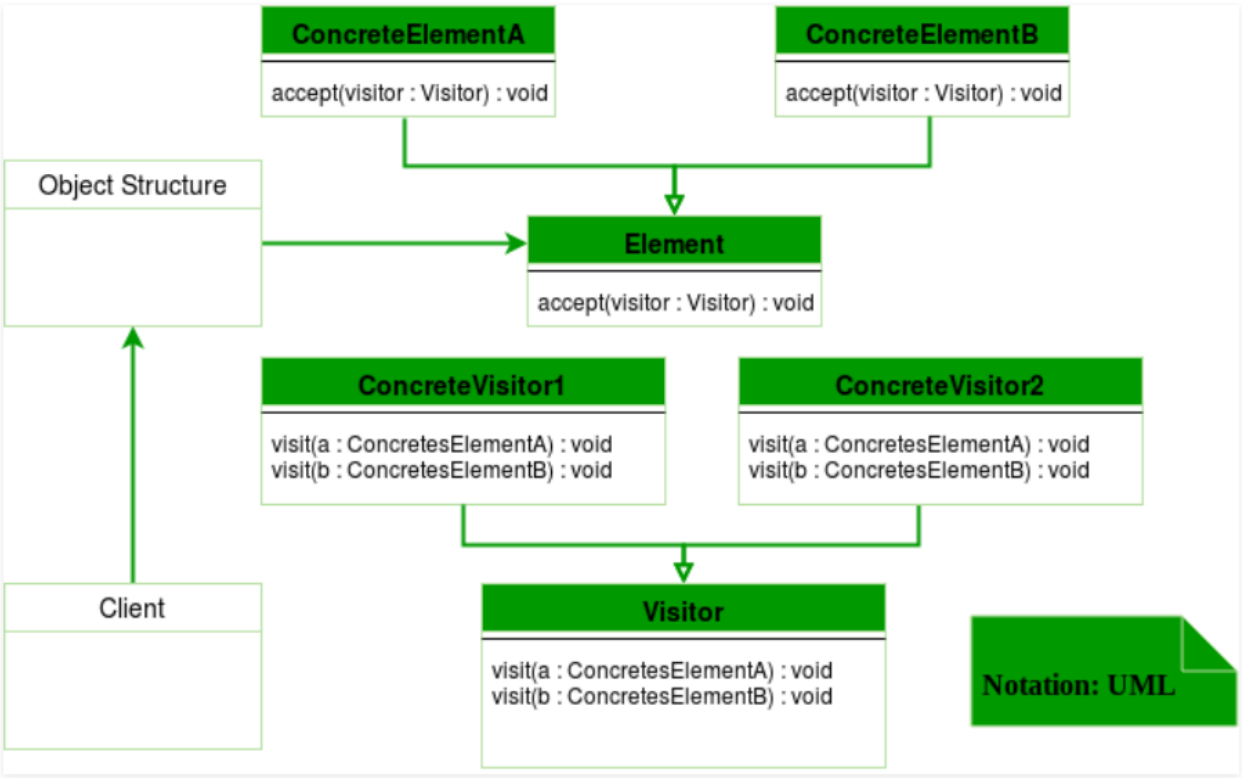
Visitor design pattern

Visitor design pattern is one of the behavioral design patterns. It is used when we have to perform an operation on a group of similar kind of Objects. With the help of visitor pattern, we can move the operational logic from the objects to another class.

The visitor pattern consists of two parts:

- A method called Visit() which is implemented by the visitor and is called for every element in the data structure.
- Visitable classes providing Accept() methods that accept a visitor.

UML Diagram Visitor design pattern



Design components

- **Client:** The Client class is a consumer of the classes of the visitor design pattern. It has access to the data structure objects and can instruct them to accept a Visitor to perform the appropriate processing.
- **Visitor:** This is an interface or an abstract class used to declare the visit operations for all the types of visitable classes.
- **ConcreteVisitor:** For each type of visitor all the visit methods, declared in abstract visitor, must be implemented. Each Visitor will be responsible for different operations.
- **Visitable:** This is an interface which declares the accept operation. This is the entry point which enables an object to be “visited” by the visitor object.
- **ConcreteVisitable:** These classes implement the Visitable interface or class and defines the accept operation. The visitor object is passed to this object using the accept operation.

Let's see an example of Visitor design pattern in Java.

```
interface ItemElement
{
    public int accept(ShoppingCartVisitor visitor);
}
```

```
class Book implements ItemElement
{
    private int price;
    private String isbnNumber;

    public Book(int cost, String isbn)
    {
        this.price=cost;
        this.isbnNumber=isbn;
    }

    public int getPrice()
    {
        return price;
    }

    public String getIsbnNumber()
    {
        return isbnNumber;
    }

    @Override
    public int accept(ShoppingCartVisitor visitor)
    {
        return visitor.visit(this);
    }

}
```

```
class Fruit implements ItemElement
{
    private int pricePerKg;
    private int weight;
    private String name;

    public Fruit(int priceKg, int wt, String nm)
    {
        this.pricePerKg=priceKg;
        this.weight=wt;
        this.name = nm;
    }

    public int getPricePerKg()
    {
        return pricePerKg;
    }

    public int getWeight()
    {
        return weight;
    }

    public String getName()
    {
        return this.name;
    }

    @Override
    public int accept(ShoppingCartVisitor visitor)
    {
        return visitor.visit(this);
    }

}
```

```
interface ShoppingCartVisitor
{

    int visit(Book book);
    int visit(Fruit fruit);
}
```

```
class ShoppingCartVisitorImpl implements ShoppingCartVisitor
{

    @Override
    public int visit(Book book)
    {
        int cost=0;
        //apply 5$ discount if book price is greater than 50
        if(book.getPrice() > 50)
        {
            cost = book.getPrice()-5;
        }
        else
            cost = book.getPrice();

        System.out.println("Book ISBN::"+book.getIsbnNumber() + " cost ="+cost);
        return cost;
    }

    @Override
    public int visit(Fruit fruit)
    {
        int cost = fruit.getPricePerKg()*fruit.getWeight();
        System.out.println(fruit.getName() + " cost = "+cost);
        return cost;
    }

}
```

```
class ShoppingCartClient
{

    public static void main(String[] args)
    {
        ItemElement[] items = new ItemElement[]{new Book(20, "1234"),
                                                new Book(100, "5678"), new Fruit(10, 2, "Banana"),
                                                new Fruit(5, 5, "Apple")};

        int total = calculatePrice(items);
        System.out.println("Total Cost = "+total);
    }

    private static int calculatePrice(ItemElement[] items)
    {
        ShoppingCartVisitor visitor = new ShoppingCartVisitorImpl();
        int sum=0;
        for(ItemElement item : items)
        {
            sum = sum + item.accept(visitor);
        }
        return sum;
    }

}
```

Output:

```
Book ISBN::1234 cost =20
Book ISBN::5678 cost =95
Banana cost = 20
Apple cost = 25
Total Cost = 160
```

Here, in the implementation if accept() method in all the items are same but it can be different. For example, there can be logic to check if item is free then don't call the visit () method at all.

Advantages:

- If the logic of operation changes, then we need to make change only in the visitor implementation rather than doing it in all the item classes.
- Adding a new item to the system is easy, it will require change only in visitor interface and implementation and existing item classes will not be affected.

Disadvantages:

- We should know the return type of visit () methods at the time of designing otherwise we will have to change the interface and all of its implementations.
- If there are too many implementations of visitor interface, it makes it hard to extend.

Prototype Design Pattern

Prototype design pattern is one of the Creational Design pattern, so it provides a mechanism of object creation.

Prototype design pattern is used when the Object creation is a costly affair and requires a lot of time and resources and you have a similar object already existing.

Prototype pattern provides a mechanism to copy the original object to a new object and then modify it according to our needs. Prototype design pattern uses java cloning to copy the object.

Prototype Design Pattern Example

It would be easy to understand prototype design pattern with an example. Suppose we have an Object that loads data from database. Now we need to modify this data in our program multiple times, so it's not a good idea to create the Object using **new** keyword and load all the data again from database.

The better approach would be to clone the existing object into a new object and then do the data manipulation.

Prototype design pattern mandates that the Object which you are copying should provide the copying feature. It should not be done by any other class. However, whether to use shallow or deep copy of the Object properties depends on the requirements and it's a design decision.

Here is a sample program showing Prototype design pattern example in java.

Employees.java

```
public class Employees implements Cloneable {

    private List<String> empList;

    public Employees(){
        empList = new ArrayList<String>();
    }

    public void loadData(){
        //read all employees from database and put into the list
        empList.add("Pankaj");
        empList.add("Raj");
        empList.add("David");
        empList.add("Lisa");
    }

    public List<String> getEmpList() {
        return empList;
    }

    @Override
    public Object clone()
    {
        Employees cloneEmp = null;
        try
        {
            cloneEmp = (Employees)super.clone();

            List<String> tempList = new ArrayList<String>();
            for(String s : this.empList){
                tempList.add(s);
            }

            cloneEmp.empList = tempList;
        }
         catch (CloneNotSupportedException e)
         {
             e.printStackTrace();
         }
         return cloneEmp;
     }
}
```

Notice that the **clone** method is overridden to provide a deep copy of the employees list.

Here is the prototype design pattern example test program that will show the benefit of prototype pattern.

PrototypePatternTest.java

```
public class PrototypePatternTest {

    public static void main(String[] args) throws CloneNotSupportedException {
        Employees emps = new Employees();
        emps.loadData();

        //Use the clone method to get the Employee object
        Employees empsNew = (Employees) emps.clone();
        Employees empsNew1 = (Employees) emps.clone();
        List<String> list = empsNew.getEmpList();
        list.add("John");
        List<String> list1 = empsNew1.getEmpList();
        list1.remove("Pankaj");

        System.out.println("emps List: "+emps.getEmpList());
        System.out.println("empsNew List: "+list);
        System.out.println("empsNew1 List: "+list1);
    }

}
```

Output of the above prototype design pattern example program is:

```
emps List: [Pankaj, Raj, David, Lisa]
empsNew List: [Pankaj, Raj, David, Lisa, John]
empsNew1 List: [Raj, David, Lisa]
```

If the object cloning was not provided, we will have to make database call to fetch the employee list every time. Then do the manipulations that would have been resource and time consuming.