<< Chapter 6 Notes >>

1. Historical Perspective

- The evolution of programming languages started from machine languages to high-level languages.
- Early programming focused on hardware-specific instructions, leading to the development of Assembly languages.
- High-level languages like Fortran, COBOL, and Lisp emerged in the 1950s-60s to simplify programming and improve productivity.
- Over time, languages adapted to solve new problems, leading to different paradigms like procedural, object-oriented, and functional programming.

The best-known early examples are

- FORTRAN (FORmula TRANslator), which was developed for scientific and engineering applications, and
- COBOL (COmmon Business-Oriented Language), which was developed by the U.S. Navy for business applications.

2. Programming Paradigms

- **Procedural Programming**: Focuses on procedures (functions) and sequential execution (e.g., C, Pascal).
- **Object-Oriented Programming (OOP)**: Based on objects that encapsulate data and behavior (e.g., Java, Python, C++).
- **Functional Programming**: Treats computation as evaluation of mathematical functions, avoiding state (e.g., Haskell, Lisp).
- **Declarative Programming**: Focuses on what to achieve rather than how (e.g., SQL, Prolog).
- Paradigms reflect different approaches to solving computational problems.

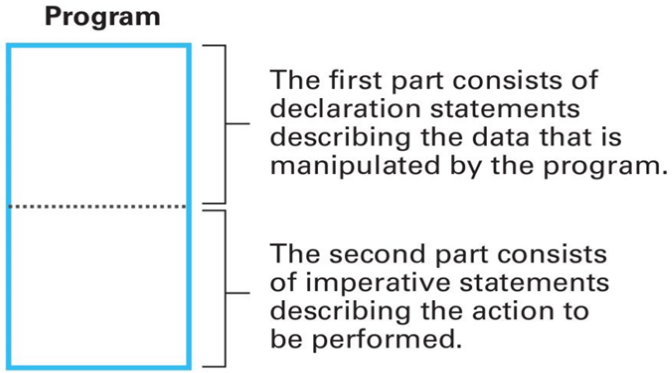
3. Traditional Programming Concepts

- **Variables**: Storage for data, identified by names and associated with a type.
- **Data Types**: Define the type of data (e.g., integer, float, string).
- **Control Structures**: Logical constructs like loops (for, while) and conditionals (if, else).
- **Functions**: Encapsulate reusable blocks of code.

Traditional Programming Concepts

A program consists of a collection of statements th tend to fall into three categories:

- declarative statements,
- imperative statements, and
- comments.



4. Procedural Units

- Central to procedural programming.
- **Procedures/Functions**: Modules that execute specific tasks and can be reused.
- Promote modularity, making programs easier to debug and maintain.
- Variables can have different scopes (local vs. global).

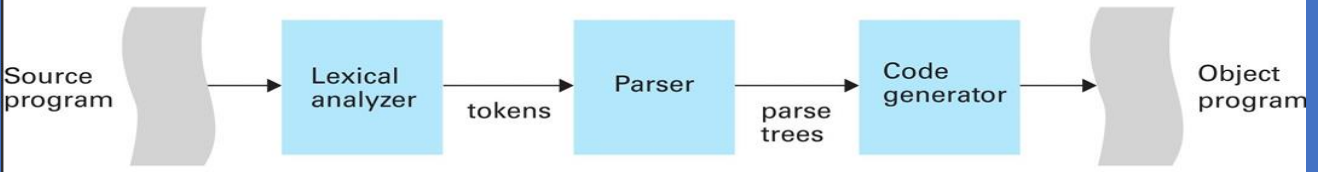
Procedural Units

Functions are often written using generic terms that are made specific when the function is applied.

- Such generic terms within functions are called **parameters (formal parameters)**
- The precise meanings assigned to these formal parameters when the function is applied are called **actual parameters.**
  - The task of transferring data between actual and formal parameters is handled in a variety of ways
    - passed by VALUE
    - passed by REFERENCE

Language Implementation

Translation: The process of converting a program from one language to another



1. **Source Program:** This is the original code written by the programmer in a specific programming language, such as C++ or Python.
2. **Lexical Analyzer:**
   - Breaks down the source code into smaller meaningful units called tokens.
   - These tokens represent the basic building blocks of the language, such as keywords, variable names, numbers, and operators.
   - For example, the statement "int x = 5;" would be broken down into tokens like "int", "x", "=", "5", and ";".
3. **Parser:**
   - Analyzes the sequence of tokens to ensure they conform to the grammar rules of the programming language.
   - Constructs a parse tree, which represents the hierarchical structure of the program.
4. **Code Generator:**
   - Uses the parse tree to generate equivalent code in the target language, usually machine code.
   - This generated code is directly understandable by the computer.
5. **Object Program:**
   - The resulting executable program that can be run on a computer.

Lexical analysis is the process of recognizing which strings of symbols from the source program represent a single entity, or token.

Lexical analyzer reads the source program symbol by symbol, identifying which groups of symbols represent tokens, and classifying those tokens according to whether they are numeric values, words, arithmetic operators, and so on.

The **parsing process** is based on a set of rules that define the syntax of the programming language.

Collectively, these rules are called a grammar.

One way of expressing these rules is by means of syntax diagrams, which are pictorial representations of a language's grammatical structure.

7. Object-Oriented Programs

- **Core Principles**:
  - **Encapsulation**: Data and behavior are bundled in objects.
  - **Inheritance**: Enables a new class to derive properties from an existing class.
  - **Polymorphism**: Objects can take multiple forms (e.g., method overloading/overriding).
  - **Abstraction**: Hides complex implementation details, showing only essential features.
- Promotes code reusability, scalability, and maintainability.
- Widely used in modern programming (e.g., Java, Python).

Requirements Analysis

The software life cycle begins with requirements analysis

- The goal: to specify what services the proposed system will provide,
  - To identify any conditions (time constraints, security, and so on) on those services,
  - To define how the outside world will interact with the system.

Requirements Analysis

The requirements analysis process:

- Compiling and analyzing the needs of the software user;
- Negotiating with the project's stakeholders ○ over trade-offs between wants, needs, costs, and feasibility;

- Developing a set of requirements
  - that identify the features and services that the finished software system must have.

Requirements are recorded in a document called a **Software Requirements Specification (SRS)**.

- A written agreement between all parties concerned,
  - Intends to guide the software's development and
  - provides a means of resolving disputes that may arise later in the development process.

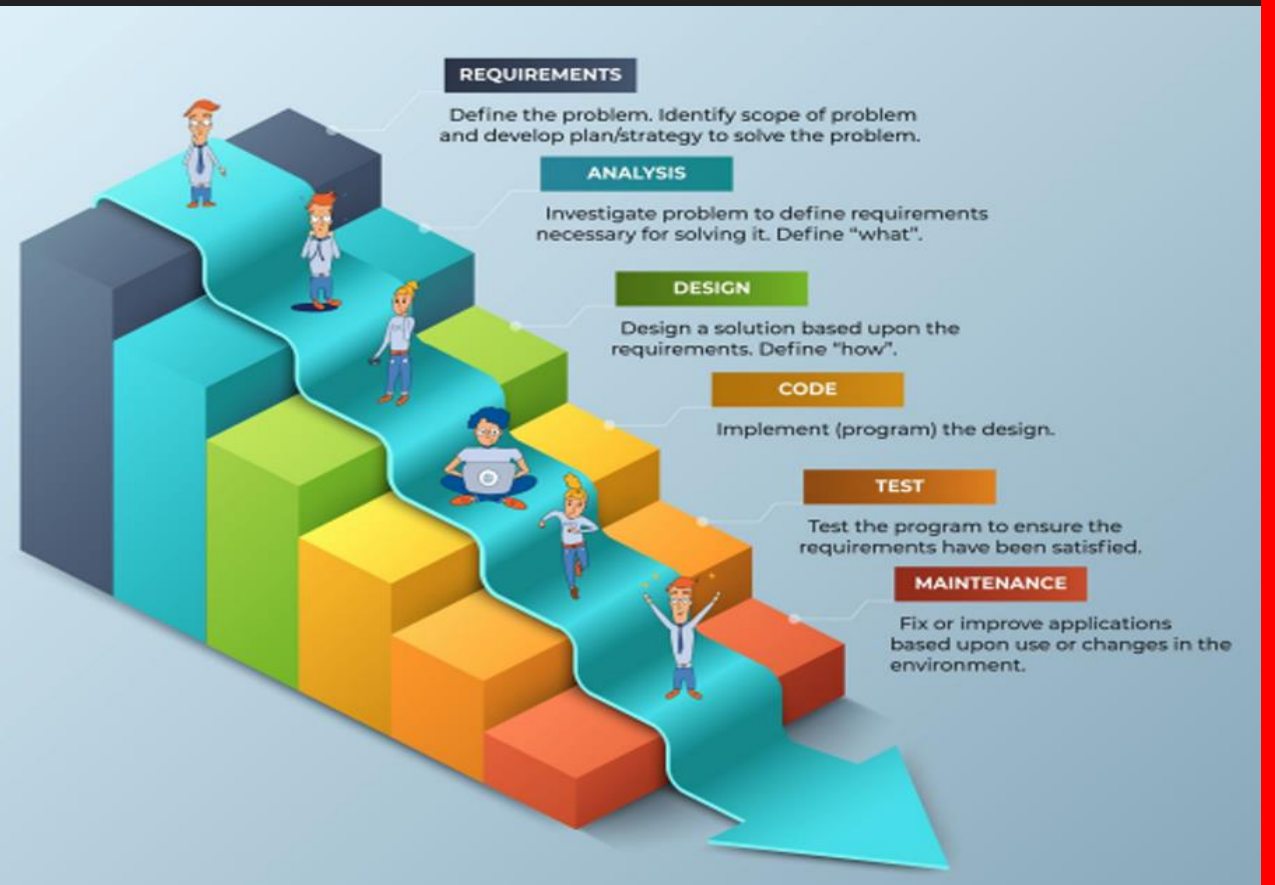
Professional organizations such as IEEE and large software clients such as the U.S. Department of Defense have adopted standards.

Straightforward and **frequent communication** with the project's stakeholders is mandatory.

4. Software Engineering Methodologies

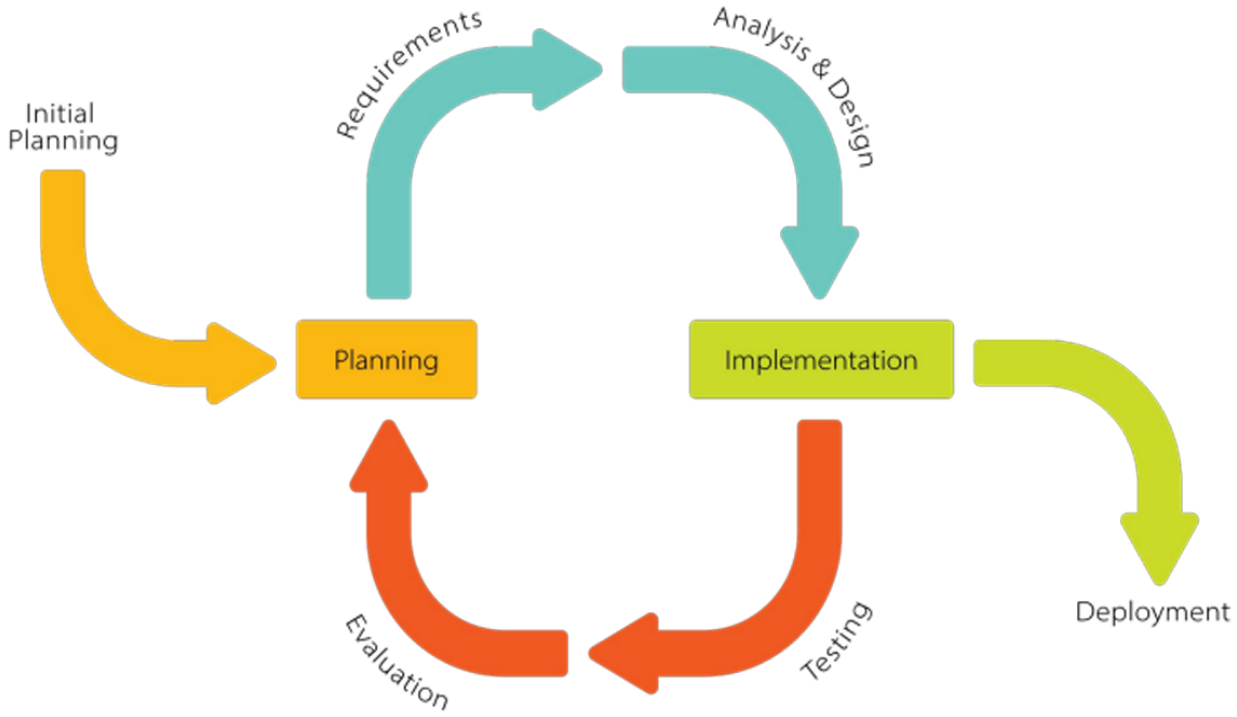
i. Waterfall Model

- **Sequential process**: Moves through phases in a linear order (e.g., requirements → design → implementation).
- **Advantages**:
  - Simple and easy to manage.
  - Clearly defined stages and deliverables.
- **Disadvantages**:
  - Limited flexibility; changes are costly and difficult.
  - Assumes all requirements can be defined upfront.



ii. Iterative & Incremental Methodologies

- Develop software in **small, manageable pieces** (increments) with regular feedback.
- Repeatedly improve and refine (iteration).
- **Advantages**:
  - Flexibility to adapt to changing requirements.
  - Early delivery of working components.
- Examples: Spiral Model, Rational Unified Process (RUP).

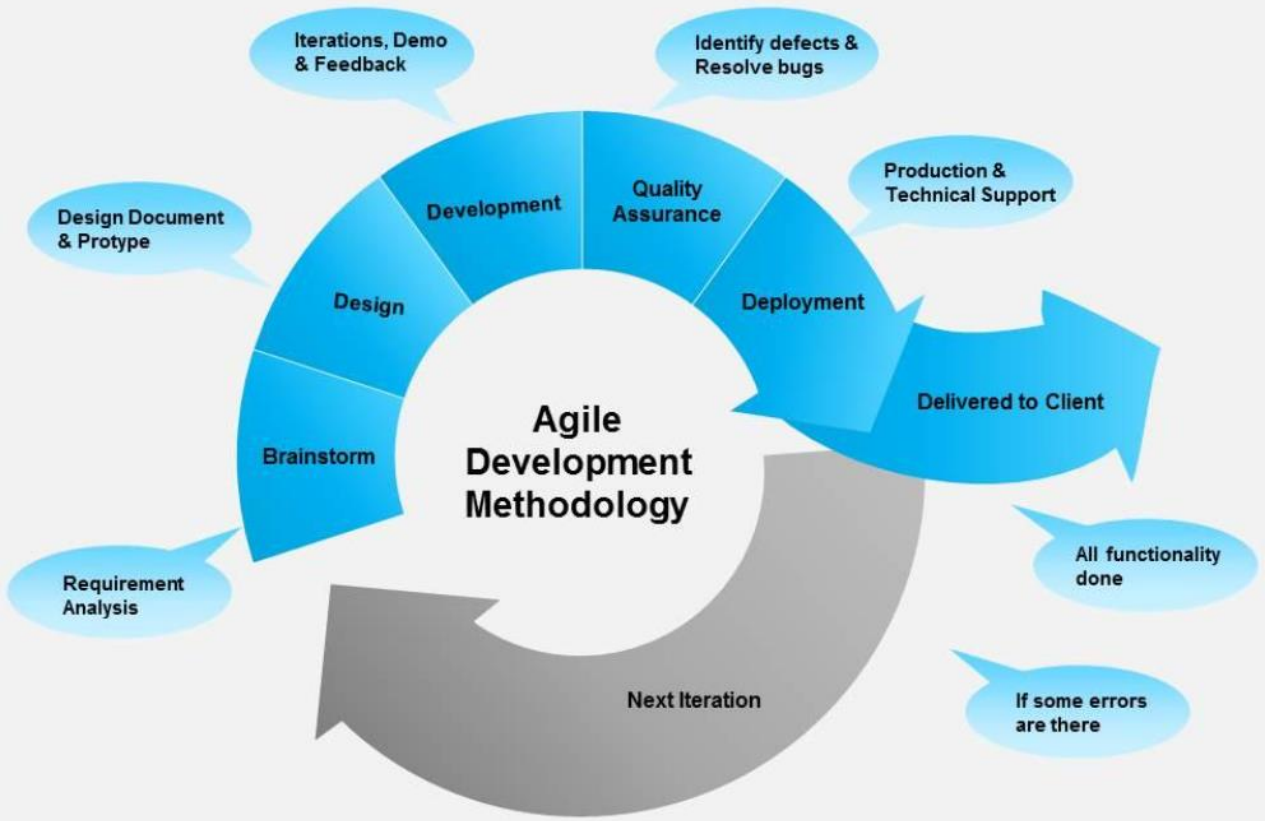


Whereas the incremental model carries the notion of extending each preliminary version of a product into a larger version, the iterative model encompasses the concept of refining each version. In reality, the incremental model involves an underlying iterative process, and the iterative model may incrementally add features.

iii. Agile Methodologies

- Focus on **collaboration, adaptability**, and delivering value quickly.
- Breaks development into **sprints** or **iterations**, each producing a functional deliverable.
- **Key Principles**:
  - Prioritize individuals and interactions over processes.
  - Welcome changing requirements, even late in development.
  - Deliver working software frequently (every few weeks).
- Popular Frameworks: Scrum, Kanban, Extreme Programming (XP).

Semicircular Arrow For Agile Development Methodology



Dataflow

Although the imperative paradigm seeks to build software in terms of procedures or functions, a way of identifying those functions is to consider the data to be manipulated rather than the functions themselves.

Dataflow analysis leads to the identification of functions.

A dataflow diagram is a means of representing the information gained from such dataflow studies.

- Assist in identifying procedures during the design stage of software development,

- useful when trying to **gain an understanding** of the proposed system during the analysis stage.
- improves **communication** between clients and software engineers

Dataflow

In a dataflow diagram,

- **arrows** represent **data paths**,
- **ovals** represent **points at which data manipulation occurs**, and
- **rectangles** represent **data sources** and **stores**.

Unified Modeling Language -UML

Modern **collection** of tools

- Has been developed with the object-oriented paradigm in mind.

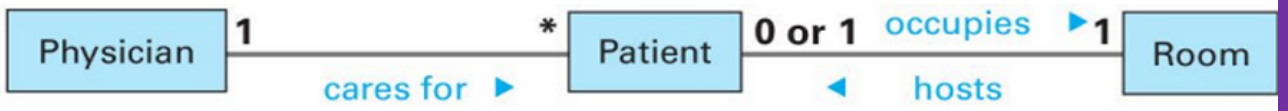
[redacted]

- Captures the image of the proposed system from the user's point of view

- Depicts the proposed system as a large rectangle in which
  - **interactions** (called use cases) **between the system and its users** are represented as **ovals** and
  - **users** of the system (called actors) are represented as **stick figures** (even though an actor may not be a person).

A notational system for representing the structure of classes and relationships between classes.

- **Classes** are represented by **rectangles** and
- **Associations** are represented by **lines**.
  - Association lines may or may not be **labeled**.
  - If they are labeled, a bold **arrowhead** can be used to indicate the **direction**.
  - A class diagram can also convey the multiplicities of those associations.



Use Case Diagram: Depicts the interactions between the users (actors) of a system and the system itself (use cases).

- **Example:** In an e-commerce system, a use case might be "Purchase a product". The actor would be the customer.

Class Diagram: Represents the classes in a system, their attributes, and the relationships between them.

- **Example:** In a library system, a class diagram might show classes like "Book", "Member", and "Library", and their relationships (e.g., a book can be borrowed by a member).

Sequence Diagram: Illustrates the interactions between objects over time, showing the sequence of messages exchanged.

- **Example:** A sequence diagram for an online order might show the sequence of interactions between a customer, a shopping cart, and a payment gateway.

Statechart Diagram: Models the behavior of an object over its lifetime, showing the different states it can be in and the transitions between those states.

- **Example:** A statechart diagram for an order might show states like "Placed", "Shipped", and "Delivered".

Activity Diagram: Represents the workflow of a system, showing the sequence of activities and decisions.

- **Example:** An activity diagram can be used to model the process of registering a new user on a website.

SOFTWARE TESTING DOCUMENTATION

User documentation

The purpose of user documentation is to explain the features of the software and describe how to use them.

- intended to be read by the user of the software and

is therefore expressed in the terminology of the application.

- an important marketing tool
- help packages

System documentation

The purpose of system documentation is **to describe the software's internal composition** so that the software can be **maintained later** in its life cycle.

2 major components of system documentation:

- The source version of all the programs in the system
- The design documents including
  - the software requirements specification and
  - records showing how these specifications were

obtained during design

Technical documentation

The purpose of technical documentation is to describe **how a software system should be installed and serviced**

- (such as adjusting operating parameters, installing updates, and reporting problems back to the software's developer).

The distinction between technical documentation and user documentation is blurred in the PC arena

- Especially for the cases where the user is the person who also installs and services the software
- However, in multiuser environments, the distinction **sharper.**