4.1.2 Programming Paradigms

What is a Programming Paradigm?

A **programming paradigm** is a style or approach to writing computer programs.

It describes:

- How programs are structured.
- How data is represented.
- How instructions are organised.

Two important programming paradigms in A Level Computer Science are:

1. **Procedural programming**
2. **Object-oriented programming (OOP)**

Procedural Programming

Procedural programming is a programming paradigm where a program is organised as a sequence of instructions and procedures (subroutines).

The program is broken down into smaller procedures/functions that perform specific tasks.

Characteristics of Procedural Programming:

- Sequence
- Selection
- Iteration
- Uses of procedures and functions

Structured Approach to Program Design and Construction

A **structured approach** means designing a program in a clear, organised way by dividing it into smaller sections.

This process is also called:

- **Top-down design**
- **Stepwise refinement**
- **Decomposition**

Hierarchy chart

A **hierarchy chart** is a diagram used during program design to show the structure of a program.

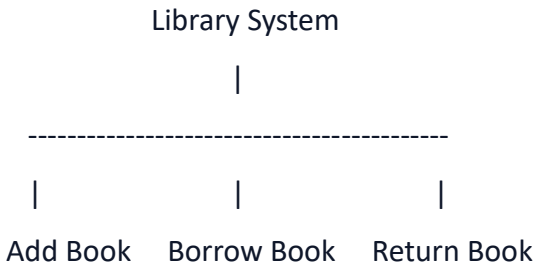
It shows:

- The main program.
- The subroutines/modules it calls.
- The relationship between different sections.

It does **not** show detailed code or algorithms.

Example Hierarchy Chart

A simple library management system:



Advantages of the Structured Approach

- **Easier to Understand.** Breaking a large problem into smaller sections makes the program easier to understand. A programmer can focus on one part at a time.
- **Easier Testing and Debugging.** Each module can be tested separately.
- **Code Reuse.** Subroutines can be called multiple times.
- **Easier Maintenance.** Changes can be made to individual modules without affecting the entire program.
- **Team Development.** Different programmers can work on different modules. Example:
    - Programmer A → Login system
    - Programmer B → Payment system
    - Programmer C → Database system
    - The modules can later be combined.
- **Reduces Complexity.** Large problems can become overwhelming. Decomposition reduces complexity by creating smaller problems that are easier to solve.

Object-Oriented Programming (OOP)

Object-oriented programming is a programming paradigm where programs are organised around **objects**.

An object represents something from the real world and contains:

- **Attributes** (data)
- **Methods** (functions that operate on the data)

Key Concepts of OOP

Classes

A **class** is a blueprint for creating objects.

It defines:

- The data an object stores.
- The actions an object can perform.

```
class Employee:
    def __init__(self, name, position):
        self.__name = name
        self.__position = position
        self.__hours_worked = 0
        self.__hourly_rate = 0

    def get_name(self):
        return self.__name

    def set_hours_worked(self, hours):
        self.__hours_worked = hours

    def set_hourly_rate(self, rate):
        self.__hourly_rate = rate

    def calculate_pay(self):
        return self.__hours_worked * self.__hourly_rate
```

Objects

An **object** is an instance of a class.

Instantiation

Instantiation is the process of creating an object from a class. This is done by calling the class's constructor.

```
employee1 = Employee("John Doe", "Software Engineer")
```

Private, Protected and Public Specifiers

- **Private** – Accessible only in the class in which it was defined
- **Protected** – Accessible in the class in which it defined or any subclass of that class
- **Public** – Accessible anywhere in the program

Encapsulation

Encapsulation means combining data and methods into one object and controlling access to the data.

Attributes are private and only accessible via public methods.

```
class Student:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def get_name(self):
        return self.__name
```

Inheritance

Inheritance allows one class to take characteristics from another class.

A subclass takes on the attributes and methods of a superclass and can declare additional

Attributes and methods of its own.

```
class Vehicle:
    def __init__(self, x, forward_speed):
        self._position = (x, 0)
        self._forward_speed = forward_speed

    def move(self):
        self._position[0] = self._position[0] + self._forward_speed


class Helicopter(Vehicle):
    def __init__(self, x, y, forward_speed, upward_speed):
        super().__init__(x, forward_speed)
        self._position = (x, y)
        self.__upward_speed = upward_speed

    def move(self):
        super().move()
        self._position[1] = self._position[1] + self.__upward_speed
```

`Helicopter` has inherited the attributes `position` and `forward_speed` from the `Vehicle` class. It has also inherited the method `move`.

It has also declared the additional attribute `upward_speed`.

Polymorphism

Polymorphism means different objects can respond differently to the same method call.

A method in a subclass overrides an inherited method to provide new behaviour.

In the code above, `Helicopter` has overridden the `move` method inherited from `Vehicle`. It extended it to also change its vertical position.

Abstract, Virtual and Static Methods

- **Abstract Method** – a method declared in a parent class but has no implementation. A child class MUST provide an implementation.
- **Virtual method** - a method in a parent class that can be overridden by a child class.
- **Static method** – a method that belongs to a class rather than to an object instance.

Relationships between objects

As well as inheritance, there can be other relationships between objects.

Aggregation and composition are both 'has' relationships. **i.e.** A folder has files**.** A car has wheels.

Aggregation is a weak association. If the containing object is deleted, the contained object still exists independently.

Composition is a strong association. If the containing object is deleted, the contained object is also deleted.

OOP design principles

Encapsulate What Varies

Keep parts of the program that may change separate from the parts that stay the same.

Example:

Instead of hard-coding tax calculations throughout a program:

- Place the calculation inside its own class or method.
- If tax rules change, only one place needs updating.

Favour Composition Over Inheritance

Where possible, use **composition** rather than inheritance.

Inheritance creates strong links between classes.

Composition allows objects to be combined more flexibly.

Program to Interfaces, Not Implementation

An **interface** specifies **what** an object can do, not **how** it does it.

Programs should depend on the interface rather than the specific class.

This allows one implementation to be replaced with another without changing the rest of the program.