**Assignment no 2**

**Submitted To: Ma’am Mariyam Gillani**

**Submitted By: Iqra Sarfraz,**

**Filzah Amjad.**

**Roll No: FA16-BSE-073,**

**FA16-BSE-072.**

**Q-NO-1 BEHAVIORAL DESIGN PATTERNS**

**ITERATOR DESIGN PATTERN:**

**NAME:**

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

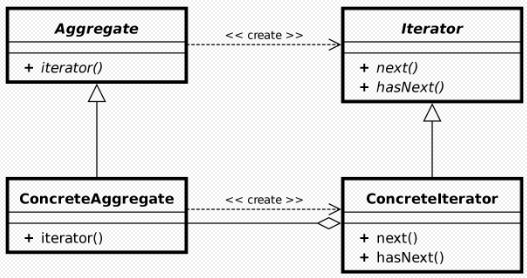
**MOTIVATION:**

* One of the most common data structures in software development is what is generic called a collection. A collection is just a grouping of some objects. They can have the same type or they can be all cast to a base type like object. A collection can be a list, an array, a tree and the examples can continue.
* But what is more important is that a collection should provide a way to access its elements without exposing its internal structure. We should have a mechanism to traverse in the same way a list or an array. It doesn't matter how they are internally represented.
* The idea of the iterator pattern is to take the responsibility of accessing and passing trough the objects of the collection and put it in the iterator object. The iterator object will maintain the state of the iteration, keeping track of the current item and having a way of identifying what elements are next to be iterated.

**APPLICABILITY:**

* Iterator pattern is useful when you want to provide a standard way to iterate over a collection and hide the implementation logic from client program.
* The logic for iteration is embedded in the collection itself and it helps client program to iterate over them easily.
* We all know that Collection framework Iterator is the best example of iterator pattern implementation but do you know that java.util.Scanner class also Implements Iterator interface.
* Access contents of a collection without exposing its internal structure.
* Support multiple simultaneous traversals of a collection.
* Provide a uniform interface for traversing different collection.

**STRUCTURE:**



**PARTICIPATORS:**

* Iterator (AbstractIterator):
  + Defines an interface for accessing and traversing elements.
* ConcreteIterator (Iterator):
  + Implements the Iterator interface.
  + Keeps track of the current position in the traversal of the aggregate.
* Aggregate (AbstractCollection):
  + Defines an interface for creating an Iterator object.
* ConcreteAggregate (Collection):
  + implements the Iterator creation interface to return an instance of the proper ConcreteIterator

**CONSEQUENCES:**

* Iterator supports variations of traversal of the same Collection (depth-first, breadth-first traversal of Trees for exp).
* Iterators simplify the Collection interface.
* Iterators permit more than one traversal to operate simultaneously on the Collection - this is fine for read-only traversal but can be dangerous if there are writes or removals.

**IMPLEMENTATION:**

**Step 1:**

Iterator.java

public interface Iterator {

public boolean hasNext();

public Object next();

}

Container.java

public interface Container {

public Iterator getIterator();

}

**Step 2:**

NameRepository.java

public class NameRepository implements Container {

public String names[] = {"Robert" , "John" ,"Julie" , "Lora"};

@Override

public Iterator getIterator() {

return new NameIterator();

}

private class NameIterator implements Iterator {

int index;

@Override

public boolean hasNext() {

if(index < names.length){

return true;

}

return false;

}

@Override

public Object next() {

if(this.hasNext()){

return names[index++];

}

return null;

}

}

}

**Step 3:**

IteratorPatternDemo.java

public class IteratorPatternDemo {

public static void main(String[] args) {

NameRepository namesRepository = new NameRepository();

for(Iterator iter = namesRepository.getIterator(); iter.hasNext();){

String name = (String)iter.next();

System.out.println("Name : " + name);

}

}

}

**KNOWN USERS:**

* Iterators are common in object-oriented systems. Most collection class libraries offer iterators in one form or another.
* Iterators don't have to be defined as explicitly in Smalltalk. The standard collection classes (Bag, Set, Dictionary, OrderedCollection, String, etc.) define an internal iterator method do: which takes a block (i.e., closure) as an argument. Each element in the collection is bound to the local variable in the block; then the block is executed.
* Polymorphic iterators and the cleanup Proxy described earlier are provided by the ET++ container.

**RELATED PATTERN:**

* Composite - uses an Iterator to hide complexities and varities of methods of traversal Factory Method used tocreate the particular Iterator for a specific ConcreteCollection subclass.
* Memento is often used by an Iterator to keep track of the state of the traversal such as tree level and ancestor Stack during Tree traversal for example Visitor uses an Iterator to traverse the Collection of nodes it visits.

**MEDIATOR DESIGN PATTERN:**

**NAME:**

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".

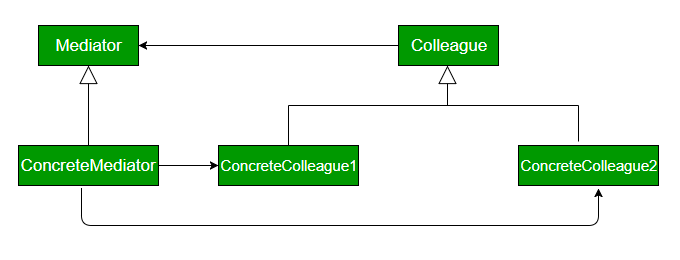
**MOTIVATION:**

* In order to have a good object oriented design we have to create lots of classes interacting one with each other. If certain principles are not applied the final framework will end in a total mess where each object relies on many other objects in order to run. In order to avoid tight coupled frameworks, we need a mechanism to facilitate the interaction between objects in a manner in that objects are not aware of the existence of other objects.
* Let's take the example of a screen. When we create it we add all sort of controls to the screen. This control need to interact with all the other control. For example when a button is pressed it must know if the data is valid in other controls. As you have seen if you created different applications using forms you don't have to modify each control class each time you add a new control to the form. All the operations between controls are managed by the form class itself. This class is called mediator.

**APPLICABILITY:**

* The Mediator Pattern is a good choice if we have to deal with a set of objects that are tightly coupled and hard to maintain. This way we can reduce the dependencies between objects and decrease the overall complexity.
* Additionally, by using the mediator object, we extract the communication logic to the single component, therefore we follow the Single Responsibility Principle. Furthermore, we can introduce new mediators with no need to change the remaining parts of the system. Hence, we follow the Open-Closed Principle.
* Sometimes, however, we may have too many tightly coupled objects due to the faulty design of the system. If this is a case, we should not apply the Mediator Pattern. Instead, we should take one step back and rethink the way we've modeled our classes.
* As with all other patterns, we need to consider our specific use case before blindly implementing the Mediator Pattern.

**STRUCTURE:**



**PARTICIPANTS:**

* 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.

**CONSEQUENCES:**

* Comprehension - The mediator encapsulate the logic of mediation between the colleagues. From this reason it' more easier to understand this logic since it is kept in only one class.
* Decoupled Colleagues - The colleague classes are totally decoupled. Adding a new colleague class is very easy due to this decoupling level.
* Simplified object protocols - The colleague objects need to communicate only with the mediator objects. Practically the mediator pattern reduce the required communication channels(protocols) from many to many to one to many and many to one.
* Limits Subclassing - Because the entire communication logic is encapsulated by the mediator class, when this logic need to be extended only the mediator class need to be extended.
* Complexity - in practice the mediators tends to become more complex and complex. A good practice is to take care to make the mediator classes responsible only for the communication part. For example when implementing different screens the the screen class should not contain code which is not a part of the screen operations. It should be put in some other classes.

**IMPLEMENTATION:**

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();

}

}

**KNOWN USERS:**

* In the following section, we'll discuss some real-world uses of the Mediator pattern. You'll find the Mediator in many situations where there are many components that must interact with one another in complex ways. User Interfaces.
* Maybe the mediator pattern is mostly used in the user interfaces. Almost any GUI framework is build around it. Like discussed before, the classes representing forms (Dialog, Form,... ) represents the the mediator while each control represents a colleague. The form class provides the mechanism to facilitate the interaction between controls; an inherited class is created each time a new screen is created and the specific code is written in this class. This way, the controls communication is mediated by this form class. Java Message Service
* The Java Message Service (JMS) API is a Java Message Oriented Middleware (MOM) API for sending messages between two or more clients. The JMS API supports 2 models. One of them is the publish-subscribe model. It is an implementation of the mediator pattern. The messages can be publisehd based on a particular topic. The publisher has to create a subscription to which different subscribers may subscribe. Zero or more subscribers may subscribe to receive messages on a particular message topic. The publisher and the subscriber doesn't know one about eachother, the subscriber can be even inactive. In this case the subscriber receives the messages when it will become active

**RELATED PATTERN:**

* Chain of Responsibility, Command, Mediator and Observer address various ways of connecting senders and receivers of requests.
* Facade and Mediator have similar jobs: they try to organize collaboration between lots of tightly coupled classes.
* The difference between Mediator and Observer is often elusive. In most cases, you can implement either of these patterns; but sometimes you can apply both simultaneously. Let’s see how we can do that.