Visitor Pattern

Michal Moravik, SD20w2

Name: Visitor Pattern

Category: Behavioral

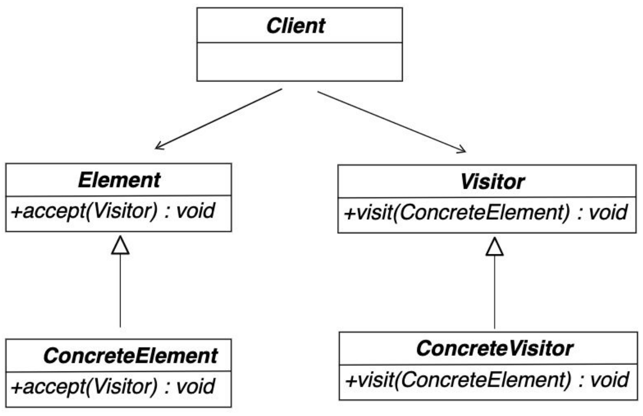
Intent: Let's you add a new behaviour to objects without changing their classes.

Motivation:
A "visitor" is an object which brings a new functionality to the objects without touching their classes. It is something like an extension. We can create as many of them as we want. This way, we can encapsulate new behaviour in a separate class. It is always beneficial to apply *single responsibility principle* and encapsulate new functionality in one class instead of forcing every class to implement a new behaviour which logically just does not belong there.

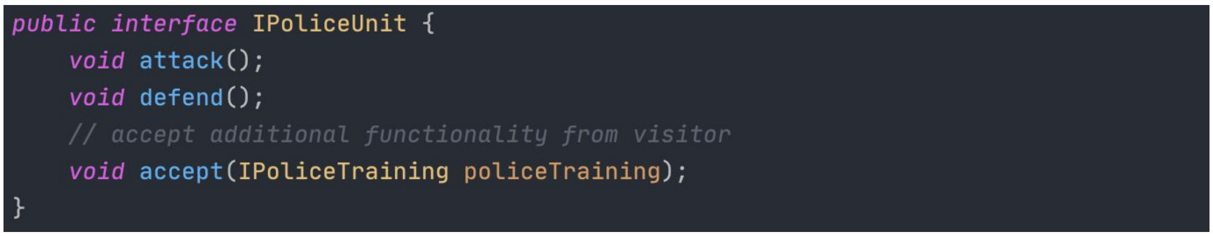
This behaviour can be even applied to multiple classes which avoid having this behaviour in each class separately. The Visitor Pattern uses "Double Dispatch" technique to execute this new behaviour.

Double dispatch:
A trick which places a method (e.g. "accept" method) inside of the classes we want to add the new behaviour to. This method calls the visitor's method and passes *this* as a parameter so we know exactly what object called this method. You can see the exact implementation below, look for a method called "accept" in different police units.

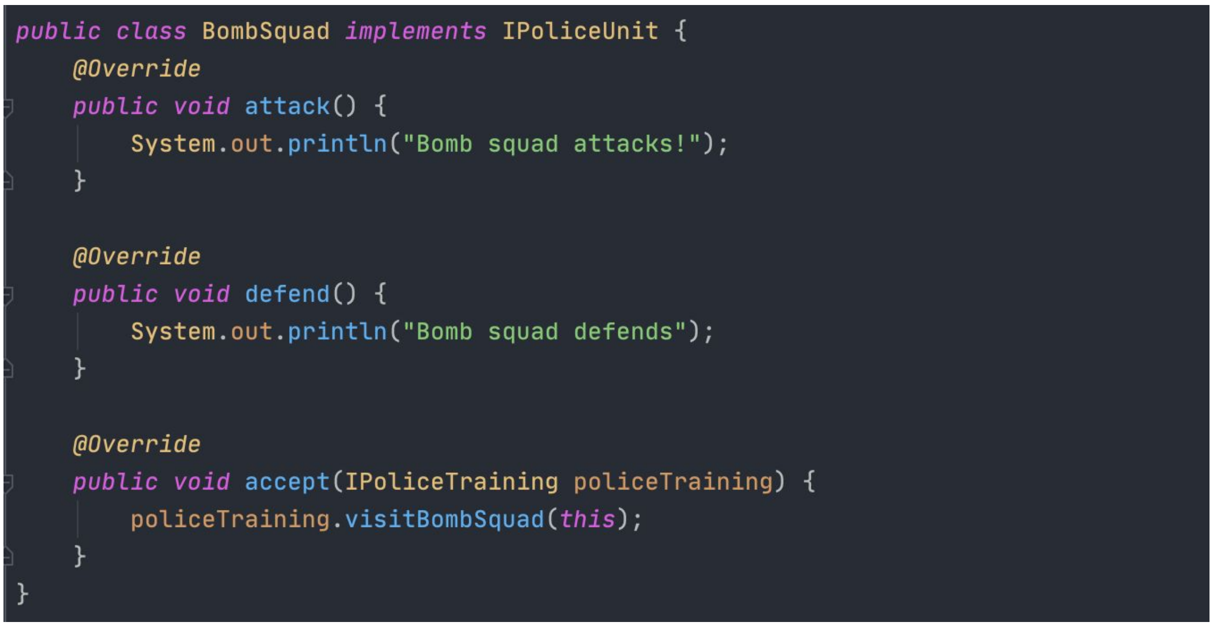
UML:



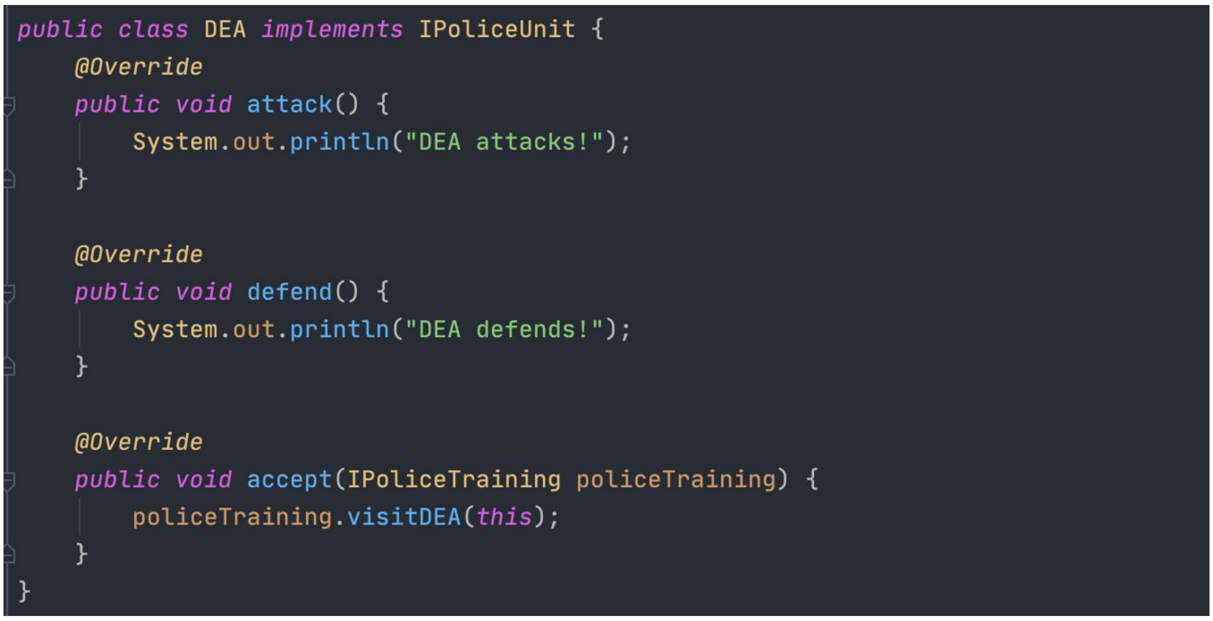
Implementation:



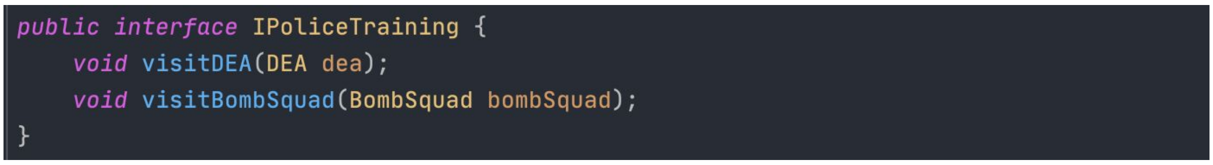
```java
public interface IPoliceUnit {
    void attack();
    void defend();
    // accept additional functionality from visitor
    void accept(IPoliceTraining policeTraining);
}
```



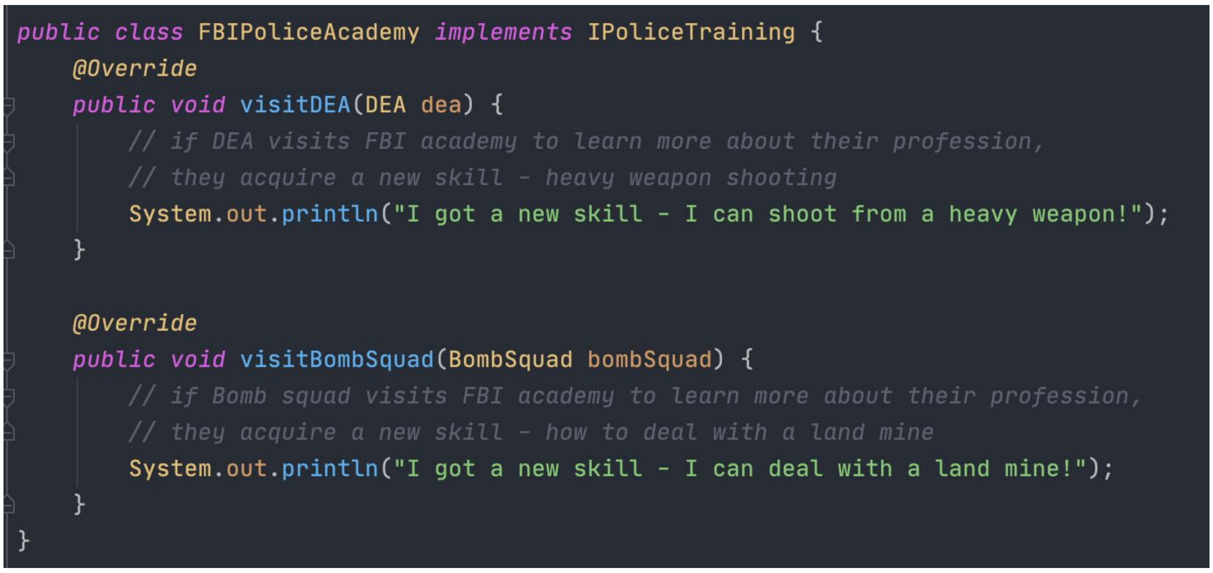
```java
public class BombSquad implements IPoliceUnit {
    @Override
    public void attack() {
        System.out.println("Bomb squad attacks!");
    }

    @Override
    public void defend() {
        System.out.println("Bomb squad defends");
    }

    @Override
    public void accept(IPoliceTraining policeTraining) {
        policeTraining.visitBombSquad(this);
    }
}
```



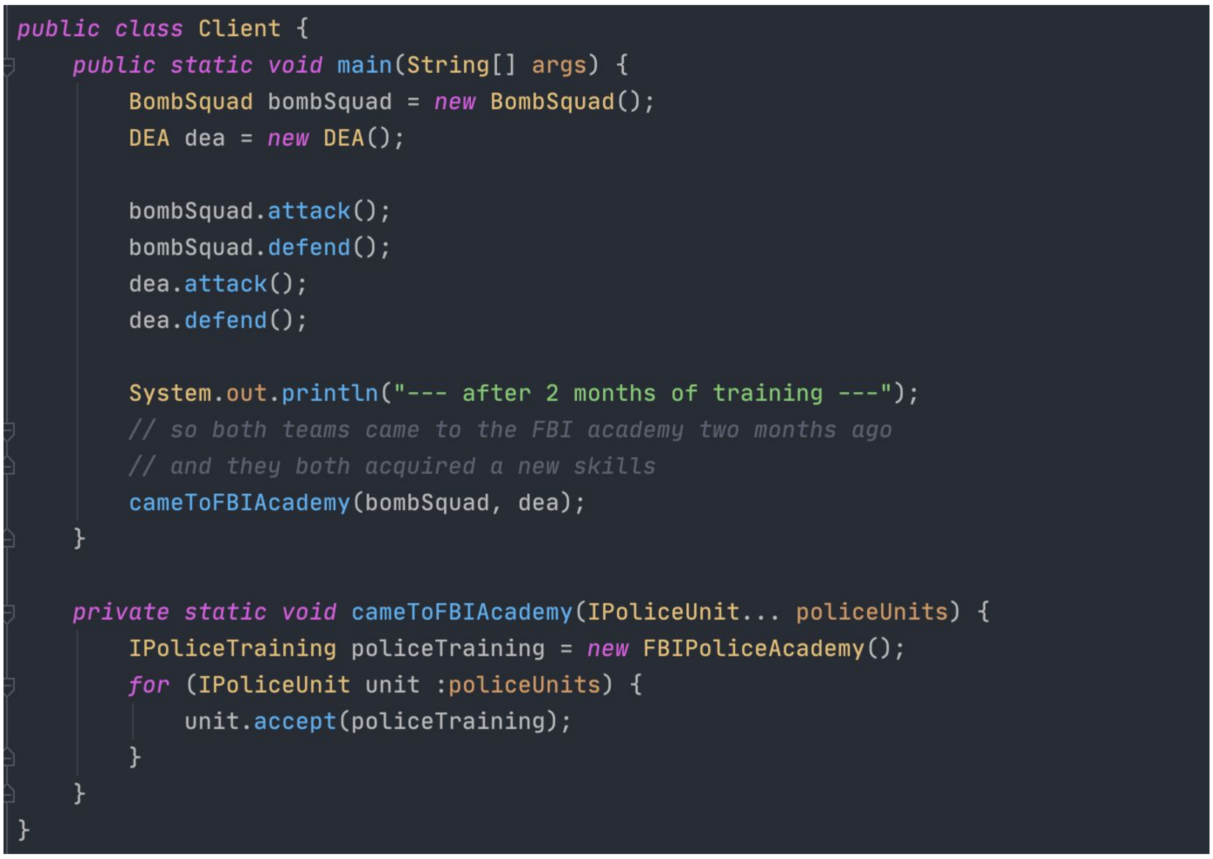
```java
public class DEA implements IPoliceUnit {
    @Override
    public void attack() {
        System.out.println("DEA attacks!");
    }

    @Override
    public void defend() {
        System.out.println("DEA defends!");
    }

    @Override
    public void accept(IPoliceTraining policeTraining) {
        policeTraining.visitDEA(this);
    }
}
```



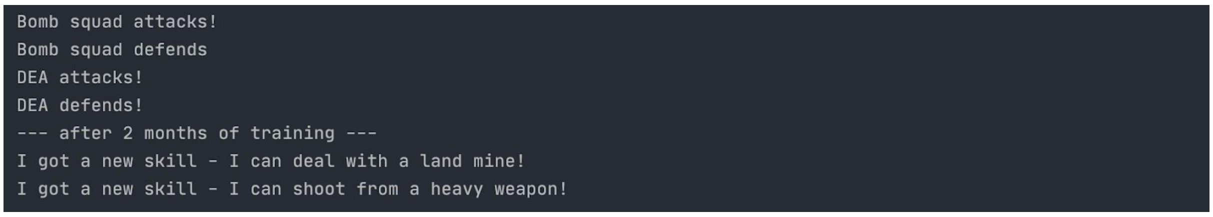
```java
public interface IPoliceTraining {
    void visitDEA(DEA dea);
    void visitBombSquad(BombSquad bombSquad);
}
```



```java
public class FBIPoliceAcademy implements IPoliceTraining {
    @Override
    public void visitDEA(DEA dea) {
        // if DEA visits FBI academy to learn more about their profession,
        // they acquire a new skill - heavy weapon shooting
        System.out.println("I got a new skill - I can shoot from a heavy weapon!");
    }

    @Override
    public void visitBombSquad(BombSquad bombSquad) {
        // if Bomb squad visits FBI academy to learn more about their profession,
        // they acquire a new skill - how to deal with a land mine
        System.out.println("I got a new skill - I can deal with a land mine!");
    }
}
```



```java
public class Client {
    public static void main(String[] args) {
        BombSquad bombSquad = new BombSquad();
        DEA dea = new DEA();

        bombSquad.attack();
        bombSquad.defend();
        dea.attack();
        dea.defend();

        System.out.println("--- after 2 months of training ---");
        // so both teams came to the FBI academy two months ago
        // and they both acquired a new skills
        cameToFBIAcademy(bombSquad, dea);
    }

    private static void cameToFBIAcademy(IPoliceUnit... policeUnits) {
        IPoliceTraining policeTraining = new FBIPoliceAcademy();
        for (IPoliceUnit unit :policeUnits) {
            unit.accept(policeTraining);
        }
    }
}
```



```
Bomb squad attacks!
Bomb squad defends
DEA attacks!
DEA defends!
--- after 2 months of training ---
I got a new skill - I can deal with a land mine!
I got a new skill - I can shoot from a heavy weapon!
```

- IPoliceTraining represents the visitor interface while FBIPoliceAcademy is a concrete visitor

- IPoliceUnit is a common interface for all police units, elements we want to add a new behaviour to
- In client, we can see how the specific accept methods are called based on the object which is passed inside

Consequences:
Pros:
- We can add various number of new behaviours to objects of various classes without changing those classes, the only thing we need is to add accept function to the Element's interface.
- You can encapsulate multiple versions of the same behaviour into the same class - single responsibility principle

Cons:
- You need to update all visitors each time a class gets added to or removed from the element hierarchy since the class is referenced in the visitor's interface
- Lacks access to the private fields and methods

Known uses:
- Use when you want to add logic to classes you do not want to change and instead you want to have this logic encapsulated in different classes

Related patterns:
- Can be used with iterator to traverse data structure while executing a new operation in them, even if they all have different classes