CHAPTER 7
Object Oriented Design

7.1 Responsibility implies non-interface

- 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.
- 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-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.

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 might work for both a simulation of balls on a billiards table and a simulation of fish in a fish tank. This ability to reuse code implies that the software can have almost no domain-specific components; it must totally delegate responsibility for domain-specific behavior to application-specific portions of the system. The ability to create such reusable code is not one that is easily learned-it requires experience, careful examination of case studies (paradigms, in the original sense of the word), and use of a programming language in which such delegation is natural and easy to express.

7.2 **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.

7.3 Role of Behavior in OOP

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.

7.4 **Responsibility-Driven Design**

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 focuses on:

What action is object responsible for?
What information does the object share?

- 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.
- One of the technique is use of index cards to represent individual class. such cards are known as index cards.

7.4 Case study in RDD

7.4.1 The interactive Intelligent Kitchen Helper

<u>Brief Description</u>

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.

7.4.2 Working through scenarios

Initial specifications are almost always ambiguous and unclear on anything except the most general points. So, the first task is to refine the specification.

The specifications for the final application will may change during the creation of the software system, so it is important that, the design be developed to easily accommodate change and that potential changes be noted as early as possible.

Equally important, at this point very high level decisions can be made concerning the structure of the eventual software system. In particular, the activities to be performed can be mapped onto components.
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. An example scenario is as below.

7.4.3 Identification of components

- 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-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.

7.5 CRC Cards (Class Responsibility Collaborators)

CRC stands for **Class Responsibility Collaborators**.

CRC cards are a brainstorming tool used in the design of object-oriented software.

It was first introduced by Kent beck and Ward Cunningham.

The cards are arranged to show the flow of messages among instances of each class.

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

Class Name	
Responsibilities	Collaborators

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

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

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

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

Professor	
Name	Seminar
Address	
Email address	
Salary	
Provides information	
Instructing seminar	

Transcript	
Student name	Student
Marks obtained	
Enrolled year	
Calculate final grade	

Fig: CRC cards for class student

Things to be understand:

Here, Student is used as class name across the top of a CRC card .We use name of a class as a singular noun because each class represents a generalized version of a singular object.

As we know responsibility is anything that a class knows or does. In this example, students have names, addresses, and phone numbers. These are the things a student knows. Students also enroll in seminars and request transcripts. These are the things a student does.

Sometimes a class has a responsibility to fulfill, but not have enough information to do it. In above example, as you see students enroll in seminars. To do this, a student needs to know if a spot is available in the seminar and, if so, he then needs to be added to the seminar. However, students only have information about themselves (their names and so forth), and not about seminars. What the student needs to do is collaborate/interact with the card labeled Seminar to sign up for a seminar. Therefore, Seminar is included in the list of collaborators of Student. In this way we can draw CRC cards.

7.5.1 Give components a physical Representation

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.

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 (which we will describe shortly).

7.5.2 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.

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

7.5.3 Documentation

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

The user manual describes the interaction with the system from the user's point of view.

Before any actual code has been written, the mindset of the software team is most similar to that of the eventual users.

The system Design Documentation

- 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.
- 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 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.

7.6 Interaction Diagram /Sequence 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.

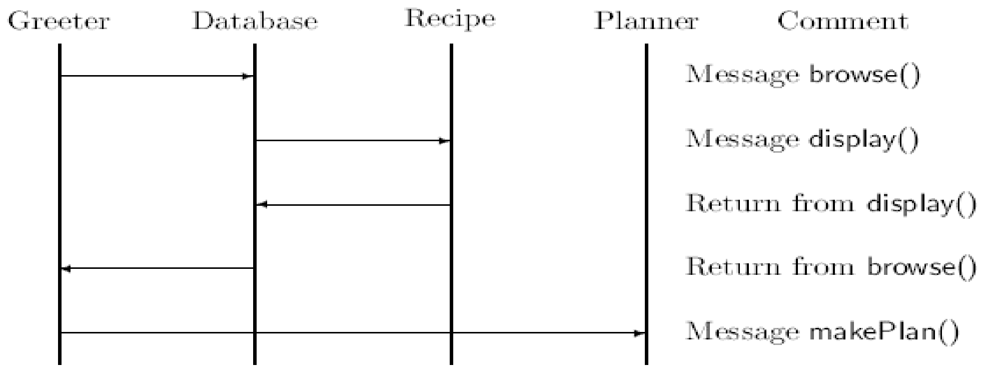


Figure 0-1: An Example interaction diagram.

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

7.7 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.

7.8 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.

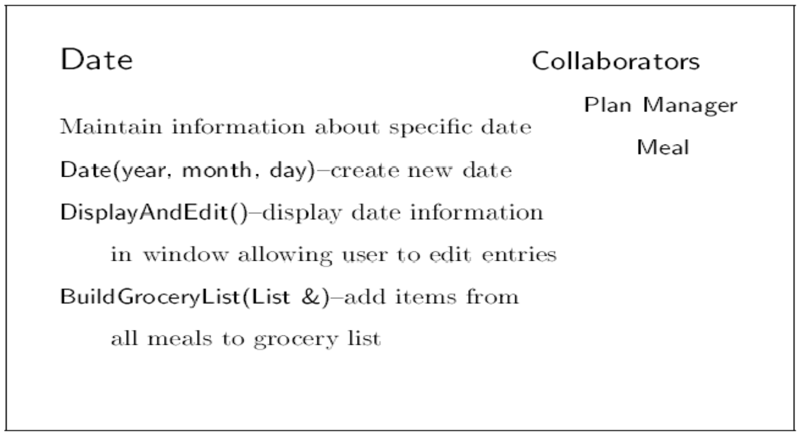
7.8.1 Coming up with names

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 .



Date

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

Collaborators

Plan Manager

Meal

Figure: Revised CRC card for the Date component.

7.9 **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.

7.10 **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.

7.11 **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.**

7.12 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.

Previous Old Questions from this chapter

1) Explain Responsibility implies non-interface. Explain with example.
2) Explain the terms:
    - Responsibility implies non-interface
    - Programming in small and Programming in large[PU:2014 fall]

3) What are the difference between programming in small and programming in large?[PU:2010 spring]
4) What is Role behavior of OOP? Along with figure and an example of CRC card .Explain its significance in object oriented Design.[PU:2006 spring[[PU:2015 fall]
5) 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

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

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

7) What do you mean by responsibility driven design? Also explain what is meant by CRC card.[PU:2010 spring]
8) What do you mean by software component? Explain Integration of components with suitable example scenario to support your answer.[PU:2010 spring]
9) What are the advantages of adopting RDD? Explain with the example of suitable example.[PU:2009 spring]
10) Do you find any advantages of adopting Responsibility Driven Design? Explain with help of suitable example.
11) 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]
12) How are object oriented Programs are designed and developed according to the concept of RDD? Describe entire process in brief.[PU:2013 spring]
13) 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]
15) What are the different aspects of software components?[PU:2016 spring]
16) Explain in brief about interface and implementation.
17) Draw CRC cards of students.[PU:2017 spring]
18) Explain CRC card and sequence diagram with suitable example.[PU:2019 fall]
19) Differentiate between the concept of computation as simulation and Responsibility implies non-interface. [PU:2017 spring]

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]

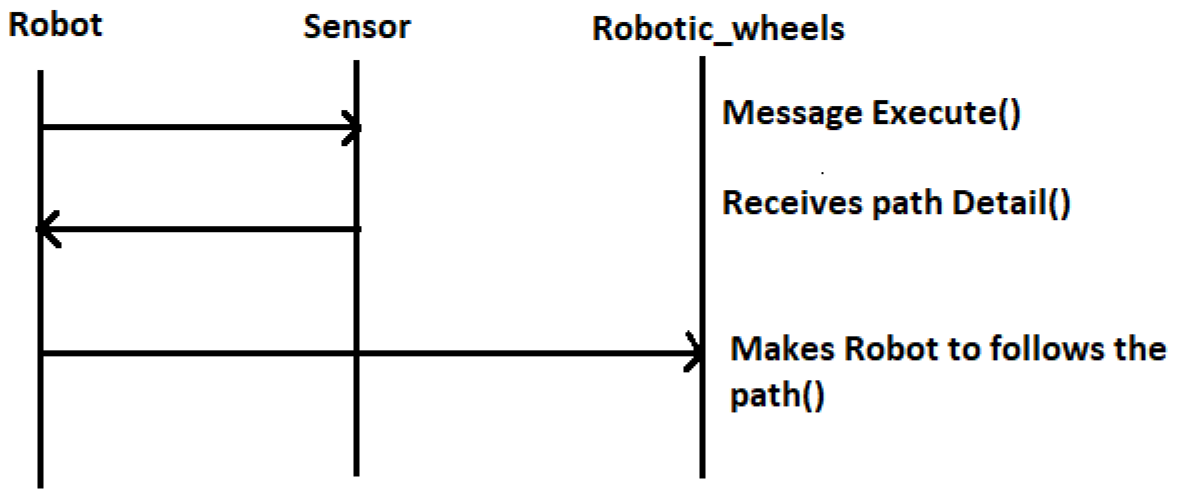
Three CRC cards are given below.

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


Robot
Sensor
Robotic_wheels
Message Execute()
Receives path Detail()
Makes Robot to follows the path()