SOFTWARE DESIGN & ARCHITECTURE

LAB 14

# ITERATOR & MEDIATOR DESIGN PATTERN

# ITERATOR DESIGN PATTERN:

## Iterator Pattern is a relatively simple and frequently used design pattern. There are a lot of data structures/collections available in every language. Each collection must provide an iterator that lets it iterate through its objects. However while doing so it should make sure that it does not expose its implementation. Suppose we are building an application that requires us to maintain a list of notifications. Eventually, some part of your code will require to iterate over all notifications. If we implemented your collection of notifications as array you would iterate over them as:

## ***// If a simple array is used to store notifications***

## *for (int i = 0; i < notificationList.length; i++)*

## *Notification notification = notificationList[i]);*

## ***// If ArrayList is Java is used, then we would iterate***

## ***// over them as:***

## *for (int i = 0; i < notificationList.size(); i++)*

## *Notification notification = (Notification)notificationList.get(i);*

## And if it were some other collection like set, tree etc. way of iterating would change slightly. Now, what if we build an iterator that provides a generic way of iterating over a collection independent of its type.

## // Create an iterator

## Iterator iterator = notificationList.createIterator();

## // It wouldn’t matter if list is Array or ArrayList or

## // anything else.

## while (iterator.hasNext())

## {

## Notification notification = iterator.next());

## }

## Iterator pattern lets us do just that. Formally it is defined as below: The iterator pattern provides a way to access the elements of an aggregate object without exposing its underlying representation. **Conceptual Example**

This example illustrates the structure of the **ITERATOR** design pattern.

#### **Program.java:** Conceptual example

// A Java program to demonstrate implementation

// of iterator pattern with the example of

// notifications

// A simple Notification class

**class** Notification

{

    // To store notification message

    String notification;

**public** Notification(String notification)

    {

**this**.notification = notification;

    }

**public** String getNotification()

    {

**return** notification;

    }

}

// Collection interface

**interface** Collection

{

**public** Iterator createIterator();

}

// Collection of notifications

**class** NotificationCollection **implements** Collection

{

**static** **final** **int** MAX\_ITEMS = 6;

**int** numberOfItems = 0;

    Notification[] notificationList;

**public** NotificationCollection()

    {

        notificationList = **new** Notification[MAX\_ITEMS];

        // Let us add some dummy notifications

        addItem("Notification 1");

        addItem("Notification 2");

        addItem("Notification 3");

    }

**public** **void** addItem(String str)

    {

        Notification notification = **new** Notification(str);

**if** (numberOfItems >= MAX\_ITEMS)

            System.err.println("Full");

**else**

        {

            notificationList[numberOfItems] = notification;

            numberOfItems = numberOfItems + 1;

        }

    }

**public** Iterator createIterator()

    {

**return** **new** NotificationIterator(notificationList);

    }

}

// We could also use Java.Util.Iterator

**interface** Iterator

{

    // indicates whether there are more elements to

    // iterate over

**boolean** hasNext();

    // returns the next element

    Object next();

}

// Notification iterator

**class** NotificationIterator **implements** Iterator

{

    Notification[] notificationList;

    // maintains curr pos of iterator over the array

**int** pos = 0;

    // Constructor takes the array of notificationList are

    // going to iterate over.

**public**  NotificationIterator (Notification[] notificationList)

    {

**this**.notificationList = notificationList;

    }

**public** Object next()

    {

        // return next element in the array and increment pos

        Notification notification =  notificationList[pos];

        pos += 1;

**return** notification;

    }

**public** **boolean** hasNext()

    {

**if** (pos >= notificationList.length ||

            notificationList[pos] == **null**)

**return** **false**;

**else**

**return** **true**;

    }

}

// Contains collection of notifications as an object of

// NotificationCollection

**class** NotificationBar

{

    NotificationCollection notifications;

**public** NotificationBar(NotificationCollection notifications)

    {

**this**.notifications = notifications;

    }

**public** **void** printNotifications()

    {

        Iterator iterator = notifications.createIterator();

        System.out.println("-------NOTIFICATION BAR------------");

**while** (iterator.hasNext())

        {

            Notification n = (Notification)iterator.next();

            System.out.println(n.getNotification());

        }

    }

}

// Driver class

**class** Main

{

**public** **static** **void** main(String args[])

    {

        NotificationCollection nc = **new** NotificationCollection();

        NotificationBar nb = **new** NotificationBar(nc);

        nb.printNotifications();

    }

}

# MEDIATOR DESIGN PATTERN:

## Mediator design pattern is one of the important and widely used behavioral design pattern. Mediator enables decoupling of objects by introducing a layer in between so that the interaction between objects happen via the layer. If the objects interact with each other directly, the system components are tightly coupled with each other that makes higher maintainability cost and not hard to extend. Mediator pattern focuses on providing a mediator between objects for communication and help in implementing lose-coupling between objects.

## Air traffic controller is a great example of mediator pattern where the airport control room works as a mediator for communication between different flights. Mediator works as a router between objects and it can have its own logic to provide way of communication.

## Lightbox

## **Design components**

## **Mediator:** It defines the interface for communication between colleague objects.

## **ConcreteMediator:** It implements the mediator interface and coordinates communication between colleague objects.

## **Colleague:** It defines the interface for communication with other colleagues

## **ConcreteColleague**: It implements the colleague interface and communicates with other colleagues through its mediator

## **Conceptual Example**

This example illustrates the structure of the **MEDAITOR** design pattern.

#### **Program.java:** Conceptual example

**class** ATCMediator **implements** IATCMediator

{

**private** Flight flight;

**private** Runway runway;

**public** **boolean** land;

**public** **void** registerRunway(Runway runway)

    {

**this**.runway = runway;

    }

**public** **void** registerFlight(Flight flight)

    {

**this**.flight = flight;

    }

**public** **boolean** isLandingOk()

    {

**return** land;

    }

    @Override

**public** **void** setLandingStatus(**boolean** status)

    {

        land = status;

    }

}

**interface** Command

{

**void** land();

}

**interface** IATCMediator

{

**public** **void** registerRunway(Runway runway);

**public** **void** registerFlight(Flight flight);

**public** **boolean** isLandingOk();

**public** **void** setLandingStatus(**boolean** status);

}

**class** Flight **implements** Command

{

**private** IATCMediator atcMediator;

**public** Flight(IATCMediator atcMediator)

    {

**this**.atcMediator = atcMediator;

    }

**public** **void** land()

    {

**if** (atcMediator.isLandingOk())

        {

            System.out.println("Successfully Landed.");

            atcMediator.setLandingStatus(**true**);

        }

**else**

            System.out.println("Waiting for landing.");

    }

**public** **void** getReady()

    {

        System.out.println("Ready for landing.");

    }

}

**class** Runway **implements** Command

{

**private** IATCMediator atcMediator;

**public** Runway(IATCMediator atcMediator)

    {

**this**.atcMediator = atcMediator;

        atcMediator.setLandingStatus(**true**);

    }

    @Override

**public** **void** land()

    {

        System.out.println("Landing permission granted.");

        atcMediator.setLandingStatus(**true**);

    }

}

**class** MediatorDesignPattern

{

**public** **static** **void** main(String args[])

    {

        IATCMediator atcMediator = **new** ATCMediator();

        Flight sparrow101 = **new** Flight(atcMediator);

        Runway mainRunway = **new** Runway(atcMediator);

        atcMediator.registerFlight(sparrow101);

        atcMediator.registerRunway(mainRunway);

        sparrow101.getReady();

        mainRunway.land();

        sparrow101.land();

    }

}

**Task:** Implement the given design patterns in C++ or C# and identify the working of each Design Pattern respectively.