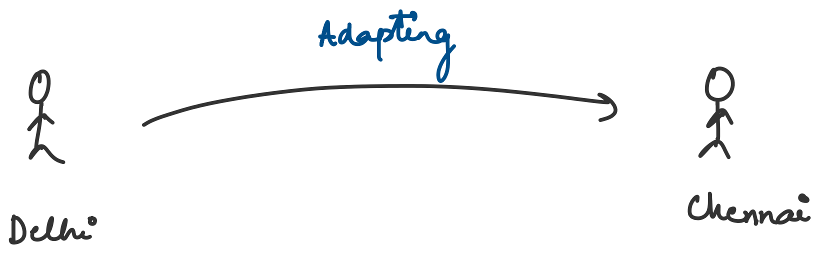
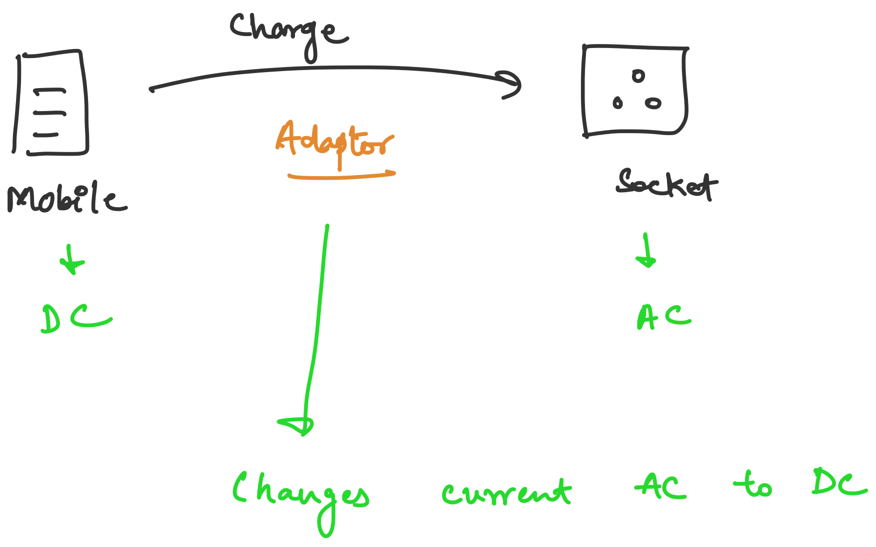
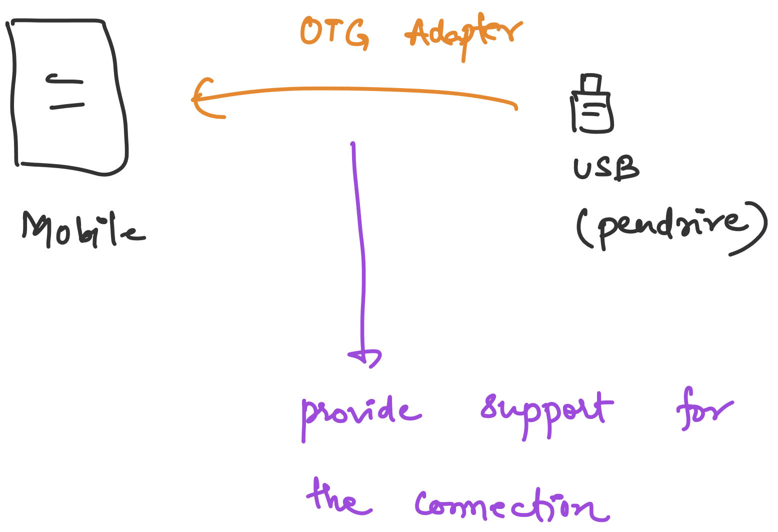
22__24-09-2022

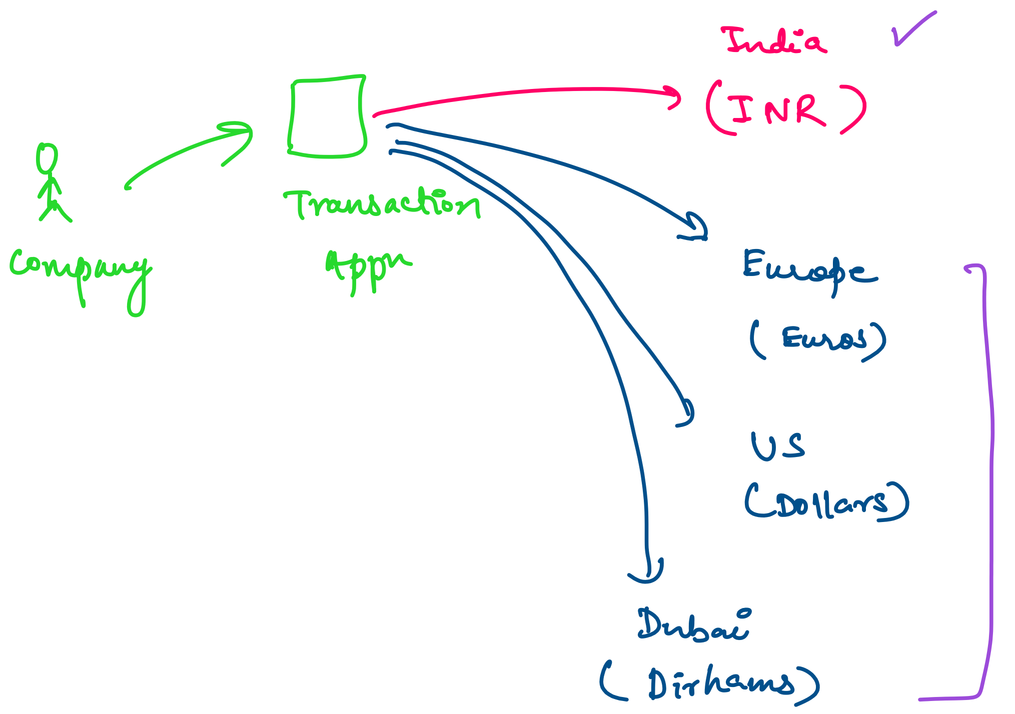
Saturday, 24 September 2022 8:05 PM

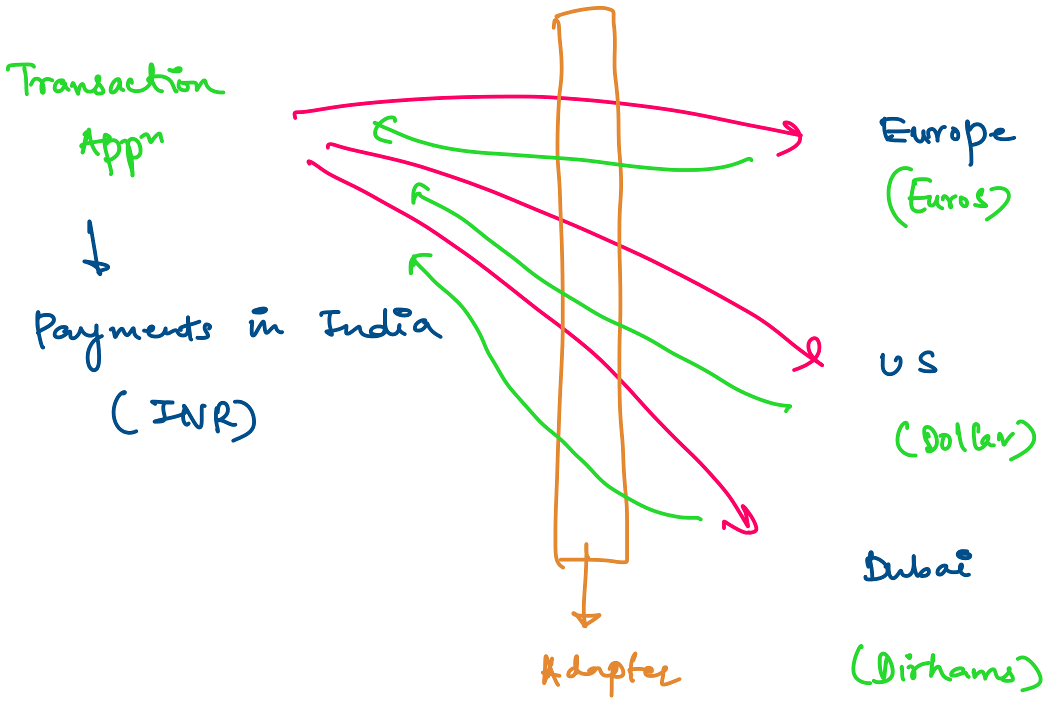
Adaptee design pattern





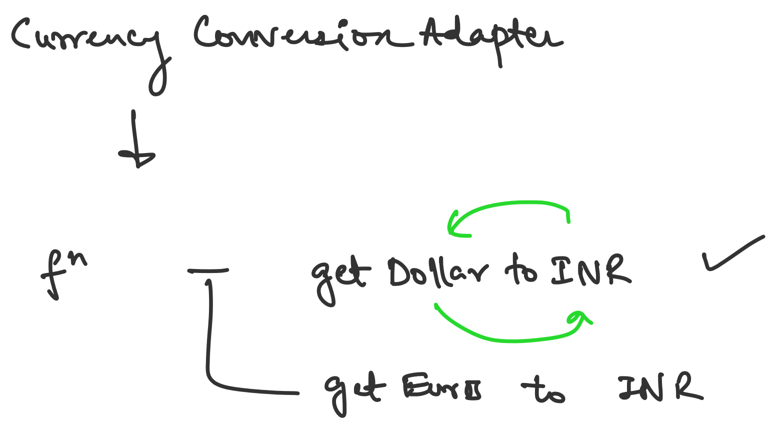






The idea here is to reuse the system as much as you can without changing functionality.

Even if some change has happened then the main dynamics of using this app^{n} should remain same. You should be able to use your app^{n} with the changing time and with the changing functionalities.



Adapter design pattern says – just convert the interface of a class into another interface that a client wants.

Adapter design pattern describes how to solve recurring design problems to design flexible and reusable object-oriented system, that means objects that are easier to implement, change, test and reuse.

As per gang of four — Adapter design pattern match interfaces of different classes.

This means an adapter allows two incompatible interfaces to work together.

Adaptor design pattern acts as a bridge between two incompatible interfaces.

In software engineering, the adapter design pattern is a software design pattern that allows the interfaces of an existing class to be used from another interface.

It is often used to make existing classes work with each other without modifying the source code.

Implementation guidelines

We use adapter design pattern when–

1. When an object needs to utilize an existing class with an incompatible interface.

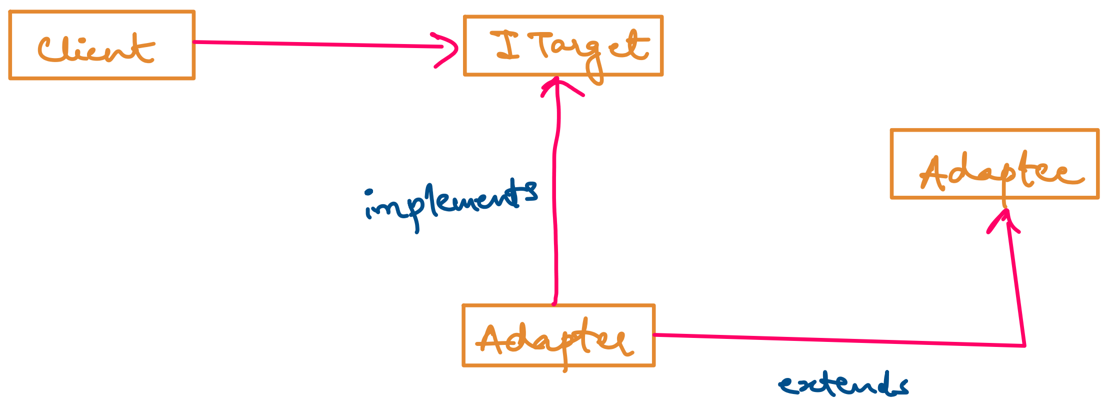
2. When you want to create a reusable class that cooperates with classes with incompatible types.

3. We need to work through a separate adapter that adapts the interface of an

existing class without changing it.

4. when clients do not know whether they work with a Target class directly or indirectly with the help of an adapter.

Adaptee design pattern representation



Client

↳ Client class is used to communicate with the adapter class that implements the ITarget interface.

This is the class that is used for creating the instance of adapter class.

ITarget

↳ Interface created to make the client achieve its purpose.

Adaptee class

↳ Adapter class implements the ITarget interface and extends the Adaptee class as well.

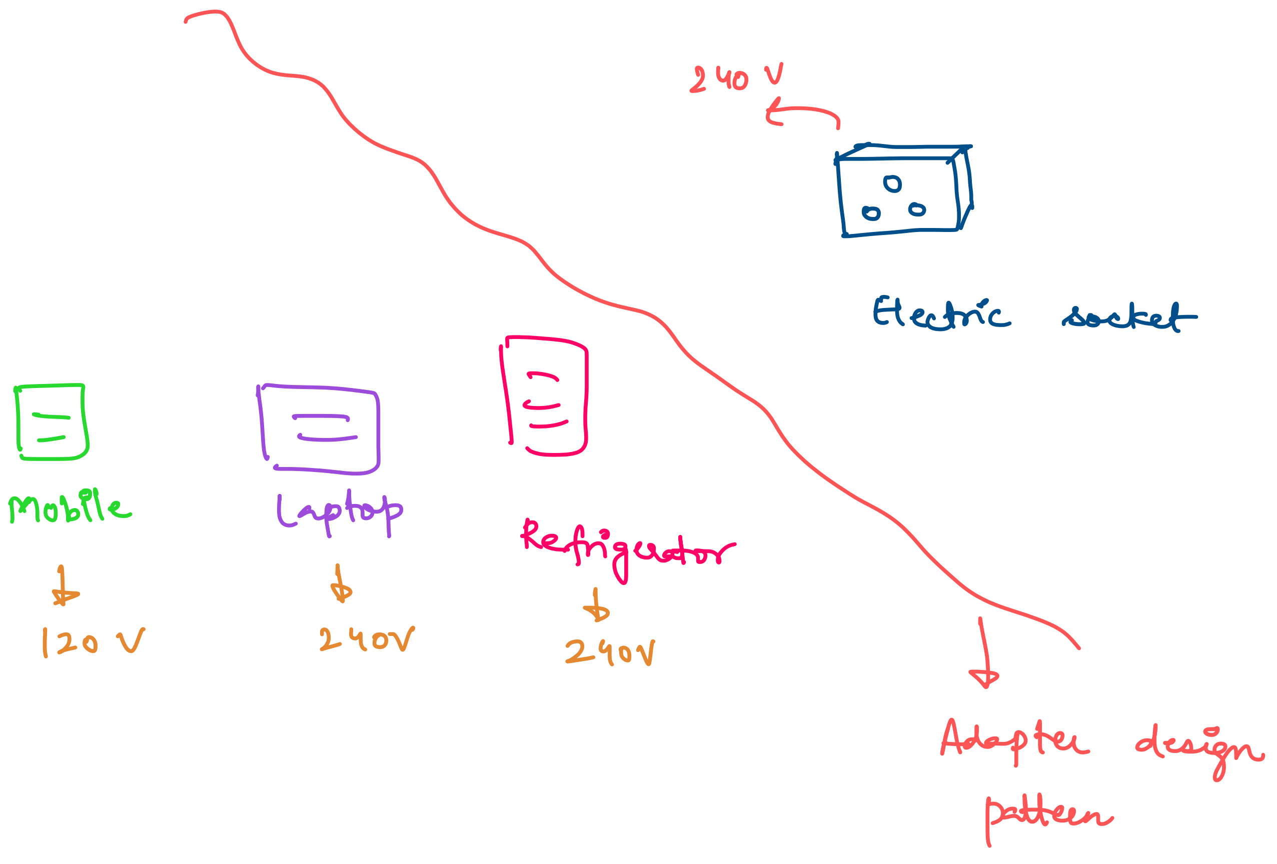
Adaptee class

↳ Adaptee class contains the main

functionality that client is looking for.

Advantages –

1. It allows reusability of existing functionality.
2. It allows two or more previously incompatible objects to interact.



1. Create Voltage class.

- constructor
- getter
- setter

2. Create Electric Socket class.
   - fetchElectricVoltage()

3. Create Socket Voltage Adapter Interface.
   - get120Voltage
   - get 60 Voltage
   - get 240 voltage

4. Create Socket voltage Adapter Impl class
   - implements SocketVoltageAdapter
   - Extends Electric Socket class

5. Create client class.

```java
package AdapterDesignPattern;

public class Voltage {
   private int voltageReading;

   public Voltage(int voltageReading) {
      this.voltageReading = voltageReading;
   }

   public int getVoltageReading() {
      return voltageReading;
   }

   public void setVoltageReading(int voltageReading) {
      this.voltageReading = voltageReading;
   }

   @Override
   public String toString() {
      return "Voltage{" +
            "voltageReading=" + voltageReading +
            '}';
   }
}
```

```java
package AdapterDesignPattern;

//Adaptee class
public class ElectricSocket {
   public Voltage fetchElectricVoltage()
   {
      return new Voltage(240);
   }
}
```

```
package AdapterDesignPattern;

public interface ISocketVoltageAdapter {
   /*
   This adapter is responsible for granting voltage to any device
   * */

   public Voltage get120VoltFromSocket();
   public Voltage get240VoltFromSocket();
   public Voltage get60VoltFromSocket();
   public Voltage get3VoltFromSocket();
}
```

```
package AdapterDesignPattern;

public class SocketVoltageAdapterImpl extends ElectricSocket implements
ISocketVoltageAdapter {
   @Override
   public Voltage get120VoltFromSocket() {
      Voltage voltage = fetchElectricVoltage();
      voltage.setVoltageReading(voltage.getVoltageReading()/2);
      return voltage;
   }

   @Override
   public Voltage get240VoltFromSocket() {
      return fetchElectricVoltage();
   }

   @Override
   public Voltage get60VoltFromSocket() {
      Voltage voltage = fetchElectricVoltage();
      voltage.setVoltageReading(voltage.getVoltageReading()/4);
      return voltage;
   }

   @Override
   public Voltage get3VoltFromSocket() {
      Voltage voltage = fetchElectricVoltage();
      voltage.setVoltageReading(voltage.getVoltageReading()/80);
      return voltage;
   }
}
```

```
package AdapterDesignPattern;

public class Client {
   public static void main(String[] args) {
      ISocketVoltageAdapter adapter = new SocketVoltageAdapterImpl();

      Voltage v3 = adapter.get3VoltFromSocket();
      System.out.println(v3);

      Voltage v120 = adapter.get120VoltFromSocket();
      System.out.println(v120);
   }
}
```

Output:

```
Voltage{voltageReading=3}
Voltage{voltageReading=120}
```