# Adapter Pattern - Bridging the Gap

## What is the Adapter Pattern?

The **Adapter Pattern** is a structural design pattern that allows objects with incompatible interfaces to work together. It acts as a bridge, converting one interface into another expected by the client.

The Adapter Pattern is particularly useful when integrating new components into an existing system without modifying its codebase.

---

## Key Components of the Adapter Pattern

1. **Target Interface**: The interface expected by the client.
2. **Adapter**: The class that bridges the gap between the Target Interface and the Adaptee.
3. **Adaptee**: The class that needs to be adapted to work with the Target Interface.
4. **Client**: The class that interacts with the Target Interface.

---

## Why Use the Adapter Pattern?

- **Reusability**: Reuse existing functionality by adapting its interface.
- **Flexibility**: Integrate third-party or legacy code into a new system.
- **Decoupling**: Decouples the client from specific implementations of the Adaptee.

---

## How It Works

1. Define the **Target Interface** that the client expects.
2. Create the **Adaptee** class, which has an incompatible interface.
3. Implement the **Adapter** class to bridge the gap by implementing the Target Interface and delegating calls to the Adaptee.
4. The **Client** interacts only with the Target Interface.

---

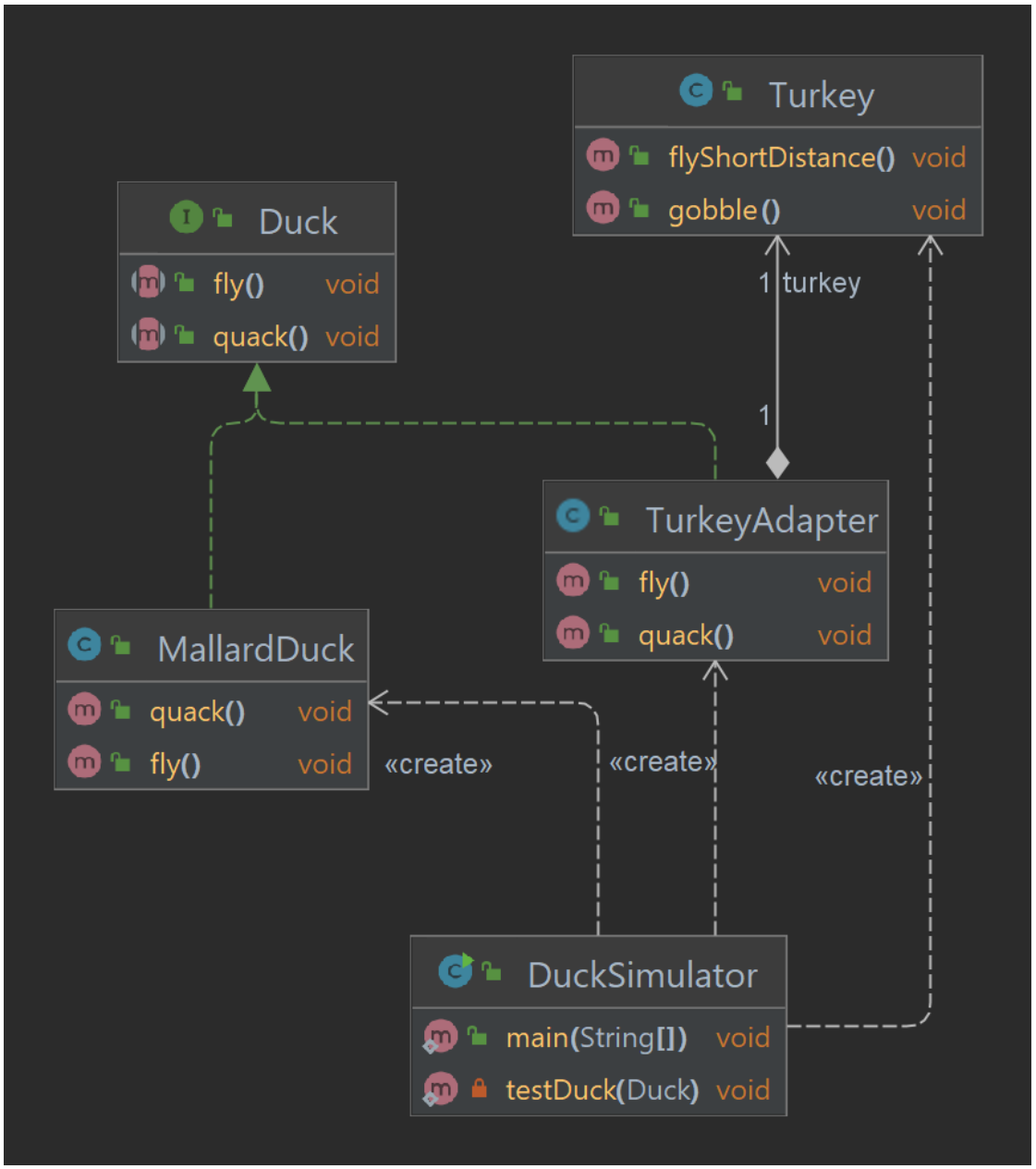
## Example: Duck Adapter

Consider a scenario where a client expects a `Duck` interface but we need to adapt a `Turkey` class to fit into this system.

## Project Structure

```
src/main/java/com/headfirst/chapter7/adapterPattern/
├── duck/
│   ├── Duck.java                        # Target Interface
│   ├── MallardDuck.java                 # Duck Implementation
│   ├── Turkey.java                      # Adaptee
│   ├── TurkeyAdapter.java               # Adapter
│   └── DuckSimulator.java               # Client
├── payment/
│   ├── PaymentProcessor.java            # Target Interface
│   ├── ThirdPartyPaymentGateway.java # Adaptee
│   ├── PaymentGatewayAdapter.java       # Adapter
│   └── PaymentProcessorClient.java      # Client
```

## Class Diagram



## Example-Code

### Target Interface

```
public interface Duck { void
    quack();
    void fly();
}
```

### Adaptee

```
public class Turkey {
    public void gobble() {
        System.out.println("Turkey gobbles!");
    }

    public void flyShortDistance() { System.out.println("Turkey
        flies a short distance.");
    }
}
```

### Adapter

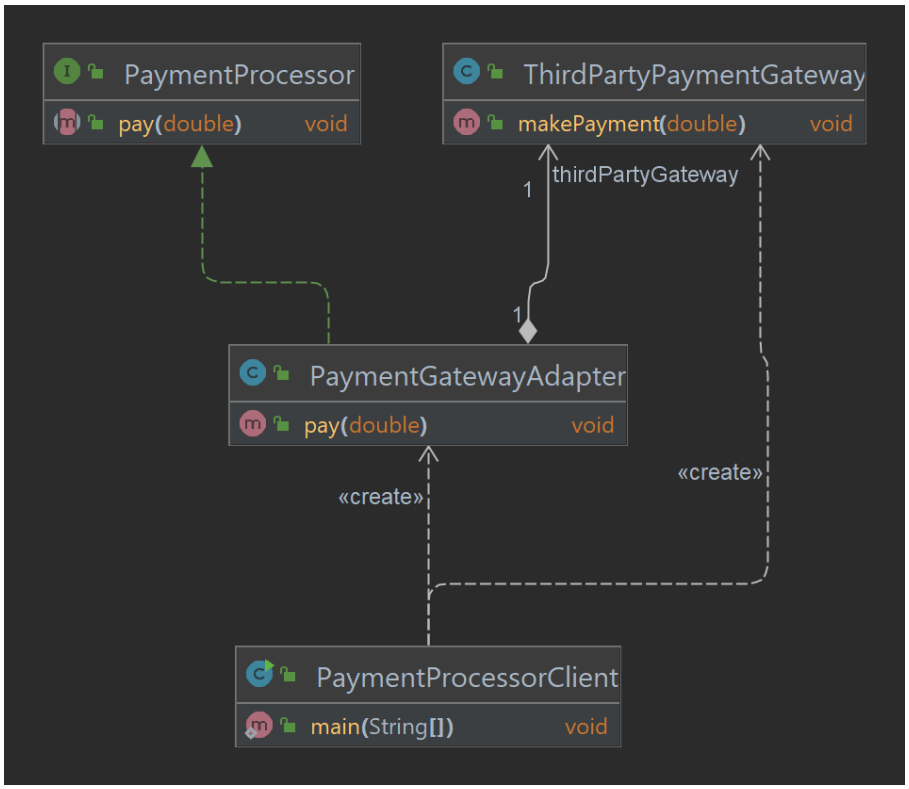
```
public class TurkeyAdapter implements Duck {
    private final Turkey turkey;

    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }

    @Override
    public void quack() {
        turkey.gobble();
    }

    @Override
    public void fly() {
        for (int i = 0; i < 5; i++) { turkey.flyShortDistance();
        }
    }
}
```

### Client

```
public class DuckSimulator {
    public static void main(String[] args) { Duck
        mallardDuck = new MallardDuck(); Turkey
        wildTurkey = new Turkey();
        Duck turkeyAdapter = new TurkeyAdapter(wildTurkey);

        System.out.println("The Turkey says...");
        wildTurkey.gobble(); wildTurkey.flyShortDistance();

        System.out.println("\nThe Duck says...");
        testDuck(mallardDuck);

        System.out.println("\nThe TurkeyAdapter says...");
        testDuck(turkeyAdapter);
    }

    private static void testDuck(Duck duck) {
        duck.quack();
        duck.fly();
    }
}
```

# Real-World Applications

### Class Diagram



## Example 1: Payment Gateways

Imagine integrating a third-party payment processor into an existing e-commerce platform. The **Adapter** can map the existing payment interface to the third-party API, ensuring seamless integration.

### Target Interface

```
public interface PaymentProcessor { void
    pay(double amount);
}
```

### Adaptee

```
// Third-party payment gateway with a different API
public class ThirdPartyPaymentGateway {
    public void makePayment(double amountInDollars) {
        System.out.println("Payment of $" + amountInDollars + " made via
        Third-Party Gateway.");
    }
}
```

### Adapter

```
// Adapter to bridge the common interface and third-party gateway
public class PaymentGatewayAdapter implements PaymentProcessor {
    private final ThirdPartyPaymentGateway thirdPartyGateway;

    public PaymentGatewayAdapter(ThirdPartyPaymentGateway
        thirdPartyGateway) {
        this.thirdPartyGateway = thirdPartyGateway;
    }

    @Override
    public void pay(double amount) {
        thirdPartyGateway.makePayment(amount);
    }
}
```

### Client

```
public class PaymentProcessorClient {
    public static void main(String[] args) {
        // Adaptee: Third-party payment gateway
        ThirdPartyPaymentGateway thirdPartyGateway = new
        ThirdPartyPaymentGateway();

        // Adapter: Adapts the third-party gateway to the common
        interface
```

```
        PaymentProcessor paymentProcessor = new
        PaymentGatewayAdapter(thirdPartyGateway);

        // Client: Uses the common PaymentProcessor interface
        paymentProcessor.pay(150.75); // Payment of $150.75 made via
        Third-Party Gateway.
    }
}
```

### Example 2: File Format Conversions

When working with different file formats (e.g., XML and JSON), an **Adapter** can translate one format to another, enabling compatibility between systems.

### Example 3: Legacy System Integration

Legacy systems often use outdated interfaces. The **Adapter** bridges these older interfaces with modern applications, allowing them to work together without modifying the legacy code.

---

## Summary Table

| Component | Responsibility |
|---|---|
| Target Interface | Defines the expected interface |
| Adapter | Bridges the gap between Target and Adaptee |
| Adaptee | The incompatible class being adapted |
| Client | Uses the Target Interface |

The Adapter Pattern is your go-to solution for making incompatible interfaces work together seamlessly. It's an essential tool for modernizing legacy systems and integrating external libraries.

# Facade Pattern–Simplifying-Interactions

## What is the Facade Pattern?

The **Facade Pattern** is a structural design pattern that provides a simplified interface to a larger body of code, hiding the complexities of the subsystems behind it.

The Facade acts as a high-level interface that makes a subsystem easier to use, allowing clients to interact with the system through the facade instead of dealing with its complex structure.

---

## Key Components of the Facade Pattern

1. **Facade**: Provides a simplified interface to the subsystems.
2. **Subsystems**: Complex components or services that the Facade simplifies.
3. **Client**: Interacts with the Facade rather than the subsystems directly.

---

## Why Use the Facade Pattern?

- **Simplified Interface**: Provides an easy-to-use interface for clients.
- **Loose Coupling**: Decouples clients from complex subsystem implementations.
- **Improved Maintainability**: Changes in subsystems are isolated from the client.

---

## How It Works

1. Identify subsystems that need simplification.
2. Create a **Facade** class that provides high-level methods to interact with the subsystems.
3. Use the Facade in the client to hide subsystem complexities.

---

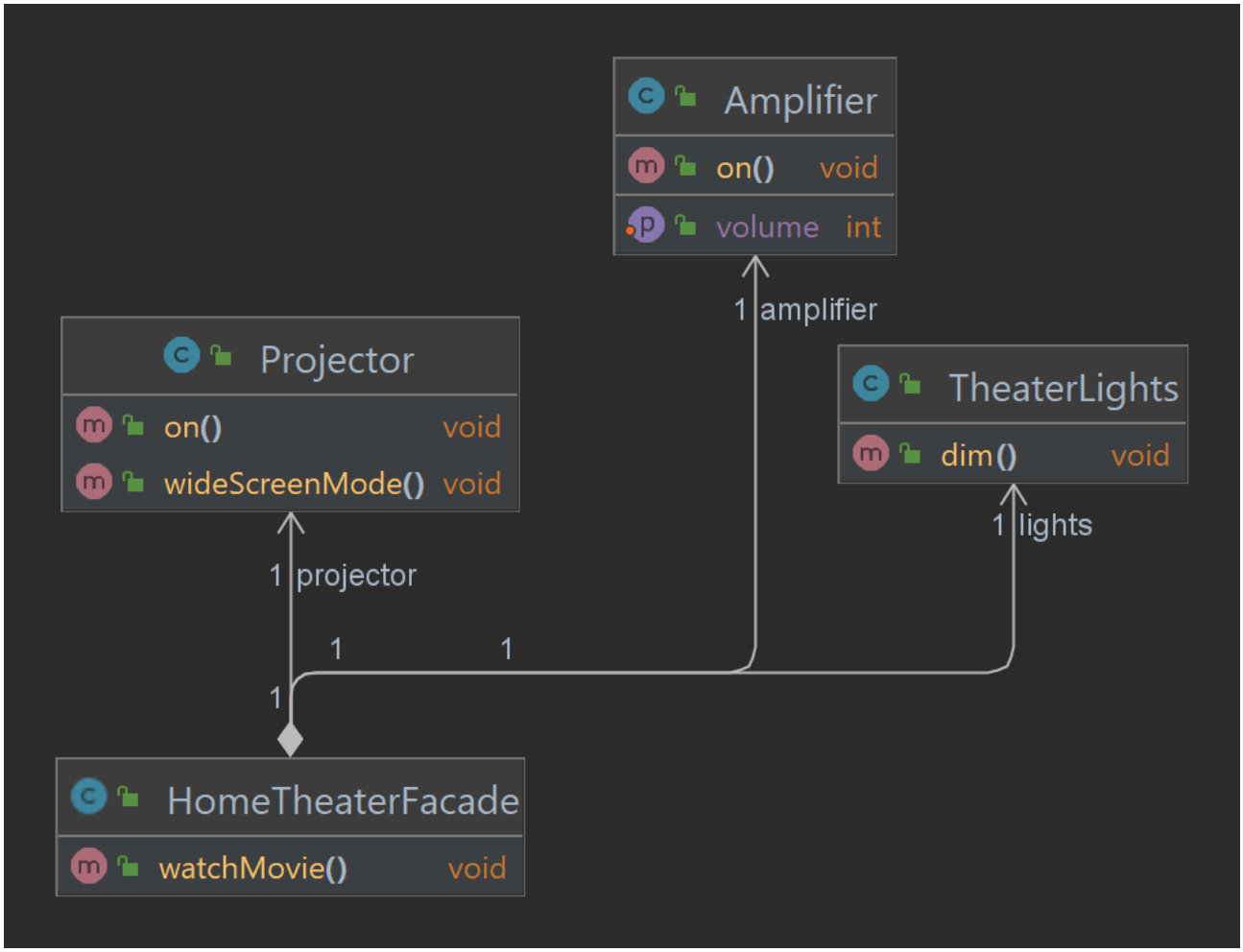
## Example: Home Theater System

Consider a complex home theater system with multiple components such as an amplifier, projector, lights, and speakers. The Facade Pattern simplifies the process of turning on the theater.

## Project Structure

```
src/main/java/com/headfirst/chapter7/facadePattern/
├── subsystems/
│   ├── Amplifier.java              # Subsystem: Amplifier
│   ├── Projector.java              # Subsystem: Projector
│   ├── TheaterLights.java          # Subsystem: Theater Lights
├   ├── HomeTheaterFacade.java      # Facade Class
└── HomeTheaterTest.java            # Client
```

## Class Diagram




## Example-Code

### Subsystems

```java
public class Amplifier {
    public void on() { System.out.println("Amplifier
        is ON");
    }

    public void setVolume(int level) { System.out.println("Setting
```

```
        volume to " + level);
    }
}

public class Projector {
    public void on() { System.out.println("Projector
        is ON");
    }

    public void wideScreenMode() { System.out.println("Setting
        projector to widescreen mode");
    }
}

public class TheaterLights {
    public void dim() {
    System.out.println("Dimming theater lights");
    }
}
```

**Facade**

```
public class HomeTheaterFacade { private
    final Amplifier amplifier; private final
    Projector projector; private final
    TheaterLights lights;

    public HomeTheaterFacade(Amplifier amplifier, Projector
        projector, TheaterLights lights) {
        this.amplifier = amplifier;
        this.projector = projector;
        this.lights = lights;
    }

    public void watchMovie() {
        System.out.println("Getting ready to watch a movie...");
        lights.dim();
        amplifier.on(); amplifier.setVolume(10);
        projector.on(); projector.wideScreenMode();
        System.out.println("Movie is starting!");
    }
}
```

**Client**

```
public class HomeTheaterTest {
    public static void main(String[] args) { Amplifier
        amplifier = new Amplifier(); Projector projector =
        new Projector(); TheaterLights lights = new
        TheaterLights();

        HomeTheaterFacade homeTheater = new
        HomeTheaterFacade(amplifier, projector, lights);
        homeTheater.watchMovie();
    }
}
```

# Real-World Applications

## Example 1: Simplifying API Usage

Imagine integrating a complex third-party API that involves multiple classes and configurations. A **Facade** can wrap the intricate details into simple methods, such as: - `initPaymentGateway()` - `processTransaction()` This simplifies the developer experience, reducing errors and improving productivity.

## Example 2: Library Wrappers

Consider a graphics library where rendering an image requires multiple steps, such as setting up a canvas, loading resources, and drawing shapes. A **Facade** can provide a method like `renderImage(String filePath)` that handles everything internally.

## Example 3: Subsystem Management

In enterprise software, a Facade can simplify access to subsystems like authentication, logging, and notification services. For example, a `UserManagementFacade` can provide methods like `registerUser()` and `sendWelcomeEmail()`, hiding complex interactions with multiple subsystems.

---

# Summary Table

| Component | Responsibility |
|---|---|
| Facade | Simplifies interaction with subsystems |
| Subsystems | Perform the actual operations |

| | |
|---|---|
| Client | Uses the Facade to interact with the subsystems |

---

The Facade Pattern is perfect for creating a simplified interface for complex subsystems, improving usability and maintainability.