# Chapter 2

# 2 Object-Oriented Design:

People often tend to compare the **syntactic features** of *C++ , Java* with non object oriented versions *C ,Pascal* in response to questions regarding Object Oriented Programming

Then about *classes, inheritance , message passing* etc

They are overlooking the crucial point of *Object Oriented Programming*

Object Oriented Programming has **nothing to do with *syntax***

The most important aspect of OOP is a **design technique driven by the determination and delegation of responsibilities.**

This technique is called ***Responsibility Driven Design (RDD).***

## 2.1 Responsibility Implies Noninterference:

When we make an object (*be it a child or software*) responsible for **specific actions** we expect a **certain behavior** e.g. a light bulb glowing (behavior) when it is switched on (action)

Responsibility implies *a degree of independence or non-interference,* i.e. an object is fully responsible for doing certain thing that it has taken as its responsibility. e.g. a light bulb object has the responsibility of glowing and it can do it on its own

When we assign a child the responsibility of cleaning her room, we expect that the desired outcome will be produced (*a clean room*)

Normally, we do not interfere nor do we stand over her and watch while that task is being performed

We leave it up to her to apply the method she desires to clean her room (use a vacuum cleaner, broom)

Conventional programming proceeds largely by doing something to something else e.g. modifying a record or updating an array

Here one portion of code in a software system is often intimately tied by control and data connections to many other sections of the system due to the use of global variables , use of pointers etc

A responsibility driven design attempts to cut these links

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

## 2.2 Programming in the Small and Programming in the Large:

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

**Programming in small** characterizes projects with the following attributes:

Code is developed by a **single programmer** or by a very small collection of programmers

A single individual can **understand all aspects of a project** form top to bottom, beginning to end

Major problem in the software development process is the design and development of algorithms for dealing with the problem at hand

**Programming in large** characterizes projects with the following attributes:

Software system is developed by a **large team of programmers**

Different team members involved in

- Specification or **design** of a system

- **Coding** of individual components
- **Integration** of various components in the final product

No single individual can be considered responsible for the entire project

No single individual understands all aspects of the project

Major problem in the software development process is the **management of detail** and the **communication of information** between diverse portions of the project

## 2.3 Role of Behavior in OOP:

It is better to start the design process with an **analysis of behavior** as the behavior of a system is usually understood long before any other aspect

Earlier software development techniques concentrated on ideas such as characterizing the basic data structures or the overall structure of function calls 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 can be described almost from the moment an idea is conceived to both the programmers and the client** e.g. player throws dice, **dice rolls** and returns a value and **result** is **declared** based on the comparison of the face value of the dice

In case of OOP, the design is based on responsibility (*Responsibility Driven Design*)

## 2.4 Case Study: *Responsibility-Driven Design*

### 2.4.1 Objective: Develop an Interactive Intelligent Kitchen Helper (*IIKH*)

### 2.4.2 Brief Description:

A PC based application to be designed to replace the index card system of recipes in the kitchen

Maintains a database of recipes and assists in the planning of meals for an extended period

User sits down at a terminal browses the database of recipes and interactively create a series of menus

The IIKH should automatically scale the recipes to any number of servings and print out menus for the entire week, for a particular day, or for a particular meal

Print out an integrated grocery list of all items needed for the recipes for the entire period

As with most software systems, initial descriptions of IIKH are highly ambiguous

The project will require the efforts of several programmers working together

The initial goal of the team must be to clarify the ambiguities in the description and to outline how the project can be divided into components to be assigned for development to individual team members

The fundamental cornerstone of Object-Oriented Programming is to characterize software in terms of behavior

Initially, the team will try to characterize the behavior of the entire application at a very high level of abstraction

This then leads to a description of the behavior of various software subsystems

The software design team proceeds to the coding step only when all behavior has been identified and described

### 2.4.3 Identification of Components:

Just as in case of generic engineering where simplification is achieved by dividing the design into smaller units

Similarly the engineering of software is simplified by the identification and development of software components

A component is simply **an abstract entity that can perform tasks** or fulfill responsibilities

A component may ultimately be turned into a function, a structure or class

#### Characteristics of components

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

## 2.5 CRC Cards:

A CRC card is an **object oriented design method** that uses ordinary 3x5 index cards. It was developed by Ward Cunningham at Textronix, a card is made for each class containing responsibilities (knowledge and services) and collaborators (interaction with other objects)

The first step of system development is to **gather user requirements.**

We can't build a system if we don't know what it should do.

A CRC modeling provides a simple and effective technique for working with the users to determine their needs.

The team **discovers the components** and **their responsibilities by walking thru the scenario**

Every activity that must take place is identified and assigned to some component as a responsibility

Components are represented using small index cards with the name of the software component, its responsibilities and the names of other components it interacts with, written on the face.

These cards are known as CRC Cards (*Component, Responsibility, Collaborator*) and associated with each software component

| **Component Name** | |
|---|---|
| **Description of responsibilities assigned to this component.** | **Collaborators**<br><br>**List of other Components** |

#### Collaborators:

Sometimes classes do not have enough information to fulfill their responsibilities, so they need to **work (*collaborate*) with other classes** to get the job done

A collaboration will be in either of the two forms:

- A request for information
- A request to perform a task

Class B is listed as a collaborator of Class A if and only if B does something to A

During scenario simulation, CRC cards can be used by assigning them to the members of the design team to give physical representation of the components.

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

A component might be assigned more tasks than it can handle, in such case some responsibilities can be shared with other new components

### 2.5.1 The What/Who Cycle:

The identification of components takes place during the process of imagining the execution of a working system

This initiates the cycle of *what/who* questions

The programming team identifies **what activity** needs to be performed next

Then identifies **who** performs the action, **BUT NOT HOW, YET!!**

As in real world scenario, some component must be assigned a responsibility to perform an activity

The secret to good object oriented design is to first establish an agent for each action

### 2.5.2 Documentation:

At this point the development of documentation should begin.

Two types of documents as an essential part of any software system

i. **The User Manual**

Describes he interaction with the system from the user's point of view

It should be developed before any actual code has been written, the mindset of the team is similar to the eventual users at this point in time

ii. **The System Design Documentation**

Records the major decisions made during software design and should be produced when these decisions are fresh in the minds of the creators

Gives the initial picture of the larger structure hence should be carried out early in the development cycle

CRC cards are one aspect of the design documentation but not many other important decisions are not reflected in them

## 2.6 Components and Behavior:

### 2.6.1 Characteristics, behavior and responsibilities of Components identified for the IIKH System:

#### a. Greeter

The IIKH team decides that **system greets the user with an attractive informative window**.
The **responsibility for displaying this window** is assigned to a component called the *Greeter*

The user can select one of the five actions as designed and identified by the team:

i. Casually browse the database of existing recipes without reference to any particular meal plan

ii. Add a new recipe to the data base

iii. Edit or annotate an existing recipe

iv. Review an existing plan for several means

v. Create a new plan of meals

| **Greeter** | |
|---|---|
| **Display Informative Initial Message**<br>**Offer User Choice of Options**<br>**Pass Control to either:**<br>**Recipe Database Manager**<br>**Plan Manager**<br>**for processing** | **Collaborators**<br>**Database Manager**<br>**Plan Manager** |

The whole activities can be divided into two groups.

i. First three as being associated with the recipe database

ii. The latter two are associated with the menu plans

The team decides to create components corresponding to these two responsibilities

### b. Recipe Component

- Each recipe will be identified with a specific recipe component.
- Once a recipe is selected control is passed to the associated recipe object
- It consists of a list of ingredients and steps needed to transform the ingredients into the final product
- It should display the recipe interactively on the screen
- It should enable the user to modify the instructions or list of ingredients
- It gives printed copy of the recipe

### c. Recipe Database Manager

In response to the user's desire to add a new recipe the database manager has to decide in which category to place the new recipe

It is done by requesting the name of the new recipe and then creating a new recipe component permitting the user to edit the new blank entry, a subset of tasks already identified for allowing the user to exit existing recipes

These task handle the browsing and creation of new recipes

Now we investigate the development of daily menu plans which is the plan manager's task

### d. Plan Manager

Can either be started by retrieving an already developed plan or by creating a new plan (use can specify the date)

Each date is associated with a date component

**Print out the recipe for the planning period, produce grocery list** for the period

### e. Date Component

**Maintain a collection of meals as well as any other remarks provided by the user (anniversary, birthday)**

**Out puts information concerning the specific date**

Allows the user to print all the information concerning a specific date find out more about a specific meal (control is passed to the meal component in this case)

### f. Meal Component

**Displays information about the meal**

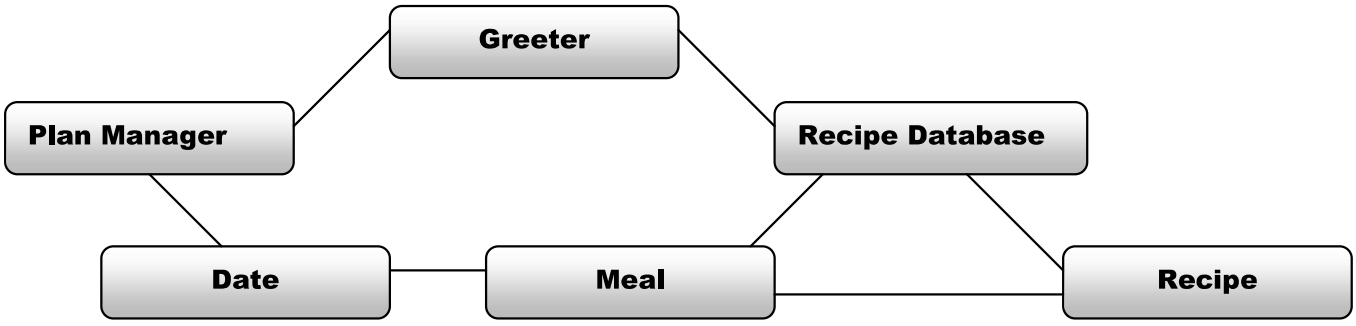
Maintains a collection of augmented recipes (user's desire to double , triple or increase a recipe)

The user can add or remove recipes from the meal and print meal information

It must interact with database component to allow user to discover new recipes by browsing the recipe database

In this manner the design team investigate every possible scenario

Exceptional cases like no matching recipe found for the user input, user wants to cancel an activity etc must be handled by assigning the task to some component.




Communication Between the Six Components in the IIKH

The team analyzed that all activities can be adequately handled by six components

- The *Greeter* needs to communicate only with the *Plan Manager* and the *Recipe Database* component
- The *Plan Manager* needs to communicate only with the *Date* Component
- The *Date Agent* needs to communicate only with the *Meal* Component
- The *Meal* component communicates with individual recipes thru the *Recipe* Manager

### 2.6.2 Preparing for a Change:

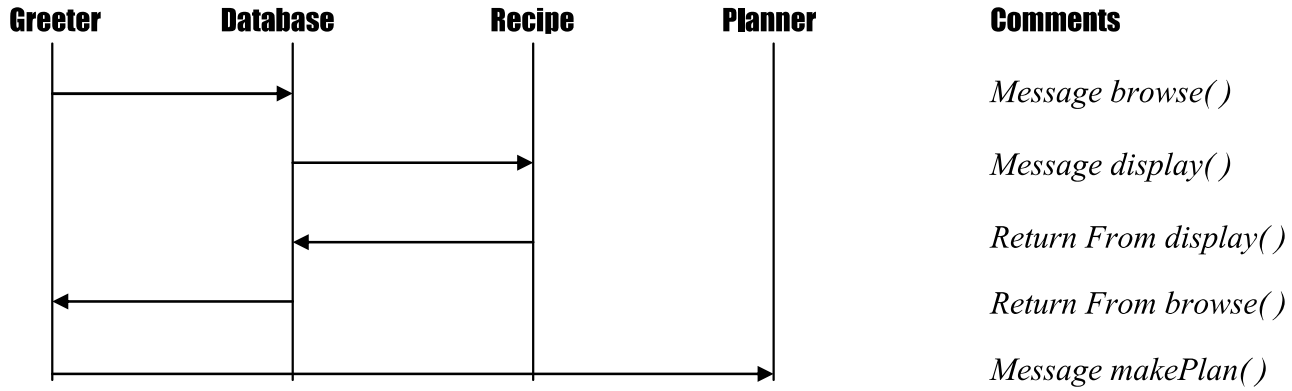
Changes in user's needs or requirements will force changes to be made in the software

Programmers need to anticipate this and plan accordingly

- Changes should affect as few components as possible
- Predict most likely sources of change and isolate the effects of such changes to as few components as possible
- Reduce coupling between software components (*will reduce the dependency of one upon another*)
- Isolate and reduce the dependency of software on hardware
- In design documentation, maintain careful records of the design process and discussions on decisions

### 2.6.3 Interaction Diagram

Used for describing the **dynamic interaction between components** during the execution of a scenario




Here, time moves forward from the top to bottom.

Each component is represented by a labeled vertical line.

A component sending a message to another is represented by a horizontal arrow from one line to another

Similarly a component returning control or a result value back to the called is represented by an arrow

Commentary on the right side of the figure fully explains the interaction taking place

It describes the sequencing of events during a scenario with a time axis, so can be used as a documentation tool for complex software systems

## 2.7 Software Components:

Each component is characterized by:

i. **Behavior**

- Set of actions it can perform (*what they can do*)
- The complete description of all the behavior for a component is called a *protocol*

- For Recipe component, this includes activities like editing the preparation instructions, displaying the recipe on the screen, printing the recipe

#### ii. State

- Each component holds certain information
- For Recipe component, the state includes the ingredients and preparation instructions
- State is not static as it can change over time
- The user can make changes to the preparation instructions (*state*) by editing a recipe (*behavior*)

### 2.7.1 Instances and Classes:

In case of a real application, there will be many different recipes.

However, all of these recipes will perform in the same manner i.e. behavior of each recipe is same

But the state (*e.g. individual list of ingredients, instructions etc*) differs between individual recipes

Class refers to a set of objects with similar behavior

An individual representation of class is known as an *instance*

All instances of a class respond to the same instructions and perform in a similar manner

But these instances all may be in different state as state is the property of an individual

### 2.7.2 Coupling and Cohesion:

Cohesion is a qualitative indication of the degree to which a module focuses on just one thing

A *cohesive module* should do just one thing

High cohesion is achieved by associating in a single component tasks that are related in some manner

Coupling is a qualitative indication of the degree to which a module is connected to other modules an to the outside world

The amount of coupling should be reduced as connections between software components decrease ease of development, modification and reuse (*ripple effect*)

Coupling is increased when one software component must access data values (the state) held by another component

This situation should be avoided as far as possible

For example, in case Recipe Database in IIKH, while performing tasks associated with this component the need to edit a recipe first occurs

So it would be wise to assign responsibility for editing a recipe first

### 2.7.3 Interface and Implementation:

The deliberate omission of implementation details behind a simple interface is known as information hiding

It is possible for one programmer to know how to use a component developed by another programmer with out needing to know how the component is implemented

The component encapsulates the behavior showing only how the component can be used and not the detailed actions it performs

#### Two views of a Software System:

##### i. Interface View

- The face seen by other programmers
- Describes what a software component can perform

ii. **Implementation View**

- The face seem by the programmer working on a particular component
- Describes how a component goes about completing a task

The separation of interface and implementation is the most important concept in software engineering

Information hiding is largely meaningful only in context of multi person programming projects

Here, the limiting factor is not the amount of coding involved but the amount of communication required between programmers and their software systems

For the reuse of general purpose software components in multiple projects there must be minimal and well understood interconnections between the various portion so the system

**Parnas's Principles** (**David Parnas**):

- The developer of a software component **must provide the intended user with all 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

## 2.8 Formalize the Interface:

This step includes making decision regarding the general structure that will be used to implement each component:

i. A component with only **one behavior and no internal state** may be made into a **function** e.g. a component that takes a string of text and translates all capital letter to lowercase

ii. Components with **many tasks responsibilities** are more easily implemented as **classes**, use a CRC card to record all responsibilities and convert them into procedures or functions

iii. Along with the **names**, the **types of arguments to be passed** to the function are identified

iv. Next, the information maintained with in the component itself should be described in detail e.g. source of data must be clearly identified i.e. either local , global or passed thru an argument

### 2.8.1 Coming up with names

- Names associated with various activities should be carefully chosen
- Names create the vocabulary with which the eventual design will be formulated
- Names should be meaningful, short and suggestive in the context of the problem

**General guidelines for choosing names:**

- Use pronounceable names (*one that you can read out loud*)
- Use capitalization (*or underscores*) to the beginning of a new word with in a name "*CardReader*" or "*Card_Reader*"
- Abbreviations should not be confusing
  TermProcess: a terminal process or something that terminates a process?
- Avoid names with many interpretations. e.g. empty(), full()
- Avoid digits within a name , 1 and l, 2 and Z , 0 and o
- Name functions and variables that yield *Boolean values* so that they describe clearly the interpretation of a true or false value. e.g. *PrinterIsReady* --> a true value means the printer is working where as *PrinterStatus* is 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

The CRC cards for each component, along with the name and formal arguments of the function depicting its behavior, are redrawn after the names have been developed for each activity

| **Date** | |
|---|---|
| **Maintain information about specific date**<br>*Date(year, month, day)* **–create new date**<br>*DisplayAndEdit( )***- display date information in window allowing user to edit entries**<br>*BuildGroceryList(List &)***- add items from all means to grocery list** | **Collaborators**<br>**Meal Manager**<br>**Meal** |

Revised CRC card for the Date Component

## 2.9 Design the Representation for Components:

The design team is divided into groups each responsible for one or more software component

Transformation of the description of a component into a software system implementation is done now

Designing of the data structures that will be used by each subsystem to maintain the state information required to fulfill the assigned responsibilities is done

Once the data structure is chosen (an important task) the code used by the component in the fulfillment of a responsibility is often self evident

A wrong choice can lead to complex and inefficient program

Now the description of behavior must be transformed into algorithms

These descriptions should fulfill the expectations of each collaborator component

## 2.10 Implementing Components:

Each component's desired behavior is implemented in a computer language after the design phase is over

Proper design yields a short description of each responsibility or behavior

As one programmer will not work on all aspects of a system, he needs to acquire skills to:

- Understand how one section of code fits into a larger framework
- Learn to work well with other members of a team

There might be components that work in back ground (facilitators) that needs to be taken into account

Other aspects include

- documenting the necessary pre conditions a software component requires to complete a task
- verifying that the software will perform correctly when presented with legal input values

## 2.11 Integration of Components:

After software subsystems have been individually designed and tested they can be integrated into the final product

"*If all modules work individually should we doubt that they will work then we put them together?*"

Problems that arise during integration of components:

- Data can be lost across an interface
- One module can have an inadvertent effect on another
- Sub functions when combined may not produce desired result
- Global data structures can present problems

Starting from a simple base elements are slowly added to the system and tested using stubs (simple dummy with no or very limited behavior)

Next, one or the other stubs can be replaced by more complete code and tested to see if the system is working as desired, this is called *Integration Testing*

The application is finally complete when all stubs have been replaced with working components

Less coupling facilitates testing components in isolation.

Errors might creep in during integration that requires changes to come of the components

This follows retesting of the components in isolation before attempt to reintegrate the software once more

Re-executing previously developed test cases following a change to a software component is called *Regression Testing*

**Example:**

In case of IIKH, we start integration with the *Greeter* component

To test *Greeter* in isolation stubs are written for the *Recipe Database* and the daily *Meal Plan* manager

These stubs need not do any more than print an informative message and return

With this the development team can test various aspects of the *Greeter* system (*e.g. whether button press provides correct response or not*)

Testing of an individual component is called *Unit Testing*

The team then might decide to replace the stub for the *Recipe Database* component with the actual code maintaining the stub for other portion.

## 2.12 Maintenance and Evolution:

After the delivery of the working version of an application comes the phase of Software Maintenance

**Activities:**

- Bugs must be corrected in the form of patches to existing releases or in subsequent releases
- Change in requirements due to government regulations, standardization among similar products
- Hardware change (*input/output technology*) e.g. pen based to touch panel, text to GUI based system
- Change in user expectations (*greater functionality, lower cost, easier user*) , competition
- Request for better documentation by users

A good design recognizes the inevitability of changes and plans an accommodation for them from the very beginning.