Advanced Programming

→ Structural Design Patterns

Hello!

I am Hamzeh Asefan

Master Degree from King Abdullah II School of Information Technology in Information Systems (2020/2021) / The University of Jordan.

Structural Design Patterns

Adapter

Converts the interface of a class into another interface client expect

Bridge

Decouple an abstraction from its implementation so that the two can vary independently

Provides a simplified interface to complex set of classes

Facade

Hides the original object and control access to it

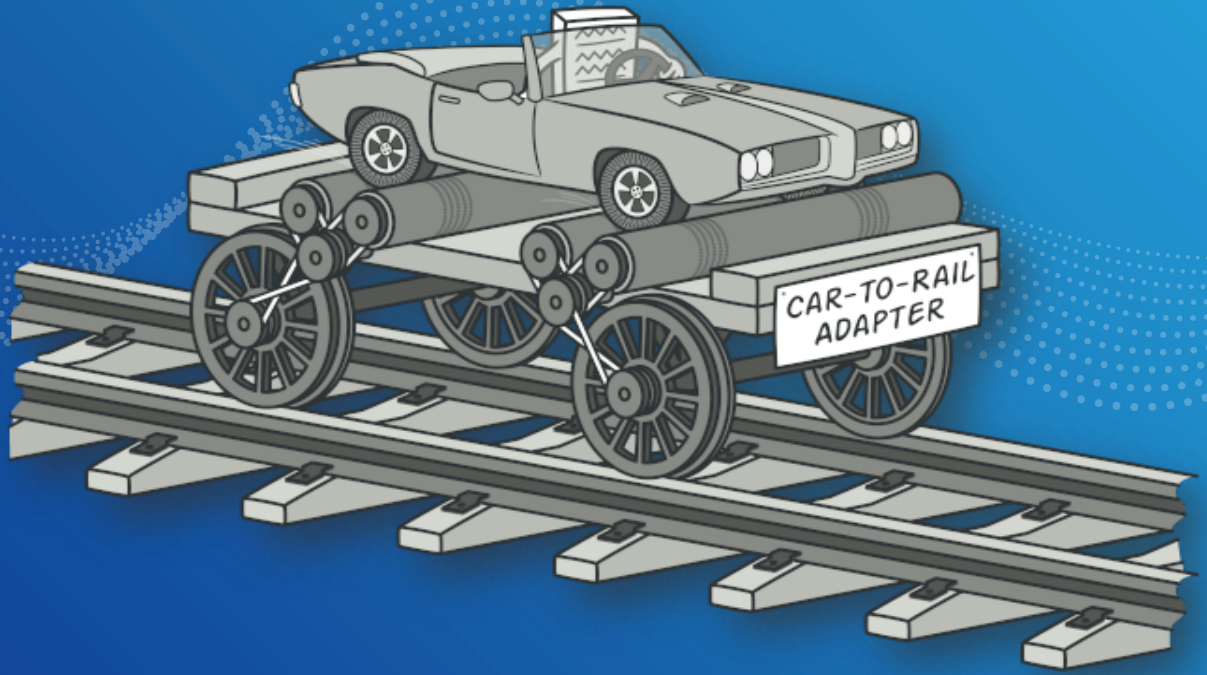
Proxy

1. Adapter

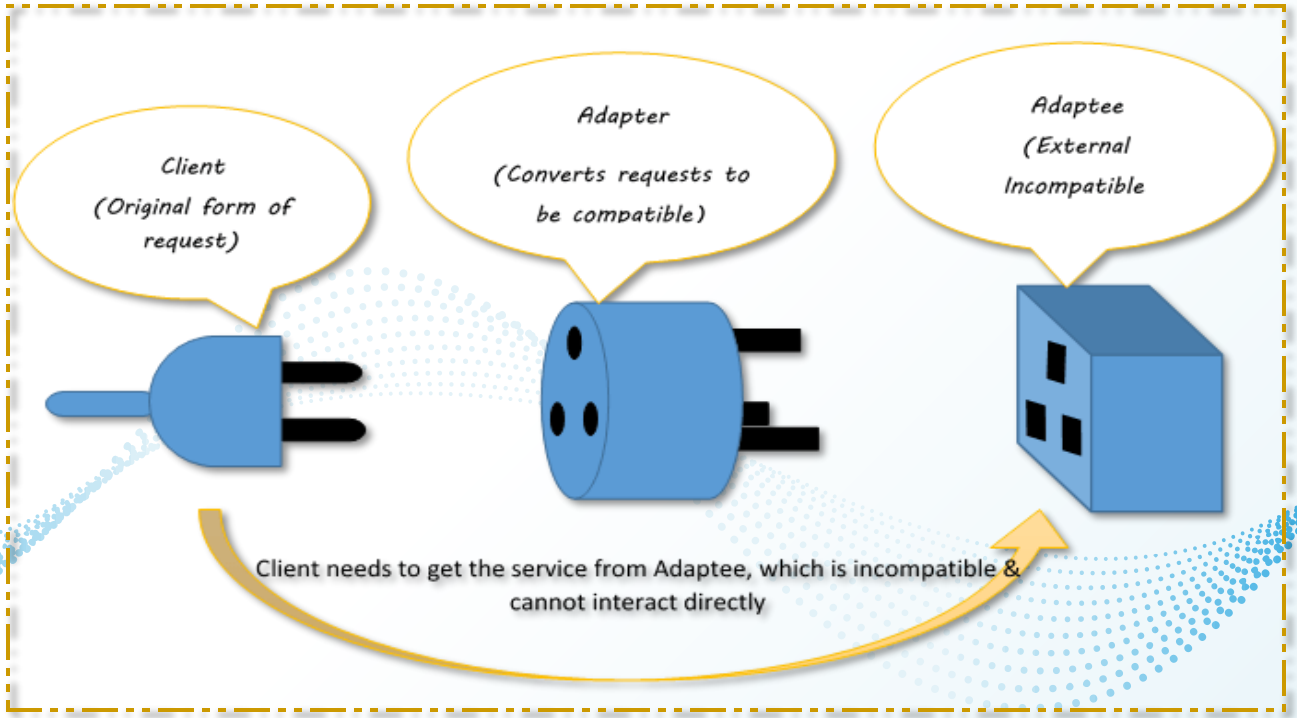
Adapter

- The adapter pattern converts the interface of a class into another interface client expect.
- Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.
- Also Known as Wrapper.

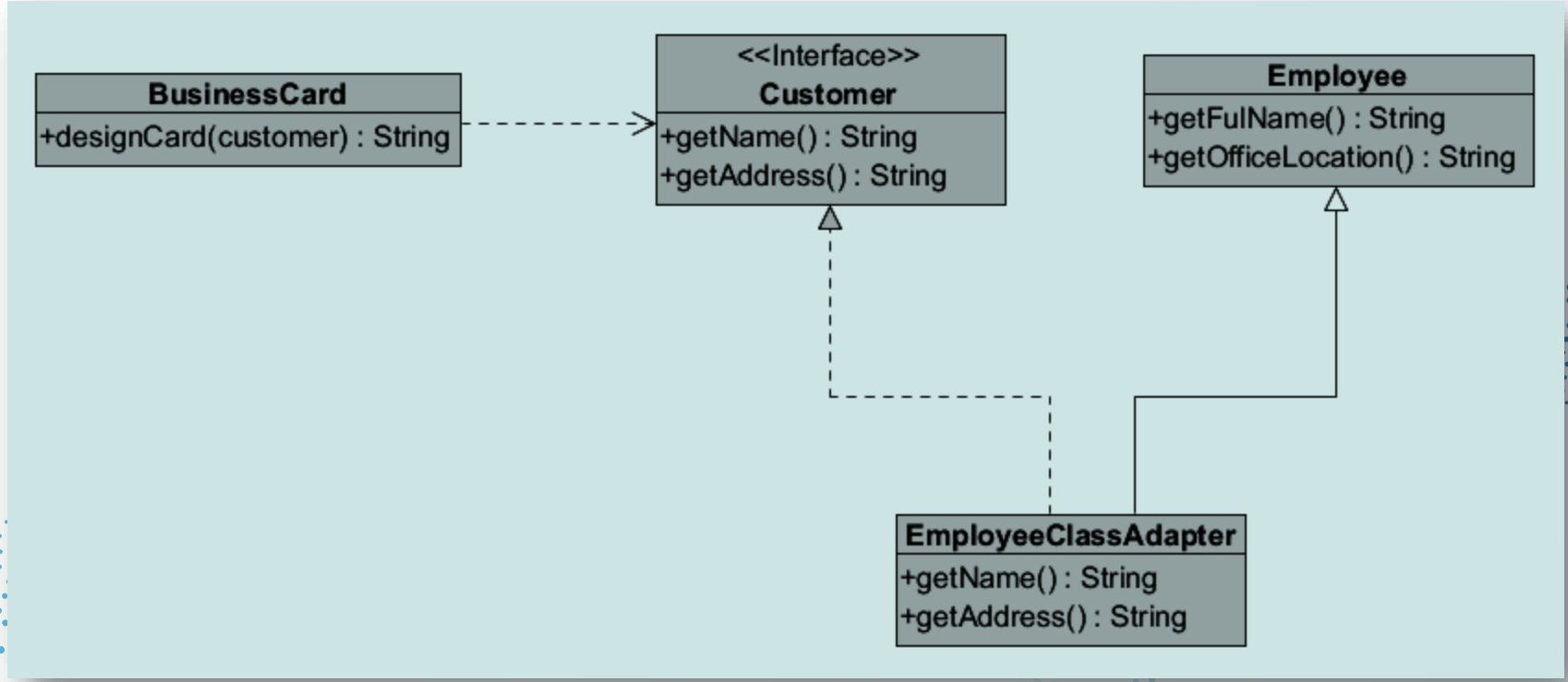
Adapter



Adapter


Client
(Original form of request)
Adapter
(Converts requests to be compatible)
Adaptee
(External Incompatible
Client needs to get the service from Adaptee, which is incompatible & cannot interact directly

Adapter Example



Adapter Example

```java
public interface Customer {

    String getName();

    String getAddress();
}
```

```java
public class Employee {

    private String fullName;
    private String officeLocation;

    public String getFullName() {
        return fullName;
    }

    public void setFullName(String fullName) {
        this.fullName = fullName;
    }

    public String getOfficeLocation() {
        return officeLocation;
    }

    public void setOfficeLocation(String officeLocation) {
        this.officeLocation = officeLocation;
    }

}
```

Adapter Example

```
public class BusinessCard {

    public String designCard(Customer customer) {

        return customer.getName() + "\t" + customer.getAddress();

    }
}
```

```
public class EmployeeClassAdapter extends Employee implements Customer {

    @Override
    public String getName() {
        return this.getFullName();
    }

    @Override
    public String getAddress() {
        return this.getOfficeLocation();
    }

}
```

Adapter Example

```
public class Main {

    public static void main(String[] args) {

        EmployeeClassAdapter adapter = new EmployeeClassAdapter();
        populateEmployeeData(adapter);

        BusinessCard designer = new BusinessCard();
        String card = designer.designCard(adapter);
        System.out.println(card);
    }

    private static void populateEmployeeData(Employee e) {
        e.setFullName("Hamzeh Asefan");
        e.setOfficeLocation("Amman, Jordan");
    }

}
```

2. Facade

Facade

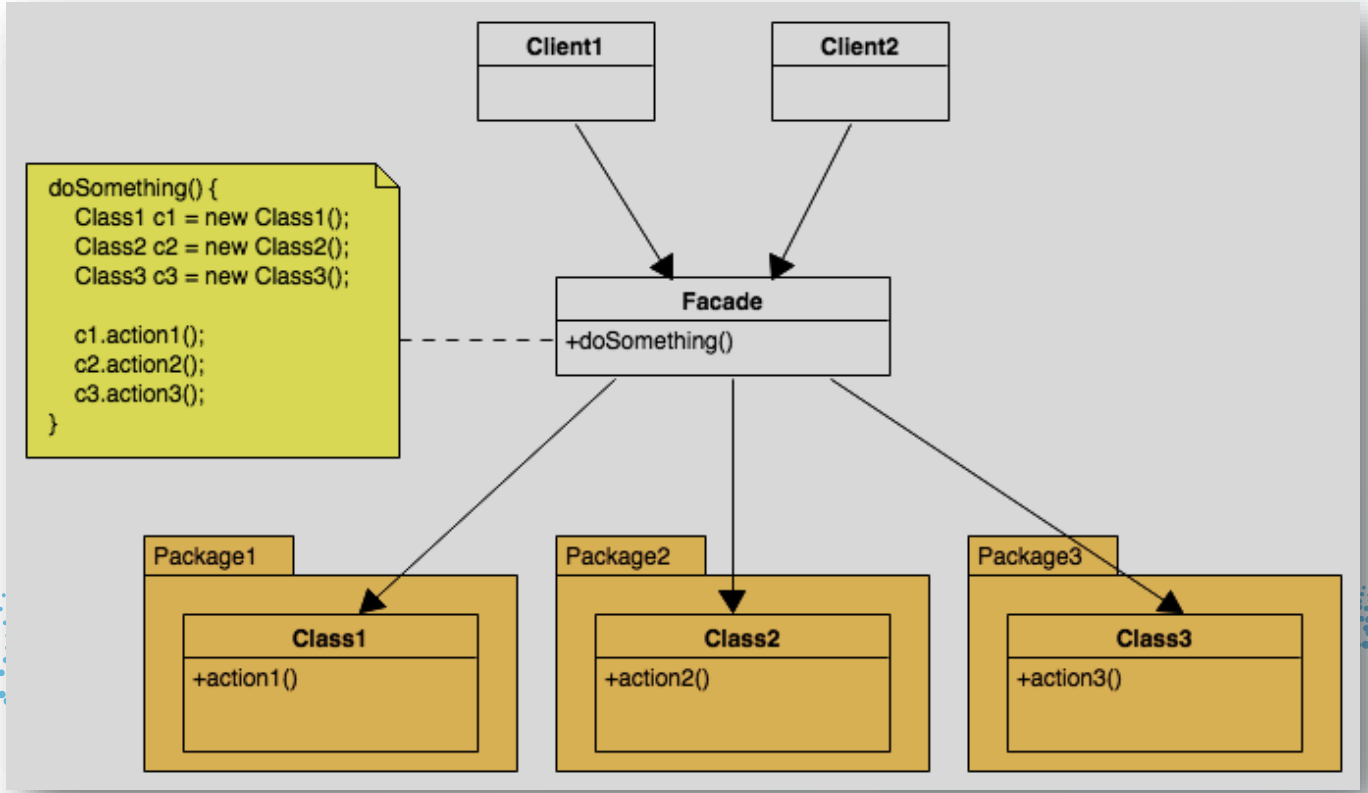
- Provides a simplified interface to a library, a framework, or any other complex set of classes
- It hides the complexities of the subsystem from the client.
- This structure helps you to minimize the effort of upgrading to future versions of the framework or replacing it with another one. The only thing you would need to change in your app would be the implementation of the facade's methods.

Facade

When you call a shop to place a phone order →

Operator is your facade to all services and departments of the shop.

Facade Class Diagram



Facade Example

```java
public class Product {

    public void getProductDetails() {
        System.out.println("Show the Product Details");
    }
}
```

```java
public class Payment {

    public void makePayment() {
        System.out.println("Payment done successfully");
    }
}
```

```java
public class Invoice {

    public void sendInvoice() {
        System.out.println("Invoice send successfully");
    }
}
```

Facade Example

```
public class OrderFacade {

    public void placeOrder() {

        Product product = new Product();
        product.getProductDetails();

        Payment payment = new Payment();
        payment.makePayment();

        Invoice invoice = new Invoice();
        invoice.sendInvoice();
    }

}
```

```
public class TestFacade {

    public static void main(String[] args) {

        OrderFacade order = new OrderFacade();
        order.placeOrder();

    }
}
```

```
Show the Product Details
Payment done successfully
Invoice send successfully
```

Facade

We can make changes to the existing subsystem and don't affect a client.

Facade

We can make changes to the existing subsystem and don't affect a client.

Facade

Improve the readability and usability.

Facade Example

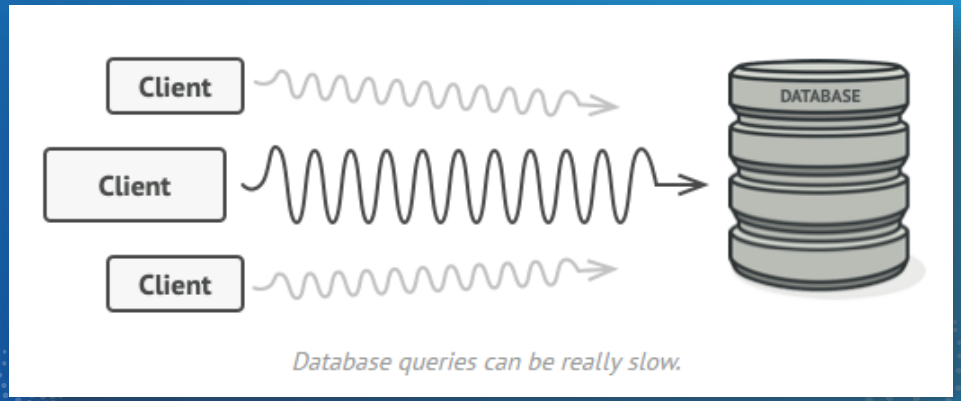
Add a new class to the previous example, so that the customer can provide the MasterCard information: card number, expire date, username and password. If the card is valid, the payment and invoice can proceed, otherwise the system will display error message.

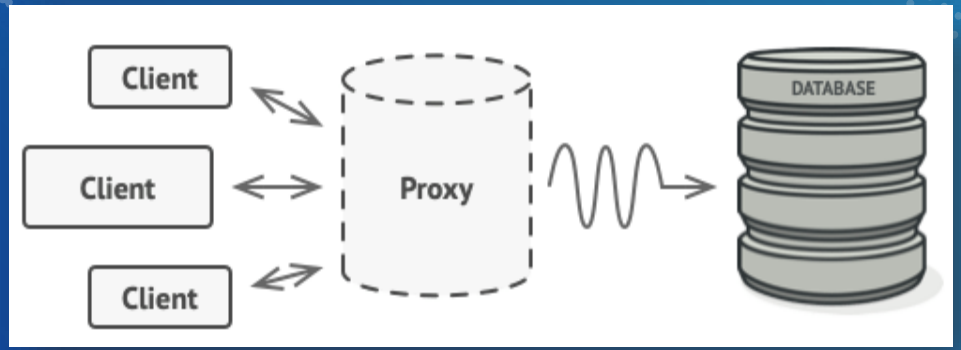
3. Proxy

Proxy

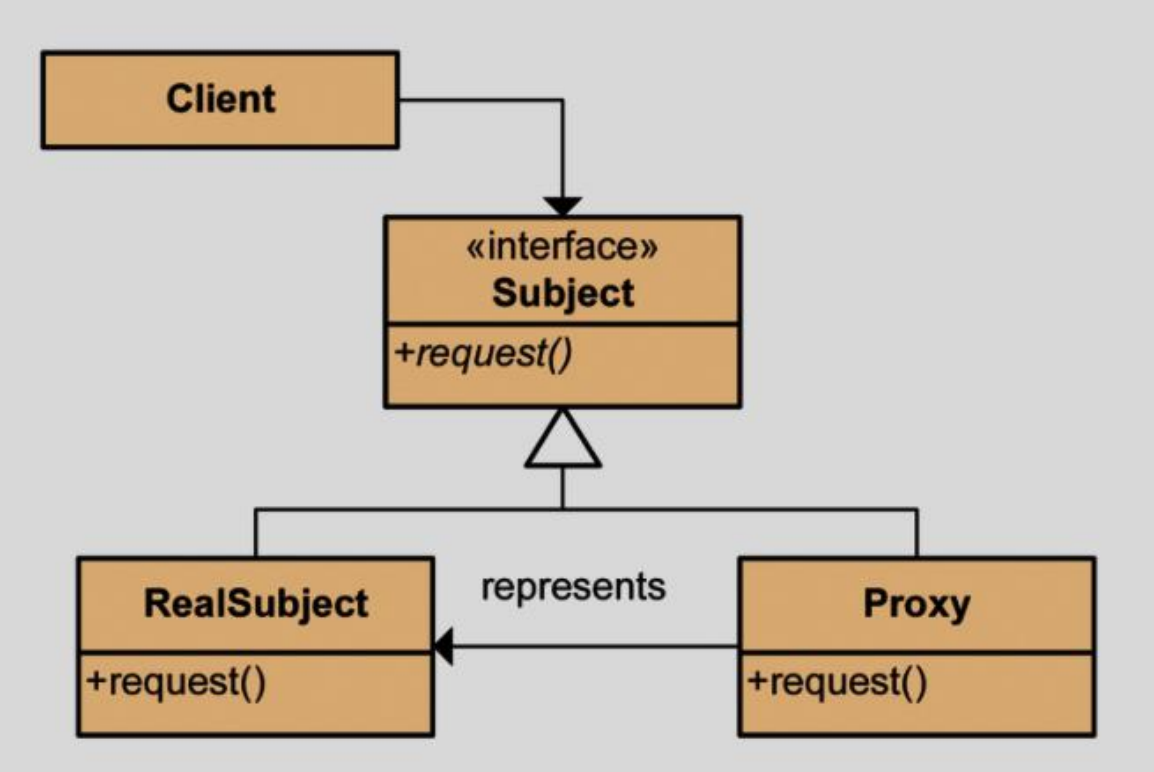
- Gives a way to create a class that represents the functionality of another class.
- Provide a surrogate for another object to control access to it.
- Hides the original object and control access to it.
- Allowing you to perform something either before or after the request gets through to the original object.

Proxy


Client
Client
Client
DATABASE
Database queries can be really slow.


Client
Client
Client
Proxy
DATABASE

Proxy Class Diagram



Proxy Implementation

- Proxy must implement same interface as real object.
- In method implementations of proxy, we implement proxy's functionality before delegating to real object.

Proxy Example [1]

```java
public interface Internet {

    public void connectTo(String serverHost) throws Exception;

}
```

```java
public class RealInternet implements Internet {

    @Override
    public void connectTo(String serverHost) throws Exception {

        System.out.println("Connecting to " + serverHost);

    }

}
```

Proxy Example

```
public class ProxyInternet implements Internet {

    private Internet internet = new RealInternet();
    private static List<String> unauthorizedSites;

    static {
        unauthorizedSites = new ArrayList<String>();
        unauthorizedSites.add("aaa.com");
        unauthorizedSites.add("bbb.com");
        unauthorizedSites.add("ccc.com");
    }

    @Override
    public void connectTo(String serverHost) throws Exception {

        if (unauthorizedSites.contains(serverHost.toLowerCase())) {
            throw new Exception("Access Denied");
        }
        internet.connectTo(serverHost);
    }

}
```

Proxy Example

```
public class TestProxy {

    public static void main(String[] args) {

        Internet internet = new ProxyInternet();

        try {
            internet.connectTo("https://htu.edu.jo/");
            internet.connectTo("aaa.com");
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

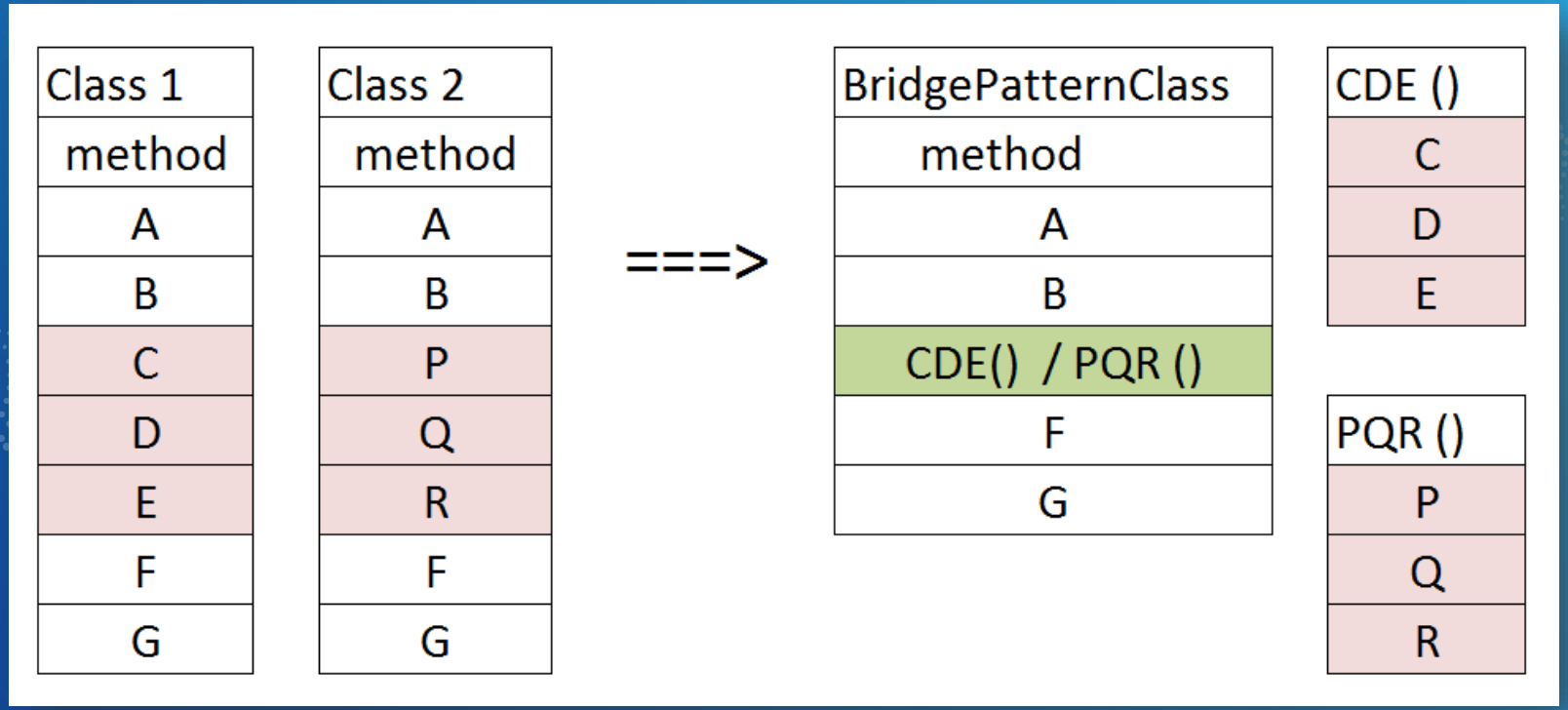
```
Connecting to https://htu.edu.jo/
Access Denied (aaa.com)
```

4. Bridge

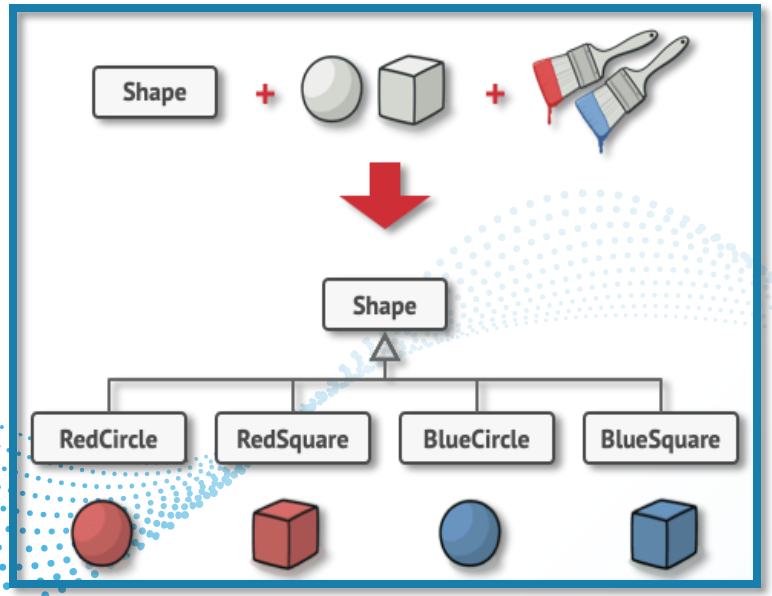
Bridge

- Decouple an abstraction from its implementation so that the two can vary independently.
- Allows you to separate the abstraction from the implementation.
- There are 2 parts in Bridge design pattern:
  - Abstraction
  - Implementation

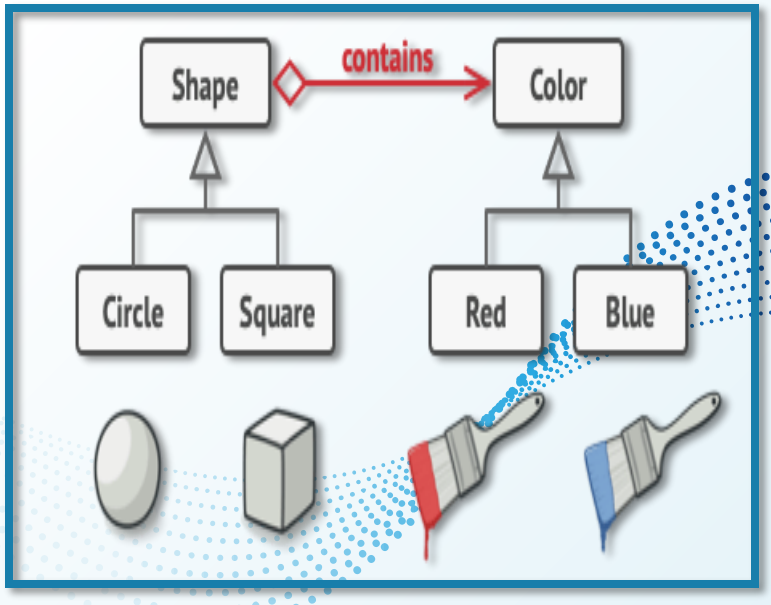
Bridge Example



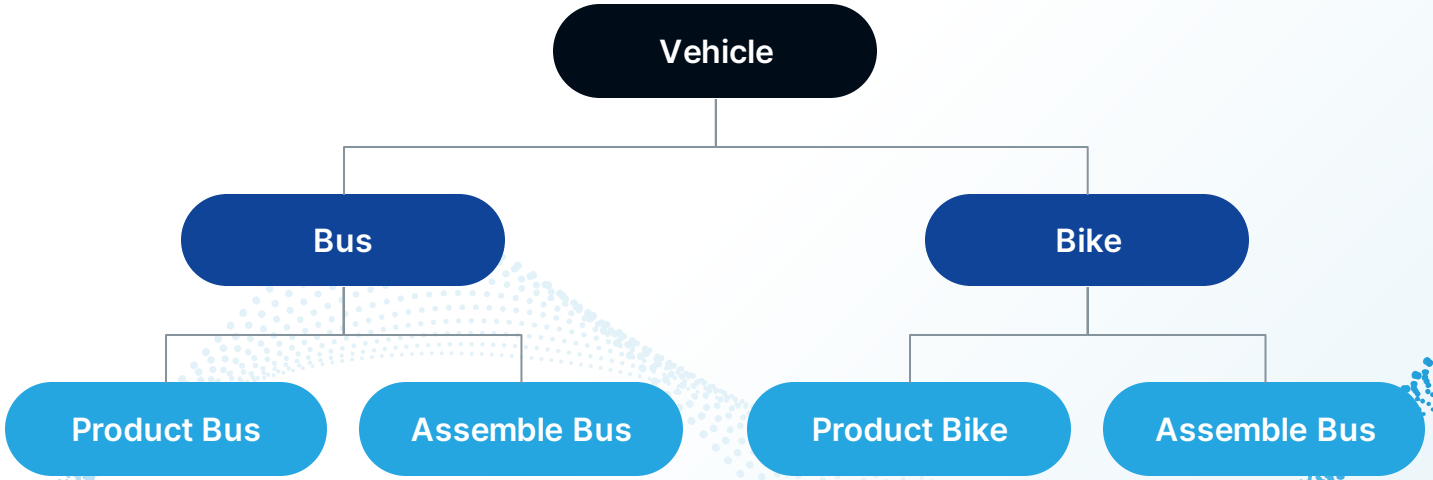
Bridge Example



→ Bridge →



Without Bridge Design Pattern



With Bridge Design Pattern

Bridge Example [2]

```java
public interface Workshop {

    void work();

}
```

```java
public class Produce implements Workshop {

    @Override
    public void work() {
        System.out.print("Produced");

    }

}
```

```java
public class Assemble implements Workshop {

    @Override
    public void work() {
        System.out.print(" And");
        System.out.println(" Assembled.");

    }

}
```

Bridge Example

```
public abstract class Vehicle {

    public Workshop workShop1;
    public Workshop workShop2;

    public Vehicle(Workshop workShop1, Workshop workShop2)
    {
        this.workShop1 = workShop1;
        this.workShop2 = workShop2;
    }

    abstract public void manufacture();
}
```

Bridge Example

```java
public class Car extends Vehicle {

    public Car(Workshop workShop1, Workshop workShop2) {
        super(workShop1, workShop2);
    }

    @Override
    public void manufacture() {
        System.out.print("Car ");
        workShop1.work();
        workShop2.work();
    }

}
```

```java
public class Bike extends Vehicle {

    public Bike(Workshop workShop1, Workshop workShop2) {
        super(workShop1, workShop2);
    }

    @Override
    public void manufacture() {
        System.out.print("Bike ");
        workShop1.work();
        workShop2.work();
    }

}
```

Bridge Example

```
public class BridgePattern {

    public static void main(String[] args) {
        Vehicle vehicle1 = new Car(new Produce(), new Assemble());
        vehicle1.manufacture();
        Vehicle vehicle2 = new Bike(new Produce(), new Assemble());
        vehicle2.manufacture();
    }

}
```

```
Car Produced And Assembled.
Bike Produced And Assembled.
```

Thanks!

Any questions?

References

[1] https://www.geeksforgeeks.org/proxy-design-pattern/

[2] https://www.geeksforgeeks.org/bridge-design-pattern/