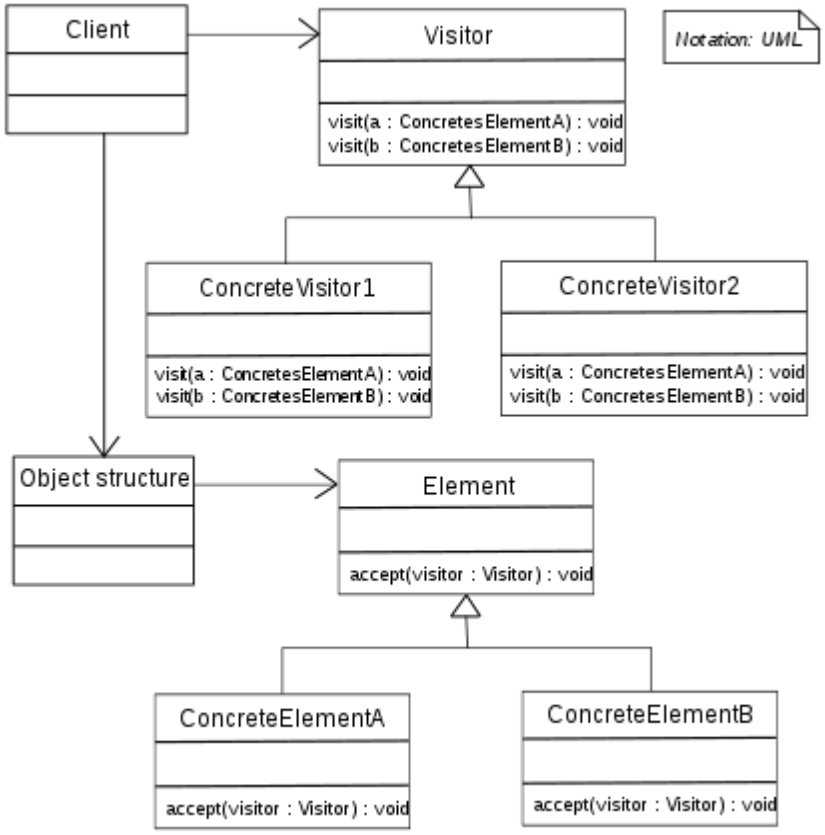
VISITOR PATTERN

In Visitor pattern, we use a visitor class which changes the executing algorithm of an element class. By this way, execution algorithm of element can vary as and when visitor varies. This pattern comes under behavior pattern category. As per the pattern, element object has to accept the visitor object so that visitor object handles the operation on the element object. 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 object structure
- visitable classes providing Accept() methods that accept a visitor



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 :** 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 :** Those classes implements the Visitable interface or class and defines the accept operation. The visitor object is passed to this object using the accept operation.

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.

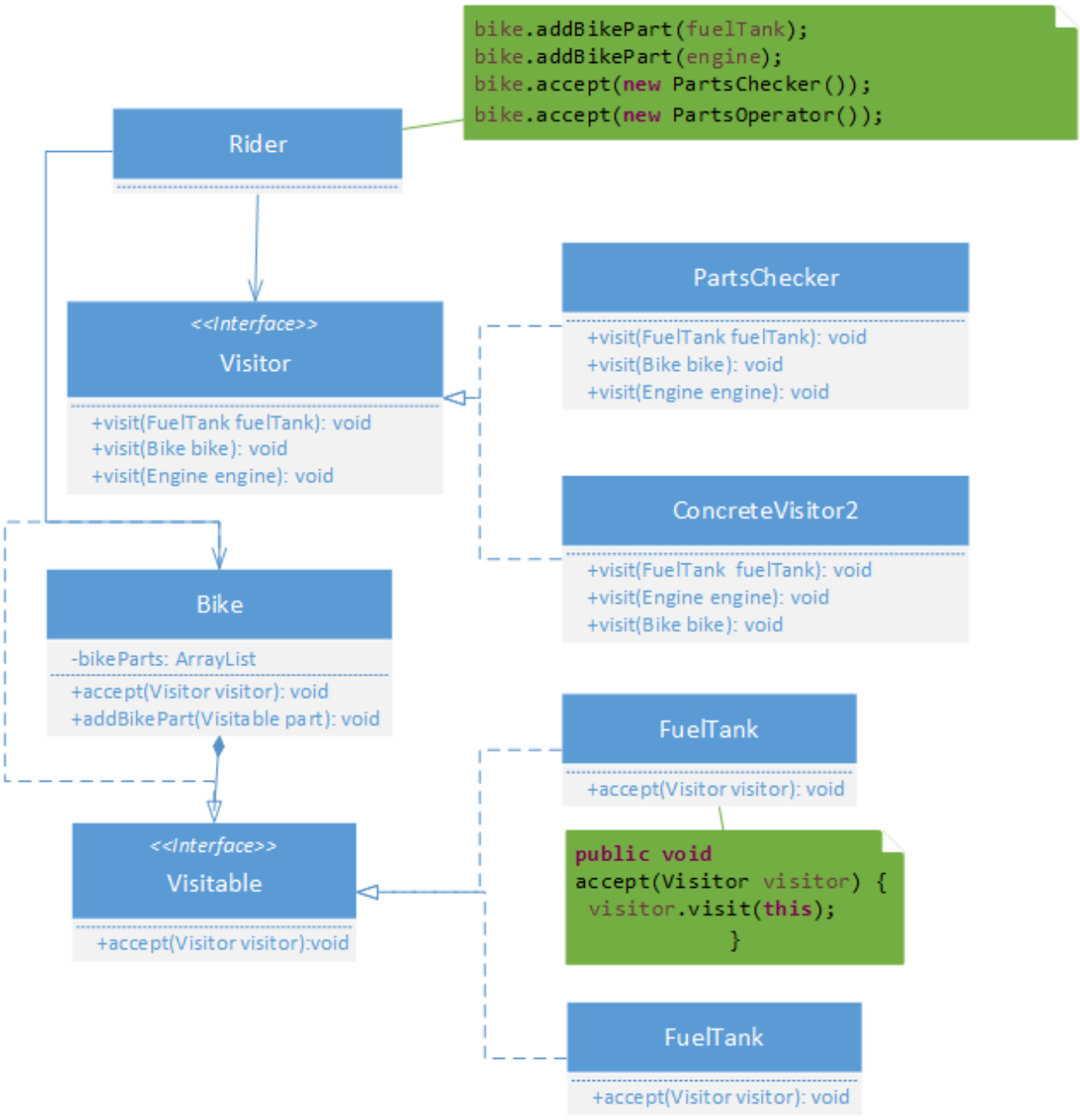
Disadvantages :

- 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.

Example:

Consider a scenario where we have a **Bike** and **Engine** and **FuelTank** are parts of **Bike.** We have two types of visitors, i.e. **PartsChecker** which checks whether all bike parts are working fine, and **PartsOperator** which operates all bike parts. We will write visitor design pattern implementation as below.

- **Bike, Engine and FuelTank** implement **Visitable,** and implement **accept(Visitor visitor);**
- **PartsChecker** and **PartsOperator** implement **visit()** methods.


bike.addBikePart(fuelTank);
bike.addBikePart(engine);
bike.accept(new PartsChecker());
bike.accept(new PartsOperator());
Rider
<<Interface>>
Visitor
+visit(FuelTank fuelTank): void
+visit(Bike bike): void
+visit(Engine engine): void
PartsChecker
+visit(FuelTank fuelTank): void
+visit(Bike bike): void
+visit(Engine engine): void
ConcreteVisitor2
+visit(FuelTank fuelTank): void
+visit(Engine engine): void
+visit(Bike bike): void
Bike
-bikeParts: ArrayList
+accept(Visitor visitor): void
+addBikePart(Visitable part): void
<<Interface>>
Visitable
+accept(Visitor visitor):void
FuelTank
+accept(Visitor visitor): void
public void
accept(Visitor visitor) {
visitor.visit(this);
}
FuelTank
+accept(Visitor visitor): void

Visitor.java

```
public interface Visitor {
   public abstract void visit(FuelTank fuelTank);
   public abstract void visit(Engine engine);
   public abstract void visit(Bike bike);
}
```

PartChecker.java

```
public class PartsChecker implements Visitor {

   @Override
   public void visit(FuelTank fuelTank) {
      System.out.println( "Checking whether there is fuel in
fuel tank");
   }

   @Override
   public void visit(Engine engine) {
      System.out.println( "Checking whether ignition switch is
on");
   }

   @Override
   public void visit(Bike bike) {
      System.out.println("Going to the bike");

   }
```

PartsOperaor.java

```
public class PartsOperator implements Visitor {

   @Override
   public void visit(FuelTank fuelTank) {
      System.out.println("Releasing fuel from fuel tank to
Engine");
   }

   @Override
   public void visit(Engine engine) {
      System.out.println("Accepting fuel from fuel tank and
running engine");
   }
```

```
    @Override
    public void visit(Bike bike) {
        System.out.println("Now going to start bike");

    }

}
```

Visitable.java

```
public interface Visitable {
   public abstract void accept(Visitor visitor);
}
```

FuelTank.java

```
public class FuelTank implements Visitable {

  public void accept(Visitor visitor) {
    visitor.visit(this);
  }
}
```

Engine.java

```
public class Engine implements Visitable {

  @Override
  public void accept(Visitor visitor) {
    visitor.visit(this);
  }

}
```

Bike.java

```
import java.util.ArrayList;
import java.util.List;

public class Bike implements Visitable {

  private List<Visitable> bikeParts = new ArrayList<Visitable>();
```

```
  @Override
  public void accept(Visitor visitor) {

    visitor.visit(this);
    for(Visitable part : bikeParts){
      part.accept(visitor);
    }

  }

  public void addBikePart(Visitable part) {
    bikeParts.add(part);
  }

}
```

Ride.java

```
public class Rider {

  public static void main(String args[]) {
    Bike bike = new Bike();
    Visitable engine = new Engine();
    Visitable fuelTank = new FuelTank();

    bike.addBikePart(fuelTank);
    bike.addBikePart(engine);

    bike.accept(new PartsChecker());
    bike.accept(new PartsOperator());

  }
}
```

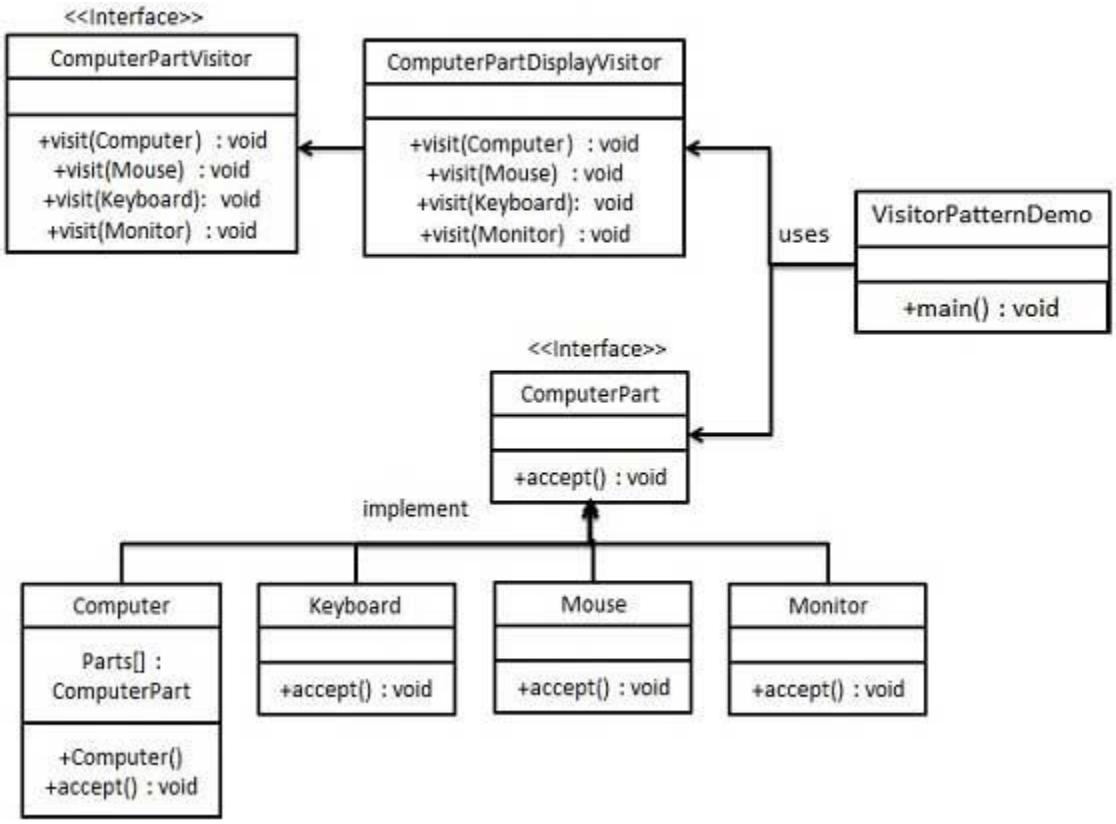
Output

```
     Going to the bike
Checking whether there is fuel in fuel tank
Checking whether ignition switch is on
ow going to start bike
Releasing fuel from fuel tank to Engine
Accepting fuel from fuel tank and running engine
```

Another example of visitor pattern

We are going to create a *ComputerPart* interface defining accept opearation.*Keyboard*, *Mouse*, *Monitor* and *Computer* are concrete classes implementing *ComputerPart* interface. We will define another interface *ComputerPartVisitor* which will define a visitor class operations. *Computer*uses concrete visitor to do corresponding action.

VisitorPatternDemo, our demo class, will use *Computer* and *ComputerPartVisitor* classes to demonstrate use of visitor pattern.



Step 1

Define an interface to represent element.

ComputerPart.java

```
public interface ComputerPart {
   public void accept(ComputerPartVisitor computerPartVisitor);
}
```

Step 2

Create concrete classes extending the above class.

Keyboard.java

```
public class Keyboard implements ComputerPart {

   @Override
   public void accept(ComputerPartVisitor computerPartVisitor) {
      computerPartVisitor.visit(this);
   }
}
```

Monitor.java

```
public class Monitor implements ComputerPart {
   @Override
   public void accept(ComputerPartVisitor computerPartVisitor) {
      computerPartVisitor.visit(this);
   }
```

```
}
```

Mouse.java

```
public class Mouse implements ComputerPart {
   @Override
   public void accept(ComputerPartVisitor computerPartVisitor) {
      computerPartVisitor.visit(this);
   }
}
```

Computer.java

```
public class Computer implements ComputerPart {
   ComputerPart[] parts;
   public Computer(){
      parts = new ComputerPart[] {new Mouse(), new Keyboard(), new
Monitor()};
   }
   @Override
   public void accept(ComputerPartVisitor computerPartVisitor) {
      for (int i = 0; i < parts.length; i++) {
         parts[i].accept(computerPartVisitor);
      }
      computerPartVisitor.visit(this);
   }
}
```

Step 3

Define an interface to represent visitor.

ComputerPartVisitor.java

```
public interface ComputerPartVisitor {
        public void visit(Computer computer);
        public void visit(Mouse mouse);
        public void visit(Keyboard keyboard);
        public void visit(Monitor monitor);
}
```

Step 4

Create concrete visitor implementing the above class.

ComputerPartDisplayVisitor.java

```
public class ComputerPartDisplayVisitor implements ComputerPartVisitor {
  @Override
  public void visit(Computer computer) {
    System.out.println("Displaying Computer.");
  }
  @Override
  public void visit(Mouse mouse) {
    System.out.println("Displaying Mouse.");
  }
```

```
  @Override
  public void visit(Keyboard keyboard) {
    System.out.println("Displaying Keyboard.");
  }
  @Override
  public void visit(Monitor monitor) {
    System.out.println("Displaying Monitor.");
  }
}
```

Step 5

Use the *ComputerPartDisplayVisitor* to display parts of *Computer*.

VisitorPatternDemo.java

```
public class VisitorPatternDemo {
  public static void main(String[] args) {
    ComputerPart computer = new Computer();
    computer.accept(new ComputerPartDisplayVisitor());
  }
}
```

Step 6

Verify the output.

```
Displaying Mouse.
Displaying Keyboard.
Displaying Monitor.
Displaying Computer.
```