Subsequently, the lower level DFDs are developed.
- Level 0 and Level 1 consist of only one DFD each.
- Level 2 may contain up to 7 separate DFDs, and level 3 up to 49 DFDs, and so on.
- However, there is only a single data dictionary for the entire DFD model.

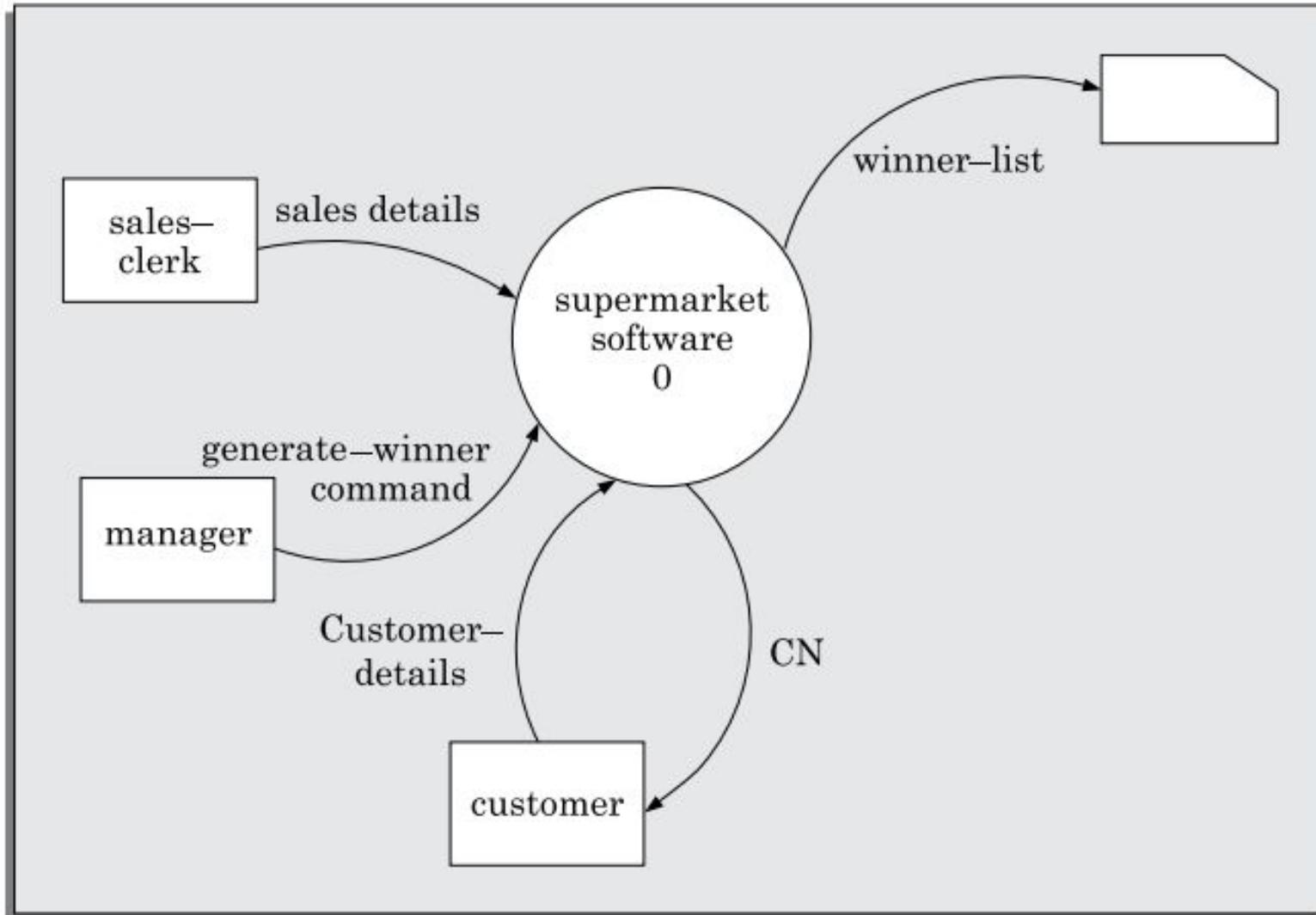


Context Diagram

- The context diagram is the most abstract (highest level) data flow representation of a system. It represents the entire system as a single bubble. The bubble in the context diagram is annotated with the name of the software system being developed (usually a noun). This is the only bubble in a DFD model, where a noun is used for naming the bubble. The bubbles at all other levels are annotated with verbs according to the main function performed by the bubble. This is expected since the purpose of the context diagram is to capture the context of the system rather than its functionality
- As an example of a context diagram, consider the context diagram a software developed to automate the book keeping activities of a supermarket. The context diagram has been labelled as 'Supermarket software'.

Context Diagram

Context diagram for Super Market Software



Level 1 DFD

- The level 1 DFD usually contains three to seven bubbles. That is, the system is represented as performing three to seven important functions.
- To develop the level 1 DFD, examine the high-level functional requirements in the SRS document. If there are three to seven high-level functional requirements, then each of these can be directly represented as a bubble in the level 1 DFD.
- If a system has more than seven high-level requirements identified in the SRS document, then, some of the related requirements have to be combined and represented as a single bubble in the level 1 DFD. These can be split appropriately in the lower DFD levels.
- If a system has less than three high-level functional requirements, then some of the high-level requirements need to be split into their subfunctions so that we have roughly about five to seven bubbles represented on the diagram.

Decomposition

- Each bubble in the DFD represents a function performed by the system. The bubbles are decomposed into subfunctions at the successive levels of the DFD model.
Decomposition of a bubble is also known as *factoring* or *exploding* a bubble.
- Each bubble at any level of DFD is usually decomposed to anything three to seven bubbles. A few bubbles at any level make that level superfluous.
- For example, if a bubble is decomposed to just one bubble or two bubbles, then this decomposition becomes trivial and redundant.
- Too many bubbles (i.e. more than seven bubbles) at any level of a DFD makes the DFD model hard to understand.
- Decomposition of a bubble should be carried on until a level is reached at which the function of the bubble can be described using a simple algorithm.

Construction of context diagram

Examine the SRS document to determine:

- Different high-level functions that the system needs to perform.
- Data input to every high-level function.
- Data output from every high-level function.
- Interactions (data flow) among the identified high-level functions.
- Represent these aspects of the high-level functions in a diagrammatic form. This would form the top-level *data flow diagram* (DFD), usually called the DFD 0.

Construction of level 1 diagram

- Examine the high-level functions described in the SRS document.
- If there are three to seven high-level requirements in the SRS document, then represent each of the high-level function in the form of a bubble.
- If there are more than seven bubbles, then some of them have to be combined.
- If there are less than three bubbles, then some of these have to be split.

Construction of lower-level diagrams

Decompose each high-level function into its constituent subfunctions through the following set of activities:

- Identify the different subfunctions of the high-level function.
- Identify the data input to each of these subfunctions.
- Identify the data output from each of these subfunctions.
- Identify the interactions (data flow) among these subfunctions. Represent these aspects in a diagrammatic form using a DFD.
- Recursively repeat Step 3 for each subfunction until a subfunction can be represented by using a simple algorithm.

Numbering of bubbles

- It is necessary to number the different bubbles occurring in the DFD.
- These numbers help in uniquely identifying any bubble in the DFD from its bubble number.
- The bubble at the context level is usually assigned the number 0 to indicate that it is the 0 level DFD.
- Bubbles at level 1 are numbered, 0.1, 0.2, 0.3, etc.
- When a bubble numbered x is decomposed, its children bubble are numbered $x.1, x.2, x.3$, etc.
- In this numbering scheme, by looking at the number of a bubble we can unambiguously determine its level, its ancestors, and its successors.

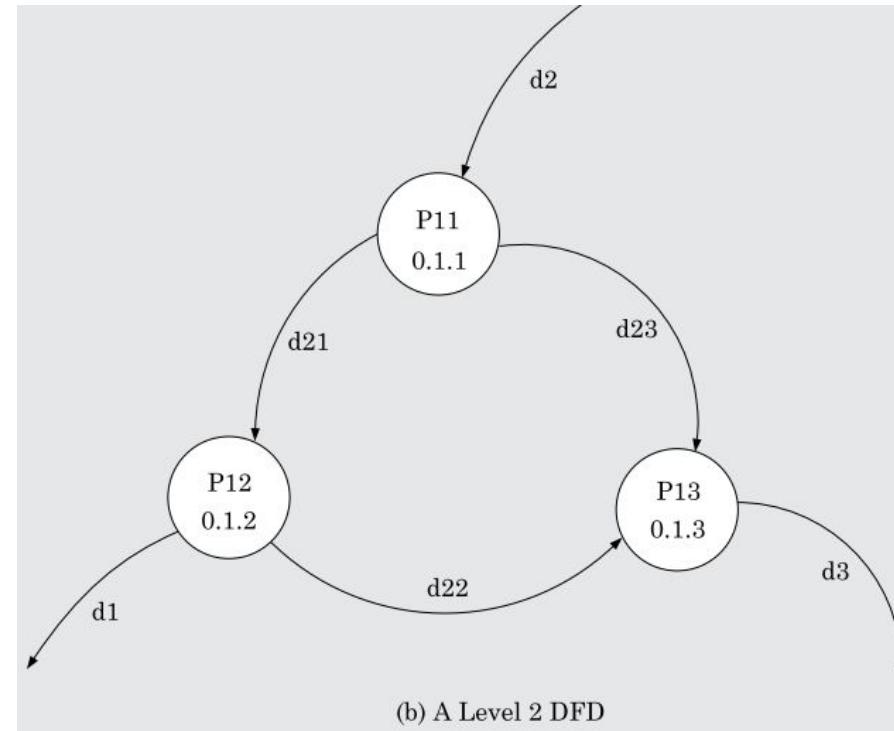
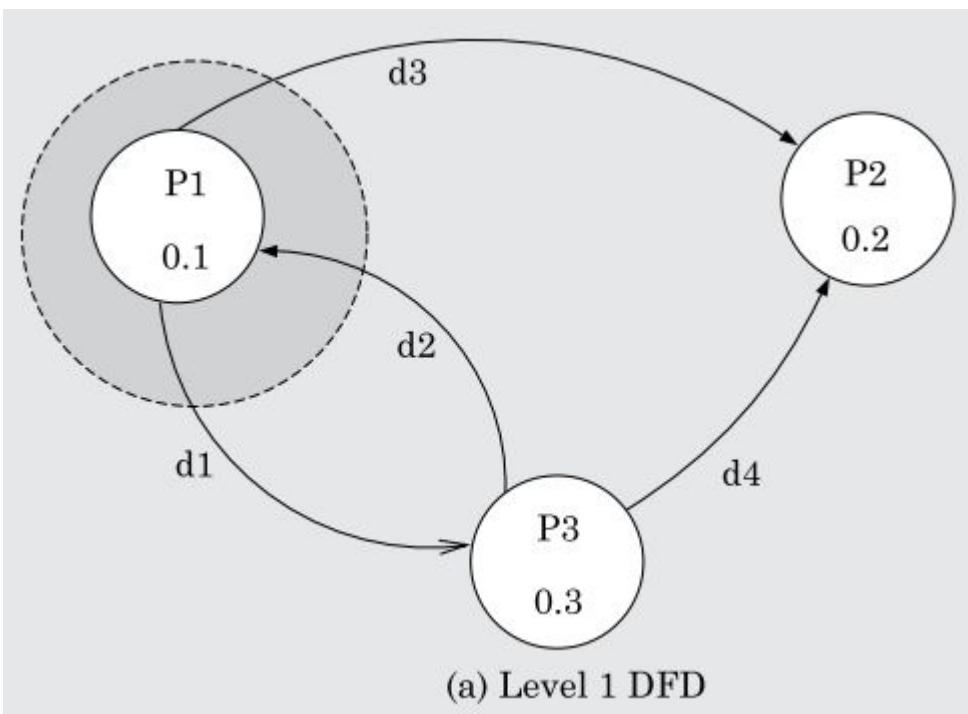
Balancing DFDs

The DFD model of a system usually consists of many DFDs that are organized in a hierarchy. In this context, a DFD is required to be balanced with respect to the corresponding bubble of the parent DFD.

Example:

We illustrate the concept of balancing a DFD in figure next. In the level 1 DFD, data items d1 and d3 flow out of the bubble 0.1 and the data item d2 flows into the bubble 0.1 (shown by the dotted circle). In the next level, bubble 0.1 is decomposed into three bubbles (0.1.1, 0.1.2, 0.1.3).

The decomposition is balanced, as d1 and d3 flow out of the level 2 diagram and d2 flows in.



DFD: RMS Calculating Software

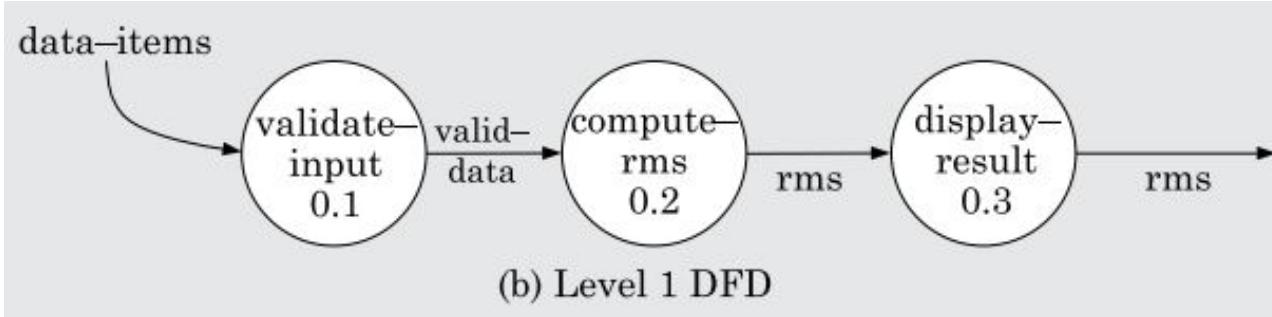
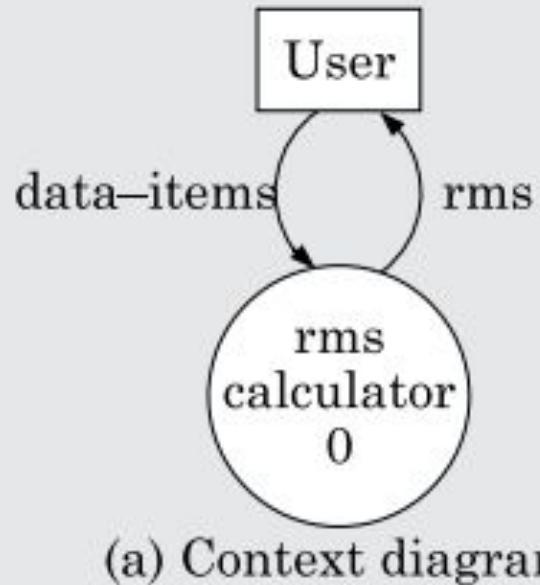
A software system called RMS calculating software would read three integral numbers from the user in the range of -1000 and $+1000$ and would determine the *root mean square (RMS)* of the three input numbers and display it.

In this example, the context diagram is simple to draw. The system accepts three integers from the user and returns the result to him. This has been shown in Figure-(a). To draw the level 1 DFD, from a cursory analysis of the problem description, we can see that there are four basic functions that the system needs to perform—

- Accept the input numbers from the user,
- Validate the numbers,
- Calculate the root mean square of the input numbers
- Display the result.

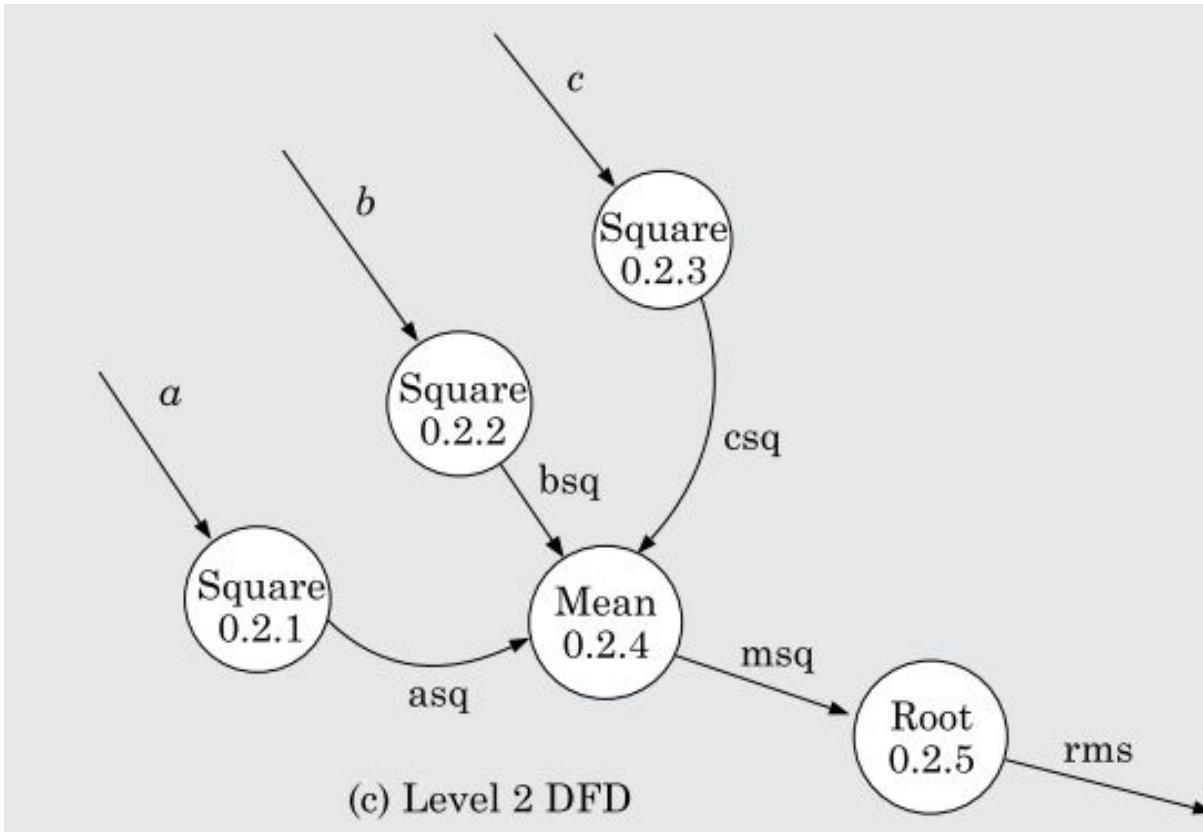
After representing these four functions in Figure-(b), we observe that the calculation of root mean square essentially consists of the functions—calculate the squares of the input numbers, calculate the mean, and finally calculate the root. This decomposition is shown in the level 2 DFD in Figure-(c).

DFD: RMS Calculating Software



Data dictionary for the DFD model

```
data-items: {integer}3
rms: float
valid-data:data-items
a: integer
b: integer
c: integer
asq: integer
bsq: integer
csq: integer
msq: integer
```



Structure Chart

Structure Chart

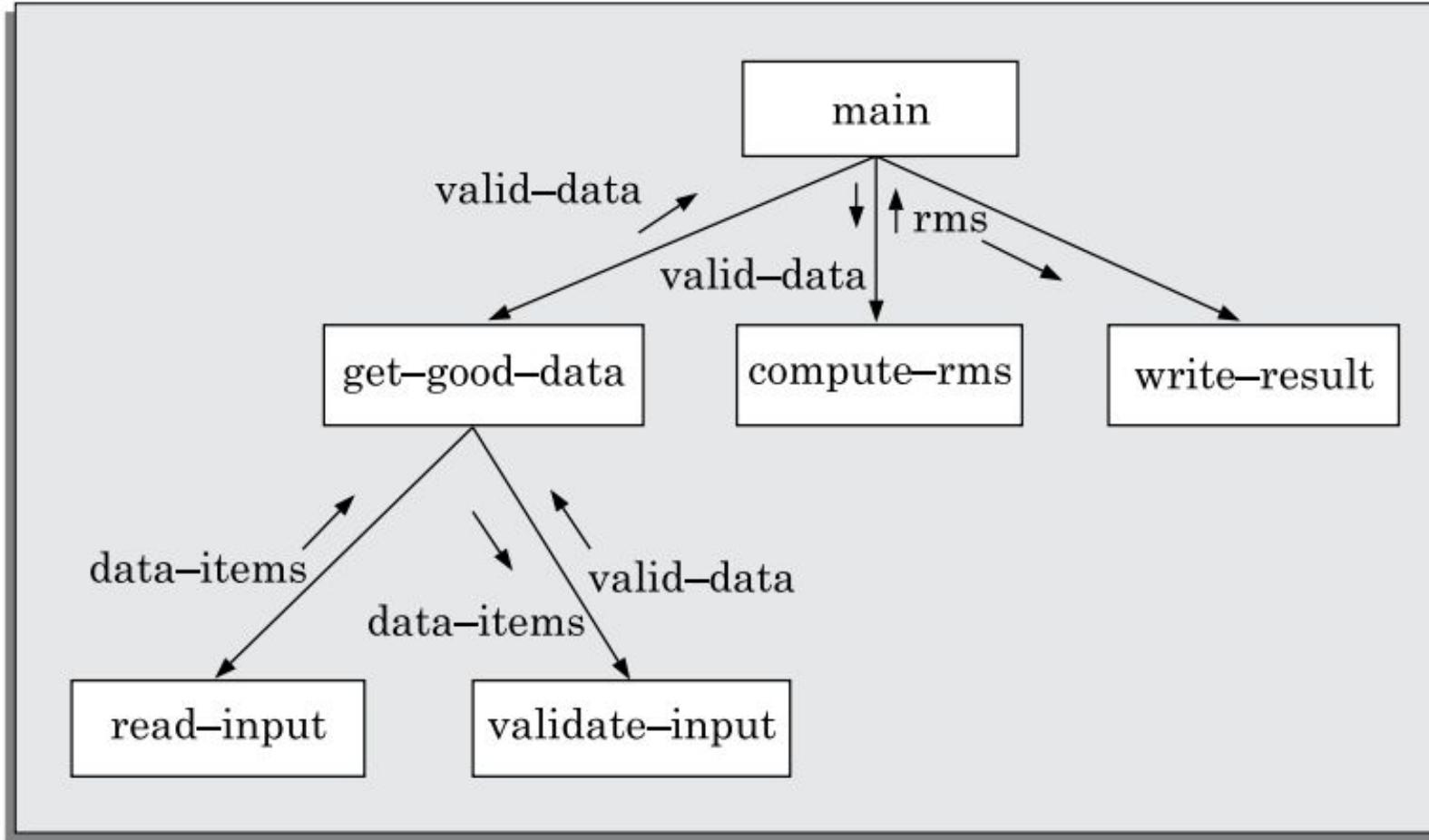
- A structure chart represents the software architecture. The various modules making up the system, the module dependency (i.e., which module calls which other modules), and the parameters that are passed among the different modules.
- The structure chart representation can be easily implemented using some programming language.
- Since the main focus in a structure chart representation is on module structure of a software and the interaction among the different modules, the procedural aspects (e.g., how a particular functionality is achieved) are not represented.

Structure Chart

The basic building blocks using which structure charts are designed are as following:

- **Rectangular boxes:** A rectangular box represents a **module**. Usually, every rectangular box is annotated with the name of the module it represents.
- **Module invocation arrows:** An arrow connecting two modules implies that during program execution **control is passed** from one module to the other in the direction of the connecting arrow. However, just by looking at the structure chart, we cannot say whether a module calls another module just **once or many times**. Also, just by looking at the structure chart, we cannot tell **the order** in which the different modules are invoked.
- **Data flow arrows:** These are **small arrows** appearing **alongside the module invocation arrows**. The data flow arrows are annotated with the corresponding data name. Data flow arrows represent the fact that the named data passes from one module to the other in the direction of the arrow.
- **Library modules:** A library module is usually represented by a **rectangle with double edges**. Libraries comprise the frequently called **modules**. Usually, when a module is **invoked by many other modules**, it is made into a library module.
- **Selection:** The **diamond symbol** represents the fact that one module of several modules connected with the diamond symbol is invoked **depending on the outcome of the condition** attached with the diamond symbol.
- **Repetition:** A **loop around the control flow arrows** denotes that the respective modules are invoked repeatedly.

Structure Chart

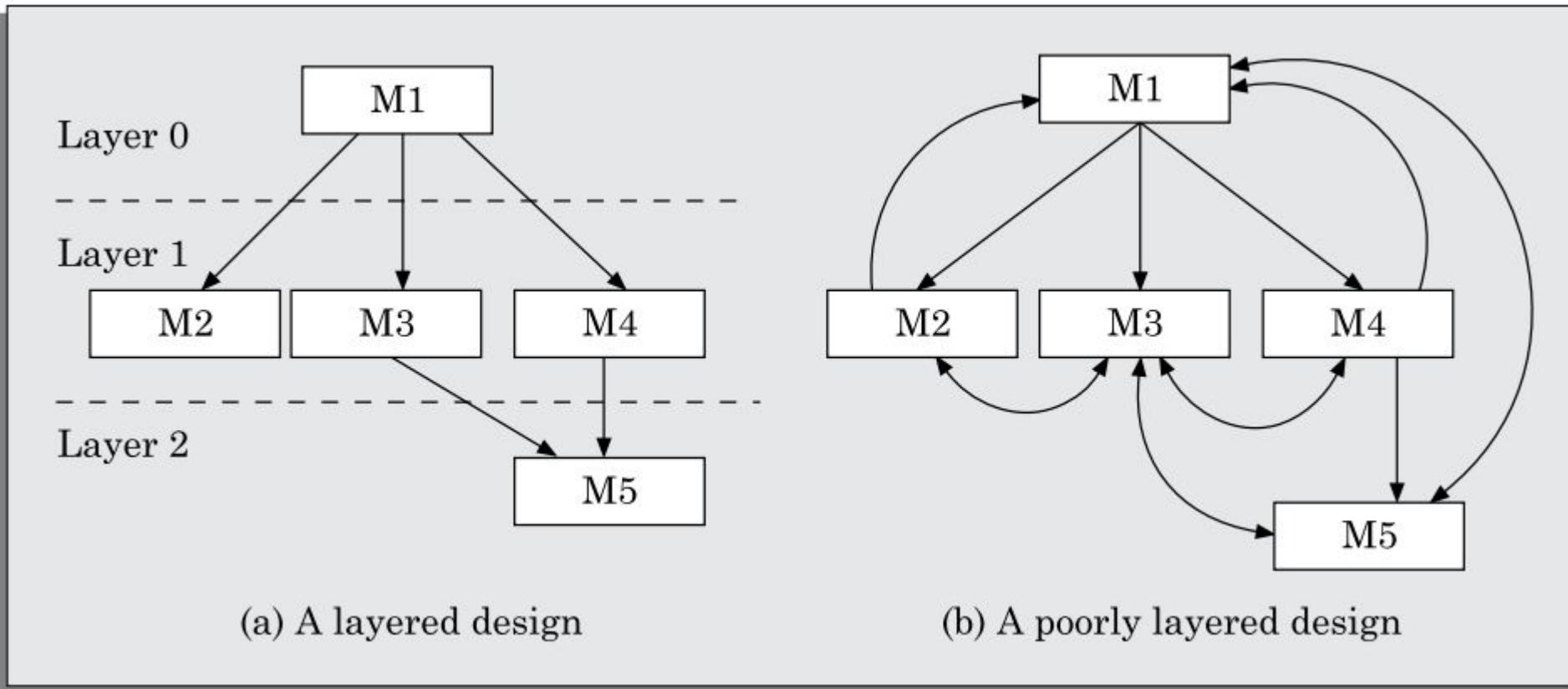


A simple structure chart for the problem of RMS computation.

Structure Chart

- In any structure chart, there should be one and **only one module at the top**, called the **root**.
- There should be **at most one control relationship** between any two modules in the structure chart. **This means that if module A invokes module B, module B cannot invoke module A.** The main reason behind this restriction is that we can consider the different modules of a structure chart to be arranged in layers or levels.
- The principle of abstraction does not allow lower-level modules to be aware of the existence of the high-level modules.
- However, it is possible for two higher-level modules to invoke the same lower-level **module**.

Structure Chart



Examples of properly and poorly layered designs.

Structure Chart

Flowchart VS. Structure Chart

Flow chart is a convenient technique to represent the flow of control in a program. A structure chart differs from a flow chart in three principal ways:

- It is usually difficult to identify the different modules of a program from its flow chart representation.
- Data interchange among different modules is not represented in a flow chart.
- Sequential ordering of tasks that is inherent to a flow chart is suppressed in a structure chart.

Transformation of a DFD Model into Structure Chart

Systematic techniques are available to transform the DFD representation of a problem into a module structure represented by a structure chart. There are two strategies to guide transformation of a DFD into a structure chart:

- Transform analysis
- Transaction analysis

At each level of transformation, it is important to first determine whether the transform or the transaction analysis is applicable to a particular DFD.

Whether to apply Transform or Transaction Processing

- First, examine the data input to the diagram.
- If all the data flow into the diagram are processed in similar ways (i.e., if all the input data flow arrows are incident on the same bubble in the DFD) then transform analysis is applicable.
- Otherwise, transaction analysis is applicable. Normally, transform analysis is applicable only to very simple processing.
- Transform analysis is normally applicable at the lower levels of a DFD model.
- Each different way in which data is processed corresponds to a separate transaction. Each transaction corresponds to a functionality that lets a user perform a meaningful piece of work using the software.

Transform analysis

Transform analysis identifies the primary functional components (modules) and the input and output data for these components. The first step in transform analysis is to divide the DFD into three types of parts:

- Input.
 - Processing.
 - Output.
- The input portion in the DFD includes processes that transform input data from physical (e.g., character from terminal) to logical form (e.g., internal tables, lists, etc.). Each input portion is called an *afferent branch*.
 - The output portion of a DFD transforms output data from logical form to physical form. Each output portion is called an *efferent branch*.
 - The remaining portion of a DFD is called *central transform*.

OBJECT MODELLING USING UML

Model

A model captures aspects important for some application while omitting (or abstracting) the rest. A model in the context of software development can be graphical, textual, mathematical, or program code-based. Models are very useful in documenting the design and analysis results. Models also facilitate the analysis and design procedures themselves. Graphical models are very popular because they are easy to understand and construct. UML is primarily a graphical modeling tool. However, it often requires text explanations to accompany the graphical models.

Need for a model

An important reason behind constructing a model is that it helps manage complexity. Once models of a system have been constructed, these can be used for a variety of purposes during software development, including the following:

- Analysis
- Specification
- Code generation
- Design
- Visualize and understand the problem and the working of a system
- Testing, etc.

In all these applications, the UML models can not only be used to document the results but also to arrive at the results themselves. Since a model can be used for a variety of purposes, it is reasonable to expect that the model would vary depending on the purpose for which it is being constructed.

Unified Modeling Language (UML)

UML, as the name implies, is a modeling language. It may be used to visualize, specify, construct, and document the artifacts of a software system. It provides a set of notations (e.g. rectangles, lines, ellipses, etc.) to create a visual model of the system. Like any other language, UML has its own syntax (symbols and sentence formation rules) and semantics (meanings of symbols and sentences). Also, we should clearly understand that UML is not a system design or development methodology, but can be used to document object-oriented and analysis results obtained using some methodology.

UML Diagrams

UML can be used to construct nine different types of diagrams to capture five different views of a system. Just as a building can be modeled from several views (or perspectives) such as ventilation perspective, electrical perspective, lighting perspective, heating perspective, etc.; the different UML diagrams provide different perspectives of the software system to be developed and facilitate a comprehensive understanding of the system. Such models can be refined to get the actual implementation of the system. The UML diagrams can capture the following five views of a system:

- User's view
- Structural view
- Behavioral view
- Implementation view
- Environmental view

UML Diagrams

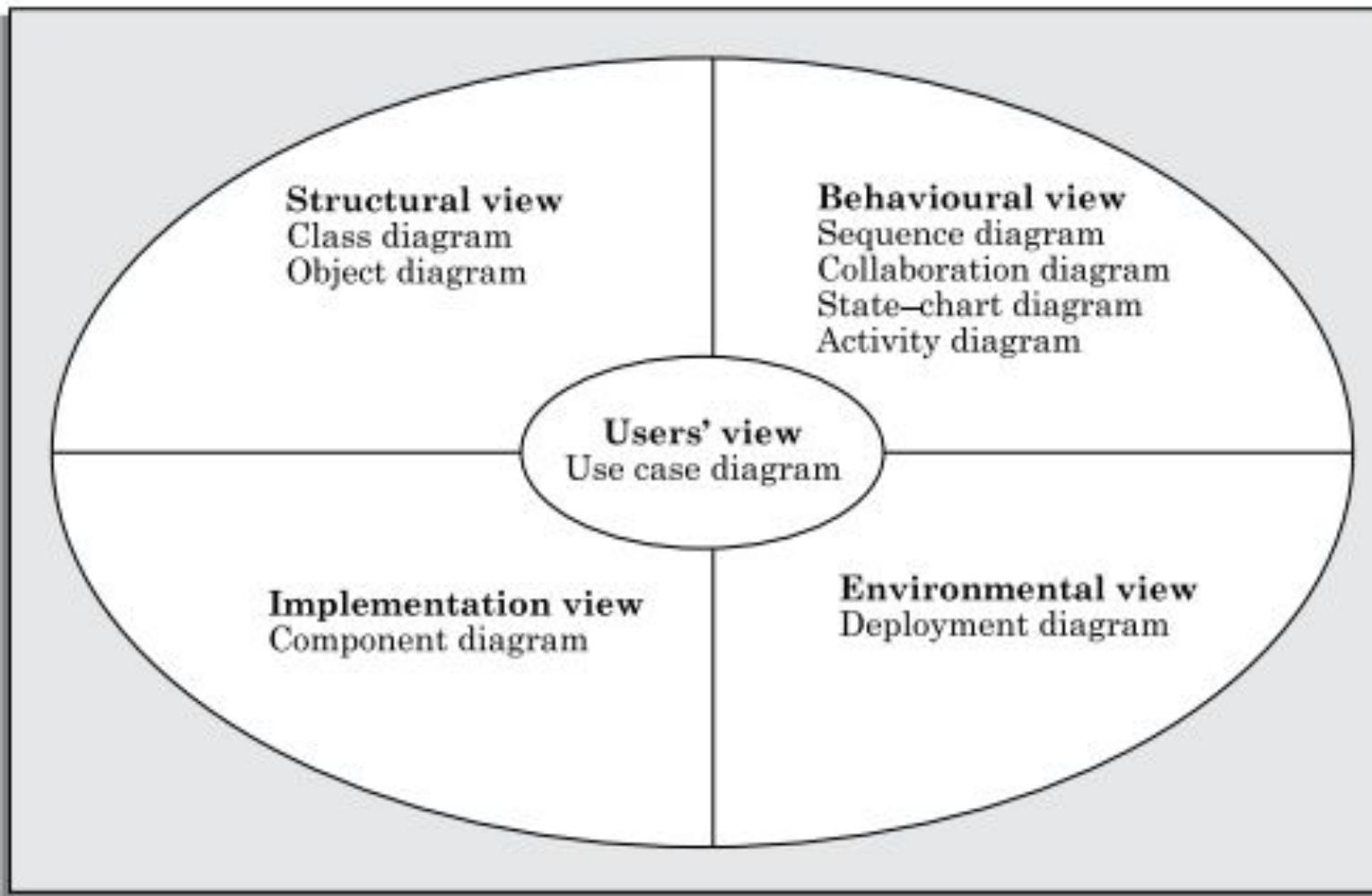


FIGURE: Different types of diagrams and views supported in UML.

UML Diagrams

User's view: This view defines the functionalities (facilities) made available by the system to its users. The users' view captures the external users' view of the system in terms of the functionalities offered by the system. The users' view is a black-box view of the system where the internal structure, the dynamic behavior of different system components, the implementation etc. are not visible. The users' view is very different from all other views in the sense that it is a functional model compared to the object model of all other views. The users' view can be considered as the central view and all other views are expected to conform to this view.

Structural view: The structural view defines the kinds of objects (classes) important to the understanding of the working of a system and to its implementation. It also captures the relationships among the classes (objects). The structural model is also called the static model, since the structure of a system does not change with time.

UML Diagrams

Behavioral view: The behavioral view captures how objects interact with each other to realize the system behavior. The system behavior captures the time-dependent (dynamic) behavior of the system.

Implementation view: This view captures the important components of the system and their dependencies.

Environmental view: This view models how the different components are implemented on different pieces of hardware.

Use Case Model

The use case model for any system consists of a set of “use cases”. Intuitively, use cases represent the different ways in which a system can be used by the users. A simple way to find all the use cases of a system is to ask the question: “What the users can do using the system?”

Example: For the Library Information System (LIS), the use cases could be:

- issue-book
- query-book
- return-book
- create-member
- add-book, etc

Use Case Model

- Use cases correspond to the high-level functional requirements.
- The use cases partition the system behavior into transactions, such that each transaction performs some useful action from the user's point of view.
- To complete each transaction may involve either a single message or multiple message exchanges between the user and the system to complete.
- The purpose of a use case is to define a piece of coherent behavior without revealing the internal structure of the system.
- The use cases do not mention any specific algorithm to be used nor the internal data representation, internal structure of the software.
- A use case typically involves a sequence of interactions between the user and the system.

Use Case Model

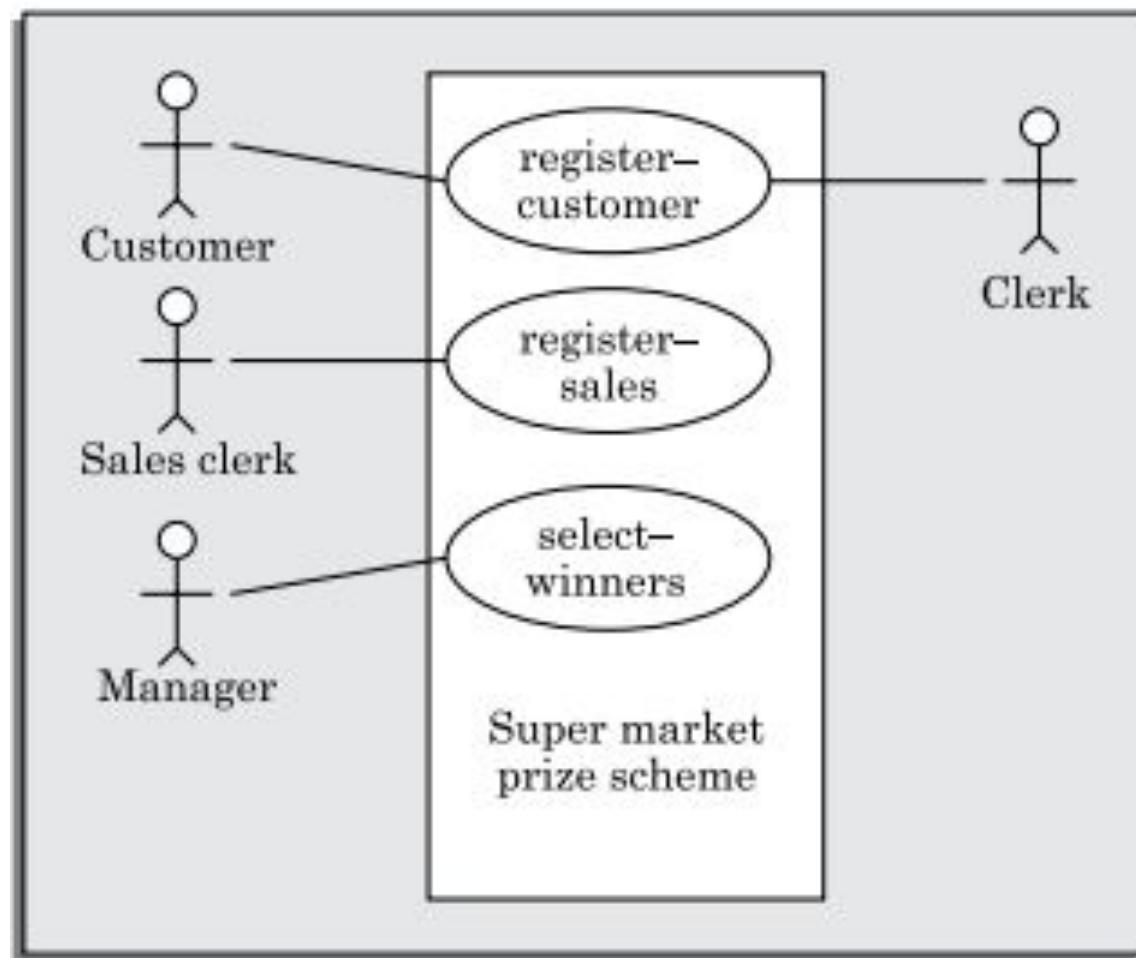


Figure: The use case diagram of the Super market prize scheme

OBJECT MODELLING USING UML

Model

A model captures aspects important for some application while omitting (or abstracting) the rest. A model in the context of software development can be graphical, textual, mathematical, or program code-based. Models are very useful in documenting the design and analysis results. Models also facilitate the analysis and design procedures themselves. Graphical models are very popular because they are easy to understand and construct. UML is primarily a graphical modeling tool. However, it often requires text explanations to accompany the graphical models.

Need for a model

An important reason behind constructing a model is that it helps manage complexity. Once models of a system have been constructed, these can be used for a variety of purposes during software development, including the following:

- Analysis
- Specification
- Code generation
- Design
- Visualize and understand the problem and the working of a system
- Testing, etc.

In all these applications, the UML models can not only be used to document the results but also to arrive at the results themselves. Since a model can be used for a variety of purposes, it is reasonable to expect that the model would vary depending on the purpose for which it is being constructed.

Unified Modeling Language (UML)

UML, as the name implies, is a modeling language. It may be used to visualize, specify, construct, and document the artifacts of a software system. It provides a set of notations (e.g. rectangles, lines, ellipses, etc.) to create a visual model of the system. Like any other language, UML has its own syntax (symbols and sentence formation rules) and semantics (meanings of symbols and sentences). Also, we should clearly understand that UML is not a system design or development methodology, but can be used to document object-oriented and analysis results obtained using some methodology.

UML Diagrams

UML can be used to construct nine different types of diagrams to capture five different views of a system. Just as a building can be modeled from several views (or perspectives) such as ventilation perspective, electrical perspective, lighting perspective, heating perspective, etc.; the different UML diagrams provide different perspectives of the software system to be developed and facilitate a comprehensive understanding of the system. Such models can be refined to get the actual implementation of the system. The UML diagrams can capture the following five views of a system:

- User's view
- Structural view
- Behavioral view
- Implementation view
- Environmental view

UML Diagrams

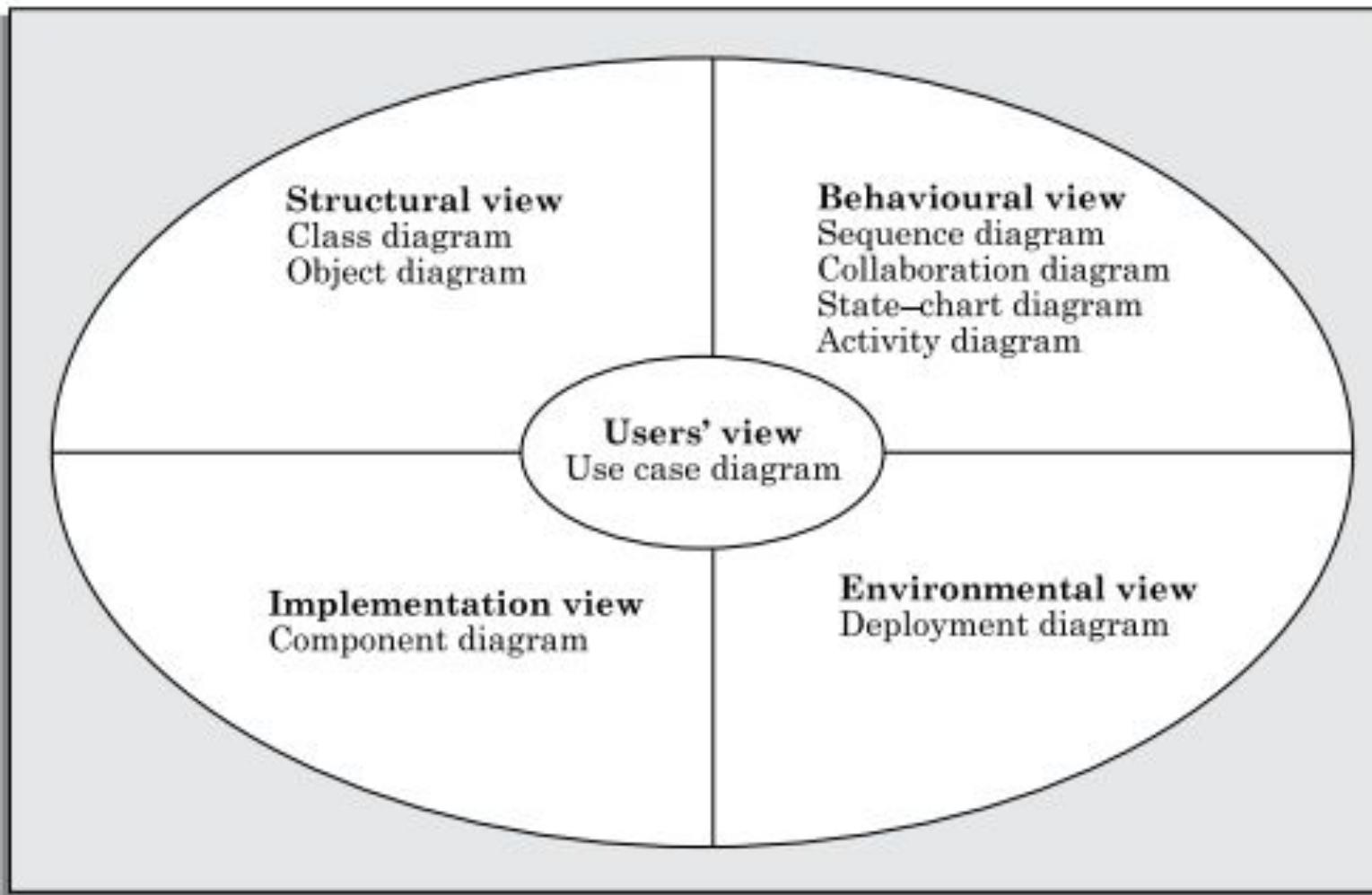


FIGURE: Different types of diagrams and views supported in UML.

UML Diagrams

User's view: This view defines the functionalities (facilities) made available by the system to its users. The users' view captures the external users' view of the system in terms of the functionalities offered by the system. The users' view is a black-box view of the system where the internal structure, the dynamic behavior of different system components, the implementation etc. are not visible. The users' view is very different from all other views in the sense that it is a functional model compared to the object model of all other views. The users' view can be considered as the central view and all other views are expected to conform to this view.

Structural view: The structural view defines the kinds of objects (classes) important to the understanding of the working of a system and to its implementation. It also captures the relationships among the classes (objects). The structural model is also called the static model, since the structure of a system does not change with time.

UML Diagrams

Behavioral view: The behavioral view captures how objects interact with each other to realize the system behavior. The system behavior captures the time-dependent (dynamic) behavior of the system.

Implementation view: This view captures the important components of the system and their dependencies.

Environmental view: This view models how the different components are implemented on different pieces of hardware.

Use Case Model

The use case model for any system consists of a set of “use cases”. Intuitively, use cases represent the different ways in which a system can be used by the users. A simple way to find all the use cases of a system is to ask the question: “What the users can do using the system?”

Example: For the Library Information System (LIS), the use cases could be:

- issue-book
- query-book
- return-book
- create-member
- add-book, etc

Use Case Model

- Use cases correspond to the high-level functional requirements.
- The use cases partition the system behavior into transactions, such that each transaction performs some useful action from the user's point of view.
- To complete each transaction may involve either a single message or multiple message exchanges between the user and the system to complete.
- The purpose of a use case is to define a piece of coherent behavior without revealing the internal structure of the system.
- The use cases do not mention any specific algorithm to be used nor the internal data representation, internal structure of the software.
- A use case typically involves a sequence of interactions between the user and the system.

Use Case Model

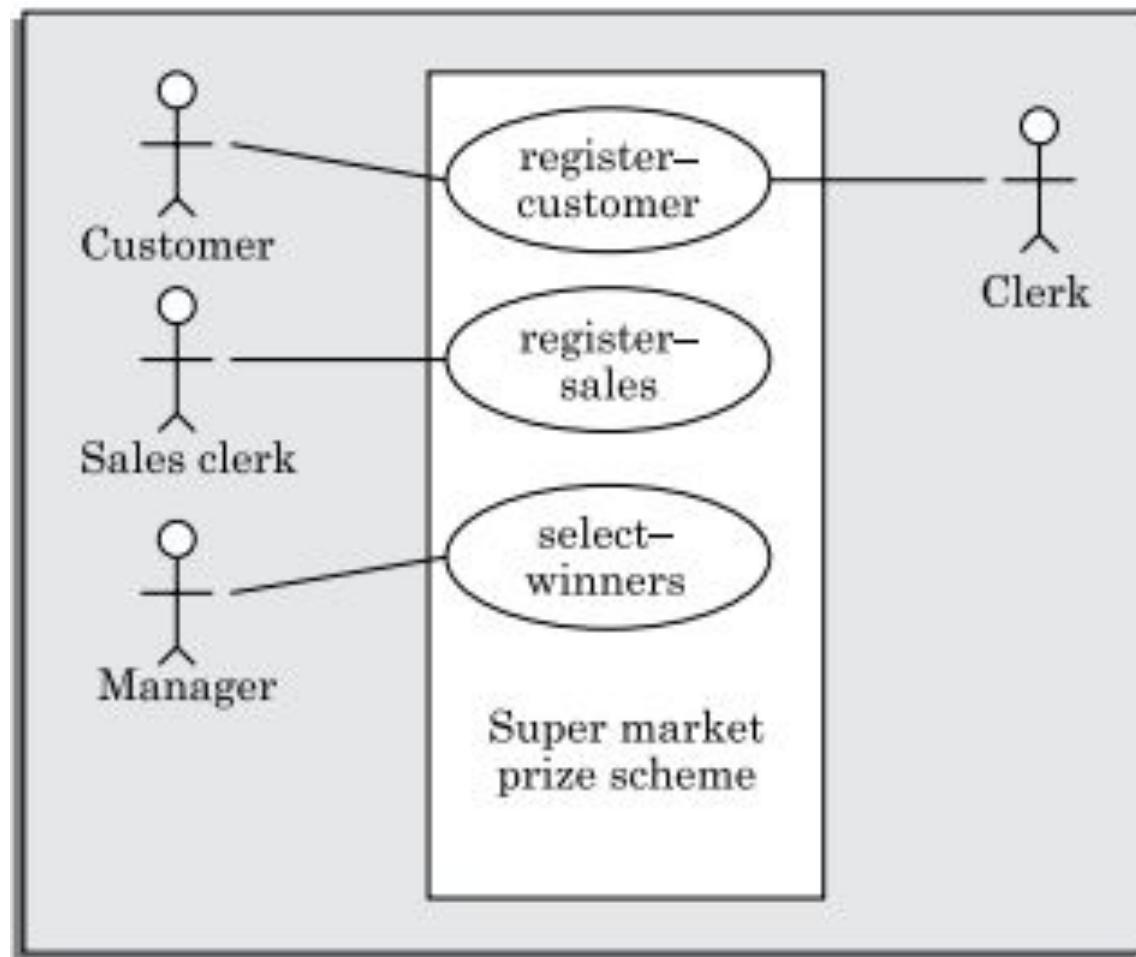


Figure: The use case diagram of the Super market prize scheme

Use Case Model

The Main Line Sequence:

- The mainline sequence represents the normal interaction between a user and the system. The mainline sequence is the most occurring sequence of interaction.
- For example, the mainline sequence of the withdraw cash use case supported by a bank ATM drawn, complete the transaction, and get the amount.
- Several variations to the main line sequence may also exist. Typically, a variation from the mainline sequence occurs when some specific conditions hold. For the bank ATM example, variations or alternate scenarios may occur, if the password is invalid or the amount to be withdrawn exceeds the amount balance. The variations are also called alternative paths. A use case can be viewed as a set of related scenarios tied together by a common goal. The mainline sequence and each of the variations are called scenarios or instances of the use case. Each scenario is a single path of user events and system activity through the use case.

Use Case Model

Representation of Use Cases:

- Use cases can be represented 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 (such as Library Information System) appears inside the rectangle.

Use Case Model

Representation of Use Cases:

- The different users of the system are represented by using the stick person icon.
- Each stick person icon is normally referred to as an actor.
- An actor is a role played by a user with respect to the system use. It is possible that the same user may play the role of multiple actors. Each actor can participate in one or more use cases.
- The line connecting the actor and the use case is called the communication relationship. It indicates that the actor makes use of the functionality provided by the use case.
- Both the human users and the external systems can be represented by stick person icons.
- When a stick person icon represents an external system, it is annotated by the stereotype <>.

Use Case Model

Text Description:

Each ellipse on the use case diagram should be accompanied by a text description to define the details of the interaction between the user and the computer and other aspects of the use case. It should include all the behavior associated with the use case in terms of the mainline sequence, different variations to the normal behavior, the system responses associated with the use case, the exceptional conditions that may occur in the behavior, etc.

The behavior description is often written in a conversational style describing the interactions between the actor and the system. The text description may be informal, but some structuring is recommended. The following are some of the information which may be included in a use case text description in addition to the mainline sequence, and the alternative scenarios.

Use Case Model

Text Description:

The following are some of the information which may be included in a use case text description in addition to the mainline sequence, and the alternative scenarios.

Contact persons: This section lists the personnel of the client organization with whom the use case was discussed, date and time of the meeting, etc.

Actors: In addition to identifying the actors, some information about actors using this use case which may help the implementation of the use case may be recorded.

Pre-condition: The preconditions would describe the state of the system before the use case execution starts.

Post-condition: This captures the state of the system after the use case has successfully completed.

Use Case Model

Text Description:

Non-functional requirements: This could contain the important constraints for the design and implementation, such as platform and environment conditions, qualitative statements, response time requirements, etc.

Exceptions, error situations: This contains only the domain-related errors such as lack of user's access rights, invalid entry in the input fields, etc. Obviously, errors that are not domain related, such as software errors, need not be discussed here.

Sample dialogs: These serve as examples illustrating the use case.

Specific user interface requirements: These contain specific requirements for the user interface of the use case. For example, it may contain forms to be used, screen shots, interaction style, etc.

Document references: This part contains references to specific domain-related documents which may be useful to understand the system operation

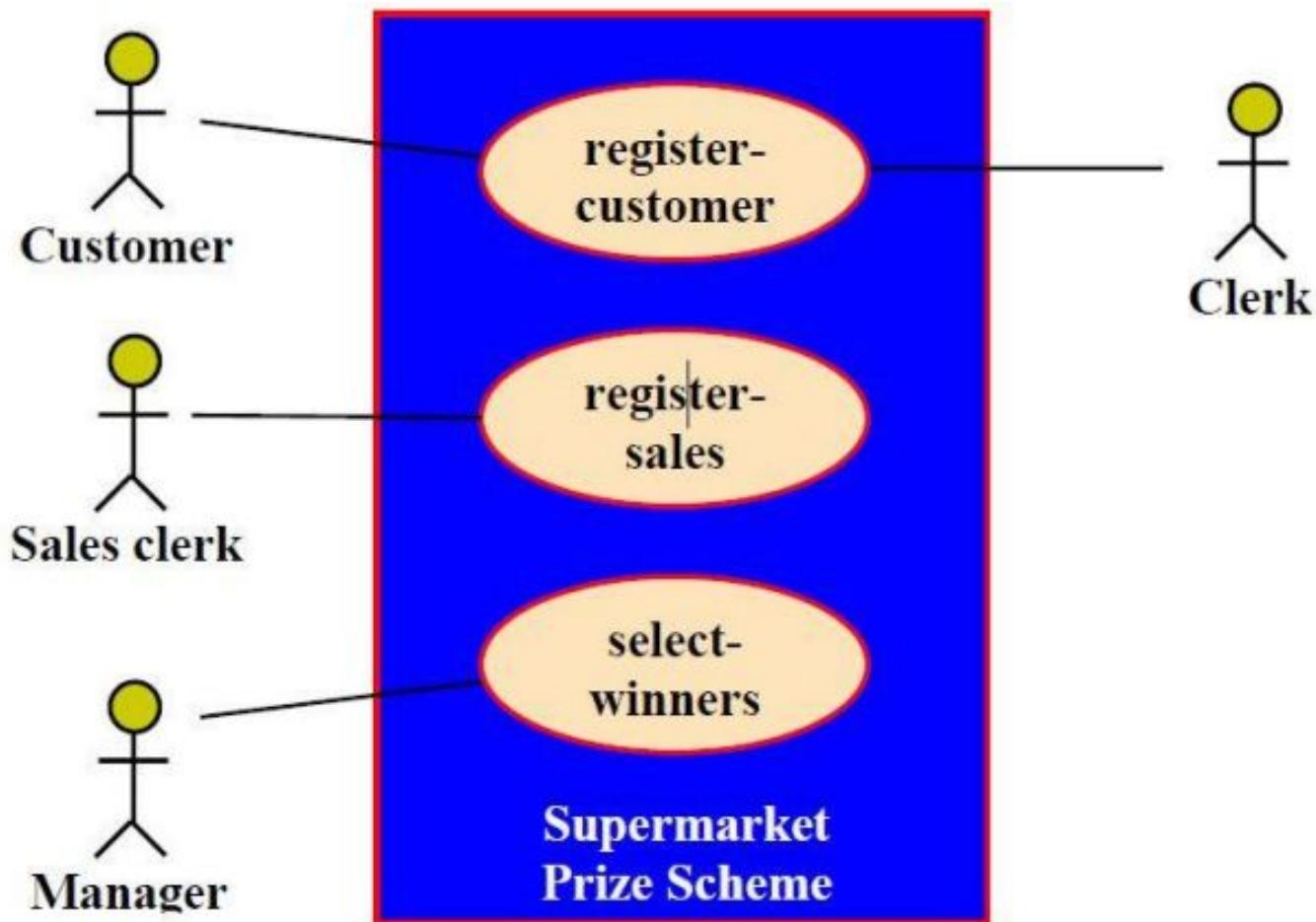
Use Case Model

Example: A supermarket needs to develop the following software to encourage regular customers. For this, the customer needs to supply 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 checkout 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 Tk.100,000. The entries against the CN are reset on the day of every year after the prize winners' lists are generated.

Use Case Model

Example:

The use case model for the Supermarket Prize Scheme is shown in the figure. We can identify three use cases: “register-customer”, “register-sales”, and “select-winners”.



Use Case Model

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 been successfully registered.

Scenario 2: at step 4 of mainline sequence

1. System: displays the message that the customer has already registered.

Scenario 2: at step 4 of mainline sequence

1. System: displays the message that some input information has not been entered. The system displays a prompt to enter the missing value.

The description for other use cases is written in a similar fashion.

OBJECT MODELLING USING UML

Model

A model captures aspects important for some application while omitting (or abstracting) the rest. A model in the context of software development can be graphical, textual, mathematical, or program code-based. Models are very useful in documenting the design and analysis results. Models also facilitate the analysis and design procedures themselves. Graphical models are very popular because they are easy to understand and construct. UML is primarily a graphical modeling tool. However, it often requires text explanations to accompany the graphical models.

Need for a model

An important reason behind constructing a model is that it helps manage complexity. Once models of a system have been constructed, these can be used for a variety of purposes during software development, including the following:

- Analysis
- Specification
- Code generation
- Design
- Visualize and understand the problem and the working of a system
- Testing, etc.

In all these applications, the UML models can not only be used to document the results but also to arrive at the results themselves. Since a model can be used for a variety of purposes, it is reasonable to expect that the model would vary depending on the purpose for which it is being constructed.

Unified Modeling Language (UML)

UML, as the name implies, is a modeling language. It may be used to visualize, specify, construct, and document the artifacts of a software system. It provides a set of notations (e.g. rectangles, lines, ellipses, etc.) to create a visual model of the system. Like any other language, UML has its own syntax (symbols and sentence formation rules) and semantics (meanings of symbols and sentences). Also, we should clearly understand that UML is not a system design or development methodology, but can be used to document object-oriented and analysis results obtained using some methodology.

UML Diagrams

UML can be used to construct nine different types of diagrams to capture five different views of a system. Just as a building can be modeled from several views (or perspectives) such as ventilation perspective, electrical perspective, lighting perspective, heating perspective, etc.; the different UML diagrams provide different perspectives of the software system to be developed and facilitate a comprehensive understanding of the system. Such models can be refined to get the actual implementation of the system. The UML diagrams can capture the following five views of a system:

- User's view
- Structural view
- Behavioral view
- Implementation view
- Environmental view

UML Diagrams

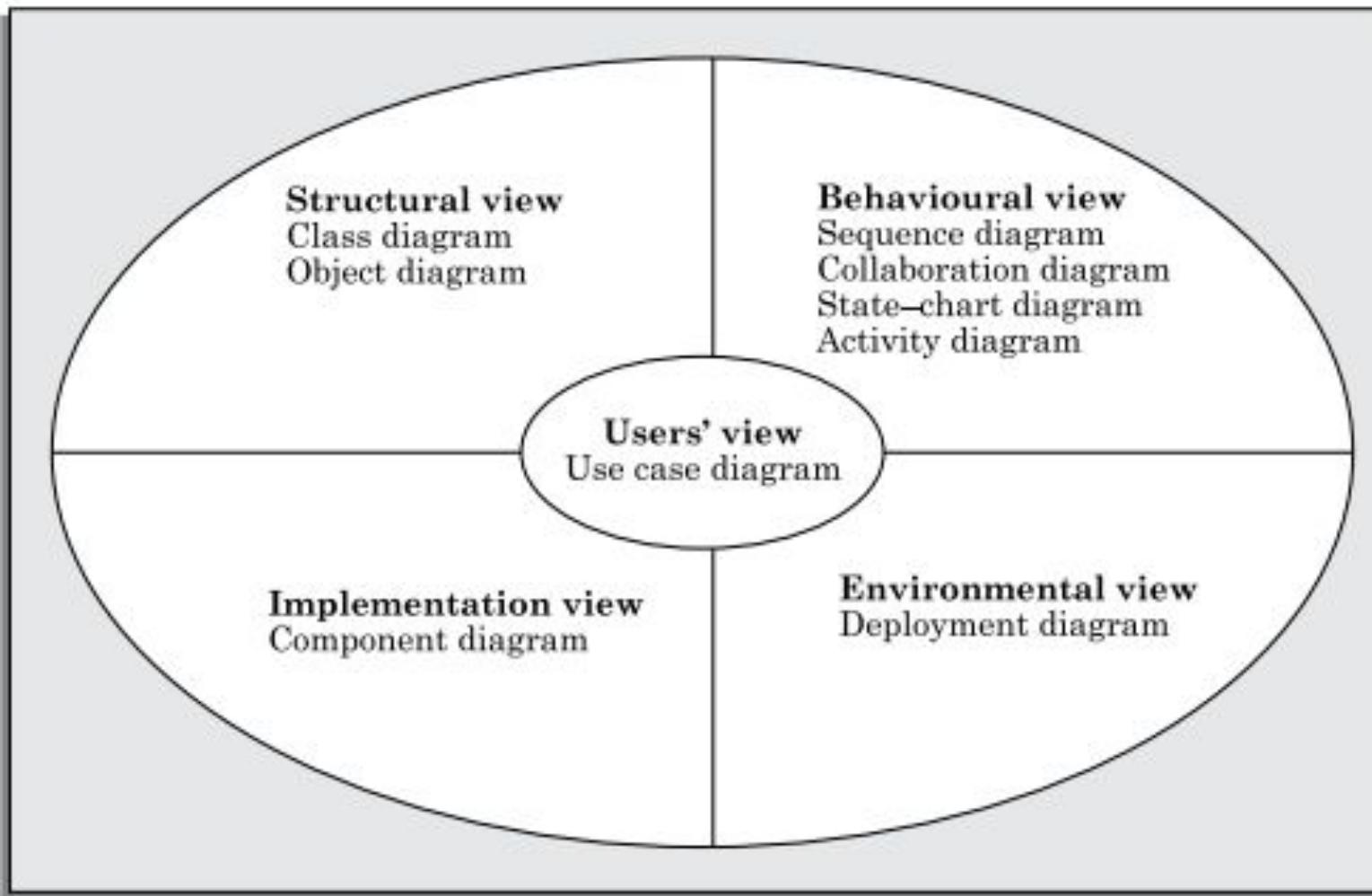


FIGURE: Different types of diagrams and views supported in UML.

UML Diagrams

User's view: This view defines the functionalities (facilities) made available by the system to its users. The users' view captures the external users' view of the system in terms of the functionalities offered by the system. The users' view is a black-box view of the system where the internal structure, the dynamic behavior of different system components, the implementation etc. are not visible. The users' view is very different from all other views in the sense that it is a functional model compared to the object model of all other views. The users' view can be considered as the central view and all other views are expected to conform to this view.

Structural view: The structural view defines the kinds of objects (classes) important to the understanding of the working of a system and to its implementation. It also captures the relationships among the classes (objects). The structural model is also called the static model, since the structure of a system does not change with time.

UML Diagrams

Behavioral view: The behavioral view captures how objects interact with each other to realize the system behavior. The system behavior captures the time-dependent (dynamic) behavior of the system.

Implementation view: This view captures the important components of the system and their dependencies.

Environmental view: This view models how the different components are implemented on different pieces of hardware.

Use Case Model

The use case model for any system consists of a set of “use cases”. Intuitively, use cases represent the different ways in which a system can be used by the users. A simple way to find all the use cases of a system is to ask the question: “What the users can do using the system?”

Example: For the Library Information System (LIS), the use cases could be:

- issue-book
- query-book
- return-book
- create-member
- add-book, etc

Use Case Model

- Use cases correspond to the high-level functional requirements.
- The use cases partition the system behavior into transactions, such that each transaction performs some useful action from the user's point of view.
- To complete each transaction may involve either a single message or multiple message exchanges between the user and the system to complete.
- The purpose of a use case is to define a piece of coherent behavior without revealing the internal structure of the system.
- The use cases do not mention any specific algorithm to be used nor the internal data representation, internal structure of the software.
- A use case typically involves a sequence of interactions between the user and the system.

Use Case Model

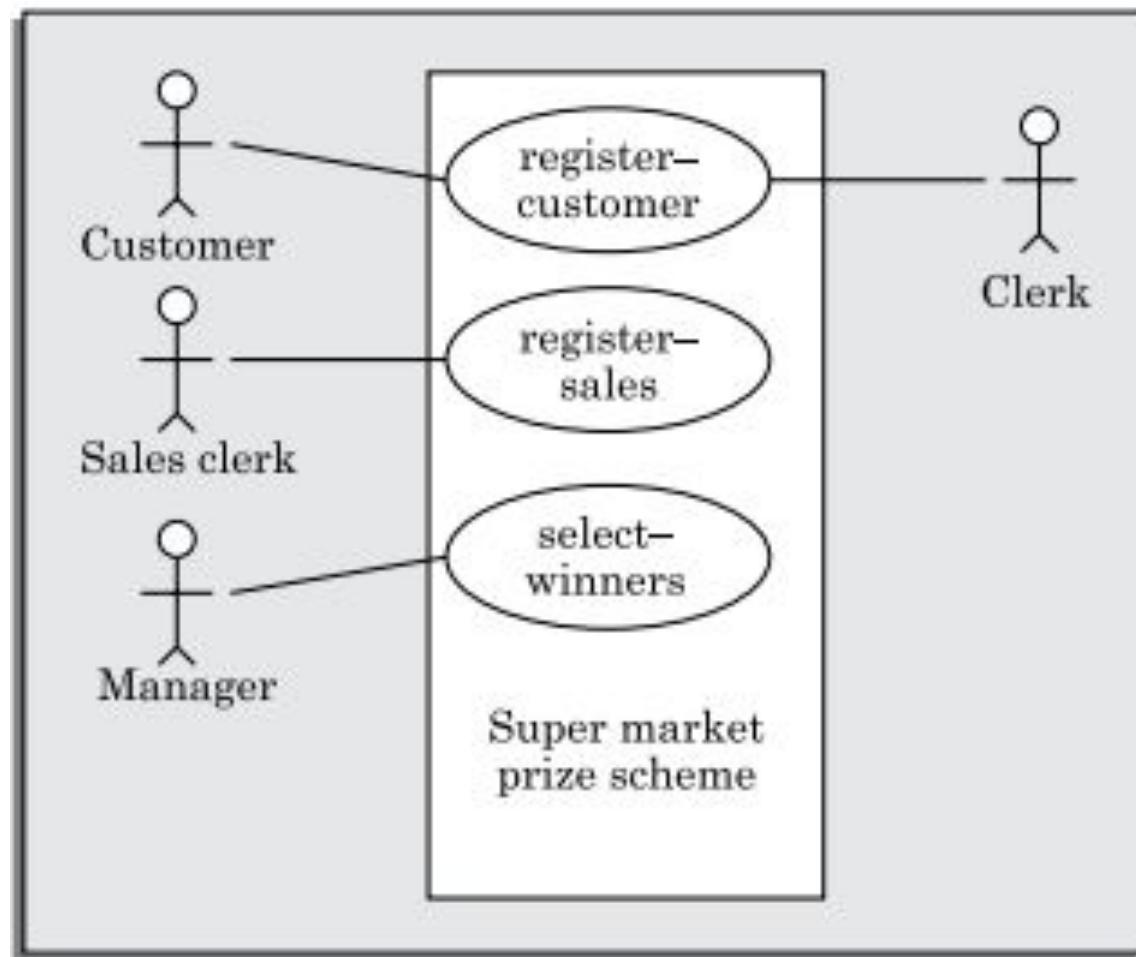


Figure: The use case diagram of the Super market prize scheme

Use Case Model

The Main Line Sequence:

- The mainline sequence represents the normal interaction between a user and the system. The mainline sequence is the most occurring sequence of interaction.
- For example, the mainline sequence of the withdraw cash use case supported by a bank ATM drawn, complete the transaction, and get the amount.
- Several variations to the main line sequence may also exist. Typically, a variation from the mainline sequence occurs when some specific conditions hold. For the bank ATM example, variations or alternate scenarios may occur, if the password is invalid or the amount to be withdrawn exceeds the amount balance. The variations are also called alternative paths. A use case can be viewed as a set of related scenarios tied together by a common goal. The mainline sequence and each of the variations are called scenarios or instances of the use case. Each scenario is a single path of user events and system activity through the use case.

Use Case Model

Representation of Use Cases:

- Use cases can be represented 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 (such as Library Information System) appears inside the rectangle.

Use Case Model

Representation of Use Cases:

- The different users of the system are represented by using the stick person icon.
- Each stick person icon is normally referred to as an **actor**.
- An actor is a role played by a user with respect to the system use. It is possible that the same user may play the role of multiple actors. Each actor can participate in one or more use cases.
- The line connecting the actor and the use case is called the communication relationship. It indicates that the actor makes use of the functionality provided by the use case.
- Both the human users and the external systems can be represented by stick person icons.
- When a stick person icon represents an external system, it is annotated by the stereotype <>.

Use Case Model

Text Description:

Each ellipse on the use case diagram should be accompanied by a text description to define the details of the interaction between the user and the computer and other aspects of the use case. It should include all the behavior associated with the use case in terms of the mainline sequence, different variations to the normal behavior, the system responses associated with the use case, the exceptional conditions that may occur in the behavior, etc.

The behavior description is often written in a **conversational** style describing the interactions between the actor and the system. The text description may be informal, but some structuring is recommended. The following are some of the information which may be included in a use case **text description** in addition to the mainline sequence, and the **alternative scenarios**.

Use Case Model

Text Description:

The following are some of the information which may be included in a use case text description in addition to the mainline sequence, and the alternative scenarios.

Contact persons: This section lists the personnel of the client organization with whom the use case was discussed, date and time of the meeting, etc.

Actors: In addition to identifying the actors, some information about actors using this use case which may help the implementation of the use case may be recorded.

Pre-condition: The preconditions would describe the state of the system before the use case execution starts.

Post-condition: This captures the state of the system after the use case has successfully completed.

Use Case Model

Text Description:

Non-functional requirements: This could contain the important constraints for the design and implementation, such as platform and environment conditions, qualitative statements, response time requirements, etc.

Exceptions, error situations: This contains only the domain-related errors such as lack of user's access rights, invalid entry in the input fields, etc. Obviously, errors that are not domain related, such as software errors, need not be discussed here.

Sample dialogs: These serve as examples illustrating the use case.

Specific user interface requirements: These contain specific requirements for the user interface of the use case. For example, it may contain forms to be used, screen shots, interaction style, etc.

Document references: This part contains references to specific domain-related documents which may be useful to understand the system operation

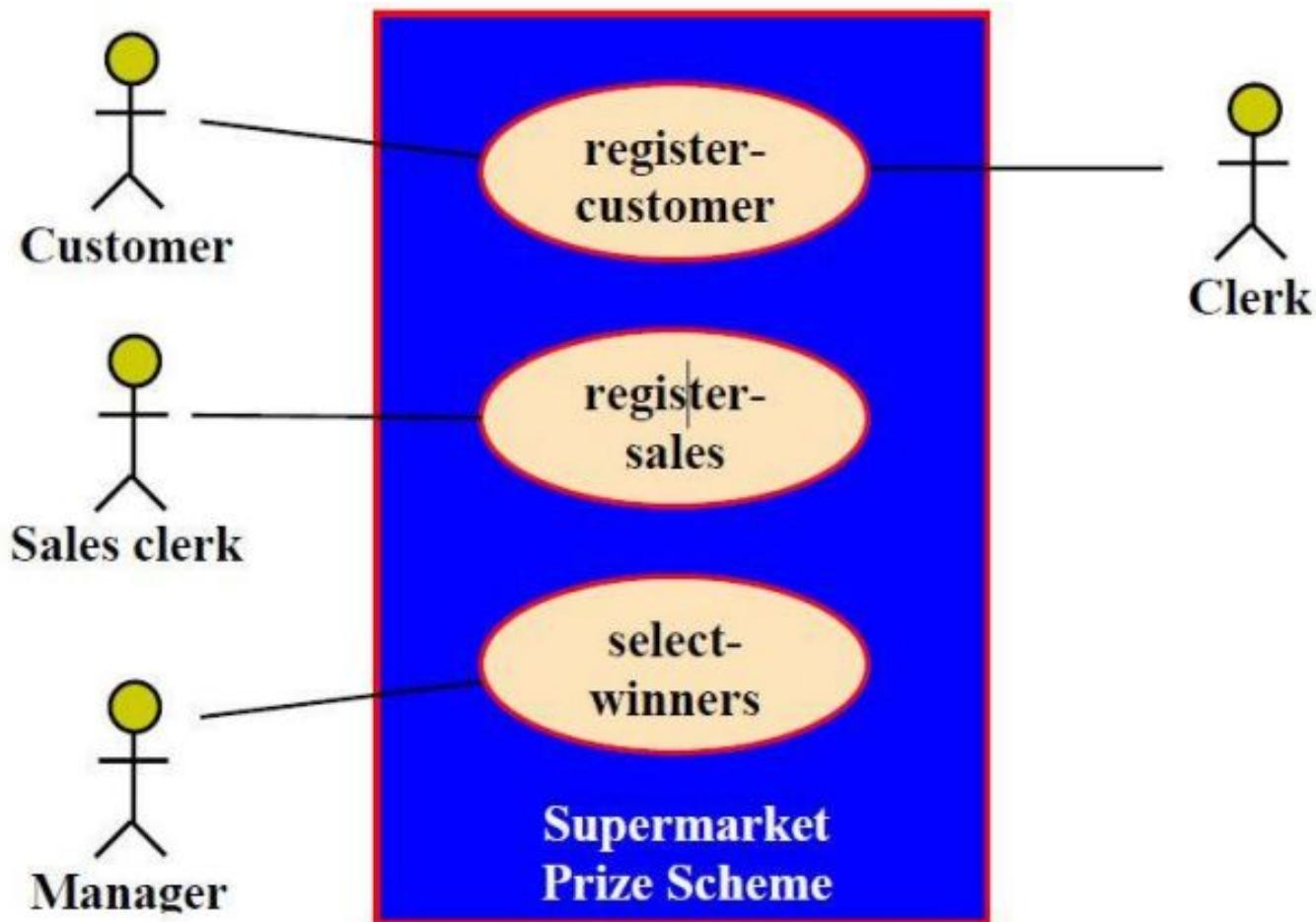
Use Case Model

Example: A supermarket needs to develop the following software to encourage regular customers. For this, the customer needs to supply 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 checkout 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 Tk.100,000. The entries against the CN are reset on the day of every year after the prize winners' lists are generated.

Use Case Model

Example:

The use case model for the Supermarket Prize Scheme is shown in the figure. We can identify three use cases: “register-customer”, “register-sales”, and “select-winners”.



Use Case Model

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 been successfully registered.

Scenario 2: at step 4 of mainline sequence

1. System: displays the message that the customer has already registered.

Scenario 2: at step 4 of mainline sequence

1. System: displays the message that some input information has not been entered. The system displays a prompt to enter the missing value.

The description for other use cases is written in a similar fashion.

How to Identify the Use Cases of a System?

Identification of the use cases involves brain storming and reviewing the SRS document. Typically, the **high-level requirements** specified in the SRS document correspond to the use cases. In the absence of a well-formulated SRS document, a popular method of identifying the use cases is actor-based. This involves first identifying the different types of actors and their usage of the system. Subsequently, for each actor the different functions that they might initiate or participate are identified.

Example: For a Library Automation System, the categories of users can be members, librarian, and the accountant. Each user typically focuses on a set of functionalities. For example, the member typically concerns himself with book issue, return, and renewal aspects. The librarian concerns himself with creation and deletion of the member and book records. The accountant concerns itself with the amount collected from membership fees and the expenses aspects.

Factoring of Use Cases

It is often desirable to factor use cases into component use cases.

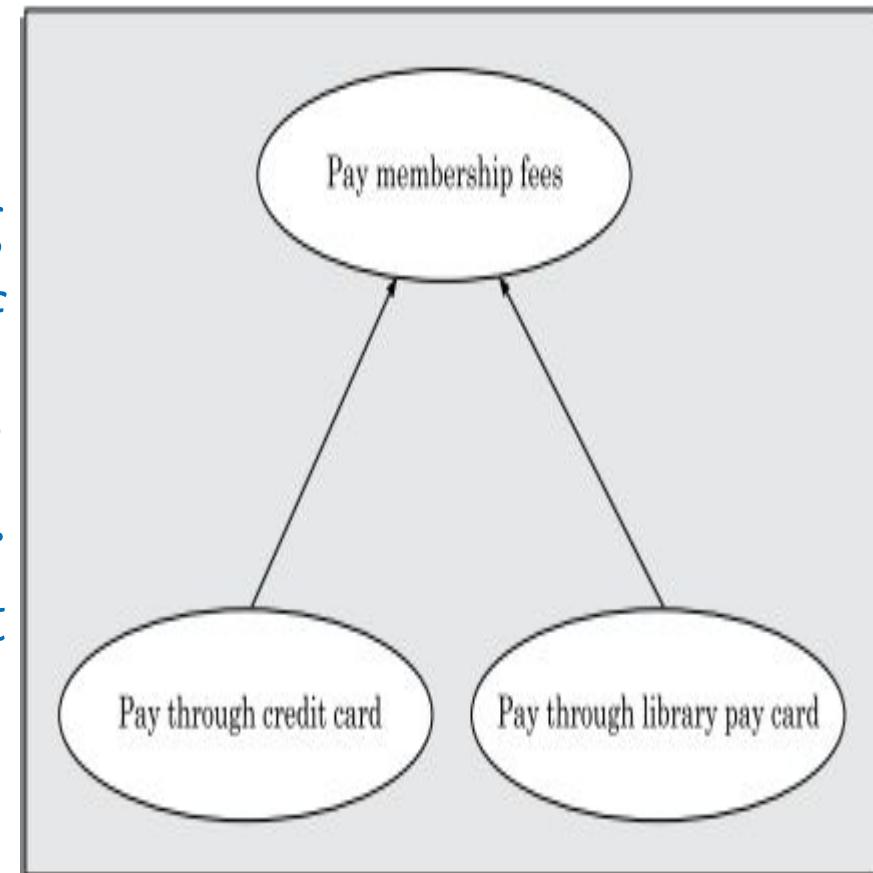
Actually, factoring of use cases are required under two situations-

Decompose complex use cases: Complex use cases need to be factored into simpler use cases. This would not only make the behavior associated with the use case much more comprehensible, but also make the corresponding interaction diagrams more tractable. Without decomposition, the interaction diagrams for complex use cases may become too large to be accommodated on a single sized (A4) paper.

Identify commonality: Secondly, use cases need to be factored whenever there is common behavior across different use cases. Factoring would make it possible to define such behavior only once and reuse it whenever required. It is desirable to factor out common usage such as error handling from a set of use cases. This makes analysis of the class design much simpler and elegant.

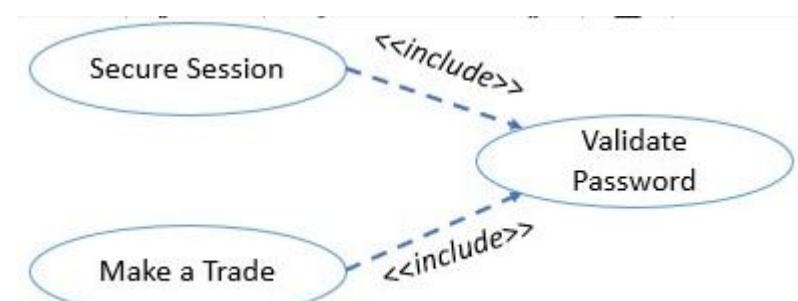
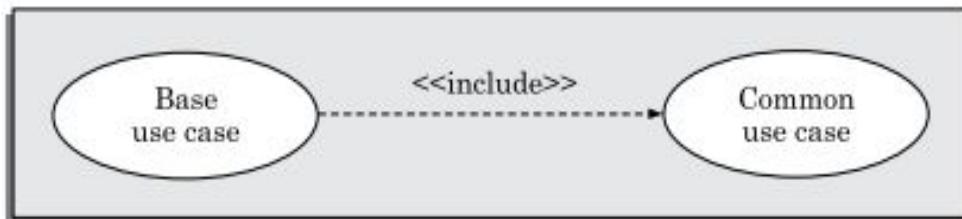
Factoring of Use Cases

Generalization: Use case generalization can be used when one use case is quite similar to another, but does something slightly different or something more. That is one use case is a special case of another use case. In this sense, generalization works the same way with use cases as it does with classes. The child use case inherits the behavior of the parent use case. The notation is the same too (See Figure). It is important to remember that the base and the derived use cases are separate use cases and should have separate text descriptions.



Factoring of Use Cases

Includes: The includes relationship implies one use case includes the behavior of another use case in its sequence of events and actions. The includes relationship is appropriate when you have a chunk of behavior that is similar across a number of use cases. The factoring of such behavior will help in explores the issue of reuse by factoring out the commonality across use cases. It can also be gainfully employed to decompose a large and complex use case into more manageable parts. The includes relationship is represented using a predefined stereotype «include». In the includes relationship, a base use case compulsorily and automatically includes the behavior of the common use case.



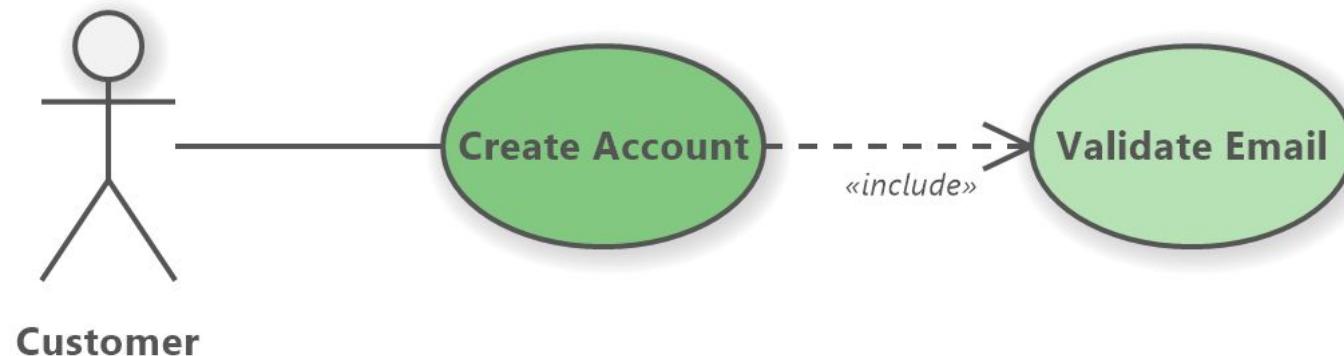
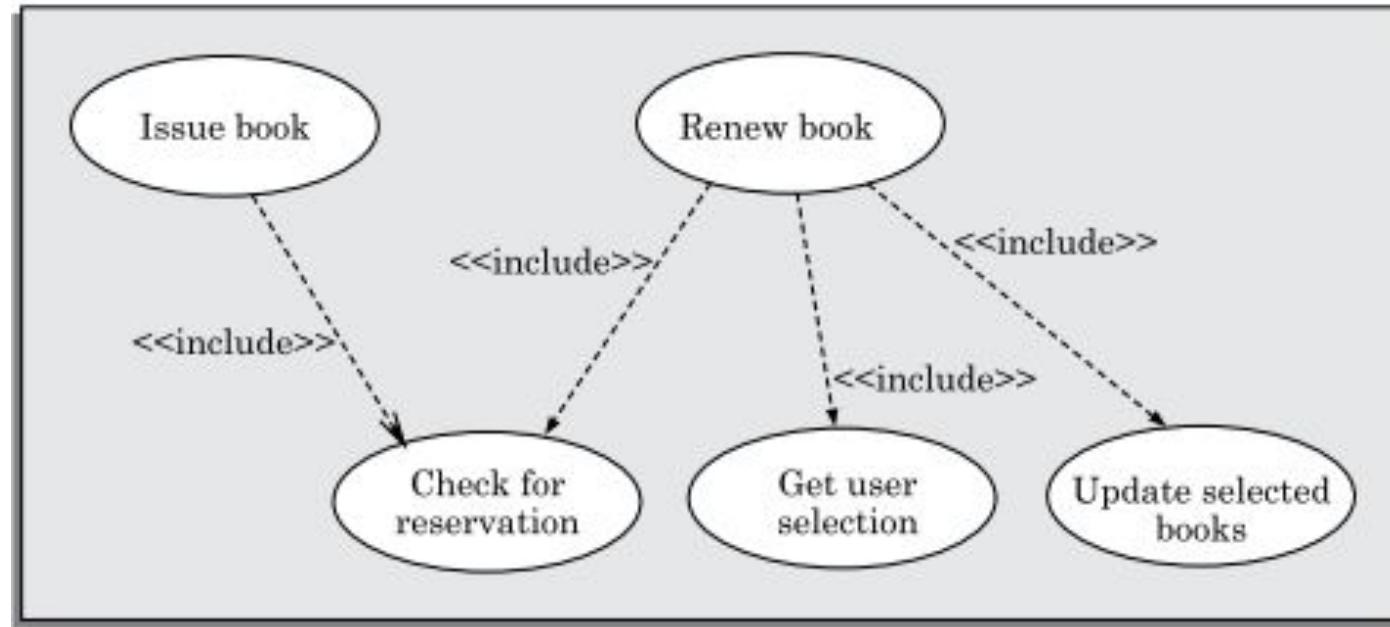
Factoring of Use Cases

Includes Example:

- 1) The use cases issue-book and renew-book both include check-reservation use case. The implication is that during execution of either of the use cases issue-book and renew-book, the use case check-reservation is executed. The base use case may include several use cases.
- 2) We have a use case called "Create Account" which represents the process of creating a new account on a website. This use case might include the use case "Validate Email," which represents the process of verifying that the email address entered by the user is valid. In this case, the "Validate Email" use case would be included in the "Create Account" use case.

Factoring of Use Cases

Includes Example:



Factoring of Use Cases

Extends: The extends relationship among use cases is that it allows one to show optional behavior that may occur during the execution of the use case. An optional system behavior is executed only if certain conditions hold, otherwise the optional behavior is not executed.



The extends relationship is similar to generalization. But unlike generalization, the extending use case can add additional behavior only at an extension point only when certain conditions are satisfied.

Factoring of Use Cases

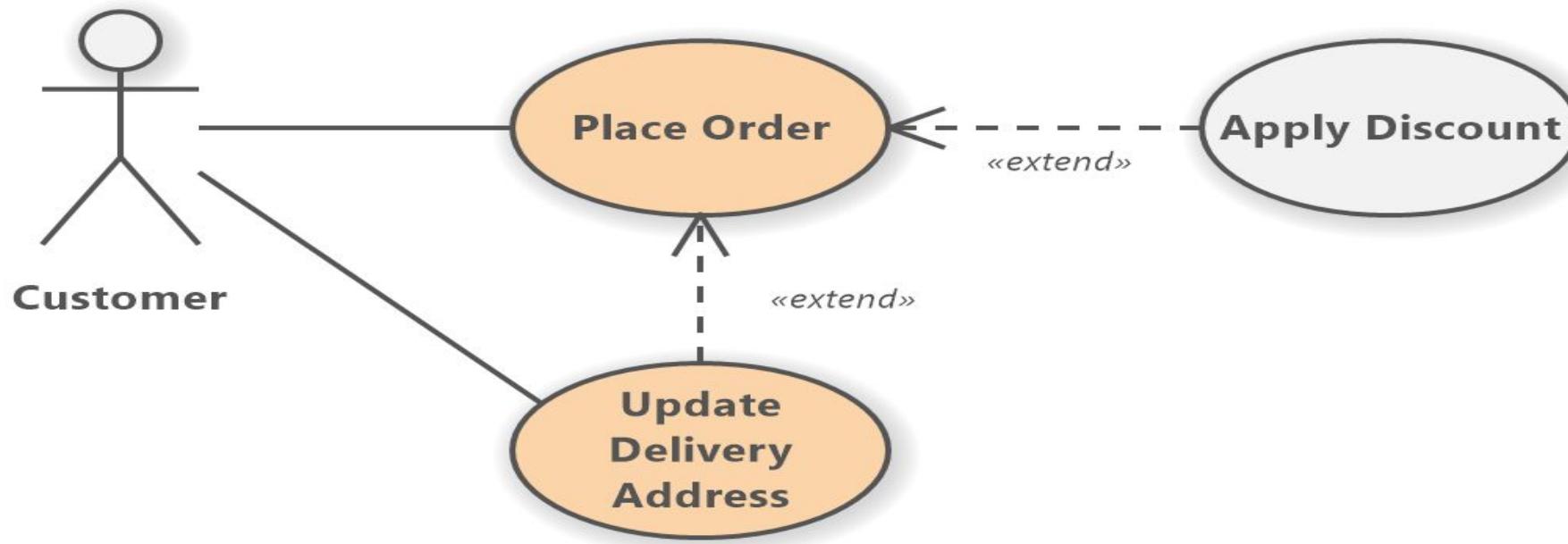
Extends Example:

- We have a use case called "Place Order" which represents the process of placing an order on an online store. This use case might be extended by the use case "Apply Discount," which represents the process of applying a discount to the order. In this case, the "Apply Discount" use case would extend the "Place Order" use case.
- We have also a use case called "Update Delivery Address" which represents the process of updating the delivery address for an existing order. This use case can be extended by the "Place Order" use case, as the customer may want to update their delivery address while placing a new order.

Factoring of Use Cases

Extends Example:

However, "Update Delivery Address" can also be used directly by the customer actor, as they may need to update their delivery address at any time before or after placing an order. In this case, the "Update Delivery Address" use case extends the "Place Order" use case, but can also be used independently by the customer actor.



Unified Modeling Language

Class Diagram

- A class diagram can describe the static structure of a simple system.
- It shows how a system is structured rather than how it behaves.
- The static structure of a more complex system usually consists of a number of class diagrams.
- Each class diagram represents classes and their inter-relationships.
- Classes may be related to each other in four ways:
 - Generalization,
 - Aggregation,
 - Association,
 - Different dependencies.

Class Diagram

Classes:

- A class represents entities (objects) with common features. That is, objects having similar attributes and operations constitute a class.
- A class is represented by a solid outlined rectangle 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 are written using a mixed case convention, but starts with a small case letter (e.g., studentMember).
- Class names are usually chosen to be singular nouns.

Class Diagram

Representations of Classes:

Classes can optionally have attributes and operations compartments. When a class appears in several diagrams, its attributes and operations are suppressed on all but one diagram.

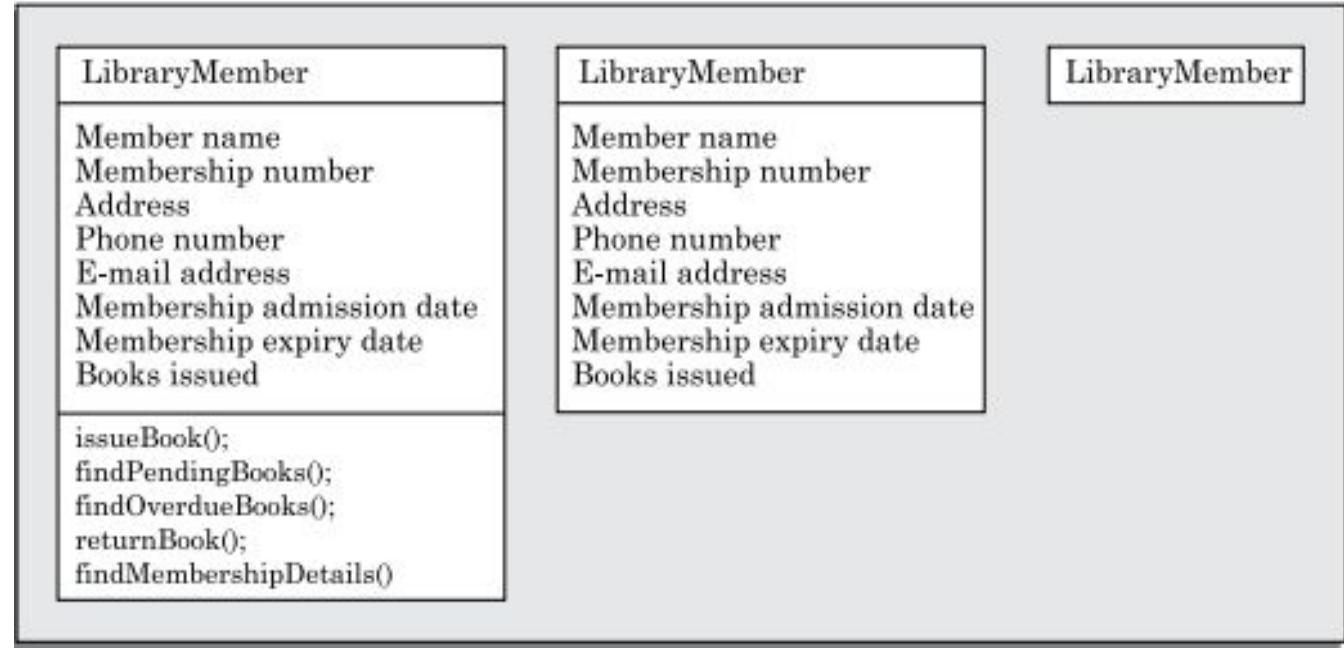


Figure: Different representations of the **LibraryMember** class.

At the start of the design process, only the names of the classes is identified. This is the most abstract representation for the class. Later in the design process the methods for the class and the attributes are identified and the more elaborate represents are used.

Class Diagram

Attributes:

- An attribute is a named property of a class. It represents the kind of data that an object might store.
- Attributes are listed by their names, and optionally their types (that is, their class, e.g., Int, Book, Employee, etc.), an initial value, and some constraints may be specified.
- Attribute names begin with a lower case letter, and are written left-justified using plain type letters.
- An attribute name may be followed by square brackets containing a multiplicity expression, e.g., sensorStatus[10]. The multiplicity expression indicates the number of attributes that would be present per instance of the class.
- The type of an attribute is written by following the attribute name with a colon and the type name, (e.g. sensorStatus:Int).

Class Diagram

Association:

- Association between two classes is represented by drawing a straight line between the concerned classes.
- The name of the association is written alongside the association line.
- An arrowhead may be placed on the association line to indicate the reading direction of the annotated association name.
- On each side of the association relation, the multiplicity is either noted as a single number or as a value range. The multiplicity indicates how many instances of one class are associated with one instance of the other class. Value ranges of multiplicity are noted by specifying the minimum and maximum value, separated by two dots, e.g., 1..5.

Class Diagram

Association:

- An asterisk is used as a wild card and means many (zero or more). The association of the given figure should be read as “Many books may be borrowed by a LibraryMember and a book may be borrowed by exactly one member”.



Figure: Association between two classes

Class Diagram

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. That is, an aggregate object not only “knows” the ids of its parts (and therefore can invoke the methods of its parts), but also takes the responsibility of creating and destroying its parts.

Aggregation is represented by an empty diamond symbol at the aggregate end of a relationship.

Class Diagram

Aggregation:

An example of the aggregation relationship has been shown in the figure. The figure represents the fact that a document can be considered as an aggregation of paragraphs. Each paragraph can in turn be considered as an aggregation of sentences. Observe that the number 1 is annotated at the diamond end, and the symbol * is annotated at the other end. This means that one document can have many paragraphs. On the other hand, if we wanted to indicate that a document consists of exactly 10 paragraphs, then we would have written number 10 in place of the symbol *.

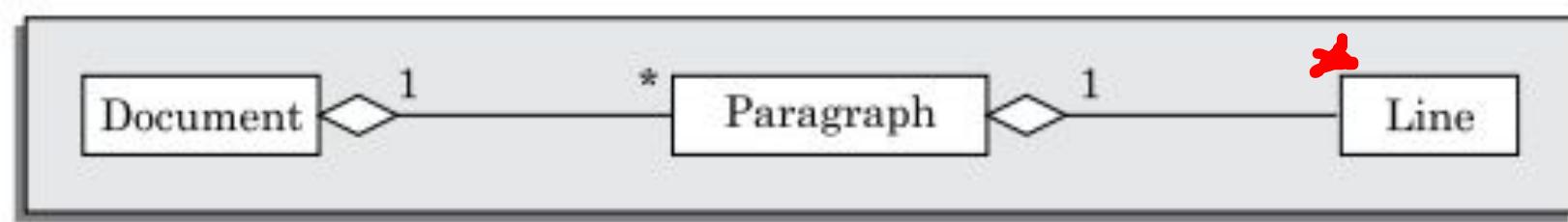


Figure: Representation of aggregation

Class Diagram

Composition:

Composition is a stricter form of aggregation, in which the parts are existence-dependent on the whole. This means that 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.

Example:

- An example of composition is an order object where after placing the order, no item in the order can be changed.
- If any changes to any of the order items are required after the order has been placed, then the entire order has to be cancelled and a new order has to be placed with the changed items.

Class Diagram

Composition:

- As soon as an order object is created, all the order items in it are created and as soon as the order object is destroyed, all order items in it are also destroyed.
- The life of the components (order items) is identical to that of the aggregate (order). The composition relationship is represented as a filled diamond drawn at the composite-end.



Figure: Representation of composition

Class Diagram

Aggregation Vs. Composition:

Both aggregation and composition represent part/ whole relationships. If components can dynamically be added to and removed from the aggregate, then the relationship is expressed as aggregation.

If the components are not required to be dynamically added/delete, then the components have the same life time as the composite, then the relationship should be represented by composition. In the implementation of a composite relationship, the component objects are created by the constructor of the composite object. Therefore, when a composite object is created, the components are automatically created. No methods in the composite class for adding, deleting, or modifying the component is implemented.

Class Diagram

Aggregation Vs. Composition:

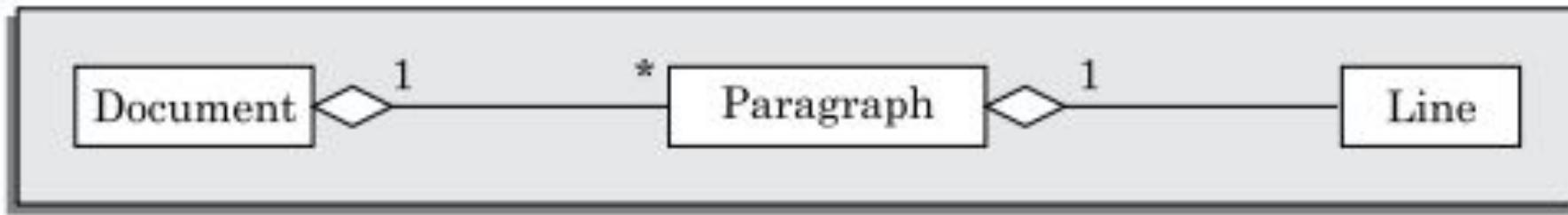


Figure: Representation of aggregation



Figure: Representation of composition

Class Diagram

Aggregation Vs. Composition:

We illustrate the subtle difference between the aggregation and composition relation between classes using the example of order and item objects. Consider that an order consists of many order items. If the order once placed, the items cannot be changed at all. In this case, the order is a composition of order items. However, if order items can be changed (added, delete, and modified) after the order has been placed, then aggregation relation can be used to model the relationship between the order and the item classes.

Class Diagram

Inheritance:

- The inheritance relationship is represented by means of an empty arrow pointing from the subclass to the superclass. The arrow may be directly drawn from the subclass to the superclass.
- When there are many subclasses of a base class, the inheritance arrow from the subclasses may be combined to form a single line (as shown in the figure) and is labelled with the specific aspect of the derived classes that is abstracted by the base class.
- The combined arrows emphasize the collectivity of the subclasses. This is useful when specialization for different groups derived classes has been done on the basis of some discriminators.

Class Diagram

Inheritance:

In the example of given figure, issuable and reference are the discriminators. This highlights the facts that some library book types are issuable, whereas other types are for reference.

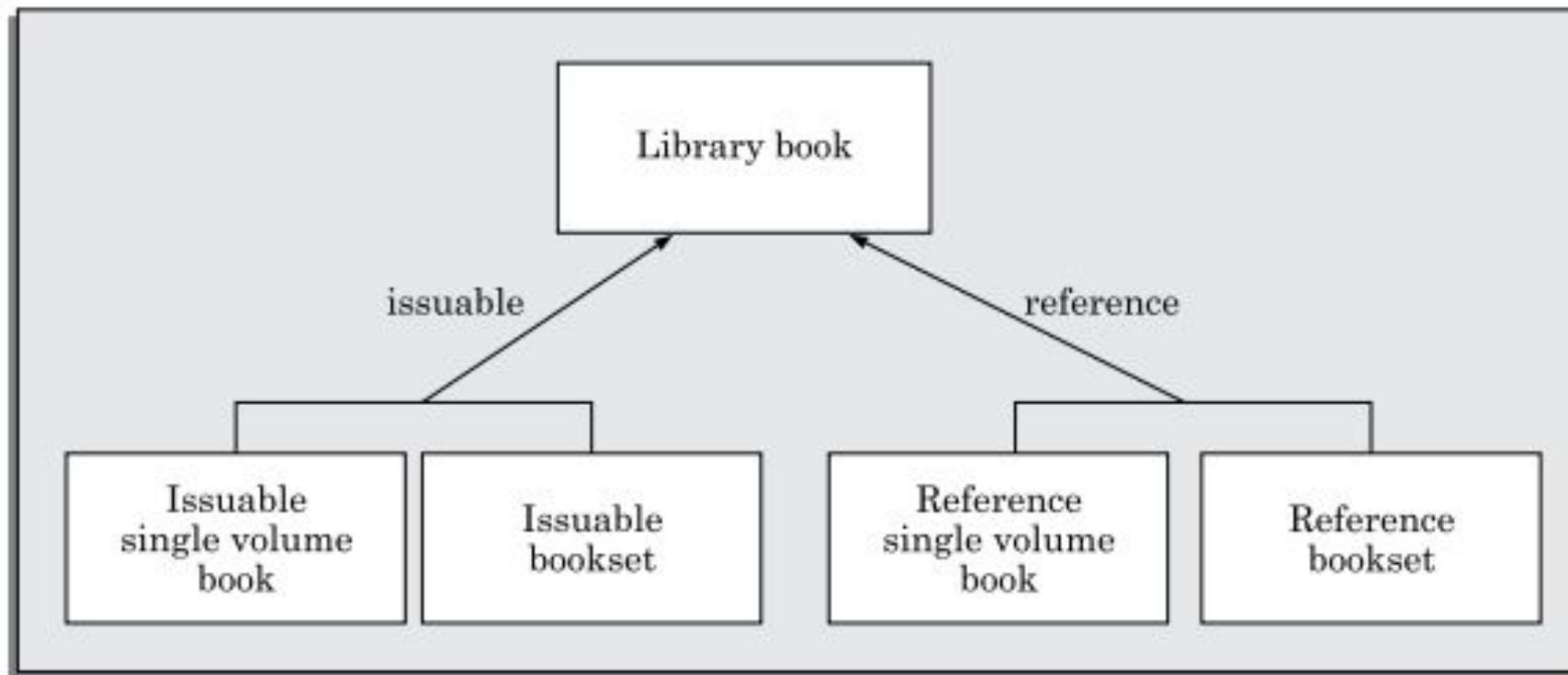


Figure: Representation of the inheritance relationship

UML: Class Diagram

Dependency Relationship

- In UML, a dependency relationship is a relationship in which one element, the client, uses or depends on another element, the supplier.
- a change to the supplier might require a change to the client.
- Dependency A dependency relationship is represented by a dotted arrow (see in the figure) that is drawn from the dependent class to the independent class.

Example: In an e-commerce application, a **Cart class depends on a Product class** because the Cart class uses the Product class as a parameter for an add operation. In a class diagram, a dependency relationship points from the Cart class to the Product class. As the following figure illustrates, the Cart class is, therefore, the client, and the Product class is the supplier.



Example of dependency Relationship

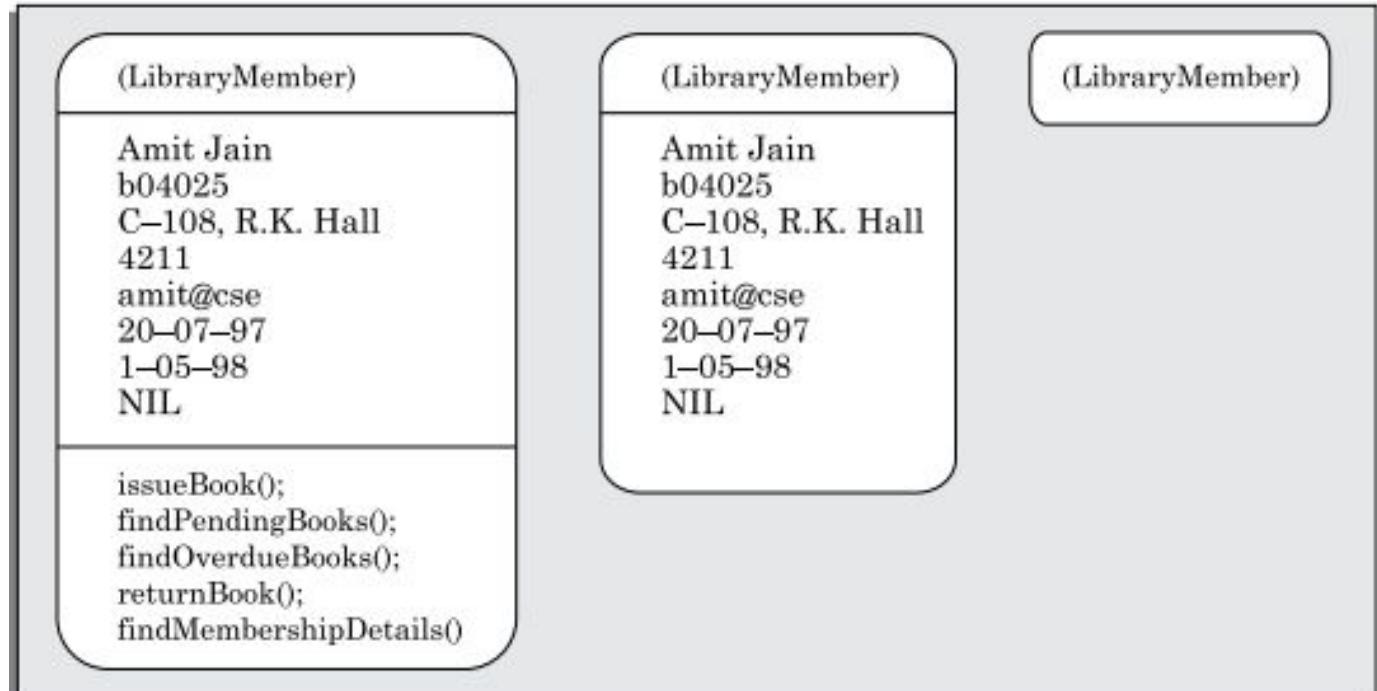
Figure: Representation of dependence between classes

Constraints

- A constraint describes either a condition that needs to be satisfied or an integrity rule for some entity.
- Constraints are typically used to describe aspects such as: permissible set of values of an attribute, specification of the pre- and post-conditions for operations, and definition of certain ordering of items.
- Example: To denote that the books in a library are maintained sorted on ISBN number, we can annotate the book class with the constraint {sorted}.
- UML allows the flexibility to use any set of words to describe the constraints. The only rule is that they are to be enclosed within braces.

Object Diagram

- During the execution of a program, objects may dynamically get created and also destroyed.
- An object diagrams shows a snapshot of the objects in a system at a point in time. Since an object diagram shows instances of classes, rather than the classes themselves, it is often called as instance diagram.
- The objects are drawn using rounded rectangles (as shown in the figure).
- As the class diagrams, the object diagrams may just indicate the name of the object or more details.



UML: Interaction Diagrams

Interaction Diagram

- When a user invokes any one of the use cases of a system, the required behavior is realized through the interaction of several objects in the system.
- An interaction diagram describes how groups of objects interact among themselves through message passing to realize some behavior (execution of a use case).
- For complex use cases, more than one interaction diagram may be necessary to capture the behavior.
- There are two kinds of interaction diagrams—
 - Sequence diagrams
 - Collaboration diagrams.
- Both Sequence and Collaboration diagrams are equivalent in the sense that any one diagram can be derived automatically from the other.

Sequence Diagram

A sequence diagram shows the interactions among objects as a two dimensional chart.

The chart is read from top to bottom.

The objects participating in the interaction are drawn 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.

When no name is specified, it indicates that we are referring any arbitrary instance of the class. For example, in the given figure, Book represents any arbitrary instance of the Book class.

Sequence Diagram

- An object appearing at the top of a sequence diagram signifies that the object existed even before the time the use case execution was initiated.
- If some object is created during the execution of a use case and participates in the interaction (e.g., a method call), then the object should be shown at the appropriate place on the diagram where it is created.

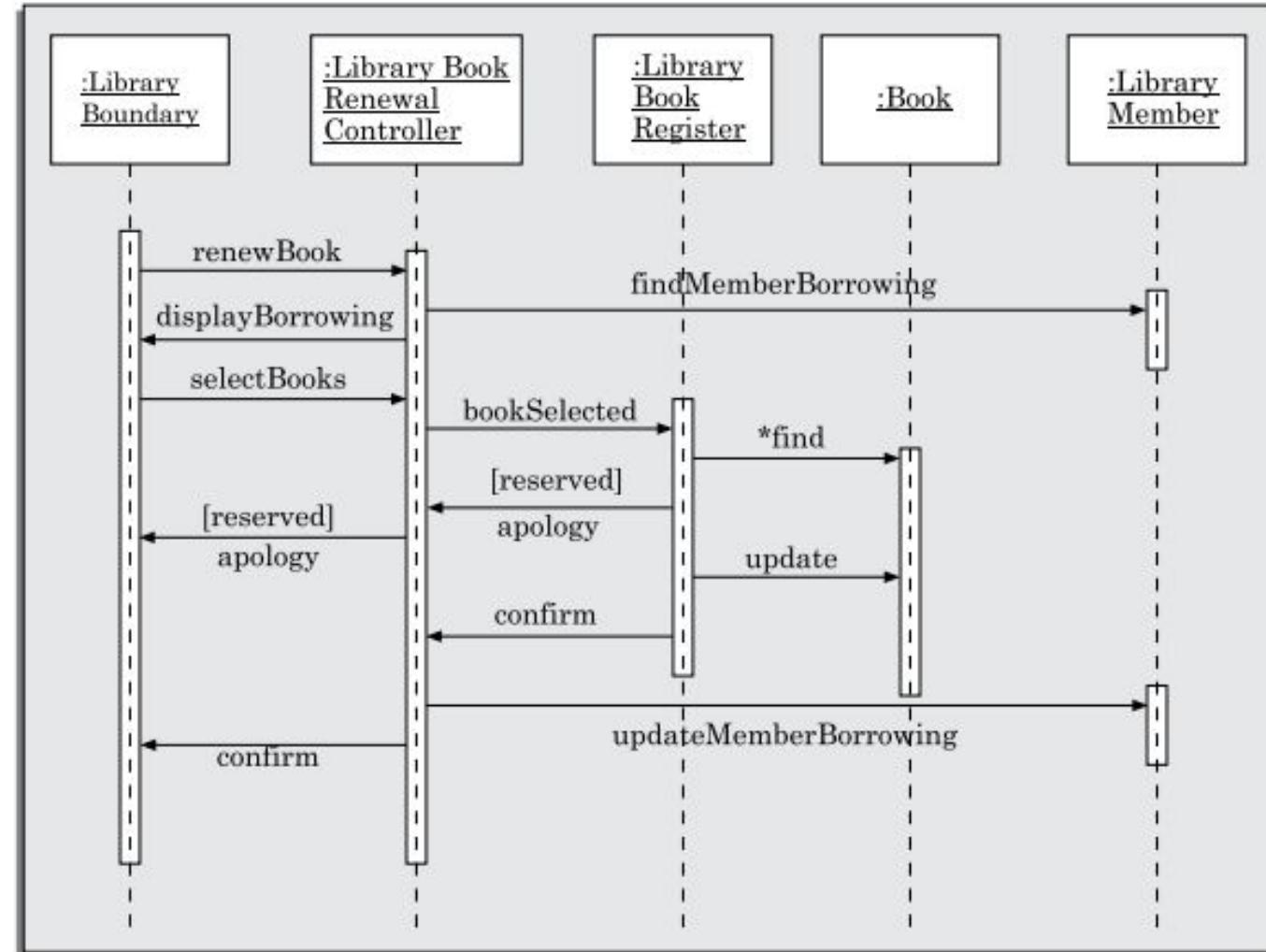


Figure: Sequence diagram for the renew book use case

Sequence Diagram

- The vertical dashed line attached to an object is called the object's lifeline. An object exists as long as its lifeline exists. Absence of lifeline after some point indicates that the object ceases to exist after that point in time.
- Normally, at a certain point if an object is destroyed, the lifeline of the object is crossed at that point and the lifeline for the object is not drawn beyond that point.
- When an object receives a message, a rectangle called the *activation symbol* is drawn on the lifeline of an object to indicate the points of time at which the object is active.
- An activation symbol indicates that an object is active as long as the symbol (rectangle) exists on the lifeline.
- Each message is indicated as an arrow between the lifelines of two objects. The messages are drawn in chronological order from the top to the bottom.
- Each message is labelled with the message name, and optionally some control information can also be included along with the message name.

Collaboration Diagram

- A collaboration diagram shows both structural and behavioral aspects explicitly. This is unlike a sequence diagram which shows only the behavioral aspects.
- The structural aspect of a collaboration diagram consists of objects and links among them indicating association between the corresponding classes.
- In this diagram, each object is also called a collaborator.
- The behavioral aspect is described by the set of messages exchanged among the different collaborators.
- The messages are numbered to indicate the order in which they occur in the corresponding sequence diagram. The link between objects is shown as a solid line and messages are annotated on the line.
- A message is drawn as a labelled arrow and is placed near the link. Messages are prefixed with sequence numbers because they are the only way to describe the relative sequencing of the messages in this diagram.

Collaboration Diagram

The collaboration diagram for the example of use-case sequence ‘renew book’ is shown in figure. A collaboration diagram explicitly shows which class is associated with other classes. Therefore, we can say that the collaboration diagram shows structural information more clearly than the sequence diagram.

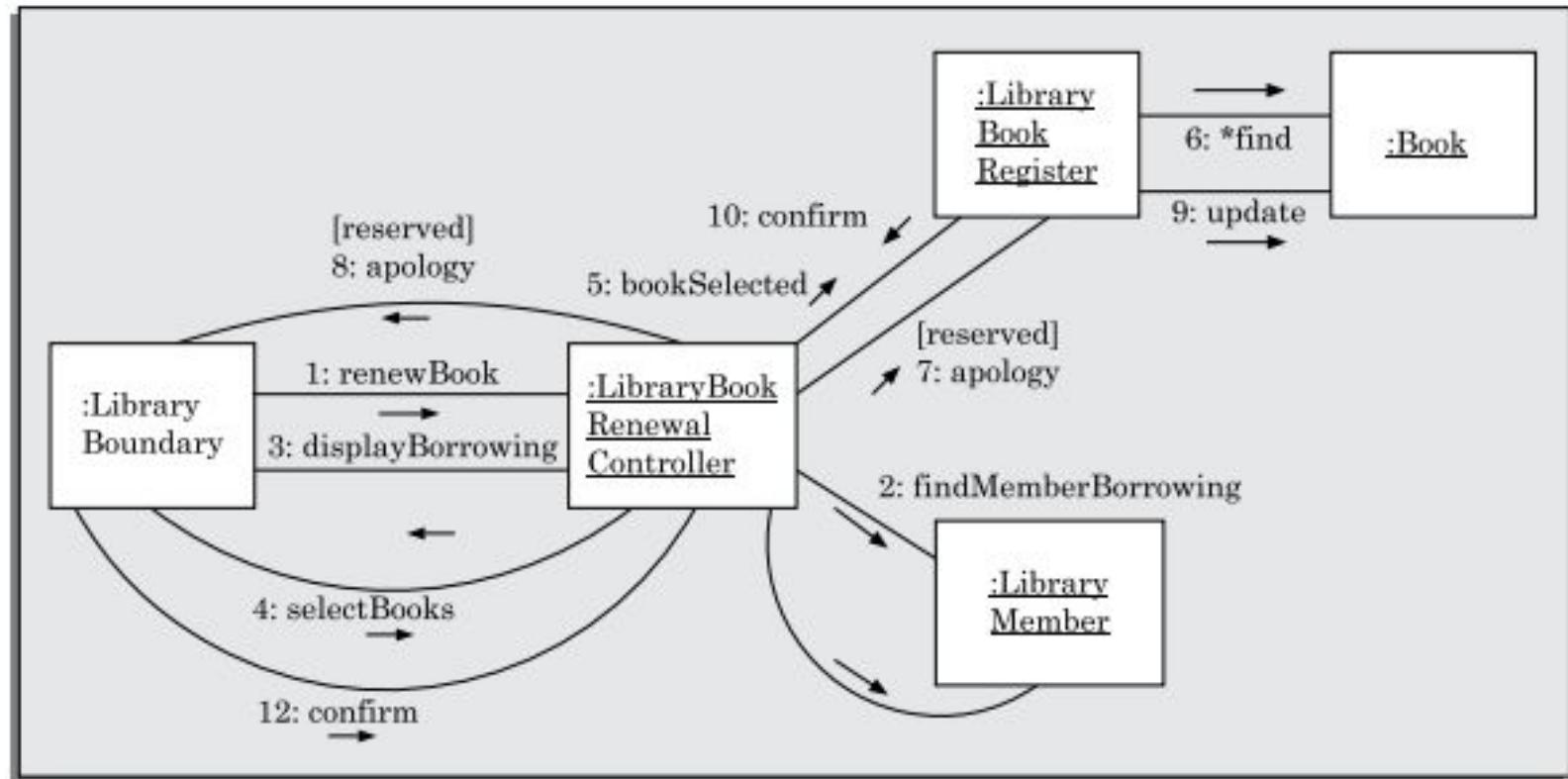


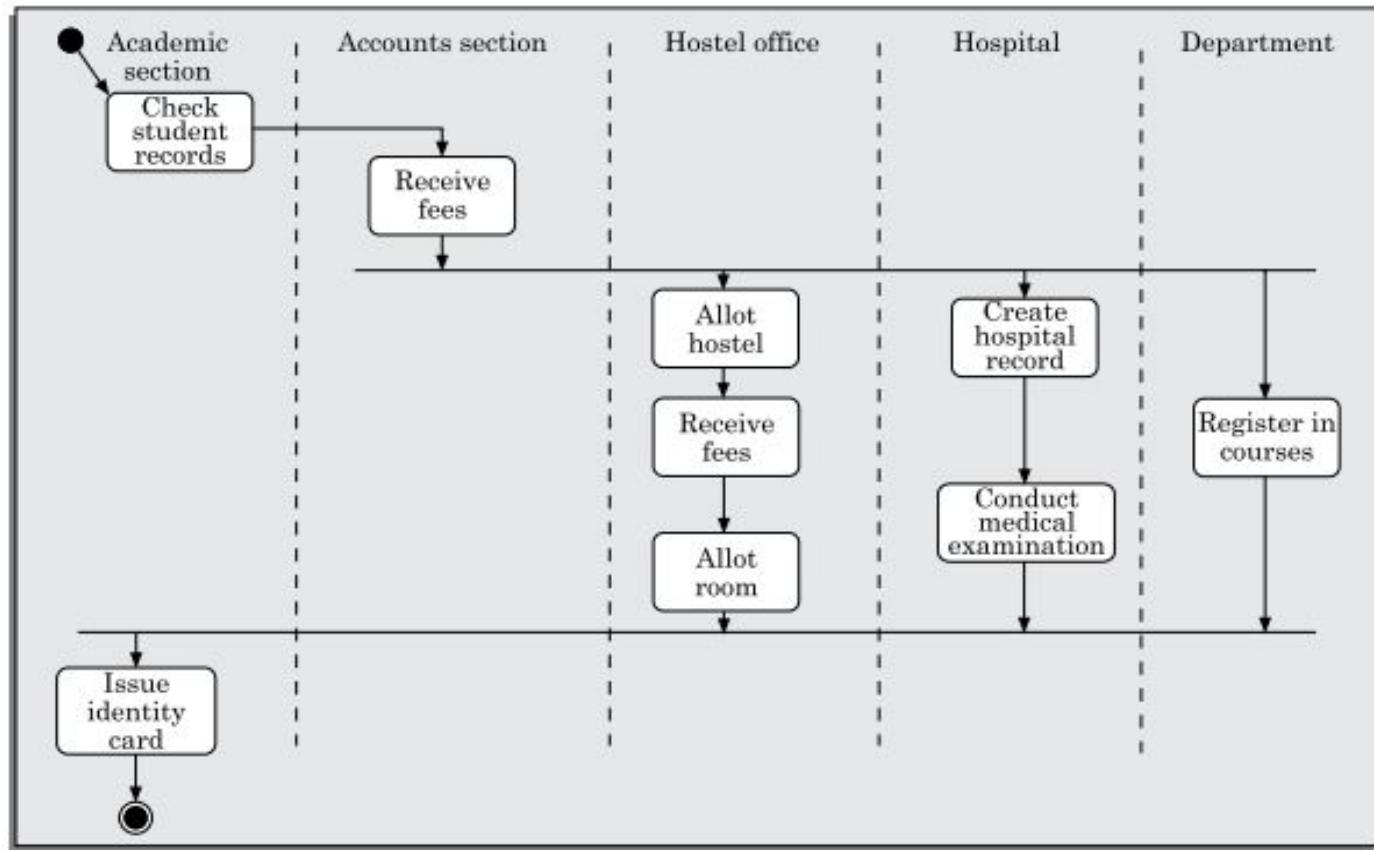
Figure: Collaboration diagram for the renew book use case

Activity Diagram

- An activity diagram can be used to represent various activities (or chunks of processing) that occur during execution of the software and their sequence of activation.
- An activity is a state with an internal action and one or more outgoing transitions. On termination of the internal activity, the appropriate transition is taken. If an activity has more than one outgoing transition, then the exact conditions under which each is executed must be identified through use of appropriate conditions.
- Activity diagrams are to some extent similar to flow charts. The main difference is that activity diagrams support description of parallel activities and synchronization aspects involved in different activities.
- Activity diagrams incorporate swim lanes to indicate which components of the software are responsible for which activities.

Activity Diagram

- Parallel activities are represented on an activity diagram by using swim lanes.
- Swim lanes make it possible to group activities based on who is performing them.
- Swim lanes subdivide activities based on the responsibilities of various components.
- For each component participating in the use case execution, it becomes clear as to the specific activities for which it is responsible.



- The swim lane corresponding to the academic section, the activities that are carried out by the academic section (check student record and issue identity card) and the specific situation in which these are carried out are shown in the figure above.

State Chart Diagram

- A state chart diagram is normally used to model how the state of an object may change over its life time.
- State chart diagrams are good at describing how the behavior of an object changes across several use case executions.
- State chart diagrams are based on the finite state machine (FSM) formalism which consists of a finite number of states corresponding to those that the object being modeled can take. The object undergoes state changes when specific events occur.

State Chart Diagram

Basic elements of a state chart

Initial state: This represented by 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. In general, the name of the event which causes the transition is placed alongside the arrow. A guard can be assigned to the transition. A guard is a Boolean logic condition. The transition can take place only if the guard evaluates to true. Transition having no event or guard annotated with it is called pseudo transition. The syntax for the label of the transition is shown in 3 parts—[guard]event/action.

State Chart Diagram

An example state chart model for the order object of the Trade House Automation software is shown in the figure. Observe that from the Rejected order state, there is an automatic and implicit transition to the end state. Such transitions which do not have any event or guard annotated with it are called pseudo transitions.

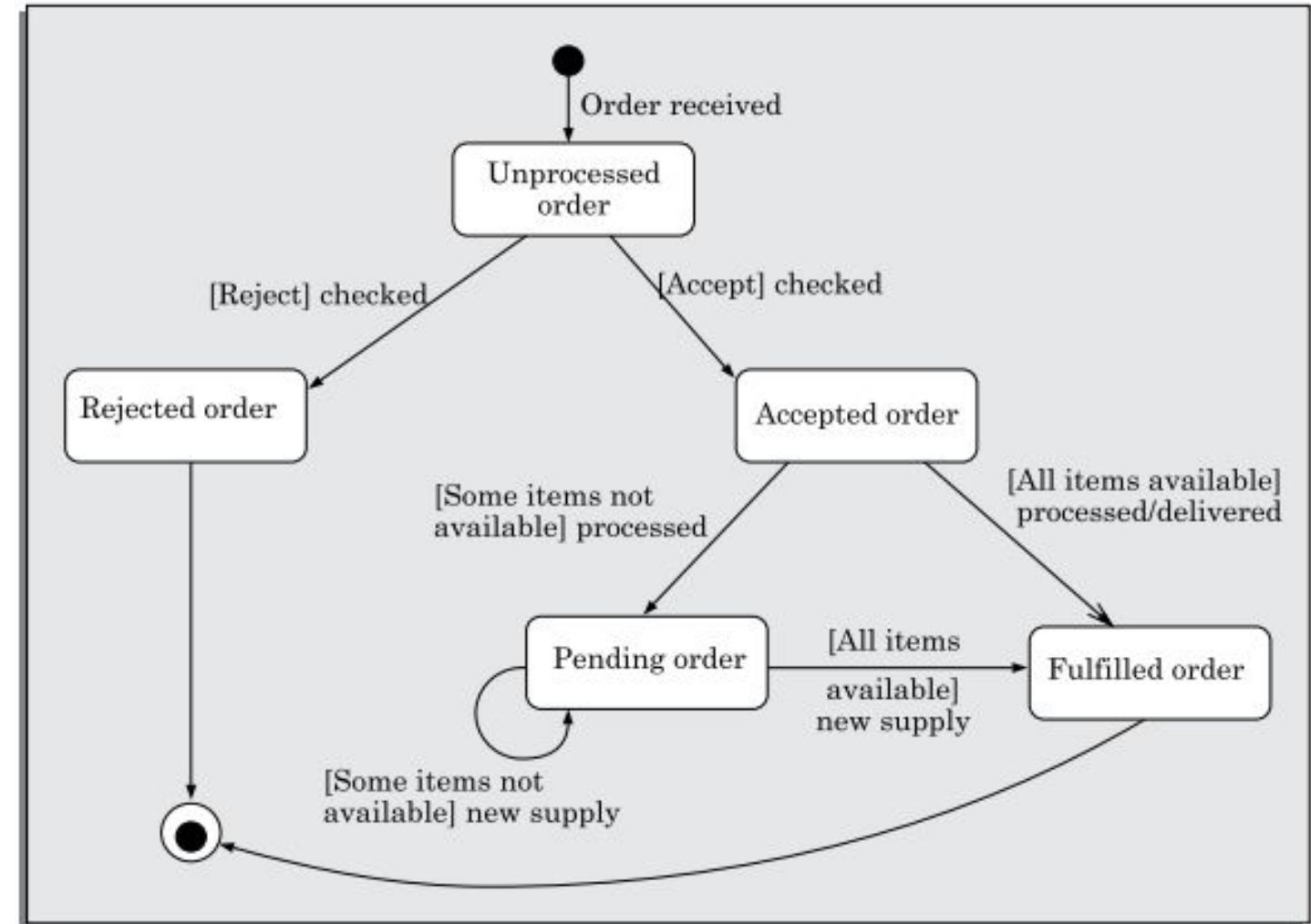


Figure: State chart diagram for an order object

Coding

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. The programmers adhere to standard and well defined style of coding which they call their coding standard. The main advantages of adhering to a standard style of coding are as follows:

- A coding standard gives uniform appearances to the code written by different engineers.
- It facilitates code of understanding.
- Promotes good programming practices.

For implementing our design into a code, we require a good high level language. A programming language should have the following features:

Characteristics of a Programming Language

Readability: A good high-level language will allow programs to be written in some ways that resemble a quite-English description of the underlying algorithms. If care is taken, the coding may be done in a way that is essentially self-documenting.

Portability: High-level languages, being essentially machine independent, should be able to develop portable software.

Generality: Most high-level languages allow the writing of a wide variety of programs, thus relieving the programmer of the need to become expert in many diverse languages.

Brevity: Language should have the ability to implement the algorithm with less amount of code. Programs expressed in high-level languages are often considerably shorter than their low-level equivalents.

Error checking: Being human, a programmer is likely to make many mistakes in the development of a computer program. Many high-level languages enforce a great deal of error checking both at compile-time and at run-time.

Cost: The ultimate cost of a programming language is a function of many of its characteristics.

Characteristics of a Programming Language

Familiar notation: A language should have familiar notation, so it can be understood by most of the programmers.

Quick translation: It should admit quick translation.

Efficiency: It should permit the generation of efficient object code.

Modularity: It is desirable that programs can be developed in the language as a collection of separately compiled modules, with appropriate mechanisms for ensuring self-consistency between these modules.

Widely available: Language should be widely available and it should be possible to provide translators for all the major machines and for all the major operating systems.

Coding Standards and Guidelines

Good software development organizations usually develop their own coding standards and guidelines depending on what best suits their organization and the type of products they develop. The following are some representative coding standards.

1. Rules for limiting the use of global: These rules list what types of data can be declared global and what cannot.
2. Contents of the headers preceding codes for different modules: The information contained in the headers of different modules should be standard for an organization. The exact format in which the header information is organized in the header can also be specified.

The following are some standard header data:

- Name of the module.
- Date on which the module was created.
- Author's name.
- Modification history.
- Synopsis of the module.
- Different functions supported, along with their input/output parameters.
- Global variables accessed/modified by the module.

Coding Standards and Guidelines

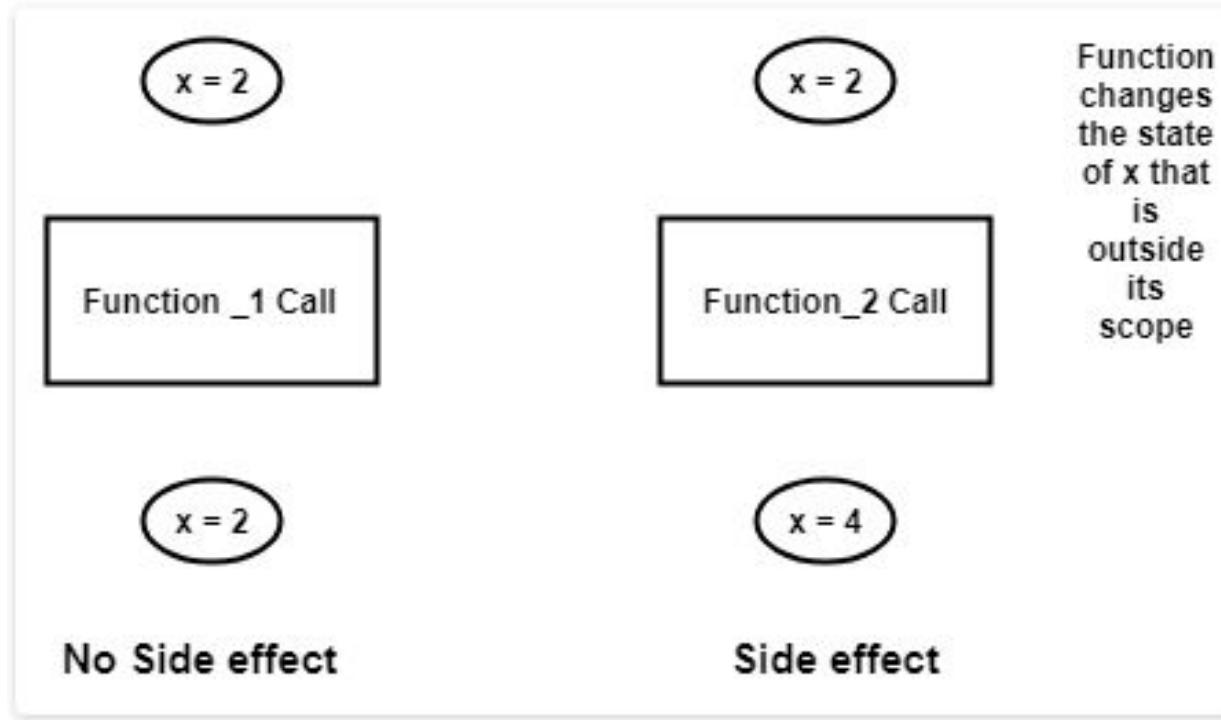
3. Naming conventions for global variables, local variables, and constant identifiers: A possible naming convention can be that global variable names always start with a capital letter, local variable names are made of small letters, and constant names are always capital letters.
4. Error return conventions and exception handling mechanisms: The way error conditions are reported by different functions in a program are handled should be standard within an organization. For example, different functions while encountering an error condition should either return a 0 or 1 consistently.

Coding Standards and Guidelines

The following are some representative coding guidelines recommended by many software development organizations.

1. Do not use a coding style that is too clever or too difficult to understand: Code should be easy to understand. Many inexperienced engineers actually take pride in writing cryptic and incomprehensible code. Clever coding can obscure meaning of the code and hamper understanding. It also makes maintenance difficult.
2. Avoid obscure side effects: The side effects of a function call include modification of parameters passed by reference, modification of global variables, and I/O operations. An obscure side effect is one that is not obvious from a casual examination of the code. Obscure side effects make it difficult to understand a piece of code. For example, if a global variable is changed obscurely in a called module or some file I/O is performed which is difficult to infer from the function's name and header information, it becomes difficult for anybody trying to understand the code.

Coding Standards and Guidelines



Modifying a non-local variable:

```
1 x=3
2 def square(x):
3     x=x*x
4     return x
5 def square_side_effect():
6     global x
7     x=x*x
8     return x
9 print(x)          # 3
10 print(square(x)) # 9
11 print(x)          # 3
12 print(square_side_effect()) # 9
13 print(x)          # 9
```

Figures: Examples of obscure side effects

Coding Standards and Guidelines

3. Do not use an identifier for multiple purposes: Programmers often use the same identifier to denote several temporary entities. For example, some programmers use a temporary loop variable for computing and a storing the final result. The rationale that is usually given by these programmers for such multiple uses of variables is memory efficiency, e.g. three variables use up three memory locations, whereas the same variable used in three different ways uses just one memory location.

However, there are several things wrong with this approach and hence should be avoided. Some of the problems caused by use of variables for multiple purposes as follows:

- Each variable should be given a descriptive name indicating its purpose. This is not possible if an identifier is used for multiple purposes. Use of a variable for multiple purposes can lead to confusion and make it difficult for somebody trying to read and understand the code.
- Use of variables for multiple purposes usually makes future enhancements more difficult.

Coding Standards and Guidelines

4. The code should be well-documented: As a rule of thumb, there must be at least one comment line on the average for every three-source line.
5. The length of any function should not exceed 10 source lines: A function that is very lengthy is usually very difficult to understand as it probably carries out many different functions. For the same reason, lengthy functions are likely to have disproportionately larger number of bugs.
6. Avoid goto statements: Use of goto statements makes a program unstructured and very difficult to understand.

Coding

Code Review

Code review for a model is carried out after the module is successfully compiled and all the syntax errors have been eliminated.

Code reviews are extremely cost-effective strategies for reduction in coding errors and to produce high quality code.

In general, two types of reviews are carried out on the code of a module-

- Code walkthrough
- Code Inspection

Code Walkthrough

Code walk through is an informal code analysis technique. The main objectives of the walk through are to discover the algorithmic and logical errors in the code.

After a module has been coded, successfully compiled and all syntax errors eliminated, a few members of the development team are given the code few days before the walk through meeting to read and understand code.

Each member selects some test cases and simulates execution of the code by hand (i.e. trace execution through each statement and function execution).

The members note down their findings to discuss these in a walk through meeting where the coder of the module is present.

Although, it is an informal analysis technique, several guidelines have evolved over the years for making this naïve but useful analysis technique more effective. These guidelines are based on personal experience, common sense, and several subjective factors.

Code Walkthrough

Walkthrough Guidelines:

- The team performing code walk through should not be either too big or too small. Ideally, it should consist of between three to seven members.
- Discussion should focus on discovery of errors and not on how to fix the discovered errors.
- In order to foster cooperation and to avoid the feeling among engineers that they are being evaluated in the code walk through meeting, managers should not attend the walk through meetings.

Code Inspection

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

During code inspection the code is examined for the presence of certain kinds of errors, in contrast to the hand simulation of code execution done in code walk throughs.

The classical error of writing a procedure that modifies a formal parameter while the calling routine calls that procedure with a constant actual parameter. It is more likely that such an error will be discovered by looking for these kinds of mistakes in the code, rather than by simply hand simulating execution of the procedure.

In addition to the commonly made errors, adherence to coding standards is also checked during code inspection.

Good software development companies collect statistics regarding different types of errors commonly committed by their engineers and identify the type of errors most frequently committed. Such a list of commonly committed errors can be used during code inspection to look out for possible errors.

Code Inspection

Following is a list of some classical programming errors which can be checked during code inspection:

- Use of uninitialized variables.
- Jumps into loops.
- Nonterminating loops.
- Incompatible assignments.
- Array indices out of bounds.
- Improper storage allocation and deallocation.
- Mismatches 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 equally of floating point variables, etc.

Clean Room Testing

- Clean room testing was pioneered by IBM. This type of testing relies heavily on walk throughs, 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. The software development philosophy is based on avoiding software defects by using a rigorous inspection process.
- The name ‘clean room’ was derived from the analogy with semi-conductor fabrication units. In these units (clean rooms), defects are avoided by manufacturing in ultra-clean atmosphere.
- In this kind of development, inspections to check the consistency of the components with their specifications has replaced unit-testing.

Clean Room Testing

The clean room approach to software development is based on five characteristics:

Formal specification: The software to be developed is formally specified. A state transition model which shows system responses to stimuli is used to express the specification.

Incremental development: The software is partitioned into increments which are developed and validated separately using the clean room process. These increments are specified, with customer input, at an early stage in the process.

Structured programming: Only a limited number of control and data abstraction constructs are used. The program development process is process of stepwise refinement of the specification.

Static verification: The developed software is statically verified using rigorous software inspections. There is no unit or module testing process for code components

Statistical testing of the system: The integrated software increment is tested statistically to determine its reliability. These statistical tests are based on the operational profile which is developed in parallel with the system specification.

Software Documentation

When various kinds of software products are developed then not only the executable files and the source code are developed but also various kinds of documents such as users' manual, software requirements specification (SRS) documents, design documents, test documents, installation manual, etc are also developed as part of any software engineering process. All these documents are a vital part of good software development practice.

Good documents are very useful and serve the following purposes:

- Good documents enhance understandability and maintainability of a software product. They reduce the effort and time required for maintenance.
- Use documents help the users in effectively using the system.
- Good documents help in effectively handling the manpower turnover problem. Even when an engineer leaves the organization, and a new engineer comes in, he can build up the required knowledge easily.
- Production of good documents helps the manager in effectively tracking the progress of the project. The project manager knows that measurable progress is achieved if a piece of work is done and the required documents have been produced and reviewed.

Software Documentation

Different types of software documents can broadly be classified into the following:

Internal documentation is the code comprehension features provided as part of the source code itself. Internal documentation is provided through appropriate module headers and comments embedded in the source code. Internal documentation is also provided through the useful variable names, module and function headers, code indentation, code structuring, use of enumerated types and constant identifiers, use of user-defined data types, etc.

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

Testing

Testing

Testing a program consists of providing the program with a set of test inputs (or test cases) and observing if the program behaves as expected. If the program fails to behave as expected, then the conditions under which failure occurs are noted for later debugging and correction. Some commonly used terms associated with testing are:

Failure: This is a manifestation of an error (or defect or bug). But, the mere presence of an error may not necessarily lead to a failure.

Test case: This is the triplet $[I, S, O]$, where I is the data input to the system, S is the state of the system at which the data is input, and O is the expected output of the system.

Test suite: This is the set of all test cases with which a given software product is to be tested.

Aim of Testing

The aim of the testing process is to identify all defects existing in a software product. For most practical systems, even after satisfactorily carrying out the testing phase, it is not possible to guarantee that the software is error free. This is because of the fact that the input data domain of most software products is very large. It is not practical to test the software exhaustively with respect to each value that the input data may assume. Even with this practical limitation of the testing process, the importance of testing should not be underestimated. It must be remembered that testing does expose many defects existing in a software product. Thus testing provides a practical way of reducing defects in a system and increasing the users' confidence in a developed system.

Verification Vs Validation

Verification is the process of determining whether the output of one phase of software development conforms to that of its previous phase, whereas validation is the process of determining whether a fully developed system conforms to its requirements specification. Thus while verification is concerned with phase containment of errors, the aim of validation is that the final product be error free.

Design of Test Cases

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.

An optional test suite must be signed that is of reasonable size and can uncover as many errors existing in the system as possible.

If test cases are **selected randomly**, many of these randomly selected test cases do not contribute to the significance of the test suite, i.e. they do not detect any additional defects not already being detected by other test cases in the suite.

Thus, the number of random test cases in a test suite is, in general, not an indication of the effectiveness of the testing.

Testing a system using a large collection of test cases that are selected at random does not guarantee that all (or even most) of the errors in the system will be uncovered.

Design of Test Cases

Consider the following example code segment which finds the greater of two integer values x and y . This code segment has a simple programming error.

```
if ( $x > y$ )
```

```
max =  $x$ ;
```

```
else
```

```
max =  $x$ ;
```

For the above code segment, the test suite, $\{(x=3,y=2);(x=2,y=3)\}$ can detect the error, whereas a larger test suite $\{(x=3,y=2);(x=4,y=3);(x=5,y=1)\}$ does not detect the error. So, it would be incorrect to say that a larger test suite would always detect more errors than a smaller one, unless of course the larger test suite has also been carefully designed. This implies that the test suite should be carefully designed than picked randomly. 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.

Design of Test Cases

Functional Testing Vs. Structural Testing

In the black-box testing approach, test cases are designed using only the functional specification of the software, i.e. without any knowledge of the internal structure of the software. For this reason, black-box testing is known as functional testing. On the other hand, in the white-box testing approach, designing test cases requires thorough knowledge about the internal structure of software, and therefore the white-box testing is called structural testing.

Black-Box Testing

In the 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 to designing black box test cases.

- Equivalence class partitioning
- Boundary value analysis

Equivalence Class Partitioning

- The domain of input values to a program is partitioned into a set of equivalence classes.
- This partitioning is done such that the behavior of the program is similar for every input data belonging to the same equivalence class.
- Testing the code with any one value belonging to an equivalence class is as good as testing the software with any other value belonging to that equivalence class.
- Equivalence classes for a software can be designed by examining the input data and output data.

Equivalence Class Partitioning

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

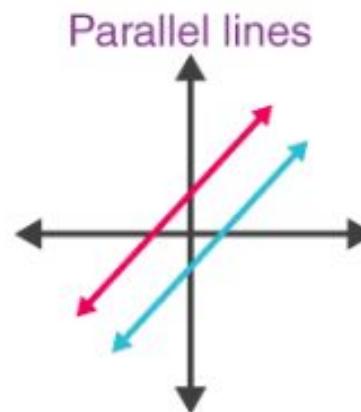
1. If the input data values to a system can be specified by a range of values, then one valid and two invalid equivalence classes should be defined.
2. If the input data assumes values from a set of discrete members of some domain, then one equivalence class for valid input values and another equivalence class for invalid input values should be defined.

Example 1: For a software that computes the square root of an input integer which can assume values in the range of 0 to 5000, 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.

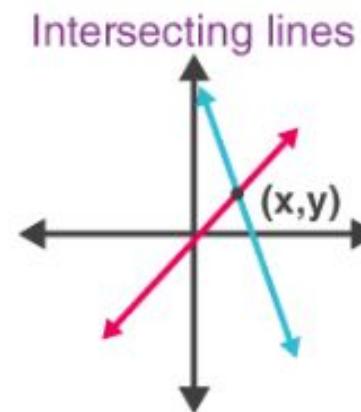
Therefore, the test cases must include representatives for each of the three equivalence classes and a possible test set can be: {-5,500,6000}.

Equivalence Class Partitioning

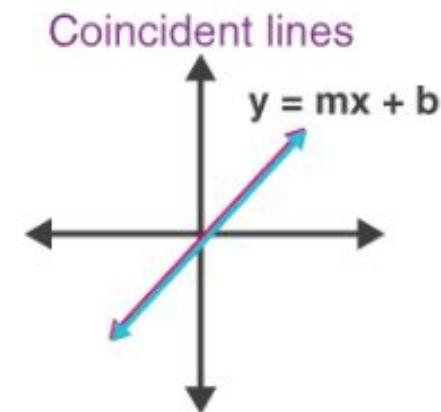
Example 2: Design the black-box test suite for the following program. The program computes the intersection point of two straight lines and displays the result. It reads two integer pairs (m_1, c_1) and (m_2, c_2) defining the two straight lines of the form $y=mx + c$.



No points in common
Solution ; \emptyset



One point in common
Solution ; (x, y)



Infinity many points in
common;
 $\{(x, y) : y = mx + b\}$

The equivalence classes are the following:

- Parallel lines ($m_1=m_2, c_1 \neq c_2$)
- Intersecting lines ($m_1 \neq m_2$)
- Coincident lines ($m_1=m_2, c_1=c_2$)

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.

Boundary Value Analysis

A type of programming error frequently occurs at the boundaries of different equivalence classes of inputs. The reason behind such errors might purely be due to psychological factors. Programmers often fail to see 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 `<`. Boundary value analysis leads to selection test cases at the boundaries of the different equivalence classes.

Example 1: 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}.

Example 2: Assume, we have to test a field which accepts Age 18 – 56.

Minimum boundary value is 18
Maximum boundary value is 56
Valid Inputs: 18, 19, 55, 56
Invalid Inputs: 17 and 57

AGE	Enter Age	*Accepts value 18 to 56
BOUNDARY VALUE ANALYSIS		
Invalid (min -1)	Valid (min, +min, -max, max)	Invalid (max +1)
17	18, 19, 55, 56	57

White-Box Testing

White-Box Testing

White-box testing is an important type of unit testing. A large number of white-box testing strategies exist. Each testing strategy essentially designs test cases based on analysis of some aspect of source code and is based on some heuristic.

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. 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, and path coverage-based testing.

Coverage-based Testing

Testing criterion for coverage-based testing:

A coverage-based testing strategy typically targets to execute (i.e., cover) certain program elements for discovering failures.

For example, if a testing strategy requires all the statements of a program to be executed at least once, then we say that the testing criterion of the strategy is statement coverage.

A test suite is adequate with respect to a criterion, if it covers all program elements of the domain defined by that criterion.

Stronger versus weaker testing

When none of two testing strategies fully covers the program elements exercised by the other, then the two are called complementary testing strategies.

The concepts of stronger, weaker, and complementary testing are schematically illustrated in the given figure.

In the Figure(a), testing strategy A is stronger than B since B covers only a proper subset of elements covered by A. On the other hand, Figure(b) shows A and B are complementary testing strategies since some elements of A are not covered by B and vice versa.

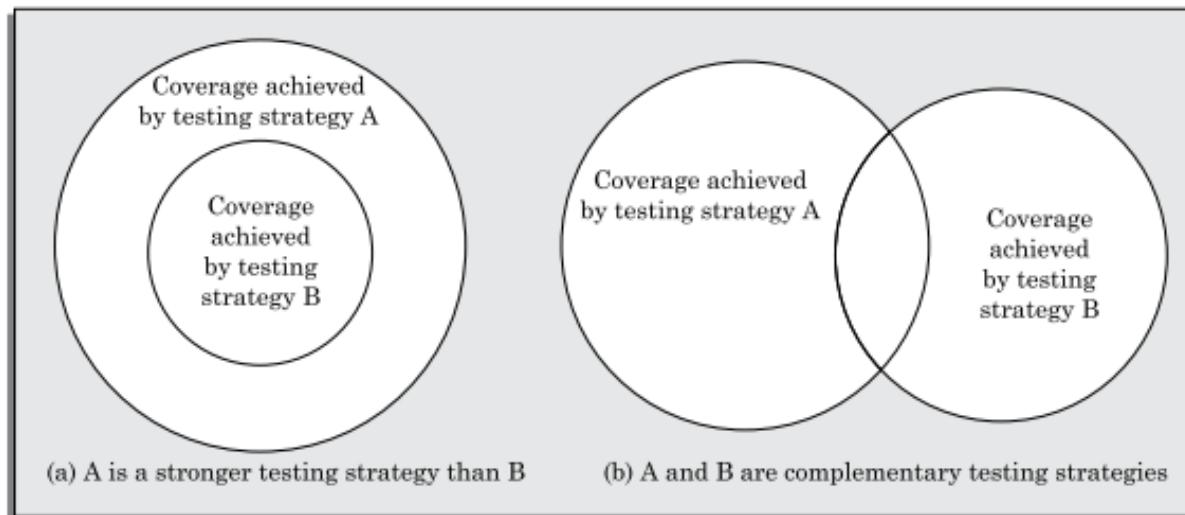


Figure: Illustration of stronger, weaker, and complementary testing strategies.

Coverage-based Testing

Testing criterion for coverage-based testing:

A coverage-based testing strategy typically targets to execute (i.e., cover) certain program elements for discovering failures.

For example, if a testing strategy requires all the statements of a program to be executed at least once, then we say that the testing criterion of the strategy is statement coverage.

A test suite is adequate with respect to a criterion, if it covers all program elements of the domain defined by that criterion.

Control Flow Testing

- Control flow testing is a testing technique to determine the execution order of statements or instructions of the program through a control structure.
- The control structure of a program is used to develop a test case for the program.
- In this technique, a particular part of a large program is selected by the tester to set the testing path.
- It is mostly used in unit testing.
- Test cases represented by the control graph of the program.

Control Flow Graph

Control Flow Graph is formed from the node, edge, decision node, junction node to specify all possible execution path.

Notations used for Control Flow Graph

- Node
- Edge
- Decision Node
- Junction node

Node

- Nodes in the control flow graph are used to create a path of procedures. Basically, it represents the sequence of procedures which procedure is next to come.

Control Flow Graph

Example:

```
public class VoteEligibilityAge{  
    public static void main(String []args){  
        int n=45;  
        if(n>=18)  
        {  
            System.out.println("You are eligible for voting");  
        } else  
        {  
            System.out.println("You are not eligible for voting");  
        }  
    }  
}
```

Control Flow Graph

Example:

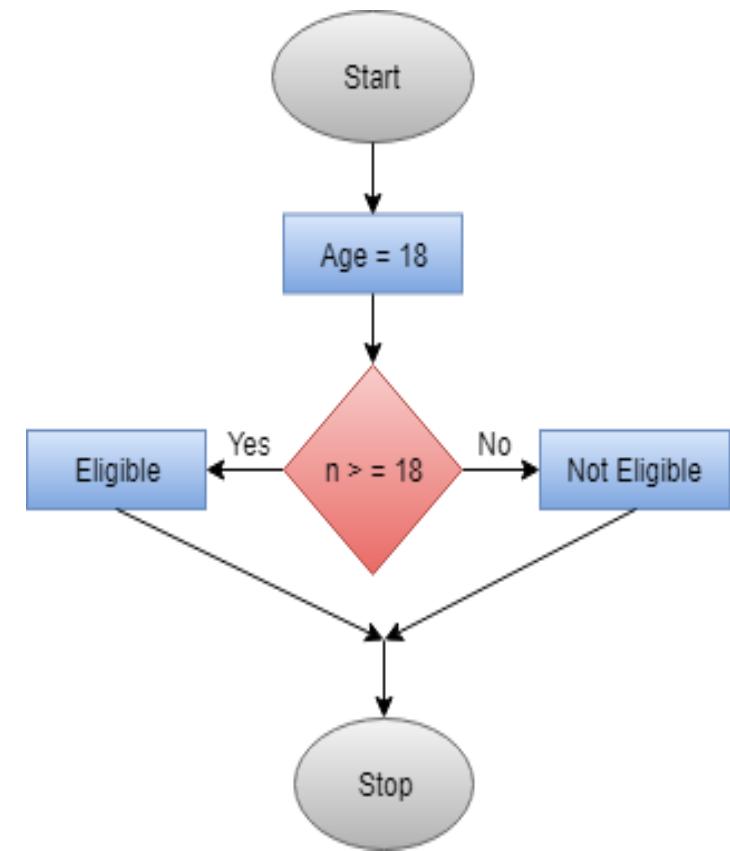
- In the example, the first node represent the start procedure and the next procedure is to assign the value of n.
- After assigning the value there is decision node to decide next node of procedure as per the value of n if it is 18 or more than 18 so Eligible procedure will execute otherwise if it is less than 18 Not Eligible procedure executes.
- The next node is the junction node, and the last node is stop node to stop the procedure.

Control Flow Graph

The example shows eligibility criteria of age for voting where if age is 18 or more than 18 so print message "You are eligible for voting" if it is less than 18 then print "You are not eligible for voting."

In the control flow graph, start, age, eligible, not eligible and stop are the nodes, $n \geq 18$ is a decision node to decide which part (if or else) will execute as per the given value. Connectivity of the eligible node and not eligible node is there on the stop node.

Test cases are designed through the flow graph of the programs to determine the execution path is correct or not. All nodes, junction, edges, and decision are the essential parts to design test cases.



Statement Coverage Testing

- This technique involves execution of all statements of the source code at least once.
- It is used to calculate the total number of executed statements in the source code out of total statements present in the source code.
- Statement coverage derives scenario of test cases under the white box testing process which is based upon the structure of the code.

Statement Coverage= (Number of executed statements/Total number of statements)*100

- Generally, in the internal source code, there is a wide variety of elements like operators, methods, arrays, looping, control statements, exception handlers, etc. Based on the input given to the program, some code statements are executed and some may not be executed. The goal of statement coverage technique is to cover all the possible executing statements and path lines in the code.

Statement Coverage Testing

Example:

```
print (int a, int b) {  
    int sum = a+b;  
    if (sum>0)  
        print ("This is a positive result")  
    else  
        print ("This is negative result")  
}
```

Statement Coverage Testing

Example:

Test suite-1:

If a = 5, b = 4

- In this scenario, the value of sum will be 9 that is greater than 0 and as per the condition result will be "**This is a positive result**".
- To calculate statement coverage of this scenario, take the total number of statements that is 7 and the number of executed statements is 5.
- Total number of statements = 7
- Number of executed statements = 5
- So, statement coverage=(5/7)*100, that is, 71%

Statement Coverage Testing

Example:

Test suite-1:

If a = -2, b = -7

- In scenario 2, we can see the value of sum will be -9 that is less than 0 and as per the condition, result will be "**This is a negative result.**"
- To calculate statement coverage of this scenario, take the total number of statements that is 7 and the number of executed statements is 6.
- Total number of statements = 7
- Number of executed statements = 6
- So, statement coverage=(6/7)*100, that is, 85%

Branch Coverage Testing

- Branch coverage technique is used to cover all branches of the control flow graph.
- It covers all the possible outcomes (true and false) of each condition of decision point at least once.
- Branch coverage technique is a whitebox testing technique that ensures that every branch of each decision point must be executed.
- However, branch coverage technique and decision coverage technique are very similar, but there is a key difference between the two. Decision coverage technique covers all branches of each decision point whereas branch testing covers all branches of every decision point of the code.
- The most basic metrics for finding the percentage of program and paths of execution during the execution of the program.
- It uses a control flow graph to calculate the number of branches.

How to calculate Branch coverage?

- There are several methods to calculate Branch coverage, but pathfinding is the most common method.
- In path finding method, the number of paths of executed branches is used to calculate Branch coverage. Branch coverage technique can be used as the alternative of decision coverage.

Example:

Read X

Read Y

IF X+Y > 100 THEN

Print "Large"

ENDIF

If X + Y<100 THEN

Print "Small"

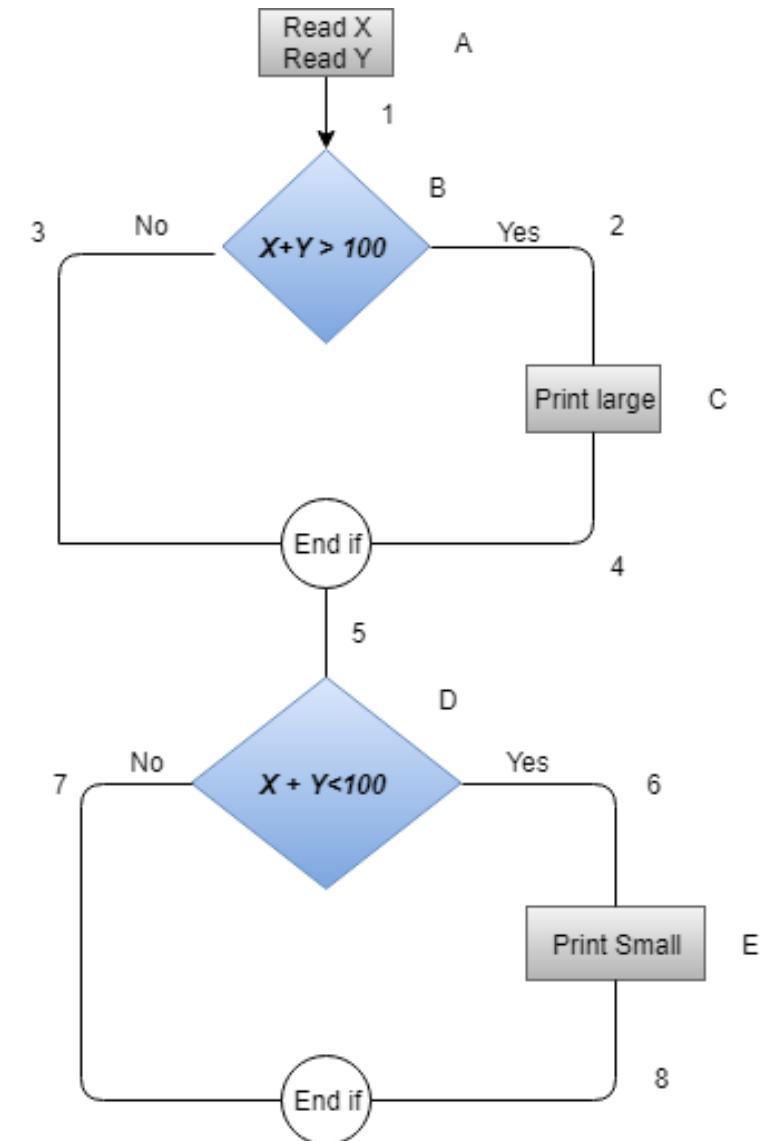
ENDIF

This is the basic code structure where we took two variables X and Y and two conditions. If the first condition is true, then print "Large" and if it is false, then go to the next condition. If the second condition is true, then print "Small."

How to calculate Branch coverage?

Control flow graph of code structure

In the adjacent diagram, control flow graph of the given code is depicted. In the first case traversing through "Yes" decision, the path is **A1-B2-C4-D6-E8**, and the number of covered edges is 1, 2, 4, 5, 6 and 8 but edges 3 and 7 are not covered in this path. To cover these edges, we have to traverse through "No" decision. In the case of "No" decision the path is A1-B3-5-D7, and the number of covered edges is 3 and 7. So by traveling through these two paths, all branches have covered.



How to calculate Branch coverage?

Path 1 - A1-B2-C4-D6-E8

Path 2 - A1-B3-5-D7

Branch Coverage (BC) = Number of paths =2

Case	Covered Branches	Path	Branch coverage
Yes	1, 2, 4, 5, 6, 8	A1-B2-C4-D6-E8	2
No	3,7	A1-B3-5-D7	

White-Box Testing

Condition Coverage

- **Condition coverage testing** is a type of white-box testing that tests all the conditional expressions in a program for all possible outcomes of the conditions. It is also called *predicate coverage*.
- Condition coverage testing tests the conditions independently of each other.

□ Condition coverage vs. branch coverage

In *branch coverage*, all conditions must be executed at least once. On the other hand, in *condition coverage*, all possible outcomes of all conditions must be tested at least once.

Example: Consider the code snippet below:

1. int a = 10;
2. if (a > 0)
3. {
4. cout<<"a is positive";
5. }

Condition Coverage

Branch coverage requires that the condition $a>0$ is executed at least once.

Condition coverage requires that both the outcomes $a>0=True$ and $a>0=False$ for the condition $a>0$ are executed at least once.

Example 1:

Consider the code snippet below, which will be used to conduct *condition coverage testing*:

```
1. int num1 = 0;  
2. if(num1>0)  
3. {  
4.     cout<<"valid input";  
5. }  
6. Else  
7. {  
8.     cout<<"invalid input";  
9. }
```

Condition Coverage

Condition coverage testing

The condition coverage testing of the code above will be as follows:

Test case number	num1>0	Final output
1	True	True
2	False	False

Example 2

Consider the code snippet below, which will be used to conduct *condition coverage testing*:

Multiple Condition Coverage

- Multiple condition coverage (MCC) is achieved, if the test cases make the component conditions of a composite conditional expression to assume all possible combinations of true and false values.
- For example, consider the composite conditional expression $[(c_1 \text{ and } c_2) \text{ or } c_3]$. A test suite would achieve MCC, if all the component conditions c_1 , c_2 , and c_3 are each made to assume all combinations of true and false values. Therefore, at least eight test cases would be required in this case to achieve MCC.
- For a composite conditional expression of n components, 2^n test cases are required for multiple condition coverage.
- 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 atomic conditions in the decision expression) is small.

Multiple Condition Coverage

Example: Give an example of a fault that is detected by multiple condition coverage, but not by branch coverage.

Solution: 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.

Determining the set of test cases to achieve MC/DC

- The first step is to draw the truth table involving the basic conditions of the given decision expression.
- Next by inspection of the truth table, the MC/DC test suite can be determined such that each condition independently affects the outcome of the decision.
- Typically, we have extra columns in the truth table, which we fill during analysis of the truth table to keep track of which test cases make a condition to independently affect the outcome of the decision.
- We now illustrate this procedure through the following problems. Though it is harder to prove, for various examples, we can observe that at least $n + 1$ test cases are necessary to achieve MC/DC for decision expressions with n conditions.

Problem-1: Design MC/DC test suite for the following decision statement:

if (A and B) then

Determining the set of test cases to achieve MC/DC

Solution:

We first draw the truth table:

<i>Test case number</i>	<i>A</i>	<i>B</i>	<i>Decision</i>	<i>Test case pair for A</i>	<i>Test case pair for B</i>
1	T	T	T	3	2
2	T	F	F		1
3	F	T	F	1	
4	F	F	F		

From the truth table given above, we can observe that the test cases 1, 2 and 3 together achieve MC/DC for the given expression.

Determining the set of test cases to achieve MC/DC

Problem-2: Design a test suite that would achieve MC/DC for the following decision statement:

~~if((A && B) || C)~~ if((A && (B|| C))

Solution: We first draw the truth table:

From the table, we can observe that different sets of test cases achieve MC/DC. The sets are {2,3,4,6}, {2,3,4,7} and {1,2,3,4,5}.

Test case	ABC	Result	A	B	C
1	TTT	T	5		
2	TTF	T	6	4	
3	TFT	T	7		4
4	TFF	F		2	3
5	FTT	F	1		
6	FTF	F	2		
7	FFT	F	3		
8	FFF	F			

Multiple Condition Coverage

Example :

Consider the code snippet below, which will be used to conduct *condition coverage testing*:

```
1. int num1 = 0;  
2. int num2 = 0;  
3. if((num1>0 || num2<10))  
4. {  
5.     cout<<"valid input";  
6. }  
7. Else  
8. {  
9.     cout<<"invalid input";  
10. }
```

Condition Coverage

Condition coverage testing

The condition coverage testing of the code above will be as follows:

Test case number	num1>0	num2<10	Final output
1	True	Not required	True
2	False	True	True
3	False	False	False

Condition Coverage

Example

Consider the code snippet below, which will be used to conduct *condition coverage testing*:

```
1. int num1 = 0;  
2. int num2 = 0;  
3. if((num1>0) && (num1+num2<15))  
4. {  
5.     cout<<"valid input";  
6. }  
7. Else  
8. {  
9.     cout<<"invalid input";  
10. }
```

Condition Coverage

Condition coverage testing

The condition coverage testing of the code above will be as follows:

Test case number	num1>0	num1+num2<1	Final output
1	True	True	True
2	True	False	False
3	False	Not required	False

Condition Coverage

Example

Consider the code snippet below, which will be used to conduct *condition coverage testing*:

```
1. int num1 = 0;  
2. int num2 = 0;  
3. if((num1>0 || num2<10) && (num1+num2<15))  
4. {  
5.     cout<<"valid input";  
6. }  
7. Else  
8. {  
9.     cout<<"invalid input";  
10. }
```

Condition Coverage

Condition coverage testing

The condition coverage testing of the code above will be as follows:

Test case number	num1>0	num2<10	num1+num2<15	Final output
1	True	don't care	True	True
2	True	don't care	False	False
3	False	True	True	True
4	False	True	False	False
5	False	False	Not required	False

White-Box Testing

Path Coverage

A test suite achieves path coverage if it executes each linearly independent paths (or 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 CFG is a directed graph consisting of a set of nodes and edges (N, E), such that each node $n \in N$ corresponds to a unique program statement and an edge exists between two nodes if control can transfer from one node to the other. We can easily draw the CFG for any program, if we know how to represent the sequence, selection, and iteration types of statements in the CFG.

- A control flow graph describes how the control flows through the program.
- In order to draw the control flow graph of a program, we need to first number all the statements of a program.
- The different numbered statements serve as nodes of the control flow graph.
- There exists an edge from one node to another, if the execution of the statement representing the first node can result in the transfer of control to the other node.

Control flow graph (CFG)

- The CFG for any program can be easily drawn, if it is known how to represent the sequence, selection, and iteration types of statements in the CFG. After all, every program is constructed by using these three types of constructs only.
- The CFG representation of the sequence and decision types of statements is straight forward.
- For iteration type of constructs such as the while construct, the loop condition is tested only at the beginning of the loop and therefore always control flows from the last statement of the loop to the top of the loop. That is, the loop construct terminates from the first statement (after the loop is found to be false) and does not at any time exit the loop at the last statement of the loop.

Example: Using the basic ideas, the CFG of the program given in Figure(a) can be drawn as shown in Figure (b).

Control flow graph (CFG)

```
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)

Sequence:

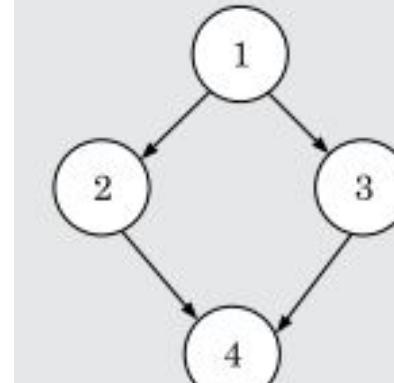
1. a=5;
2. b=a*2-1



(b)

Selection:

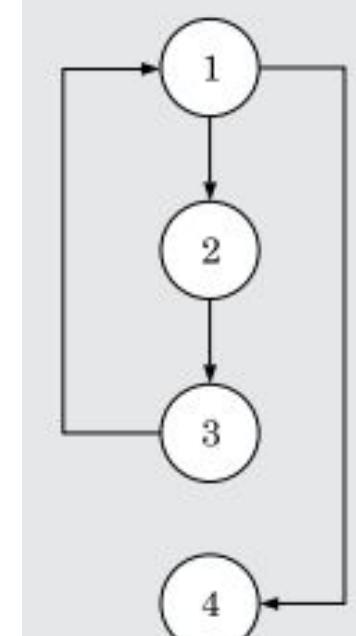
1. if(a>b)
2. c=3;
3. else c=5;
4. c=c*c;



(c)

Iteration:

1. while(a>b){
2. b=b-1;
3. b=b*a;}
4. c=a+b;



(d)

Figure(a): Example Program.

Figure(b): CFG for Sequence of statements.

Figure(c): CFG for Selection Construct.

Figure(d): CFG for Iterative Construct.

Control flow graph (CFG)

```
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;  
}
```

Figure (a): Example program.

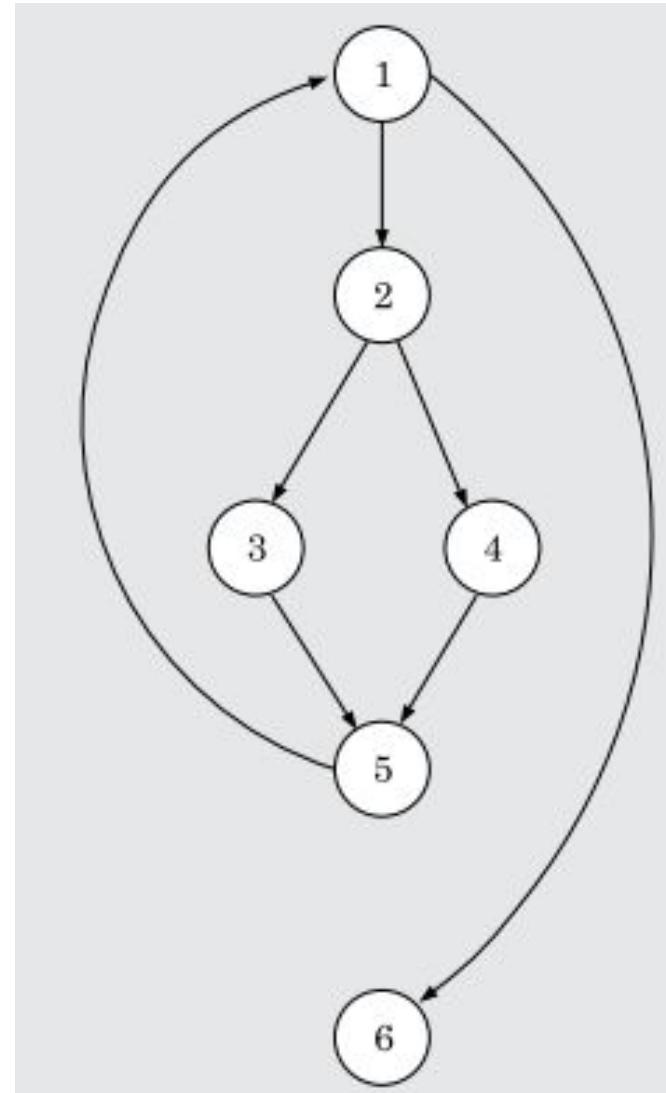


Figure (b): Control-flow graph.

Path

- A path through a program is any node and edge sequence from the start node to a terminal node of the control flow graph of a program.
- A program can have more than one terminal nodes when it contains multiple exit or return type of statements.
- Writing test cases to cover all paths of a typical program is impractical since there can be an infinite number of paths through a program in presence of loops.
- If coverage of all paths is attempted, then the number of test cases required would become infinitely large.
- Testing of all paths is impractical.
- Path coverage testing does not try to cover all paths, but only a subset of paths called linearly independent paths (or basis paths) is tested.

Linearly independent set of paths (or basis path set)

A set of paths for a given program is called linearly independent set of paths (or the set of basis paths or simply the basis set), if each path in the set introduces at least one new edge that is not included in any other path in the set. If a path has one new node compared to all other linearly independent paths, then the path is also linearly independent.

In the above example, we can see there are few conditional statements that is executed depending on what condition it suffice. Here there are 3 paths or condition that need to be tested to get the output.

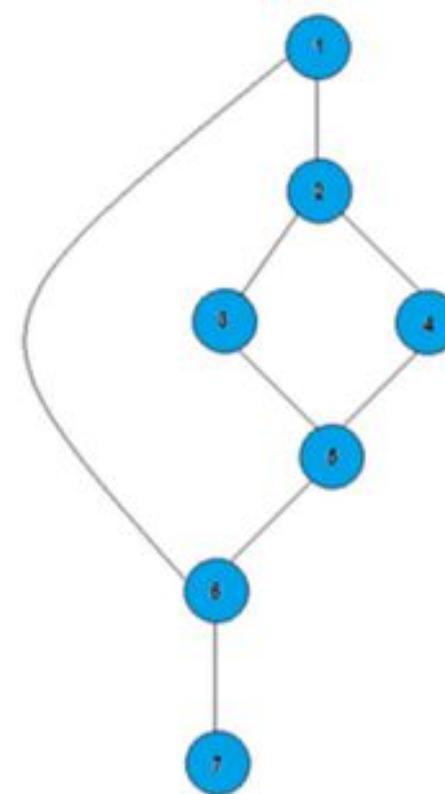
The paths are-

Path 1: 1,2,3,5,6, 7

Path 2: 1,2,4,5,6, 7

Path 3: 1, 6, 7

1. If A= 50
2. THEN IF B>C
3. THEN A=B
4. ELSE A=C
5. ENDIF
6. ENDIF
7. Print A



McCabe's Cyclomatic Complexity Metric

- It is easy and straight forward to identify the linearly independent paths for a simple program.
- For complex programs it is not easy to determine the number of independent paths.
- McCabe's cyclomatic complexity metric is an important result that helps us to compute the number of linearly independent paths for any arbitrary program.
- McCabe's cyclomatic complexity defines an upper bound for the number of linearly independent paths through a program.
- McCabe's metric does not directly identify the linearly independent paths, but it provides us with a practical way of determining approximately how many paths to look for.
- McCabe's metric is only an upper bound and does not give the exact number of paths.

McCabe's Cyclomatic Complexity Metric

McCabe obtained his results by applying graph-theoretic techniques to the control flow graph of a program. McCabe's cyclomatic complexity metric defines an upper bound on the number of independent paths in a program.

We have three different ways to compute the cyclomatic complexity. For structured programs, the results computed by all the three methods are guaranteed to agree.

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.

For the CFG of the example program shown in previous figure, $E = 7$ and $N = 6$. Therefore, the value of the Cyclomatic complexity $= 7 - 6 + 2 = 3$.

McCabe's Cyclomatic Complexity Metric

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:

$$V(G) = \text{Total number of non-overlapping bounded areas} + 1$$

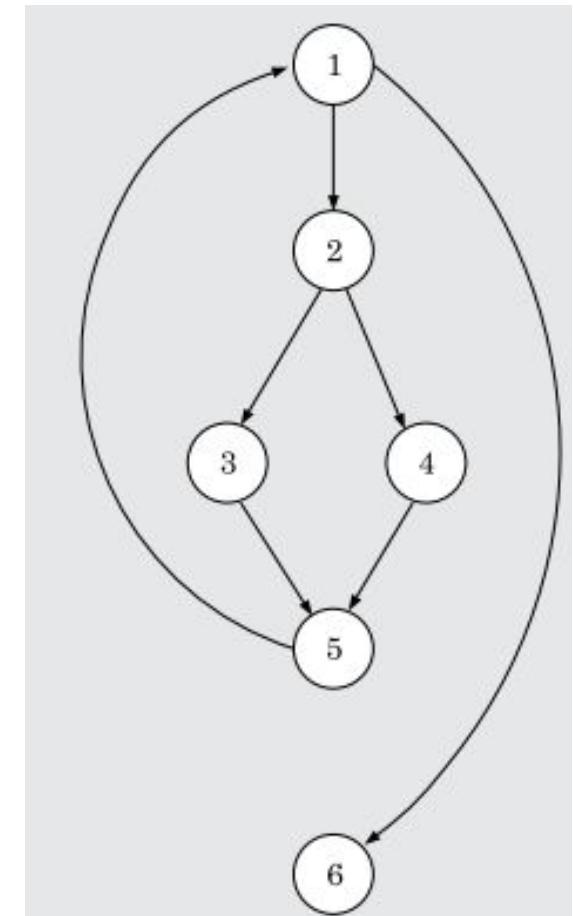
- In the program's control flow graph G , any region enclosed by nodes and edges can be called as a bounded area.
- From a visual examination of the CFG given earlier, the number of bounded areas is 2. Therefore, the cyclomatic complexity, computed with this method is also $2+1=3$.
- The number of bounded areas in a CFG increases with the number of decision statements and loops.

McCabe's Cyclomatic Complexity Metric

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 $N + 1$.

```
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;  
    }
```

In the given example, the total number of loop and decision statements is 2. So, the McCabe's cyclomatic complexity is $2+1=3$.



Steps to carry out path coverage-based testing

The following is the sequence of steps that need to be undertaken for deriving the path coverage-based test cases for a program:

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.

Debugging Integration and System Testing

Debugging

Once errors are identified in a program code, it is necessary to first identify the precise program statements responsible for the errors and then to fix them. Identifying errors in a program code and then fix them up are known as debugging.

Debugging Approaches

Brute Force Method:

- This is the most common method, but the least efficient.
- The program is loaded with print statements to print the intermediate values with the hope that some of the printed values will help to identify the statement in error.
- 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.

Debugging

Backtracking:

- A fairly common approach.
- Beginning from the statement at which an error symptom has been observed, the source code is traced backwards until the error is discovered.
- As the number of source lines to be traced back increases, the number of potential backward paths increases and may become unmanageably large thus limiting the use of this approach.

Cause Elimination Method:

A list of causes which could possibly have contributed to the error 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. Here the search space is reduced by defining slices. A slice of a program for a particular variable at a particular statement is the set of source lines preceding this statement that can influence the value of that variable.

Debugging

Debugging Guidelines:

Debugging is often carried out by programmers based on their ingenuity. The following are some general guidelines for effective debugging:

- Many times debugging requires a thorough understanding of the program design. Trying to debug based on a partial understanding of the system design and implementation may require an inordinate amount of effort to be put into debugging even simple problems.
- Debugging may sometimes even require full redesign of the system. In such cases, a common mistake that novice programmers often make is attempting not to fix the error but its symptoms.
- One must be beware of the possibility that an error correction may introduce new errors.
- Therefore after every round of error-fixing, regression testing must be carried out.

Integration Testing

- Integration testing is carried out after all (or at least some of) the modules have been unit tested.
- Successful completion of unit testing, to a large extent, ensures that the unit (or module) as a whole works satisfactorily.
- The objective of integration testing is to detect the errors at the module interfaces (call parameters).
- For example, it is checked that no parameter mismatch occurs when one module invokes the functionality of another module.
- Thus, the primary objective of integration testing is to test the module interfaces, that is, there are no errors in parameter passing, when one module invokes the functionality of another module

Integration Testing

- During integration testing, different modules of a system are integrated in a planned manner using an integration plan.
- The integration plan specifies the steps and the order in which modules are combined to realize the full system. After each integration step, the partially integrated system is tested.
- An important factor that guides the integration plan is the module dependency graph or structure chart which specifies the order in which different modules call each other.
- By examining the structure chart, the integration plan can be developed.

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

- Big-bang testing is the most obvious approach to integration testing. In this approach, all the modules making up a system are integrated in a single step.
- All the unit tested modules of the system are simply linked together and tested.
- 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 localize the error as the error may potentially exist in any of the modules.
- Therefore, debugging errors reported during big-bang integration testing are very expensive to fix.
- 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. In bottom-up integration testing, first the modules for each subsystem are integrated. Thus, the subsystems can be integrated separately and independently.
- Large software systems normally require several levels of subsystem testing, lower-level subsystems are successively combined to form higher-level subsystems.
- The principal advantage of bottom-up integration testing is that several disjoint subsystems can be tested simultaneously. Another advantage of bottom-up testing is that the low-level modules get tested thoroughly, since they are exercised in each integration step.
- Since the low-level modules do I/O and other critical functions, testing the low-level modules thoroughly increases the reliability of the system.
- A disadvantage of bottom-up testing is the complexity that occurs when the system is made up of a large number of small subsystems that are at the same level. This extreme case corresponds to the big-bang approach.

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.

Top-down integration testing approach requires the use of program stubs to simulate the effect of lower-level routines that are called by the routines under test. A pure top-down integration does not require any driver routines.

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.
- Therefore, this is one of the most commonly used integration testing approaches.
- In this approach, both stubs and drivers are required to be designed.

System Testing

After all the units of a program have been integrated together and tested, system testing is taken up.

The test cases for system testing are designed solely based on the SRS document and the actual implementation (procedural or object-oriented) is immaterial.

There are three main kinds of system testing.

Alpha Testing: Alpha testing refers to the system testing carried out by the test team within the developing organization.

Beta Testing: Beta testing is the system testing performed by a select group of friendly customers.

Acceptance Testing: Acceptance testing is the system testing performed by the customer to determine whether to accept the delivery of the system

Smoke Testing

- Smoke testing is performed to check whether at least the main functionalities of the software are working properly. Unless the software is stable and at least the main functionalities are working satisfactorily, system testing is not undertaken.
- 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.

Performance Testing

Performance testing is an **important type of system testing**. For a specific system, the types of performance testing to be carried out on a system depends on the different non-functional requirements of the system documented in its SRS document. All **performance tests can be considered as black-box tests.**

There are several types of performance testing corresponding to various types of non-functional requirements-

- **Stress testing**
- **Volume testing**
- **Configuration testing**
- **Compatibility testing**
- **Regression testing**
- **Recovery testing**
- **Maintenance testing**
- **Documentation testing**
- **Usability testing**
- **Security testing**

Performance Testing

Stress testing

- ❖ Stress testing is also known as *endurance testing*. Stress testing evaluates system performance when it is stressed for short periods of time. Stress tests are black-box tests which are designed to impose a range of abnormal and even illegal input conditions so as to stress the capabilities of the software. Input data volume, input data rate, processing time, utilization of memory, etc., are tested beyond the designed capacity.
- ❖ Stress testing is especially important for systems that under normal circumstances operate below their maximum capacity but may be severely stressed at some peak demand hours.

Performance Testing

Volume testing

- ❖ Volume testing checks whether the data structures (buffers, arrays, queues, stacks, etc.) have been designed to successfully handle extraordinary situations. For example, the volume testing for a compiler might be to check whether the symbol table overflows when a very large program is compiled.

Configuration testing

- ❖ Configuration testing is used to test system behavior in various hardware and software configurations specified in the requirements. Sometimes systems are built to work in different configurations for different users. For instance, a minimal system might be required to serve a single user, and other extended configurations may be required to serve additional users. During configuration testing, the system is configured in each of the required configurations and it is checked if the system behaves correctly in all required configurations.

Performance Testing

Compatibility testing

- ❖ This type of testing aims to check whether the interfaces with the external systems are performing as required. For instance, if the system needs to communicate with a large database system to retrieve information, compatibility testing is required to test the speed and accuracy of data retrieval.

Regression testing

- ❖ This type of testing is required when a software is maintained to fix some bugs or enhance functionality, performance, etc.

Recovery testing

- ❖ Recovery testing tests the response of the system to the presence of faults, or loss of power, devices, services, data, etc. The system is subjected to the loss of the mentioned resources (as discussed in the SRS document) and it is checked if the system recovers satisfactorily.

Performance Testing

Maintenance testing

- ❖ This addresses testing the diagnostic programs, and other procedures that are required to help maintenance of the system. It is verified that the artifacts exist and they perform properly.

Documentation testing

- ❖ It is checked whether the required user manual, maintenance manuals, and technical manuals exist and are consistent. If the requirements specify the types of audience for which a specific manual should be designed, then the manual is checked for compliance of this requirement.

Performance Testing

Usability testing

- ❖ Usability testing concerns checking the user interface to see if it meets all user requirements concerning the user interface. During usability testing, the display screens, messages, report formats, and other aspects relating to the user interface requirements are tested.

Security testing

- ❖ Security testing is essential for software that handle or process confidential data that is to be guarded against pilfering. It needs to be tested whether the system is fool-proof from security attacks such as intrusion by hackers. A large number of security testing techniques have been proposed, and these include password cracking, penetration testing, and attacks on specific ports, etc.

Project Scheduling

Scheduling

Scheduling the project tasks is an important project planning activity. Once a schedule has been worked out and the project gets underway, the project manager monitors the timely completion of the tasks and takes any corrective action that may be necessary whenever there is a chance of schedule slippage. In order to schedule the project activities, a **software project manager** needs to do the following:

1. Identify all the major activities that need to be carried out to complete the project.
2. Break down each activity into tasks.
3. Determine the dependency among different tasks.
4. Establish the estimates for the time durations necessary to complete the tasks.
5. Represent the information in the form of an activity network.
6. Determine task starting and ending dates from the information represented in the activity network.
7. Determine the *critical path*. A critical path is a chain of tasks that determines the duration of the project.
8. Allocate resources to tasks.

Work Breakdown Structure

- *Work breakdown structure (WBS)* is used to recursively decompose a given set of activities into smaller activities.
- WBS provides a notation for representing the activities, sub-activities, and tasks needed to be carried out in order to solve a problem. Each of these is represented using a rectangle.
- The root of the tree is labelled by the project name. Each node of the tree is broken down into smaller activities that are made the children of the node.
- To decompose an activity to a sub-activity, a good knowledge of the activity can be useful.
- The figure given in the next slide represents the WBS of a management information system (MIS) software.

Work Breakdown Structure

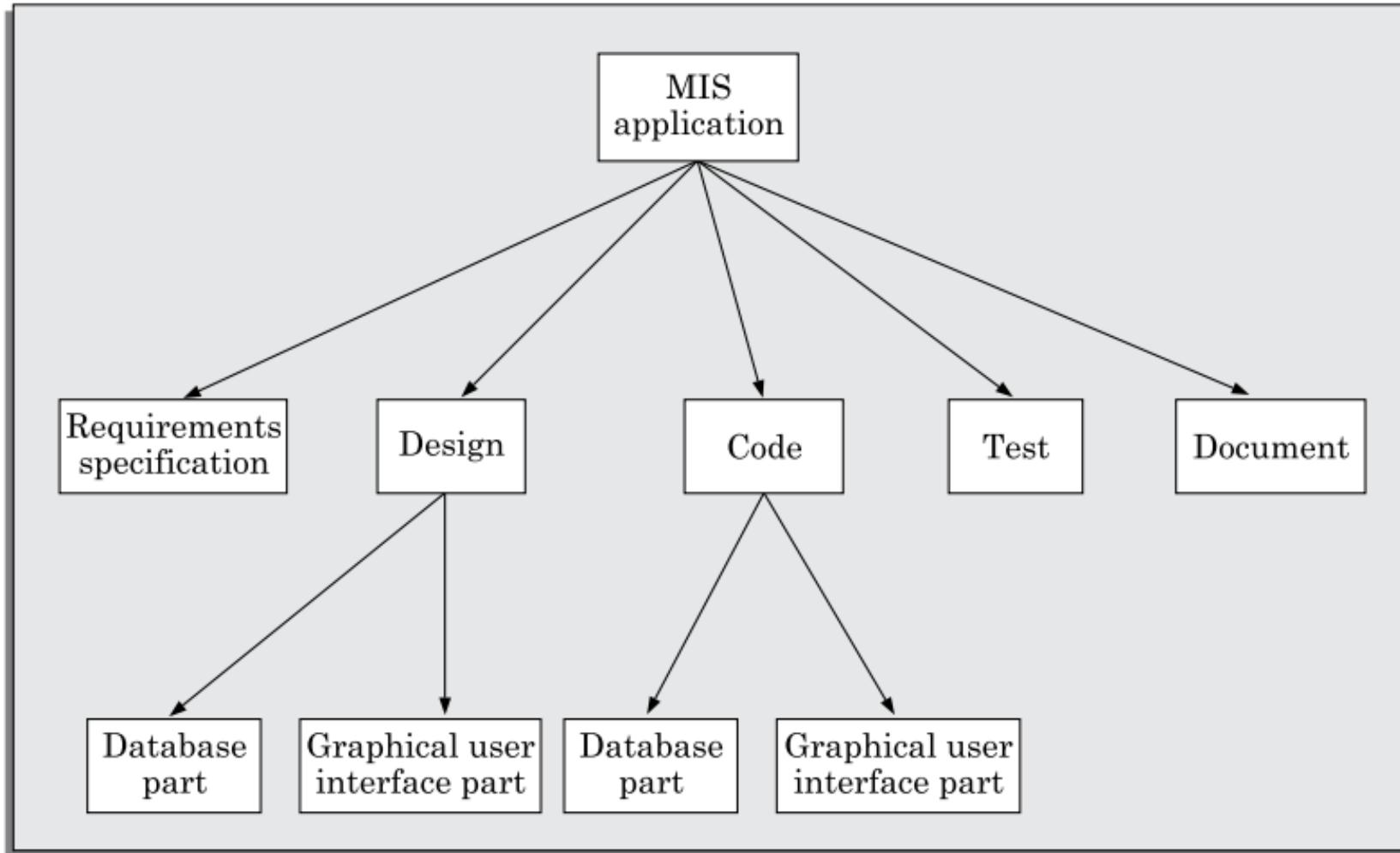


FIGURE: Work breakdown structure of an MIS problem.

Work Breakdown Structure

How long to decompose?

- The decomposition of the activities is carried out until any of the following is satisfied:
- A leaf-level sub-activity (a task) requires approximately two weeks to develop.
- Hidden complexities are exposed, so that the job to be done is understood and can
- be assigned as a unit of work to one of the developers.
- Opportunities for reuse of existing software components is identified.

Activity Network

An activity network shows the different activities making up a project, their estimated durations, and their interdependencies. Two equivalent representations for activity networks are possible and are in use:

Activity on Node (AoN): In this representation, each activity is represented by a rectangular (some use circular) node and the duration of the activity is shown alongside each task in the node. The inter-task dependencies are shown using directional edges.

Activity on Edge (AoE): In this representation tasks are associated with the edges. The edges are also annotated with the task duration. The nodes in the graph represent project milestones.

Activity networks were originally represented using *activity on edge* (AoE) representation.

However, later *activity on node* (AoN) has become popular since this representation is easier to understand and revise.

Activity Network

PROBLEM: Determine the Activity network representation for the MIS development project for which the relevant data is given in Table given below. Assume that the manager has determined the tasks to be represented from the work breakdown structure of figure given earlier, and has determined the durations and dependencies for each task as shown in the table.

<i>Task Number</i>	<i>Task</i>	<i>Duration</i>	<i>Dependent on Tasks</i>
T1	Specification	15	-
T2	Design database	45	T1
T3	Design GUI	30	T1
T4	Code database	105	T2
T5	Code GUI part	45	T3
T6	Integrate and test	120	T4 and T5
T7	Write user manual	60	T1

TABLE: Project Parameters Computed from Activity Network

Activity Network

SOLUTION:

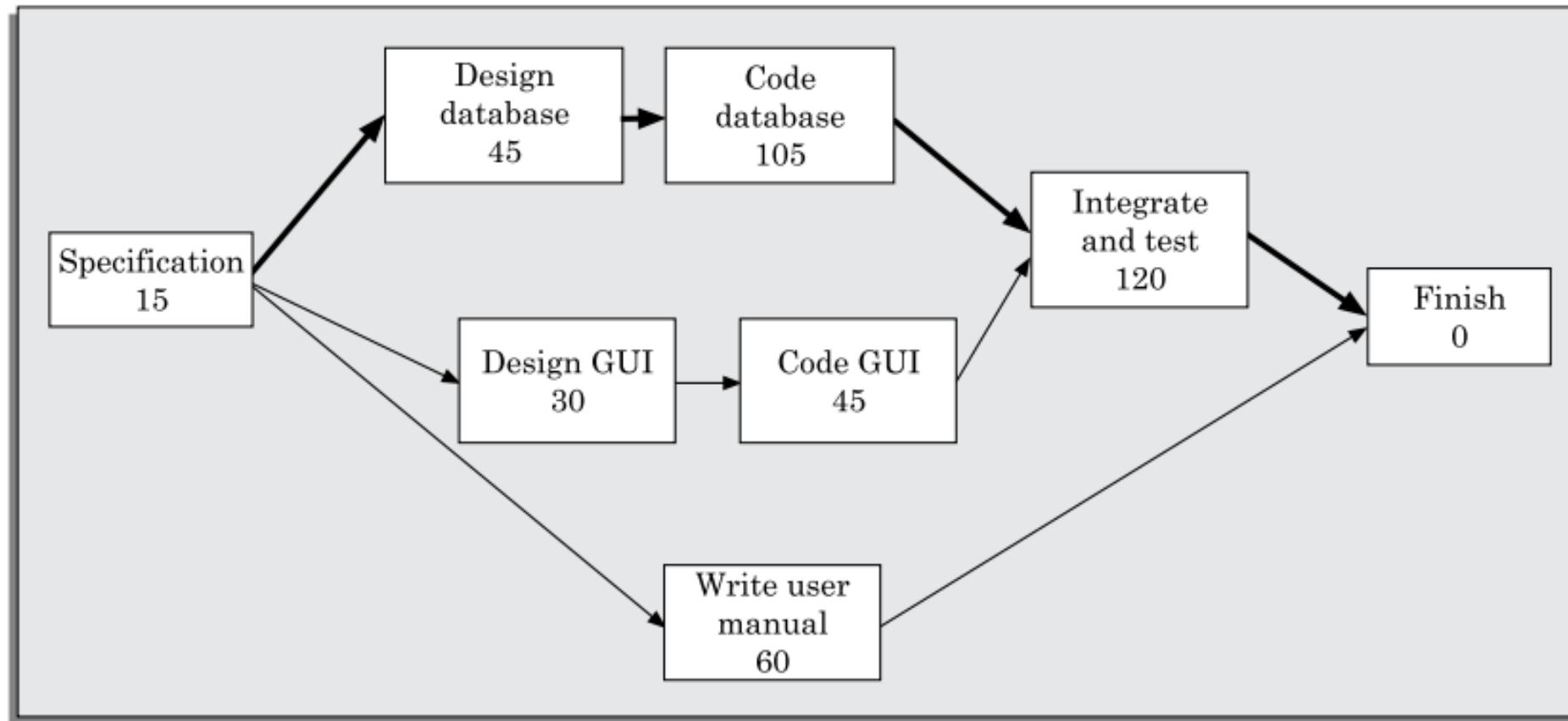


FIGURE: Activity network representation of the MIS problem.

Critical Path Method (CPM)

A path in the activity network graph is any set of consecutive nodes and edges in this graph from the starting node to the last node. A critical path consists of a set of dependent tasks that need to be performed in a sequence and which together take the longest time to complete.

CPM is an algorithmic approach to determine the critical paths and slack times for tasks not on the critical paths involves calculating the following quantities:

Minimum time (MT): It is the minimum time required to complete the project. It is computed by determining the maximum of all paths from start to finish.

Earliest start (ES): It is the time of a task is the maximum of all paths from the start to this task. The ES for a task is the ES of the previous task plus the duration of the preceding task.

Critical Path Method (CPM)

Latest start time (LST): It is the difference between MT and the maximum of all paths from this task to the finish. The LST can be computed by subtracting the duration of the subsequent task from the LST of the subsequent task.

Earliest finish time (EF): The EF for a task is the sum of the earliest start time of the task and the duration of the task.

Latest finish (LF): LF indicates the latest time by which a task can finish without affecting the final completion time of the project. A task completing beyond its LF would cause project delay. LF of a task can be obtained by subtracting maximum of all paths from this task to finish from MT.

Slack time (ST): The slack time (or float time) is the total time that a task may be delayed before it will affect the end time of the project. The slack time indicates the "flexibility" in starting and completion of tasks. ST for a task is LS-ES and can equivalently be written as LF-EF.

Critical Path Method (CPM)

PROBLEM: Use the Activity network of figure given earlier to determine the ES and EF for every task for the MIS problem.

Solution: The activity network with computed ES and EF values has been shown below:

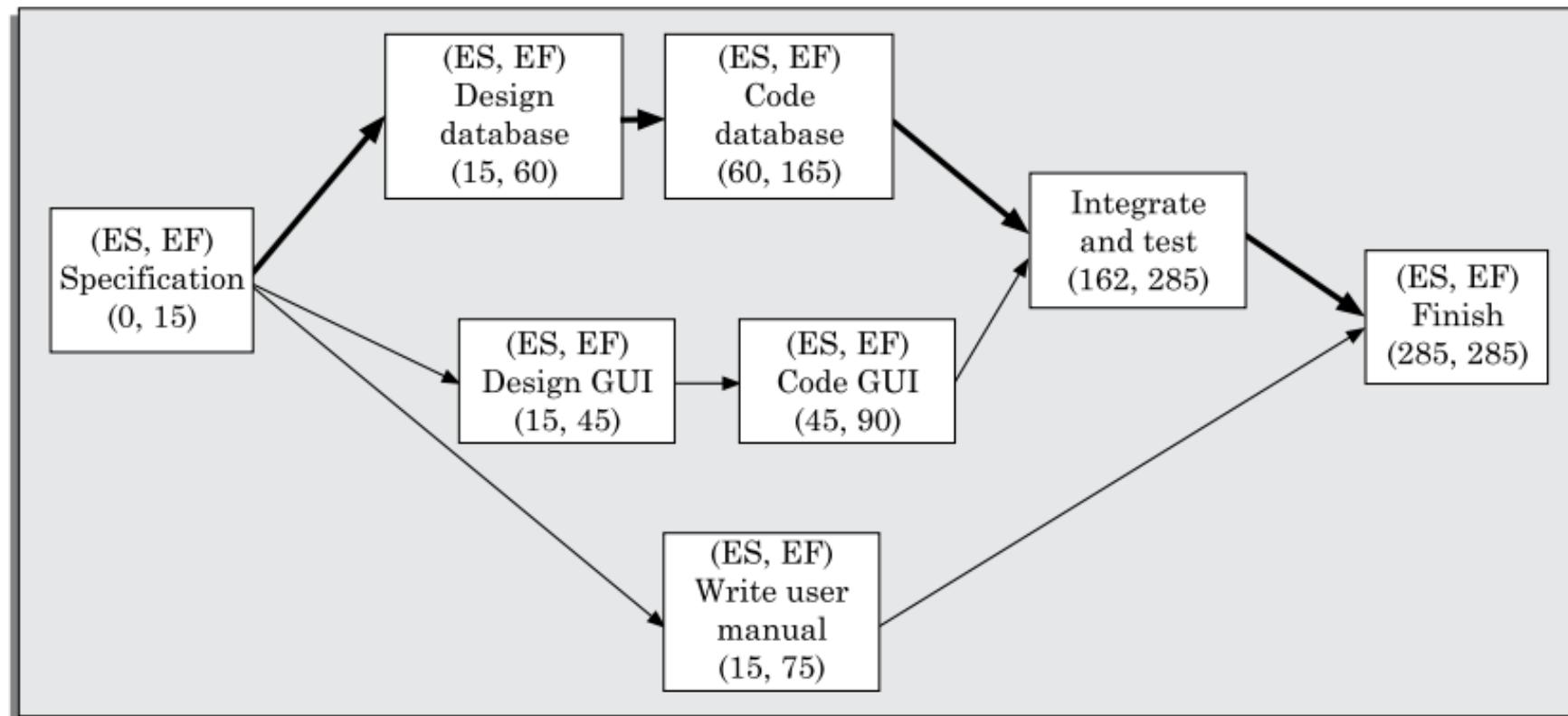


FIGURE: AoN for MIS problem with (ES, EF).

Critical Path Method (CPM)

PROBLEM 3.8 Use the Activity network given the previous problem to determine the LS and LF for every task for the MIS problem.

Solution: The activity network with computed LS and LF values has been shown in figure given below:

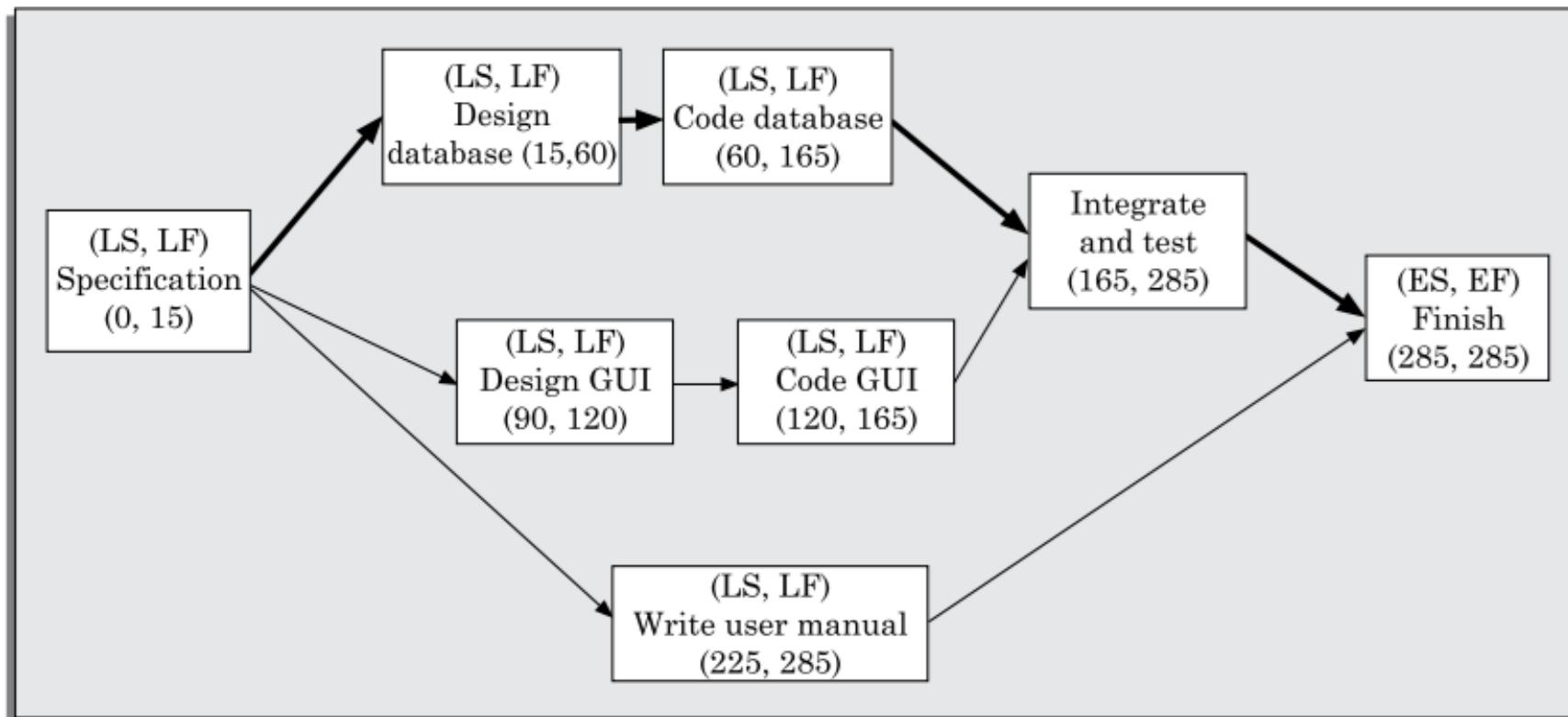


FIGURE: AoN of MIS problem with (LS, LF).

Critical Path Method (CPM)

The project parameters for different tasks for the MIS problem can be computed as follows:

□ Compute ES and EF for each task.

 Use the rule: ES is equal to the largest EF of the immediate predecessors

□ Compute LS and LF for each task.

 Use the rule: LF is equal to the smallest LS of the immediate successors

□ Compute ST for each task.

 Use the rule: $ST=LF-EF$

GANTT Charts

- A Gantt chart is a form of bar chart.
- The vertical axis lists all the tasks to be performed.
- The bars are drawn along the y-axis, one for each task.
- Gantt charts used in software project management are actually an enhanced version of the standard Gantt charts. In the Gantt charts used for software project management, each bar consists of an unshaded part and a shaded part.
- The shaded part of the bar shows the length of time each task is estimated to take. The unshaded part shows the *slack* time or lax time.
- The lax time represents the leeway or flexibility available in meeting the latest time by which a task must be finished.

GANTT Charts

A Gantt chart representation for the MIS problem of shown in figure given below:

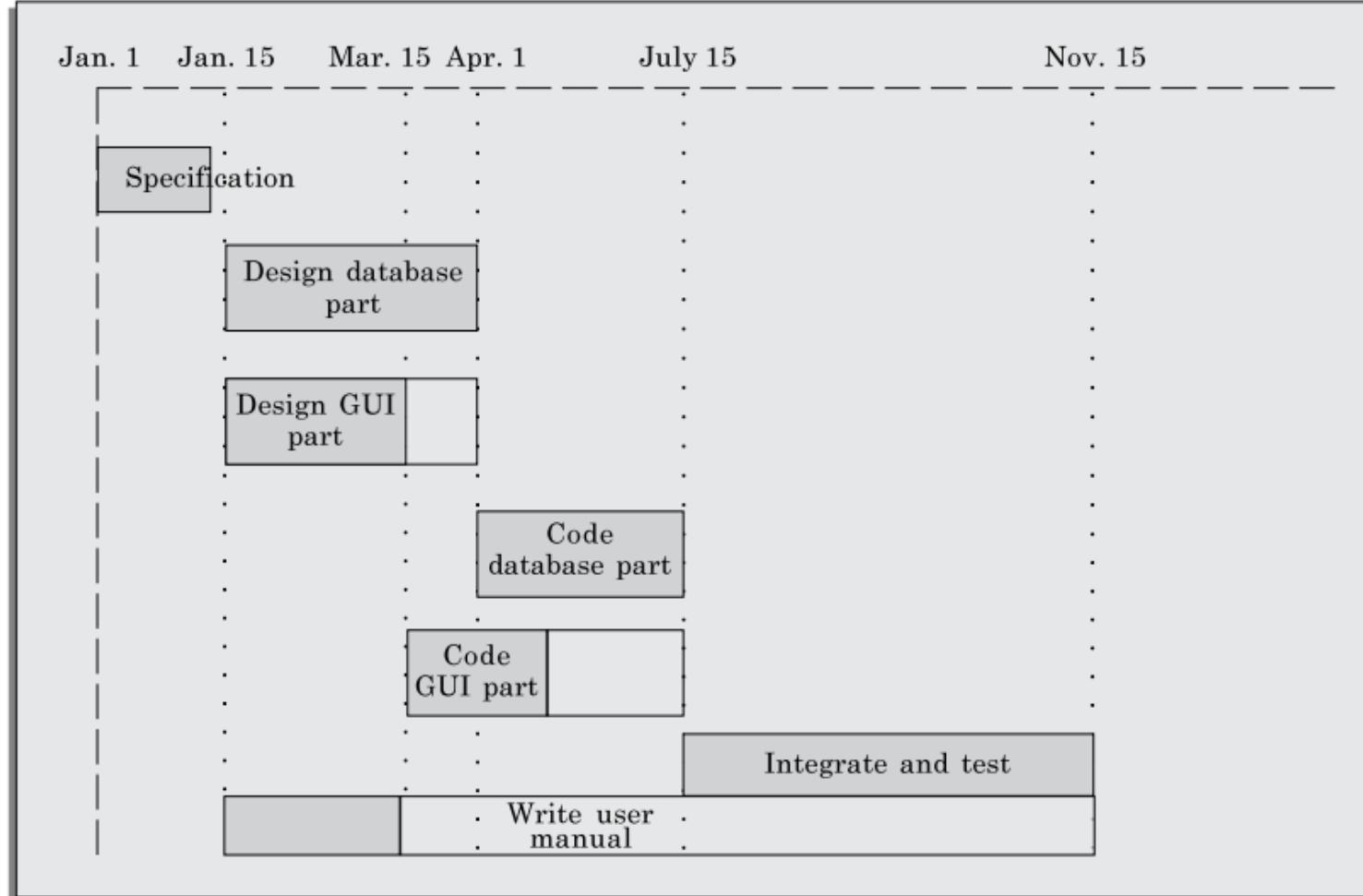


FIGURE: Gantt chart representation of the MIS problem.