Adapter Pattern

Idaho State University | Computer Science

Isaac Griffith

CS 2263
Department of Informatics and Computer Science
Idaho State University

ROAR

Outcomes

After today's lecture you will be able to:

- Understand the use of the Adapter Design Pattern
- Use and implement the Adapter Pattern

Inspiration

“Every program has (at least) two purposes: the one for which it was written, and another for which it wasn’t.” – Alan J. Perlis

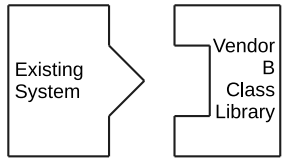
Adapters in the Real World

- Our next pattern provides techniques for converting an interface that is not compatible with an existing system into a different interface that is
  - Real World Example: AC Power Adapters
  - Electronic products made for the USA cannot be used directly with electrical outlets found in most other parts of the world
  - US 3-prong (grounded) plugs are not compatible with European wall outlets
  - To use, you need either
    - an AC power adapter, if the US product has a “universal” power supply, or
    - an AC power converter/adapter, if it doesn’t
- By example, OO adapters may simply provide adaptation services from one interface to another, or may require more smarts to convert information from one interface passing it to the second interface

OO Adapters (I)

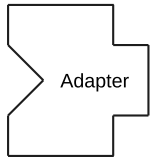
- Pre-Condition: You are maintaining an existing system that makes use of a third-party class library from Vendor A
- Stimulus: Vendor A goes belly up and corporate policy does not allow you to make use of an unsupported class library
- Response: Vendor B provides a similar class library but its interface is completely different from the interface provided by vendor A
- Assumptions: You don't want to change your code, and you can't change vendor B's code.
- Solution?: Write new code that adapts vendor B's interface to the interface expected by your original code

OO Adapters (II)



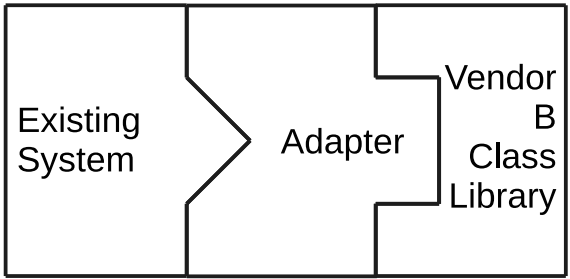
Interface Mismatch
Need Adapter

Create Adapter



And then...

OO Adapters (III)



...plug it in

Benefit: Existing system and new vendor library do not change, new code is isolated within the adapter.

A Turkey Hiding Among Ducks! (I)

- If it walks like a duck and quacks like a duck, then it must be a duck!

A Turkey Hiding Among Ducks! (II)

- If it walks like a duck and quacks like a duck, then it **might** be a **turkey wrapped with a duck adapter**... (!)
- Recall the Duck simulator from HFDP Ch. 1?

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

public class MallardDuck implements Duck {
  public void quack() {
    System.out.println("Quack");
  }

  public void fly() {
    System.out.println("I'm flying");
  }
}
```

A Turkey Hiding Among Ducks! (III)

- An interloper wants to invade the simulator

```
public interface Turkey {
  void gobble();
  void fly();
}

public class WildTurkey implements Turkey {
  public void gobble() {
    System.out.println("Gobble Gobble");
  }

  public void fly() {
    System.out.println("I'm flying a short distance");
  }
}
```

A Turkey Hiding Among Ducks! (IV)

- Write an adapter, that makes a turkey look like a duck

```
public class TurkeyAdapter implements Duck {

  private Turkey turkey;

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

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

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

1. Adapter implements target interface (`Duck`)
2. Adaptee (`turkey`) is passed via constructor and stored internally
3. Calls by client code are delegated to the appropriate methods in the adaptee
4. Adapter is full-fledged class, could contain additional vars and methods to get its job done

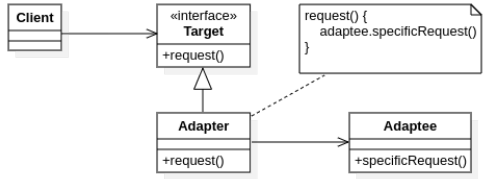
Adapter Pattern: Definition

- The Adapter pattern converts the interface of a class into another interface that clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces
  - The client makes a request on the adapter by invoking a method from the target interface on it
  - The adapter translates that request into one or more calls on the adaptee using the adaptee interface
  - The client receives the results of the call and never knows there is an adapter doing the translation

Adapter Pattern: Structure (I)

Object Adapter

1. Client codes to an interface, not an implementation. Allows creation of multiple adapter classes, if needed.
2. Adapter makes use of composition to access the behavior of Adaptee. We can pass any subclass of Adaptee to the Adapter, if needed.



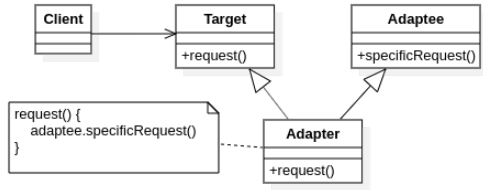
Adapter Pattern: Structure (II)

Class Adapter

1. Requires use of multiple inheritance, but now adapter does not need to re-implement target and/or adaptee behavior.

   It simply overrides or inherits that behavior instead.

   Trade-Offs?



Real World Adapters

- Before Java's new collection classes, iteration over a collection occurred via `java.util.Enumeration`
  - `hasMoreElements() : boolean`
  - `nextElement() : Object`
- With the collection classes, iteration was moved to a new interface: `java.util.Iterator`
  - `hasNext() : boolean`
  - `next() : Object`
  - `remove() : void`
- There's a lot of code out there that makes use of the `Enumeration` interface
  - New code can still make use of that code by creating an adapter that converts from the `Enumeration` interface to the `Iterator` interface
    - **Demonstration**

Difference between Adapter and Decorator

- Adapter and Decorator's seem similar: how so?
- Answers
  - They both wrap objects at run-time
  - They both delegate requests to their wrapped objects
- How are they different?
- Answers
  - Adapter converts one interface into another while maintaining functionality
  - Decorator leaves the interface alone but adds new functionality
    - Decorators are designed to be "stacked"; that's less likely to occur with adapters

Wrapping Up

- Adapter Allows you to covert one interface into another, allowing the client code and the adaptee to remain unchanged
- Decorator seen in new light: an adapter that “converts” an interface into itself while adding new behaviors

Are there any questions?