Visitor Pattern

The **visitor pattern** changes the executing algorithm of an element class. The new behaviour for the class is placed under a separate class (the visitor) instead of trying to integrate it into the existing classes. Since it deals with the behaviour of a class, the visitor pattern is a **behavioural pattern**.

Simply put, if you have a class that you don’t want to mess with, the visitor pattern will let you add new behaviour or modify existing behaviour without extensively modifying the original class.

Full disclosure, there is a very common misconception that the visitor pattern ‘visits’ things and is used similar to an iterator in that you iterate over a group of objects and just call methods on each object. That’s wrong. That’s not what the visitor pattern is supposed to be. The example below kind of does this too. This is the fault of the gang of four. They used a confusing name.

public interface *ComputerPart* {  
 public void accept(*ComputerPartVisitor* computerPartVisitor);  
}  
  
public class Keyboard implements *ComputerPart* {  
  
 @Override  
 public void accept(*ComputerPartVisitor* computerPartVisitor) {  
 computerPartVisitor.visit(this);  
 }  
}

public class Monitor implements *ComputerPart* {  
  
 @Override  
 public void accept(*ComputerPartVisitor* computerPartVisitor) {  
 computerPartVisitor.visit(this);  
 }  
}  
  
public class Mouse implements *ComputerPart* {  
  
 @Override  
 public void accept(*ComputerPartVisitor* computerPartVisitor) {  
 computerPartVisitor.visit(this);  
 }  
}  
  
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);  
 }  
}  
  
public interface *ComputerPartVisitor* {  
 public void visit(Computer computer);  
 public void visit(Mouse mouse);  
 public void visit(Keyboard keyboard);  
 public void visit(Monitor monitor);  
}

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

JAVA

You see how none of that makes any sense? We forced the entire *ComputerPart* hierarchy to implement this method with which they pass themselves to this new class called ComputerPartDisplayVisitor which in turn implements these methods which…display the computer part? There are a lot of examples like this one on the internet and they go all support this iterator style implementation of the code. Especially the Computer class which I personally found super confusing.

But consider this. We have this complex hierarchy which has a lot of functionality and we suddenly realize we need to add a new behaviour (the display behaviour). Since it’s a hierarchy, adding a new method to the main interface means we need to implement that in every single class. Lots of room for mistakes, especially if we end up doing this a lot. Instead, we can add this simple visit method to each class (which is literally copy and paste) and then get the new ComputerPartVisitor class to implement the new behaviour. Ignore the whole for loop going on with the Computer class. In fact, ignore the entire Computer class it’s nothing special and the for loop has nothing to do with the visitor pattern.

## Double Dispatch

The way we went about our objective was quite weird. Why not just make a class that has a bunch of methods for each of the classes and does something different in each method depending on the class? That DOES seem like what the *ComputerPartDisplayVisitor* class is doing. Why go through the trouble of making each of the computer part classes implement a method that takes the visitor as an argument, just to then go call the same method of the visitor everywhere.

The answer to this has to do with the concept of **Double Dispatching**. If we have a hierarchy (like the computer parts and the common interface they implement) we can fearlessly call one of the common methods they implement without knowing exactly which of the classes is being used. For example:

void printManufacturingDetails(*ComputerPart* computerPart) {  
 data = computerPart.getManufacturingDetails();  
 System.*out*.println(data);  
}

JAVA

Even though we don’t know which computer part we are working with, we can call this method because we know for certain that every computer part implements this method (since the method is defined in the interface). This process is called **dynamic binding** or **late binding**. The compiler sees that it cannot find the correct object immediately, but doesn’t complain. It will be done at runtime.

On the other hand, suppose we do this:

void displayComputerPart(*ComputerPart* computerPart) {  
 ComputerPartDisplayer displayer = new ComputerPartDisplayer();  
 displayer.display(computerPart);  
}

JAVA

The ComputerPartDisplayer class has to have an implementation for the specific type of computer part that we give it for this code to work. But we don’t know which type of computer part we are giving it, so there is no way to confirm that an equivalent implementation exists. To stay on the safe side, if there is a generic implementation (for any ComputerPart object), then the compiler will use that instead of using the implementation for a specific computer part. This is called **static binding**. This takes away several benefits of dynamic binding, mainly that the dynamic nature of the computerPart object in the above code is lost.

To get around this issue, we use the **double dispatch** method. This is what the visitor pattern is doing. Since a specific computer part class is passing ITSELF to the visitor, the compiler does know for a fact what type of object is being passed and therefore knows for a fact that an implementation for that type of object exists. Thus, there are no issues.

A more in-depth analysis of how double dispatch works and the problem it solves can be found [here](https://refactoring.guru/design-patterns/visitor-double-dispatch).