Laborator MDS (design patterns)

Ciprian Paduraru

Design patterns:

- Ajuta la rezolvarea problemelor recurente in OOD
- Anticipeaza posibilile schimbari in diferite arii ale unui sistem.

Cateogorii discutate in acest laborator:

I. **Creational patterns** (Factory, Abstract Factory, Builder, Prototype, Singleton, Object pool...) Incearca simplificarea procesului de creare a obiectelor.

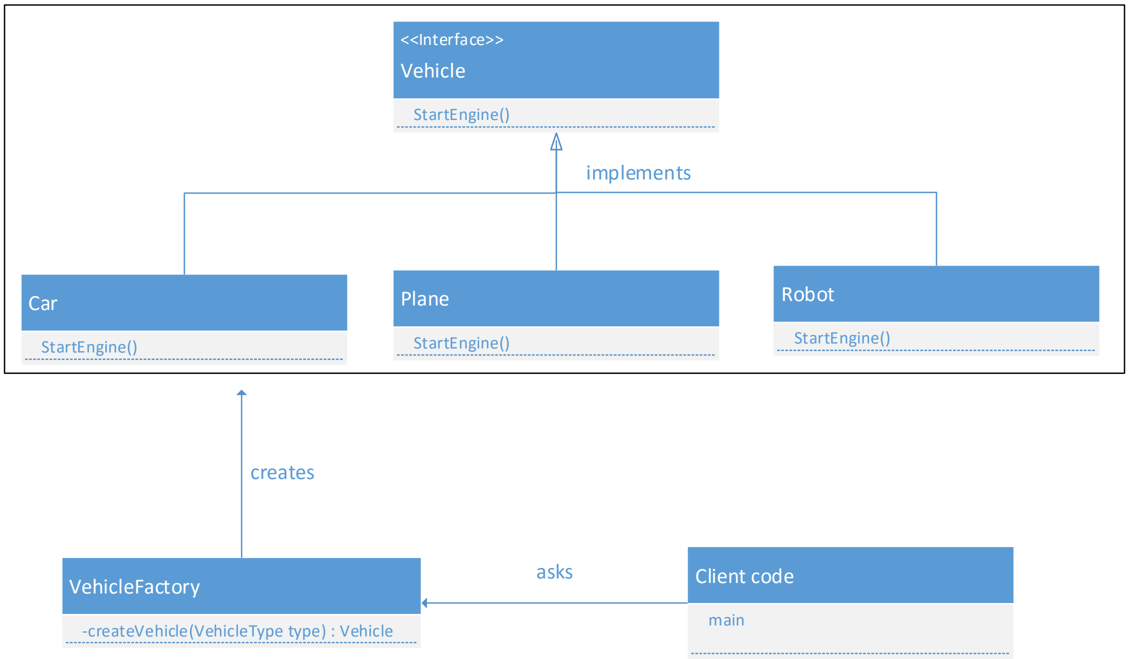
1. Singleton –asigura ca exista doar o singura instanta a unui obiect.

- Constructor/Copy-constructor privat
- Crearea/accesarea instantei se face folosind o functie statica.

```
class Logger
{
   private static Logger instance = new Logger();
   private Logger() { }
   private Logger(Logger other) {}

   public static Logger getInstance()
   {   // In C++ nu putem instatia ca mai sus Logger deci ar trebui sa mai adaugam:
      //if (instance == NULL ) {  instance = new Logger(); }

      return instance;
   }

   // Util functions:
   public void LogPlayer(String s)
   {
      System.out.println("Logging player " + s);
   }
}

public class JavaApplication3
{
   public static void main(String[] args)
   {
      Logger logger = Logger.getInstance();
      logger.LogPlayer("First");

      Logger.getInstance().LogPlayer("Second");
   }
 }
```

2. Factory – folosit pentru a returna/instantia un obiect folosind datele de intrare date factory-ului.

- Codul client doreste sa instantieze obiecte de anumite catogorii/parametri. Pentru a simplifica aceasta operatie implementer-ul obiectelor poate construi si o interfata de instantiere (factory).



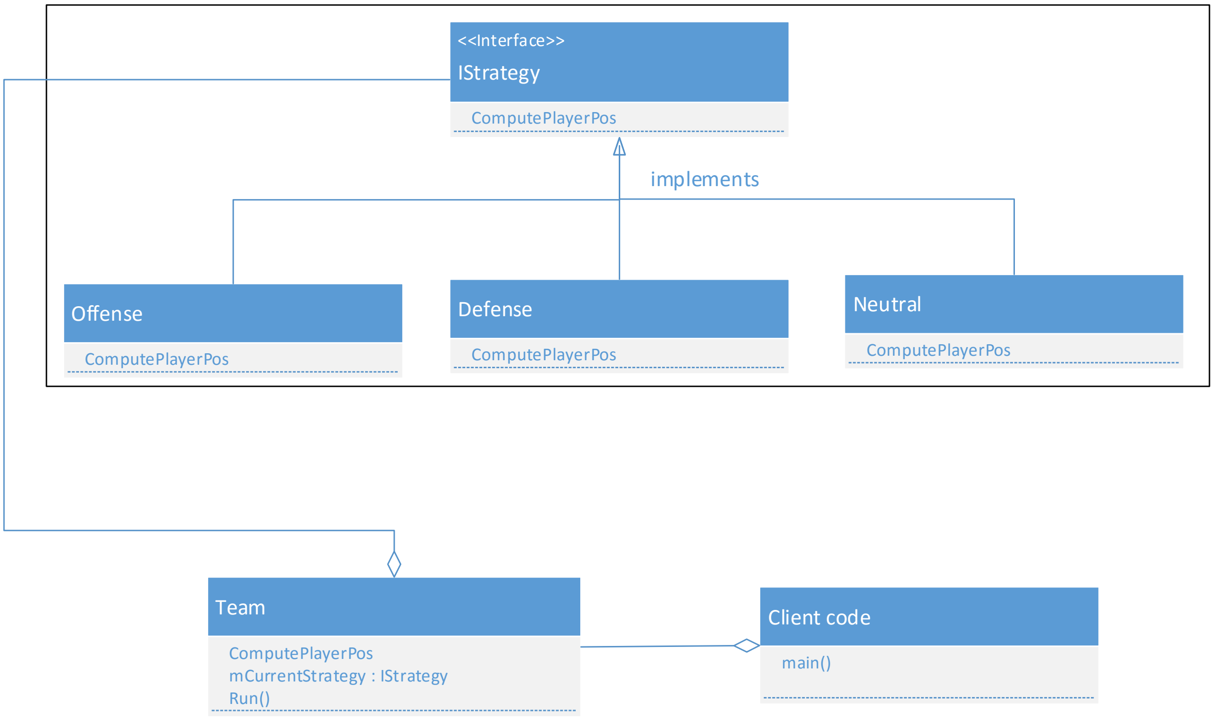
```
interface Vehicle
{
  void StartEngine();
}
class Car implements Vehicle
{
  public void StartEngine() { System.out.println("Starting car engine"); }
}
class Plane implements Vehicle
{
   public void StartEngine() { System.out.println("Starting plane engine"); }
}
class Robot implements Vehicle
{
   public void StartEngine() { System.out.println("Starting robot engine"); }
}
class VehicleFactory
{
   public enum VehicleType
   {
      E_VEHICLE_ROBOT,
      E_VEHICLE_CAR,
      E_VEHICLE_PLANE,
   }
   public Vehicle createVehicle(VehicleType type)
   {
      switch(type)
      {  // In practice we do some additional things not just instantiating
         // We could have different parameters as input, compute things then write to each instance
         case E_VEHICLE_ROBOT:
            return new Robot();
         case E_VEHICLE_CAR:
            return new Car();
         case E_VEHICLE_PLANE:
            return new Plane();
         default:
            assert false: "unknown type: " + type;
            return null;
      }
   }
}

public class JavaApplication3
{
   public static void main(String[] args)
   {
      VehicleFactory VF = new VehicleFactory();
      Vehicle ex = VF.createVehicle(VehicleFactory.VehicleType.E_VEHICLE_PLANE);
      ex.StartEngine();
   }
}
```

- Sunt multe variatii ale implemenatarii, unele folosind de exemplu functii statice pentru functia de “create” pentru a nu fi nevoie de instantierea unui factory.

3. **Abstract factory –** factory of factories (homework)

II. **Behavioral design patterns** (State, Strategy, Observer, Visitor, Iterator, Command, etc)

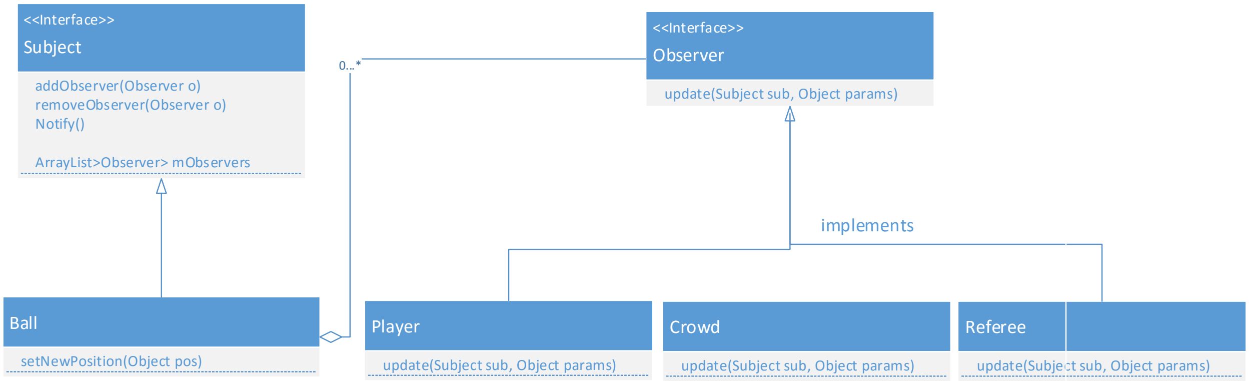
1. Strategy –decupleaza implementarea algoritmilor de codul client. Algoritmul / strategia devine abstracta pentru client (doar o interfata). De asemenea, strategia poate fi schimbata la runtime.



```
interface IStrategy
{
   void ComputePlayerPos();
}

class Offense implements IStrategy
{
   public void ComputePlayerPos() { System.out.println("Offense!"); }
}

class Defense implements IStrategy
{
   public void ComputePlayerPos() { System.out.println("Defense!"); }
}

class Neutral implements IStrategy
{
   public void ComputePlayerPos() { System.out.println("Neutral!"); }
}
```

```
class Team
{
   public void SetStrategy(IStrategy strategy) { mCurrentStrategy = strategy; }
   public void Run() { mCurrentStrategy.ComputePlayerPos(); }

   private IStrategy mCurrentStrategy = null;
}

public class JavaApplication3
{
   public static void main(String[] args)
   {
      Offense strategyOffense = new Offense();
      Defense strategyDefense = new Defense();

      Team team = new Team();
      team.SetStrategy(strategyDefense);
      team.Run();

      team.SetStrategy(strategyOffense);
      team.Run();
   }
}
```

2. Observer – decupleaza subiectul de observer si anunta observer-ii unui subiect ca starea acestuia s-a schimbat.


<<Interface>>
Subject
addObserver(Observer o)
removeObserver(Observer o)
Notify()
ArrayList>Observer> mObservers
0...*
<<Interface>>
Observer
update(Subject sub, Object params)
implements
Ball
setNewPosition(Object pos)
Player
update(Subject sub, Object params)
Crowd
update(Subject sub, Object params)
Referee
update(Subject sub, Object params)

```
interface Observer
{
   void Update(Subject o, Object argument);
}

abstract class Subject
```

```
{
   void AddObserver(Observer o)    { mObservers.add(o); }
   void RemoveObserver(Observer o) { mObservers.remove(o); }
   public void Notify()
   {
      for (Observer o : mObservers)
      {
         o.Update(this, null);
      }
   }

   ArrayList<Observer> mObservers = new ArrayList<Observer>();
}




class Ball extends Subject
{
   public void SetNewPosition(Object pos)
   {
      // Do some internal work...
      Notify();
   }
}

class Crowd implements Observer
{
  public void Update(Subject o, Object argument) {System.out.println("Crowd observed!");}
}

class Referee implements Observer
{
   public void Update(Subject o, Object argument) {System.out.println("Referee observed!");}
}

class Player implements Observer
{
   public void Update(Subject o, Object argument) {System.out.println("Player observed!");}
}

public class JavaApplication3
{
   public static void main(String[] args)
   {
      Crowd crowd = new Crowd();
      Player player1 = new Player();
```

```
        Player player2 = new Player();
        Referee ref = new Referee();
        Ball ball = new Ball();

        ball.AddObserver(crowd);
        ball.AddObserver(player1);
        ball.AddObserver(player2);
        ball.AddObserver(ref);

        ball.SetNewPosition(null);

        System.out.println("\nRemoving some observers....\n");
        ball.RemoveObserver(player1);
        ball.RemoveObserver(player2);
        ball.SetNewPosition(null);
    }
}
```