



# Adapter Design Patterns



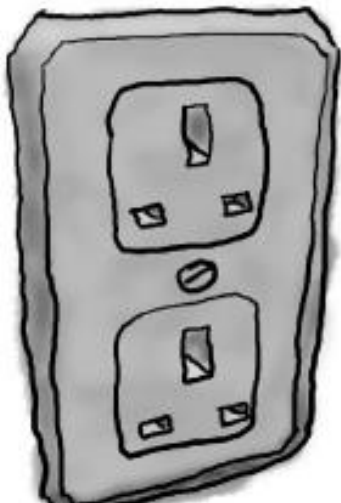
Structural Design Pattern

Lecture 18

# ■ Adapter Design Pattern

- Gang of Four state the intent of Adapter is to
  - *Convert the interface of a class into another interface that the clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.*

European Wall Outlet



AC Power Adapter



Standard AC Plug

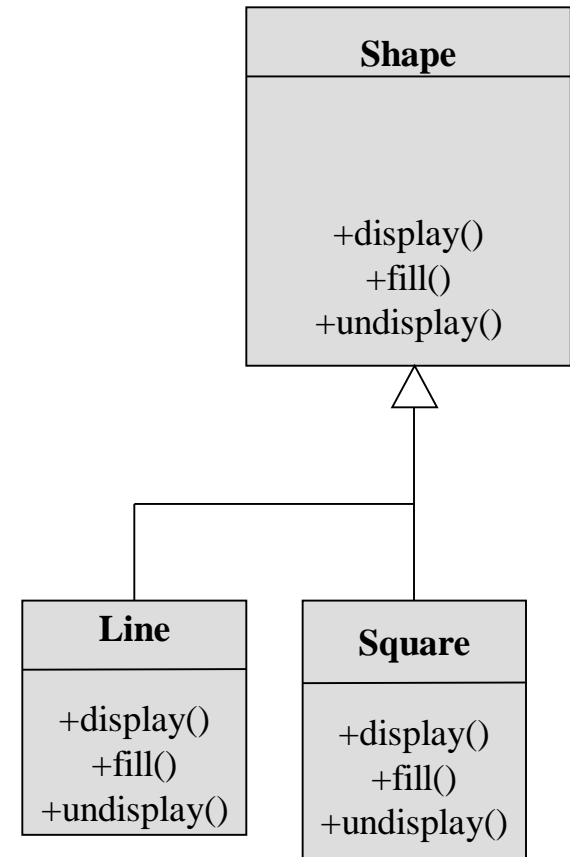


The US laptop expects another interface.

# Problem Specification:

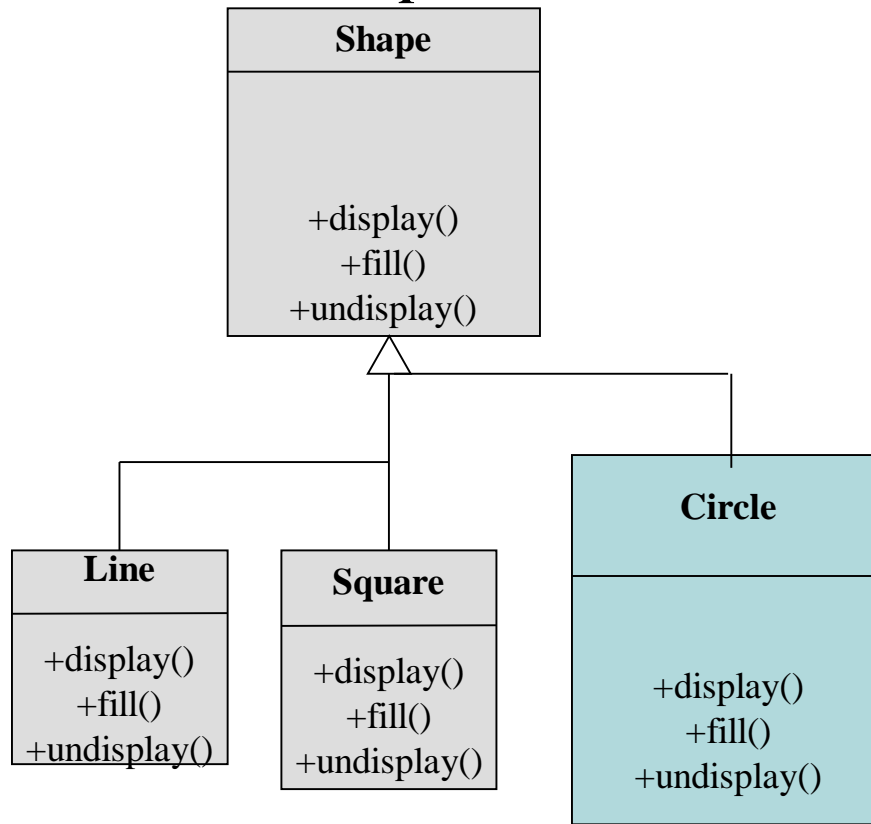
## *Step 1:*

- *You are given the following class library to draw shapes.*



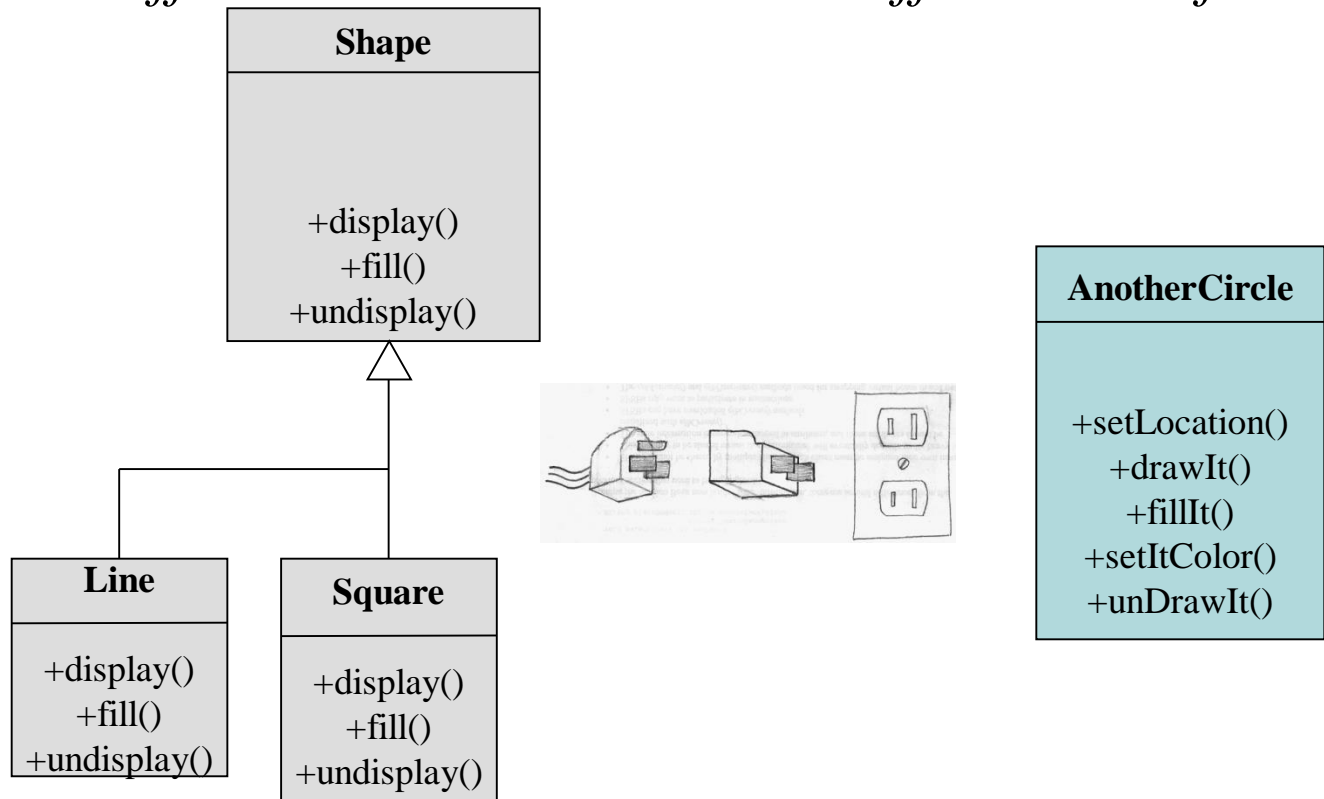
## ■ Step 2:

- *Now you are asked to add another class to deal with circle. Your plan was:*



## Step 3:

- Then your client said: “No,no,no”. You have to use this nice class library for drawing circles. Unfortunately, it is from a different vendor and it has a different interface.



# ■ Adapter Design Pattern

- **Problem:** An "off the shelf" component offers compelling functionality that you would like to reuse, but its "view of the world" is not compatible with the philosophy and architecture of the new system
- Adapter is about creating an intermediary abstraction that translates, or maps, the old component to the new system
- Use Adapter when you need a way to create a new interface for an object that does the right stuff but has the wrong interface

# ■ Two kinds of adapter pattern

---

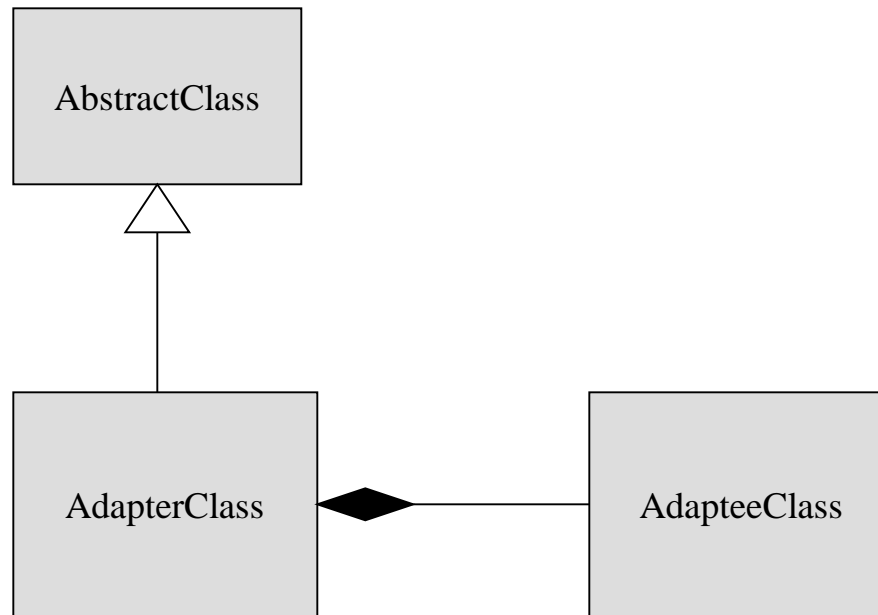
- Object adapter pattern

The adapter class contains adaptee object

- Class adapter pattern

Adapter class is inherited from both adaptee and abstract class.

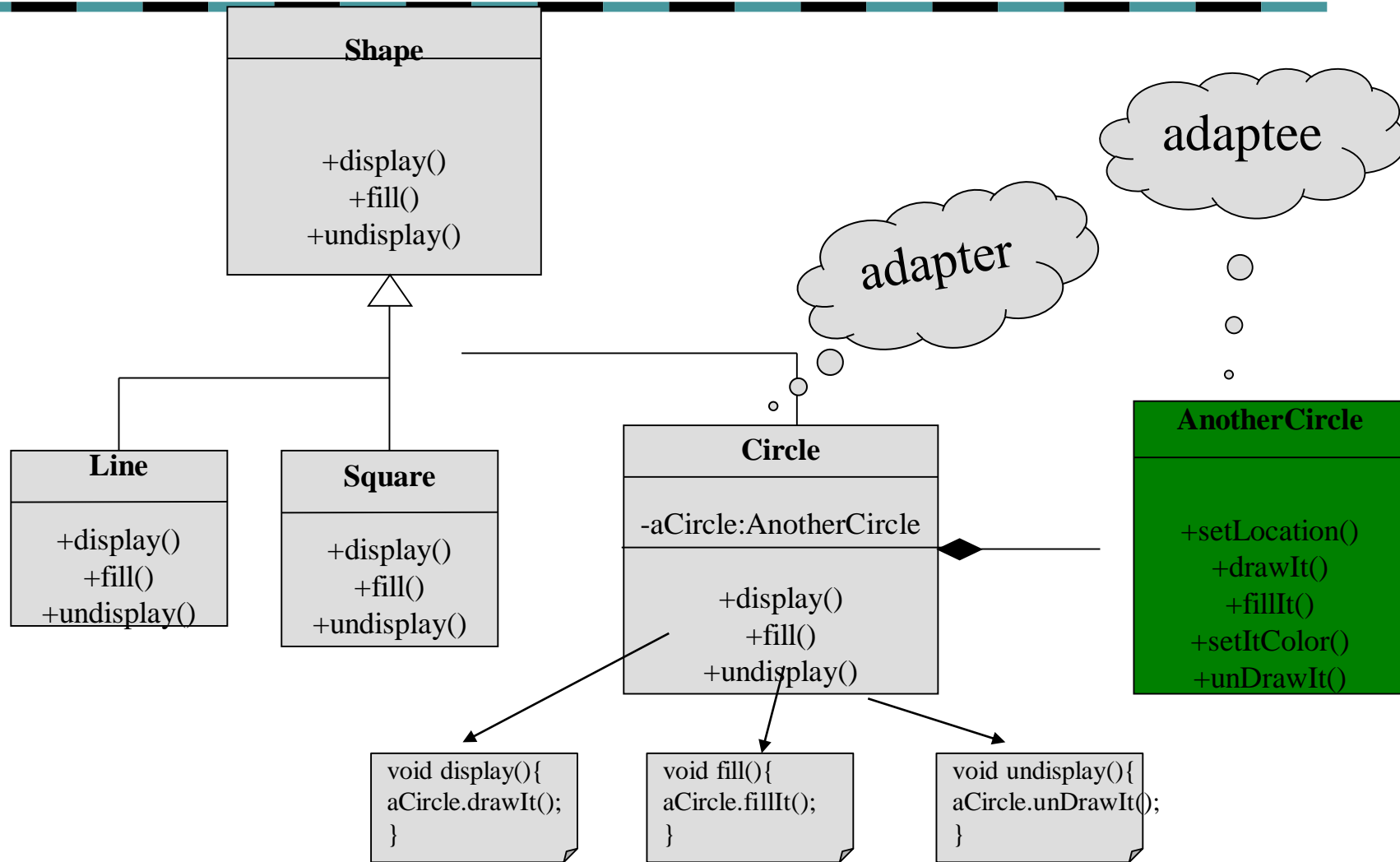
# Object Adapter



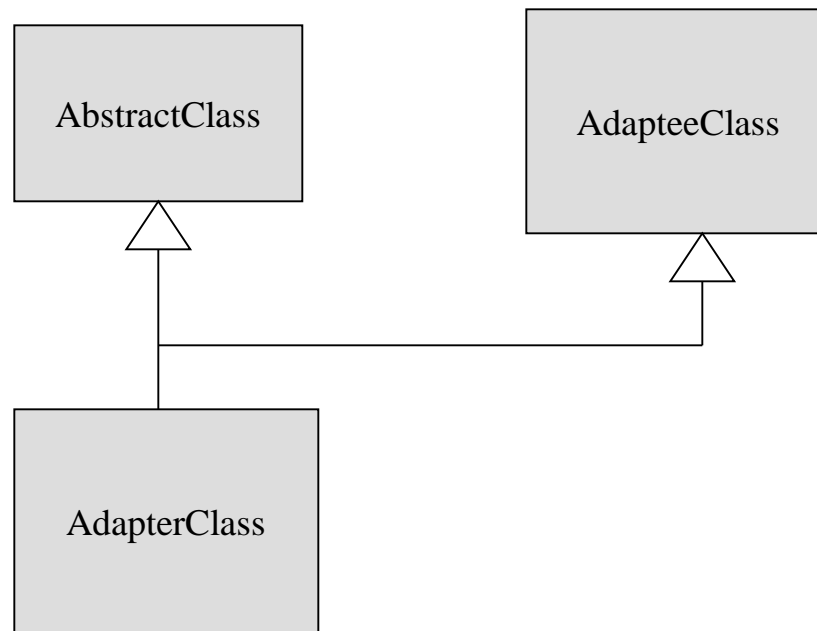


# Adapter Design Pattern Solution

## Object Adapter

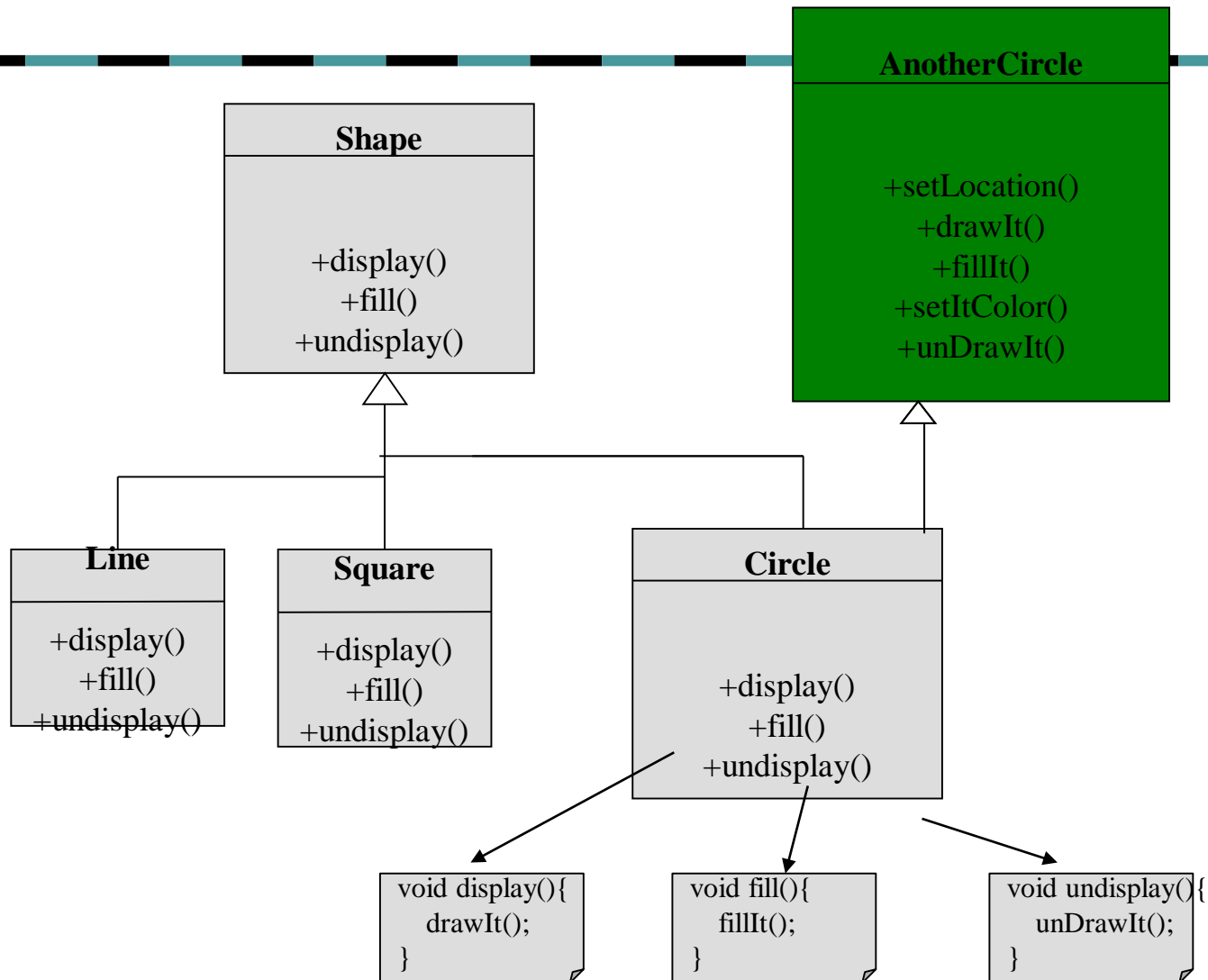


# ■ Class Adapter



# Adapter Design Pattern Solution

## Class Adapter



# ■ Class Adapter Vs Object Adapter

## ■ Class Adapter

- uses *inheritance* and can only wrap a **class**. It cannot wrap an interface since by definition it must derive from some base class.

## ■ Object Adapter

- uses *composition* and can wrap classes or interfaces, or both. It can do this since it contains, as a private, encapsulated member, the class or interface **object instance** it wraps.

# ■ Goal of Adapter Pattern

---

- Keeping the client code intact we need to write a new class which will make use of services offered by the class.
- Convert the services offered by class to the client in such a way that functionality will not be changed and the class will become reusable.

# Flow of Events in Adapter Pattern

- Client call operations on Adaptor instance, which in return call adaptee operations that carry out the request.
- To use an adapter:
  - The client makes a request to the adapter by calling a method on it using the target interface.
  - The adapter translates that request on the adaptee using the adaptee interface.
  - Client receive the results of the call and is unaware of adapter's presence.

# ■ Implementation steps

---

- Identify the desired interface.
- Design a "wrapper" class that can "impedance match" the old to the new.
- The adapter/wrapper class "has a" instance of the legacy class.
- The adapter/wrapper class "maps" (or delegates) to the legacy object.
- The client uses (is coupled to) the new interface.

# ■ Components of Adapter Class

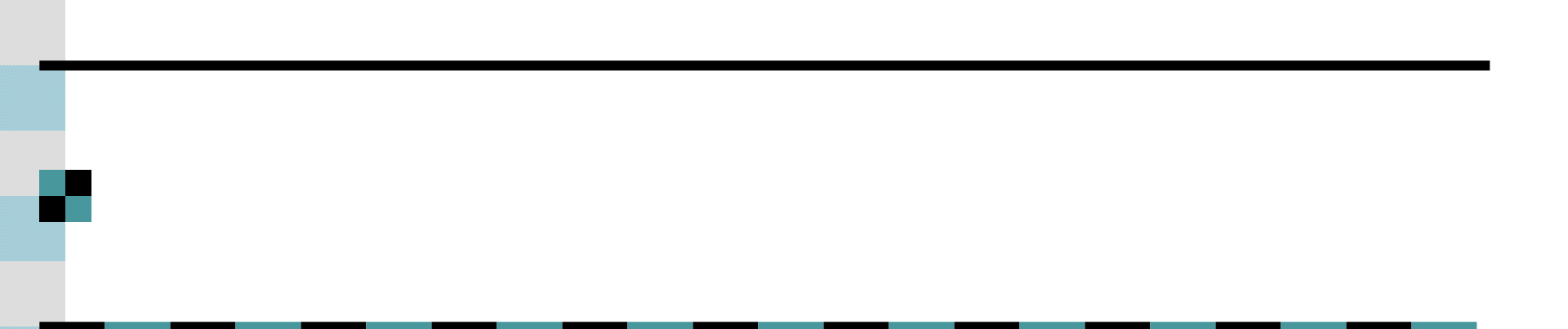
1. **Adaptee:** Defines an existing interface that needs adapting; it represents the component with which the client wants to interact with.
2. **Target:** Defines the domain-specific interface that the client uses; it basically represents the interface of the adapter that helps the client interact with the adaptee.
3. **Adapter:** Adapts the interface Adaptee to the Target interface; in other words, it implements the Target interface, defined above and connects the adaptee, with the client, using the target interface implementation
4. **Client:** The main client that wants to get the operation, is done from the Adaptee.

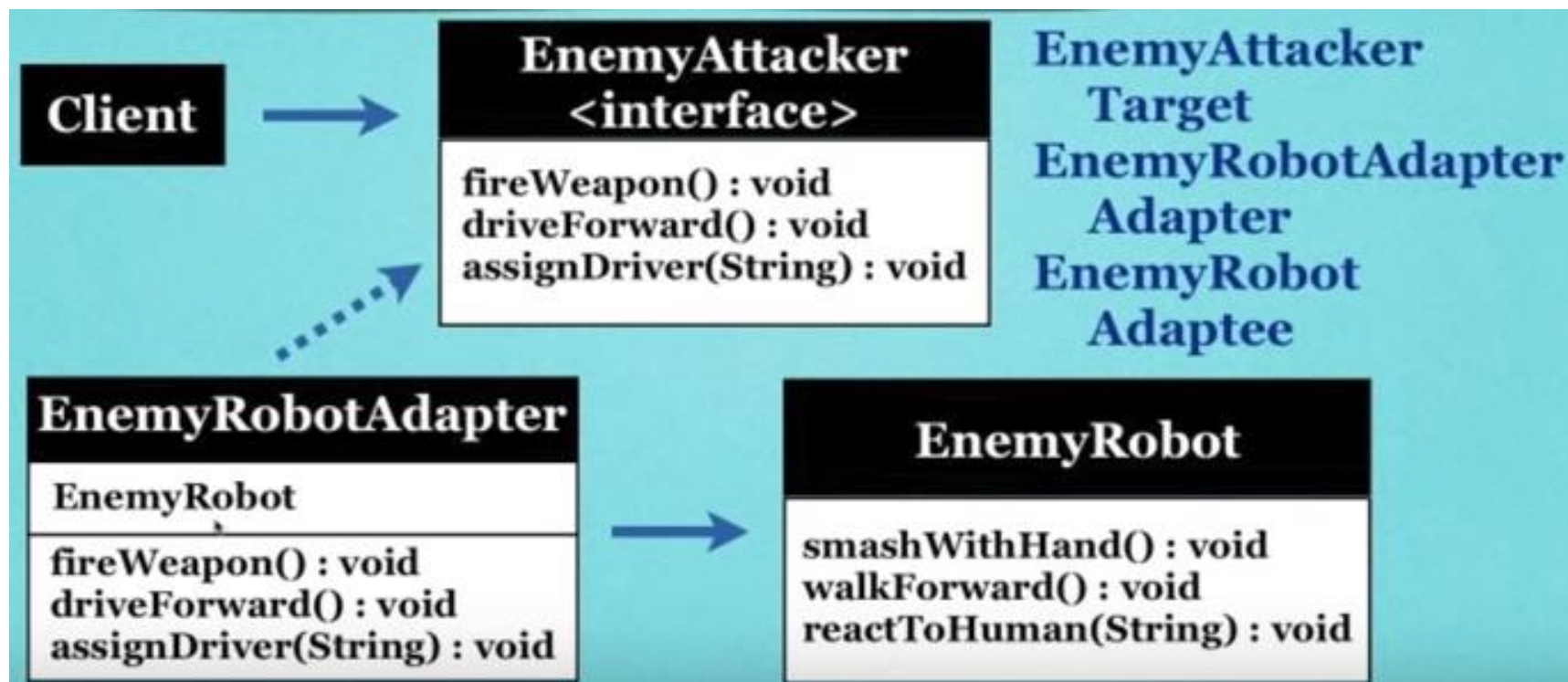


# Limitations of Adapter Design Pattern

---

- Due to adapter class the changes are encapsulated within it and client is decoupled from the changes in the class.

- 
- A client in video game wants to use an enemy attacker. Any enemy attacker can fire a weapon, drive forward and assign driver to it. However, you want to create an artificially intelligent enemy robot. Now, an enemy robot has no need for a driver and also it is not going to drive forward ,it's going to walk forward, and has no weapons it just smashes things with its feet's and its hand and react to human. We want to see enemy robot as an enemy attacker.



# Step 1: Create interface for Enemy Attacker

// This is the Target Interface : This is what the client  
// expects to work with. It is the adapters job to make new  
// classes compatible with this one.

```
public interface EnemyAttacker {  
  
    public void fireWeapon();  
  
    public void driveForward();  
  
    public void assignDriver(String driverName);  
  
}
```

## Step 2: Adaptee

```
// This is the Adaptee. The Adapter sends method calls
// to objects that use the EnemyAttacker interface
// to the right methods defined in EnemyRobot
import java.util.Random;
public class EnemyRobot{
    Random generator = new Random();
    public void smashWithHands() {
        int attackDamage = generator.nextInt(10) + 1;
        System.out.println("Enemy Robot Causes " + attackDamage + " Damage
With Its Hands");    }
    public void walkForward() {
        int movement = generator.nextInt(5) + 1;
        System.out.println("Enemy Robot Walks Forward " + movement + "
spaces"); }
    public void reactToHuman(String driverName) {
        System.out.println("Enemy Robot Tramps on " + driverName);

    }

}
```

# Step 3: Adapter

```
// The Adapter must provide an alternative action for  
// the the methods that need to be used because  
// EnemyAttacker was implemented.
```

```
// This adapter does this by containing an object  
// of the same type as the Adaptee (EnemyRobot)  
// All calls to EnemyAttacker methods are sent  
// instead to methods used by EnemyRobot
```

```
public class EnemyRobotAdapter implements EnemyAttacker{  
    EnemyRobot theRobot;  
    public EnemyRobotAdapter(EnemyRobot newRobot){  
        theRobot = newRobot; }  
    public void fireWeapon() {  
        theRobot.smashWithHands(); }  
    public void driveForward() {  
        theRobot.walkForward(); }  
}
```



---

## ■ Step 3:Adapter

---

```
public void assignDriver(String driverName) {  
  
    theRobot.reactToHuman(driverName);  
  
}  
  
}
```



# Step 4: Client

```
public class TestEnemyAttackers{  
    public static void main(String[] args){  
        EnemyRobot fredTheRobot = new EnemyRobot();  
        EnemyAttacker robotAdapter = new EnemyRobotAdapter(fredTheRobot);  
        System.out.println("The Robot");  
        fredTheRobot.reactToHuman("Paul");  
        fredTheRobot.walkForward();  
        fredTheRobot.smashWithHands();  
        System.out.println();  
        System.out.println("The Robot with Adapter");  
        robotAdapter.assignDriver("Mark");  
        robotAdapter.driveForward();  
        robotAdapter.fireWeapon();  
    }  
}
```



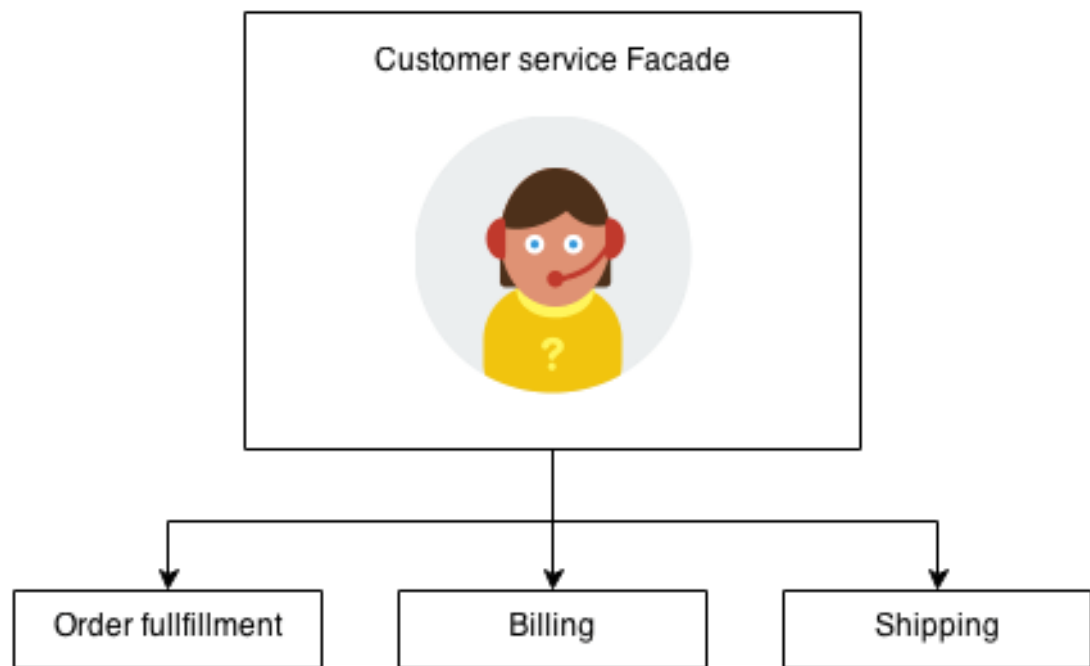
# ■ Output

```
The Robot
Enemy Robot Tramps on Paul
Enemy Robot Walks Forward 4 Spaces
Enemy Robot Causes 4 Damage With Its Hands
The Robot with Adapter
Enemy Robot Tramps on Mark
Enemy Robot Walks Forward 1 Spaces
Enemy Robot Causes 9 Damage With Its Hands
```



# ■ Façade Pattern





# Intent

- A facade or façade is generally one exterior side of a building, usually, but not always, the front. The word comes from the French language, literally meaning “frontage” or “face”.
- Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.



# Problem/Solution pair - Facade

---

- What problems can the Facade design pattern solve?
  - To make a complex subsystem easier to use, a simple interface should be provided for a set of interfaces in the subsystem.
  - The dependencies on a subsystem should be minimized.

# Problem/Solution pair – Façade (Continued)

- What solution does the Facade design pattern describe?
  - Define a Facade object that implements a simple interface in terms of (by delegating to) the interfaces in the subsystem and may perform additional functionality before/after forwarding a request.

# Participants

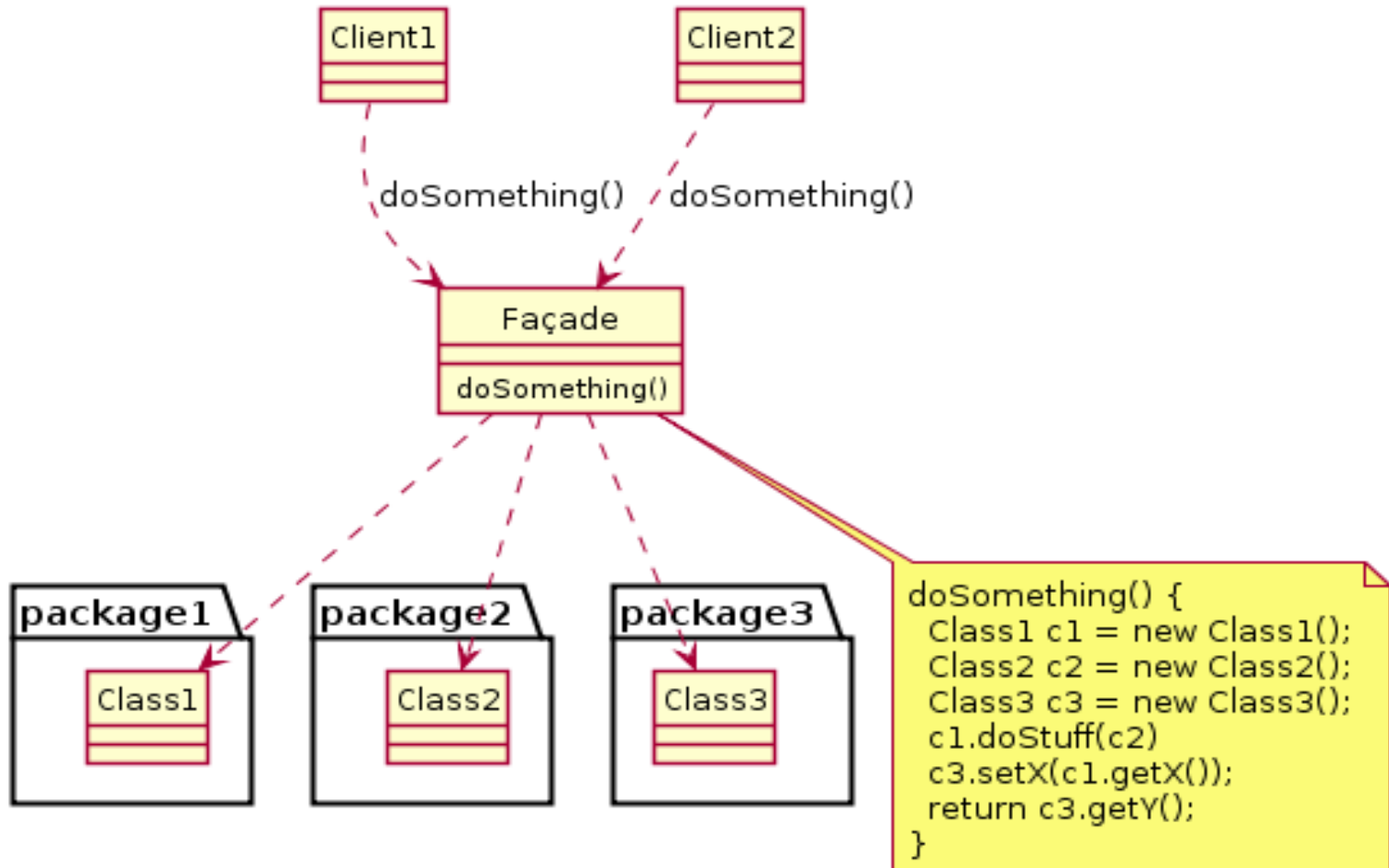
- **Facade:**

- Knows which subsystem classes are responsible for a request. Delegates client requests to appropriate subsystem objects.

- **Subsystem classes:**

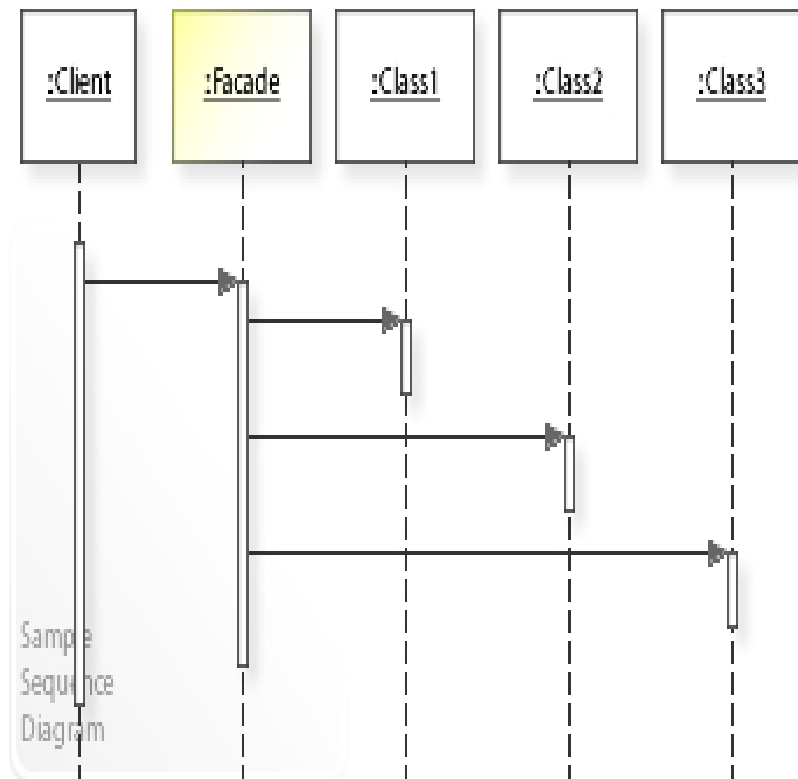
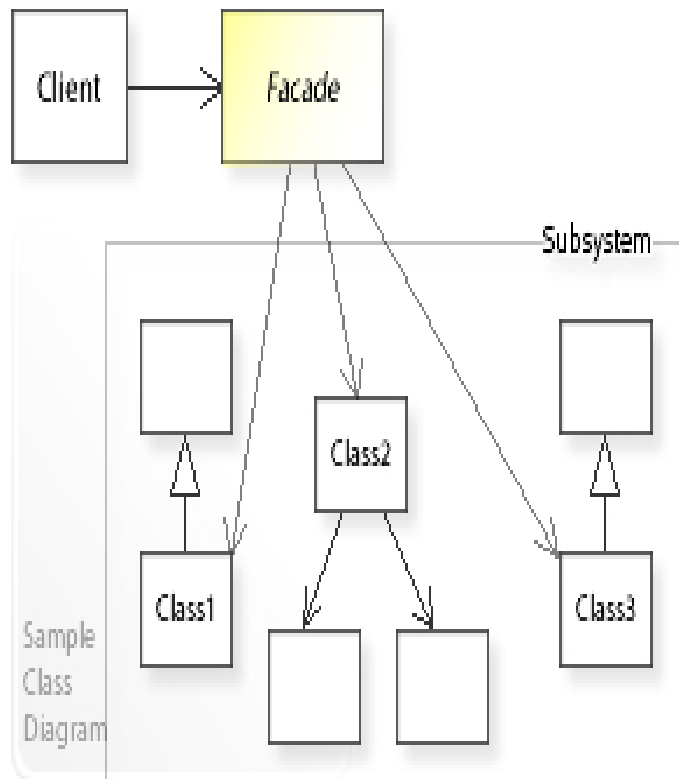
- Implements subsystem functionality.
- Handle work assigned by Facade object.
- Have no knowledge of the facade; that is, they keep no references to it.

# ■ Example – Façade





# Example - Facade



# ■ Usage

---

- Provide simple interface to complex subsystems
- When there are many dependencies between clients and implementation class
- You want to neatly layer your subsystem

# ■ Example

- Let's consider a hotel. This hotel has a hotel keeper. There are a lot of restaurants inside hotel e.g. Veg restaurants, Non-Veg restaurants and Veg/Non Both restaurants.
- You, as client want access to different menus of different restaurants . You do not know what are the different menus they have. You just have access to hotel keeper who knows his hotel well. Whichever menu you want, you tell the hotel keeper and he takes it out of from the respective restaurants and hands it over to you. Here, the hotel keeper acts as the **facade**, as he hides the complexities of the system hotel.

# ■ Interface of Hotel

---

```
package structural.facade;  
public interface Hotel  
{  
    public Menu getMenu();  
}
```

# VegRestaurant.java

---

```
package structural.facade;
```

```
public class VegRestaurant implements Hotel
{
    public Menu getMenu()
    {
        VegMenu v = new VegMenu();
        return v;
    }
}
```

# ■ NonVegRestaurant.java

```
package structural.facade;
```

```
public class NonVegRestaurant implements Hotel
{
    public Menu getMenu()
    {
        NonVegMenu nv = new NonVegMenu();
        return nv;
    }
}
```

# VegNonBothRestaurant.java

```
package structural.facade;
```

```
public class VegNonBothRestaurant implements Hotel
{
    public Menu getMenu()
    {
        Both b = new Both();
        return b;
    }
}
```

# HotelKeeper.java

```
package structural.facade;
```

```
public class HotelKeeper
```

```
{  
    public VegMenu getVegMenu()    {  
        VegRestaurant v = new VegRestaurant();  
        VegMenu vegMenu = (VegMenu)v.getMenus();  
        return vegMenu;    }  
    public NonVegMenu getNonVegMenu() {  
        NonVegRestaurant v = new NonVegRestaurant();  
        NonVegMenu NonvegMenu =  
(NonVegMenu)v.getMenus();  
        return NonvegMenu;    }  
  
    public Both getVegNonMenu()    {  
        VegNonBothRestaurant v = new VegNonBothRestaurant();  
        Both bothMenu = (Both)v.getMenus();  
        return bothMenu;    } }
```



# How will the client program access this façade?

```
package structural.facade;
```

```
public class Client
```

```
{
```

```
    public static void main (String[] args)
```

```
    {
```

```
        HotelKeeper keeper = new HotelKeeper();
```

```
        VegMenu v = keeper.getVegMenu();
```

```
        NonVegMenu nv = keeper.getNonVegMenu();
```

```
        Both b= keeper.getVegNonMenu();
```

```
    }
```

```
}
```