Responsibility implies non-interference:

- When we make an object responsible for specific actions, we expect a certain behavior.
- Responsibility implies a degree of independence or noninterference.
- If we tell a younger sister 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 an instruction in the correct fashion, the desired outcome will be produced.
- In case of conventional programming we tend to actively supervise the sister while she's performing a task.
- In case of OOP we tend to handover to the sister the responsibility for that performance.
- In conventional programming, one portion of code in a software system is often bonded to other sections by control or data connections. 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.
- One of the major benefits of object-oriented programming occurs when software subsystems are reused from one project to the next.
- The ability to reuse code implies that the software can have almost no domain specific components; it must assign responsibility for domain-specific behavior to application-specific portions of the system.

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 number 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.

Role of behavior in OOP:

- Earlier software development methodologies concentrated on analyzing the desired application structurally (basic data structures, structure of function calls) or with formal specification.
- But structural elements of the application can be identified only after a considerable amount of problem analysis.
- Similarly, a formal specification often ended up as a document understood by neither programmer nor client.
- But behavior is something that can be described almost from the moment an idea is conceived, and can be described in terms meaningful to both the programmers and the client.
- So, behaviors are helpful for the understanding and early analysis of the problem and enhances the implementation phase.

Responsibility-driven design (RDD):

- Responsibility-Driven Design (RDD) is an object-oriented design technique that is driven by an emphasis on behavior at all levels of development.
- It focuses on: What action is the object responsible for?
    - What information does the object share?
- It focuses on what action must be accomplished and which object will accomplish them.

- It focuses on modeling object behavior and identifying patterns of communication between objects.
- One objective of this design approach 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.
- It initially focuses on what should be accomplished not how.
- It 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.

CRC Cards: CRC stands for Component Responsibility Collaborators cards. These are the index cards used in the design of object oriented software that contains components, their assigned responsibilities and the collaborators.

In the design of object oriented software, the components are identified in the initial stage. A component is an entity that can perform certain tasks and fulfill the assigned responsibilities. A responsibility is something that a component is aware of to do i.e. what action to perform and what information to share.

It is often handy to represent the components using small index cards known as CRC cards. These cards are divided into three sections: software component, the responsibilities of the component, and the collaborators. Collaborators are the names of other components with which the components must interact to fulfill their responsibilities.

Component Name	Collaborators
Description of the responsibilities assigned to this component	*List of other components*

.

Figure: Template of a CRC card

An example of a CRC card is given below:

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

Figure: CRC cards for student

Sequence Diagram/Interaction Diagram: The diagram that is used for describing the dynamic interactions between various components during the execution of a scenario is known as sequence diagram. It is the most commonly used interaction diagram.

Sequence diagram describes how a group of components work together and in what order. It shows the interaction between the components in a sequential order, i.e. the order in which the interactions take place. It represents the flow of messages in the system. It can be a useful documentation tool for complex software systems.

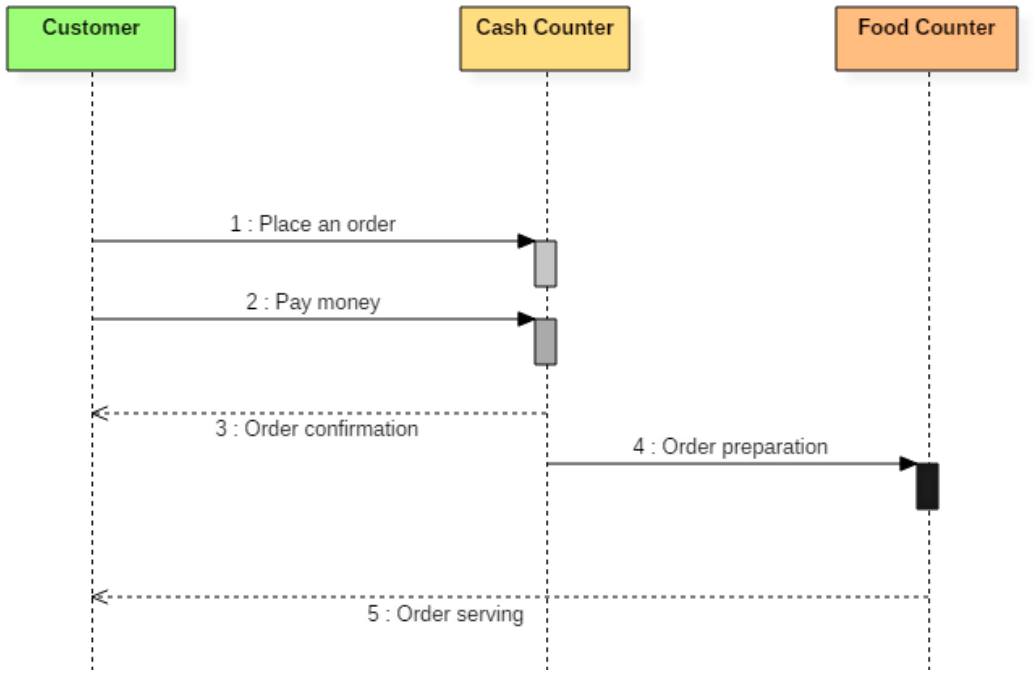

Figure: Sequence diagram of a fast-food restaurant's ordering system

Another example of a sequence diagram that depicts the seat reservation system in a theater is shown below.

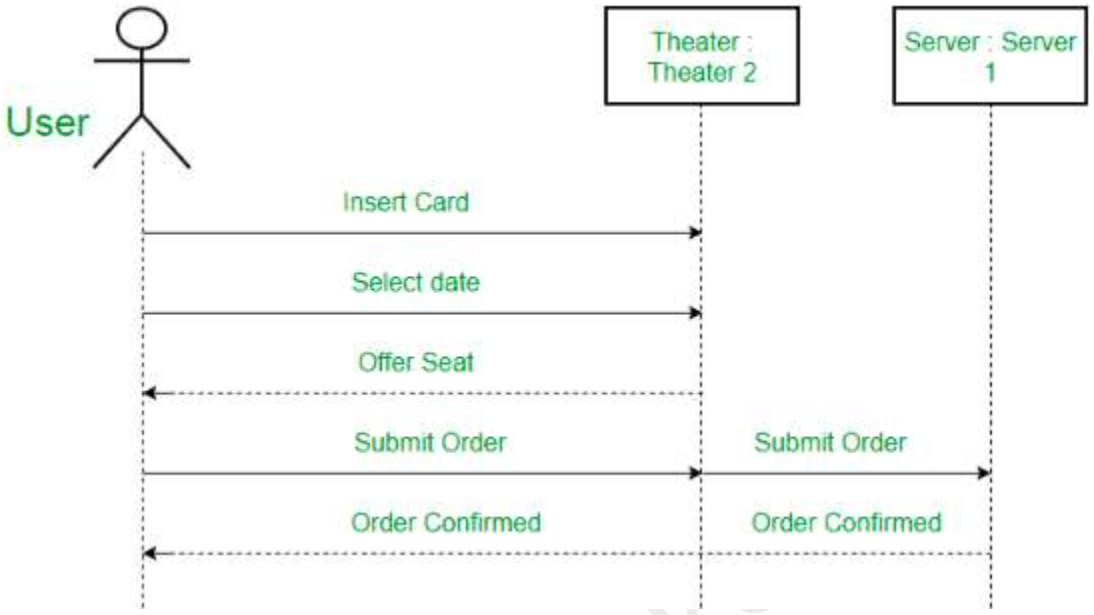

Figure: Sequence diagram of a seat reservation system of a theater

In these diagrams, the time moves forward from 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 (usually solid) horizontal arrow from one line to another. Similarly, a component returning control back to the caller is represented by a (usually dashed) horizontal arrow.

Software components:

A software component is an identifiable part of a larger program or system. Generally, it provides a particular function or a group of related functions.

In a software design, a system is divided into components that in turn are further divided into modules. In object oriented programming, a component is a building block of reusable program that can be combined with other components to form an application.

Examples of components include: a cash counter in a restaurant ordering system, an interface to a database manager etc.

Each component is characterized by following elements:

a) Behavior and state
b) Instances and classes
c) Coupling and Cohesion
d) Interface and Implementation

a) 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.

The state of a component represents all the information held within it at a given point of time. It tells how the component looks or what properties it has.

b) Instances and classes: Instances are the real-world entities. These are also known as the objects of a class. Example of instances could be any entity such as particular book, a student, an employee, etc.

Class is regarded as a template or blueprint of the instances or the objects. Example of a class could be a template that represents a collection of books in a library, or a number of students at a college.

c) Coupling and Cohesion:
Cohesion is used to indicate the degree to which a component has a single, well-focused purpose. It is the degree to which the responsibilities of a single component form a meaningful unit. Highly cohesive components are easier to maintain and less frequently changed. Higher the cohesiveness of the component better is the OO design.

Coupling describes the relationship between the software components. It indicates how the components interact with each other. Usually, it is desirable to reduce the amount of coupling as much as possible. It is because the connections between software components prevent the ease of development, modification or reuse.

d) Interface and Implementation: It is possible for one programmer to know how to use a component developed by another programmer, without knowing how the component is implemented.

For example, a student and seminar component is assigned to a different programmer. But the student needs to know if a spot is available in the seminar or not and if available, he needs to be added to the seminar. So here, the student has to interact with the seminar to get the required information without much knowing the details of it.

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.

Formalizing the interface:

The first step in this process is to formalize the patterns and channels of communication. A decision about how to structure and implement each component has to be made. 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.

And then, the information maintained within the component itself should be described. All the information must be taken into account. 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.

Coming up with names: The selection of useful names for each activity or responsibilities is extremely important, as names create the vocabulary with which the eventual design will be expressed. Names should be consistent, meaningful, preferably short, and suggestive 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 “StudentInfo" or “Student_Info," rather than the less readable “studentinfo."
- Abbreviations should not be confusing.
- Avoid names with several interpretations.
- 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 the functions and variables that yield Boolean values so that 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.

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.

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.

One programmer will not work on all aspects of a system. But the programmer needs to understand how one section of code fits into a larger framework and also how to work well with other members of a team.

An important part of analysis and coding at this point is:

- Characterizing and documenting the necessary preconditions that are required by a software component to complete a task.
- Verifying that the software component will perform correctly when presented with legal input values.

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.

Testing of an individual component is often referred to as unit testing. Further testing can be performed until it appears that the system is working as desired. 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 into the larger system. Re-executing previously developed test cases following a change to a software component is sometimes referred to as regression testing.

Maintenance and evolution: The term software maintenance and evolution describes the activities that need to be considered after the delivery of the initial working version of a software system. It includes a wide variety of activities:

- 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.
- 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.

OLD/MODEL/IMPORTANT QUESTIONS:

1. What is CRC card? Prepare a CRC card for library management system.

2. Explain CRC cards and sequence diagram with examples.

3. Draw a CRC card for student. Explain the significance of CRC card in OO design.

4. Do you find any advantages of adopting responsibility driven design? Explain with the help of suitable example.

Answer: Responsibility-Driven Design (RDD) is an object-oriented design technique that is driven by an emphasis on behaviors at all levels of development. It focuses on: what action is the object responsible for and what information does the object share. It identifies what action must be accomplished and which object will accomplish them.

RDD helps to maximize the abstraction mechanism. It categorizes the responsibilities of objects and initially hides the distinction between data and behavior. It tries to avoid dealing with the details.

Consider a system as a room in which objects live in. When a request is sent to the system for performing some tasks, the objects work together to fulfill the request. The RDD identifies which object is suitable for which task and it is given some responsibility. The RDD design also emphasize on what information

the object will share. So, the objects living in the room are given responsibilities and they do what they are capable of doing. These objects then share the information and delegate the remaining things to its collaborators (other objects/components).

There are many passive concepts from the real world which do not have any behavior. For e.g., a reservation ticket of a movie theatre is a passive concept which becomes active if a real-world client books and pays for it. The responsibilities are needed to make the passive things active and so the responsibilities are assigned to them.

While any system might be complicated, the objects within the system and their associated behaviors shall remain easy to understand. The RDD helps to achieve this and as a result, any updates or changes can easily be made.

In addition to that, by the help of RDD, a low coupling and high cohesion is achieved. Cohesion is used to indicate the degree to which an object has a single, well-focused purpose. Higher the cohesiveness of the object better is the design. Coupling indicates how the objects interact with each other. So, if there is low coupling between objects, it will be easier for modification and reuse and software development becomes better.

5. What is software component? Explain about implementation and integration of component with real world problem.

Or,

How different components of software design can be represented and integrated? Discuss in brief.

Or,

Explain the different steps for developing and implementing software components in object-oriented programming.

6. Write short notes on:

(a) Responsibility implies non-interference
(b) Responsibility driven design (RDD)
(c) Role of behavior in OOP

7. Differentiate between

(a) Interface and implementation
(b) Coupling and cohesion
(c) Programming in small and programming in large