UpGrad

Lecture Notes

Data Abstraction

In this module, you learnt about the design principle called Separation of Concerns (SoC) and how the concept of abstraction implements the same.

Separation of Concerns (SoC)

The design principle called Separation of Concerns helps you to separate the concerns of the users from those of the providers (software developer).

Let's take the example of a mobile phone, where the concerns of the user are as follows:

- The phone should produce clear audio
- The phone should not be too heavy
- The phone should not lag
- The phone should be affordable

Whereas the implementer or the provider of the phone is concerned with the following:

- The power consumption
- Where to purchase the various modules of the device from
- Can the cost of components be cheaper?

Now, since the concerns of both the user and the provider are different, the provider needs to take its concerns into consideration and think about their implementations separately. A software designer must identify and separate these concerns, look at the perspective of the user versus the perspective of the provider, and identify what needs to be done for each of these. These concerns form the design elements of any software.

Abstraction

Abstraction is the process of helping the software developer address the concerns of both the user and the provider separately. Using abstraction, you can hide the actual implementation details of the software from the user and only provide him with the ability to use a particular functionality. In simple terms, this means that the user only knows what the software does but not how it does it.

Abstraction example:

You learnt about the auto meter or the fare meter of an auto rickshaw. For a user riding an auto rickshaw, the expectation is that when he presses a button, a ticket with the total fare is printed. This can be implemented as a function named getfare(), which returns the total amount to be paid to the driver. On the other hand, the provider needs to figure out how to compute the fare. The provider needs to consider the base fare, the distance rate (rate

per KM), and the total distance covered by the user and, based on the accumulated information, calculate the total fare. This can be done by writing the formula as shown below:

Total Fare = Base Fare + ((Fare per Km) *Total Distance)

To implement this formula, the provider must calculate the distance, which can be done in multiple ways.
Now, using abstraction, we can hide the details of the actual implementation of these calculations from the user, and just provide him with the total fare.

Data Abstraction

Data abstraction allows you to separate the concerns of users from providers with respect to data.
The user concerns with data are —

- What is the type of data?
- What is the behaviour associated with the data, i.e. what operations can be performed on the data?

The provider's concerns with data are —

- How to represent or store this data?
- How to implement the operations that are associated with this data?

Abstract Data Types (ADT)

An Abstract Data Type (ADT) is a model of a data structure that defines the following:

1. The properties of the collection of data
2. The operations that can be performed on the data

You already know that Abstract Data Types (ADT) help us achieve abstraction of a program. Some other benefits of using ADTs are as follows:

1. It helps understand the code easily
2. It makes it easier to implement changes to the actual data structure code since only a small portion needs to be edited.
3. They can be reused later

A simple way to understand ADTs is to take an example ADT, say, a box with numbers in it. This is similar to the individual word blocks in a Scrabble set. Now, in this box, neither the sequence of numbers matters because both [12, 23, 5, 6] and [23, 12, 5, 6] are the same, and nor does the uniqueness of a number.
Some things (operations) that can be performed with this box are —

1. Add a number to this box: addNum()
2. Search for a number: searchNum()
3. Remove numbers from the box: removeNum()
4. Find the number of items in the box: sizeBox()

Implementing ADTs Efficiently

While implementing an ADT, the provider will have a wide range of data structures with him, and he can implement the ADT using any one of the data structures. You saw how a Stack ADT could be implemented using either an array or a linked list. The important question that needs to be answered is, “How does a provider decide the best possible ADT for a given problem?”

Well, there is no sure shot way to arrive at an answer, but there are some things that the provider should do to arrive at the best possible solution. These are —

- The provider should find out the operations that will be used most frequently for an ADT in any scenario.
- The provider should then consider the cost of implementing the most frequent operations of the ADT using a different data structure and choose the best option.

To better understand this, let's take a look at an example of the dictionary ADT. You can use either an array or a linked list to implement it. Now, you have to finalise one data structure by considering the fact that the most used operations on the dictionary ADT would be 'find' and 'add'. But if you were implementing the ADT for a phone book on a mobile phone, you would be using the 'find' operation far more frequently than the 'add' operation. Hence, it would be better to implement the ADT using a sorted list. Similarly, you can then choose the best possible implementation for different use cases.