**TUTORIAL NO. 06**

**AIM:** Consider expression evaluation. Implement expression evaluation using composite design pattern [Consider basic arithmetic operation].

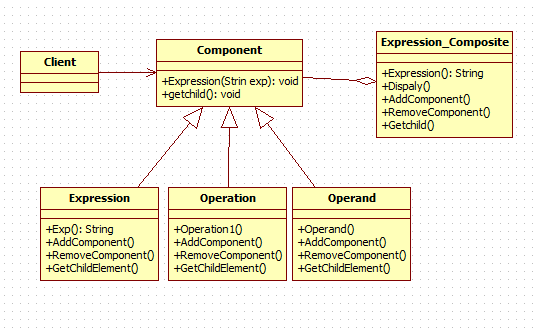
**THEORY:**

* **Design assumptions and Design pattern:**
  1. Composite pattern is a partitioning design pattern and describes a group of objects that is treated the same way as a single instance of the same type of object. The intent of a composite is to “compose” objects into tree structures to represent part-whole hierarchies. It allows you to have a tree structure and ask each node in the tree structure to perform a task.
  2. When dealing with Tree-structured data, programmers often have to discriminate between a leaf-node and a branch. This makes code more complex, and therefore, error prone. The solution is an interface that allows treating complex and primitive objects uniformly.
  3. In object-oriented programming, a composite is an object designed as a composition of one-or-more similar objects, all exhibiting similar functionality. This is known as a “has-a” relationship between objects.
  4. The key concept is that you can manipulate a single instance of the object just as you would manipulate a group of them. The operations you can perform on all the composite objects often have a least common denominator relationship.
  5. **The Composite Pattern has four participants:**
* **Component** – Component declares the interface for objects in the composition and for accessing and managing its child components. It also implements default behavior for the interface common to all classes as appropriate.
* **Leaf** – Leaf defines behavior for primitive objects in the composition. It represents leaf objects in the composition.
* **Composite** – Composite stores child components and implements child related operations in the component interface.
* **Client** –
  1. Client manipulates the objects in the composition through the component interface. Client use the component class interface to interact with objects in the composition structure.
  2. If recipient is a leaf then request is handled directly. If recipient is a composite, then it usually forwards request to its child components, possibly performing additional operations before and after forwarding.

**When to use Composite Design Pattern?**

1. Composite Pattern should be used when clients need to ignore the difference between compositions of objects and individual objects.
2. If programmers find that they are using multiple objects in the same way, and often have nearly identical code to handle each of them, then composite is a good choice, it is less complex in this situation to treat primitives and composites as homogeneous.
   1. Less number of objects reduces the memory usage, and it manages to keep us away from errors related to memory like java.lang.OutOfMemoryError.
   2. Although creating an object in Java is really fast, we can still reduce the execution time of our program by sharing objects.

* **UML Design Architecture:**



* **CODE:**

**Interface Component.java**

|  |
| --- |
| public interface Calculator  {      public void showCalculatorDetails();  } |

**Leaf1.java**

|  |
| --- |
| public class expression implements Calculator  {      private String expression;        public expression (String expression)      {          this. expression = expression;      }        @Override      public void showCalculatorDetails()      {          System.out.println(expression);      }  }  Leaf2.java |

public class operation implements Calculator

{

    private String operation;

    public operation(String name)

    {

        this.operation= operation;

    }

    @Override

    public void showCalculatorDetails()

    {

        System.out.println(operation);

    }

}

**Composite.java**

|  |
| --- |
| import java.util.ArrayList;  import java.util.List;    public class CalculatorDirectory implements Calculator  {      private List<Employee> CalculatorList = new ArrayList<Calculator>();        @Override      public void showCalculatorDetails()      {          for(Calculator cal: Calculator List)          {              emp.showCalculatorDetails();          }      }        public void addCalculator (Calculator cal)      {          CalculatorList.add(cal);      }        public void removeCalculator (Calculator cal)      {          CalculatorList.remove(cal);      }  } |

**Client.java**

|  |
| --- |
| public class Calculator  {      public static void main (String[] args)      {          expression exp1 = new expression(“(8\*9)+9);          expression exp2 = new expression(“(9/3)-2”);          CalculatorDirectory calDirectory = new CalculatorDirectory();          calDirectory. addCalculator (cal1);          calDirectory. addCalculator (cal2);            operation op1 = new operation (“\*”,”+”);  operation op2 = new operation (“/”,”-“);    CalculatorDirectory accDirectory = new CalculatorDirectory ();          acc CalculatorDirectory.add Calculator(op1);          acc CalculatorDirectory.add Calculator(op2);        }  } |

* **OBERVATION:**

Using composite design pattern is basically work as partitioning pattern. It partitions the group of object and treated it as same way. Composite pattern provides an easy way to build application.