Adapter

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 4448/6448 — Lecture 23 — 11/13/2007

Lecture Goals

- Cover Material from Chapter 7 of the Design Patterns Textbook
  - Adapter Pattern
  - Facade Pattern

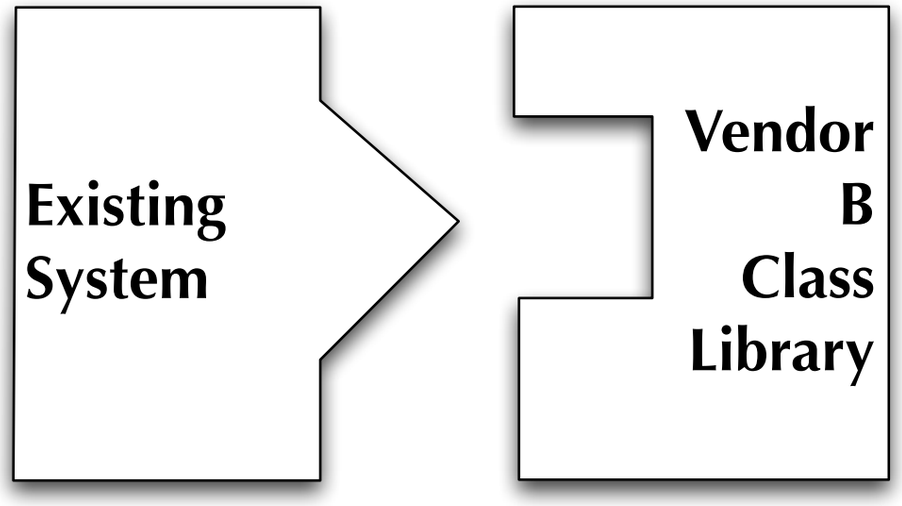
Adapters in the Real World

- 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 convertor/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 before passing it to the second interface

OO Adapters (I)

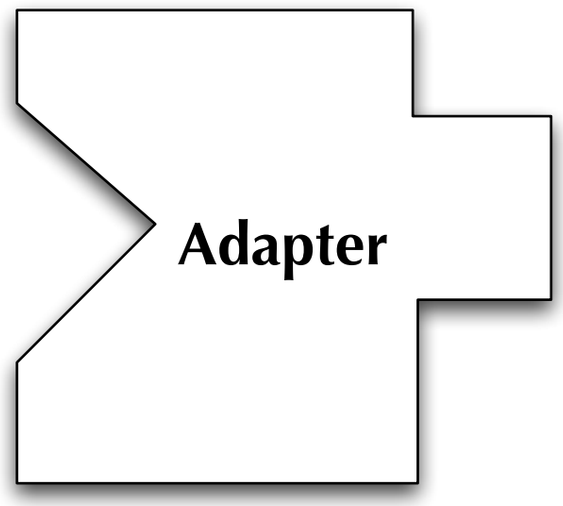
- Pre-Condition: You are maintaining an existing system that makes use of a third-party class library from vendor A
- Stimulus: Vendor A goes belly up and corporate policy does not allow you to make use of an unsupported class library.
- Response: Vendor B provides a similar class library but its interface 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)



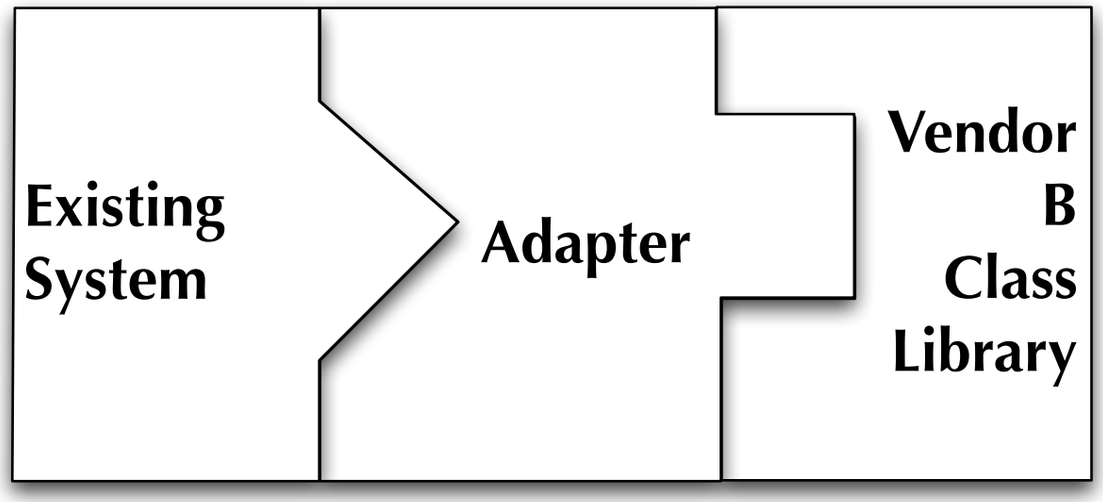
Interface Mismatch
Need Adapter

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.

Simple Example: A turkey hiding among ducks! (I)

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

Simple Example: A turkey hiding among ducks! (II)

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

```
1  public interface Duck {
2      public void quack();
3      public void fly();
4  }
5
6  public class MallardDuck implements Duck {
7
8      public void quack() {
9          System.out.println("Quack");
10     }
11
12     public void fly() {
13         System.out.println("I'm flying");
14     }
15 }
16
```

Simple Example: A turkey hiding among ducks! (III)

- An interloper wants to invade the simulator

```
1  public interface Turkey {
2      public void gobble();
3      public void fly();
4  }
5
6  public class WildTurkey implements Turkey {
7
8      public void gobble() {
9          System.out.println("Gobble Gobble");
10     }
11
12     public void fly() {
13         System.out.println("I'm flying a short distance");
14     }
15
16 }
17
```

Simple Example: A turkey hiding among ducks! (IV)

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

```
1  public class TurkeyAdapter implements Duck {
2
3      private Turkey turkey;
4
5      public TurkeyAdapter(Turkey turkey) {
6          this.turkey = turkey;
7      }
8
9      public void quack() {
10         turkey.gobble();
11     }
12
13     public void fly() {
14         for (int i = 0; i < 5; i++) {
15             turkey.fly();
16         }
17     }
18
19 }
20
```

Demonstration

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

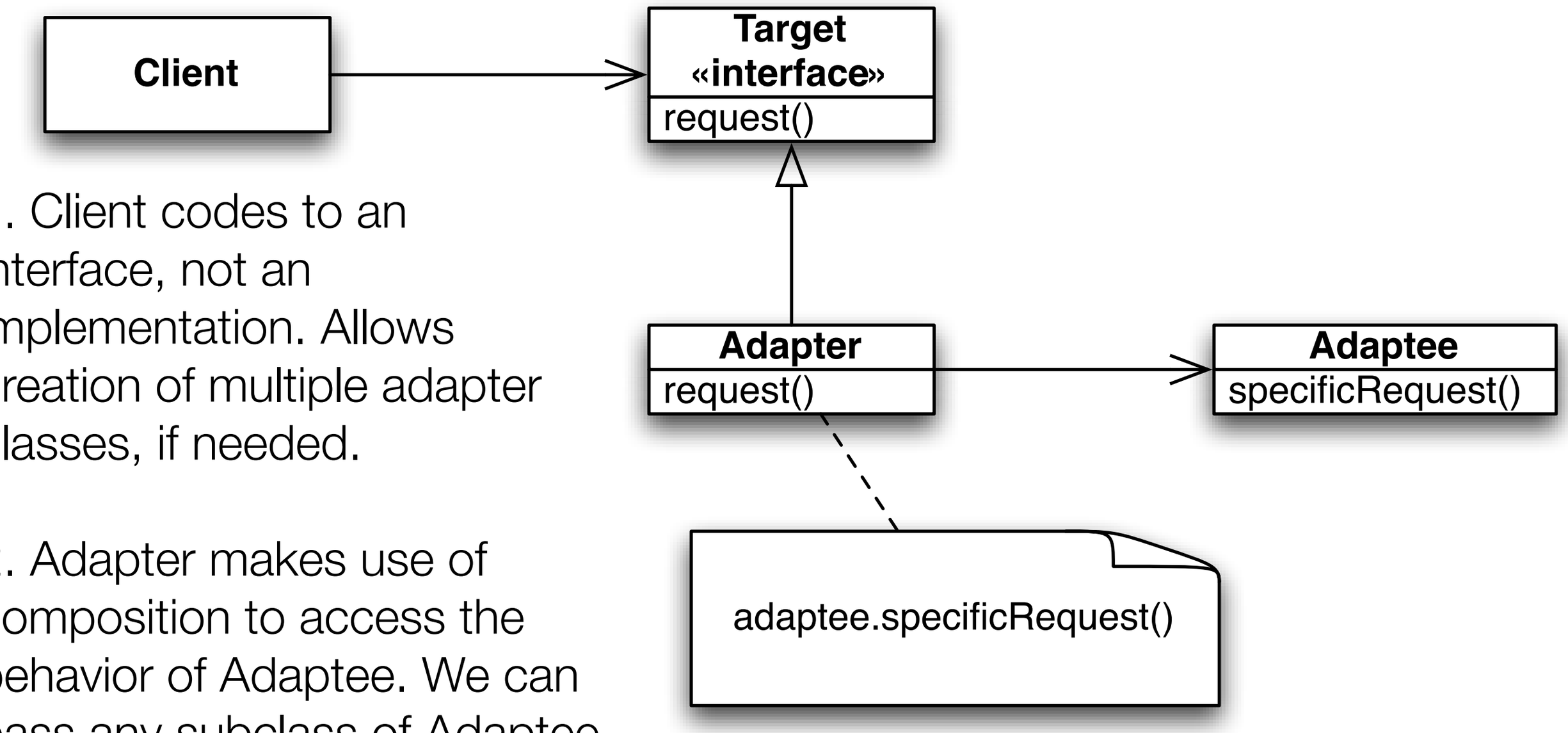
4. Adapter is full-fledged class, could contain additional vars and methods to get its job done

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

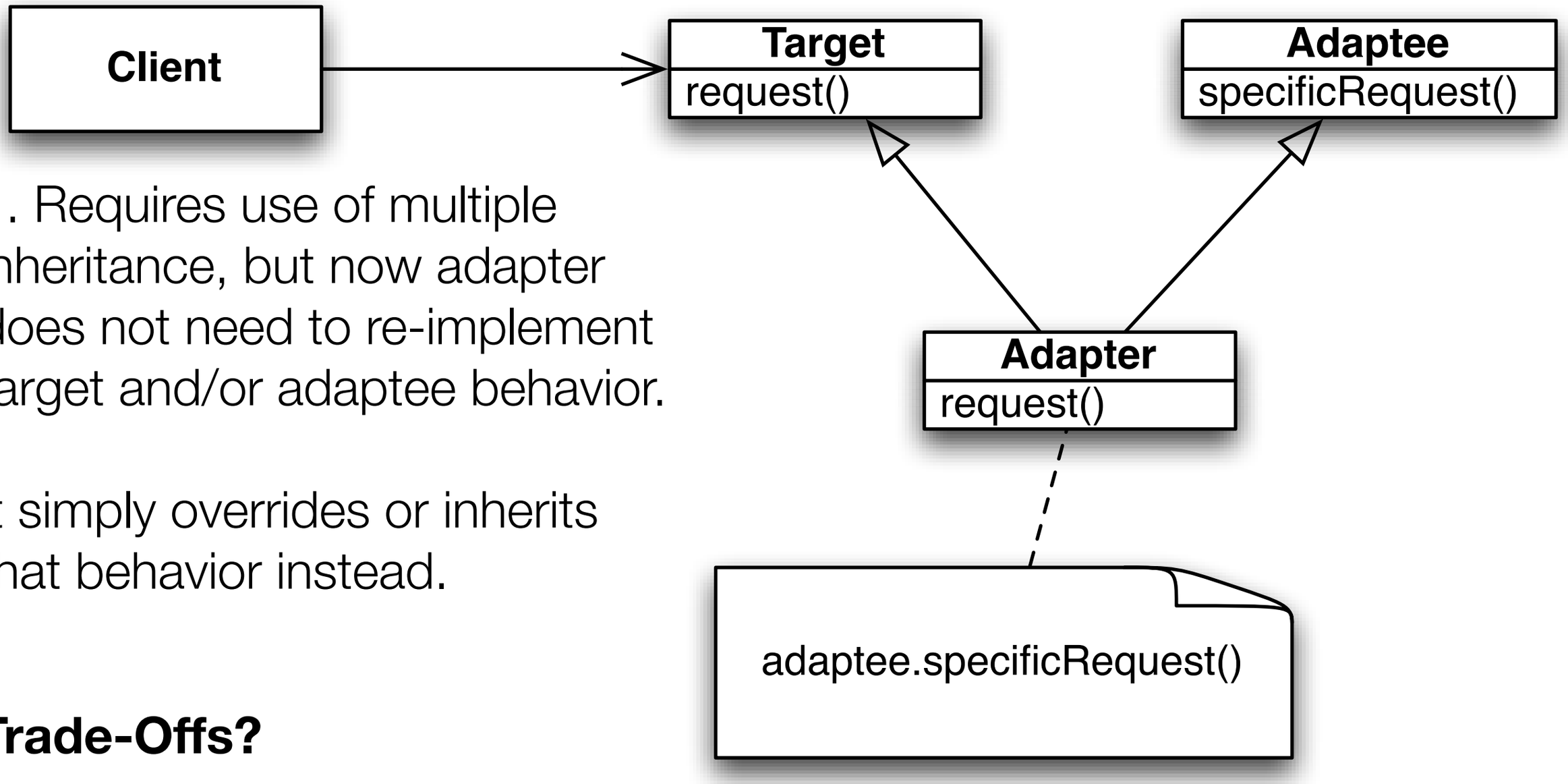


1. Client codes to an interface, not an implementation. Allows creation of multiple adapter classes, if needed.

2. Adapter makes use of composition to access the behavior of Adaptee. We can pass any subclass of Adaptee to the Adapter, if needed.

Adaptee 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?

Demonstration

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 Iteration 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

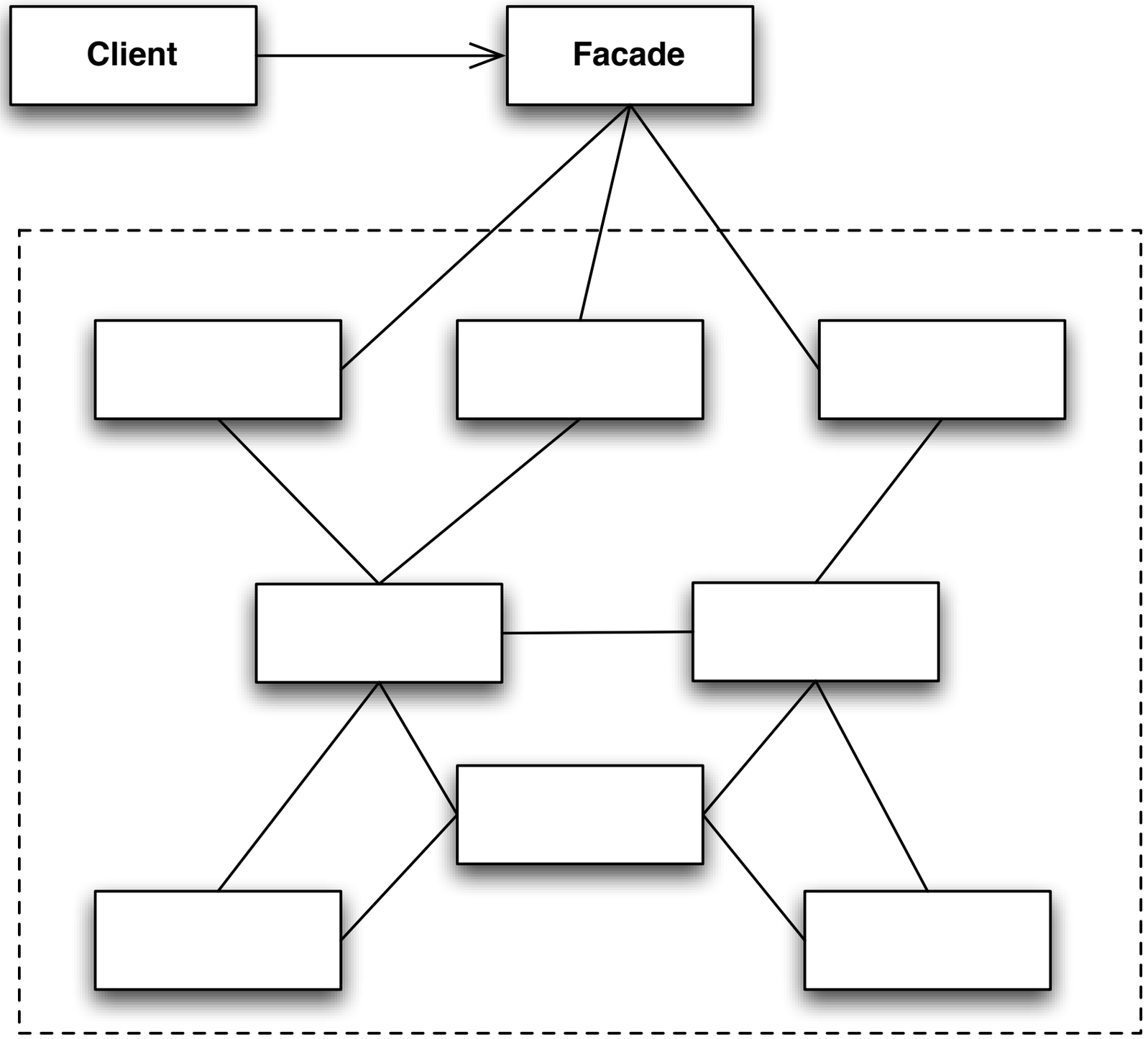
Yet Another Adapter: Facade Pattern

- There is another way in which an adapter can be used between a client and an adaptee: to simplify the interface of the adaptee(s)
- Imagine a library of classes with a complex interface and/or complex interrelationships
  - Book's Example: Home Theater System
    - Amplifier, DvdPlayer, Projector, CdPlayer, Tuner, Screen, PopcornPopper (!), and TheatreLights
      - each with its own interface and interclass dependencies
  - Imagine steps for "watch movie"
    - turn on popper, make popcorn, dim lights, screen down, projector on, set projector to DVD, amplifier on, set amplifier to DVD, DVD on, etc.
  - Now imagine resetting everything after the movie is done, or configuring the system to play a CD, or play a video game, etc.

Facade Pattern: Definition

- The Facade Pattern provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
    - We place high level methods like “watch movie”, “reset system”, “play cd” in a facade object and encode all of the steps for each high level service in the facade.
    - Client code is simplified and the client’s dependencies are greatly reduced
        - A facade not only simplifies an interface, it decouples a client from a subsystem of components
- Relationship to Adapter Pattern?
    - Both facades and adapters may wrap multiple classes, but a facade’s intent is to simplify, while an adapter’s is to convert between interfaces

Facade Pattern: Structure



Demonstration

New Design Principle

- The facade pattern demonstrates a new design principle
- Principle of Least Knowledge: “Talk only to your immediate friends”
    - reminds you to create loosely coupled systems of cohesive objects
    - also known as “The Law of Demeter”

- We want to reduce an object’s class dependencies to the bare minimum
- How many classes is this code coupled to?

```
public float getTemp() {
    return station.getThermometer().getTemperature();
};
```

Principle of Least Knowledge: Heuristics

- In order to implement the principle of least knowledge, follow these guidelines
  - For any object
    - Within any method of that object
      - you may invoke methods that belong to
        - the object itself
        - objects passed in as a parameter to the method
        - any object the method creates or instantiates
        - any object that is stored as an instance variable of the host object
- The code on the previous slide violates these guidelines because we invoke the method getTemperature() on a “friend of a friend”
  - Change code to “return station.getTemperature()” to follow guidelines
  - Requires adding “wrapper” method to station class

Example of all the “legal” method invocations

```
 1  public class Car {
 2
 3      private Engine engine;
 4
 5      public Car() {
 6      }
 7
 8      public void start(Key key) {
 9
10          Door doors = new Doors();
11
12          boolean authorized = key.turns();
13
14          if (authorized) {
15              engine.start();
16              updateDashboardDisplay();
17              doors.lock();
18          }
19      }
20
21      public void updateDashboardDisplay() {
22      }
23
24  }
25
```

object passed as parameter

component method

local method

object created by method

Wrapping Up

- Adapter allows you to convert 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
- Facade is a variant of the adapter pattern in which the purpose is to (greatly) simplify the adaptee’s interface
- Facade demonstrates the use of a new design principle, the Principle of Least Knowledge, also known as the Law of Demeter
  - often phrased “talk only to your friends”
  - focus is on reducing coupling between classes

Coming Up Next

- Lecture 24: Template Method
    - Read Chapter 8 of the Design Patterns Textbook
- Lecture 25: Iterator and Composite
    - Read Chapter 9 of the Design Patterns Textbook