What is Strategy Pattern?

In computer programming, the strategy pattern (also known as the policy pattern) is a software design pattern that enables selecting an algorithm at runtime. Instead of implementing a single algorithm directly, code receives run-time instructions as to which in a family of algorithms to use.

Strategy pattern is used when we have multiple algorithm for a specific task and client decides the actual implementation to be used at runtime.

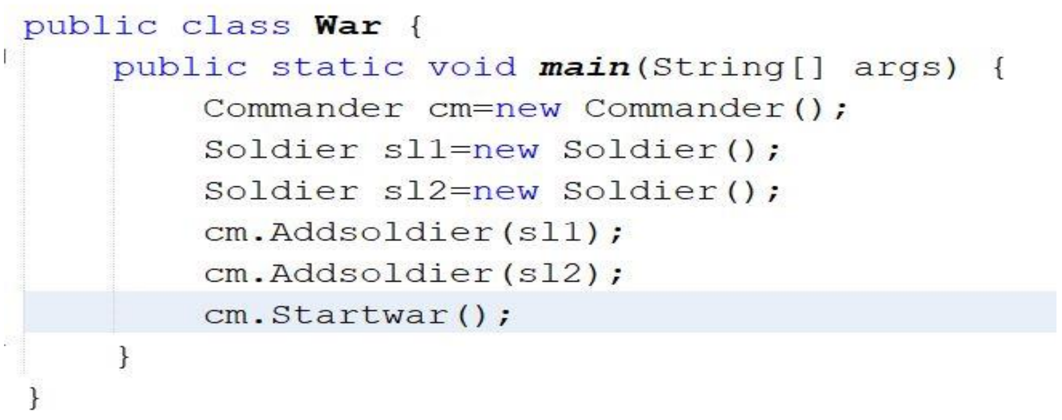
It provides a substitute to subclassing. It defines each behavior within its own class, eliminating the need for conditional statements.

Which pattern category it belongs?

Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

War Planning By Using Strategy Pattern:

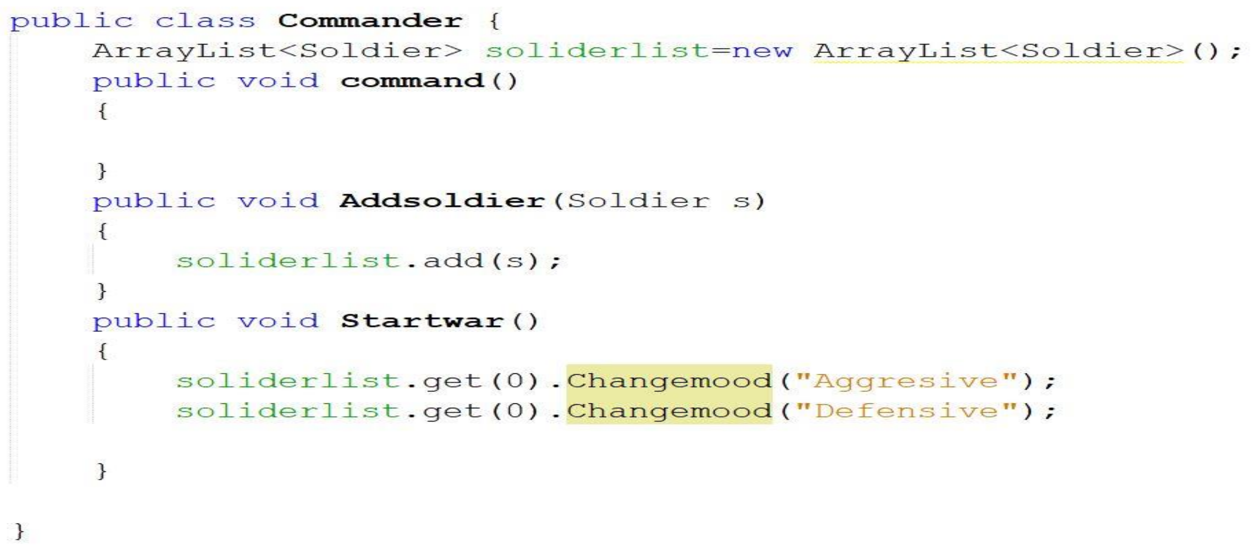
First of all we have to understand what happens when a war is being started in certain conditions, i.e. what features we have to focus on.
Actually, the war is called by a client. He commands the commander of the battalion under whom a huge number of soldiers are working all the time.
Now lets see client code implementation: The part of the code that controls the behavior of the other classes is known as the client code of those classes. As an instance,



```
public class War {
    public static void main(String[] args) {
        Commander cm=new Commander();
        Soldier sl1=new Soldier();
        Soldier sl2=new Soldier();
        cm.Addsoldier(sl1);
        cm.Addsoldier(sl2);
        cm.Startwar();
    }
}
```

So, we can see that this client code Main () function can control both commanders and soldiers. And client code is responsible for start the war.

Now lets see **commander** class implementation:



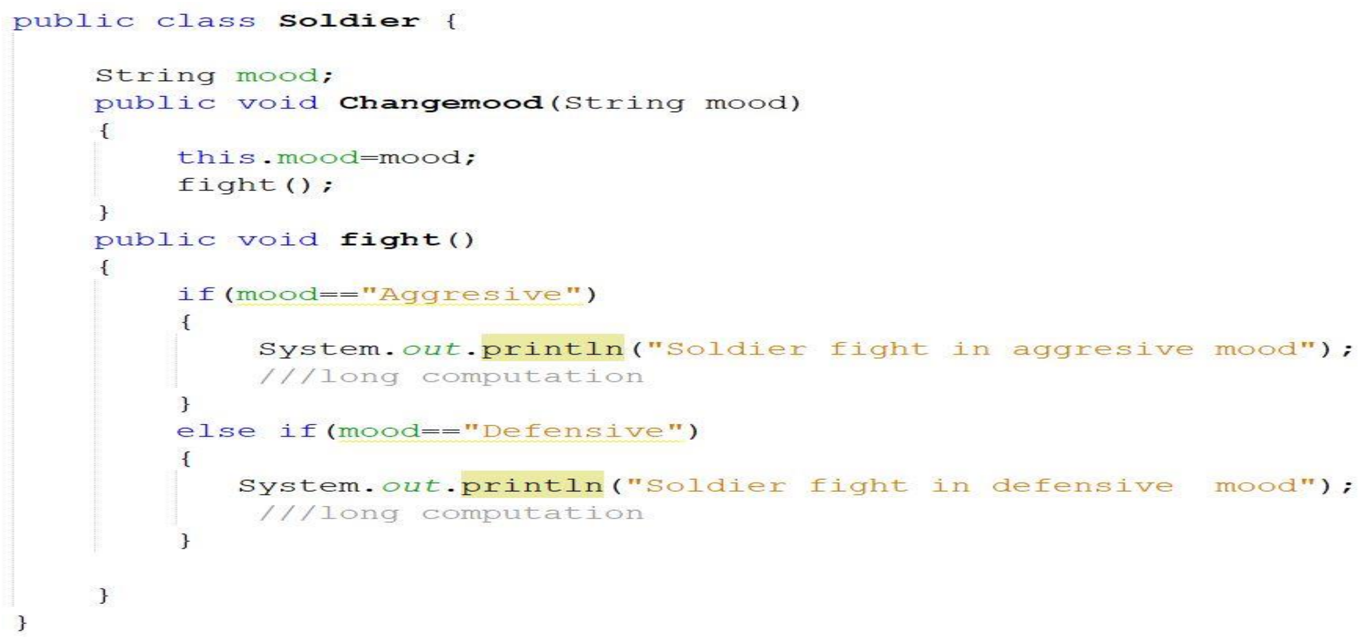
```
public class Commander {
    ArrayList<Soldier> soliderlist=new ArrayList<Soldier>();
    public void command()
    {

    }
    public void Addsoldier(Soldier s)
    {
        soliderlist.add(s);
    }
    public void Startwar()
    {
        soliderlist.get(0).Changemood("Aggresive");
        soliderlist.get(0).Changemood("Defensive");

    }

}
```

Now lets see **Soldier** class implementation:



```
public class Soldier {

    String mood;
    public void Changemood(String mood)
    {
        this.mood=mood;
        fight();
    }
    public void fight()
    {
        if(mood=="Aggresive")
        {
            System.out.println("Soldier fight in aggresive mood");
            ///long computation
        }
        else if(mood=="Defensive")
        {
           System.out.println("Soldier fight in defensive  mood");
            ///long computation
        }

    }
}
```

Problems: (Refactoring)

Now, we have to understand that, what are the problems that we may face in solving in the above implementation.

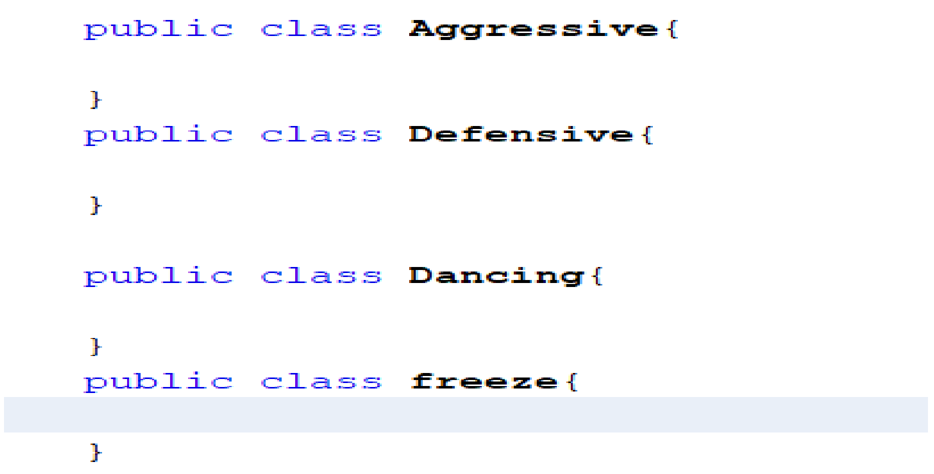
Problem 1:

If we try to add some more fighting modes like neutral, dancing, freeze etc then we have to add all such things in the same class soldier. Hence, the soldier class will eventually become huge which is not suitable for handling a good code.

Solution:

we separate the modes into distinguished classes. That means we implement all moods into separate classes. The coder of soldier will not be disturbed for any change in the fighting mode.

fighting mode



```
public class Aggressive{

}
public class Defensive{

}

public class Dancing{

}
public class freeze{

}
```

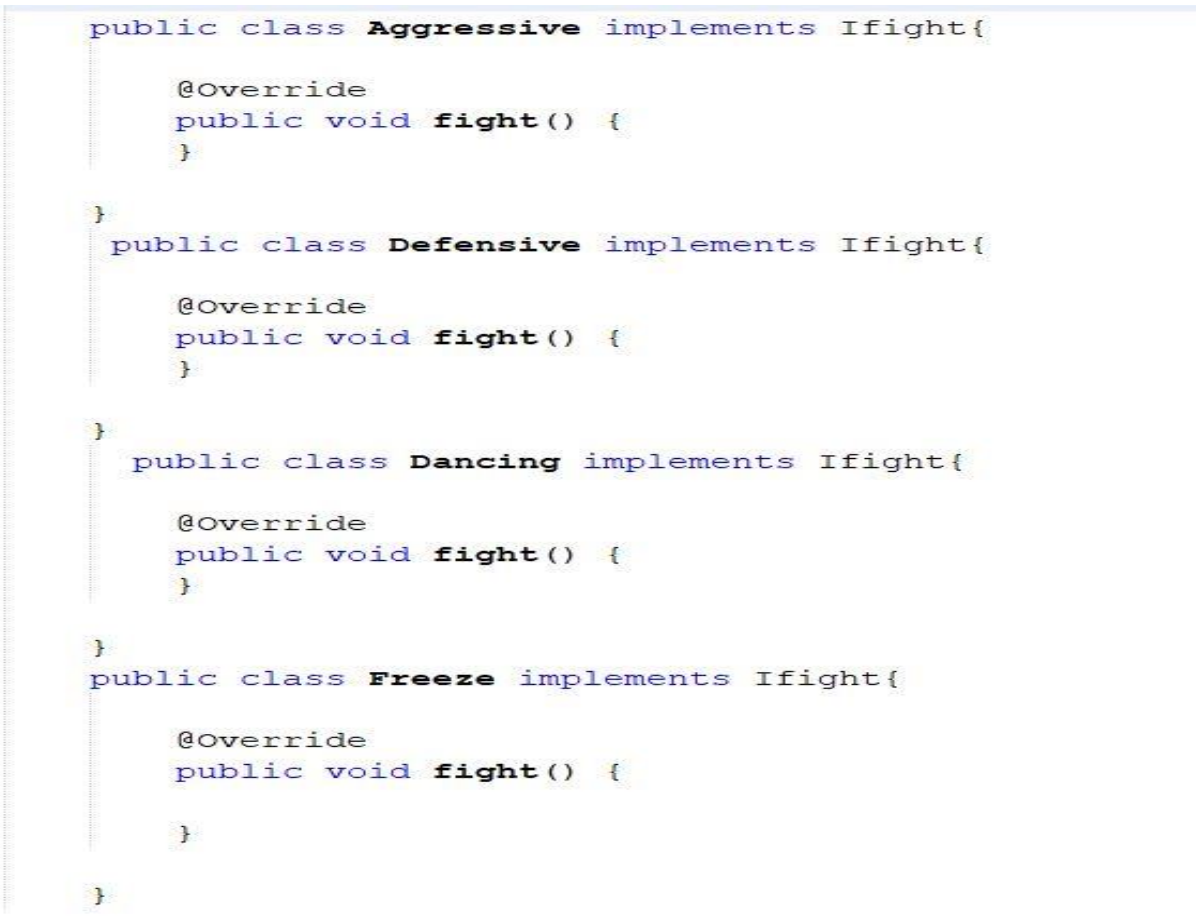
In this way, we can add different modes as much as possible whenever needed.

Problem 2:

If we separate the modes into distinguished classes, then another problem will arise, ambiguity.The modes will be described in three different functions by three different programmers who contributed in coding the codes of three different classes namely soldier, aggressive & defensive respectively.

Solution:

We have to implement a specific function & every class should be made bound to use that specific function. For this we can declare an interface & every class should implement that interface.



```
public class Aggressive implements Ifight{

    @Override
    public void fight() {
    }

}
 public class Defensive implements Ifight{

    @Override
    public void fight() {
    }

}
  public class Dancing implements Ifight{

    @Override
    public void fight() {
    }

}
public class Freeze implements Ifight{

    @Override
    public void fight() {

    }

}
```

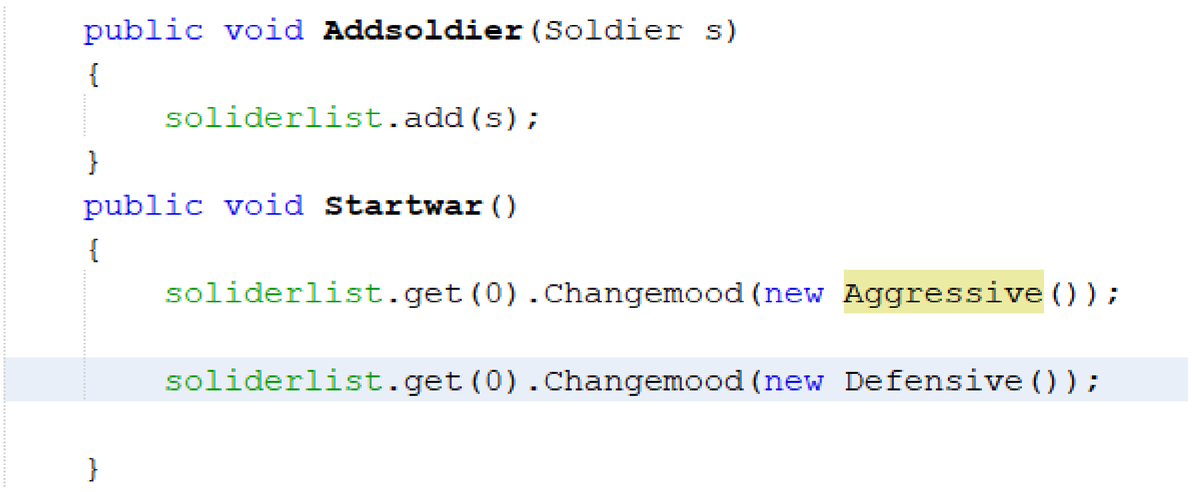
So, every class here is bound to use the function fight(). So, three different coders coding three different classes do not have to depend on each other.

Problem 3:

If we keep the mode as string/integer , then for every change in different fighting modes, we have to change in soldier class also which is not expected at all.

Solution:

If we keep modes as string/integer then such problems will arise often. So, it is better to use the modes as object. Only then, such problems will never happen in near future. Like, in the commander class we can use the modes as objects.



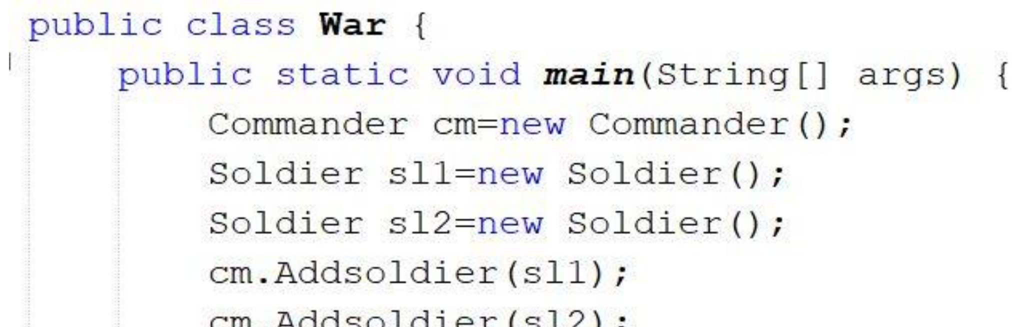
```
    public void Addsoldier(Soldier s)
    {
        soliderlist.add(s);
    }
    public void Startwar()
    {
        soliderlist.get(0).Changemood(new Aggressive());

        soliderlist.get(0).Changemood(new Defensive());

    }
```

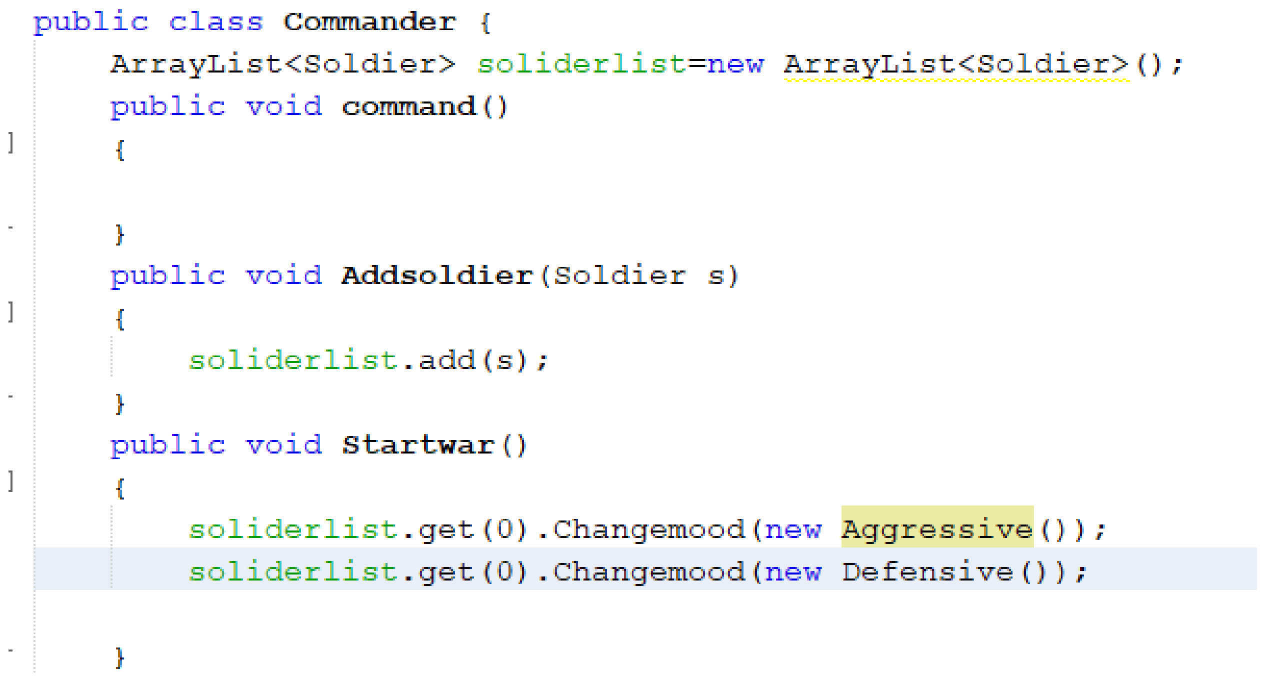
Main Code Sample Pic:

War Class:



```
public class War {
    public static void main(String[] args) {
        Commander cm=new Commander();
        Soldier sl1=new Soldier();
        Soldier sl2=new Soldier();
        cm.Addsoldier(sl1);
        cm.Addsoldier(sl2);
```

Commander Class

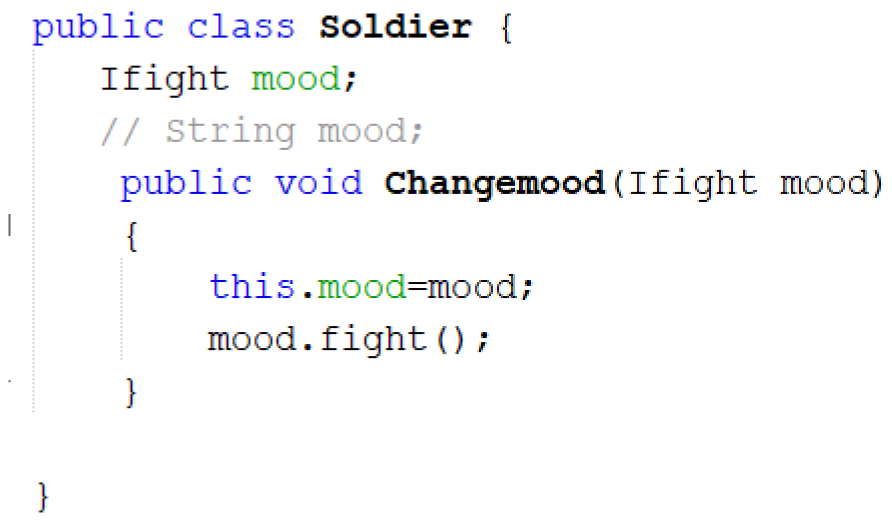


```
public class Commander {
    ArrayList<Soldier> soliderlist=new ArrayList<Soldier>();
    public void command()
    {

    }
    public void Addsoldier(Soldier s)
    {
        soliderlist.add(s);
    }
    public void Startwar()
    {
        soliderlist.get(0).Changemood(new Aggressive());
        soliderlist.get(0).Changemood(new Defensive());

    }
```

Soldier Class



```
public class Soldier {
    Ifight mood;
    // String mood;
     public void Changemood(Ifight mood)
     {
         this.mood=mood;
         mood.fight();
     }

}
```

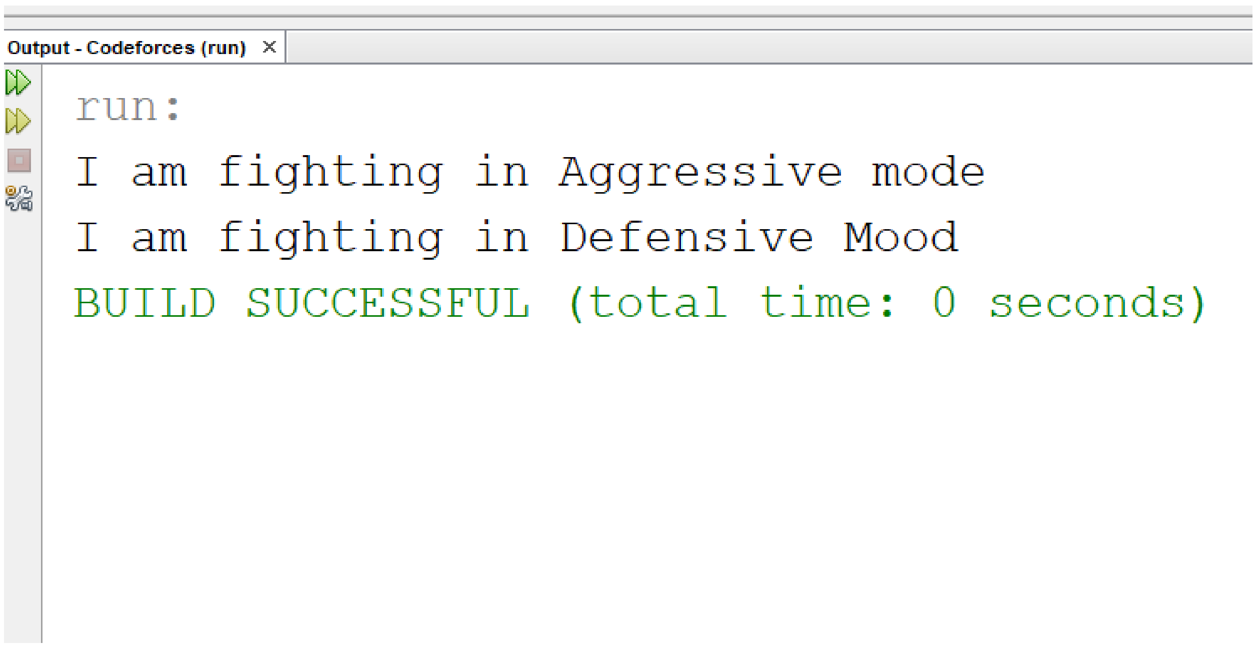
Ifight Interface:

```java
public interface Ifight {

    public void fight();

}
```

Output Sample:



```
run:
I am fighting in Aggressive mode
I am fighting in Defensive Mood
BUILD SUCCESSFUL (total time: 0 seconds)
```