Chapter 7
Object Oriented Design

Reusability Implies Non-interference

- 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. This ability to reuse code implies that the software can have almost no domain-specific components; it must totally delegate responsibility for domain-specific behaviour 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.

Programming in small

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 large**

Programming in the large 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. No single individual can be considered responsible for the entire project, or even necessarily understands all aspects of the project.
- With programming in the large, coding managers place emphasis on partitioning work into modules with precisely-specified interactions. This requires careful planning and careful documentation.
- 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 behaviors in OOP

- Earlier software development methodologies looked at 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.
- A formal specification often ended up as a document understood by neither programmer nor client. But behaviour 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

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

Responsibility-Driven-Design is a way of designing complex software systems using objects and component technology. Responsibility-Driven Design was conceived in 1990 as a shift from thinking about objects as data + algorithms, to thinking about objects as roles + responsibilities. Responsibility-Driven Design catalyzes object technology with practical techniques and thinking tools.

A case study in RDD

- 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 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.

CRC cards (class 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.

i**) 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.

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

ii) **The What/Who Cycle**

Design 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.

iii) **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. CRC cards are one aspect of the design documentation, but many other important decisions are not rejected in them. Arguments for and against any major design alternatives should be recorded, as well as factors that influenced the final decisions.

Classname	
Responsibilities	Collaborators

Fig: CRC card format

Greeter	
Display informative initial message Offer user choice of options Pass control to either • Recipe database manager • Plan manager for processing	Database manager Recipe manager

Fig: Greeter class in IIKH scenario

ATM CRC card example

Card reader	
Tell ATM when card is inserted Read information from card Eject card Retain card	**ATM** **Card**

Cash dispenser	
✓ Keep track of cash on hand, starting with initial amount ✓ Report whether enough cash is available ✓ Dispense cash	**Log**

Advantages

• Uses index cards.

– Cheap, readily available, amenable to group use.
• Forces you to be concise and clear.
– Puts your attention on the what, not how.
• Simple, easy methodology
• Intuitive
• Portable

Sequence Diagram

It is used to show dynamic behavior of the components.

- In the diagram, time moves forward from the top to the bottom. Each component is represented by a labelled 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. The commentary on the right side of the figure explains more fully the interaction taking place.

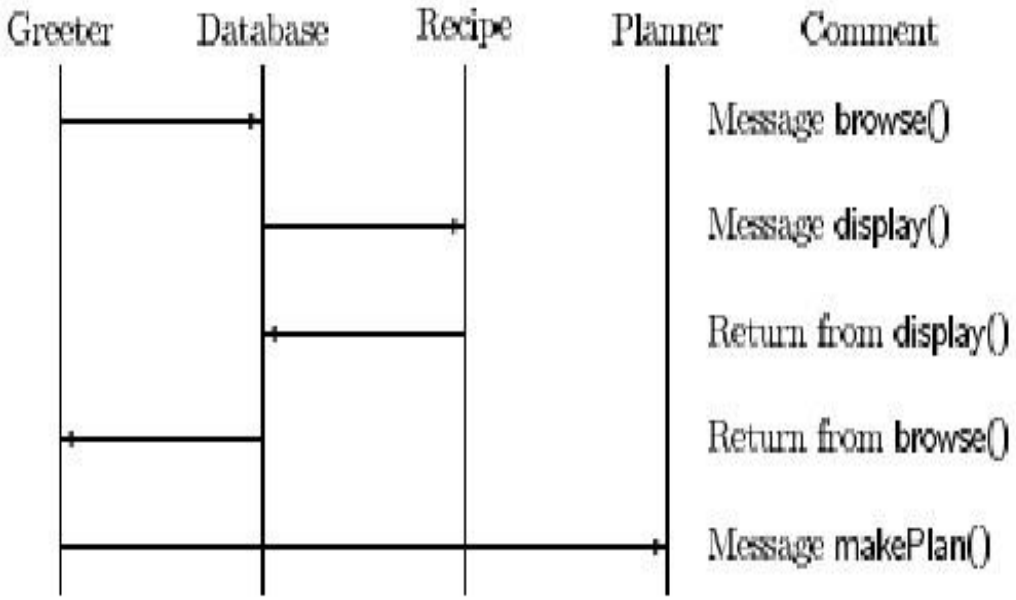

Fig: a sequence diagram for IIKH

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 work 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 a component include: a single button in a graphical user interface, a small interest calculator, an interface to a database manager.

i) Behavior and State: One way to view a component is as a pair consisting of behaviour 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).

ii) Coupling and Cohesion

- Two important concepts in the design of software components are 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

iii) Interface and Implementation

It is very important to know the difference between interface and implementation. For example, when a driver drives the car, he uses the steering to turn the car. The purpose of the steering is known very well to the driver, but the driver need not to know the internal mechanisms of different joints and links of various components connected to the steering.

An interface is the user's view of what can be done with an entity. It tells the user what can be performed. Implementation takes care of the internal operations of an interface that need not be known to

the user. The implementation concentrates on how an entity works internally. Their comparison is shown in Table

Comparison of interface and implementation

Interface	Implementation
• It is user's viewpoint. (**What** part) • It is used to interact with the outside world. • User is permitted to access the interfaces only. • It encapsulates the knowledge about the object.	• It is developer's viewpoint. (**How** part) • It describes how the delegated responsibility is carried out. • It describes how the delegated responsibility is carried out. • It provides the restriction of access to data by the user.

.

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.

Coming up with names

- Names should be internally consistent, meaningful, preferably short, and evocative in the context of the problem.

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.

Design and representation of components

- The task now is to transform the description of a component into a software system implementation major portion of this process is designing the data structures that will be used by each subsystem to maintain the state information required to fulfil the assigned responsibilities
- It is here that the classic data structures of computer science come into play. The selection of data structures is an important task, central to the software design process. Once they have been chosen, the code used by a component in the fulfilment of a responsibility is often almost selfevident. But data structures must be carefully matched to the task at hand. A wrong choice can result in complex and inefficient programs, while an intelligent choice can result in just the opposite.

- 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 fullled and necessary data items are available to carry out each process.

Implementation of components

- The next step is to implement each component's desired behavior. 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.
- An important part of analysis and coding at this point is characterizing and documenting the necessary **preconditions** a software component requires to complete a task, and 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, 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**.
- **Integration testing** can be performed until it appears that the system is working as desired.
- Re-executing previously developed test cases following a change to a software component is sometimes referred to as **regression** testing.
- Give example of car making with different components bumper, gear, engine etc.