Nesneye Yönelik Yazılım Mühendisliği (376)

Dr. Öğr. Üyesi Ahmet Arif AYDIN

Abstract Factory (Creational) Design Pattern

- Aynı arayüzü (interface) kullanan nesnelerin oluşturulması ve yönetimini sağlar. Alt sınıflara göre class nesnesi oluşturulur. *(Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses*
- Factory metodu bir method içerisinde ihtiyaç duyulan nesnenin oluşturulup kullanılması için uygun olan sınıfı seçer ve örneğini oluştur.

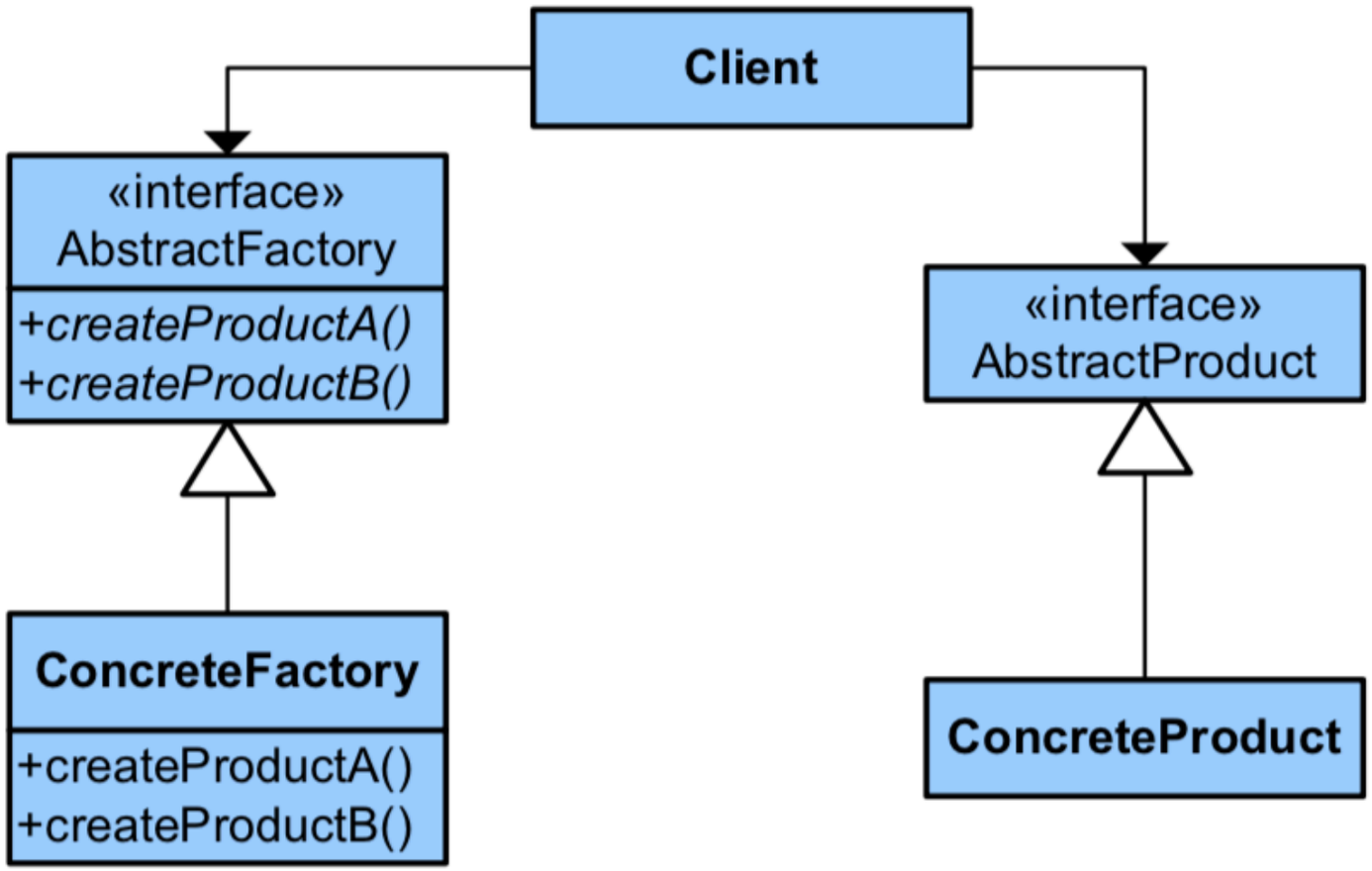
- **Abstract Factory birden fazla Factory tasarım kalıbının oluşturulup yönetilmesini saglar.** (The purpose of the Abstract Factory is to provide an interface for creating families of related objects, without specifying concrete classes)

Abstract Factory (Creational) Design Pattern

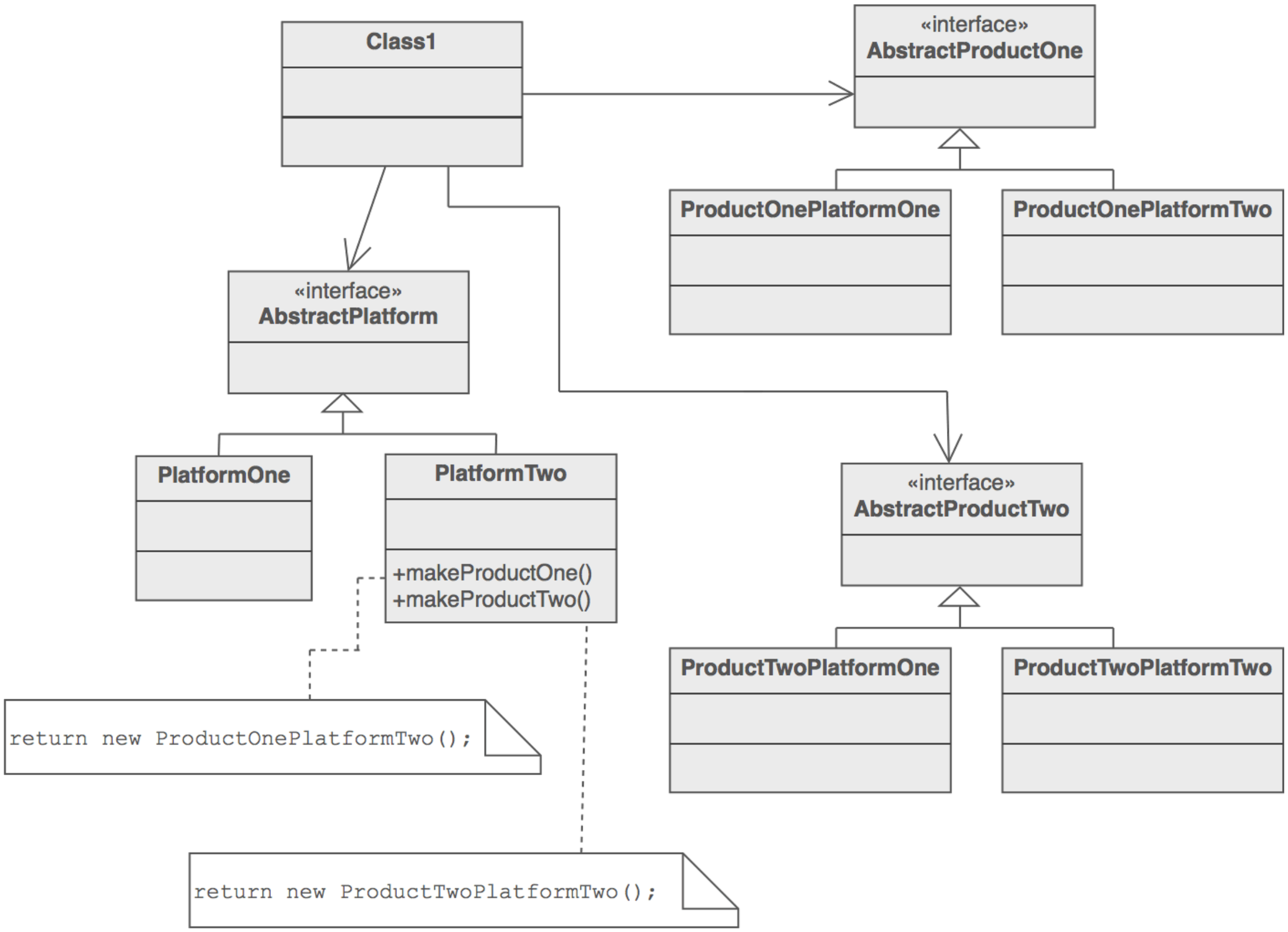
Abstract Factory

Type: Creational

What it is:
Provides an interface for creating families of related or dependent objects without specifying their concrete class.



Abstract Factory (Creational) Design Pattern



Abstract Factory (Creational) Design Pattern

```
public abstract class AbstractFactory {
   abstract Shape getShape(String shapeType) ;
}
```

```
public class ShapeFactory extends AbstractFactory {
   @Override
   public Shape getShape(String shapeType){
      if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();
      }else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }
      return null;
   }
}
```

```
public class FactoryProducer {
   public static AbstractFactory getFactory(boolean rounded){
      if(rounded){
         return new RoundedShapeFactory();
      }else{
         return new ShapeFactory();
      }
   }
}
```

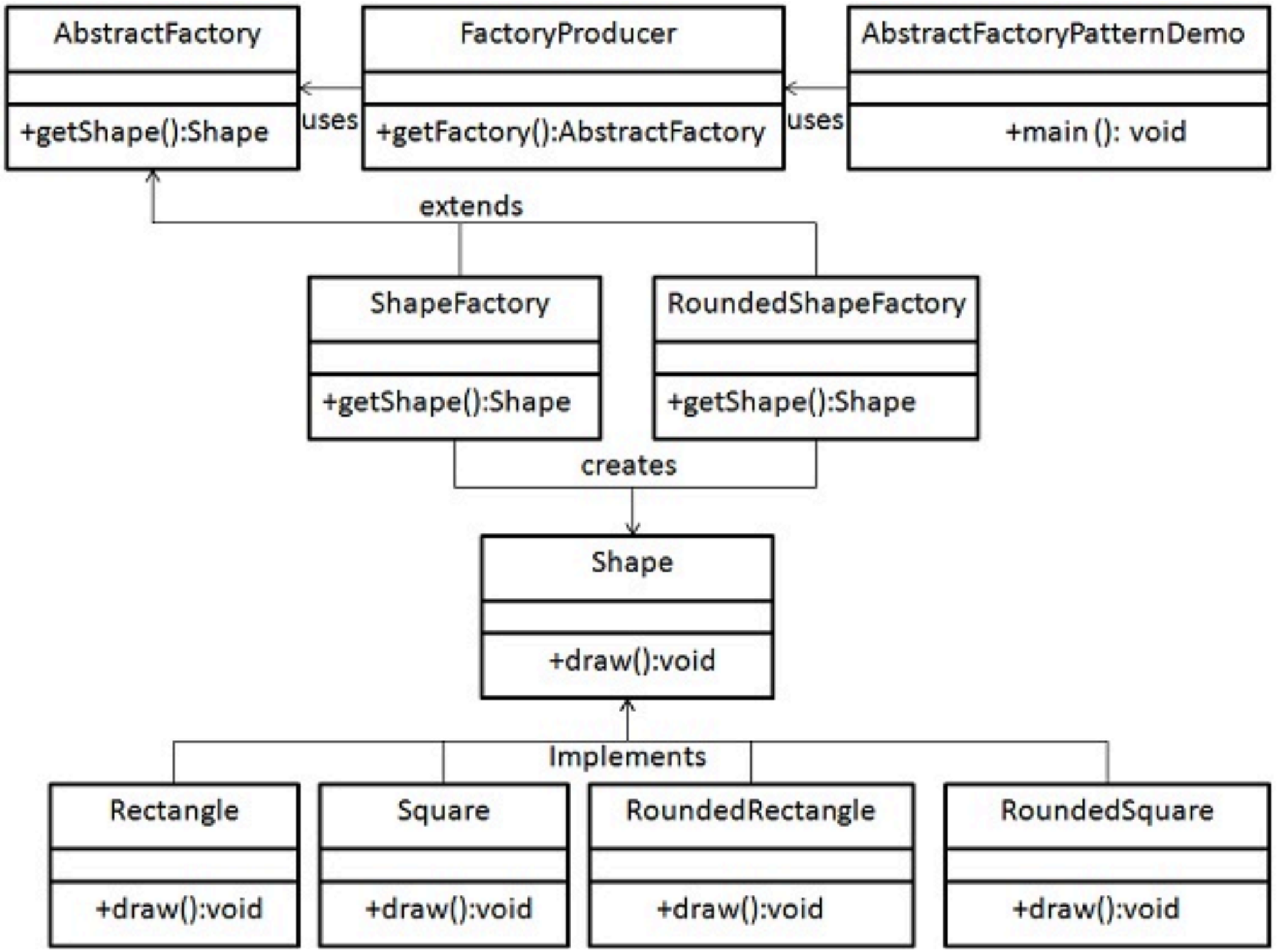
```
public class RoundedShapeFactory extends AbstractFactory {
   @Override
   public Shape getShape(String shapeType){
      if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new RoundedRectangle();
      }else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new RoundedSquare();
      }
      return null;
   }
}
```

Abstract Factory (Creational) Design Pattern

```
public class FactoryProducer {
   public static AbstractFactory getFactory(boolean rounded){
      if(rounded){
         return new RoundedShapeFactory();
      }else{
         return new ShapeFactory();
      }
   }
}
```

```
public class AbstractFactoryPatternDemo {
   public static void main(String[] args) {
      //get rounded shape factory
      AbstractFactory shapeFactory = FactoryProducer.getFactory(false);
      //get an object of Shape Rounded Rectangle
      Shape shape1 = shapeFactory.getShape("RECTANGLE");
      //call draw method of Shape Rectangle
      shape1.draw();
      //get an object of Shape Rounded Square
      Shape shape2 = shapeFactory.getShape("SQUARE");
      //call draw method of Shape Square
      shape2.draw();
      //get rounded shape factory
      AbstractFactory shapeFactory1 = FactoryProducer.getFactory(true);
      //get an object of Shape Rectangle
      Shape shape3 = shapeFactory1.getShape("RECTANGLE");
      //call draw method of Shape Rectangle
      shape3.draw();
      //get an object of Shape Square
      Shape shape4 = shapeFactory1.getShape("SQUARE");
      //call draw method of Shape Square
      shape4.draw();

   }
}
```

Abstract Factory (Creational) Design Pattern



https://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm

Proxy (Structural) Design Patterns

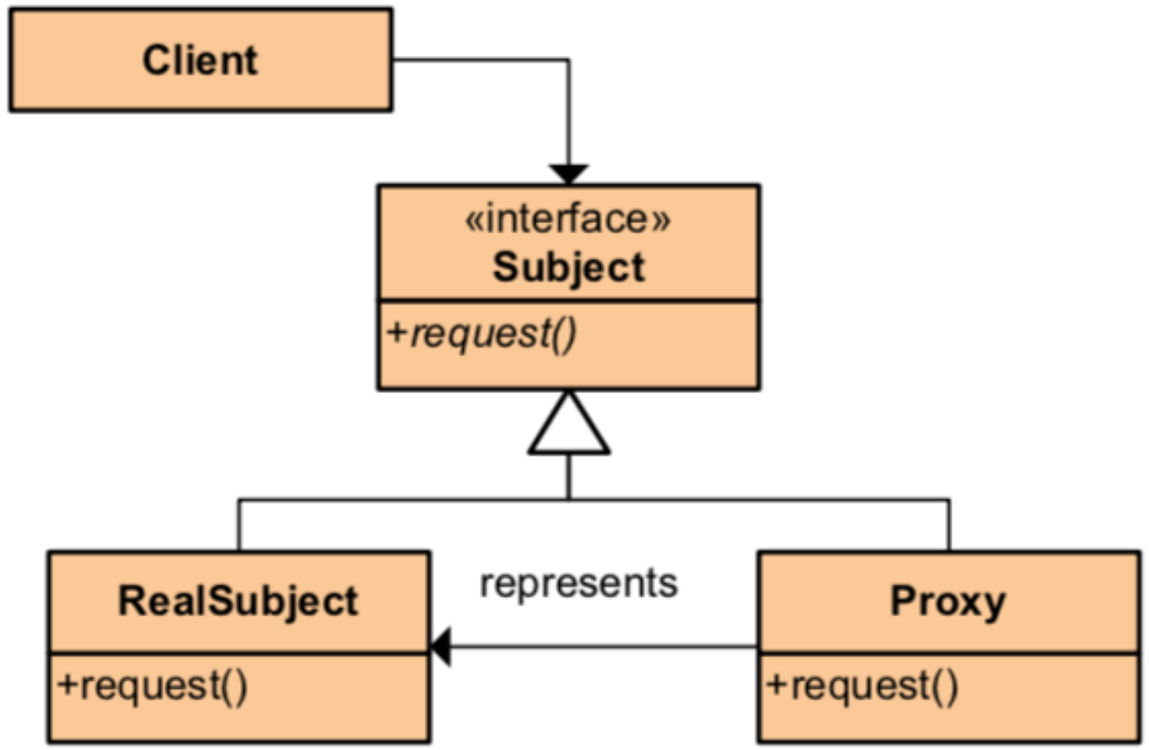
- **Proxy tasarım kalıbı bir nesneyi kullanmak için oluşturulan başka bir vekil nesne olarak tanımlanır.** (The Proxy Pattern is used to create a representative object that controls access to another object, which may be **remote**, **expensive to create** or in need of being **secured)**
-
- Kullanıcının original nesneye direkt erişimi yerine proxy yardımıyla erişim saglanır. Kullanıcı arada bir proxy'nin oldugunu bilmez.

Proxy (Structural) Design Patterns

Proxy

Type: Structural

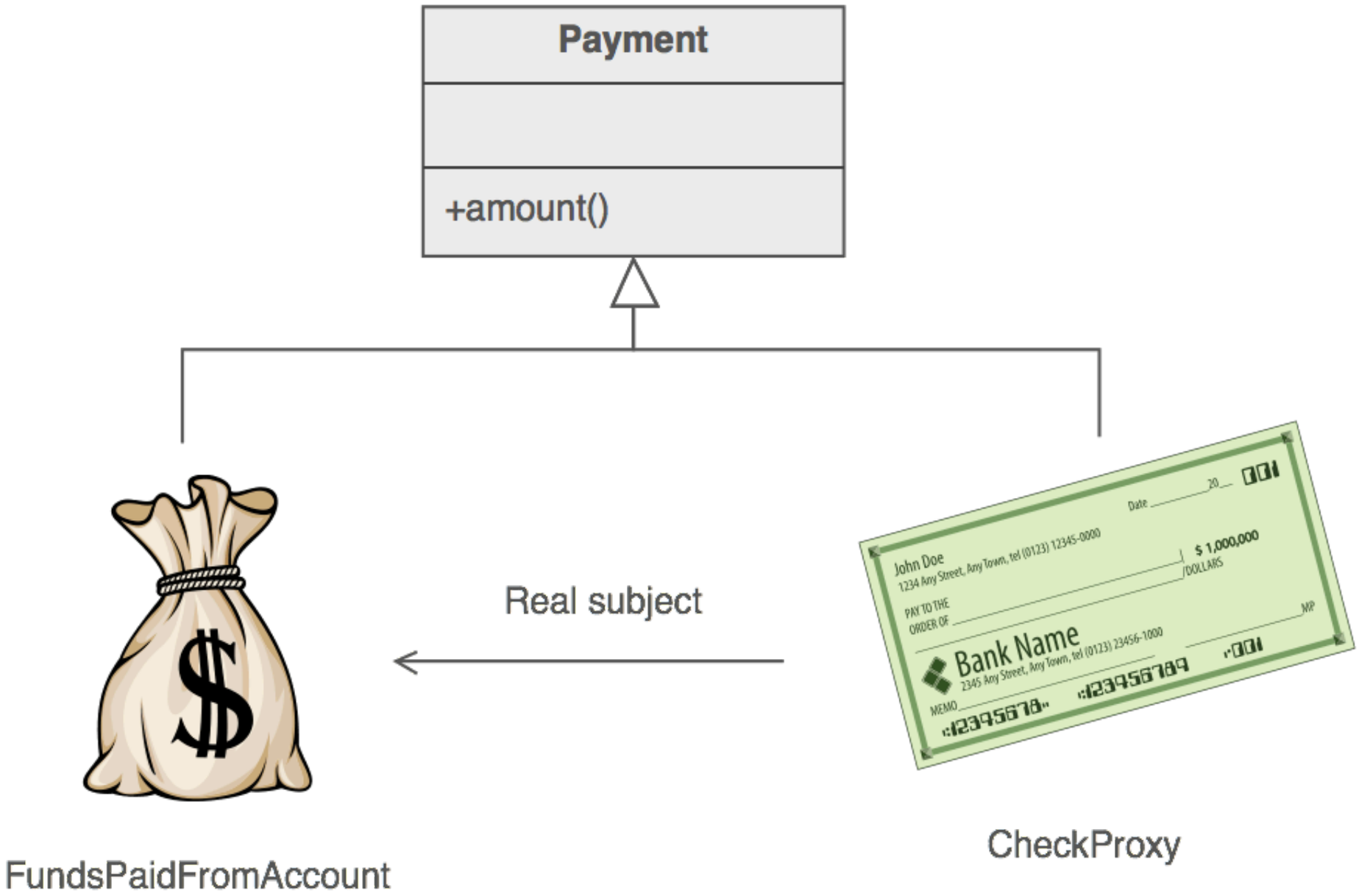
What it is:
Provide a surrogate or placeholder for another object to control access to it.



Proxy (Structural) Design Patterns

```
interface SocketInterface {
    String readLine();
    void  writeLine(String str);
    void  dispose();
}

class SocketProxy implements SocketInterface {
    // 1. Create a "wrapper" for a remote,
    // or expensive, or sensitive target
    private Socket socket;
    private BufferedReader in;
    private PrintWriter out;

    public SocketProxy(String host, int port, boolean wait) {
        try {
            if (wait) {
                // 2. Encapsulate the complexity/overhead of the
target in the wrapper
                ServerSocket server = new ServerSocket(port);
                socket = server.accept();
            } else {
                socket = new Socket(host, port);
            }
            in  = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            out = new PrintWriter(socket.getOutputStream(), true);
        } catch(IOException e) {
            e.printStackTrace();
        }
    }

    public String readLine() {
        String str = null;
        try {
            str = in.readLine();
        } catch( IOException e ) {
            e.printStackTrace();
        }
        return str;
    }

    public void writeLine(String str) {
        // 4. The wrapper delegates to the target
        out.println(str);
    }

    public void dispose() {
        try {
            socket.close();
        } catch(IOException e) {
            e.printStackTrace();
        }
    }
}

public class ProxyDemo {
    public static void main( String[] args ) {
        // 3. The client deals with the wrapper
        SocketInterface socket = new SocketProxy( "127.0.0.1", 8080,
args[0].equals("first") ? true : false );
        String  str;
        boolean skip = true;
        while (true) {
            if (args[0].equals("second") && skip) {
                skip = !skip;
            } else {
                str = socket.readLine();
                System.out.println("Receive - " + str);
                if (str.equals(null)) {
                    break;
                }
            }
            System.out.print( "Send ---- " );
            str = new Scanner(System.in).nextLine();
            socket.writeLine( str );
            if (str.equals("quit")) {
                break;
            }
        }
        socket.dispose();
    }
}
```

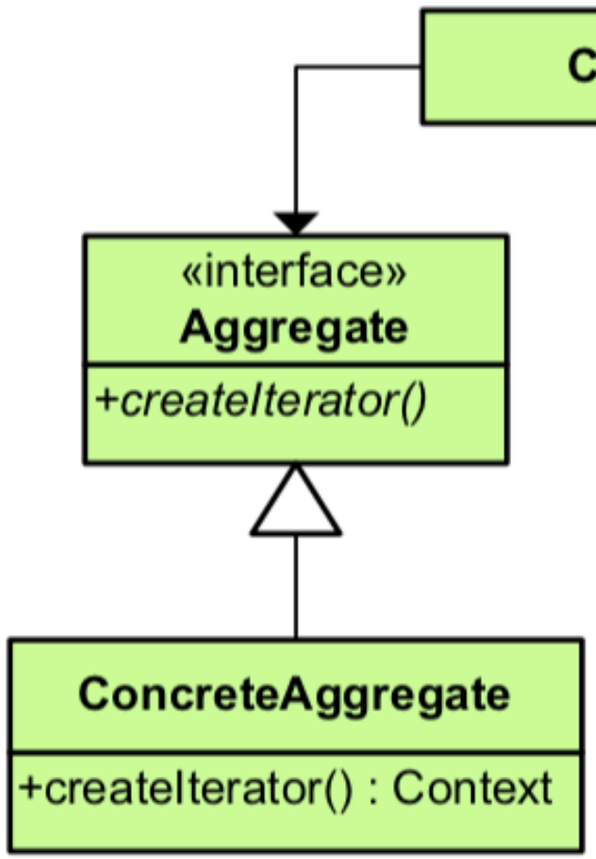
Proxy (Structural) Design Patterns

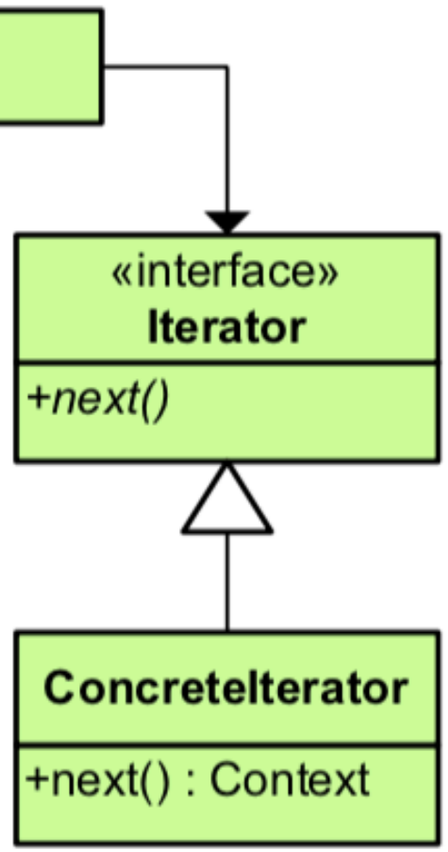


İterator (Behavioral) Design Patterns

- Veri saklayan bir nesnesin içerisinde bulunan elemanlara yapıyı dışarıya göstermeden sırasıyla erişim imkanı saglar
- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation

İterator (Behavioral) Design Patterns





Iterator

Type: Behavioral

What it is:
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

İterator (Behavioral) Design Patterns

```java
public interface Iterator {
   public boolean hasNext();
   public Object next();
}
```

```java
public interface Container {
   public Iterator getIterator();
}
```

```java
public class NameRepository implements Container {
   public String names[] = {"Robert" , "John" ,"Julie" , "Lora"};

   @Override
   public Iterator getIterator() {
      return new NameIterator();
   }

   private class NameIterator implements Iterator {

      int index;

      @Override
      public boolean hasNext() {

         if(index < names.length){
            return true;
         }
         return false;
      }

      @Override
      public Object next() {

         if(this.hasNext()){
            return names[index++];
         }
         return null;
      }
   }
}
```

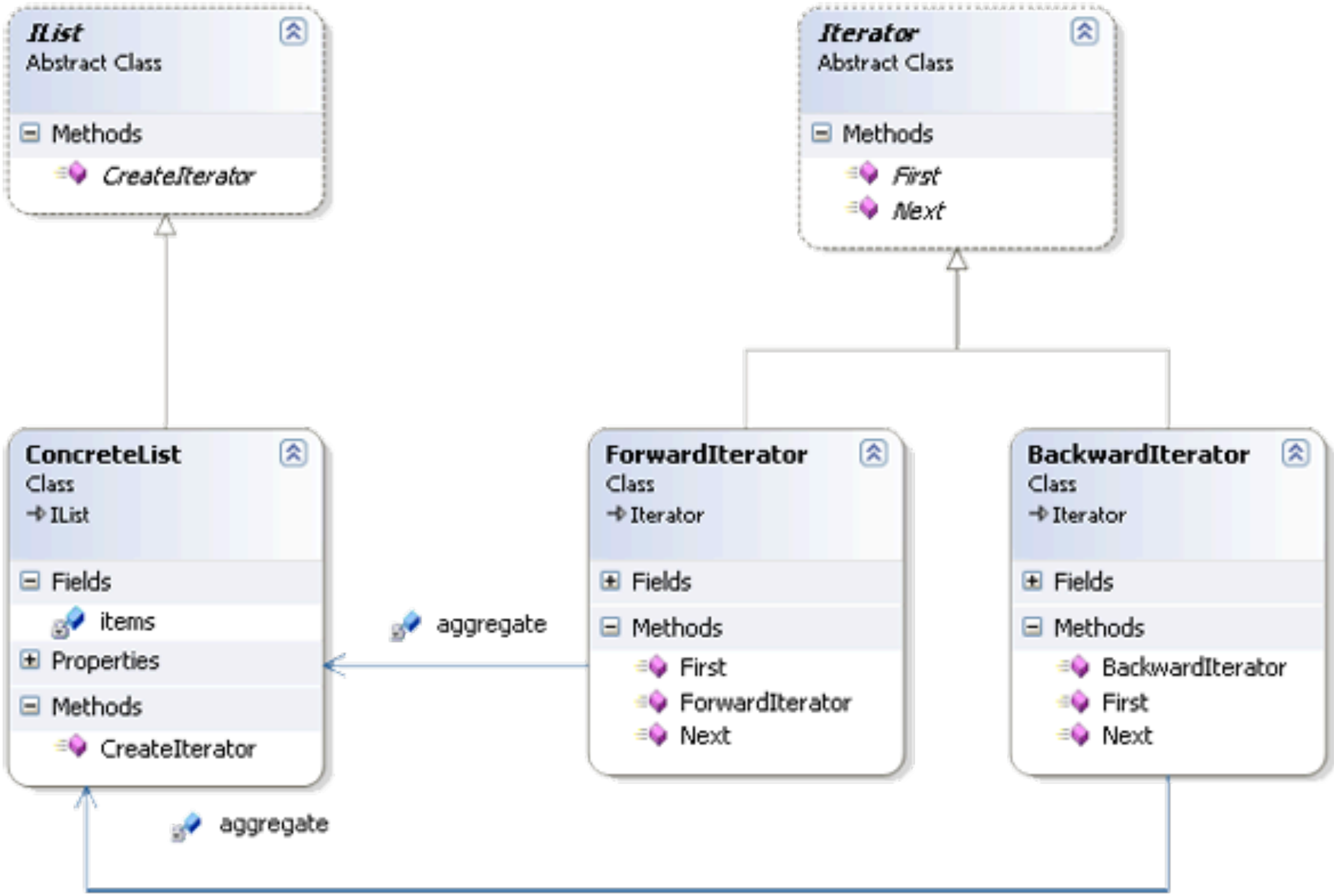
https://www.tutorialspoint.com/design_pattern/iterator_pattern.htm

İterator (Behavioral) Design Patterns

```
public class IteratorPatternDemo {

   public static void main(String[] args) {
      NameRepository namesRepository = new NameRepository();

      for(Iterator iter = namesRepository.getIterator(); iter.hasNext();){
         String name = (String)iter.next();
         System.out.println("Name : " + name);
      }
   }
}
```

https://www.tutorialspoint.com/design_pattern/iterator_pattern.htm

İterator (Behavioral) Design Patterns



İterator (Behavioral) Design Patterns