Chapter 1: Thinking Object Oriented

Object Oriented Programming – a new paradigm

- OOP is a new programming paradigm developed to overcome limitations of earlier programming approaches
  - A *revolutionary* idea: totally unlike earlier approaches
  - An *evolutionary* step: addition to abstractions mechanisms of earlier approaches
- Paradigm:
  - A typical example or pattern or model of something
  - A set of theories, standards, and methods that together represent a way of organizing knowledge – that is, "a way of viewing the world." [Kuhn, 1970]
  - A programming paradigm is a way of conceptualizing what it means to perform computation and how tasks to be carried out on a computer should be structured and organized. [Floyd, 1979]

- Other competing programming paradigms include:
  - Imperative or procedure-oriented programming paradigm (e.g., C)
  - Logic programming paradigm (e.g., Prolog)
  - Functional programming paradigm (e.g., Scala)
- The style of problem solving embodied in object-oriented technique is similar to the method used to address problems in everyday life.
- OOP follows a bottom-up approach for problem solving:
  - A design approach where parts of the system are first defined in details.
  - Then these parts are linked together to prepare a bigger component.
  - This approach is repeated until the complete system is built.
- Unlike procedural programming where a program is just a list of instructions, a program in OOP can be viewed as a collection of co-operating objects.

Why this new paradigm?

- Quickly and easily lead to increased productivity and improved reliability
- Helps to solve software crisis to some extent.
  - software Crisis = our imaginations and the tasks we would like to solve with the help of computers almost always outstrip our abilities
- Similarity to thinking about and problem-solving techniques in everyday life and other domains

A way of viewing the world:
Agents, Responsibility, Messages, and Methods

- To solve an everyday problem:
  - First, we identify an **Agent** which is able to solve the problem at hand
  - Next, we need to pass a **Message** containing a request to the agent
  - Now, it is the **Responsibility** of the agent to fulfill this request
  - Finally, there must be some **Method** (a set of standard operations) that is followed by the agent to achieve this. Either the agent will do everything by itself or it will request other agents to carry on some or all of the required tasks.
- We need not know how the agent solves our problem; we just want to get the things done.
  - An important OOP concept: "*information hiding*" or "*Abstraction*"
  - As an example: think of online shopping ...

Initiation of an activity in OOP

Similar to solving an everyday problem, initiation of an activity in OOP would undergo the following steps:

- Action is initiated in by transmission of a **message** to an **agent** (a receiver object) responsible for the action.
- The **message** encodes the request for an action and is accompanied by any additional information (arguments) needed to carry out the request.
- If the receiver object (agent) accepts the message, it accepts the **responsibility** to carry out the indicated action.
- In response to the message, the receiver agent shall perform some **method** to satisfy the request.

Computation as Simulation

- The traditional model describing the behavior of a computer executing a program is a *process-state model* or *pigeon-hole model*.
  - The computer is considered as a data manager following some pattern of instructions, wandering through memory, pulling values out of various slots (memory addresses), transforming them in some manner, and pushing the results back into other slots.
  - By examining the values of the slots, we can determine the state of the machine or the results produced by a computation.
  - Gives more or less accurate idea of what goes inside a computer, but does not help much to understand how to solve problems using computer.

- In object-oriented framework, we are not concerned with such conventional terms. Instead, we speak of objects, messages, and responsibility for some action.
    - Instead of a bit-grinding processor … plundering data structures, we have a universe of well-behaved objects that courteously ask each other to carry out their various desires. [Ingalls 1981]
    - Object oriented programming as “animistic”: a process of creating a host of helpers that form a community and assist the programmer in the solution of a problem. [Actor 1987]

- These views of programming as creating a “universe” is in many ways similar to a style of computer simulation known as “*discrete event-driven simulation*”
  - the user creates computer models of the various elements of simulation,
  - describes how they will interact with one another,
  - and sets them moving.
- The concept of discrete event-driven simulation is almost identical to most object-oriented programs:
  - describe the various entities in the universe of the program
  - determine how they will interact with one another,
  - and set them in motion.
- Hence, in OOP, we have a view that “*Computation is Simulation*.”

The Nonlinear Behavior of Complexity

- Traditionally, as programming projects became larger, an interesting phenomenon was observed:
  - A task that would take one programmer two months to perform could not be accomplished by two programmers working for one month in parallel.
  - The reason for this nonlinear behavior was complexity: particularly, the interconnections between software components were complicated, requiring large quantities of information to be communicated among the various members of the programming team.
- Since software construction is inherently a systems effort – an exercise in complex interrelationships – communication effort is huge, and it quickly dominates the decrease in individual task time brought about by partitioning. Adding more people then lengthens, not shortens, the schedule. [Brooks 1975]

- Reason for complexity is not simply just due to the size of the tasks undertaken because size itself would not be an obstruction to partitioning each into several pieces.
- Software systems developed using conventional techniques:
  - ❖ high degree of interconnectedness between different portions
  - ❖ one of the most complex systems developed by humans
- An individual section of code cannot be understood in isolation:
  - Each portion of software performs an essential task
  - Which must be useful to other parts of the software
  - There must be some communication into or out of the component
  - For a complete understanding of what is going on requires knowledge of both the portion under consideration and the portion using it
- To resolve such complexities, different Abstraction Mechanisms have evolved

Abstraction Mechanisms

- Programmers have had to deal with problem of complexity for a long time.
- Abstraction is one of the mechanisms among various others which have been used by programmers to resolve complexity.
- ***Abstraction*:** Ability to encapsulate and isolate design and execution information i.e., interface and implementation details.
- Object-oriented techniques are not revolutionary. It is a natural outcome of a long historical progression of different abstraction mechanisms:
    - Procedures / Functions / Sub-routines
    - Modules
    - Abstract data types
    - Objects

i) Procedures / Functions / Subroutines

- First abstraction mechanisms used in high level languages
- Tasks executed repeatedly or with only slight variations collected in one place and reused, rather than being duplicated several times.
- Provides a possibility of information hiding to some extent.
- One programmer can develop a set of procedures which can be used by different programmers.
- The other programmers need not know the exact details of the implementation – they just need to know the interface for using it.
- However, procedures are not a solution to all the complexity problems. Particularly, they are not an effective mechanism for information hiding.
- They only partially solve the problem of multiple programmers making use of the same names.

ii) Modules

- An improved technique for creating and managing collection of names and their associated values
- Ability to divide a namespace into two parts:
  - the *public* part accessible outside the module
  - the *private* part accessible only within the module
  - data and procedures can be defined in either of the portions
- Modules solve some complexities of software development, but not all
- Hide the implementation details (information hiding) but provides no instantiation mechanism (no reusability or creation of multiple copies)

iii) Abstract Data Types (ADT)

- A programmer defined data type that can be manipulated like system-defined data types
- Corresponds to a set of legal data values and a number of primitive operations which can be performed on those data
- Allows creating variables with legal values and can operate on those values using defined operations
- To build an abstract data type, we must be able to:
  - Export type definition
  - Make available a set of operations that can be used to manipulate its instances
  - Protect the data associated with the type so that they can be operated on only by the provided routines
  - Make multiple instances of the type

iv) Classes and Objects – Messages, Inheritance, and Polymorphism

OOP adds several important new ideas to the concept of the abstract data type.

- The most basic idea is ***Encapsulation*** of data values (state information) and operations (functions) into a single unit called class from which any number of instances called objects can be created.
- Another important feature is ***Message Passing***.
  - Activity is initiated by a request to a specific object and not by the invoking of a standalone function. The conventional approach places emphasis on the operation, whereas the object-oriented approach emphasizes the value itself.
  - *Example: Passing the stack data structure and the data to a push function versus asking a stack to push a data item to itself*
  - Additional power comes from mechanisms for overloading names and reusability.

  - Interpretation of a message can vary with different objects i.e., behavior and response that the message produce will depend upon the object receiving it.
  - Names for operations across different entities need not be unique. Meaning of push operation for a Stack object can be different from that of a Button object.
- ***Inheritance*** allows creation of a new class by extending the features of an existing class, allowing the different data types to share the same code, leading to a reduction in code size and an increase in functionality and reusability.
- ***Polymorphism*** allows this shared code to be tailored to fit the specific circumstances of individual data types via concepts like overloading and overriding.
- The emphasis on the **independence** of individual components permits an incremental development process in which individual software units are designed, programmed, and tested before combining into a larger system.

Types / Varieties of Classes

Classes in OOP can have several different forms of responsibilities and are thus used for many different purposes, broadly classified (not limited to) as:

Data Managers: Maintains data or state information

Data Source: Generates data on demand (but does not hold any data)

Data Sink: Accepts data and processes them further (but does not hold any data)

View or Observer class: Used to visualize the data in different forms

Facilitator or Helper class: Maintains little or no state information itself but assist in the execution of complex tasks.