Unit - 2
Object Oriented Programming (B.Tech II-I)

UNIT-2: **Object-Oriented Design-** Responsibility Implies Non interference, Programming in the Small and in the Large, Why Begin with behavior, Case Study in RDD, CRC Cards, Components and Behavior, Software Components, Formalize the Interface, Designing the Representation, Implementing Components, Integration of Components, Maintenance and Evolution.
Classes and Methods- Encapsulation, Class Definitions, Methods, Variations on Class themes.

Object-Oriented Design: object-oriented programming is one that supports inheritance, message passing, virtual and static methods and classes.
Responsibility: an object (be it a child or a software system) responsible for specific actions, you expect a certain behavior, at least when the rules are observed. But just as important, responsibility implies a degree of independence or noninterference.
Eg: Explaining child that he/she is responsible for cleaning their room.(or) Florist how he delivers flowers to receiver.

The **difference between conventional programming** and object-oriented programming is in many ways the difference between actively *supervising a child* while she performs a task, and *delegating to the child responsibility* for that performance. *Conventional programming proceeds largely by doing something to something else modifying a record or updating an array*, for example. 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.
One of the major benefits of object-oriented programming occurs when software subsystems are reused from one project to the next. For example, a simulation manager.

Programming in the Small and in the 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.
  Eg: modules ,backend, frontend ,designers, system engineers.
- The major problem in the software development process is the management of details and the communication of information between diverse portions of the project.

Why Begin with Behavior? :
The simple answer is that the behavior of a system is usually understood long before any other aspect.

Earlier software development methodologies (those popular before the advent of object-oriented techniques) concentrated on ideas such as characterizing the basic data structures or the overall structure of function calls, often within the creation of a formal specification of the desired application. 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 (often unlike a formal specification) can be described in terms meaningful to both the Programmers and the client.
Responsibility-Driven Design (RDD): We will illustrate the application of Responsibility-Driven Design with a case study.

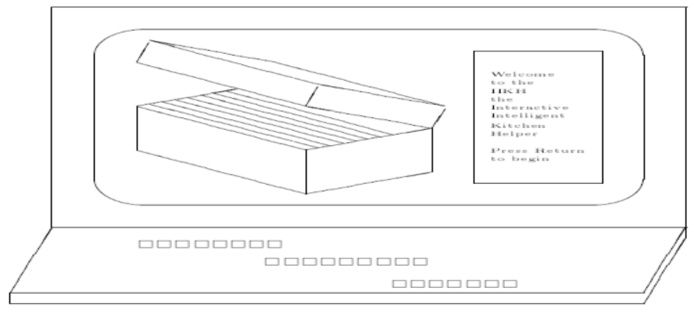


Figure 3.1: – View of the Interactive Intelligent Kitchen Helper.

A Case Study in RDD:
Imagine you are the chief software architect in a major computer rm. One day your boss walks into your oce 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.
Example :

- Amazon Alexa.
- Bixby.
- Cortana.
- DataBot.
- Google Assistant.
- Hound.
- Lyra Virtual Assistnat.
- Robin. Price.

The Interactive Intelligent Kitchen Helper:
Briefly, the Interactive Intelligent Kitchen Helper (IIKH) is a PC-based application that will replace the index-card system of recipes found in the average kitchen. But more than simply maintaining a database of recipes, the kitchen helper assists in the planning of meals for an extended period, say a week. The user of the IIKH can sit down at a terminal, browse the database of recipes, and interactively create a series of menus. The IIKH will automatically scale the recipes to any number of servings and will print out menus for the entire week, for a particular day, or for a particular meal. And it will print an integrated grocery list of all the items needed for the recipes for the entire period.

The fundamental cornerstone of object-oriented programming is to characterize software in terms of behavior; that is, actions to be performed. Initially, the team will try to characterize, at a very high level of abstraction, the behavior of the entire application. This then leads to a description of the behavior of various software subsystems. Only when all behavior has been identified and Described will the software design team proceed to the coding step.

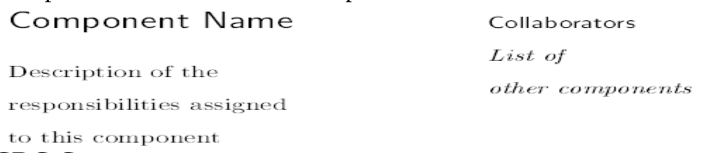
Working through Scenarios:
The first task is to refine the specification. As we have already noted, initial specification are almost always ambiguous and unclear on anything except the most general points. There are several goals for this step. One objective is to get a better handle on the \look and feel" of the eventual product. This information can then be carried back to the client (in this case, your boss) to see if it is in agreement with the original conception.

In order to uncover the fundamental behavior of the system, the design team first creates a number of scenarios. That is, the team acts out the running of the application just as if it already possessed a working system.

Identification of Components:
A component is simply an abstract entity that can perform tasks that is, full some responsibilities.
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.

CRC Cards Recording Responsibility:
CRC (Component, Responsibility, Collaborator) 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.



Component Name

Description of the
responsibilities assigned
to this component

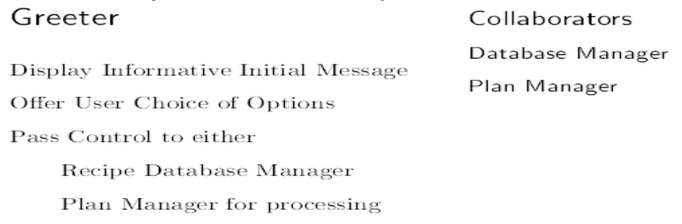
Collaborators
List of
other components

CRC Components:
While working through scenarios, it is useful to assign CRC cards to different members of the design team. The member holding the card representing a component records the responsibilities of the associated software component, and acts as the \surrogate" for the software during the scenario simulation. He or she describes the activities of the software system, passing \control" to another member when the software system requires the services of another component.
EG: ATM Card

The 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.
Eg: Stickers on Number plate

Documentation and user manuals and code content:
CRC cards are one aspect of the design documentation, but many other important decisions are not reflected in them. Arguments for and against any major design alternatives should be recorded, as well as factors that influenced the final decisions. A log or diary of the project schedule should be maintained. Both the user manual and the design documents are refined and evolve over time in exactly the same way the software is refined and evolves.

Components and Behavior:
In IIKH when the system begins, the user will be presented with an attractive informative window (see Figure 3.1). The responsibility for displaying this window is assigned to a component called the Greeter. In some as yet unspecified manner (perhaps by pull-down menus, button or key presses, or use of a pressure-sensitive screen), the user can select one of several actions. Initially, the team identifies just five actions:



Greeter

Display Informative Initial Message
Offer User Choice of Options
Pass Control to either
Recipe Database Manager
Plan Manager for processing

Collaborators
Database Manager
Plan Manager

1. Casually browse the database of existing recipes, but without reference to any particular meal plan.
2. Add a new recipe to the database.
3. Edit or annotate an existing recipe.
4. Review an existing plan for several meals.
5. Create a new plan of meals.

Postponing Decisions:
There are a number of decisions that must eventually be made concerning how best to let the user browse the database. For example, should the user first be presented with a list of categories, such as \soups," \salads," \main meals," and \desserts"? (\Almonds, Strawberries, Cheese"), or a list of previously inserted keywords (\Bob's favorite cake")?

Preparing for Change:
Eventhough one tries to develop the initial specification carefully and design of a software system, it is almost certain that changes in the user's needs or requirements will, sometime during the life of the system, force changes to be made in the software. Programmers and software designers need to anticipate this and plan accordingly.

- The primary objective is that changes should affect as few components as possible. Even major changes in the appearance or functioning of an application should be possible with alterations to only one or two sections of code.
- Try to predict the most likely sources of change and isolate the effects of such changes to as few software components as possible. The most likely sources of change are interfaces, communication formats, and output formats.
- Try to isolate and reduce the dependency of software on hardware. For example, the interface for recipe browsing in our application may depend in part on the hardware on which the system is running. Future releases may be ported to different platforms. A good design will anticipate this change.
- Reducing coupling between software components will reduce the dependence of one upon another, and increase the likelihood that one can be changed with minimal effect on the other.
- In the design documentation maintain careful records of the design process and the discussions surrounding all major decisions. It is almost certain that the individuals responsible for maintaining the software and designing future releases will be at least partially different from the team producing the initial release. The design documentation will allow future teams to know the important factors behind a decision and help them avoid spending time discussing issues that have already been resolved.

Continuing the Scenario: Explanation of Diagram

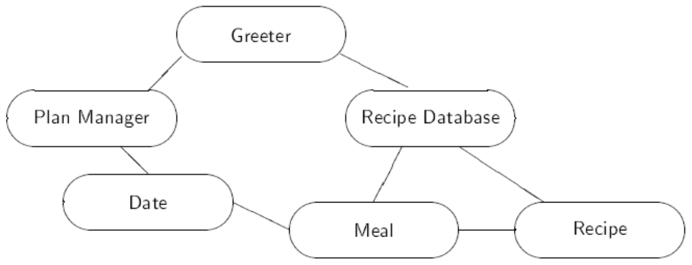


Figure 3.4: – Communication between the six components in the IIKH.

Interaction Diagrams:
Interaction Diagrams are A better tool for describing their dynamic interactions during the execution of a scenario.

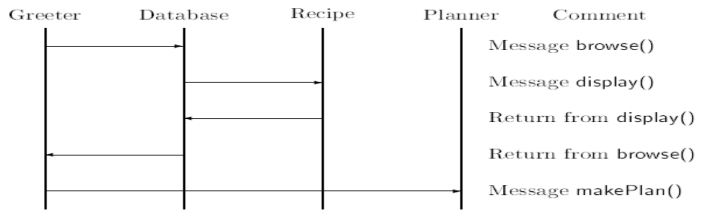


Figure 3.5: – An Example interaction diagram.

The current UML standards call for 13 different types of diagrams: class, activity, object, use case, sequence, package, state, component, communication, composite structure, interaction overview, timing, and deployment.

<u>Software Components:</u>
Software components are parts of a system or application. Components are a means of breaking the complexity of software into manageable parts.

Behavior and State:
components may also hold certain information. Let us take as our prototypical component a Recipe structure from the IIKH. One way to view such a component is as a pair consisting of 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).

Instances and Classes:
A class is a blueprint/plan/template which you use to create objects. An object is an instance of a class used to access data members and methods.
Let us take as our prototypical component a Recipe structure from the IIKH.
the behavior of each recipe is the same; it is only the state the individual lists of ingredients and instructions for preparation that differs between individual recipes. In the early stages of development our interest is in characterizing the behavior common to all recipes; the details particular to any one recipe are unimportant.
On the other hand, state is a property of an individual. 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.

Coupling and Cohesion:
Coupling refers to the interdependencies between modules, while cohesion is a degree (quality) to which a component / module focuses on the single thing, describes how related the functions within a single module are.
EG: the Recipe Database agent would need the ability to directly manipulate the state (the internal data values representing the list of ingredients and the preparation instructions) of an individual recipe.
It is better to avoid this tight connection by moving the responsibility for editing to the recipe itself.

Interface and Implementation Parnas's Principles:
An Interface defines a set of method signatures, while a Class defines the structure and behavior of its instances. A class may match several interfaces, if it implements the methods defined by each interface. Classes are instantiable, while interfaces are not.
The purposeful omission of implementation details behind a simple interface is known as information hiding.

Computer scientist David Parnas 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.

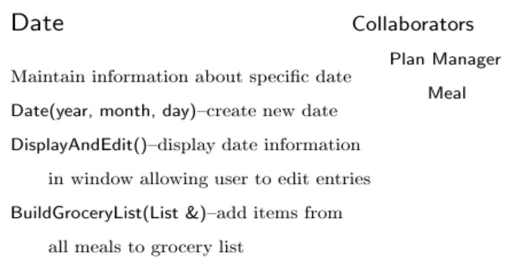
A consequence of the separation of interface from implementation is that a programmer can experiment with several different implementations of the same structure without affecting other software components.

Formalize the Interface:
IIKH as example: The first step in this process is to formalize the patterns and channels of communication.
A component with only one behavior and no internal
state may be made into a function for example, a component that simply takes a string of text and translates all capital letters to lowercase.
Components with many tasks are probably more easily implemented as classes. 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 following general guidelines have been suggested:

- 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."
- Examine abbreviations carefully. An abbreviation that is clear to one person may be confusing to the next. Is a \TermProcess" a terminal process, something that terminates processes, or a process associated with a terminal?
- 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.
- Take extra care in the selection of names for operations that are costly and infrequently used. By doing so, errors caused by using the wrong function can be avoided.

Designing the Representation:
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.


Date

Collaborators
Plan Manager
Meal

Maintain information about specific date
Date(year, month, day)–create new date
DisplayAndEdit()–display date information
in window allowing user to edit entries
BuildGroceryList(List &)–add items from
all meals to grocery list

Integration of Components:
The physical components consist of the various machine systems, computer hardware, inventory, etc. The virtual components consists of data stored in databases, software and applications. The process pf integrating all these components, so that act like a single system, is the main focus of system integration.
For example: in the development of the IIKH.

- Testing of an individual component is often referred to as unit testing.
- INTEGRATION TESTING is a level of software testing where individual units are combined and tested as a group.
- Regression Testing is defined as a type of software testing to confirm that a recent program or code change has not adversely affected existing features. Regression Testing is nothing but a full or partial selection of already executed test cases which are reexecuted to ensure existing functionalities work fine.

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 are:

- 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.