1. **(4 points) Write the code for the State Design Pattern.**

```
public interface IState
{
    void Handle();
}

public class OnState : IState
{
    public void Handle()
    {
        Console.WriteLine("Device is ON");
    }
}

public class OffState : IState
{
    public void Handle()
    {
        Console.WriteLine("Device is OFF");
    }
}

public class Device
{
    private IState state;

    public void SetState(IState newState)
    {
        state = newState;
    }

    public void PressButton()
    {
        state.Handle();
    }
}

Device device = new Device();

device.SetState(new OnState());
device.PressButton();

device.SetState(new OffState());
device.PressButton();
```

2. (4 points) Write the code for the Strategy Design Pattern.

```
public interface IStrategy
{
   void Execute();
}

public class AddStrategy : IStrategy
{
   public void Execute()
   {
      Console.WriteLine("Addition");
   }
}

public class MultiplyStrategy : IStrategy
{
   public void Execute()
   {
      Console.WriteLine("Multiplication");
   }
}

public class Context
{
   private IStrategy strategy;

   public void SetStrategy(IStrategy s)
   {
      strategy = s;
   }

   public void Run()
   {
      strategy.Execute();
   }
}

Context context = new Context();

context.SetStrategy(new AddStrategy());
context.Run();

context.SetStrategy(new MultiplyStrategy());
context.Run();
```

3. **(4 points) Write the code for the Chain of Responsibility Design Pattern.**

```
public abstract class Handler
{
   protected Handler next;

   public void SetNext(Handler handler)
   {
      next = handler;
   }

   public abstract void Handle(int value);
}

public class SmallNumberHandler : Handler
{
   public override void Handle(int value)
   {
      if (value < 10)
         Console.WriteLine("Handled by SmallNumberHandler");
      else if (next != null)
         next.Handle(value);
   }
}

public class LargeNumberHandler : Handler
{
   public override void Handle(int value)
   {
      if (value >= 10)
         Console.WriteLine("Handled by LargeNumberHandler");
   }
}

Handler h1 = new SmallNumberHandler();
Handler h2 = new LargeNumberHandler();

h1.SetNext(h2);

h1.Handle(5);
h1.Handle(20);
```

4. **(4 points) Write the code for the Builder Design Pattern.**

```
public class House
{
   public string Walls;
   public string Roof;
}

public interface IHouseBuilder
{
   void BuildWalls();
   void BuildRoof();
   House GetHouse();
}

public class HouseBuilder : IHouseBuilder
{
   private House house = new House();

   public void BuildWalls()
   {
      house.Walls = "Concrete walls";
   }

   public void BuildRoof()
   {
      house.Roof = "Metal roof";
   }

   public House GetHouse()
   {
      return house;
   }
}

IHouseBuilder builder = new HouseBuilder();

builder.BuildWalls();
builder.BuildRoof();

House house = builder.GetHouse();
```

5. (4 points) In SOLID principles, what is the Interface Segregation principle?

ინტერფეისის სეგრაგაციის პრინციპი ამბობს, რომ კლიენტები არ უნდა იყვნენ იძულებულნი, დამოკიდებული გახდნენ ინტერფეისებზე, რომლებასც ისინი არ იყენებენ. ეს ნიშნავს, რომ უმჯობესია გვქონდეს ბევრი პატარა და კონკრეტული ინტერფეისი, ვიდრე ერთი დიდი და ზოგადი. დიდი ინტერფეისი უნდა დაიშალოს პატარა ნაწილებად, რათა კლასებმა განახორციელონ მხოლოდ ის მეთოდები, რომლებიც მათ რეალურად სჭირდებათ