**ASSIGMNT NO 2:**

**BEHAVIORAL DESIGN Pattern.**

**These design patterns are all about Class's objects communication. Behavioral patterns are those patterns that are most specifically concerned with communication between objects.**

**Mediator Design Pattern:**

**Intent:**

* **Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.**
* **Design an intermediary to decouple many peers.**
* **Promote the many-to-many relationships between interacting peers to "full object status".**

**Problem:**

**We want to design reusable components, but dependencies between the potentially reusable pieces demonstrates the "spaghetti code" phenomenon (trying to scoop a single serving results in an "all or nothing clump").**

**Discussion:**

**In Unix, permission to access system resources is managed at three levels of granularity: world, group, and owner. A group is a collection of users intended to model some functional affiliation. Each user on the system can be a member of one or more groups, and each group can have zero or more users assigned to it. Next figure shows three users that are assigned to all three groups.**

**If we were to model this in software, we could decide to have User objects coupled to Group objects, and Group objects coupled to User objects. Then when changes occur, both classes and all their instances would be affected.**

**An alternate approach would be to introduce "an additional level of indirection" - take the mapping of users to groups and groups to users, and make it an abstraction unto itself. This offers several advantages: Users and Groups are decoupled from one another, many mappings can easily be maintained and manipulated simultaneously, and the mapping abstraction can be extended in the future by defining derived classes.**

**Partitioning a system into many objects generally enhances reusability, but proliferating interconnections between those objects tend to reduce it again. The mediator object: encapsulates all interconnections, acts as the hub of communication, is responsible for controlling and coordinating the interactions of its clients, and promotes loose coupling by keeping objects from referring to each other explicitly.**

**The Mediator pattern promotes a "many-to-many relationship network" to "full object status". Modelling the inter-relationships with an object enhances encapsulation, and allows the behavior of those inter-relationships to be modified or extended through subclassing.**

**An example where Mediator is useful is the design of a user and group capability in an operating system. A group can have zero or more users, and, a user can be a member of zero or more groups. The Mediator pattern provides a flexible and non-invasive way to associate and manage users and groups.**

**STRUCTURE:**

**Colleagues (or peers) are not coupled to one another. Each talks to the Mediator, which in turn knows and conducts the orchestration of the others. The "many to many" mapping between colleagues that would otherwise exist, has been "promoted to full object status". This new abstraction provides a locus of indirection where additional leverage can be hosted.**

**CODE:**

**//Mediator interface**

**public interface Mediator {**

**public void send(String message, Colleague colleague);**

**}**

**public abstract Colleague{**

**private Mediator mediator;**

**public Colleague(Mediator m) {**

**mediator = m;**

**}**

**public void send(String message) {**

**mediator.send(message, this);**

**}**

**//get access to the mediator**

**public Mediator getMediator() {return mediator;}**

**public abstract void receive(String message);**

**}**

**public class ApplicationMediator implements Mediator {**

**private ArrayList<Colleague> colleagues;**

**public ApplicationMediator() {**

**colleagues = new ArrayList<Colleague>();**

**}**

**public void addColleague(Colleague colleague) {**

**colleagues.add(colleague);**

**}**

**public void send(String message, Colleague originator) {**

**//let all other screens know that this screen has changed**

**for(Colleague colleague: colleagues) {**

**//don't tell ourselves**

**if(colleague != originator) {**

**colleage.receive(message);**

**}**

**}**

**}**

**}**

**public class ConcreteColleague extends Colleague {**

**public void receive(String message) {**

**System.out.println("Colleague Received: " + message);**

**}**

**}**

**public class MobileColleague extends Colleague {**

**public void receive(String message) {**

**System.out.println("Mobile Received: " + message);**

**}**

**}**

**MAIN CLASS:**

**public class Client {**

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

**ApplicationMediator mediator = new ApplicationMediator();**

**ConcreteColleague desktop = new ConcreteColleague(mediator);**

**ConcreteColleague mobile = new MobileColleague(mediator);**

**mediator.addColleague(desktop);**

**mediator.addColleague(mobile);**

**desktop.send("Hello World");**

**mobile.send("Hello");**

**}**

**}**

**\Iterator Design Pattern:**

**INTENT:**

* **Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.**
* **The C++ and Java standard library abstraction that makes it possible to decouple collection classes and algorithms.**
* **Promote to "full object status" the traversal of a collection.**
* **Polymorphic traversal**

**Problem:**

**Need to "abstract" the traversal of wildly different data structures so that algorithms can be defined that are capable of interfacing with each transparently.**

**Discussion**

**"An aggregate object such as a list should give you a way to access its elements without exposing its internal structure. Moreover, you might want to traverse the list in different ways, depending on what you need to accomplish. But you probably don't want to bloat the List interface with operations for different traversals, even if you could anticipate the ones you'll require. You might also need to have more than one traversal pending on the same list." And, providing a uniform interface for traversing many types of aggregate objects (i.e. polymorphic iteration) might be valuable.**

**The Iterator pattern lets you do all this. The key idea is to take the responsibility for access and traversal out of the aggregate object and put it into an Iterator object that defines a standard traversal protocol.**

**The Iterator abstraction is fundamental to an emerging technology called "generic programming". This strategy seeks to explicitly separate the notion of "algorithm" from that of "data structure". The motivation is to: promote component-based development, boost productivity, and reduce configuration management.**

**As an example, if you wanted to support four data structures (array, binary tree, linked list, and hash table) and three algorithms (sort, find, and merge), a traditional approach would require four times three permutations to develop and maintain. Whereas, a generic programming approach would only require four plus three configuration items.**

**Structure**

**The Client uses the Collection class' public interface directly. But access to the Collection's elements is encapsulated behind the additional level of abstraction called Iterator. Each Collection derived class knows which Iterator derived class to create and return. After that, the Client relies on the interface defined in the Iterator base class.**

**Iterator example**

**Example**

**The Iterator provides ways to access elements of an aggregate object sequentially without exposing the underlying structure of the object. Files are aggregate objects. In office settings where access to files is made through administrative or secretarial staff, the Iterator pattern is demonstrated with the secretary acting as the Iterator. Several television comedy skits have been developed around the premise of an executive trying to understand the secretary's filing system. To the executive, the filing system is confusing and illogical, but the secretary is able to access files quickly and efficiently.**

**On early television sets, a dial was used to change channels. When channel surfing, the viewer was required to move the dial through each channel position, regardless of whether or not that channel had reception. On modern television sets, a next and previous button are used. When the viewer selects the "next" button, the next tuned channel will be displayed. Consider watching television in a hotel room in a strange city. When surfing through channels, the channel number is not important, but the programming is. If the programming on one channel is not of interest, the viewer can request the next channel, without knowing its number.**

**Check list**

* **Add a create\_iterator() method to the "collection" class, and grant the "iterator" class privileged access.**
* **Design an "iterator" class that can encapsulate traversal of the "collection" class.**
* **Clients ask the collection object to create an iterator object.**
* **Clients use the first(), is\_done(), next(), and current\_item() protocol to access the elements of the collection class.**

**Rules of thumb**

* **The abstract syntax tree of Interpreter is a Composite (therefore Iterator and Visitor are also applicable).**
* **Iterator can traverse a Composite. Visitor can apply an operation over a Composite.**
* **Polymorphic Iterators rely on Factory Methods to instantiate the appropriate Iterator subclass.**
* **Memento is often used in conjunction with Iterator. An Iterator can use a Memento to capture the state of an iteration. The Iterator stores the Memento internally.**

**CODE:**

**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 notifiactionList 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();**

**}**

**}**