BACHELORS IN COMPUTER ENGINEERING

OBJECT ORIENTED PROGRAMMING IN C++

Er. Ganga Gautam

CHAPTER 1

OBJECT ORIENTED CONCEPTS

Er. Ganga Gautam

Outline

1. **Object Oriented Programming Paradigm**
2. **A way of viewing World Agent**
3. **Procedure Oriented vs Object-Oriented Programming**
4. **Features of Object Oriented Programming:**
   - **Class and Object,**
   - **Data Abstraction,**
   - **Encapsulation,**
   - **Inheritance,**
   - **Polymorphism,**
   - **Message passing**
5. **Computation as Simulation,**
   - **Coping with Complexity and Abstraction Mechanisms**
6. **Object Oriented Analysis and Design:**
   - **Introduction,**
   - **Responsibility Driven Design (RDD),**
   - **Component Responsibility and Collaborator (CRC) Cards,**
   - **Responsibility Implies Non-Interference,**
   - **Programming in Small and Programming in Large**

OOP Paradigm

- Object-Oriented Programming (OOP) is a programming paradigm that organizes code into objects, which are instances of classes.
- In OOP, the focus is on creating reusable and modular code through the use of objects and their interactions.
- OOP promotes modularity, making it easier to understand, maintain, and extend code by breaking it into smaller, self-contained units.
- Popular programming languages that support OOP include Java, C++, Python, and C#.

OOP Paradigm

The main principles of OOP include encapsulation, inheritance, and polymorphism.

- Encapsulation allows data and methods to be bundled together within an object, providing data hiding and abstraction.
- Inheritance enables the creation of subclasses that inherit properties and methods from a superclass, promoting code reuse and hierarchy.
- Polymorphism allows objects to take on different forms or behaviors based on the context, providing flexibility and extensibility.

OOP Paradigm

Benefits of Object-Oriented Programming (OOP)

- Modularity: The modular nature of OOP makes code easier to understand, test, and maintain by breaking it into smaller, self-contained units.
- Encapsulation: Encapsulation protects data within objects, preventing direct access and promoting data integrity and security.
- Code organization: OOP promotes a more organized and structured approach to coding, making it easier for multiple programmers to collaborate on a project.
- Extensibility: OOP facilitates the addition of new features and functionality to existing code without modifying the existing codebase extensively.
- Flexibility: Polymorphism in OOP allows for the creation of code that can work with different types of objects, enhancing flexibility and adaptability.
- Code reusability: Inheritance enables code reuse by allowing subclasses to inherit properties and methods from a superclass.

A way of viewing world agent

- One way to view the world in an object-oriented programming (OOP) context is by considering the concept of agents.
- An agent can be an object or a class that encapsulates both data and behavior, representing an entity that interacts with the world.
- In this approach, we can think of various entities or objects in the world as agents that interact with each other and their environment.

A way of viewing world agent (Contd.)

Example

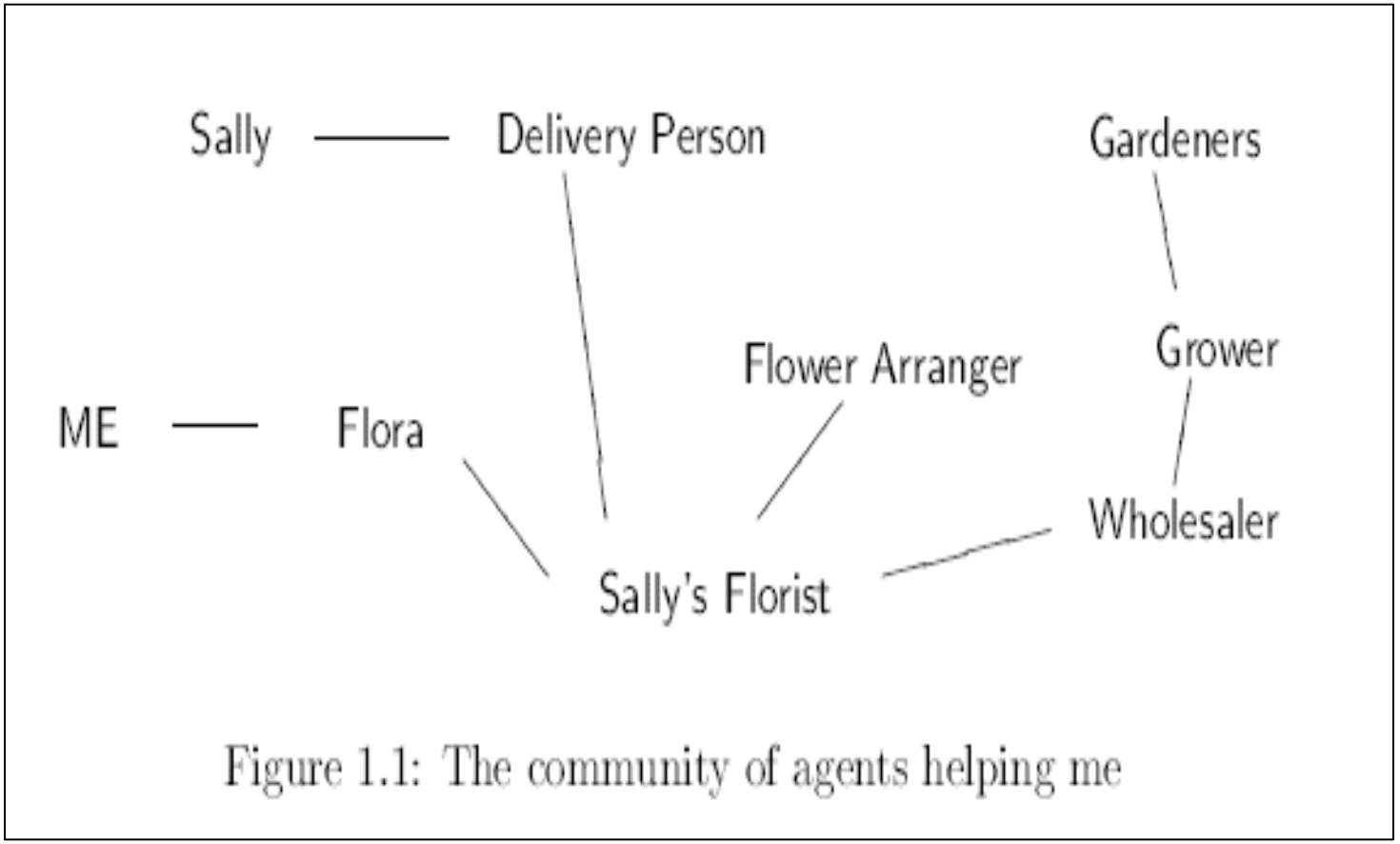


Figure 1.1: The community of agents helping me

A way of viewing world agent (Contd.)

- Suppose I would like to send flowers to a friend (say Sally) who lives in a city very far way to me.
- Because of the distance, there is no way to pick and flower and carry up to her. Nevertheless, sending her the flowers is an easy task.
- So, I would better go to a local florist (say Flora) and tell her the quantity and name of flower and also Sally's address.
- Now, I do not need to worry all about how the flower is sent.
- I can be assured that flowers are expediently and automatically sent by the florist.

A way of viewing world agent (Contd.)

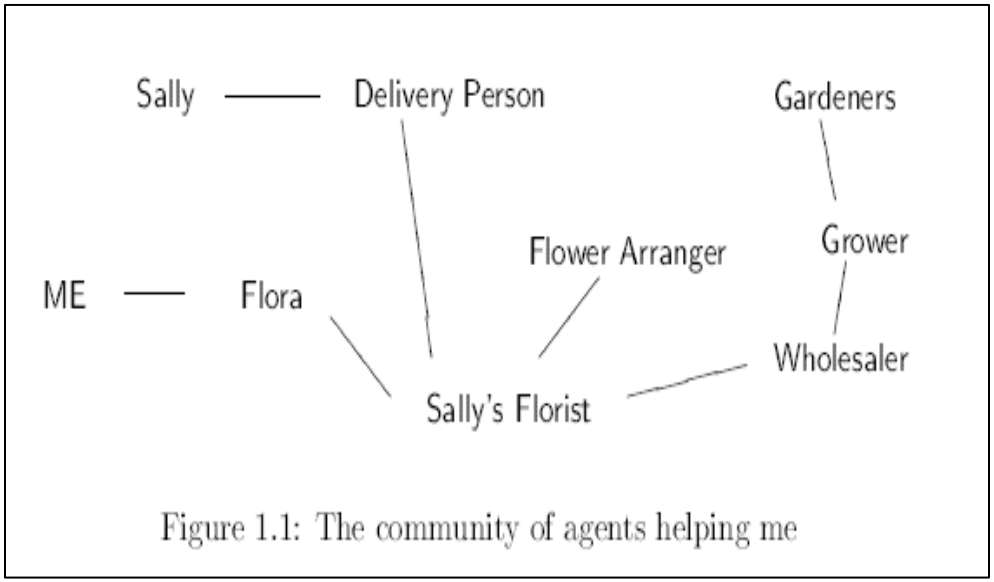


Figure 1.1: The community of agents helping me

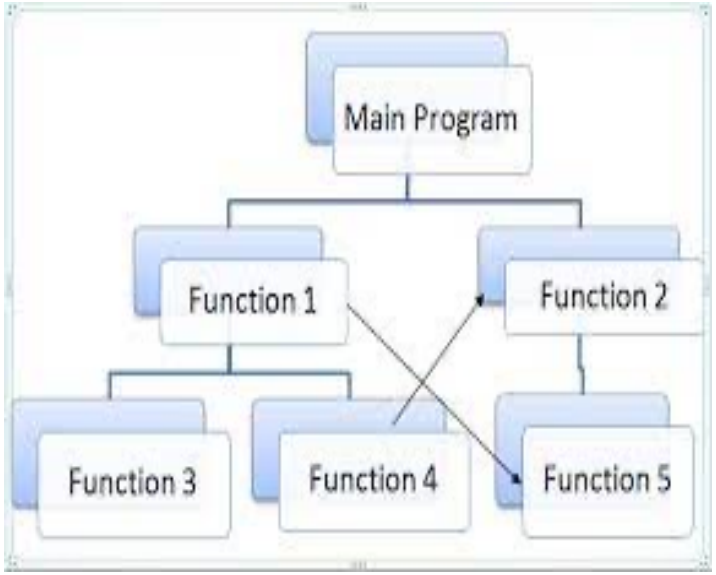
- In object-oriented programming, every action is initiated by passing a message to an agent (object), which is responsible for the action.
- The receiver is the object to whom the message was sent.
- In response to the message, the receiver performs some method to carry out the request.
- Every message may include any additional information as arguments.

Procedure Oriented Programming (POP)

- A program in procedural programming language is just a list of instructions like, get some input, add these numbers, divide by certain number, display the output.
- These each instruction says computer to do something and computer follows the instructions to give appropriate output.
- Some POP languages are: C, Pascal, QBASIC, FORTRAN, etc.

Features of POP

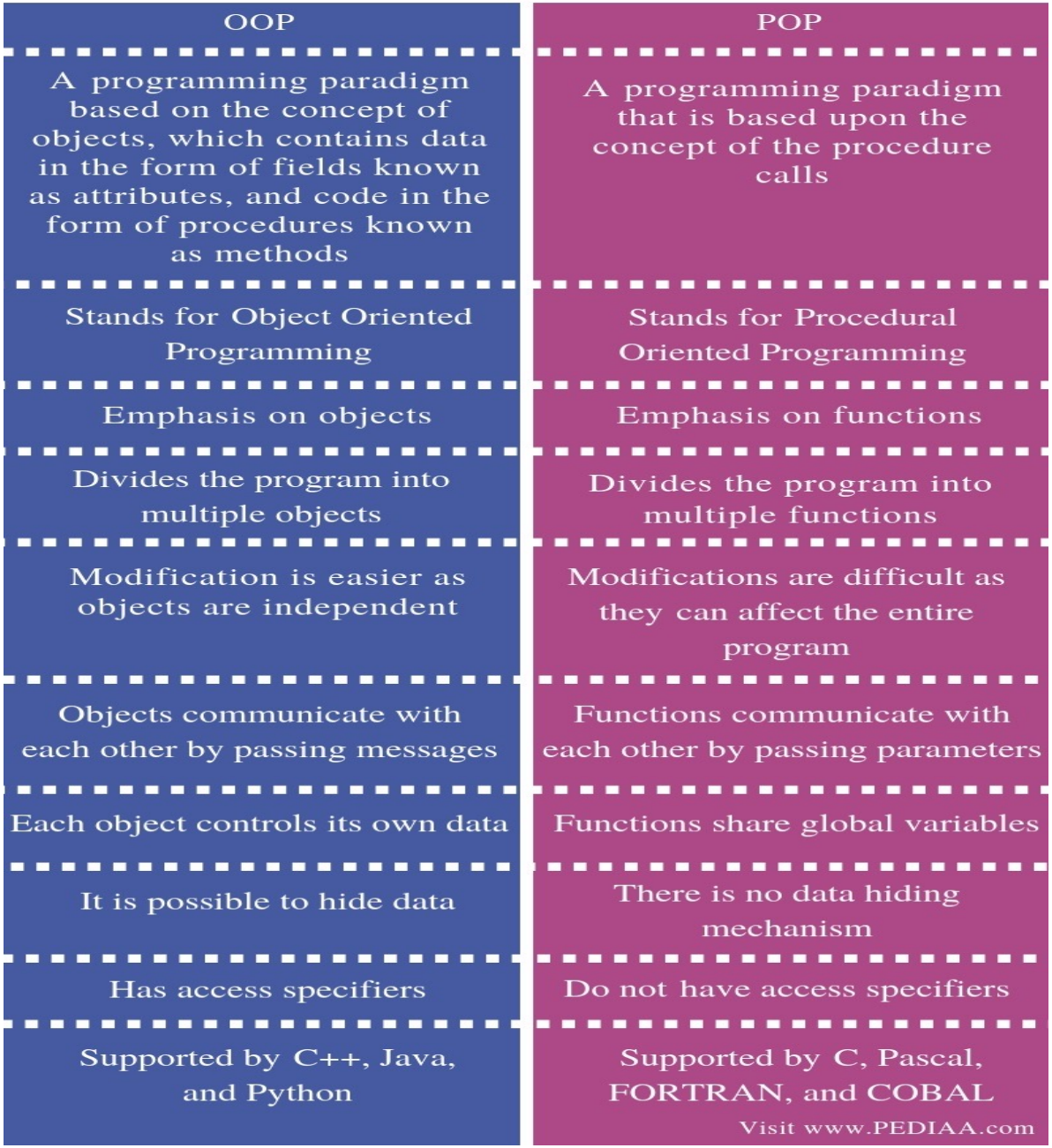
- Emphasis is on doing things, i.e. algorithms.
- Large programs are divided into individual functions. Each function has clearly defined purpose and interface to other functions in a program.
- Supports modular programming. Modules are the group of functions; so they have large entity than function.
- Most of the function share global data/variables.
- Functions transform data from one to another.
- Uses top-bottom approach in program design.



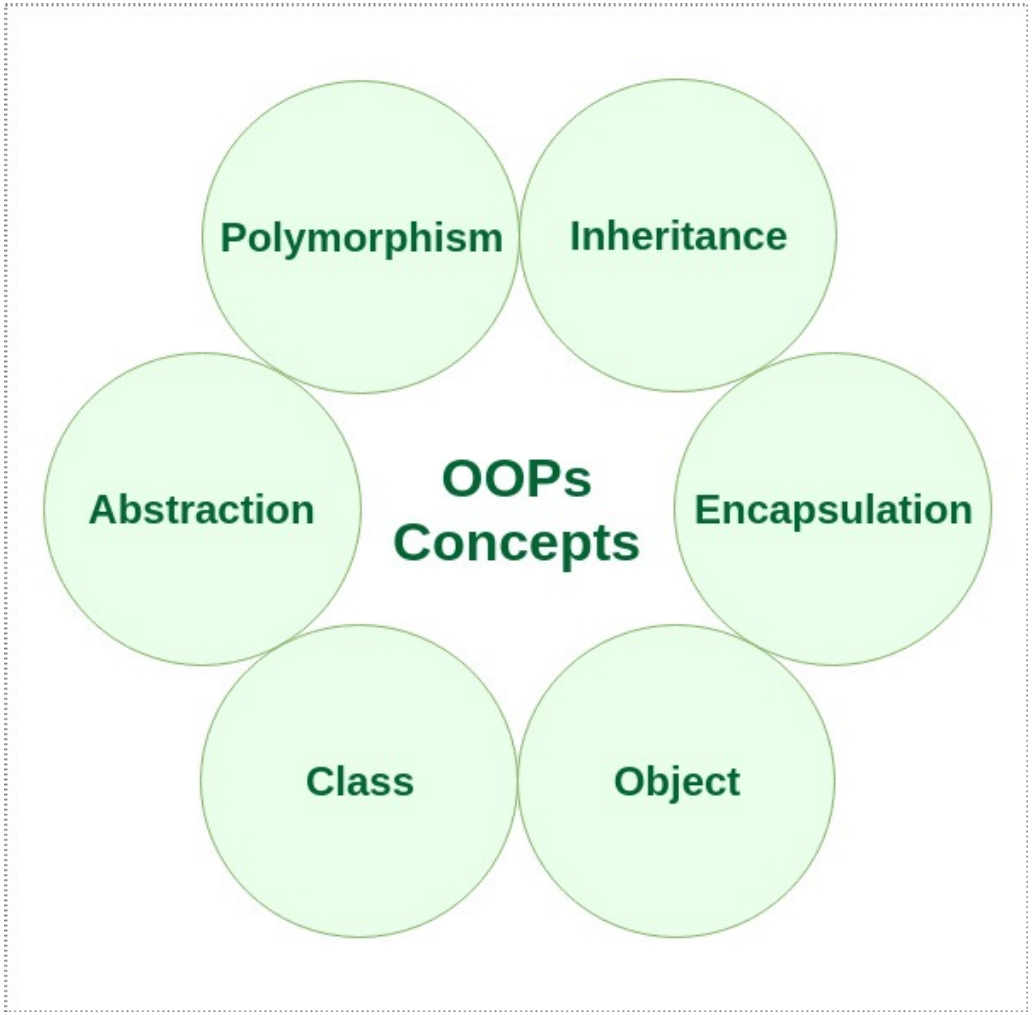
Object Oriented Programming (OOP)

- Object Oriented programming (OOP) is a programming technique that focuses on class and object concept.
- OOP focuses on data rather than procedures
- Programs are divided into objects.
- Functions and data are tied together in a single unit called class.
- Data can be hidden to prevent from accidental change by other functions or objects.
- Data structures are modeled as objects.
- Follows bottom-up approach of program design methodology.
- Some of the OOP languages are: C++, Java, Visual Basic, Python, C# and many others

OOP VERSUS POP



Features of OOP



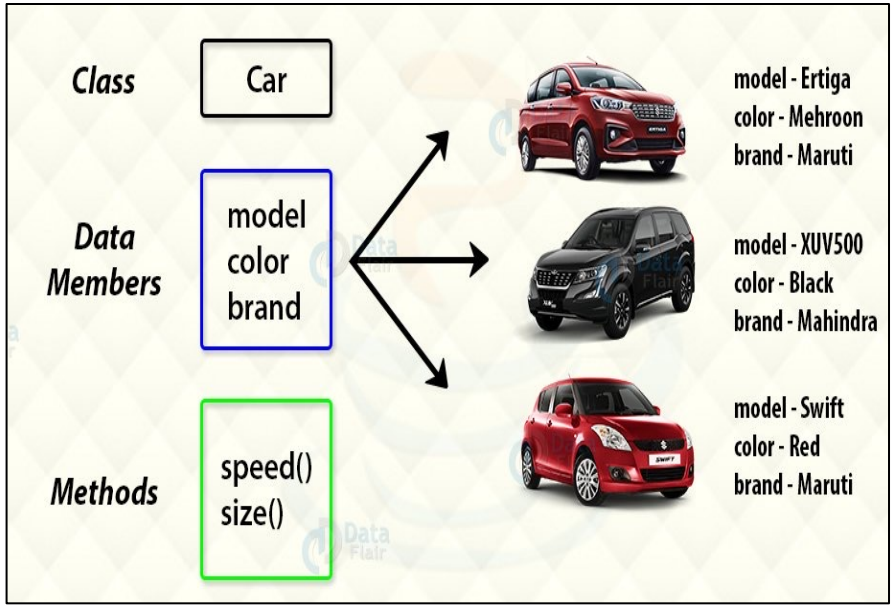
Concept of Class and Object

Class

- Class is a container that holds
  - Data members
  - Function(or methods) members
- It is like a blueprint for an object

Object

- Is an instance of class



Abstraction

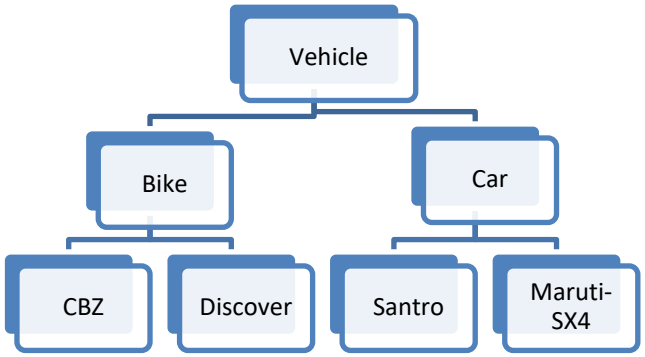
- **Abstraction** is one of the fundamental principle of OOP which helps to reduce the complexity.
- For example, a car is made from different components but we look and feel only its car, not different components

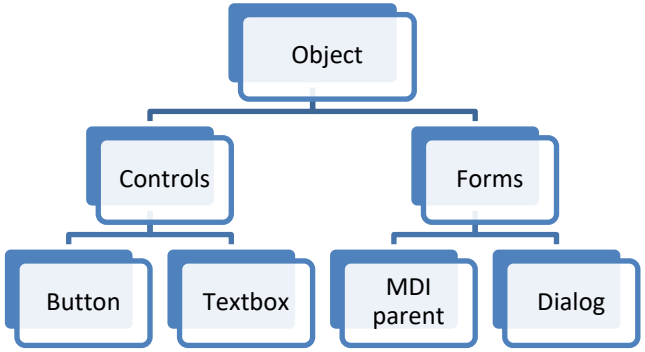
Encapsulation

- **Encapsulation** is the mechanism that binds code and data together and keeps both safe from outside misuse.
- It is the protective wall to prevent code inside class being randomly accessed.

* Note : abstraction and encapsulation seem to be similar but abstraction uses interface and encapsulation uses access modifiers(**private, public,protected**) to hide data.

Inheritance





Objects: **Button save , exit;**

- Different kinds of objects often have a certain amount in common with each other
- Bike, Car class inherit from **Vehicle** class. CBZ, Discover inherit from **Bike** Class
- The possible object for here becomes particular CBZ bike with particular engine number.
- Here **Vehicle** is **Super Class** for derived class Bike, and Bike is called **Base** or **derived** class
- In OOP, Inheritance is the way to reuse the code of existing class. Here one object acquires properties of another object.

Polymorphism

- **Polymorphism(“ Many Forms”)** is the process of making methods in OOP to perform multiple tasks.
- Eg: we can make a function called ADD() to calculate the sum of two numbers or two Strings with a single name. Making such methods is also called **Overloading**

Message Passing

- Message passing in OOP is a mechanism for objects to communicate and interact with each other by sending messages.
- It involves invoking methods on objects, which can lead to the exchange of information, execution of a specific behaviour, or modification of an object's state.

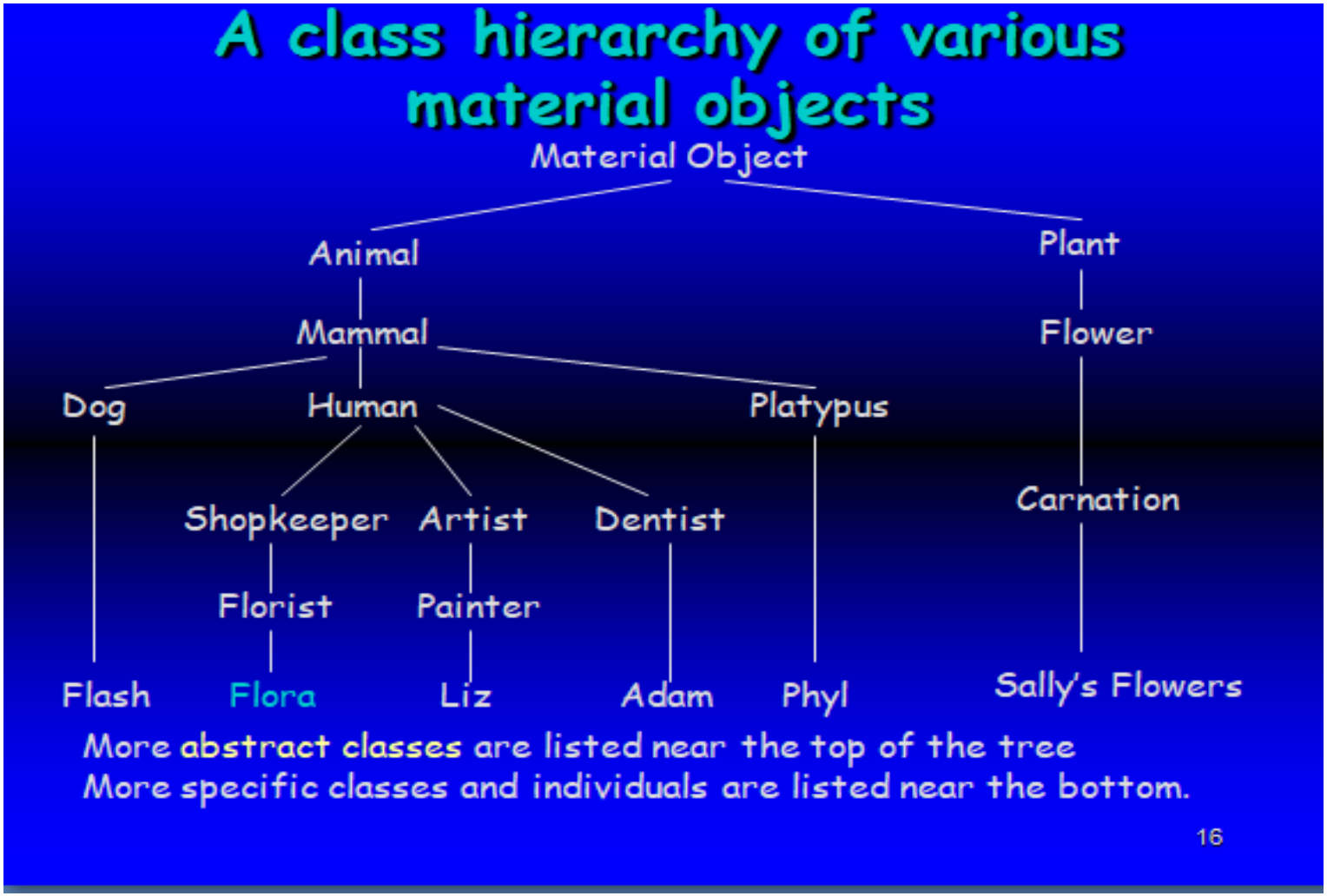
Object Oriented Design

- Object oriented design is the **art of assigning the right responsibilities** to the right objects and creating a clear structure with loose coupling and high cohesion.

Object Oriented Design

- creation of a universe of largely autonomous interacting agents.
- assigning the right responsibilities to the right objects and
- creating a clear structure with **loose coupling** and **high cohesion.**

Object Oriented Design



Responsibility-Driven Design (RDD)

- developed by *Rebecca Wirfs Brocles.*
- is a way to design
- emphasizes behavioral modeling using objects
- that emphasizes the identification and division of responsibilities within a collection of independent agents, responsibilities and collaborations.

RDD Steps

1. Working with scenario
2. Identification of components
3. Recording responsibility
4. What/who cycle
5. Documentation

RDD Steps

1. **Working with scenario:**
   - The first task is to refine specification.
   - In order to uncover the fundamental behavior of the system, the design team first creates a number of scenarios.
   - These scenarios will help to identify the components and refine the specifications.

RDD Steps (contd.)

2. Identification of components:

- The complex physical system is simplified by dividing the design into smaller units called ***software components.***
- A component is simply an abstract entity that can perform tasks- that is, fulfill some responsibilities

RDD Steps (contd.)

3. Recording responsibility

- At this stage, every activity that must take place is identified and assigned to some components as responsibility.
- CRC (Component-Responsibility-Collaborator) cards are used for recording responsibility.
- CRC card consists of:
  - the name of the software component,
  - the responsibilities of the component, and
  - the names of Collaborators (i.e. components which interact with that component).

RDD Steps (contd.)

4. What/who cycle

- First, the programming team identifies what activity needs to be performed next.
- This is immediately followed by answering the question of who performs the action.

RDD Steps (contd.)

5. **Documentation**

- Two documents should be essential parts of any software system: the **user manual** and the **system design documentation.**
- The user manual describes the interaction with the system form the user's point of view.
- The design documentation records the major decisions made during software design.

CRC

- first introduced by Kent Beck and Ward Cunningham
- (Class-Responsibility-Collaborator) Card is a powerful object-oriented analysis technique.
- used in the collaborative design of a system.
- CRC card consists of : class, responsibility and collaborator.

CRC (contd.)

1. **Class**
   - A Class represents a collection of similar objects.
   - Objects are things of interest in the system being modeled.
   - They can be a person, place, thing, or any other concept important to the system at hand.
   - The Class name appears across the top of the CRC card.

CRC (contd.)

2. Responsibility

- A Responsibility is anything that the class knows or does.
- These responsibilities are things that the class has knowledge about itself, or things the class can do with the knowledge it has.
- For example, a person class might have knowledge (and responsibility) for its name, address, and phone number.
- Other example of responsibilities might be:
  - Default and parameterized constructor
  - *Display()* method to display time in HH:MM:SS
  - Operator overloading

CRC (contd.)

3. Collaborator

- A Collaborator is another class that is used to get information for, or perform actions for the class at hand.
- It often works with a particular class to complete a step (or steps) in a scenario.
- The Collaborators of a class appear along the right side of the CRC card.

CRC (contd.)

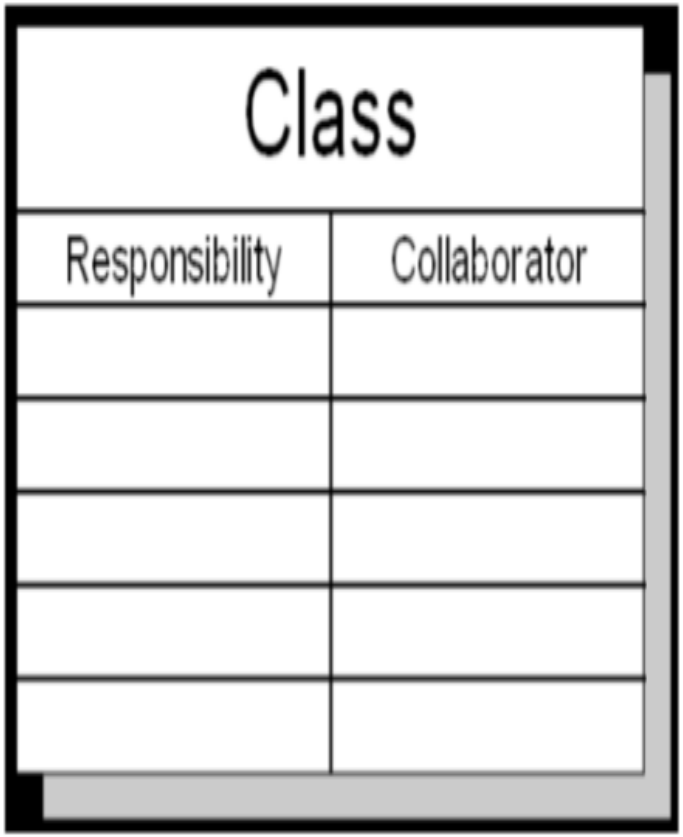


Figure 1: CRC Card Layout

Examination Division

Keeping student record	Administration Division
Conducting exams	Store
Publish result	

Computation as Simulation

- **Traditional model** describing the behavior of computer executing a program in process-state or pigeon-hole model.
- In this view, computer is data manager flowing some pattern of instructions, pulling values out of various memory slots, transforming them in some manner, and pushing the result in some other slot.

Computation as Simulation

- The focus shifts from low-level memory management and direct manipulation of data to a higher-level abstraction of entities called "objects."
- Objects encapsulate data and behavior within a single unit, and communication between objects occurs through messages.
- Rather than manipulating memory directly, objects interact with each other by sending messages and responding to those messages by performing their designated responsibilities (methods).

Computation as Simulation

- In Discrete Event- driven Simulation, the user creates computer models of various elements of the simulation, describes how they will interact with one another, and sets their moving.
- This model aligns closely with the principles of object-oriented programming.
- In simulation, entities or components of the system are represented as objects with their own attributes and behaviors.
- The interactions between objects in a simulation are typically modeled as messages sent between objects, similar to how objects communicate in an OO program.

- Let's take a simple example of a car simulation to illustrate the concept. We want to simulate the behavior of a car moving on a straight road. For this simulation, we can create a Car class and a Road class, each with its specific attributes and methods.

```
#include <iostream>
Using namespace std;

class Road {
private:
    float length;
public:
    Road(float len) : length(len) {}

    float getLength() const {
        return length;
    }
};
```

```
class Car {
private:
    float position;
    float speed;
public:
    Car() : position(0), speed(0) {}

    void accelerate(float acceleration) {
        speed += acceleration;
    }

    void move(float time) {
        position += speed * time;
    }

    float getPosition() const {
        return position;
    }
};
```

```
int main() {
    Road road(1000); // Road of length 1000 meters
    Car car;

    float time = 0; // Simulation time in seconds

    while (car.getPosition() < road.getLength()) {
        car.accelerate(2);
        car.move(1);
        cout << "Time: " << time << " sec, Car Position: " << car.getPosition()
<< " meters" << std::endl;

        time++;
    }
    return 0;
}
```

- In the main function, we create a Road object of length 1000 meters and a Car object. We then run a simulation loop where the car accelerates by 2 m/s^2 and moves for 1 second at each iteration until the car reaches the end of the road (its position becomes equal to or greater than the road length).
- Through this simulation, we can observe how the car behaves, accelerates, and moves on the road based on the defined rules of the simulation. The simulation provides a way to study the car's behavior in a virtual environment, which can be useful for testing or predicting real-world scenarios without the need for physical experimentation.

Coping with Complexity

- In earlier days, Programs were written in Assembly language by a single individual.
- Not considered large by today's standard.
- Program size gradually increased, difficult to remember all the information required for program/software development.
- Introduction of higher language.
- Problem's complexity increased.
- Team of programmers engaged together to solve complex problems.

Programming in small and programming in large

- For small system, code is developed by a single programmer or a small collection.
- single individual can understand all aspects of a project
- Solo programmer is responsible for entire project.

- Large software system is developed by a large team
- Requires proper management and communication.
- Requires Work-Break-Down structure

Abstraction mechanism

- Abstraction mechanism is to encapsulate and isolate design and extract information.
- Programmers had to deal with problem of complexity for long time.
- Object oriented techniques can be seen as natural outcome of a long historical progression from: procedure to modules to abstract data types and finally to objects.

Components and behaviors

- **A component is simply an abstract entity that performs task, i.e. fulfill some responsibilities**
- component must have a small well-defined set of responsibilities.
- component should interact with other components
- **Behavior is what the software can perform.**
- Once, the various behaviors have been identified and segregated, the system can be decomposed into subsystems

Roles of behavior in OOP

- The design process begins with the analysis of behavior.
- The behavior of a system is usually understood long before any other aspect.
- is something that can be described almost from the moment an idea is conceived
- Once, the various behaviors have been identified and segregated, the system can be decomposed into subsystems

Responsibility implies non-interference

- Whenever responsibility is assigned to an object (programmer), no interference should be done afterwards.
- Responsibility implies a degree of independence or noninterference
- if we tell a child that she is responsible for cleaning her rooms, we do not normally stand over her and watch
- When we make an object responsible for specific action, we expect a certain behavior.

Responsibility implies non-interference (contd.)

- One portion of code in a software system is often intimately tied by control and data connections to many other sections of the system.
- While passing the arguments to another function, we do not observe the called function.
- We rather expect the delegated task to be done.
- No interference should be done.

End of Chapter 1