UNIT 1

Object Oriented Concepts

Procedural Oriented Programming:

Conventional Programming, using high level languages such as COBOL (Common Business Oriented Language), FORTAN (Formula Translation) and C, is commonly known as procedural oriented programming (POP). In procedure-oriented approach, the problem is views as a sequence of things to be done such as reading, calculating and printing. POP basically consists of writing a list of instructions (or actions) for the computer to follow, and organizing these instructions into groups known as function. A typical program structure for procedural programming is shown in the figure below. The technique of hierarchical decomposition has been used to specify the task to be completed for solving a problem.

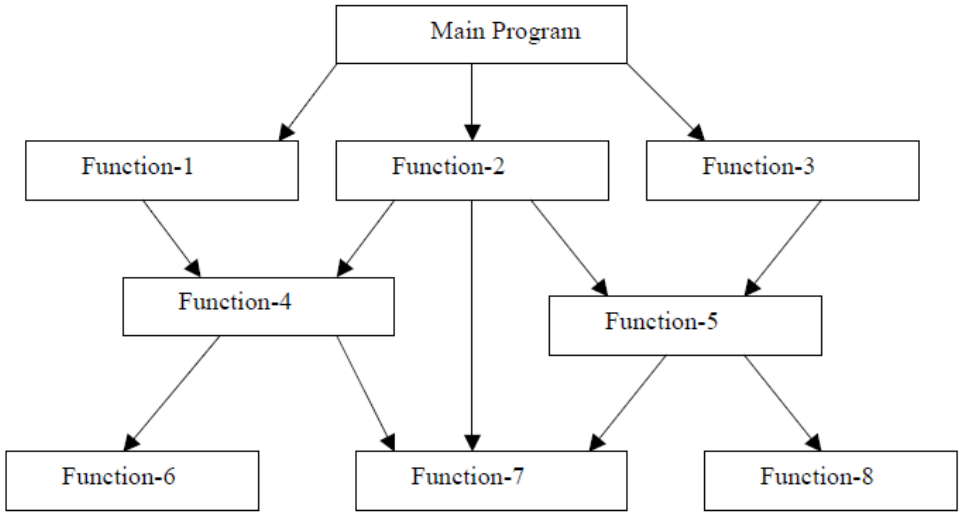
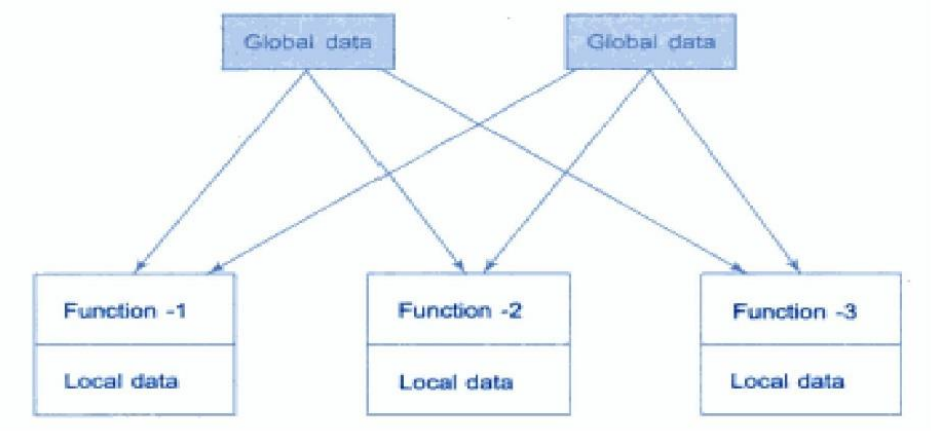


Fig. 1.2 Typical structure of procedural oriented programs

In a multi-function program, many important data items are placed as globally. So that they may be accessed by all the functions. Each function may have its own local data. The figure shown below shows the relationship of data and function in a procedure-oriented program.



Relationship of data and functions in procedural programming

Some features/characteristics of procedure-oriented programming are:

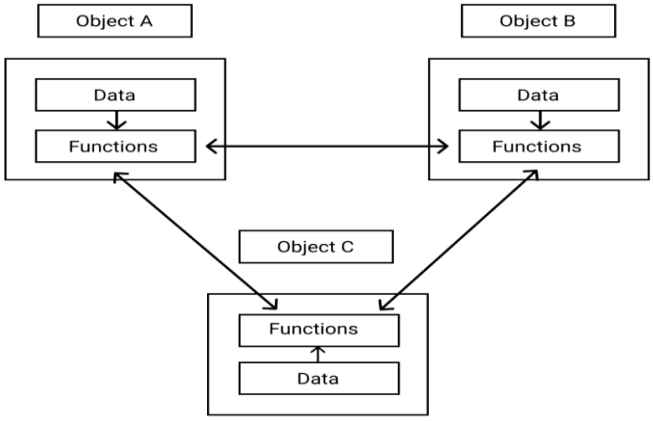
- Emphasis is on doing things (Algorithms).
- Large programs are divided into smaller programs called as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.
- Employs *top-down* approach in program design.

Limitations of Procedural Oriented Programming

- Procedural languages are difficult to relate with the real world objects.
- Procedural codes are very difficult to maintain, if the code grows larger.
- Procedural languages does not have automatic memory management .Hence, it makes the programmer to concern more about the memory management of the program.
- The data, which is used in procedural languages are exposed to the whole program. So, there is no security for the data.
- Creation of new data type is difficult. Different data types like complex numbers and two dimensional co-ordinates cannot be easily represented by POP.

Object oriented programming Paradigm

OOP allows decomposition of a problem into a number of entities called objects and then builds data and functions around these objects. The organization of data and functions in the object-oriented programs shown in the figure:



The data of an object can be accessed only by the function associated with that object.

The fundamental idea behind object-oriented programming is to combine or encapsulate both data (or instance variables) and functions (or methods) that operate on that data into a single unit. This unit is called an object. The data is hidden, so it is safe from accidental alteration. An object's functions typically provide the only way to access its data. In order to access the data in an object, we should know exactly what functions interact with it. No other functions can access the data. Hence OOP focuses on data portion rather than the process of solving the problem.

An object-oriented program typically consists of a number of objects, which communicate with each other by calling one another's functions. This is called sending a message to the object. This kind of relation is provided with the help of communication between two objects and this communication is done through information called message. In addition, object-oriented programming supports encapsulation, abstraction, inheritance, and polymorphism to write programs efficiently. Examples of object-oriented languages include Simula, Smalltalk, C++, Python, C#, Visual Basic .NET and Java etc.

Features of Object oriented programming

Some features of object oriented programming are:
• Emphasis is on data rather than procedure.
• Programs are divided into what are called objects.
• Data structures are designed such that they characterize the objects.
• Functions that operate on the data of an object are tied together in the data structure.
• Data is hidden and cannot be accessed be external functions.
• Objects may communicate with each other through functions.
• New data and functions can be easily added whenever necessary.
• Follows bottom-up approach in program design.

Advantages of Object-oriented Programming

- It is easy to model a real system as programming objects in OOP represents real objects. The objects are processed by their member data and functions. It is easy to analyze the user requirements.
- Elimination of redundant code due to inheritance, that is, we can use the same code in a base class by deriving a new class from it.
- Modularize the programs. Modular programs are easy to develop and can be distributed independently among different programmers.
- In OOP, data can be made private to a class such that only member functions of the class can access the data. This principle of data hiding helps the programmer to build a secure program that cannot be invaded by code in other part of the program.
- With the help of polymorphism, the same function or same operator can be used for different purposes. This helps to manage software complexity easily.
- Large problems can be reduced to smaller and more manageable problems. It is easy to partition the work in a project based on objects.
- Program complexity is low due to distinction of individual objects and their related data and functions.

Disadvantages of Object Oriented Programming

- Sometimes, the relation among the classes become artificial in nature.
- Designing a program in OOP concept is a little bit tricky.
- The programmer should have a proper planning before designing a program using OOP approach.
- Since everything is treated as objects in OOP, the programmers need proper skill such as design skills, programming skills, thinking in terms of objects etc.
- The size of programs developed with OOP is larger than the procedural approach.
- Since larger in size, that means more instruction to be executed, which results in the slower execution of programs.

Applications of Object-oriented Programming

1. Client-Server System
2. Object Oriented Database
3. Real Time Systems Design
4. Simulation and Modeling System
5. Hypertext, Hypermedia
6. Neural Networking and Parallel Programming
7. Decision Support and Office Automation Systems
8. CIM/CAD/CAM Systems
9. AI and Expert Systems

Procedure Oriented vs Object-Oriented Programming

Procedure Oriented Programming	Object oriented Programming
1. Procedural Programming divides the program into small programs and refers to them as functions.	1. Object Oriented Programming divides the program into small parts and refers to them as objects.
2. Focuses on process/logical structure and then data required for that process.	2. Object oriented Programming is designed which focuses on data.
3. Procedure Oriented Programming follows top-down approach.	3. Object oriented Programming follows bottom-up approach.
4. Data and functions don't tie with each other.	4. Data and functions are tied together.
5. Procedure oriented programming is less secure as there is no way of data hiding.	5. Object oriented programming is more secure as having data hiding feature.
6. Procedure Oriented programming can solve moderately complex programs.	6. Object oriented programming can solve any complex programs.
7. Provides less reusability and more function dependency.	7. Provide less function dependency and more reliability.
8. Data moves free around the system from one function to another.	8. Data is hidden and cannot accessed by external functions.
9. Eg.C, pascal ,ALGOL	9. Eg.C++,Java and C# (C sharp)

A way of viewing the world

Describe how object oriented programming models the real world problem with reference of agents, method, behavior and responsibilities. [PU: 2017 fall]

To illustrate the major idea of object oriented programming, Let us consider the real world scenario.

Suppose an individual named Chris wishes to send flowers to a friend named Robin, Who lives in another city. Because of the distance, Chris cannot simply pick the flowers and take them to Robin in person. Nevertheless, it is a task that is easily solved.

Chris simply walks to a nearby flower shop, run by a florist named Fred. Chris will tell Fred the kinds of flowers to send to Robin and the address to which they should be delivered .Chris can then can be assured that the flowers can be delivered expediently and automatically.

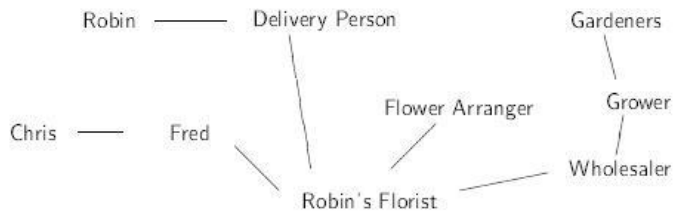
In above example,

- Fred is a **agent (objects)** to deliver flower**.**
- **Message** of wishes with specification of flowers is passed to florist (named Fred) who has **responsibility** to satisfy request.
- Fred uses some **method or algorithm or set of operations** to do this.
- There will community of agents to complete a task. Communication is established to all the concern mediators in a channel to deliver the flowers.
- How the florist satisfies the request is not Chris concern **(data hiding/information hiding).**

Agents and Communities

In above example Chris solved his problem with help of agent (object) Fred to deliver the flower. **There will be community of agents to complete a task. In above example Fred will communicate with Robin's florist**.

An object oriented program is structured as a community of interacting agents called objects. Each object has a role to play. Each object provides a service, or performs an action that is used by other members of community.



The community of agents helping delivery flowers

Messages and methods

Chris request Fred for delivering flower to his friend Robin. This request lead to other requests, which lead to still more requests, until the flowers ultimately reached Chris friend, Robin. We see therefore a members of this community interact with each other making requests.

Action is initiated in object-oriented programming by the transmission of a message to an agent (an object) responsible for the action. The messages encodes the request for an action and is accompanied by any additional information (arguments) needed to carry out the request. The receiver is the object to whom the message is sent. If the receiver accepts the messages, it accepts the responsibility to carry out the indicated action. In response to a message, the receiver will perform some method to satisfy the request.

Message Versus Procedure calls

In message passing

- There is a designated receiver for that message (the receiver is some object to which the message is sent).
- Interpretation of the message is dependent on the receiver and can vary with different receivers.
- There is a late binding between the message (function or procedure name) and the code fragment (method) used to respond the message.

In a procedure call

- There is no designated receiver.
- There is early (compile-time or link time) binding of name to code fragment in conventional procedure calls.

Responsibilities

A fundamental concept in object oriented programming is to describe behavior in term of responsibilities. Chris's Request for action indicates only the desired outcome (flowers send to Robin) .Fred is free to pursue any technique that achieves the desired objective and in doing so will not be hampered by interference from Chris.

Class and instances

In above scenario Fred is florist we can use florist to represent the category (or class) of all florists. Which means Fred is instance (object) of class Florist.
All objects are instances of a class. The method invoked by an object in response to a message is determined by the class of the receiver. All objects of given class use the same method in response to similar messages.

Summary of Object-oriented Concepts

Alan Kay, considered by some be the father of object-oriented programming, identified the following characteristics as fundamental to OOP.

1. Everything is an object.
2. Computation is performed by objects communicating with each other, requesting that other objects perform actions. Objects communicate by sending and receiving messages. A message is a request for action bundled with whatever arguments may be necessary to complete the task.
3. Each object has its own memory, which consists of other objects.
4. Every object is an instance of a class. A class simply represents a grouping of similar objects, such as integers or lists.
5. The class is the repository for behavior associated with an object. That is, all objects that are instances of the same class can perform the same actions.
6. Classes are organized into a singly rooted tree structure, called the inheritance hierarchy. Memory and behavior associated with instances of a class are automatically available to any class associated with a descendant in this tree structure.

Coping with Complexity

At early stages of computing most program were written in assembly language by a single individual.

As program becomes more complex, programmer found it difficult to remember all the information needed to know in order to develop or debug their software. such as

- Which values were contained in what registers?
- Did a new identifier name conflict with any previously defined name?
- What variables needed to be initialized before control could be transferred to another section of code?

The introduction of high level language, such as FORTAN,cobol and Algol solved some difficulties while simultaneously raising people expectations of what a computer could do.

1. **Non-linear Behavior of complexity**

As programming projects became larger, an interesting phenomenon was observed that:
A task that would take one programmer 2 months to perform could not be accomplished by two programmers working for 1 month.

"The bearing of a child takes nine months, no matter how many women are assigned to the task".(refer to Freed Brook's book Mythical Man-Month)

This behavior of complexity is called non-linear behavior of complexity.

The reason for this nonlinear behavior

- The interconnections between software components were complicated
- Large amounts of information had to be communicated among various members of the programming team.

Conventional techniques brings about high degree of interconnectedness.
This means the dependence of one portion of code on another portion (coupling)
So, a complete understanding of what is going on requires a knowledge of both portion of code we are considering and the code that uses it .An individual section of code cannot be understood in isolation.

2. **Abstraction Mechanism**

Abstraction mechanism is use to control complexity.

It is a ability to encapsulate and isolate design and execute information.

Alternatively, Abstraction is a mechanism to displaying only essential information and hiding the details.

Making use of abstraction

a)Procedure and function:

Procedure and function are main two first abstraction mechanism to be widely used in programming language. They allowed task that were executed repeatedly to be collected in one place and reused rather than being duplicated several times. In addition, procedure gave the first possibility for information hiding.

A set of procedure written by one programmer can be used by many other programmers without knowing the exact details of implementation. They needed only the necessary interface.

b)Block Scoping:

Nesting of function (one function inside another) to solve problem associated with procedure and function, the idea of block and scoping was introduced which permits functions to be nested.

This is also not truly the solution. If a data area is shared by two or more procedure, it must still be declared in more general scope than procedure. For example:

```
int sum()
{
```

```
int a,b;
…….
……
int mul()
{
….
……
}
}
```
Partially solved the abstraction mechanism

c) Modules:

A module is basically a mechanism that allows the programmers to encapsulate a collection of procedures and data and supply both import and export statement to control visibility feature from outside modules.

Parnas principle for creation and use of modules:

•One must provide the intended user with all the information needed to use the module correctly and nothing more.

•One must provide the implementor with all the information needed to complete the module and nothing more.

d) Abstract data type(ADT):

- To solve instantiation, the problem of what to do, your application needed more than one instance of software abstraction. The key to solve this problem is ADT.
- ADT is simply programmer defined data type that can be manipulated in a manner similar to system provided data types.
- This supported both information hiding as well as creating many instance of new data type.

But message passing is not possible in ADT.

e) Objects- Messages, Inheritance and Polymorphism:

OOP added new ideas to the concept of ADT.

- **Message passing:** Activity is initiated by a request to a specific object, not by invoking of a function.
- **Inheritance** allows different data types to share the same code, leading to a reduction in code size and an increase in functionality.
- **Polymorphism** allows this shared code to be designed to fit the specific circumstances of individual data types.

Computation As Simulation

- Explain Computation as Simulation?
- In Case of Object oriented Programming, Explain how do we have the view that computation is simulation?[PU:2013 Spring]

OOP framework is different from the traditional conventional behavior of computer. Traditional model can be viewed as the computer is a data manager, following some pattern of instructions, wandering through memory, pulling values out of various memory transforming them in some manner, and pushing the results back into other memory.

The behavior of a computer executing a program is a process-state or pigeon-hole model. By examining the values in the slots, one can determine the state of the machine or the results produced by a computation. This model may be a more or less accurate picture of what takes place inside a computer. Real word problem solving is difficult in the **traditional model.**

In **Object Oriented Mode**l we speak of objects, messages, and responsibility for some action. We never mention memory addresses, variables, assignments, or any of the conventional programming terms. This model is process of creating a host of helpers that forms a community and assists the programmer in the solution of a problem (Like in flower example).

The view of programming as creating a universe is in many ways similar to a style of computer simulation called **"discrete event-driven simulation".**
In, in a discrete event-driven simulation the user creates computer models of the various elements of the simulation, describes how they will interact with one another, and sets them moving. This is almost identical to OOP in which user describes what various entities, object in program are, how will interact with one another and finally set them moving. **Thus in OOP, we have view that computation is simulation.**

Power of metaphor

- When programmer think about problems in terms of behavior and responsibilities of objects, they bring with them a wealth of intuition, ideas and understanding from their everyday experience.
- When envisioned as pigeon holes, mailboxes, or slots containing values, there is a little in the programmers background to provide insight into how should be structured.

Avoiding Infinite Regression

Object cannot always respond to a message by politely asking another object to perform some action. The result would be an infinite circle of request ,like two gentle man each politely waiting for the another to go first before entering a doorway. Thus to avoid infinite regression at a time, at least few objects need to perform some work besides passing request to another agent.

Things to be understand

In the context of object-oriented programming, the view that ***computation is simulation*** *is based on the idea that the objects in the program represent real-world entities or components, and that their interactions and behavior are similar to the interactions and behavior of the real-world entities they represent.*

In object-oriented programming, objects are created to represent real-world entities such as cars, roads, traffic lights, etc. These objects have properties (also known as attributes) that define their state, and methods (also known as functions) that define their behavior. The objects interact with each other by sending messages to each other, which triggers the execution of their methods.

For example, in a simulation of a traffic system, the objects would represent cars, roads, and traffic lights, and their properties would include things like their current position, speed, and color. The methods of these objects would include things like moving, turning, and changing color. The objects would interact with each other by sending messages such as "move forward" or "change color", which would trigger the execution of the appropriate methods.

In this way, the objects in the program represent the real-world entities and their interactions, and can be used to simulate the real-world system. The behavior of the system can be observed by running the program and seeing how the objects interact with each other.

By using OOP we can write code that models the real-world system in a way that is easy to understand and modify. This makes it possible to create simulations that are more accurate and realistic, and that can be used to test different scenarios and make predictions about the behavior of the real-world system.

Additionally, the use of object-oriented programming makes it easy to create reusable and modular code, which makes it easier to create, maintain and extend simulations. Since objects can be reused across different simulations, developers can save time and effort by using existing code rather than having to write new code for each simulation.

In summary, in the context of object-oriented programming, the view that computation is simulation is based on the idea that objects in the program represent real-world entities or components, and that their interactions and behavior are similar to the interactions and behavior of the real-world entities they represent. This allows us to model real-world systems in a way that is easy to understand and modify, and create simulations that are more accurate and realistic.

Features of Object Oriented Programming

Class

It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object. We can create any number of object belonging to class.

Objects

Objects are the basic runtime entities in an object-oriented system. They may represent a person, place, a bank account, a table of data or any item that the program has to handle. Programming problem is analyzed in terms of objects and the nature of communication between them. Program objects should be chosen so that they match closely with real world objects.

Data abstraction

The wrapping up of data and function into single unit(called class) is known as encapsulation. Data encapsulation is the most striking feature of class. The data is not accessible to the outside world and only those function which are wrapped in class can access it.

Encapsulation

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight and cost, and functions to operate on these attributes. They encapsulate all the essential properties of the objects that are to be created.

Inheritance

Inheritance is the process by which objects of one class acquire the properties of another class. Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Example: Dog, Cat, Cow can be Derived Class of Animal Base Class.

Polymorphism

Polymorphism, a Greek term, means the ability to take more than one form. An operation may exhibit different behaviors in different instances .The behavior depends on the types of data uses in the operation.

For example we can use + operator to add numeric values(int or float).If we use the same + operator to adding two object(eg. performing addition of complex number c3=c1+c2) ,which is also known as operator overloading.

Dynamic binding

In dynamic binding, the code to be executed in response to function call is decided at runtime. C++ has virtual functions to support this.

Message passing

Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function and the information to be sent.

Eg. Employee.salary(name);

Object Message Information

Object Oriented Analysis and Design

Introduction:

In dealing with object-oriented technology, Object-Oriented Analysis and Design is the method of choice for the software development life-cycle.
It can be applied in the analysis and design phase and provides general instructions as for what has to be accomplished.
In discussing Object-Oriented Analysis and Design the distinction between these two phases has to be clarified first.

- In the phase of OOA the typical question starts with What...? like “What will my program need to do?”, “What will the classes in my program be?” and “What will each class be responsible for?” . Hence, OOA cares about the real world and how to model this real world without getting into much detail. OOA phase as an investigation of the problem and requirements, rather than finding a solution to the problem.
- In contrast, in the OOD phase, the question typically starts with How...? like “How will this class handle it’s responsibilities?”, “How to ensure that this class knows all the information it needs?” and “How will classes in the design communicate?” . The OOD phase deals with finding a conceptual solution to the problem – it is about fulfilling the requirements, but not about implementing the solution

Responsibility implies non-interference

When we make an object (be it a child or a software system) responsible for specific actions, we expect a certain behavior, at least when the rules are observed. Responsibility implies a degree of independence or noninterference.

If we tell a child that she is responsible for cleaning her room, we do not normally stand over her and watch while that task is being performed-that is not the nature of responsibility. Instead, we expect that, having issued a directive in the correct fashion, the desired outcome will be produced. Similarly, if a system is responsible for managing and maintaining a database, it should be able to do so without interference from other systems.

In case of, conventional programming we tend to actively supervise the child while she's performing a task. In case of OOP we tend to handover to the child responsibility for that performance

Conventional programming proceeds largely by doing something to something else .for example, modifying a record or updating an array. Thus, one portion of code in a software system is often intimately tied by control and data connections to many other sections of the system. Such dependencies can come about through the use of global variables, through use of pointer values or simply through inappropriate use of and dependence on implementation details of other portions of code.

A responsibility-driven design attempts to cut these links, or at least make them as unobtrusive as possible

A real-life example of responsibility and non-interference in programming can be seen in the design of a web application.

Consider a web application that has multiple components such as a user management system, a shopping cart, and a payment gateway. Each of these components has a specific responsibility and should be designed to work independently without interfering with the other components.

For example, the user management system is responsible for handling user registration, login, and profile management. It should be designed to work independently and should not interfere with the functioning of the shopping cart or the payment gateway. Similarly, the shopping cart is responsible for managing the items that a user adds to their cart and should not interfere with the user management system or the payment gateway.

Programming in small and Programming in large

The difference between the development of individual projects and of more sizable software systems is often described as programming in the small versus programming in the large.

Programming in the small characterizes projects with the following attributes:

- Code is developed by a single programmer, or perhaps by a very small collection of programmers. A single individual can understand all aspects of a project, from top to bottom, beginning to end.
- The major problem in the software development process is the design and development of algorithms for dealing with the problem at hand.

Programming in the large, on the other hand, characterizes software projects with features such as the following:

- The software system is developed by a large team, often consisting of people with many different skills. There may be graphic artists, design experts, as well as programmers. Individuals involved in the specification or design of the system may differ from those involved in the coding of individual components, who may differ as well from those involved in the integration of various components in the final product. No single individual can be considered responsible for the entire project, or even necessarily understands all aspects of the project.
- The major problem in the software development process is the management of details and the communication of information between diverse portions of the project.

Component Responsibility Collaborators (CRC Cards)

- **Component Responsibility Collaborators (CRC Cards)** cards are a brainstorming tool used in the design of object-oriented software.
- It was first introduced by Kent beck and Ward Cunningham.

CRC Card is divided into three sections as shown in figure.

<table>
<tr><td colspan="2">Component Name</td></tr>
<tr><td>Description of the Responsibilities assigned to this component</td><td>Collaborators
(List of other components)</td></tr>
</table>

Component: A component is a modular, self-contained unit that encapsulates a specific set of functionality.

Responsibility: Responsibility refers to the specific tasks, actions, or behaviors that a component or object is responsible for.

Collaborator: A collaborator is an object or component that interacts with another object or component to fulfill a particular responsibility or achieve a certain functionality.

Example of **Student CRC card**

Student	
Student number Name Address Phone number Enroll in a seminar Request transcripts	Seminar Transcript

Seminar	
Name Seminar number Fees Add Student Drop student Instructor	Student Professor

Professor	
Name Address Email address Salary Provides information Instructing seminar	Seminar

Transcript	
Student name Marks obtained Enrolled year Calculate final grade	Student

Fig: CRC cards for class student

.

CRC cards for library management system

Librarian	
Know all books Know all borrowers Search for borrower Check in book Check out book	Book Borrower

Book	
Know its title Know its author Know its Registration date Knows due date Knows borrowers Knows if late Knows in or out	 Date Borrower

Borrower	
Knows name Knows contact number Knows set of books Can borrow books	Book

Date	
Knows current date Can compare two dates Can compute new date	

Path follower Robot senses the path it needs to follow through its sensors. Based on the data received through its sensors, the robot make use of its actuators (Robotic Wheels) to steer itself forward. For the above mentioned systems, identify as many components (Collaborating objects) as you can draw CRC card for least three of them .Show the interaction between these components through interaction diagram.[PU:2015 fall 6b]

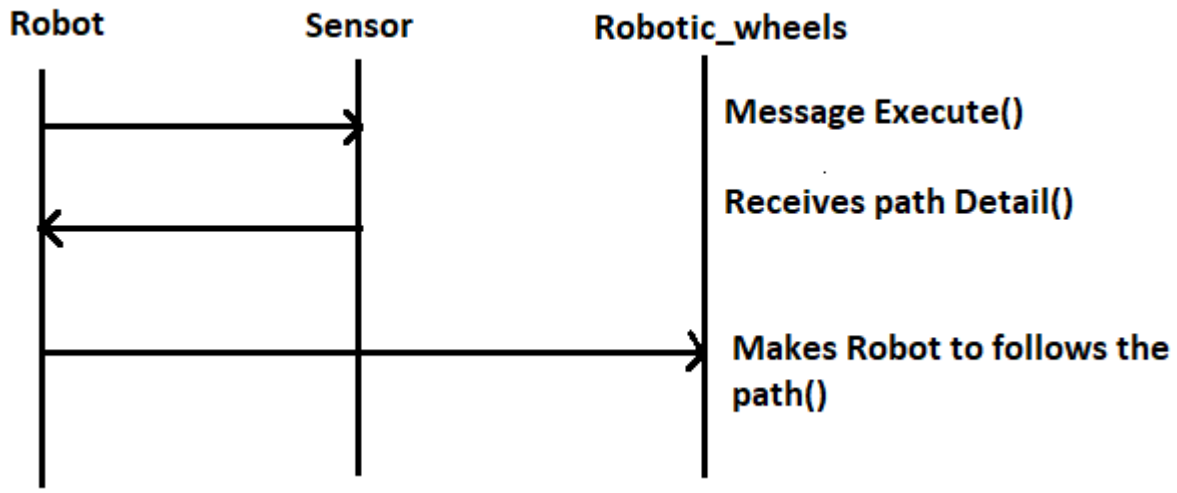
Three CRC cards are given below.

Robot	
Senses the path Follows the sensors Steer itself forward	Sensor Robotic_Wheels

Sensors	
Senses the path Knows which path to go Gives information to Robotic wheels	Robotic_Wheels

Robotic_Wheels	
Makes Robot move Receives information from sensor Follows the path	Robot Sensors

Interaction diagram for above CRC cards



Responsibility-Driven Design (RDD)

Responsibility-Driven Design (RDD), developed by Rebecca Wirfs-Brock, is an object-oriented design technique that is driven by an emphasis on behavior at all levels of development

It is a software design methodology that focuses on identifying the responsibilities of classes and objects in a system, and then designing them to fulfill those responsibilities.

RDD focuses on what action must get accomplished and which object will accomplish them.

The RDD approach focuses on modeling object behavior and identifying patterns of communication between objects.

One objective of object oriented design is first to establish who is responsible for each action to be performed. The design process consists of finding key objects during their role and responsibilities and understanding their pattern of communication. RDD initially focuses on what should be accomplished not how. RDD tries to avoid dealing with details. If any particular action is to happen, someone must be responsible for doing it. No action takes place without an agent.

Some advantages of adopting RDD include:

Increased flexibility and maintainability: By clearly defining the responsibilities of each class and object, RDD makes it easier to modify or extend the system without introducing unexpected side effects.

Improved understandability: RDD makes it easier for developers to understand the system by clearly identifying the responsibilities of each class and object, making it easy for other developers to understand the code and make changes.

Better encapsulation: RDD promotes encapsulation by clearly defining the responsibilities of each class and object, so that the implementation details are hidden and the system is less likely to be affected by changes in other parts of the system.

How are object oriented Programs are designed and developed according to the concept of RDD? Describe entire process in brief.

Case study on RDD

Here we, illustrate the application of Responsibility-Driven Design with a case study.

Imagine you are the chief software architect in a major computer firm. One day your boss walks into your office with an idea that, it is hoped, will be the next major success in your product line. Your assignment is to develop the Interactive *Intelligent Kitchen Helper.*

The task given to your software team is stated in very few words (written on what appears to be the back of a slightly-used dinner napkin, in handwriting that appears to be your boss's).

1. Brief introduction

The Interactive Intelligent Kitchen Helper (IIKH) is a PC-based application that serves as a replacement for traditional index-card recipe systems. It has the ability to assist users in meal planning for an extended period, such as a week. The user can browse the database of recipes and interactively create a series of menus. The IIKH can also automatically adjust the recipes to any number of servings, print out menus for the entire week, for a particular day, or for a particular meal and also print an integrated grocery list of all the items needed for the recipes for the entire period.

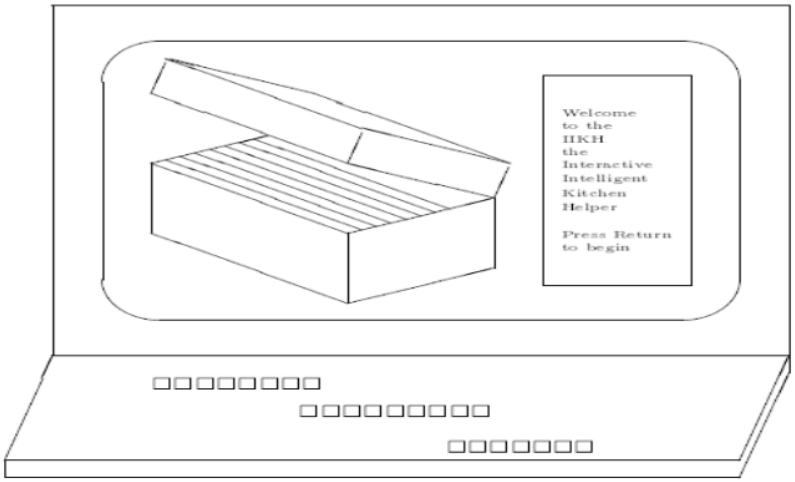


Figure : View of the Interactive Intelligent Kitchen Helper.

2. Working through scenarios

The initial specifications for a software system are often ambiguous and unclear, so the first task is to refine them. During the creation of the software, the specifications may change, so it's important that the design accommodates change easily and potential changes are noted as early as possible. At this stage, high-level decisions can be made concerning the structure of the software system, such as mapping the activities to be performed onto components. To understand the fundamental behavior of the system, the design team creates scenarios by acting out the use of the application as if it already exists.

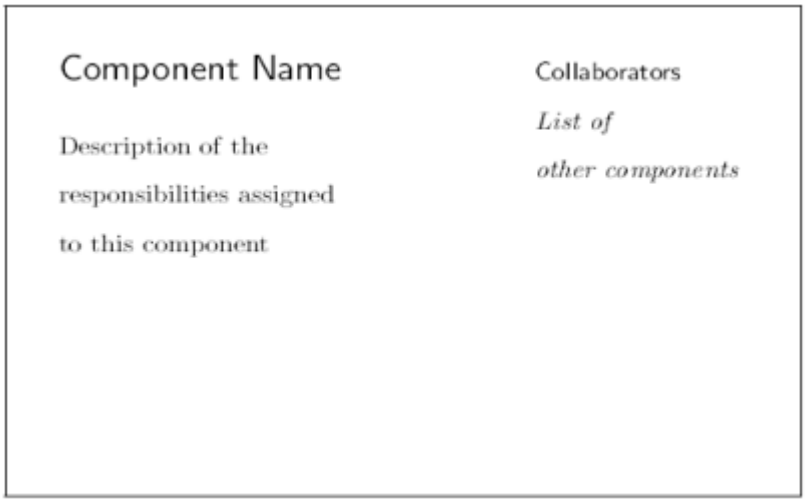
3. Identification of components

A component is simply an abstract entity that can perform tasks-that is, fulfill some responsibilities. At this point, it is not necessary to know exactly the eventual representation for a component or how a component will perform a task. A component may ultimately be turned into a function, a structure or class, or a collection of other components. At this level of development there are just two important characteristics:

- A component must have a small well-defined set of responsibilities.
- A component should interact with other components to the minimal extent possible

4. **CRC cards**

As the design team walks through the various scenarios they have created, they identify the components that will be performing certain tasks. Every activity that must take place is identified and assigned to some component as a responsibility.



As part of this process, it is often useful to represent components using small index cards. Written on the face of the card is the name of the software component, the responsibilities of the component, and the names of other components with which the component must interact. Such cards are sometimes known as CRC (Component, Responsibility, Collaborator) cards, and are associated with each software component. As responsibilities for the component are discovered, they are recorded on the face of the CRC card.

5.1 Give components physical representation

CRC cards are physical index cards that are assigned to different members of the design team, where each card represents a software component and the member holding the card records the responsibilities of the associated software component.

The team member acts as a "surrogate" for the software during the scenario simulation, describing the activities of the software system and passing "control" to another member when the software system requires the services of another component.

The physical separation of the cards encourages an intuitive understanding of the importance of the logical separation of the various components, helping to emphasize the cohesion and coupling.

5.2 What/Who cycle

The identification of components takes place during the process of imagining the execution of a working system. Often this proceeds as a cycle of what/who questions.

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

In this manner, designing a software system is much like organizing a collection of people, such as a club. Any activity that is to be performed must be assigned as a responsibility to some component.

5.3Documentation

At this point the development of documentation should begin. Two documents should be essential parts of any software system: the user manual and the system design documentation. Work on both of these can commence even before the first line of code has been written.

- The user manual describes the interaction with the system from the user's point of view; it is an excellent means of verifying that the development team's conception of the application matches the client's.
- The design documentation records the major decisions made during software design, and should thus be produced when these decisions are fresh in the minds of the creators, and not after the fact when many of the relevant details will have been forgotten.

6. Interaction Diagram

- Used for describing their dynamic interactions during the execution of a scenario.
- In the diagram, time moves forward from the top to the bottom.
- Each component is represented by a labeled vertical line.
- A component sending a message to another component is represented by a horizontal arrow from one line to another.
- Similarly, a component returning control and perhaps a result value back to the caller is represented by an arrow.
- The commentary on the right side of the figure explains more fully the interaction taking place.
- With a time axis, the interaction diagram is able to describe better the sequencing of events during a scenario. For this reason, interaction diagrams can be a useful documentation tool for complex software systems.

Figure shows the beginning of an interaction diagram for the interactive kitchen helper.

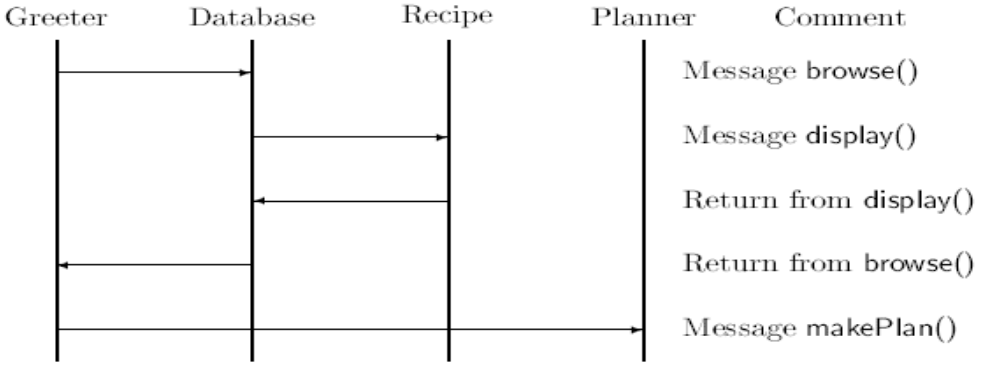


Figure: An Example interaction diagram.

Note:

Some authors use two different arrow forms, such as a solid line to represent message passing and a dashed line to represent returning control.

Sequence diagram is a sub-category of Interaction diagram

7. Software components

In programming and Engineering disciplines, a component is an identifiable part of a larger program or construction. Usually, a component provides a particular function or group of related functions. In programming design,a system is divided into components that in turn are made up of modules. Component test means testing all related modules that form a component as a group to make sure they make together.

In object oriented programming, a component is a reusable program building block that can be combined with other components to form an application. Examples of components include: a single button in graphical interface, a simple interest calculator, an interface to a database manager.

Each component is characterized by:

1) **Behavior and state**

- The behavior of a component is the set of actions it can perform. The complete description of all the behavior for a component is sometimes called the protocol. For the Recipe component this includes activities such as editing the preparation instructions, displaying the recipe on a terminal screen, or printing a copy of the recipe.
- The state of a component represents all the information held within it at a given point of time. For our Recipe component the state includes the ingredients and preparation instructions. Notice that the state is not static and can change over time. For example, by editing a recipe (a behavior) the user can make changes to the preparation instructions (part of the state).

2) **Instances and classes**

- In the real application there will probably be many different recipes. However, all of these recipes will perform in the same manner i.e. the behavior of each recipe is the same.
- But the state (individual list of ingredients and instructions for preparation) differs between individual recipes.
- The term class is used to describe a set of objects with similar behavior.
- An individual representative of a class is known as an instance.
- Note that behavior is associated with a class, not with an individual. That is, all instances of a class will respond to the same instructions and perform in a similar manner.
- On the other hand, state is a property of an individual. We see this in the various instances of the class Recipe. We see this in the various instances of the class Recipe .They can all perform the same actions (editing, displaying, printing) but use different data values.

3) Coupling and cohesion

Cohesion is the degree to which the responsibilities of a single component form a meaningful unit. High cohesion is achieved by associating in a single component tasks that are related in some manner. Probably the most frequent way in which tasks are related is through the necessity to access a common data value. This is the overriding theme that joins, for example, the various responsibilities of the Recipe component.

Coupling, on the other hand, describes the relationship between software components. In general, it is desirable to reduce the amount of coupling as much as possible, since connections between software components inhibit ease of development, modification, or reuse.

4) Interface and Implementation-Parnas's Principles

It is possible for one programmer to know how to use a component developed by another programmer, without needing to know how the component is implemented. The purposeful omission of implementation details behind a simple interface is known as information hiding. The component encapsulates the behavior, showing only how the component can be used, not the detailed actions it performs.

This naturally leads to two different views of a software system. The **interface view** is the face seen by other programmers. It describes what a software component can perform. The **implementation view** is the face seen by the programmer working on a particular component. It describes how a component goes about completing a task.

These ideas were captured by computer scientist David Parna's in a pair of rules, known as **Parnas's principles:**

- The developer of a software component must provide the intended user with all the information needed to make effective use of the services provided by the component, and should provide no other information.
- The developer of a software component must be provided with all the information necessary to carry out the given responsibilities assigned to the component, and should be provided with no other information.

8 Formalizing the Interface

- The first step in this process is to formalize the patterns and channels of communication.
- A decision should be made as to the general structure that will be used to implement each component. A component with only one behavior and no internal state may be made into a function. Components with many tasks are probably more easily implemented as classes. Names are given to each of the responsibilities identified on the CRC card for each component, and these will eventually be mapped onto method names. Along with the names, the types of any arguments to be passed to the function are identified.
- Next, the information maintained within the component itself should be described. All information must be accounted for. If a component requires some data to perform a specific task, the source of the data, either through argument or global value, or maintained internally by the component, must be clearly identified.

7.8.1 Coming up with names

The selection of useful names is extremely important, as names create the vocabulary with which the eventual design will be formulated. Names should be internally consistent, meaningful, preferably short, and evocative in the context of the problem

General Guidelines for choosing names:

- Use pronounceable names. As a rule of thumb, if you cannot read a name out loud, it is not a good one.
- Use capitalization (or underscores) to mark the beginning of a new word within a name, such as “CardReader" or “Card_reader," rather than the less readable “cardreader."
- Abbreviations should not be confusing. Is a “TermProcess" a terminal process, something that terminates processes, or a process associated with a termi- nal?
- Avoid names with several interpretations. Does the empty function tell whether something is empty, or empty the values from the object?
- Avoid digits within a name. They are easy to misread as letters (0 as O, 1 as l, 2 as Z, 5 as S).
- Name functions and variables that yield Boolean values so they describe clearly the interpretation of a true or false value. For example, “Printer-IsReady" clearly indicates that a true value means the printer is working, whereas “PrinterStatus" is much less precise
- Names for operations that are costly and infrequently used should be carefully chosen as this can avoid errors caused by using the wrong function.

Once names have been developed for each activity, the CRC cards for each component are redrawn, with the name and formal arguments of the function used to elicit each behavior identified. An example of a CRC card for the Date is shown in Figure.

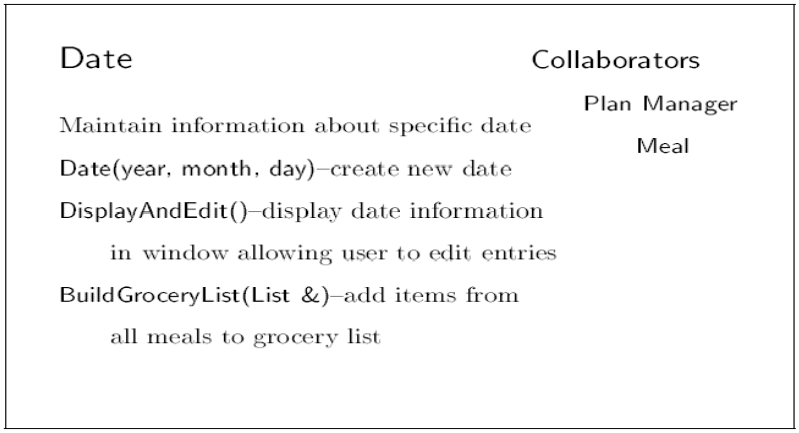


Figure: Revised CRC card for the Date component.

9 .**Designing the Representation**

At this point, the design team can be divided into groups, each responsible for one or more software components. The task now is to transform the description of a component into a software system implementation. The major portion of this process is designing the data structures that will be used by each subsystem to maintain the state information required to fulfill the assigned responsibilities.

Once data structures have been chosen, the code used by a component in the fulfillment of a responsibility is often almost self-evident. A wrong choice can result in complex and inefficient programs.

It is also at this point that descriptions of behavior must be transformed into algorithms. These descriptions should then be matched against the expectations of each component listed as a collaborator, to ensure that expectations are fulfilled and necessary data items are available to carry out each process.

10 .**Implementing components**

- If the previous steps were correctly addressed, each responsibility or behavior will be characterized by a short description. The task at this step is to implement the desired activities in a computer language.
- As one programmer will not work on all aspects of a system, programmer will need to master are understanding how one section of code fits into a larger framework and working well with other members of a team.
- There might be components that work in back ground (facilitators) that needs to be taken into account.
- An important part of analysis and coding at this point is:
    i. Characterizing and documenting the necessary preconditions a software component requires to complete a task.
    ii. Verifying that the software component will perform correctly when presented with legal input values.

11 **Integration of components**

- Once software subsystems have been individually designed and tested, they can be integrated into the final product. This is often not a single step, but part of a larger process. Starting from a simple base, elements are slowly added to the system and tested, using stubs-simple dummy routines with no behavior or with very limited behavior-for the as yet unimplemented parts.
- Testing of an individual component is often referred to as **unit testing**.
- Next, one or the other of the stubs can be replaced by more complete code. Further testing can be performed until it appears that the system is working as desired. (This is sometimes referred to as **integration testing**. The application is finally complete when all stubs have been replaced with working components.
- Testing during integration can involve the discovery of errors, which then results in changes to some of the components. Following these changes the components should be once again tested in isolation before an attempt to reintegrate the software, once more, into the larger system.
- Re-executing previously developed test cases following a change to a software component is sometimes referred to as **regression testing.**

12 Maintenance and Evolution

The term software maintenance describes activities subsequent to the delivery of the initial working version of a software system. A wide variety of activities fall into this category.

- Errors, or bugs, can be discovered in the delivered product. These must be corrected, either in updates or corrections to existing releases or in subsequent releases.
- Requirements may change, perhaps as a result of government regulations or standardization among similar products.
- Hardware may change. For example, the system may be moved to different platforms, or input devices, such as a pen-based system or a pressure-sensitive touch screen, may become available. Output technology may change-for example, from a text-based system to a graphical window-based arrangement.
- User expectations may change. Users may expect greater functionality, lower cost, and easier use. This can occur as a result of competition with similar products. Better documentation may be requested by users. A good design recognizes the inevitability of changes and plans an accommodation for them from the very beginning.

Write short notes on:

Cohesion and coupling

Cohesion refers to the degree to which the elements within a single module or component work together to achieve a single, well-defined purpose. In other words, it measures how closely related the functions within a module are to each other.

Coupling refers to the degree to which one module or component depends on or is connected to another module or component. It measures the strength of the interdependence between two modules or components.

Here is an example to illustrate the concepts of cohesion and coupling:

Suppose you have a module in a software system that is responsible for sending emails. This module has the following functions:

validate_email_address(): Validates the format of an email address.

compose_email(): Composes an email message with a subject and body.

send_email(): Sends the email to the specified recipient.

In this example, the module has high cohesion because all of the functions within it are related to the task of sending emails. There is a clear purpose to the module, and the functions within it work together to achieve that purpose.

On the other hand, suppose that the send_email() function also has a dependency on another module in the system that is responsible for authenticating users. In this case, there is a coupling between the email sending module and the user authentication module because the email sending module depends on the user authentication module to function properly.

Previous board exam questions from this chapter

1) What is object orientation? Explain the difference between structured and Object oriented Programming approach.**[PU:2006 spring]**
2) Why OOP is known as a new paradigm? Illustrate with certain examples. **[PU:2005 fall]**
3) What is class? Explain the different types of classes**.[PU:2005 fall]**
4) Describe Object Oriented Programming as a new paradigm in Computer programming field.**[PU:2015 Spring]**
5) What makes OOP a new paradigm? Explain your answer with suitable points. **[PU:2010 fall]**
6) What influence is an object oriented approach said to have on software system design? What is your own opinion ?Justify through example.**[PU:2009 fall]**
7) Explain the advantages of object oriented paradigm.
8) What are the Critical issues that are to be considered while designing the large Programming? Why? **[PU:2009 spring]**
9) Why Object oriented Programming is Superior than Procedural-Oriented Programming. Explain**.[PU:2016 fall]**
10) What are the main features of Object Oriented Programming. **[PU:2013 fall]**
11) What are the mechanism of data abstraction? Explain the difference between structured and Object Oriented Programming Approach?**[PU:2013 fall]**
12) What are the key features of object oriented programming in C++? Explain with suitable example**.[PU:2020 fall]**
13) What is the significance of forming abstractions while designing an object oriented system? In case of object oriented Programming, Explain how do we have view that computation is simulation? **[PU:2013 spring]**
14) What makes OOP better than POP. Explain with features of OOP. **[PU:2014 fall]**
15) With the help of object oriented Programming, explain how can object oriented Programming cope in solving complex problem. Explain computation as simulation. **[PU: 2014 spring][PU: 2018 fall]**
16) How does making use of abstraction help in designing of an object oriented System. Explain with an example.**[PU:2015 fall]**
17) What is the use of abstraction mechanism in C++?Explain with example.**[PU: 2019 fall]**
18) Describe how object oriented Programming models the real word object problem with reference of agents , method, behavior and responsibilities.**[PU:2017 fall]**
19) What are the shortcoming of procedural Programming? Explain the notation of "Everything is an object" in an object oriented programming. **[PU:2017 spring]**
20) Explain the encapsulation and data abstraction.
21) Software development process in not linear. Justify. Explain abstraction mechanism in technique in C++ with examples.**[PU:2019 fall]**
22) Write a short notes on:
    - Abstraction **[PU:2006 spring]**
    - Non-linear behavior of Complexity **[PU :2014 fall] [PU:2015 spring] [PU:2009 spring]**

23) Explain Responsibility implies non-interface. Explain with example.
24) Explain the terms:

- Responsibility implies non-interference
- Programming in small and Programming in large[PU:2014 fall]

25) What are the difference between programming in small and programming in large?[PU:2010 spring]
26) Write a short notes on:

- CRC cards [PU:2005 fall] [PU:2010 fall][PU:2013 fall][PU:2016 fall]
- Interface and Implementation [PU:2009 fall]
- Programming in small and large [PU:2013 spring]
- Responsibility Driven Design (RDD)[PU:2015 fall][PU:2014 spring][PU:2016 spring]
- Cohesion and coupling

27) Differentiate between **[PU:2010 fall]**
OR
Explain and contrast the following**. [PU:2015 spring]**

- Programming in small and Programming in large
- Interface and implementation

28) What do you mean by responsibility driven design? Also explain what is meant by CRC card.[PU:2010 spring]
29) What do you mean by software component? Explain Integration of components with suitable example scenario to support your answer.[PU:2010 spring]
30) What are the advantages of adopting RDD? Explain with the example of suitable example.[PU:2009 spring]
31) Do you find any advantages of adopting Responsibility Driven Design? Explain with help of suitable example.
32) Explain in brief about interface and implementation. How different components of designed software can be represented and integrated? Discuss in brief.[PU:2013 fall][PU:2017 fall]
33) How are object oriented Programs are designed and developed according to the concept of RDD? Describe entire process in brief.[PU:2013 spring]
34) What do you mean by RDD? What is the use of CRC card?[PU:2014 fall]
14) What is software component? Explain the integration of component with real world example. [PU:2016 fall]
35) What are the different aspects of software components?[PU:2016 spring]
36) Explain in brief about interface and implementation.
37) Draw CRC cards of students.[PU:2017 spring]
38) Explain CRC card and sequence diagram with suitable example.[PU:2019 fall]
39) Differentiate between the concept of computation as simulation and Responsibility implies non-interface. [PU:2017 spring]