Unit 1: Object Oriented Concepts (8 hrs)

Specific Objectives:
Understand the basic concepts of Object-Oriented Analysis and Design.

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
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.

1. Object Oriented Programming Paradigm

Object-oriented programming (OOP) is a programming paradigm based on the concept of "**objects**", which can contain data and code. The data is in the form of fields (often known as attributes or properties), and the code is in the form of procedures (often known as methods).

It is a way of designing and implementing software by organizing code into objects that represent real-world entities or concepts. OOP aims to improve the modularity, extensibility, and reusability of software by providing a structured way of modeling and interacting with complex systems.

A common feature of objects is that procedures (or methods) are attached to them and can access and modify the object's data fields. In this brand of OOP, there is usually a special name such as this or self-used to refer to the current object. In OOP, computer programs are designed by making them out of objects that interact with one another.

In OOP, objects are instances of classes, which define the properties and behaviors of the objects. Classes encapsulate data and behavior into a single unit, which can then be used to create objects with consistent attributes and actions. This allows developers to create complex systems by composing simple, reusable objects that interact with each other.

The four main principles of OOP are:

1. **Encapsulation:** This is the concept of hiding implementation details of an object and exposing only the relevant interface to the outside world. This allows the object to maintain its internal state and provides a layer of abstraction, which makes the code more maintainable and scalable.
2. **Inheritance:** This is the ability to create new classes based on existing ones, inheriting their properties and behaviors. This allows for code reuse and enables the creation of more specialized classes from more general ones.
3. **Polymorphism:** This is the ability of objects of different types to be used interchangeably. It allows for more flexibility and adaptability in software design by allowing objects to be manipulated and interacted with in a consistent way, regardless of their specific type.

4. **Abstraction:** This is the process of identifying and extracting common features of objects and generalizing them into abstract classes or interfaces. This allows for a more high-level view of the system, which can make it easier to design and maintain complex software.

OOP provides several benefits, including increased modularity, reusability, and maintainability of software. It allows developers to design and implement complex systems by breaking them down into simpler, more manageable pieces, and provides a consistent framework for organizing and manipulating those pieces.

OOP has several advantages over other programming paradigms, such as procedural programming. These advantages include:

- **Modularity:** OOP programs are easier to understand and maintain because they are divided into small, self-contained objects.
- **Reusability:** Objects can be reused in different programs, which can save time and effort.
- **Encapsulation:** OOP hides the implementation details of objects, which makes them easier to use and maintain.
- **Polymorphism:** OOP allows objects to be used in different ways, depending on their type.

OOP is a powerful programming paradigm that can be used to create complex and sophisticated software applications. It is a popular choice for many software developers because of its advantages in terms of modularity, reusability, encapsulation, and polymorphism.

Here are some examples of OOP in the real world:

- A car is an object. It has data, such as its make, model, and year, and it has methods, such as start, stop, and turn.
- A person is an object. It has data, such as its name, age, and gender, and it has methods, such as walk, talk, and eat.
- A bank account is an object. It has data, such as the account number, balance, and owner, and it has methods, such as deposit, withdraw, and transfer.

These are just a few examples of how OOP is used in the real world. OOP is a powerful paradigm that can be used to model real-world objects and systems.

C++ example that displays the message "Hello, world!" on the console:

```
#include <iostream>
int main() {
  std::cout << "Hello, world!" << std::endl;
  return 0;
}
```

Here's a brief explanation of what the code does:

- The **#include <iostream>** directive tells the compiler to include the input/output stream library, which provides functions for reading from and writing to the console and other input/output devices.
- The **int main()** function is the entry point of the program. When the program is executed, the operating system calls this function to start the program.
- Inside the **main()** function, **std::cout << "Hello, world!"** writes the message "Hello, world!" to the console using the **std::cout** object, which represents the standard output stream.
- The **std::endl** manipulator is used to insert a newline character and flush the output buffer.
- Finally, **return 0;** tells the operating system that the program exited successfully.

std::cout is an object of the **std::ostream** class in C++ standard library, which is used to write output to the console or other output devices.

In simple terms, **std::cout** is the standard output stream in C++, and it is used to write data to the console. To write data to the console using **std::cout**, you can use the **<<** operator to insert the data into the output stream. For example, **std::cout << "Hello, world!";** writes the string "Hello, world!" to the console.

Here's an example of how to use **std::cout** to write a variable value to the console:

```
#include <iostream>
```

```
int main() {
  int x = 42;
  std::cout << "The value of x is: " << x << std::endl;
  return 0;
}
```

The output will be: "The value of x is: 42" followed by a newline character.

std::cin is an object of the **std::istream** class in C++ standard library, which is used to read input from the console or other input devices.

In simple terms, **std::cin** is the standard input stream in C++, and it is used to read data from the console.

To read data from the console using **std::cin**, you can use the **>>** operator to extract the data from the input stream. For example, **std::cin >> x;** reads an integer value from the console and stores it in the variable **x**.

Here's an example of how to use **std::cin** to read an integer value from the console:

```
#include <iostream>
int main() {
  int x;
  std::cout << "Enter an integer value: ";
  std::cin >> x;
  std::cout << "You entered: " << x << std::endl;
  return 0;
}
```

*In this example, the program prompts the user to enter an integer value, and then reads the input using **std::cin**. The value is then stored in the variable **x** and printed to the console using **std::cout**.*

#include <iostream> is a preprocessor directive in C++ that tells the compiler to include the input/output stream library in the program. The input/output stream library provides functions for reading from and writing to the console and other input/output devices.

When the preprocessor encounters the **#include <iostream>** directive, it looks for the header file **iostream** in the standard library, and includes it in the program before it is compiled. This makes the functions and objects defined in **iostream** available for use in the program.

The **iostream** library provides two main classes, **std::istream** and **std::ostream**, which are used for input and output respectively. These classes provide objects like **std::cin** and **std::cout**, which are used to read input from and write output to the console.

By including **iostream** in the program, we can use these objects and functions to perform input and output operations in C++. For example, we can use **std::cout** to write data to the console, and **std::cin** to read input from the console.

In C++, the **using namespace std;** statement is used to avoid explicitly qualifying the names of standard library components with the **std::** prefix.

The **std** namespace in C++ contains various classes, functions, and objects provided by the standard library, including **cout**, **cin**, and other input/output related components. By default, these components are part of the **std** namespace.

Without the **using namespace std;** statement, you would need to explicitly use the **std::** prefix to access components from the **std** namespace. For example, instead of **cout << "Hello";**, you would need to write **std::cout << "Hello";**.

By including the **using namespace std;** statement at the beginning of your code, you can simply use the components from the **std** namespace without explicitly specifying the **std::** prefix. This allows you to write **cout << "Hello";** directly.

It's worth noting that using **using namespace std;** can lead to name clashes if you have identifiers with the same name as those in the **std** namespace. Therefore, it is generally recommended to avoid using **using namespace std;** in header files or in larger codebases to maintain better code clarity and avoid potential conflicts. Instead, you can either explicitly qualify the names with **std::** or selectively bring in specific components using the **using** statement.

```
#include<iostream>
using namespace std;
int main()
{
   cout<<"Hello World!!"<<endl;
   cout<<"My name is Deepak Bhatta";
   return 0;
}
```

You can assign an alternative name to the **std** namespace by using a namespace alias. This allows you to refer to the **std** namespace by a different name throughout your code.

Here's an example of how to create an alias for the **std** namespace:

```
#include <iostream>

namespace mylib = std;

int main() {
  mylib::cout << "Hello, world!" << mylib::endl;
  return 0;
}
```

In this example, **mylib** is an alias for the **std** namespace. By using the **mylib::** prefix, we can access the components of the **std** namespace. So, **mylib::cout** is equivalent to **std::cout**, and **mylib::endl** is equivalent to **std::endl**.

In C++, the **using** keyword has multiple purposes:

1. **Namespace Using Directive:** The **using** keyword can be used to bring an entire namespace into scope. This is called a namespace using directive. It allows you to access the members of a namespace without explicitly using the namespace prefix.

   Example:

```
using namespace std;  // Brings the entire std namespace into scope

int main()
 {
   cout << "Hello, world!" << endl;  // No need for std::
   return 0;
}
```

In this example, the **using namespace std;** statement brings the entire **std** namespace into scope. This allows direct use of **cout** and **endl** from the **std** namespace without explicitly using the **std::** prefix.

It's important to note that using namespace directives should be used judiciously to avoid potential naming conflicts and to maintain code clarity.

2. **Name Using Declaration:** The **using** keyword can also be used to bring specific names from a namespace into scope. This is called a name using declaration. It allows you to selectively import specific names from a namespace, rather than bringing the entire namespace into scope.

   Example:

```
#include <iostream>
using std::cout;  // Brings only cout from the std namespace into scope

int main() {
   cout << "Hello, world!" << std::endl;  // No need for std::
   return 0;
}
```

In this example, the **using std::cout;** statement imports only the **cout** identifier from the **std** namespace into scope. Therefore, we can use **cout** directly without the need for the **std::** prefix. Note that **std::endl** still requires the **std::** prefix since it was not imported explicitly.

The **using** keyword, whether for namespace using directives or name using declarations, helps simplify code by reducing the need for repetitive namespace prefixes, but it should be used with care to avoid potential naming conflicts and ambiguity in larger codebases.

2. A way of viewing World Agent

Object-oriented programming (OOP) is a programming paradigm that models real-world problems by organizing code into objects that represent entities or concepts. In OOP, objects have methods, behaviors, and responsibilities, which are used to define how the objects interact with each other and with the rest of the system. Let's see how OOP models the real-world problem with reference to agents, methods, behaviors, and responsibilities:

1. **Agents:** In OOP, agents are represented by objects. An object can be thought of as a software representation of a real-world entity or concept, such as a person, a car, a book, or a bank account. Each object has its own state, which consists of attributes that describe the object, and methods that define the behavior of the object.

For example, consider a banking system. In this system, we can model each bank account as an object. Each account object has its own attributes, such as account number, account holder name, balance, etc., and its own methods, such as deposit, withdraw, and transfer.

2. **Methods:** Methods define the behavior of an object. They are the functions or procedures that are associated with an object and are used to manipulate its state or perform some action. Methods can be public, private, or protected, depending on their visibility and access levels.

For example, consider the deposit method of a bank account object. This method takes a parameter for the amount to be deposited and adds that amount to the account balance. The withdraw method, on the other hand, subtracts a specified amount from the account balance.

3. **Behaviors:** Behaviors are the actions that an object can perform. They define how an object interacts with other objects in the system. In OOP, behaviors are implemented using methods, which define what an object can do.

For example, consider a car object in a traffic simulation system. The car object can perform behaviors such as accelerating, braking, turning, and changing lanes. These behaviors are implemented using methods that define the movement and navigation of the car object.

4. **Responsibilities**: Responsibilities are the tasks or functions that an object is responsible for in the system. Each object has a specific set of responsibilities, which define what it can do and how it interacts with other objects.

For example, consider a user object in a social media platform. The user object has responsibilities such as creating posts, commenting on posts, and liking posts. These responsibilities are defined by the methods associated with the user object.

In summary, object-oriented programming models real-world problems by organizing code into objects that represent entities or concepts. Objects have methods, behaviors, and responsibilities, which are used to define how they interact with other objects and with the rest of the system.

By using OOP principles, developers can create software that is easier to design, develop, test, and maintain, and that better reflects the complexity and diversity of the real world.

Why OOP is known as a new paradigm?

OOP is known as a new programming paradigm because it is a relatively recent development in the history of programming. The first object-oriented programming language, Simula, was developed in the early 1960s, and the first pure object-oriented programming language, Smalltalk, was developed in the early 1970s. OOP gained popularity in the 1980s and 1990s, and today, most modern programming languages support object-oriented programming.

OOP is a new paradigm because it breaks away from the traditional procedural programming paradigm. In procedural programming, **code is organized into functions**, which are executed one after the other. In OOP, **code is organized into objects**, which are self-contained units of data and behavior. Objects can interact with each other through messages, which are simply requests for information or action.

OOP solves this problem by organizing programs around objects. Objects are self-contained units of data and code, and they can be reused in different programs. This makes OOP programs easier to maintain and extend.

OOP is also known for its ability to model real-world objects. Objects can be used to represent anything from cars to people to animals. This makes OOP a very powerful tool for software development.

OOP is now the dominant programming paradigm, and it is used to develop a wide variety of software, including operating systems, web browsers, and video games.

Here are some of the advantages of OOP:

- **Modularity:** Objects are self-contained units of data and code, which makes them easier to understand and maintain.
- **Reusability:** Objects can be reused in different programs, which saves time and effort.
- **Encapsulation:** Objects can hide their implementation details, which makes them easier to use.
- **Inheritance**: Objects can inherit properties and methods from other objects, which reduces code duplication.
- **Polymorphism:** Objects can be used interchangeably, even if they have different implementations, which makes programs more flexible.

Here are some of the disadvantages of OOP:

- **Complexity:** OOP can be more complex than procedural programming, which can make it difficult to learn.
- **Overhead:** OOP can add some overhead to programs, which can reduce their performance.
- **Heterogeneity:** OOP programs can be more heterogeneous than procedural programs, which can make them more difficult to debug.

Overall, OOP is a powerful programming paradigm that has many advantages. However, it is important to be aware of the disadvantages before using it.

Is OOP is a new paradigms?

No, Object-Oriented Programming (OOP) is not a new paradigm in programming languages. It was introduced several decades ago and has become one of the most widely used and established programming paradigms.

OOP emerged in the late 1960s and gained significant popularity in the 1980s and 1990s. It introduced a different way of structuring and organizing code by emphasizing the concept of objects, which are instances of classes containing both data and behavior.

OOP brought several key principles and features, including encapsulation, inheritance, and polymorphism, that aimed to improve code organization, reusability, and maintainability. The approach of designing software systems around objects and their interactions provided a more modular and intuitive way of programming.

Since its inception, OOP has influenced the design and development of numerous programming languages, libraries, and frameworks. Many widely used programming languages, such as Java, C++, C#, Python, and Ruby, are object-oriented or support object-oriented features.

While OOP was once considered a "new" paradigm, it has now become a well-established and mainstream programming paradigm. It is no longer on the cutting edge of new programming paradigms but remains a fundamental and widely used approach in software development.

Note:

In the context of computer science and programming, a paradigm refers to a fundamental style or approach to designing, structuring, and implementing software. It encompasses a set of principles, concepts, and practices that guide the development process and determine how programs are written and organized.

A programming paradigm provides a way of thinking about and solving problems using a particular set of techniques and abstractions. It defines the rules and conventions that govern how code is written, how data is represented and manipulated, and how programs are executed.

Different programming paradigms have emerged over the years, each with its own characteristics and advantages. Some common programming paradigms include:

1. **Procedural Programming:** This paradigm focuses on organizing code into procedures or functions that perform specific tasks. It emphasizes sequential execution and the manipulation of shared data. *Languages like Pascal, COBOL, FORTAN, C are used for Procedural programming.*

2. **Object-Oriented Programming (OOP):** OOP emphasizes the organization of code around objects, which encapsulate both data (attributes) and behavior (methods). It promotes code reusability, modularity, and the modeling of real-world concepts. *Languages like C++, Java, Python, C#, PHP are used for OOP programming.*

3. **Functional Programming:** Functional programming treats computation as the evaluation of mathematical functions. It emphasizes immutability, the avoidance of mutable state, and the use of pure functions without side effects. *Languages like JavaScript, Haskell, Python are used for functional programming.*

4. **Logical Programming:** Logical programming is based on formal logic and emphasizes the use of rules and facts to derive new information. *Languages like Prolog are commonly used for logical programming.*
5. **Declarative Programming:** Declarative programming focuses on describing what should be achieved, rather than specifying how it should be done. It provides a higher level of abstraction and allows the underlying system to handle the details. *Languages like SQL, Prolog, HTML are commonly used for Declarative programming.*
6. **Concurrent Programming:** Concurrent programming deals with the execution of multiple tasks or processes simultaneously. It addresses challenges related to synchronization, communication, and resource sharing in concurrent systems. *Languages like Java (Threads), Python (Threading Module), Go (Goroutines) are commonly used for Concurrent programming.*

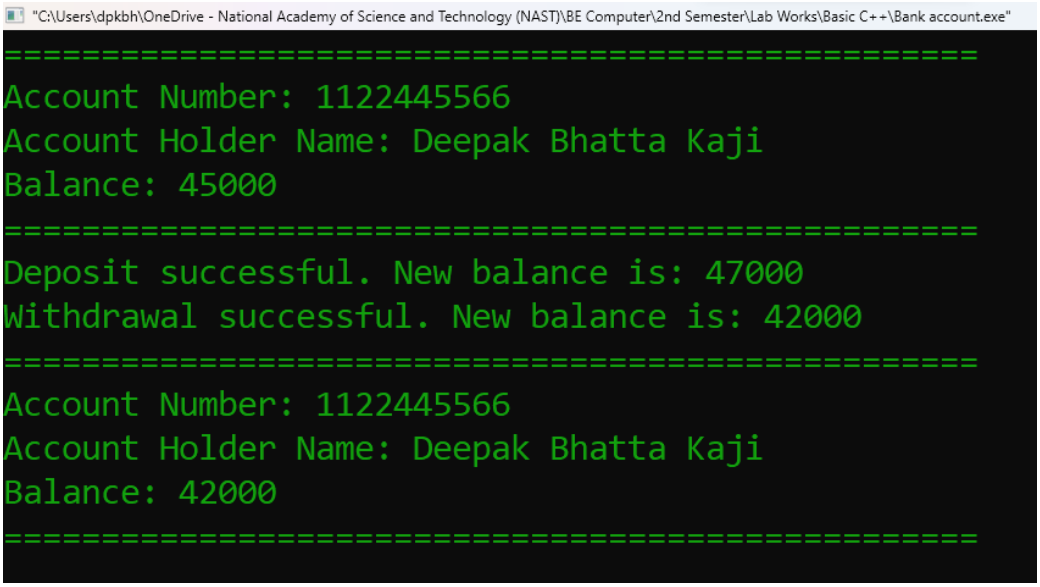
Each programming paradigm has its own strengths and weaknesses, making it suitable for different types of problems and programming scenarios. Programmers often choose a paradigm based on the nature of the problem at hand and the desired trade-offs between factors such as code readability, performance, and maintainability.

It's worth noting that programming languages can support multiple paradigms or provide features that enable developers to work in different paradigms within a single language. This flexibility allows programmers to choose the most appropriate paradigm or combination of paradigms for their specific needs.

An example of a Bank Account class implemented in C++:

```
#include <iostream>
#include <string>
using namespace std;
class BankAccount {
private:
    string accountNumber;
    string accountHolderName;
    double balance;
public:
```

```
    BankAccount(string accNum, string accHolderName, double bal) {
        accountNumber = accNum;
        accountHolderName = accHolderName;
        balance = bal;
    }
    void deposit(double amount) {
        balance += amount;
        cout << "Deposit successful. New balance is: " << balance << endl;
    }
    void withdraw(double amount) {
        if (amount > balance) {
            cout << "Error: Insufficient balance." << endl;
        } else {
            balance -= amount;
            cout << "Withdrawal successful. New balance is: " << balance << endl;
        }
    }
    void displayAccountInfo() {
        cout << "================================================="<<endl;
        cout << "Account Number: " << accountNumber << endl;
        cout << "Account Holder Name: " << accountHolderName << endl;
        cout << "Balance: " << balance << endl;
        cout << "================================================="<<endl;
}
};
int main() {
    BankAccount myAccount("1122445566", "Deepak Bhatta Kaji", 45000.0);
    myAccount.displayAccountInfo();
    myAccount.deposit(2000.0);
    myAccount.withdraw(5000.0);
    myAccount.displayAccountInfo();
    return 0;
}
```



In this example, we have defined a BankAccount class with three private member variables: **accountNumber**, **accountHolderName**, and **balance**.

```
private:
    string accountNumber;
    string accountHolderName;
    double balance;
```

This line of code defines the constructor for the **BankAccount** class. The constructor takes three parameters:

- **accNum**: a **string** representing the account number
- **accHolderName**: a **string** representing the account holder's name
- **bal**: a **double** representing the initial balance of the account

The constructor initializes the corresponding member variables **accountNumber**, **accountHolderName**, and **balance** with the values passed in as arguments.

```
public:
    BankAccount(string accNum, string accHolderName, double bal) {
```

These lines of code define the private member variables of the **BankAccount** class.

- **accountNumber**: a **string** representing the account number
- **accountHolderName**: a **string** representing the account holder's name
- **balance**: a **double** representing the current balance of the account

Declaring these variables as **private** makes them inaccessible from outside the class, preventing any accidental or unauthorized modifications to their values. Access to these

variables is provided through public member functions such as **deposit**, **withdraw**, and **displayAccountInfo**.

The constructor is declared **public**, which means it can be called from outside the class to create new **BankAccount** objects.

The class also has three public member functions: **deposit()**, **withdraw()**, and **displayAccountInfo()**, which correspond to the behaviors of a bank account.

The **deposit()** method takes a **double** parameter for the amount to be deposited, adds the amount to the current balance, and prints the new balance to the console.

The **withdraw()** method takes a **double** parameter for the amount to be withdrawn, checks if the amount is greater than the current balance, and if it is, it displays an error message. Otherwise, it subtracts the amount from the current balance and prints the new balance to the console.
The **displayAccountInfo()** method simply prints out the account number, account holder name, and current balance to the console.

In the **main()** function, we create an instance of the **BankAccount** class called **myAccount** with an initial balance of $5000. We then display the account information, deposit $2500, withdraw $1000, and display the updated account information.

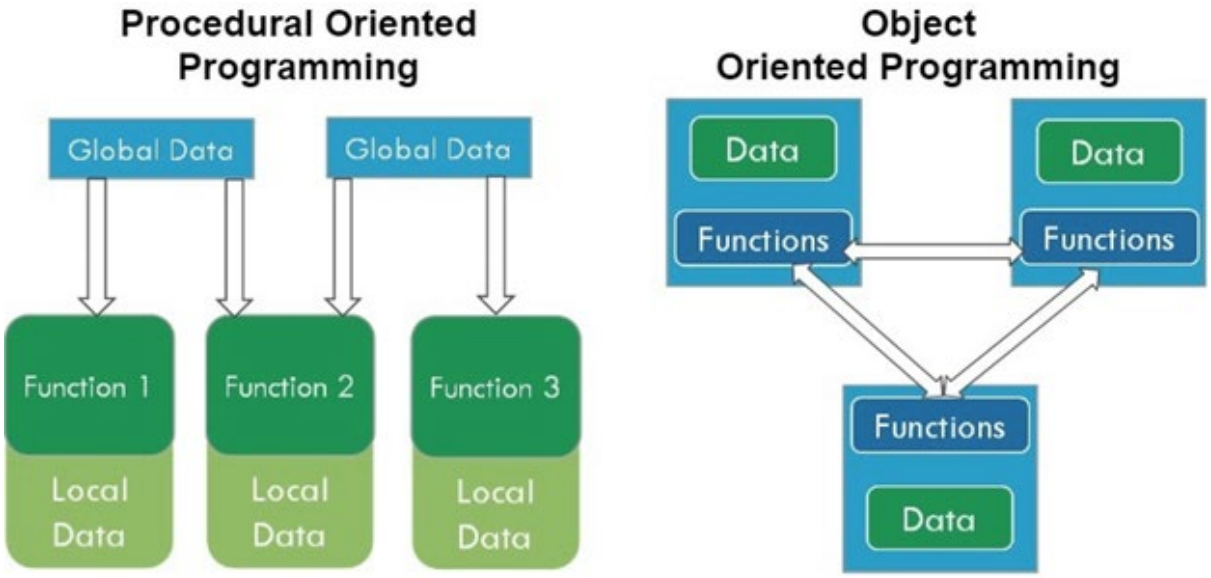
This example demonstrates how the BankAccount class models a real-world problem by representing a bank account as an object with attributes and behaviors.

---

3. Procedure Oriented vs Object-Oriented Programming

Procedural Programming Language	Object Oriented Programming Language
1. Program is divided into functions.	1. Program is divide into classes and objects..
2. The emphasis is on doing things.	2. The emphasis on data.
3. Poor modeling to real world problems.	3. Strong modeling to real world problems.
4. It is not easy to maintain project if it is too complex.	4. It is easy to maintain project even if it is too complex.
5. Provides poor data security.	5. Provides strong data Security.
6. It is not extensible programming language.	6. It is highly extensible programming language.
7. Productivity is low.	7. Productivity is high.
8. Do not provide any support for new data types.	8. Provide support to new Data types.
9. Unit of programming is function.	9. Unit of programming is class.
10. Ex. Pascal , C , Basic , Fortran.	10. Ex. C++ , Java , Oracle.

Criteria	Procedure-Oriented Language	Object-Oriented Language
Programming Paradigm	Structured Programming	Object-Oriented Programming
Data Focus	Functions/Procedures operate on data	Data and methods are combined into objects
Data Abstraction	Not emphasized	Emphasized
Data Encapsulation	Not supported	Supported
Inheritance	Not supported	Supported
Polymorphism	Not supported	Supported
Program Control	Functions are the main control structure	Objects and their methods are the main control structure
Complexity Management	Harder to manage and update as program grows in size	Easier to manage and update as program grows in size
Code Reusability	Less reusable	More reusable
Examples	C, Pascal, Fortran	Java, Python, C++



Extra Learning:

Procedure-Oriented Programming (POP) is a programming paradigm that follows a top-down approach. In this approach, the program is structured as a hierarchy of procedures, where each procedure performs a specific task. The program execution starts from the main procedure and then calls other procedures as needed.

Here are some characteristics of a top-down approach in POP:

1. **Modular design:** The program is divided into smaller modules or functions, each of which performs a specific task. These modules can be developed independently and tested separately, making it easier to maintain and update the program.
2. **Step-wise refinement:** The program design is refined in a step-by-step manner, where each step involves breaking down the problem into smaller, more manageable parts. This allows for a more systematic approach to programming and helps to reduce complexity.
3. **Sequential execution:** The program executes sequentially, starting from the main procedure and then calling other procedures as needed. This makes it easy to follow the flow of the program and understand how each procedure contributes to the overall solution.

Overall, the top-down approach in POP is a systematic and modular way of designing programs, where the problem is broken down into smaller, more manageable parts. This makes it easier to write, test, and maintain the program over time.

Object-Oriented Programming (OOP) is a programming paradigm that follows a bottom-up approach. In this approach, the program is structured around objects, which are instances of classes. Each class defines a set of attributes (data) and behaviors (methods) that are used to represent and manipulate the objects.

Here are some characteristics of a bottom-up approach in OOP:

1. **Object-oriented design:** The program is designed around objects, which encapsulate both data and behavior. This allows for a more natural representation of real-world entities and helps to reduce complexity.
2. **Abstraction and encapsulation:** The program design emphasizes abstraction and encapsulation, where objects are defined at different levels of abstraction and data and behavior are hidden behind interfaces. This allows for a more modular and flexible design, where objects can be easily reused and modified without affecting other parts of the program.
3. **Polymorphism and inheritance:** The program design makes use of polymorphism and inheritance, which allow for more flexible and extensible code. Polymorphism allows different objects to respond to the same message in different ways, while inheritance allows new classes to be created by inheriting and extending the properties of existing classes.

Overall, the bottom-up approach in OOP is a flexible and modular way of designing programs, where the program is built around objects and their relationships. This allows for a more natural representation of real-world entities and supports extensibility and reuse of code.

4. Features of Object-Oriented Programming

Features of Object-Oriented Programming:

Class and Object, Data Abstraction, Encapsulation, Inheritance, Polymorphism, Message passing

Refer to Class Note

---

5. Computation as Simulation, Coping with Complexity and Abstraction

Computation as Simulation:

In the context of object-oriented programming (OOP), the view that computation is simulation refers to the idea that OOP allows us to model and simulate real-world systems or processes through the use of objects and their interactions.

In OOP, we define objects as instances of classes, where classes act as blueprints or templates that define the properties and behavior of objects. These objects interact with each other by sending messages and invoking methods to perform various operations and exchange data.

When we design an object-oriented program, we typically try to create a system that accurately represents and simulates the behavior and relationships of real-world entities. This simulation can involve modeling complex systems, such as a banking system, an online shopping platform, or a traffic simulation.

In object-oriented programming (OOP), objects are used to represent real-world entities. These objects have state and behavior, and they interact with each other to simulate the real world.
For example, a car object might have state such as its speed, fuel level, and location. It might also have behavior such as accelerating, braking, and turning. The car object would interact with other objects in the simulation, such as other cars, traffic lights, and pedestrians.
The view that computation is simulation is based on the idea that computers can be used to model the real world. This modeling can be used to understand the real world, to predict how it will behave, and to test new ideas.

OOP is a powerful tool for simulation because it allows us to create models that are both accurate and easy to understand. By using objects to represent real-world entities, we can create models that are closer to the real world than traditional programming languages.

OOP also makes it easy to create complex models that interact with each other. This is because objects can be easily combined to create new objects. For example, a car object can be combined with a driver object to create a driving simulation.

The view that computation is simulation has many benefits. It allows us to create models that are more accurate, easier to understand, and more complex than traditional programming languages. This can lead to better understanding of the real world, better predictions of future events, and better testing of new ideas.

Here are some examples of how OOP is used for simulation:

- **Weather forecasting:** Computers are used to simulate the atmosphere and predict future weather patterns.
- **Traffic simulation:** Computers are used to simulate traffic flow and predict traffic congestion.
- **Medical simulation:** Computers are used to simulate medical procedures and train surgeons.
- **Military simulation:** Computers are used to simulate military operations and train soldiers.
- **Financial simulation:** Computers are used to simulate financial markets and predict future stock prices.

These are just a few examples of how OOP is used for simulation. As computers become more powerful, we can expect to see even more sophisticated simulations being used in a wide variety of fields.

Here's how the view that computation is simulation can be applied in OOP:

1. Objects as Models: Objects in OOP often represent real-world entities or concepts. For example, in a banking system simulation, we can have objects representing customers, accounts, transactions, etc. These objects encapsulate data and behavior related to their real-world counterparts, allowing us to model and simulate their interactions and operations.

2. Interactions and Messaging: Objects interact with each other by sending messages and invoking methods. This mimics the communication and collaboration that occurs in real-world systems. For instance, in a shopping platform simulation, objects representing customers may send messages to objects representing products to add them to their cart or place an order.

3. Behavior and State: Objects in OOP have behavior (methods) and state (attributes). The behavior defines the actions an object can perform, while the state represents the object's data and its current state. By defining the behavior and state of objects accurately, we can simulate the dynamics and behavior of real-world systems.

4. Abstraction and Encapsulation: OOP allows us to abstract complex systems into manageable components (classes and objects) and encapsulate their behavior and data. This abstraction enables us to focus on modeling and simulating specific aspects of a system without worrying about the underlying implementation details. We can simulate a simplified version of a system while ignoring unnecessary complexities.

Coping with Complexity and Abstraction:

Coping with complexity and abstraction is an essential aspect of software development, and object-oriented programming (OOP) offers several mechanisms to help manage these challenges.

Complexity and abstraction are two of the main challenges that we face when working with computers. Complexity is the degree of difficulty in understanding or using something, while abstraction is the process of representing something in a simpler or more general form.

There are a number of techniques that we can use to cope with complexity and abstraction. One technique is to use decomposition, which is the process of breaking down a complex problem into smaller, more manageable problems. Another technique is to use abstraction, which is the process of representing something in a simpler or more general form.

Decomposition can be used to break down a complex problem into smaller, more manageable problems. For example, if we are trying to design a new car, we could decompose the problem into smaller problems such as designing the engine, the body, and the suspension. By breaking down the problem into smaller problems, we can make it easier to understand and solve.

Abstraction can be used to represent something in a simpler or more general form. For example, we could represent a car as a collection of objects such as an engine, a body, and a suspension. By representing the car in this way, we can make it easier to understand and work with.
In addition to decomposition and abstraction, there are a number of other techniques that we can use to cope with complexity and abstraction. These techniques include:

- **Using metaphors and analogies:** Metaphors and analogies can help us to understand complex concepts by comparing them to something that we are already familiar with.
- **Using visual representations:** Visual representations can help us to understand complex concepts by providing us with a picture of how the concept works.
- **Using natural language:** Natural language can help us to understand complex concepts by providing us with a description of how the concept works.

Here are some specific strategies for coping with complexity and abstraction:

- Break down complex problems into smaller, more manageable pieces.
- Focus on the most important information and ignore the rest.
- Use visual aids to help you to understand complex concepts.
- Ask for help when you need it.
- Don't be afraid to make mistakes.

Let's explore how OOP assists in coping with complexity and abstraction:

1. Encapsulation: Encapsulation is a fundamental principle of OOP that involves bundling data and methods into objects. It allows you to hide the internal complexity of an object and provide a well-defined interface for interacting with it. Encapsulation helps manage complexity by dividing the system into smaller, more manageable units, where each object is responsible for its own behavior and data.

2. Abstraction: Abstraction involves simplifying complex systems by focusing on the essential features while ignoring unnecessary details. In OOP, you can create abstract classes and interfaces that define a common set of properties and behaviors for a group of related objects. By using abstraction, you can create higher-level concepts and models that capture the essence of a system, making it easier to understand and work with.

3. Modularity: Modularity is the practice of breaking down a system into separate, self-contained modules or components. In OOP, each class represents a modular unit that encapsulates a specific set of responsibilities. Modules can be developed and tested independently, which reduces complexity by isolating and managing different parts of the system. Additionally, modular design promotes reusability, as well-designed modules can be easily integrated into other projects.

4. Inheritance: Inheritance is a mechanism in OOP where a class inherits properties and behaviors from another class, known as the superclass or base class. It allows you to create hierarchies of classes, where subclasses inherit and extend the functionality of their parent classes. Inheritance helps manage complexity by promoting code reuse and organizing classes based on their common characteristics, reducing duplication and improving maintainability.

5. Polymorphism: Polymorphism is the ability of objects of different classes to respond to the same method call in different ways. It enables you to write generic code that can operate on objects of multiple types, providing flexibility and extensibility. Polymorphism helps cope with complexity by allowing you to abstract over specific object types and treat them uniformly based on their shared behavior.

6. Design Patterns: Design patterns are reusable solutions to common software design problems. They provide well-established approaches for managing complexity and abstraction. OOP supports various design patterns, such as the Singleton pattern, Observer pattern, and Factory pattern, which help structure and organize code, promote code reuse, and simplify complex interactions between objects.

6. Object Oriented Analysis and Design

Object Oriented Analysis and Design:

Introduction:

Object-Oriented Analysis and Design (OOAD) is a methodology used in software engineering to analyze, design, and model systems using object-oriented principles and concepts. It provides a structured approach to understand, define, and represent the requirements of a system, as well as design the system's architecture and components. OOAD encompasses several techniques and practices to ensure effective software development. Here are the key aspects of Object-Oriented Analysis and Design:

1. **Requirements Analysis:** In this initial phase, the focus is on understanding the system requirements, often through interactions with stakeholders and domain experts. The goal is to identify the functional and non-functional requirements of the system and capture them in a comprehensive manner.
2. **Use Case Modeling:** Use cases are used to describe the system's behavior from the perspective of its users or actors. Use case modeling helps identify the different interactions and scenarios that users have with the system. These use cases are represented graphically and serve as a basis for understanding system functionalities.
3. **Object-Oriented Modeling:** Object-oriented modeling involves creating models that represent the system's structure and behavior using objects, classes, relationships, and interactions. The Unified Modeling Language (UML) is commonly used in OOAD to create visual representations of the system's architecture, such as class diagrams, object diagrams, sequence diagrams, and state diagrams.
4. **Class Design:** Class design focuses on identifying the key classes in the system and defining their attributes (data) and methods (behavior). It involves analyzing the relationships between classes, such as associations, aggregations, and inheritances, and determining the responsibilities and collaborations among classes. Class diagrams and object diagrams are used to document and visualize the class design.
5. **Architecture Design:** Architecture design deals with defining the overall structure and organization of the system. It includes identifying the major components or subsystems, their responsibilities, and the interactions between them. The goal is to create a modular and scalable architecture that promotes flexibility, reusability, and maintainability.

6. **Iterative Development:** OOAD emphasizes an iterative and incremental development approach. It involves breaking down the system into smaller, manageable units and developing and testing them in iterations. Each iteration focuses on a specific set of requirements or functionalities, allowing for feedback and refinement throughout the development process.
7. **Design Patterns:** Design patterns are reusable solutions to common design problems. They provide proven approaches to address recurring design challenges. OOAD leverages design patterns to improve the design quality, promote code reuse, and enhance the maintainability and extensibility of the system.
8. **Object-Oriented Programming:** OOAD lays the foundation for object-oriented programming (OOP). The design models and principles developed during the analysis and design phases serve as a blueprint for implementing the system using an object-oriented programming language. The OOP principles, such as encapsulation, inheritance, and polymorphism, are applied to create the system's classes, objects, and their interactions.

Object-Oriented Analysis and Design helps ensure that software systems are well-designed, modular, and scalable. By following this methodology, developers can effectively analyze system requirements, design the system's structure, and create a flexible and maintainable codebase. OOAD provides a systematic approach to building high-quality software systems using object-oriented principles and best practices.

Some of the *terminologies* that are often encountered while studying **Object-Oriented Concepts** include:

1. **Attributes**: a collection of data values that describe a class.
2. **Class**: encapsulates the data and procedural abstractions required to describe the content and behavior of some real-world entity. In other words, A class is a generalized description that describes the collection of similar objects.
3. **Objects**: instances of a specific class. Objects inherit a class's attributes and operations.
4. **Operations**: also called methods and services, provide a representation of one of the behaviors of the class.
5. **Subclass**: specialization of the super class. A subclass can inherit both attributes and operations from a super class.

6. **Superclass**: also called a base class, is a generalization of a set of classes that are related to it.

Advantages of OOAD:

1. **Improved modularity:** OOAD encourages the creation of small, reusable objects that can be combined to create more complex systems, improving the modularity and maintainability of the software.
2. **Better abstraction:** OOAD provides a high-level, abstract representation of a software system, making it easier to understand and maintain.
3. **Improved communication:** OOAD provides a common vocabulary and methodology for software developers, improving communication and collaboration within teams.
4. **Reusability:** OOAD encourages the reuse of objects and object-oriented design patterns, reducing the amount of code that needs to be written and improving the quality and consistency of the software. OOAD emphasizes the use of reusable components and design patterns, which can save time and effort in software development by reducing the need to create new code from scratch.
5. **Scalability:** OOAD can help developers design software systems that are scalable and can handle changes in user demand and business requirements over time.
6. **Maintainability:** OOAD emphasizes modular design and can help developers create software systems that are easier to maintain and update over time.
7. **Flexibility:** OOAD can help developers design software systems that are flexible and can adapt to changing business requirements over time.
8. **Improved software quality:** OOAD emphasizes the use of encapsulation, inheritance, and polymorphism, which can lead to software systems that are more reliable, secure, and efficient.

Disadvantages of OOAD:

1. **Complexity:** OOAD can add complexity to a software system, as objects and their relationships must be carefully modeled and managed.
2. **Overhead:** OOAD can result in additional overhead, as objects must be instantiated, managed, and interacted with, which can slow down the performance of the software.
3. **Steep learning curve:** OOAD can have a steep learning curve for new software developers, as it requires a strong understanding of OOP concepts and techniques.

4. **Complexity:** OOAD can be complex and may require significant expertise to implement effectively. It may be difficult for novice developers to understand and apply OOAD principles.
5. **Time-consuming:** OOAD can be a time-consuming process that involves significant upfront planning and documentation. This can lead to longer development times and higher costs.
6. **Rigidity:** Once a software system has been designed using OOAD, it can be difficult to make changes without significant time and expense. This can be a disadvantage in rapidly changing environments where new technologies or business requirements may require frequent changes to the system.
7. **Cost:** OOAD can be more expensive than other software engineering methodologies due to the upfront planning and documentation required.

Responsibility Driven Design (RDD):

Responsibility-Driven Design (RDD) is an object-oriented design method that focuses on identifying the responsibilities of objects in a system. RDD is based on the idea that objects should be responsible for a single, well-defined task. This helps to improve the clarity and maintainability of the code.

The key principle of RDD is that software objects should have well-defined responsibilities, and these responsibilities should drive the design and architecture of the system. Each object should be responsible for a specific task or functionality and should encapsulate the data and behavior required to fulfill that responsibility.

In RDD, the design process typically starts by identifying the key responsibilities of the system. Responsibilities are determined by analyzing the requirements and understanding the problem domain. Once the responsibilities are identified, they are assigned to different objects or components within the system.

RDD is a lightweight design method that can be used in conjunction with other design methods, such as Agile Development or Waterfall Development. It is also a good fit for projects that require a high degree of flexibility or adaptability.

The basic steps of RDD are as follows:

1. Identify the system's requirements.
2. Identify the objects in the system.
3. Assign responsibilities to the objects.
4. Define the interactions between the objects.
5. Implement the design.

RDD is a powerful tool that can help to improve the quality of object-oriented software. It is a good choice for projects that require a high degree of clarity, maintainability, and flexibility.

Here are some of the benefits of using RDD:

- **Improved clarity:** RDD helps to improve the clarity of the code by identifying the responsibilities of each object. This makes the code easier to understand and maintain.
- **Improved maintainability:** RDD makes the code easier to maintain by reducing the coupling between objects. This means that changes to one object are less likely to affect other objects.
- **Improved flexibility:** RDD makes the code more flexible by allowing objects to be easily reused. This makes it easier to add new features to the system or to change the system's requirements.

Here are some of the challenges of using RDD:

- RDD can be time-consuming to use.
- RDD requires a good understanding of object-oriented design principles.
- RDD can be difficult to use for complex systems.

Overall, RDD is a powerful tool that can help to improve the quality of object-oriented software. It is a good choice for projects that require a high degree of clarity, maintainability, and flexibility.

Component Responsibility and Collaborator (CRC) Cards:

CRC (Component Responsibility and Collaborator) cards are a brainstorming and design tool used in software development and object-oriented programming to define the responsibilities and interactions of components within a system. They are often used in agile and collaborative environments to facilitate discussions and ensure a clear understanding of the system's design. *It was first introduced by Kent beck and Ward Cunningham. The cards are arranged to show the flow of messages among instances of each class.*

Each CRC card represents a component or class in the system and contains the following information:

1. **Class Name:** The name of the component or class being represented.
2. **Responsibilities:** A list of tasks and functions that the component is responsible for. This includes the behavior and functionality it should exhibit/ show/ reveal/ display.
3. **Collaborators:** Other components or classes that the current component interacts with or depends on to fulfill its responsibilities. These could be other CRC cards or external systems.

The **responsibilities** of a class are the things that the *class knows how to do*. The **collaborators** of a class are the other classes that the *class needs to interact with in order to fulfill its responsibilities*.

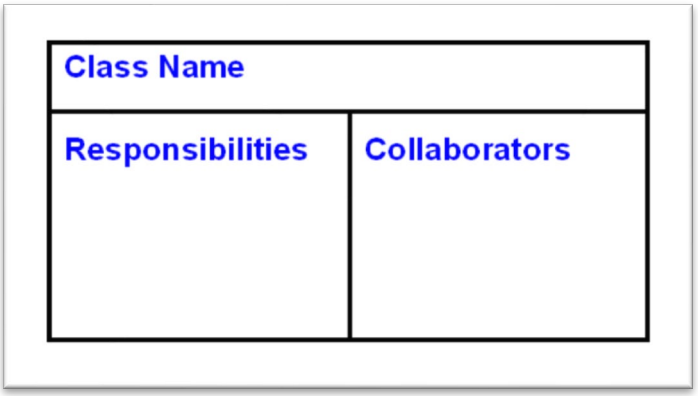
CRC cards can be used to model the design of any object-oriented system. They are a simple and effective way to capture the responsibilities and collaborations of objects. CRC cards can be used to brainstorm new designs, to review existing designs, and to communicate designs to other developers.

Here are some of the benefits of using CRC cards:

- **Increased clarity and understanding of the system:** CRC cards force you to think about the system in terms of its responsibilities, which can help you to gain a better understanding of how the system works.
- **Improved communication:** CRC cards are a great way to communicate designs to other developers. They are a simple and visual way to show the responsibilities and collaborations of objects.
- **Increased flexibility and reusability:** CRC cards can help you to create more flexible and reusable systems. By defining the responsibilities of objects in isolation, you can easily reuse objects in different systems.

Here are some examples of Bank CRC cards:

- Card for a **BankAccount** class:
  - Class name: **BankAccount**
  - Responsibilities:
    - Deposit money
    - Withdraw money
    - Check balance
  - Collaborators:
    - **Bank**
    - **Customer**

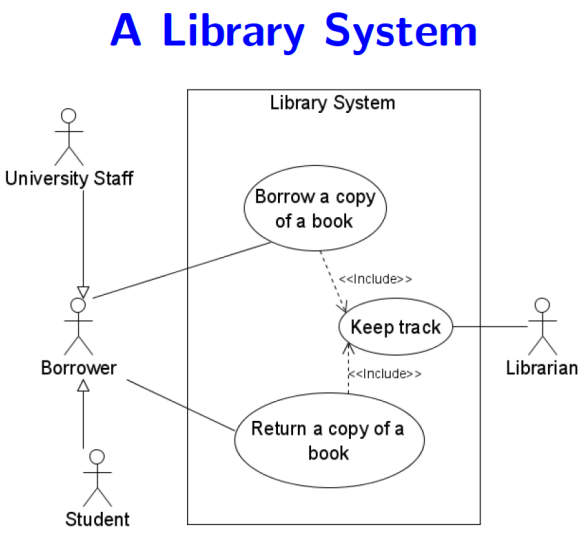


- Card for a **Customer** class:
  - Class name: **Customer**
  - Responsibilities:
    - Open a bank account
    - Close a bank account
    - Deposit money
    - Withdraw money
    - Check balance
  - Collaborators:
    - **BankAccount**
    - **Bank**

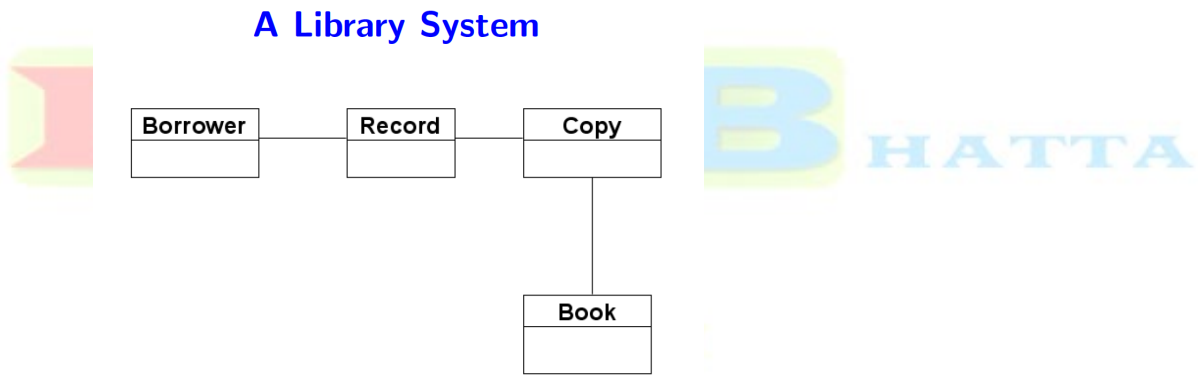
A Library System :: Example

1. The library system must keep track of when books are borrowed and returned
2. The system must support librarian work
3. The library is open to university staff and students
4. University staff can borrow up to 25 different books
5. Students can borrow up to 15 different books

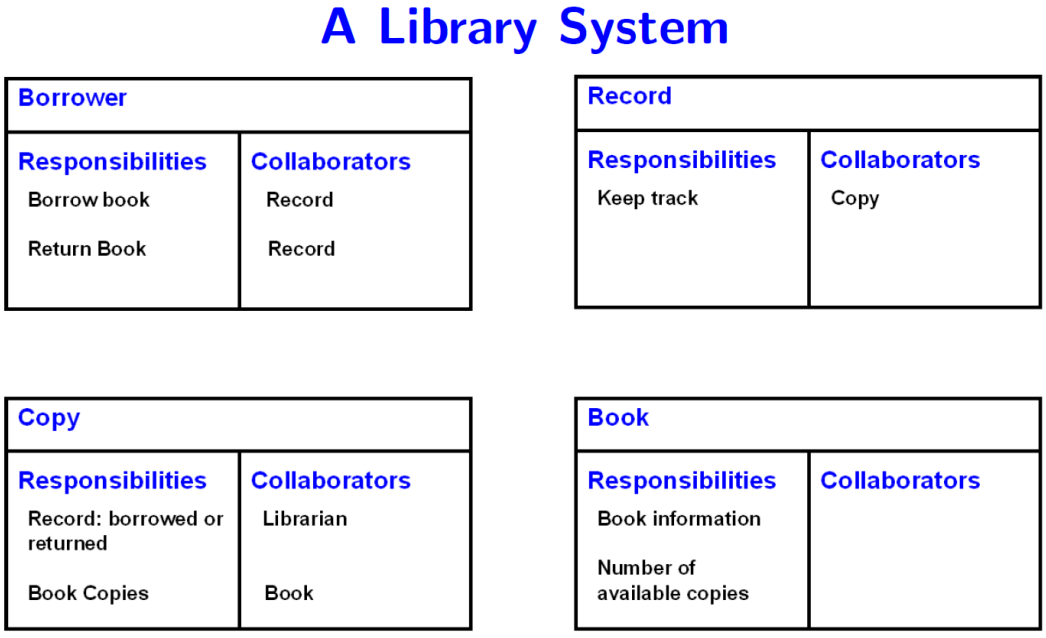
- **Use Case**



- **Class Diagram**



- **CRC Card**



Draw CRC cards of students. [PU:2017 spring]

CRC Card: **Student**

Responsibilities:

- Manage student information (name, ID, contact details, etc.)
- Enroll in courses
- View and manage course registrations
- Submit assignments and coursework
- View grades and academic progress
- Participate in discussions and forums

Collaborators:

- Course: Interacts with the Course component to enroll in courses, view course details, and submit assignments.
- Instructor: Interacts with the Instructor component for receiving instructions, grades, and feedback.
- Registrar: Interacts with the Registrar component to update student information and manage enrollment.
- Discussion Forum: Interacts with the Discussion Forum component to participate in class discussions and post questions.

A class represents a collection of similar objects, a responsibility is something that a class knows or does, and a collaborator is another class that a class interacts with to fulfill its responsibilities.

Let us illustrate these concept with an example of **Student CRC card**

Student	
Student Registration Number Name Address Phone Number Enroll in a Seminar Request Transcript	Seminar Transcript

Seminar	
Seminar Name Seminar Batch Number Ticket Fee Add Student Drop Student Instructor	Student Professor

Professor	
Professor Name Professor ID Address Phone Number Email ID Provides Information Instructing Seminar	Seminar

Transcript	
Student Name Transcript Number Marks Obtained Enrolled Year Calculate Final Grade	Student

Fig: CRC cards for class Student

Responsibility Implies Non-Interference:

The principle of responsibility implies non-interference (RIN) is a design principle in object-oriented programming (OOP) that states that objects should be designed to be independent of each other. This means that objects should not interfere with the internal workings of other objects.

RIN is based on the idea that objects should be designed to be loosely coupled. Loose coupling means that objects should be able to interact with each other without knowing too much about each other's internal details. This makes objects more reusable and easier to maintain.

RIN is a key principle in OOP because it helps to create robust and maintainable software. By designing objects to be independent of each other, we can make our software less susceptible to errors and easier to change.

Here are some examples of how RIN can be applied in OOP:

- When designing a class, we should identify the responsibilities of that class. The responsibilities of a class should be limited to a single area of functionality.
- We should avoid having classes that know too much about each other. If a class needs to interact with another class, it should only know the public interface of that class.
- We should use encapsulation to hide the implementation details of classes. This will make it more difficult for classes to interfere with each other.

RIN is a powerful principle that can help us to create better software. By following the principles of RIN, we can create software that is more robust, maintainable, and reusable.

Here are some additional benefits of RIN:

- **Increased flexibility:** RIN makes it easier to change the behavior of objects without affecting other objects. This is because objects are not tightly coupled to each other.
- **Improved performance:** RIN can improve the performance of software by reducing the amount of communication between objects. This is because objects are not constantly checking on each other's status.
- **Reduced complexity:** RIN can reduce the complexity of software by making it easier to understand how objects interact with each other. This is because objects are not tightly coupled to each other.

Overall, RIN is a valuable principle that can help us to create better software. By following the principles of RIN, we can create software that is more robust, maintainable, reusable, flexible, performant, and less complex.

Differentiate between the concept of computation as simulation and Responsibility implies noninterface. [PU:2017 spring]

Concept	Computation as Simulation	Responsibility implies Noninterface
Definition	Computation viewed as simulating or emulating processes or phenomena	Implication that responsibility excludes direct interaction
Focus	Modeling and simulating behavior of systems or phenomena	Assigning accountability without direct interaction
Purpose	Understanding and studying complex systems	Determining obligations or duties without direct engagement
Application Areas	Physics, biology, economics, computer science	Ethics, organizational management, decision-making
Nature of Interaction	Indirect and simulated interaction with modeled systems	No direct interaction or interface required for accountability
Examples	Simulating weather patterns, modeling molecular interactions	Holding a CEO responsible for company performance without meddling
Key Characteristics	Reproduces behavior, predicts outcomes	Accountability without direct control or influence
Key Considerations	Accuracy of simulation, fidelity to the real system	Clarity of responsibilities, alignment with organizational goals

Programming in Small and Programming in Large:

Programming in the small and programming in the large are two different aspects of writing software. Programming in the small refers to the activity of writing lines of code in a programming language, while programming in the large refers to the activity of designing a larger system as a composition of smaller parts.

Programming in the small is typically concerned with the following tasks:

- Writing code that is correct, efficient, and readable
- Using appropriate data structures and algorithms
- Testing and debugging code

Programming in the large is typically concerned with the following tasks:

- Designing a system architecture
- Splitting the system into modules
- Designing the interfaces between modules
- Developing a test plan

Programming in the small and programming in the large are both important aspects of software development. However, they are different activities, and they require different skills. Programming in the small requires a strong understanding of programming languages and algorithms. Programming in the large requires a strong understanding of system design and architecture.

The terms "*programming in the small*" and "*programming in the large*" were coined by Frank DeRemer and Hans Kron in their 1975 paper "Programming-in-the-large versus programming-in-the-small".

Some of the most important techniques for programming in the large include:

- **Modularity:** dividing the system into smaller, self-contained modules
- **Abstraction:** hiding the implementation details of modules from other modules
- **Encapsulation:** grouping together related data and methods into a single unit
- **Inheritance:** allowing new modules to inherit the properties of existing modules
- **Polymorphism:** allowing modules to be used in different ways, depending on their type

These techniques can help to make large software systems more manageable, easier to understand, and easier to maintain.

THE END