Lecture 08: The Adapter and Facade Patterns

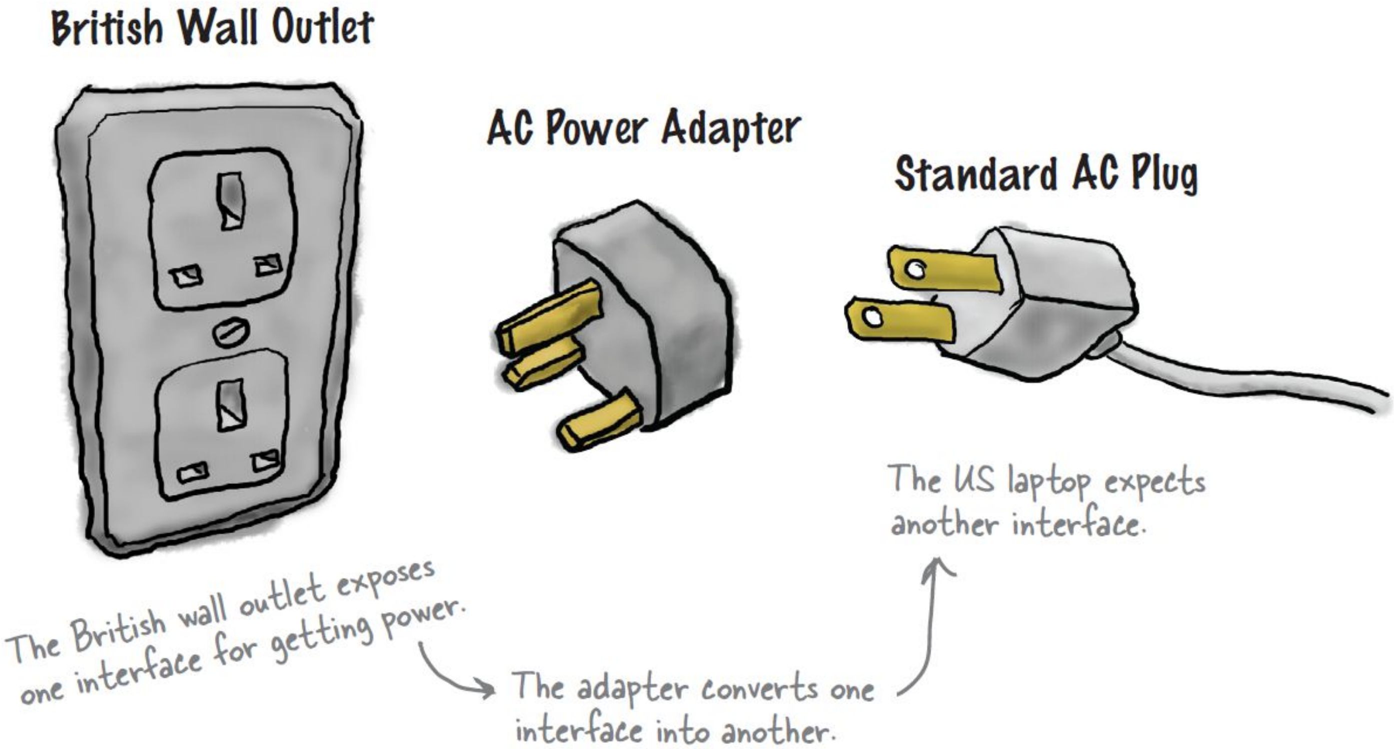
SE313, Software Design and Architecture
Damla Oguz

Chapter 7: The Adapter and Facade Patterns

- Remember the Decorator Pattern? We wrapped objects to give them new responsibilities.
- Now we're going to wrap some objects with a different purpose:
    - to make their interfaces look like something they're not.
- Why would we do that?
    - So we can adapt a design expecting one interface to a class that implements a different interface.
- That's not all; while we're at it, we're going to look at another pattern that wraps objects to simplify their interface.

Adapters all around us

- Have you ever needed to use US-made up laptop in Great Britain?



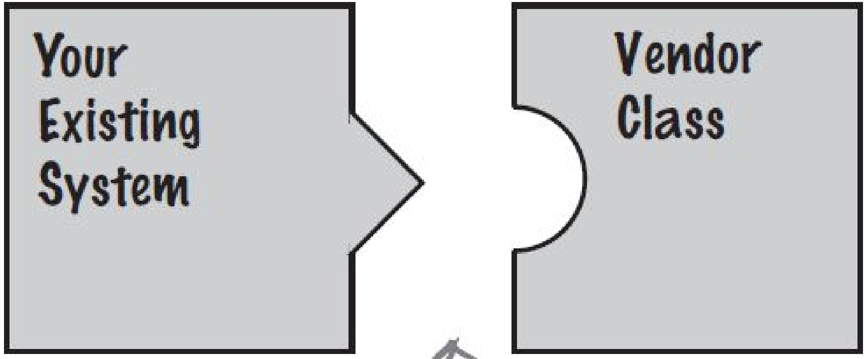
The US laptop expects another interface.

The British wall outlet exposes one interface for getting power.

The adapter converts one interface into another.

Object-oriented adapters

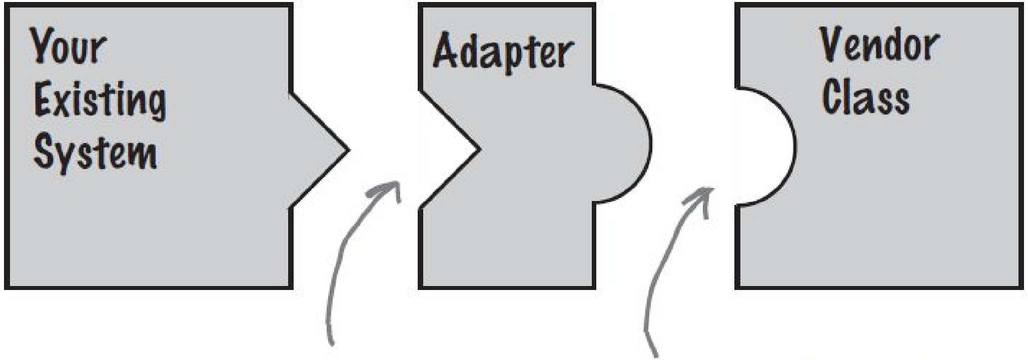
- Say you’ve got an existing software system that you need to work a new vendor class library into, but the new vendor designed their interfaces differently than the last vendor:



Their interface doesn’t match the one you’ve written your code against. This isn’t going to work!

Object-oriented adapters (cont.)

- Okay, you don’t want to solve the problem by changing your existing code.
- You can’t change the vendor’s code.
- So what do you do?
    - You can write a class that adapts the new vendor interface into the one you’re expecting.

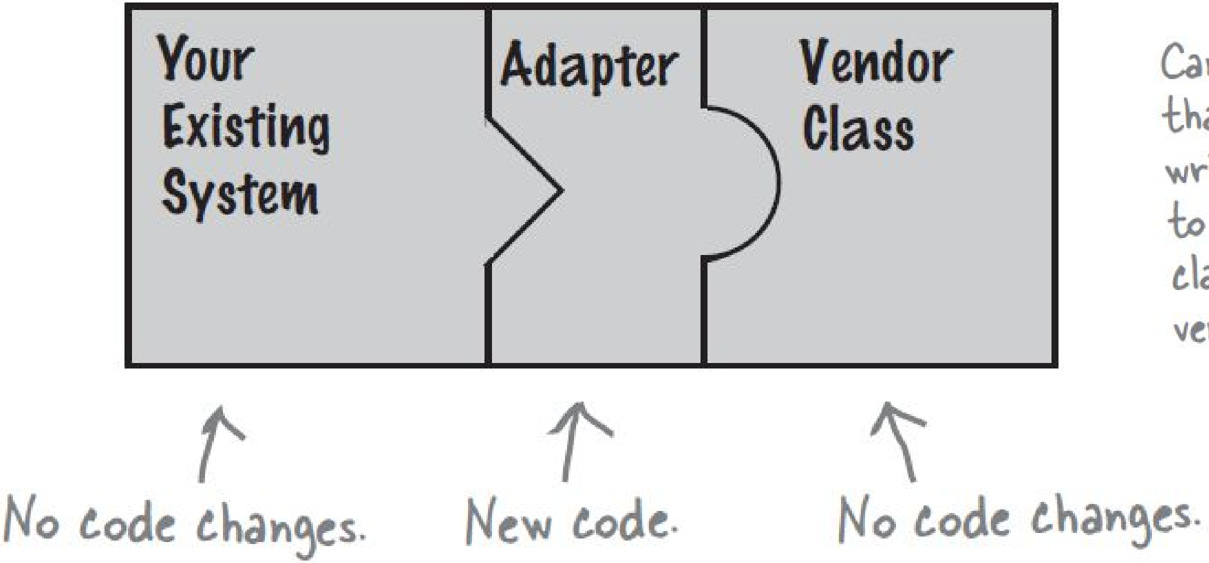


The adapter implements the interface your classes expect.

And talks to the vendor interface to service your requests.

Object-oriented adapters (cont.)

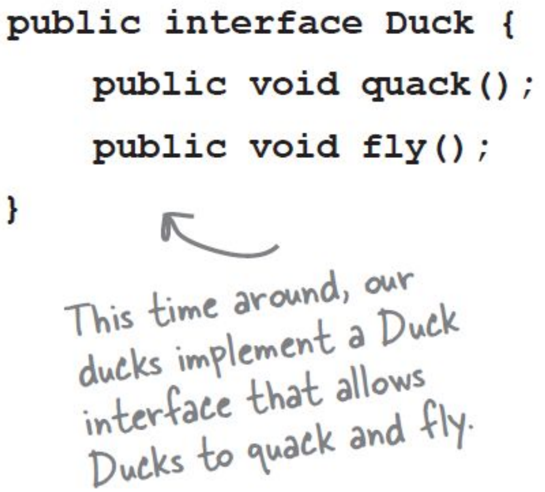
- The adapter acts as the middleman by receiving requests from the client and converting them into requests that make sense on the vendor classes.



Can you think of a solution that doesn't require YOU to write ANY additional code to integrate the new vendor classes? How about making the vendor supply the adapter class?

Adapter in action

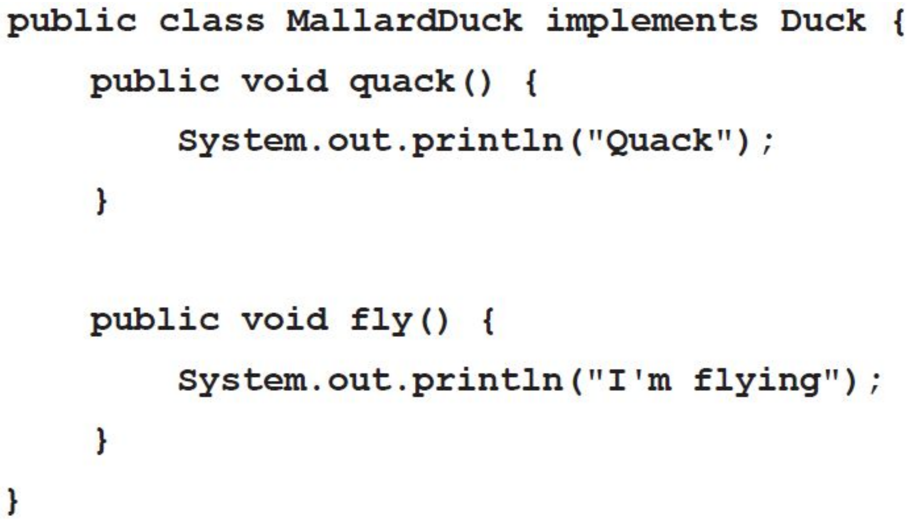
- Remember our ducks?



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

This time around, our ducks implement a Duck interface that allows Ducks to quack and fly.

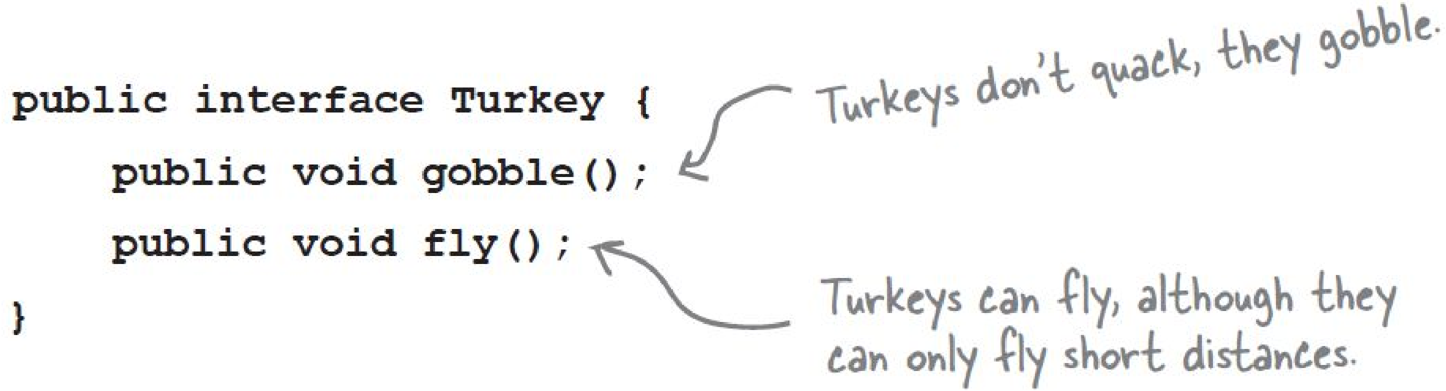
- Here’s a subclass of Duck, the MallardDuck.



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

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

Adapter in action (cont.)

- Now it’s time to meet the newest fowl on the block:



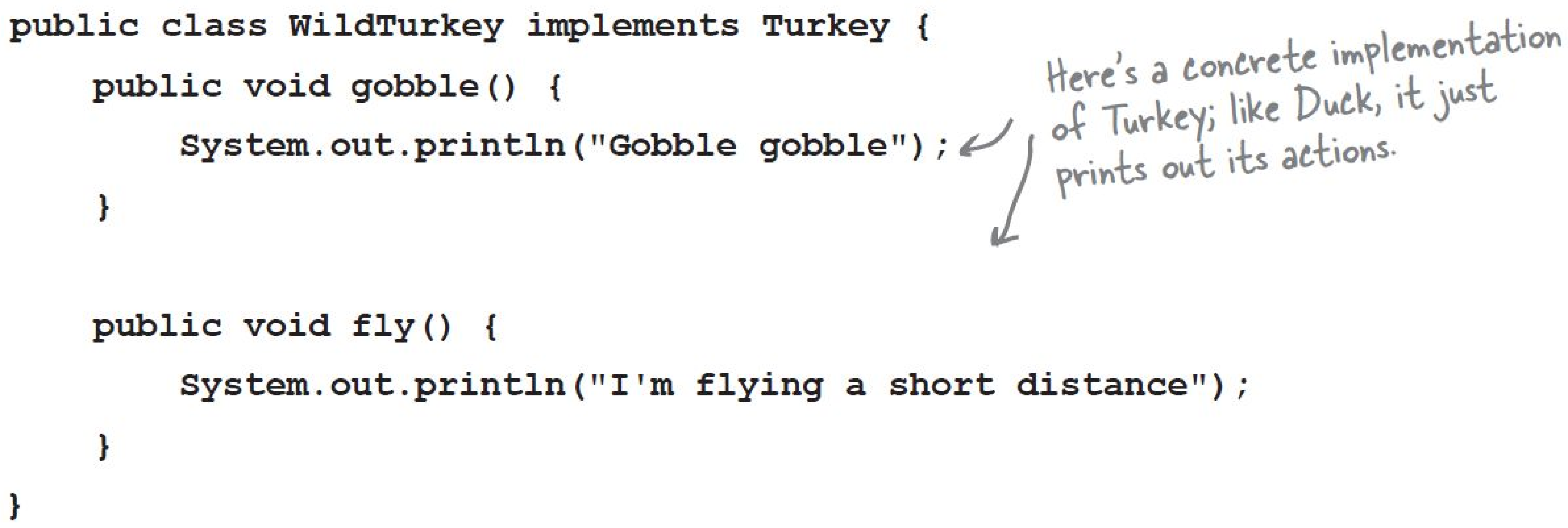
```
public interface Turkey {
    public void gobble();
    public void fly();
}
```

Turkeys don’t quack, they gobble.

Turkeys can fly, although they can only fly short distances.

- If it walks like a duck and quacks like a duck, then it ~~must~~ might be a ~~duck~~ turkey wrapped with a duck adapter…

Adapter in action (cont.)



```
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");
    }
}
```

Here's a concrete implementation of Turkey; like Duck, it just prints out its actions.

Adapter in action (cont.)

- Now, let’s say you’re short on Duck objects and you’d like to use some Turkey objects in their place.
- Obviously we can’t use the turkeys outright because they have a different interface.
- So, let’s write an Adapter…

```java
public class TurkeyAdapter implements Duck {
    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();
        }
    }
}
```

First, you need to implement the interface of the type you're adapting to. This is the interface your client expects to see.

Next, we need to get a reference to the object that we are adapting; here we do that through the constructor.

Now we need to implement all the methods in the interface; the quack() translation between classes is easy: just call the gobble() method.

Even though both interfaces have a fly() method, Turkeys fly in short spurts – they can't do long-distance flying like ducks. To map between a Duck's fly() method and a Turkey's, we need to call the Turkey's fly() method five times to make up for it.

Test drive the adapter

```
public class DuckTestDrive {
    public static void main(String[] args) {
        MallardDuck duck = new MallardDuck();

        WildTurkey turkey = new WildTurkey();
        Duck turkeyAdapter = new TurkeyAdapter(turkey);

        System.out.println("The Turkey says...");
        turkey.gobble();
        turkey.fly();

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

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

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

Let's create a Duck...

...and a Turkey.

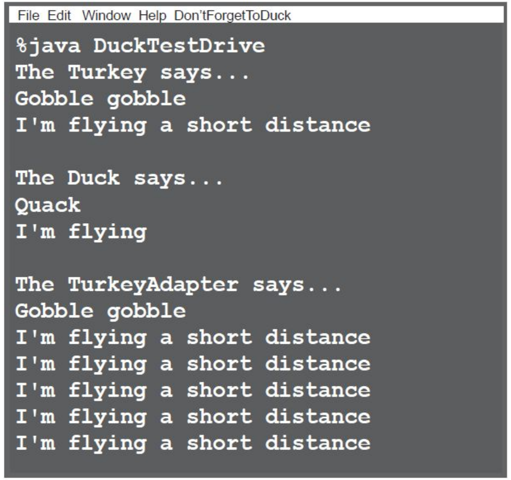
And then wrap the turkey in a TurkeyAdapter, which makes it look like a Duck.

Then, let's test the Turkey: make it gobble, make it fly.

Now let's test the duck by calling the testDuck() method, which expects a Duck object.

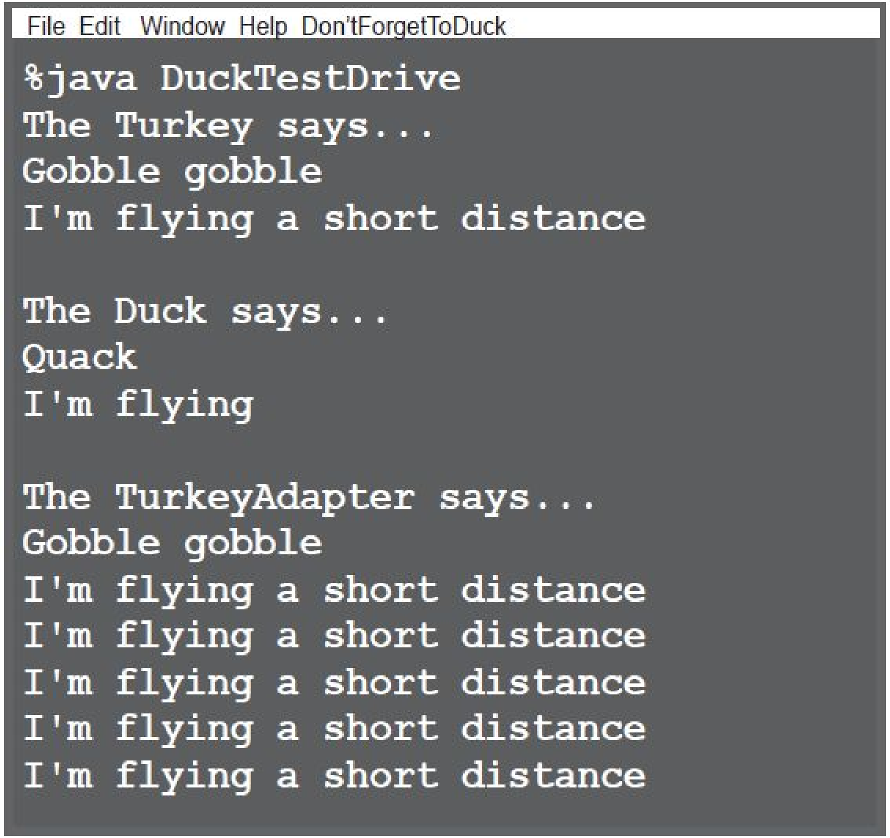
Now the big test: we try to pass off the turkey as a duck...

Here's our testDuck() method; it gets a duck and calls its quack() and fly() methods.



```
%java DuckTestDrive
The Turkey says...
Gobble gobble
I'm flying a short distance

The Duck says...
Quack
I'm flying

The TurkeyAdapter says...
Gobble gobble
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
```

Test drive the adapter (cont.)



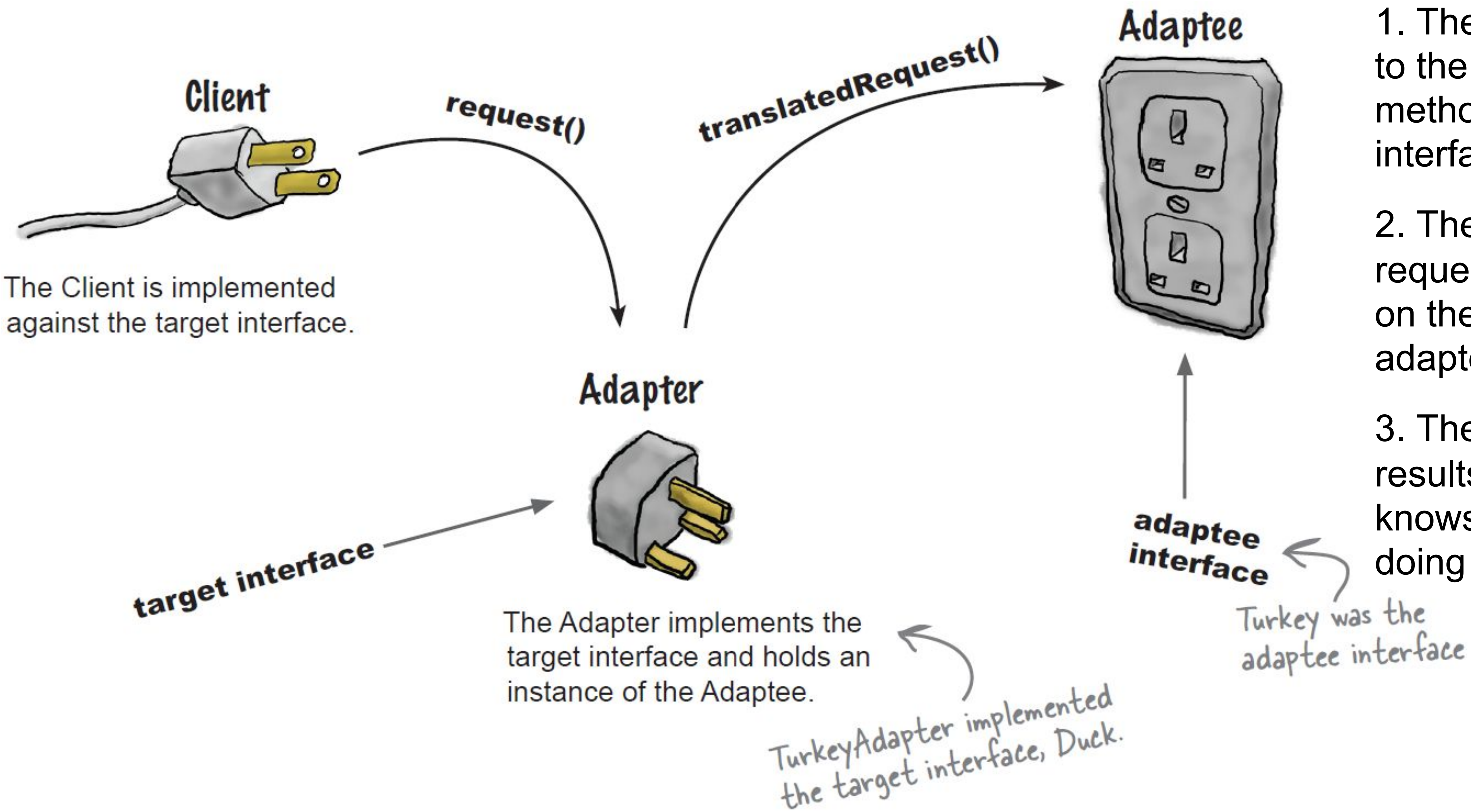
```
%java DuckTestDrive
The Turkey says...
Gobble gobble
I'm flying a short distance

The Duck says...
Quack
I'm flying

The TurkeyAdapter says...
Gobble gobble
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
```

The Turkey gobbles and flies a short distance.

The Duck quacks and flies just like you'd expect.

And the adapter gobbles when quack() is called and flies a few times when fly() is called. The testDuck() method never knows it has a turkey disguised as a duck!

The Adapter Pattern explained


Client
request()
translatedRequest()
Adaptee
The Client is implemented against the target interface.
Adapter
target interface
adaptee interface
The Adapter implements the target interface and holds an instance of the Adaptee.
Turkey was the adaptee interface
TurkeyAdapter implemented the target interface, Duck.

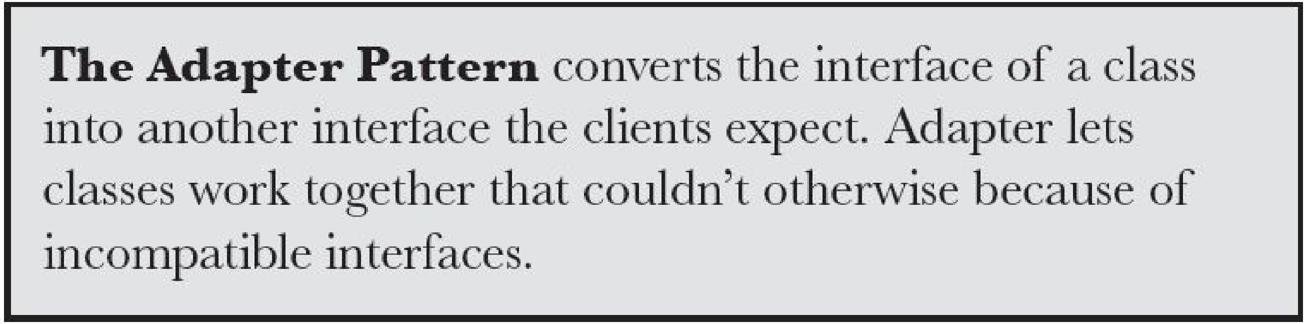
Here is how the client uses the adapter

1. The client makes a request to the adapter by calling a method on it using the target interface.

2. The adapter translates the request into one or more calls on the adaptee using the adaptee interface.

3. The client receives the results of the call and never knows there is an adapter doing the translation.

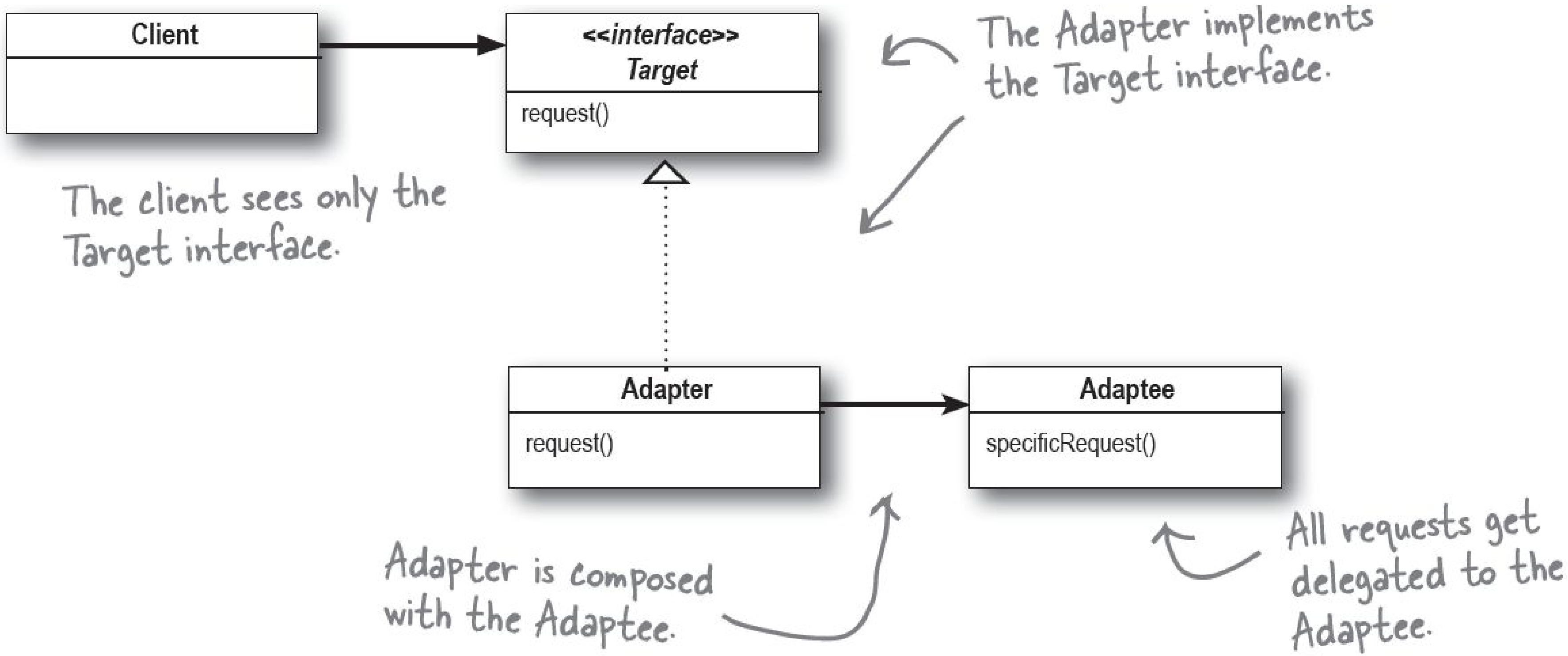
Adapter Pattern defined



The Adapter Pattern converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

- This pattern allows us to use a client with an incompatible interface by creating an Adapter that does the conversion.
- This acts to decouple the client from the implemented interface.
- If we expect the interface to change over time, the adapter encapsulates that change so that the client doesn't have to be modified each time it needs to operate against a different interface.

Adapter Pattern defined: class diagram


Client
<<interface>>
Target
request()
Adapter
request()
Adaptee
specificRequest()
The Adapter implements the Target interface.
The client sees only the Target interface.
Adapter is composed with the Adaptee.
All requests get delegated to the Adaptee.

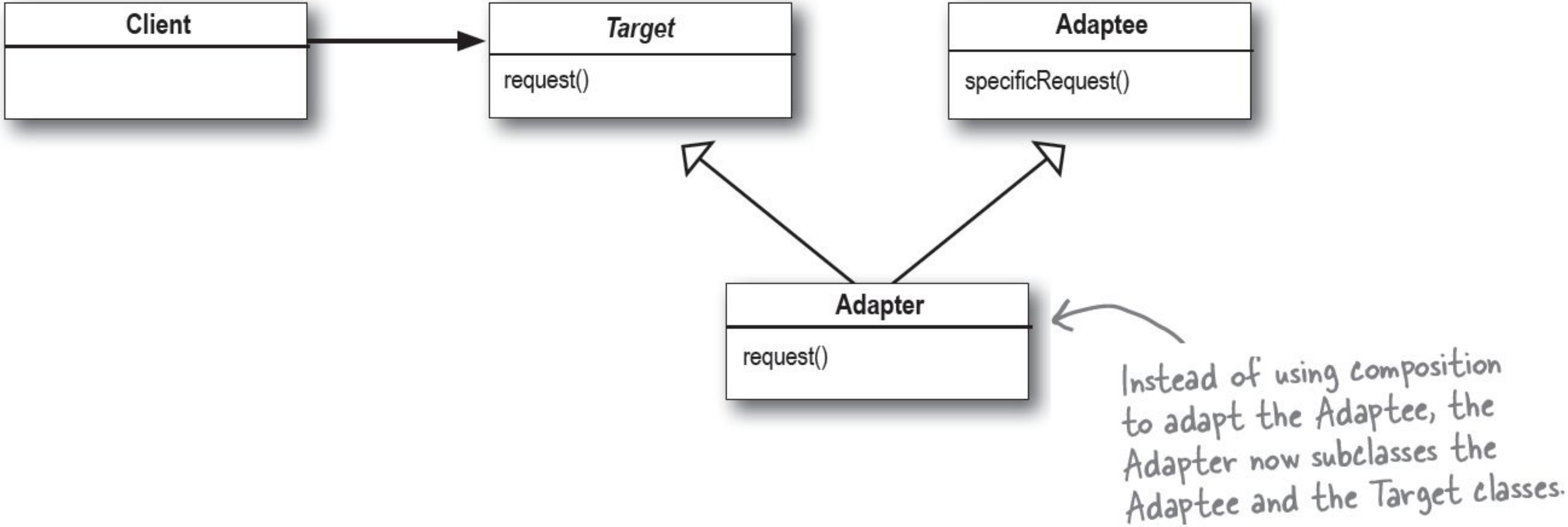
Adapter Pattern defined (cont.)

- The pattern binds the client to an interface, not an implementation:
    - We could use several adapters, each converting a different backend set of classes.
    - Or, we could add new implementations after the fact, as long as they adhere to the Target interface.

Object and class adapters

- There are actually two kinds of adapters: object adapters and class adapters.
- What we have investigated is an object adapter.
- You need multiple inheritance to implement class adapter, which isn’t possible in Java.

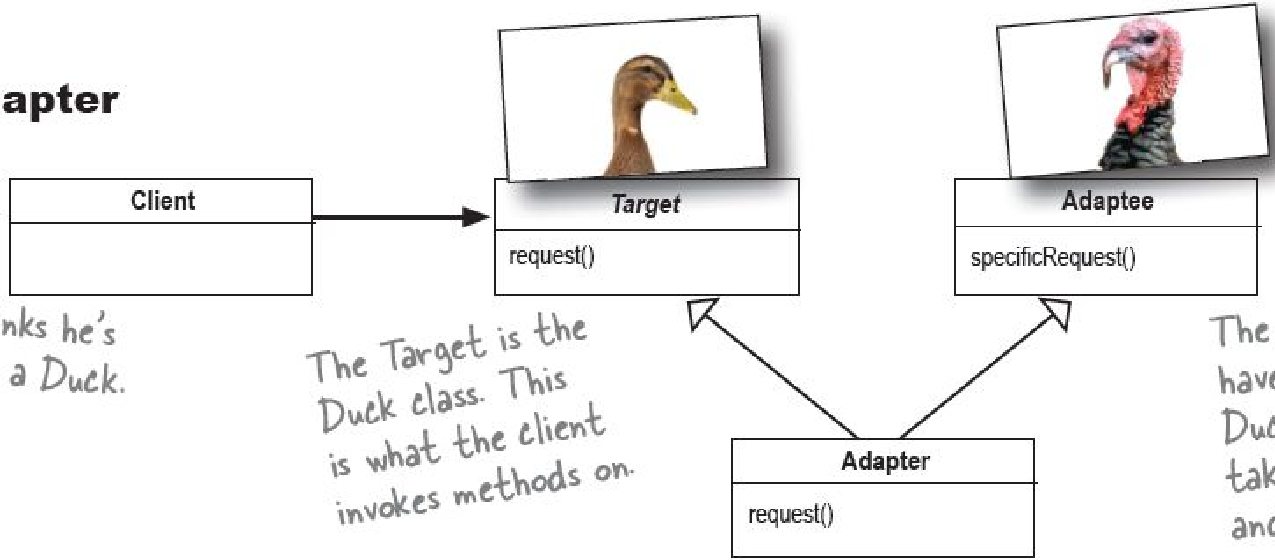
Class adapters: class diagram for multiple inheritance



Instead of using composition to adapt the Adaptee, the Adapter now subclasses the Adaptee and the Target classes.

- Look familiar? That’s right—the only difference is that
  - with class adapter we subclass the Target and the Adaptee,
  - while with object adapter we use composition to pass requests to an Adaptee.

Class Adapter



Duck class

Turkey class

Client thinks he's talking to a Duck.

The Target is the Duck class. This is what the client invokes methods on.

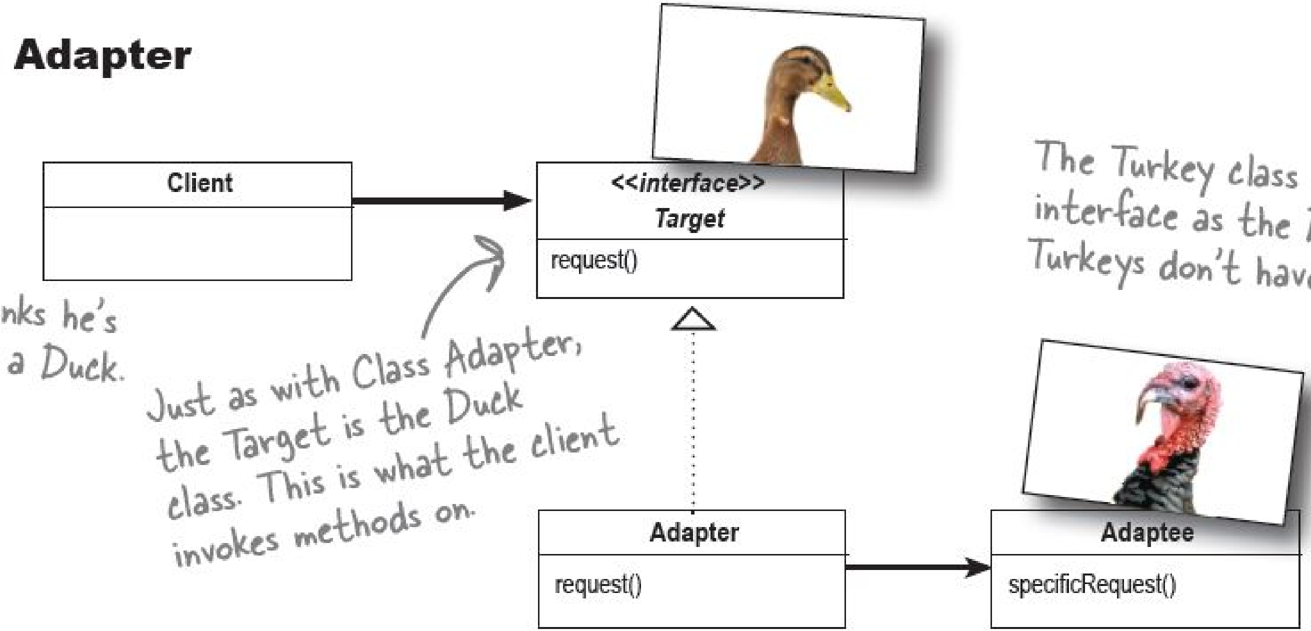
The Turkey class does not have the same methods as Duck, but the Adapter can take Duck method calls and turn around and invoke methods on the Turkey.

The Adapter lets the Turkey respond to requests on a Duck, by extending BOTH classes (Duck and Turkey).

Note: the class adapter uses multiple inheritance, so you can't do it in Java...

Duck interface

Object Adapter



Client

<<interface>>
Target

request()

Adapter

request()

Adaptee

specificRequest()

Client thinks he's talking to a Duck.

Just as with Class Adapter, the Target is the Duck class. This is what the client invokes methods on.

The Turkey class doesn't have the same interface as the Duck. In other words, Turkeys don't have quack() methods, etc.

Turkey object

The Adapter implements the Duck interface, but when it gets a method call it turns around and delegates the calls to a Turkey.

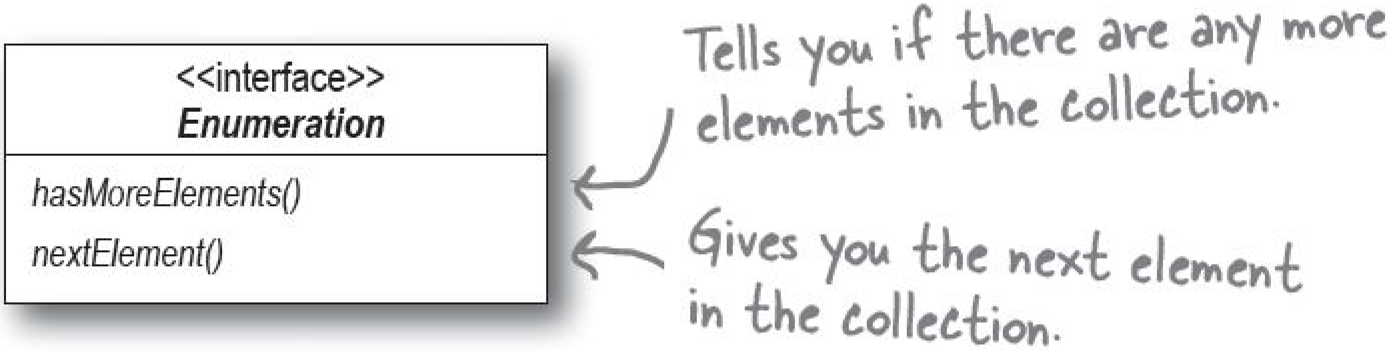
Thanks to the Adapter, the Turkey (Adaptee) will get calls that the client makes on the Duck interface.

Real-world adapters

- Let’s take a look at the use of a simple Adapter in the real world (something more serious than Ducks at least)...
- We will talk about:
  - Old-world Enumerators
  - New-world Iterators
  - And today

Real-world adapters

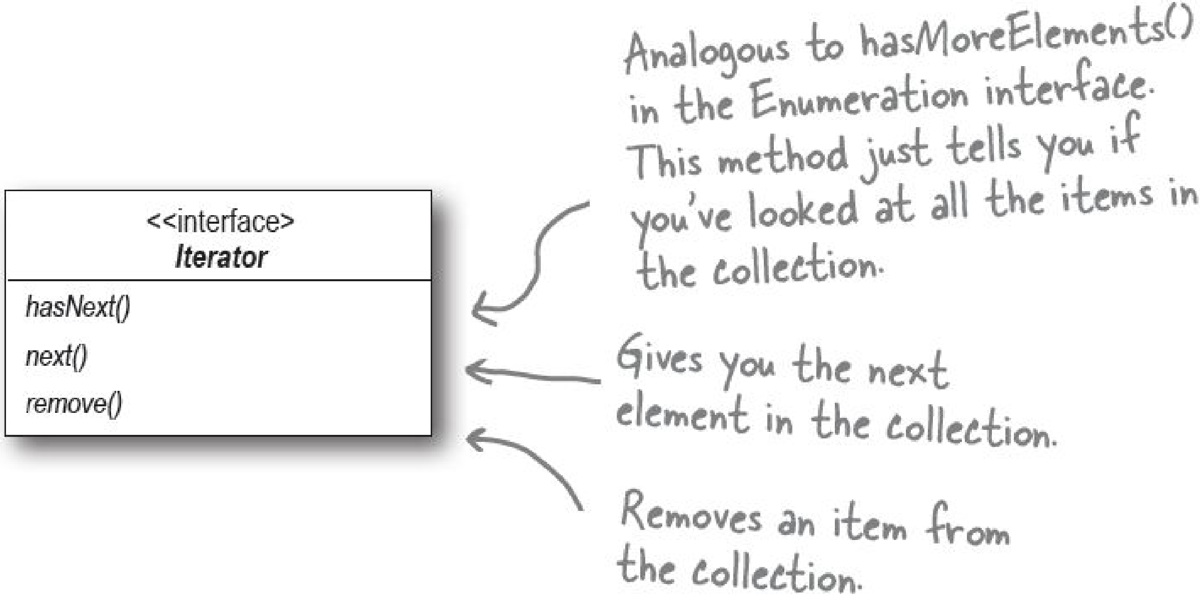
- Old-world Enumerators
  - If you’ve been around Java for a while you probably remember that the early collection types (Vector, Stack, Hashtable, and a few others) implement a method, elements(), which returns an Enumeration.
  - The Enumeration interface allows you to step through the elements of a collection without knowing the specifics of how they are managed in the collection.



<<interface>>
Enumeration
hasMoreElements()
nextElement()
Tells you if there are any more elements in the collection.
Gives you the next element in the collection.

Real-world adapters (cont.)

- New-world Iterators
  - The newer Collection classes use an Iterator interface that, like Enumeration, allows you to iterate through a set of items in a collection, but also adds the ability to remove items.



Analogous to hasMoreElements() in the Enumeration interface. This method just tells you if you've looked at all the items in the collection.

Gives you the next element in the collection.

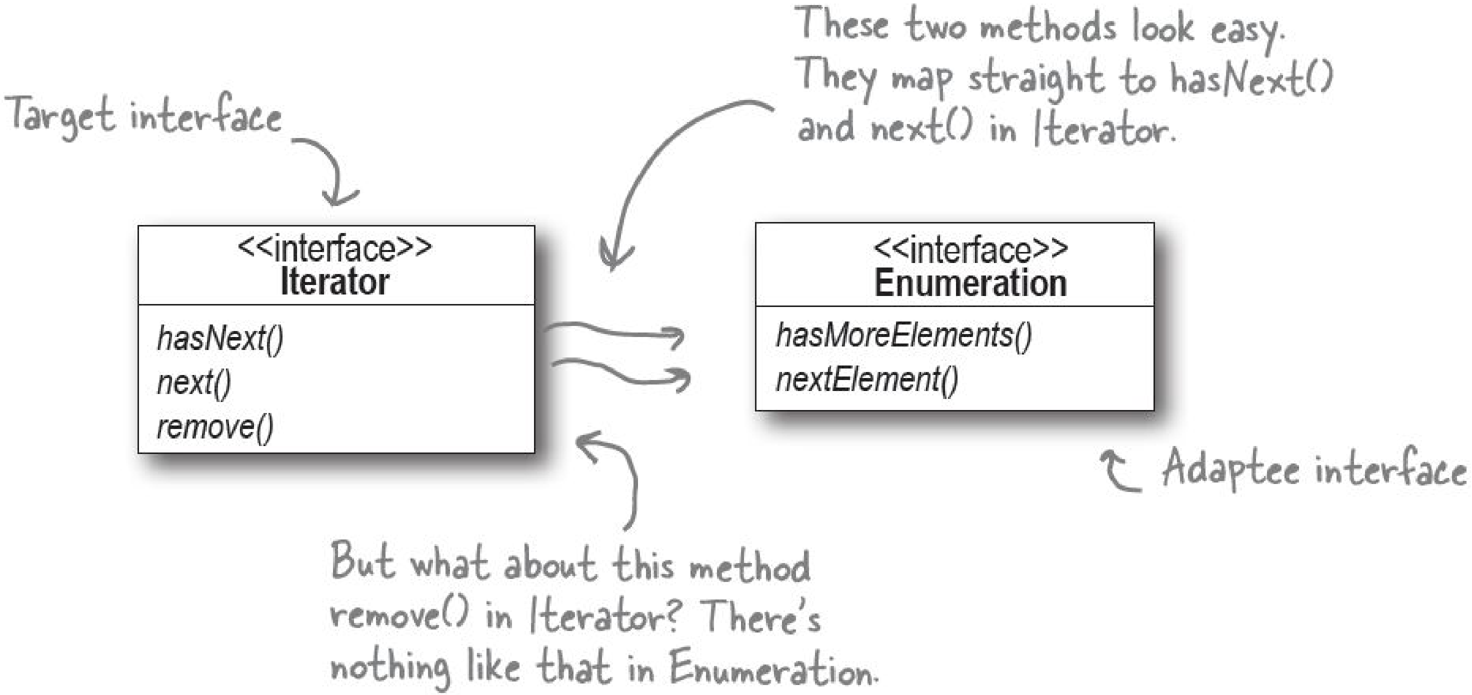
Removes an item from the collection.

Real-world adapters (cont.)

- And today
  - We are often faced with legacy code that exposes the Enumeration interface, yet we'd like for our new code to use only Iterators.
  - It looks like we need to build an adapter.

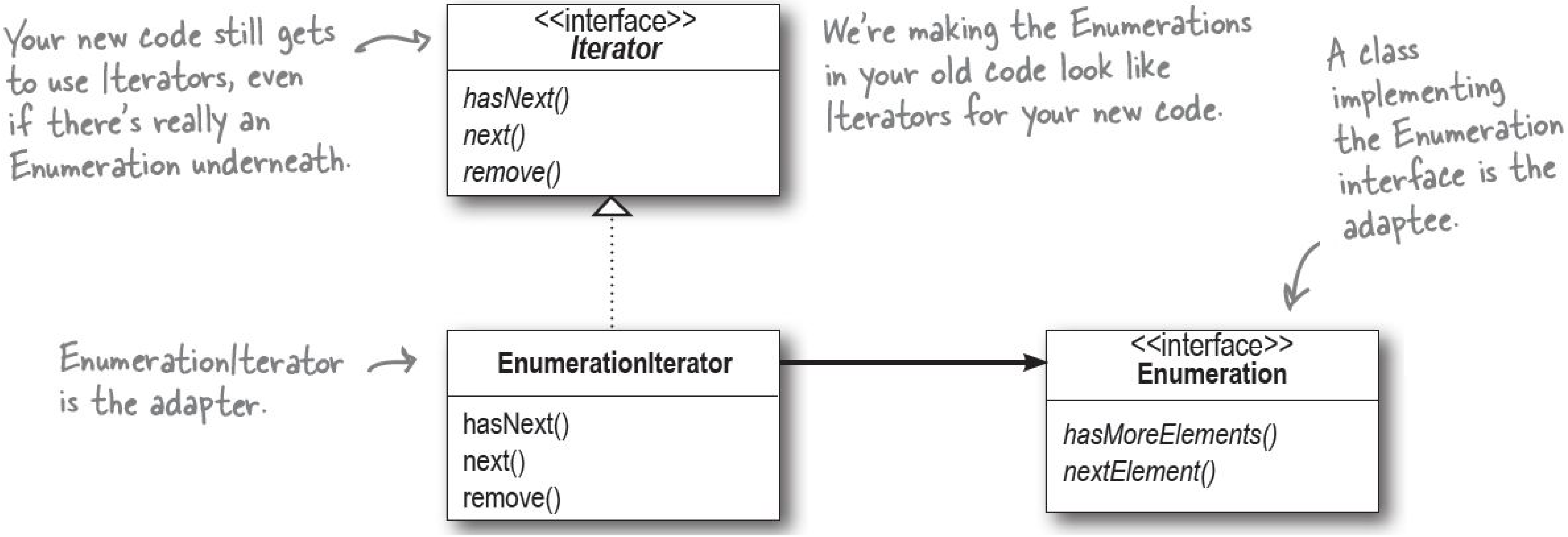
Adapting an Enumeration to an Iterator

- First we'll look at the two interfaces to figure out how the methods map from one to the other.


Target interface
<<interface>>
Iterator
hasNext()
next()
remove()
These two methods look easy. They map straight to hasNext() and next() in Iterator.
<<interface>>
Enumeration
hasMoreElements()
nextElement()
Adaptee interface
But what about this method remove() in Iterator? There's nothing like that in Enumeration.

Designing the Adapter

- We need an adapter that implements the Target interface and that is composed with an adaptee.



Your new code still gets to use Iterators, even if there's really an Enumeration underneath.

<<interface>>
Iterator

hasNext()
next()
remove()

We're making the Enumerations in your old code look like Iterators for your new code.

A class implementing the Enumeration interface is the adaptee.

EnumerationIterator is the adapter.

EnumerationIterator

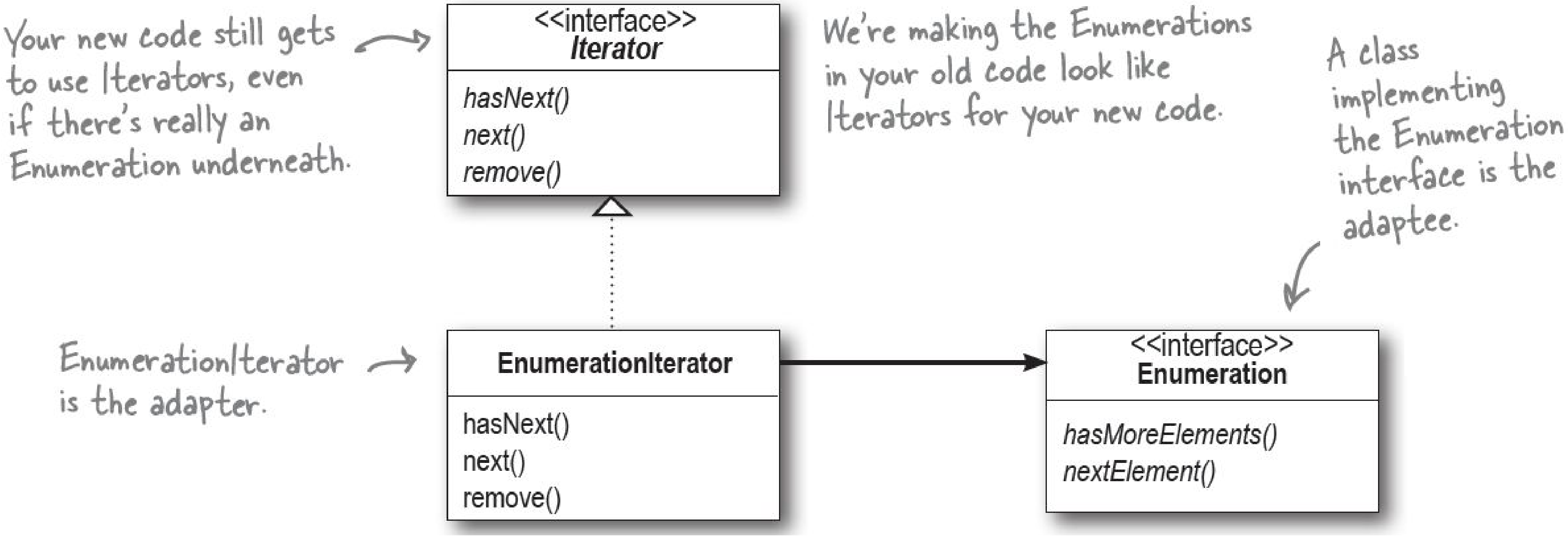
hasNext()
next()
remove()

<<interface>>
Enumeration

hasMoreElements()
nextElement()

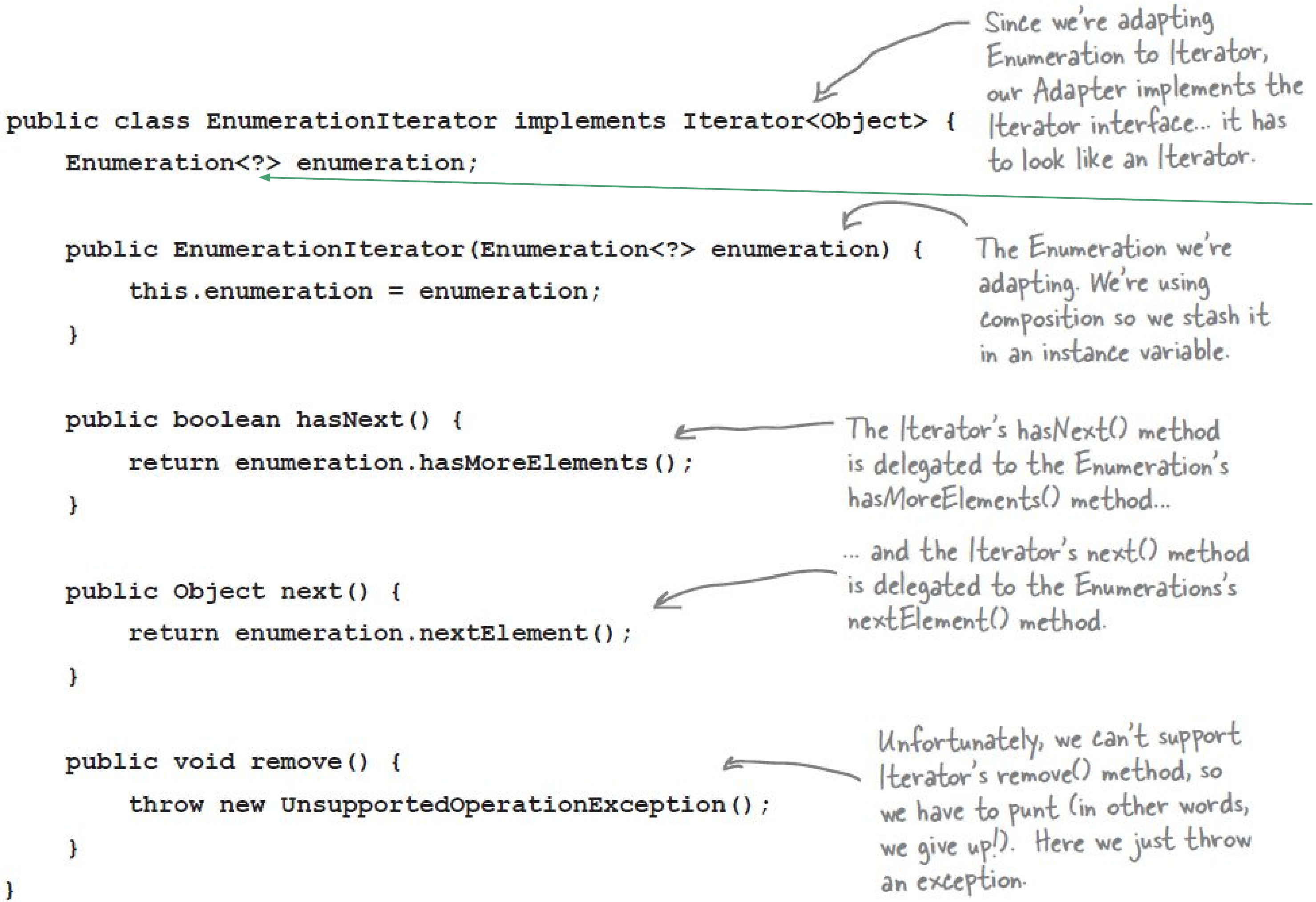
Designing the Adapter (cont.)

- The hasNext() and next() methods are going to be straightforward to map from target to adaptee. But what do you do about remove()?


Your new code still gets to use Iterators, even if there's really an Enumeration underneath.
<<interface>>
Iterator
hasNext()
next()
remove()
We're making the Enumerations in your old code look like Iterators for your new code.
A class implementing the Enumeration interface is the adaptee.
EnumerationIterator is the adapter.
EnumerationIterator
hasNext()
next()
remove()
<<interface>>
Enumeration
hasMoreElements()
nextElement()

Dealing with the remove() method

- We know Enumeration just doesn’t support remove.
- It’s a “read only” interface. There’s no way to implement a fully functioning remove() method on the adapter.
- The best we can do is throw a runtime exception.
- Luckily, the designers of the Iterator interface foresaw this need and defined the remove() method so that it supports an UnsupportedOperationException.
- This is a case where the adapter isn’t perfect; clients will have to watch out for potential exceptions, but as long as the client is careful and the adapter is well documented this is a perfectly reasonable solution.



```
public class EnumerationIterator implements Iterator<Object> {
    Enumeration<?> enumeration;

    public EnumerationIterator(Enumeration<?> enumeration) {
        this.enumeration = enumeration;
    }

    public boolean hasNext() {
        return enumeration.hasMoreElements();
    }

    public Object next() {
        return enumeration.nextElement();
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

Since we're adapting Enumeration to Iterator, our Adapter implements the Iterator interface... it has to look like an Iterator.

The Enumeration we're adapting. We're using composition so we stash it in an instance variable.

The Iterator's hasNext() method is delegated to the Enumeration's hasMoreElements() method...

... and the Iterator's next() method is delegated to the Enumerations's nextElement() method.

Unfortunately, we can't support Iterator's remove() method, so we have to punt (in other words, we give up!). Here we just throw an exception.

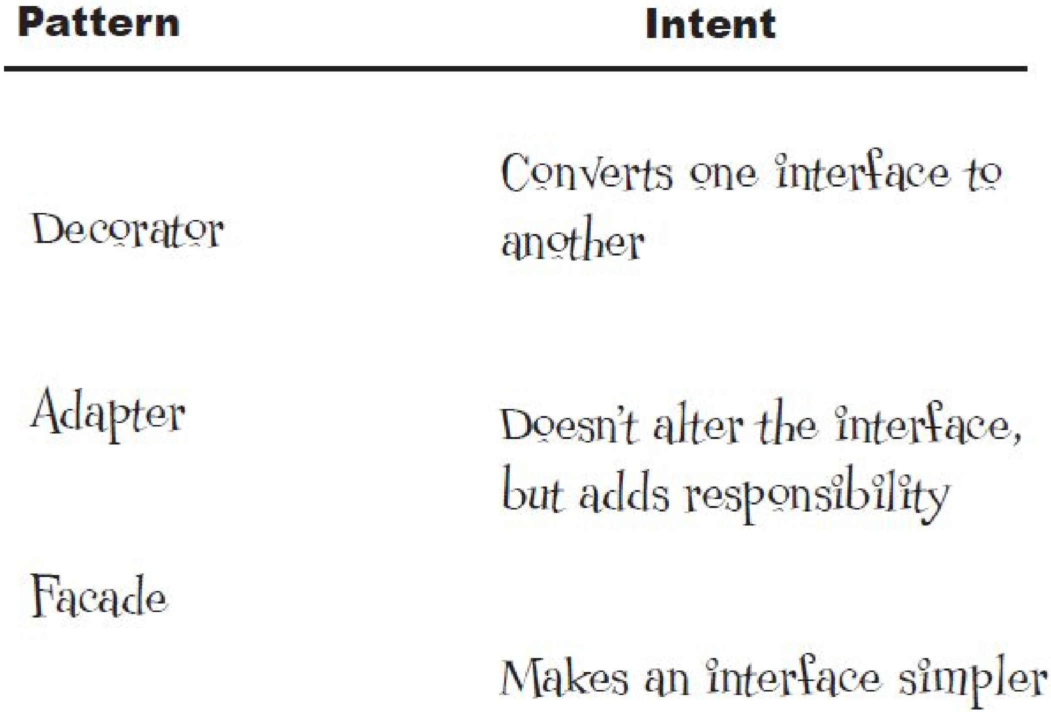
A shorthand for <? extends Object>. Also known as an *unbounded wildcard*. So you can specify any type of object in your generic.

And now for something different...

- You’ve seen how the Adapter Pattern converts the interface of a class into one that a client is expecting.
- You also know we achieve this in Java by wrapping the object that has an incompatible interface with an object that implements the correct one.
- We’re going to look at a pattern now that alters an interface, but for a different reason: to simplify the interface: the Facade Pattern
- This pattern hides all the complexity of one or more classes behind a clean, well-lit facade.

Who does what?

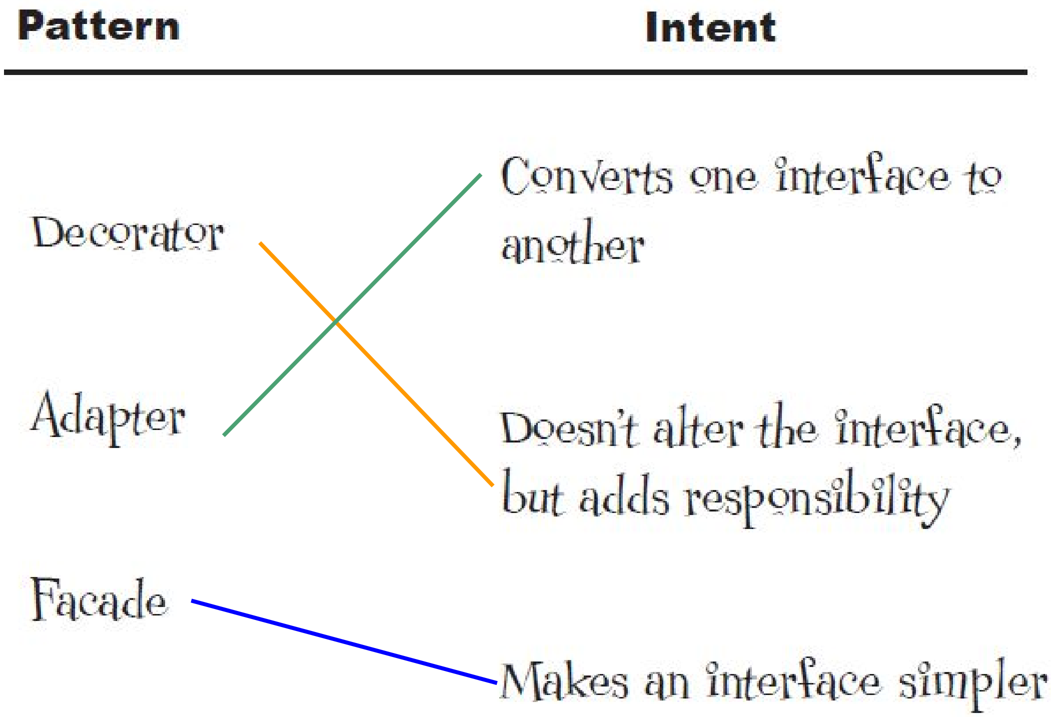
- Match each pattern with its intent:



Pattern	Intent
Decorator	Converts one interface to another
Adapter	Doesn’t alter the interface, but adds responsibility
Facade	Makes an interface simpler

Who does what?

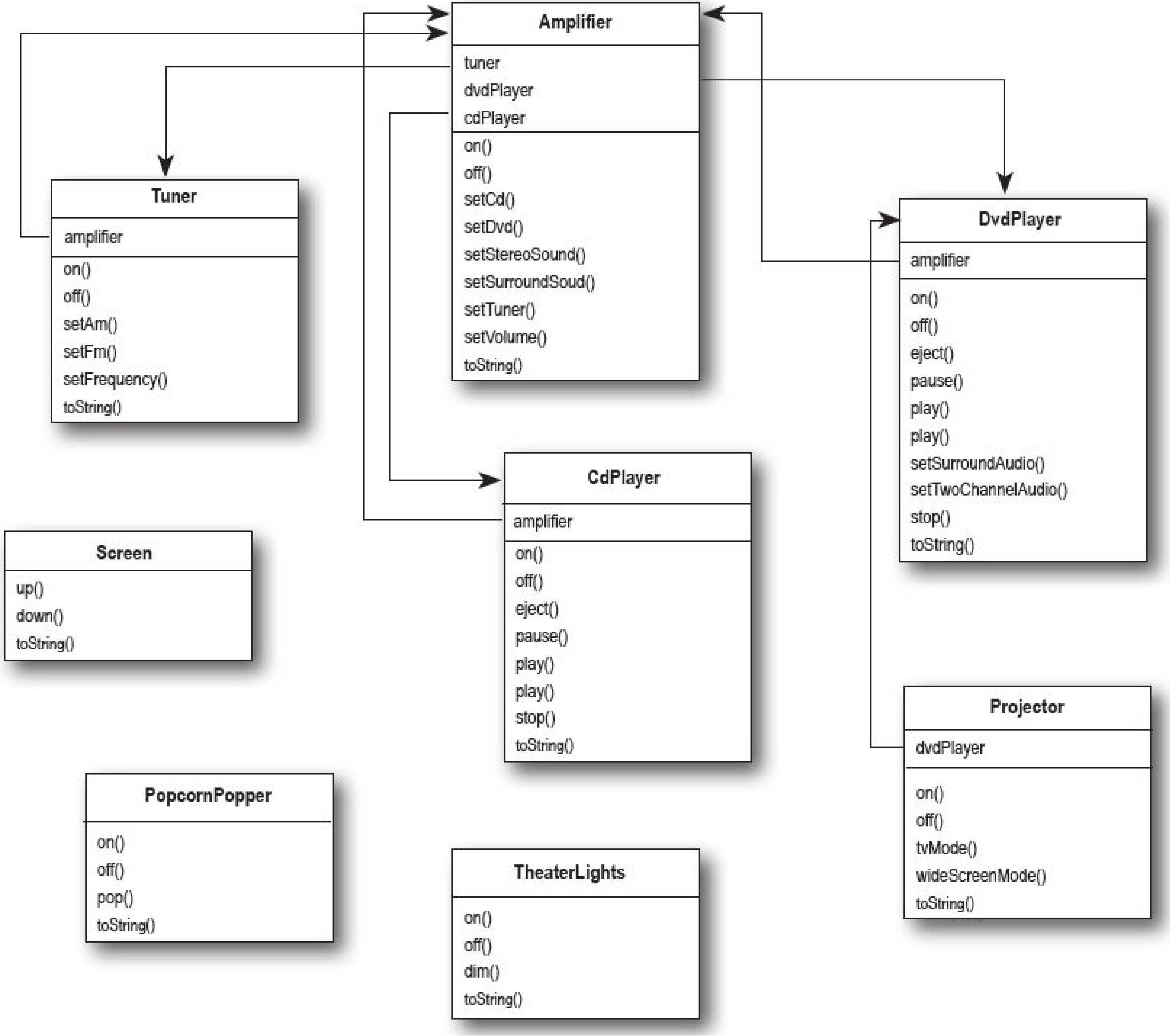
- Match each pattern with its intent:



Pattern	Intent
Decorator	Converts one interface to another
Adapter	Doesn't alter the interface, but adds responsibility
Facade	Makes an interface simpler

Home Sweet Home Theater

- Before we dive into the details of the Facade Pattern, let’s take a look at: building your own home theater.
- You need: a DVD player, a projection video system, an automated screen, surround sound, and even a popcorn popper.
- Check out all the components you’ve put together…



Amplifier

tuner
dvdPlayer
cdPlayer

on()
off()
setCd()
setDvd()
setStereoSound()
setSurroundSoud()
setTuner()
setVolume()
toString()

Tuner

amplifier

on()
off()
setAm()
setFm()
setFrequency()
toString()

DvdPlayer

amplifier

on()
off()
eject()
pause()
play()
play()
setSurroundAudio()
setTwoChannelAudio()
stop()
toString()

CdPlayer

amplifier

on()
off()
eject()
pause()
play()
play()
stop()
toString()

Screen

up()
down()
toString()

Projector

dvdPlayer

on()
off()
tvMode()
wideScreenMode()
toString()

PopcornPopper

on()
off()
pop()
toString()

TheaterLights

on()
off()
dim()
toString()

That's a lot of classes, a lot of interactions, and a big set of interfaces to learn and use.

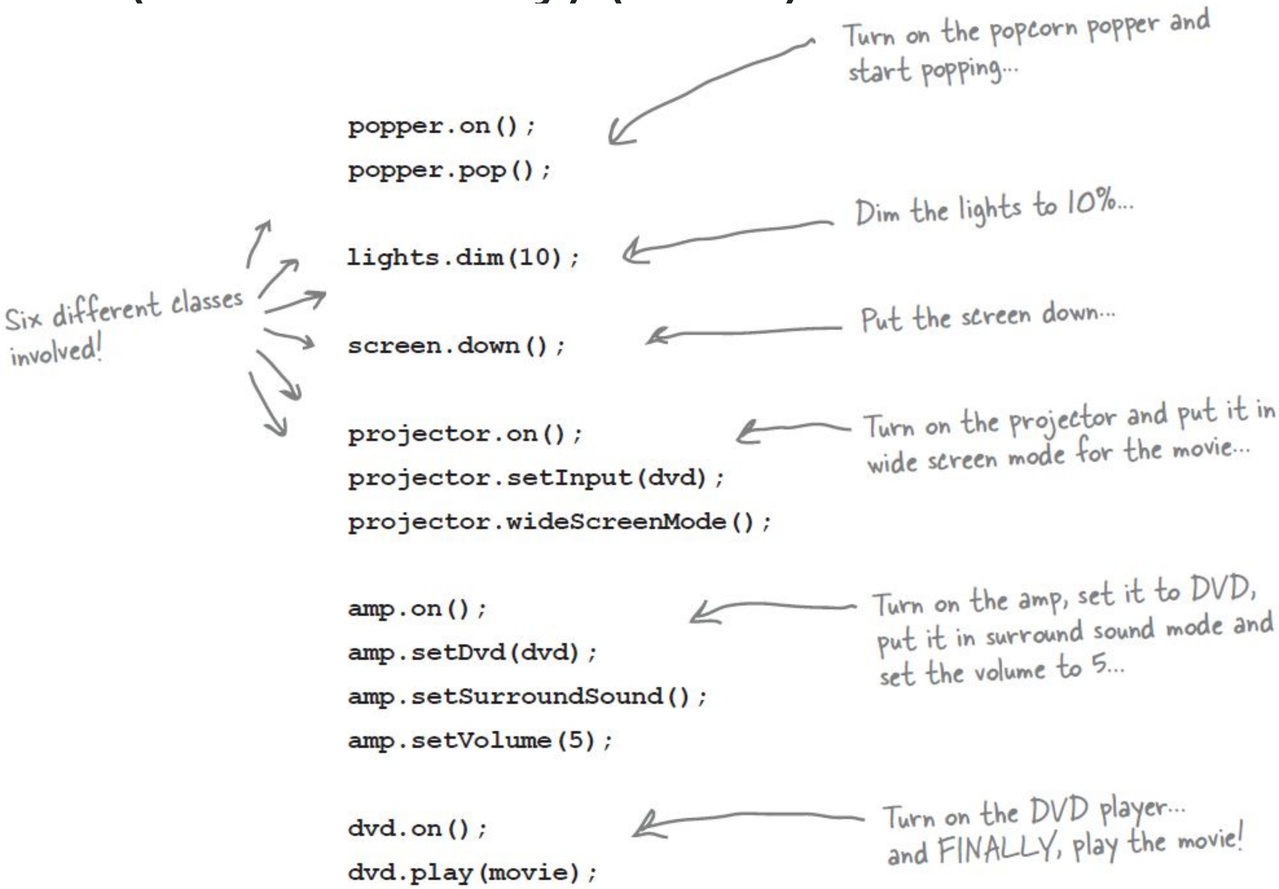
Watching a movie (the hard way)

- Pick out a DVD, relax, and get ready for movie magic. To watch the movie, you need to perform a few tasks:
  - Turn on the popcorn popper
  - Start the popper popping
  - Dim the lights
  - Put the screen down
  - Turn the projector on
  - Put the projector on wide-screen mode
  - Turn the sound amplifier on
  - Set the amplifier to DVD input
  - Set the amplifier to surround sound
  - Set the amplifier volume to medium (5)
  - Turn the DVD player on
  - Start the DVD player playing
  - Set the projector input to DVD

I'm already exhausted and all I've done is turn everything on!

Watching a movie (the hard way) (cont.)

- Let’s check out those tasks



```
popper.on();
popper.pop();

lights.dim(10);

screen.down();

projector.on();
projector.setInput(dvd);
projector.wideScreenMode();

amp.on();
amp.setDvd(dvd);
amp.setSurroundSound();
amp.setVolume(5);

dvd.on();
dvd.play(movie);
```

Six different classes involved!

Turn on the popcorn popper and start popping...

Dim the lights to 10%...

Put the screen down...

Turn on the projector and put it in wide screen mode for the movie...

Turn on the amp, set it to DVD, put it in surround sound mode and set the volume to 5...

Turn on the DVD player... and FINALLY, play the movie!

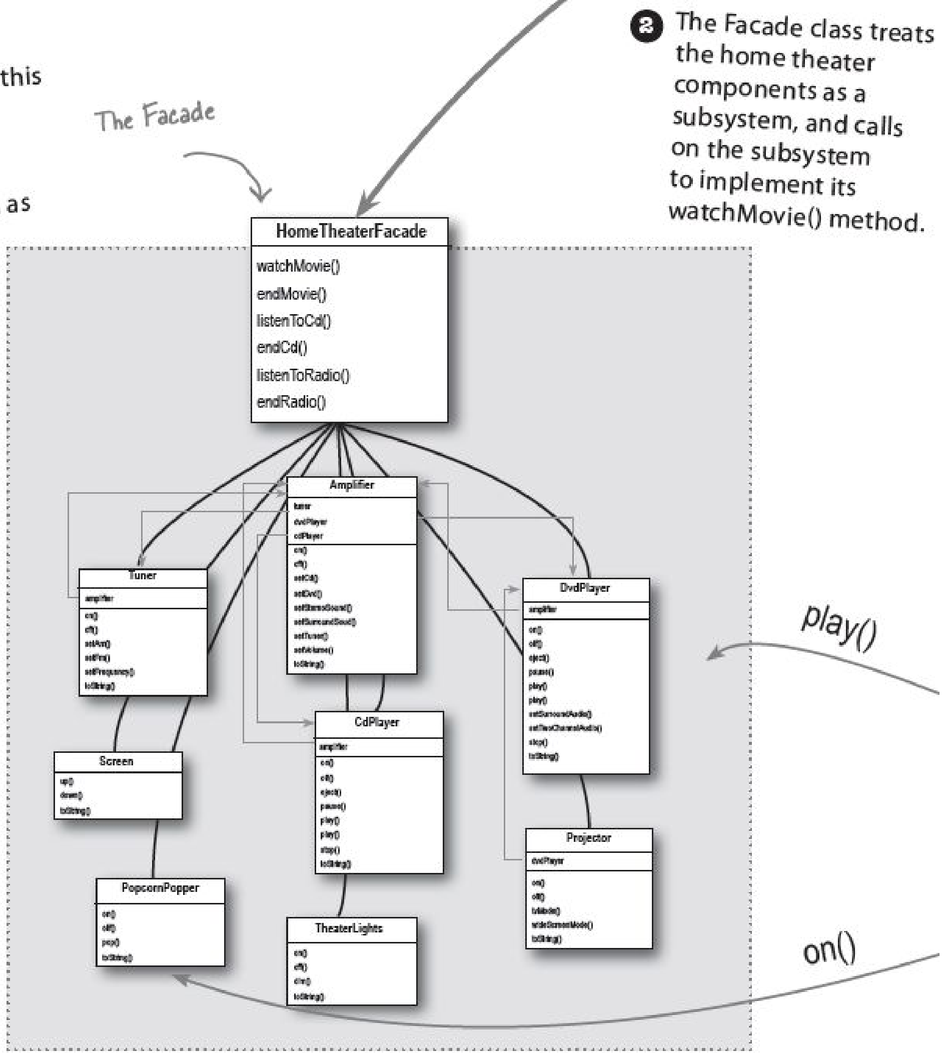
Watching a movie (the hard way) (cont.)

- But there’s more...
  - When the movie is over, how do you turn everything off? Wouldn’t you have to do all of this over again, in reverse?
  - Wouldn’t it be as complex to listen to a CD or the radio?
  - If you decide to upgrade your system, you’re probably going to have to learn a slightly different procedure.
- So what to do? The complexity of using your home theater is becoming apparent!
- Let’s see how the Facade Pattern can get us out of this mess so we can enjoy the movie...

Lights, Camera, Facade!

- A Facade is just what you need: with the Facade Pattern you can take a complex subsystem and make it easier to use by implementing a Facade class that provides one, more reasonable interface.
- If you need the power of the complex subsystem, it’s still there for you to use, but if all you need is a straightforward interface, the Facade is there for you.
- Let’s take a look at how the Facade operates…

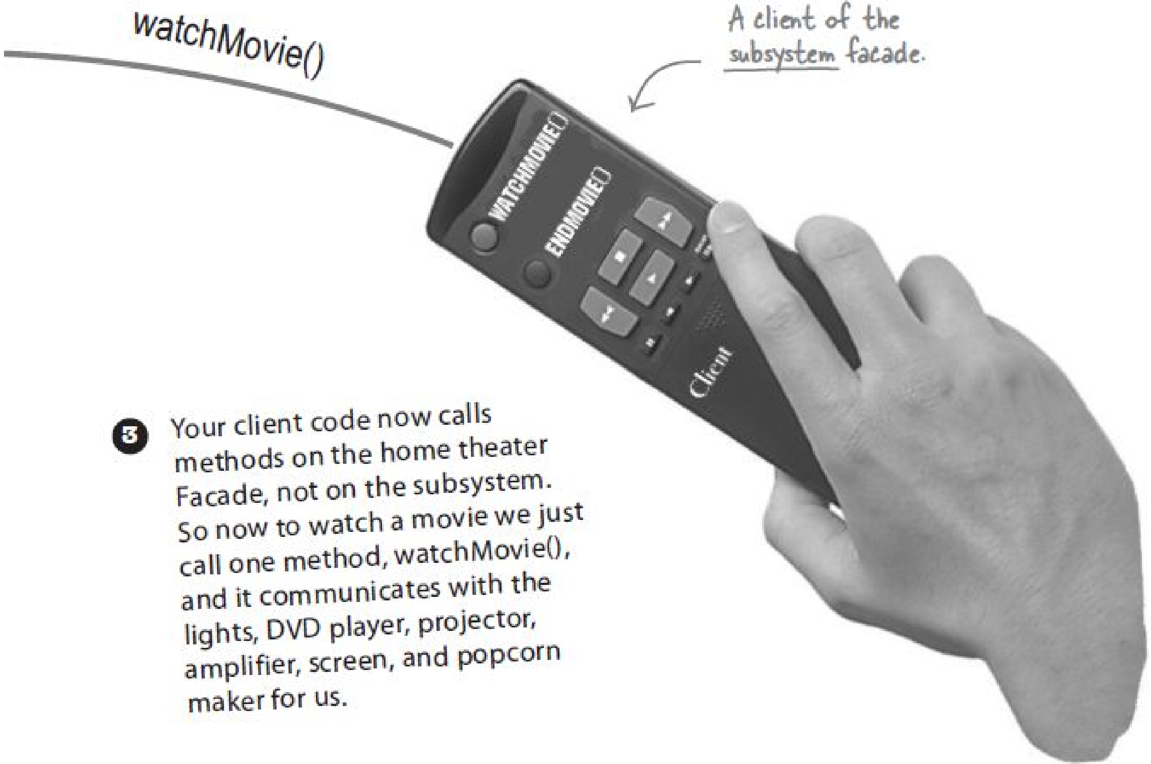
❶ Okay, time to create a Facade for the home theater system. To do this we create a new class HomeTheaterFacade, which exposes a few simple methods such as watchMovie().



The Facade

❷ The Facade class treats the home theater components as a subsystem, and calls on the subsystem to implement its watchMovie() method.

The subsystem the Facade is simplifying.



A client of the subsystem facade.

3 Your client code now calls methods on the home theater Facade, not on the subsystem. So now to watch a movie we just call one method, watchMovie(), and it communicates with the lights, DVD player, projector, amplifier, screen, and popcorn maker for us.

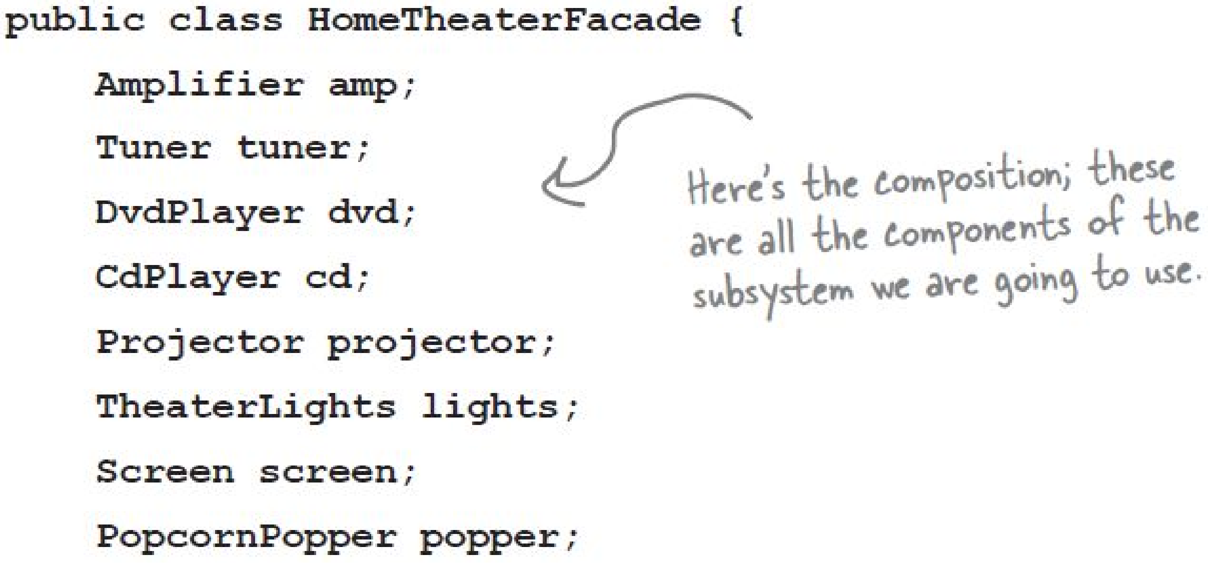
4 The Facade still leaves the subsystem accessible to be used directly. If you need the advanced functionality of the subsystem classes, they are available for your use.

The Facade and Adapter Patterns

- A facade not only simplifies an interface, it decouples a client from a subsystem of components.
- Facades and adapters may wrap multiple classes, but a facade's intent is to simplify, while an adapter's is to convert the interface to something different.

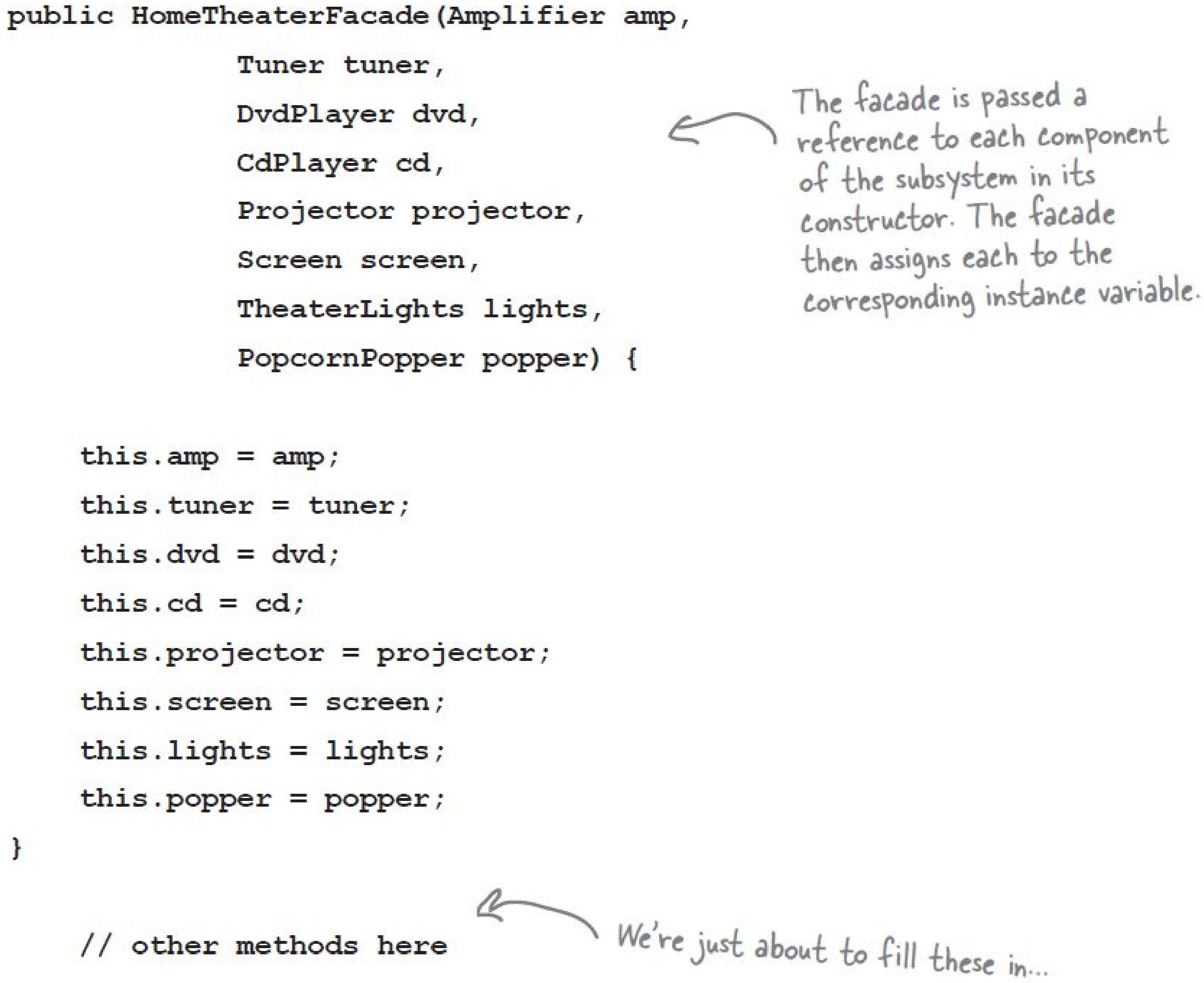
Constructing your home theater facade

- The first step is to use composition so that the facade has access to all the components of the subsystem:



```
public class HomeTheaterFacade {
    Amplifier amp;
    Tuner tuner;
    DvdPlayer dvd;
    CdPlayer cd;
    Projector projector;
    TheaterLights lights;
    Screen screen;
    PopcornPopper popper;
```

Here's the composition; these are all the components of the subsystem we are going to use.



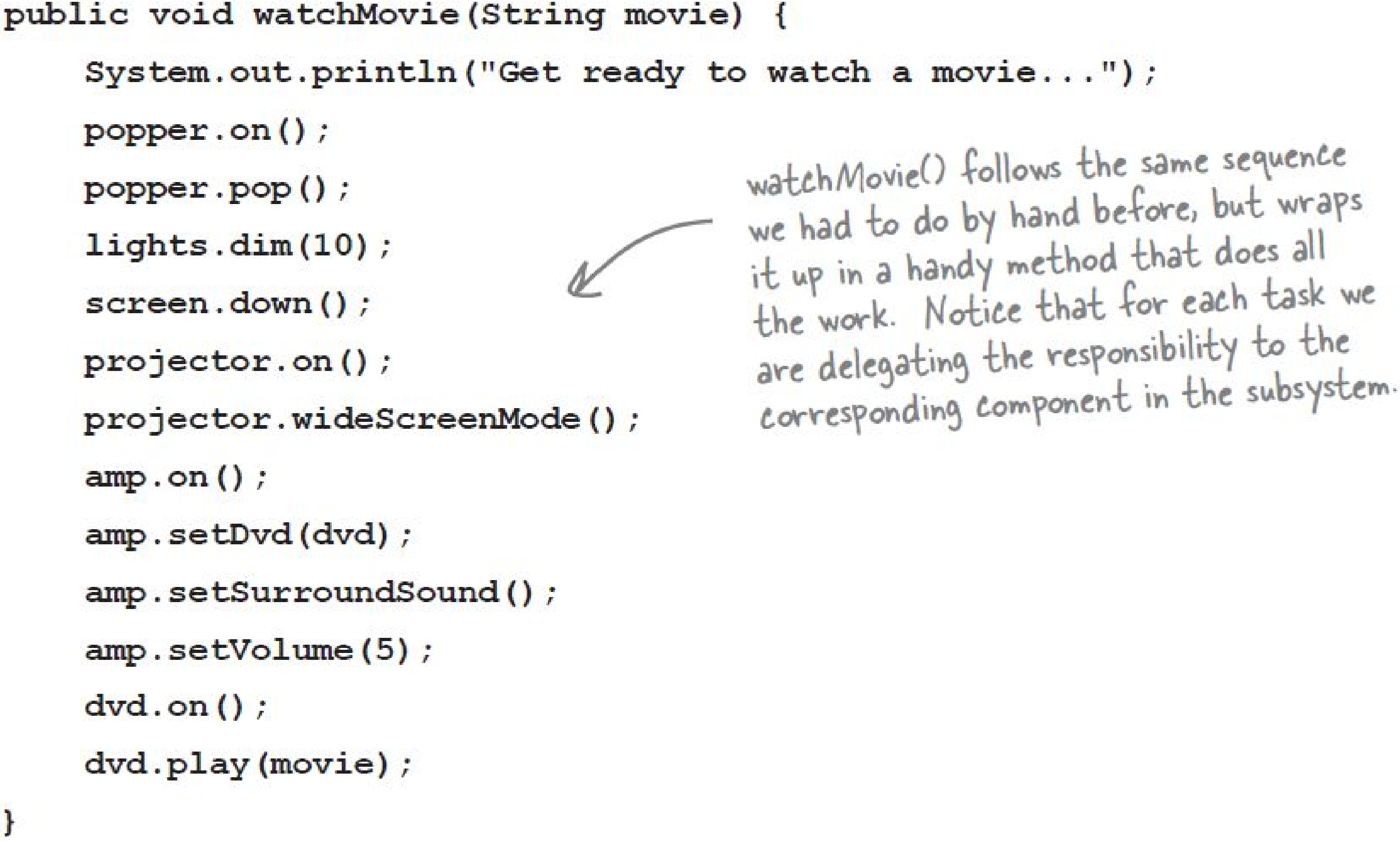
```
public HomeTheaterFacade(Amplifier amp,
             Tuner tuner,
             DvdPlayer dvd,
             CdPlayer cd,
             Projector projector,
             Screen screen,
             TheaterLights lights,
             PopcornPopper popper) {

    this.amp = amp;
    this.tuner = tuner;
    this.dvd = dvd;
    this.cd = cd;
    this.projector = projector;
    this.screen = screen;
    this.lights = lights;
    this.popper = popper;
}

    // other methods here
```

The facade is passed a reference to each component of the subsystem in its constructor. The facade then assigns each to the corresponding instance variable.

We're just about to fill these in...

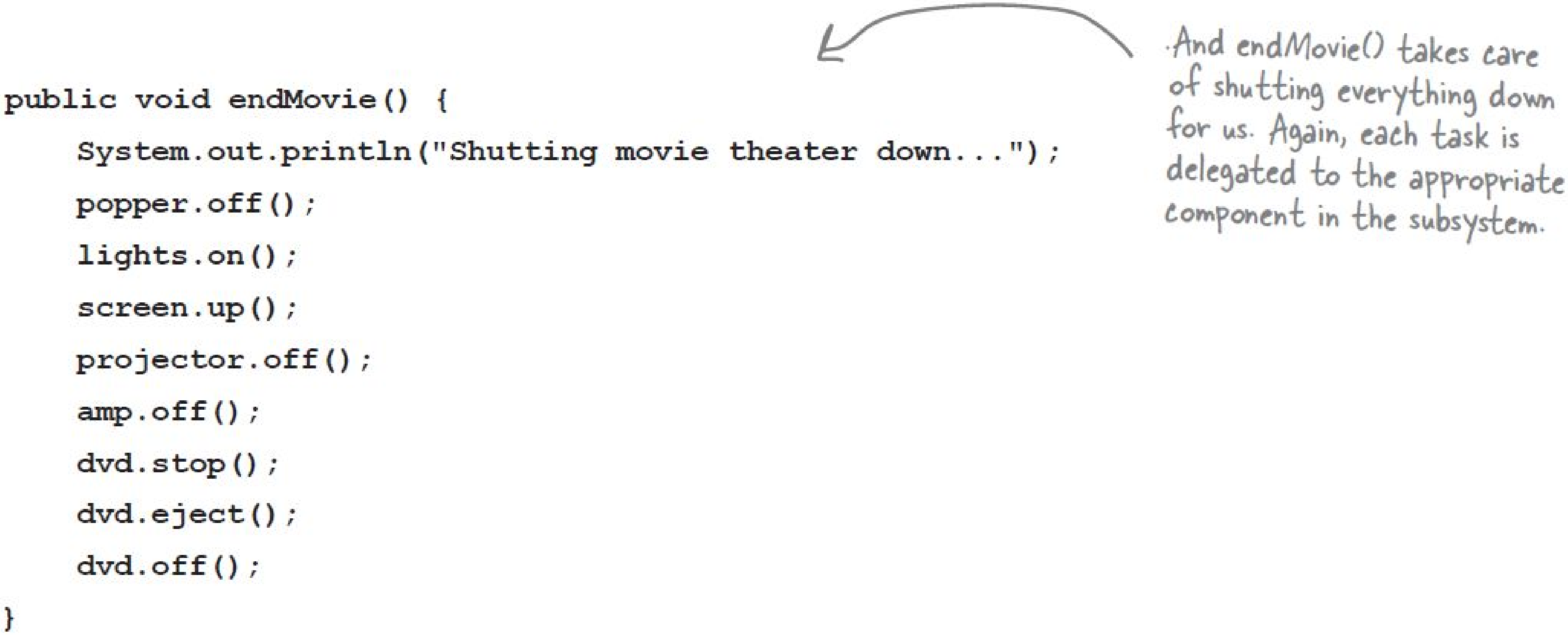
Implementing the simplified interface

- Now it’s time to bring the components of the subsystem together into a unified interface.
- Let’s implement the watchMovie() and endMovie() methods…



```
public void watchMovie(String movie) {
    System.out.println("Get ready to watch a movie...");
    popper.on();
    popper.pop();
    lights.dim(10);
    screen.down();
    projector.on();
    projector.wideScreenMode();
    amp.on();
    amp.setDvd(dvd);
    amp.setSurroundSound();
    amp.setVolume(5);
    dvd.on();
    dvd.play(movie);
}
```

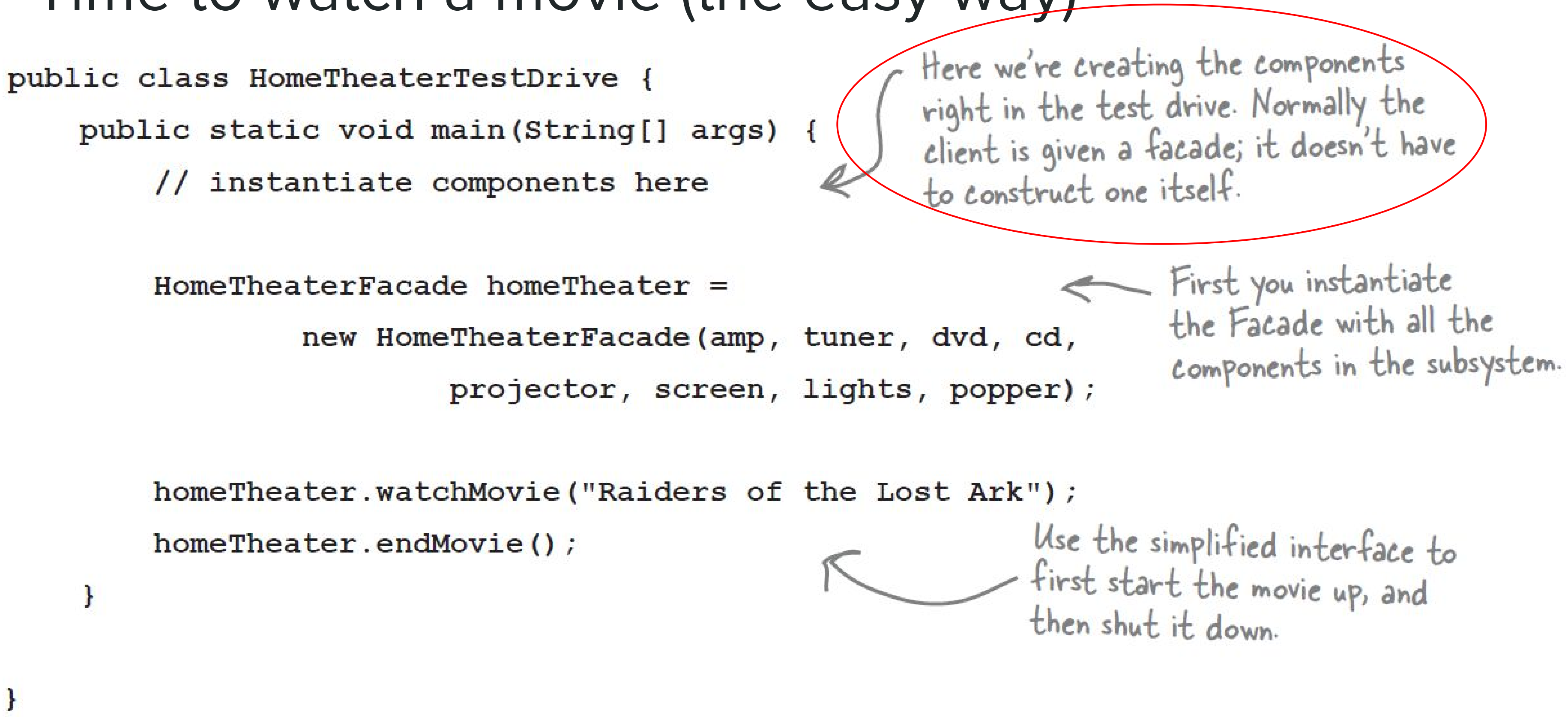
watchMovie() follows the same sequence we had to do by hand before, but wraps it up in a handy method that does all the work. Notice that for each task we are delegating the responsibility to the corresponding component in the subsystem.



```
public void endMovie() {
    System.out.println("Shutting movie theater down...");
    popper.off();
    lights.on();
    screen.up();
    projector.off();
    amp.off();
    dvd.stop();
    dvd.eject();
    dvd.off();
}
```

And endMovie() takes care of shutting everything down for us. Again, each task is delegated to the appropriate component in the subsystem.

Time to watch a movie (the easy way)



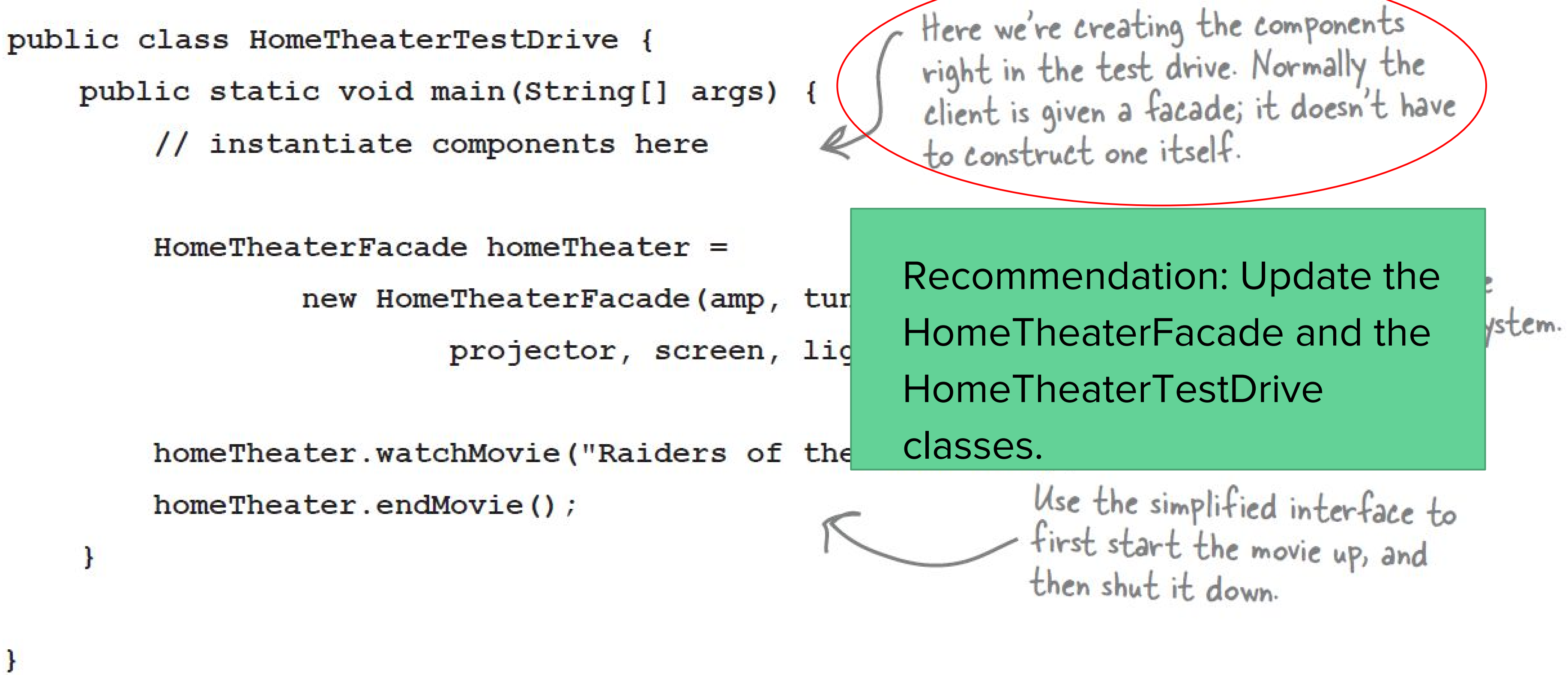
```
public class HomeTheaterTestDrive {
    public static void main(String[] args) {
        // instantiate components here

        HomeTheaterFacade homeTheater =
                new HomeTheaterFacade(amp, tuner, dvd, cd,
                        projector, screen, lights, popper);

        homeTheater.watchMovie("Raiders of the Lost Ark");
        homeTheater.endMovie();
    }
}
```

Here we're creating the components right in the test drive. Normally the client is given a facade; it doesn't have to construct one itself.

First you instantiate the Facade with all the components in the subsystem.

Use the simplified interface to first start the movie up, and then shut it down.

Time to watch a movie (the easy way)



```
public class HomeTheaterTestDrive {
    public static void main(String[] args) {
        // instantiate components here

        HomeTheaterFacade homeTheater =
                new HomeTheaterFacade(amp, tur
                        projector, screen, lig

        homeTheater.watchMovie("Raiders of the
        homeTheater.endMovie();
    }
}
```

Here we're creating the components right in the test drive. Normally the client is given a facade; it doesn't have to construct one itself.

e

ystem.

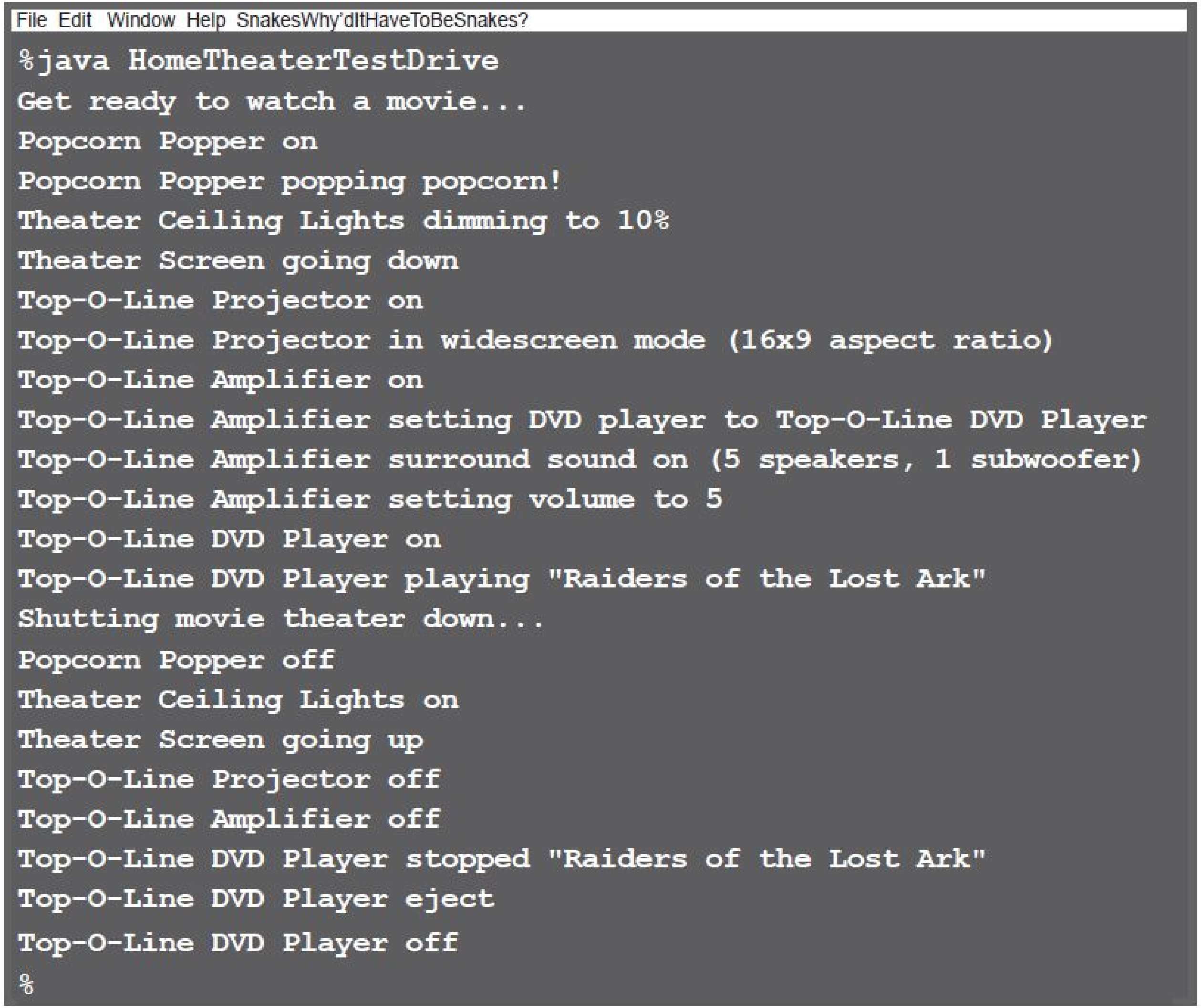
Use the simplified interface to first start the movie up, and then shut it down.

Recommendation: Update the HomeTheaterFacade and the HomeTheaterTestDrive classes.

Here's the output.

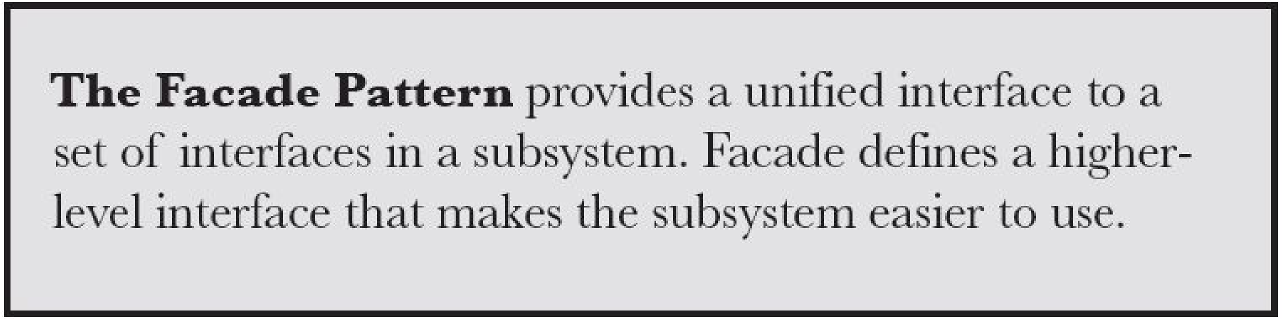
Calling the Facade's watchMovie() does all this work for us...

...and here, we're done watching the movie, so calling endMovie() turns everything off.



```
%java HomeTheaterTestDrive
Get ready to watch a movie...
Popcorn Popper on
Popcorn Popper popping popcorn!
Theater Ceiling Lights dimming to 10%
Theater Screen going down
Top-O-Line Projector on
Top-O-Line Projector in widescreen mode (16x9 aspect ratio)
Top-O-Line Amplifier on
Top-O-Line Amplifier setting DVD player to Top-O-Line DVD Player
Top-O-Line Amplifier surround sound on (5 speakers, 1 subwoofer)
Top-O-Line Amplifier setting volume to 5
Top-O-Line DVD Player on
Top-O-Line DVD Player playing "Raiders of the Lost Ark"
Shutting movie theater down...
Popcorn Popper off
Theater Ceiling Lights on
Theater Screen going up
Top-O-Line Projector off
Top-O-Line Amplifier off
Top-O-Line DVD Player stopped "Raiders of the Lost Ark"
Top-O-Line DVD Player eject
Top-O-Line DVD Player off
%
```

Facade Pattern defined

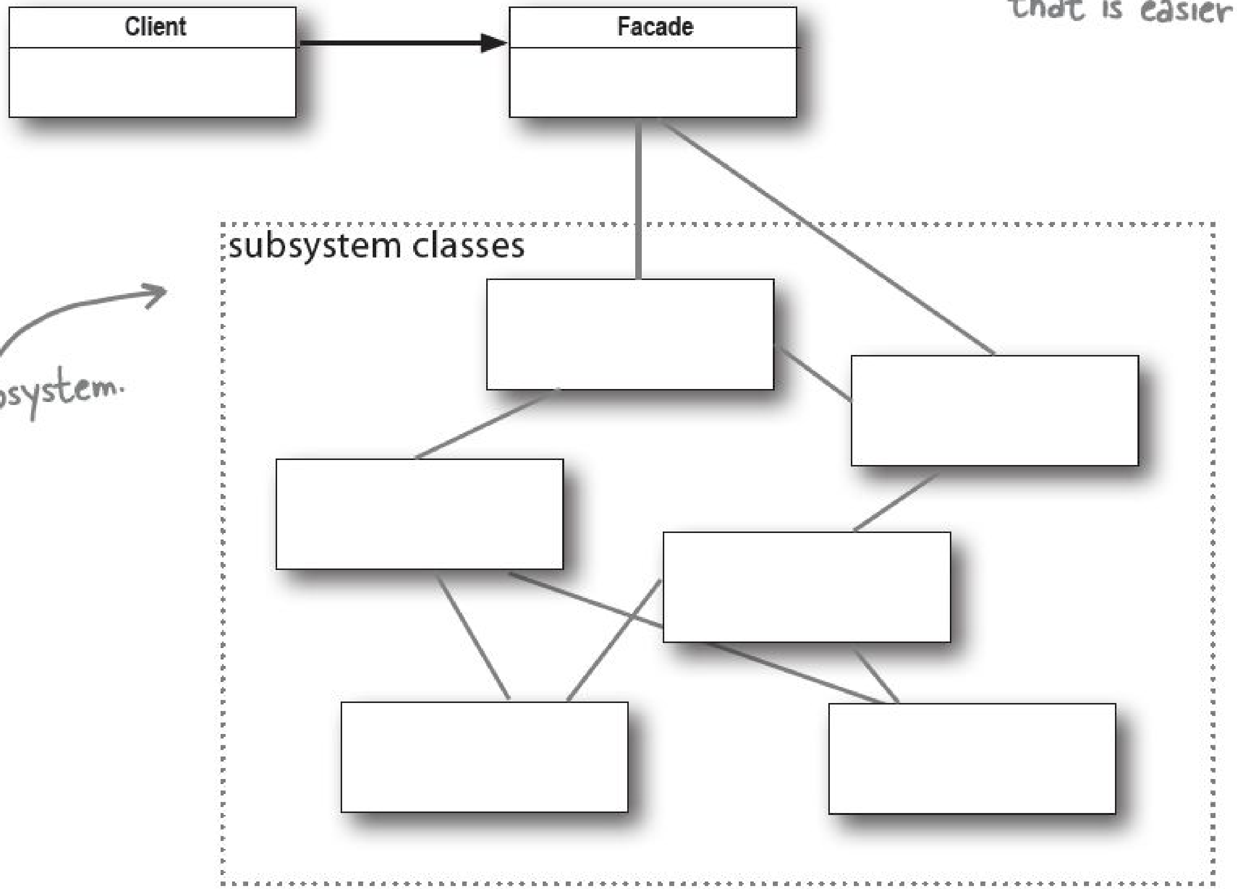


The Facade Pattern provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

- To use the Facade Pattern, we create a class that simplifies and unifies a set of more complex classes that belong to some subsystem.
- The Facade Pattern allows us to avoid tight coupling between clients and subsystems, and, as you will see shortly, also helps us adhere to a new object-oriented principle.

Facade Pattern defined (cont.)

- One of the most important things to remember about a pattern is its intent.
- This definition tells us loud and clear that the purpose of the facade is to make a subsystem easier to use through a simplified interface.
- You can see this in the pattern's class diagram...



Happy client whose job just became easier because of the facade.

Unified interface that is easier to use.

More complex subsystem.

The Principle of Least Knowledge

- The Principle of Least Knowledge guides us to reduce the interactions between objects to just a few close “friends.” The principle is usually stated as:

Design principle: Talk only to your immediate friends.

- It means when you are designing a system, for any object, be careful of the number of classes it interacts with and also how it comes to interact with those classes.

The Principle of Least Knowledge (cont.)

- This principle prevents us from creating designs that have a large number of classes coupled together so that changes in one part of the system cascade to other parts.
- When you build a lot of dependencies between many classes, you are building a fragile system that will be costly to maintain and complex for others to understand.

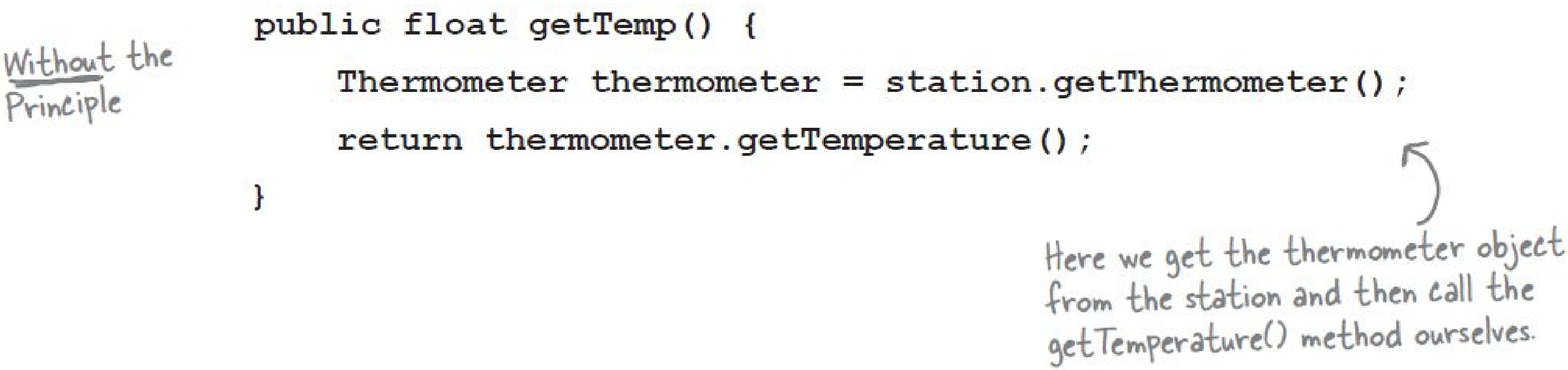
How NOT to Win Friends and Influence Objects

- Okay, but how do you keep from doing this?
- The principle provides some guidelines: take any object; now from any method in that object, the principle tells us that we should only 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 components of the object

Notice that these guidelines tell us not to call methods on objects that were returned from calling other methods!

Think of a “component” as any object that is referenced by an instance variable. In other words, think of this as a HAS-A relationship.

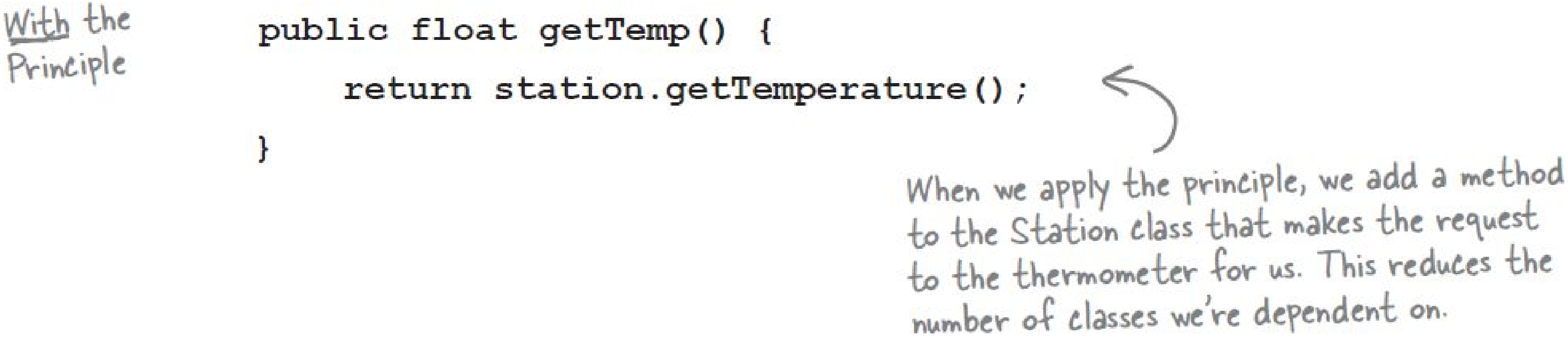
Examples



Without the Principle

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

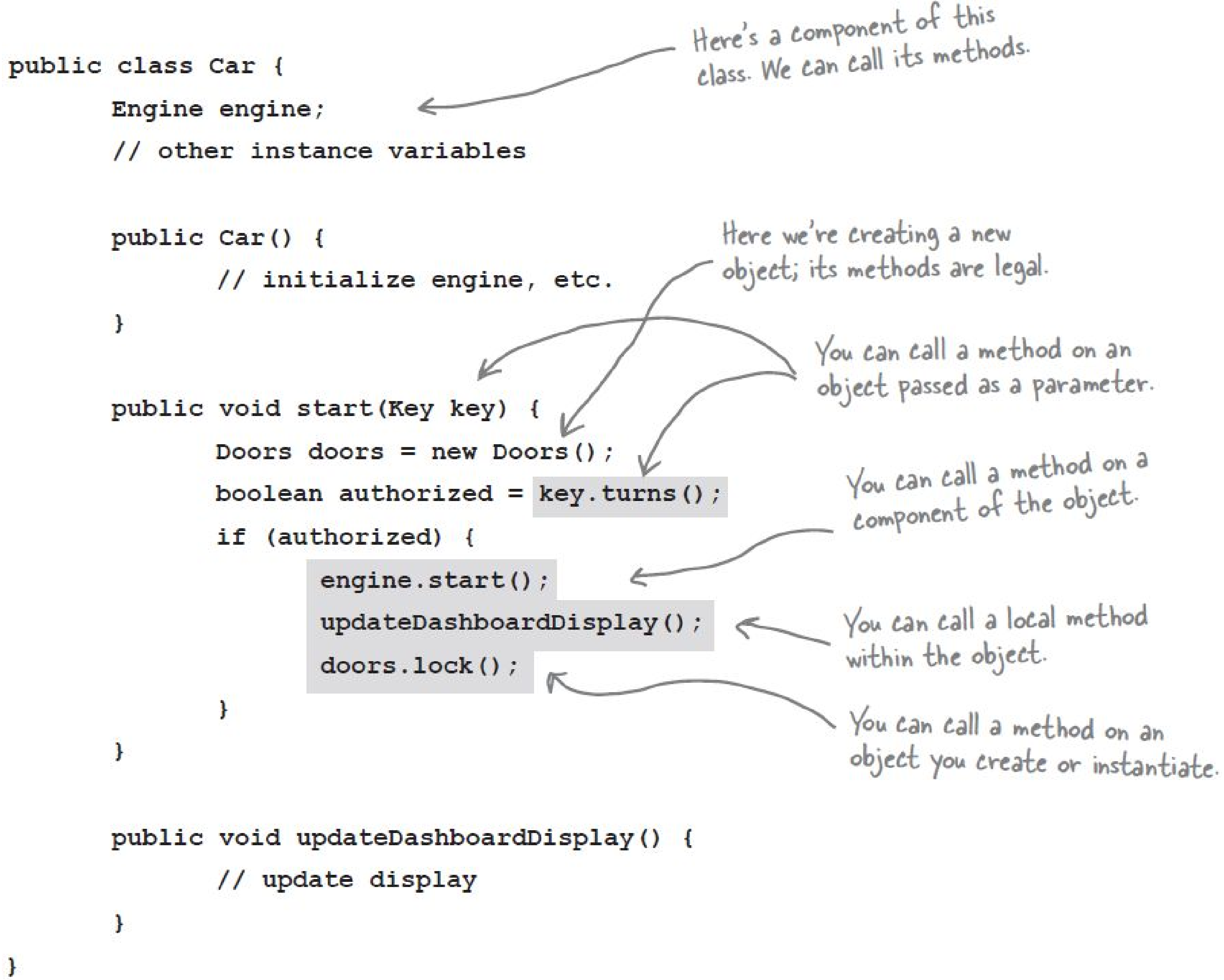
Here we get the thermometer object from the station and then call the getTemperature() method ourselves.



With the Principle

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

When we apply the principle, we add a method to the Station class that makes the request to the thermometer for us. This reduces the number of classes we're dependent on.



```
public class Car {
      Engine engine;
      // other instance variables

      public Car() {
            // initialize engine, etc.
      }

      public void start(Key key) {
            Doors doors = new Doors();
            boolean authorized = key.turns();
            if (authorized) {
                  engine.start();
                  updateDashboardDisplay();
                  doors.lock();
            }
      }

      public void updateDashboardDisplay() {
            // update display
      }
}
```

Here's a component of this class. We can call its methods.

Here we're creating a new object; its methods are legal.

You can call a method on an object passed as a parameter.

You can call a method on a component of the object.

You can call a local method within the object.

You can call a method on an object you create or instantiate.

Here's a Car class that demonstrates all the ways you can call methods and still adhere to the Principle of Least Knowledge.

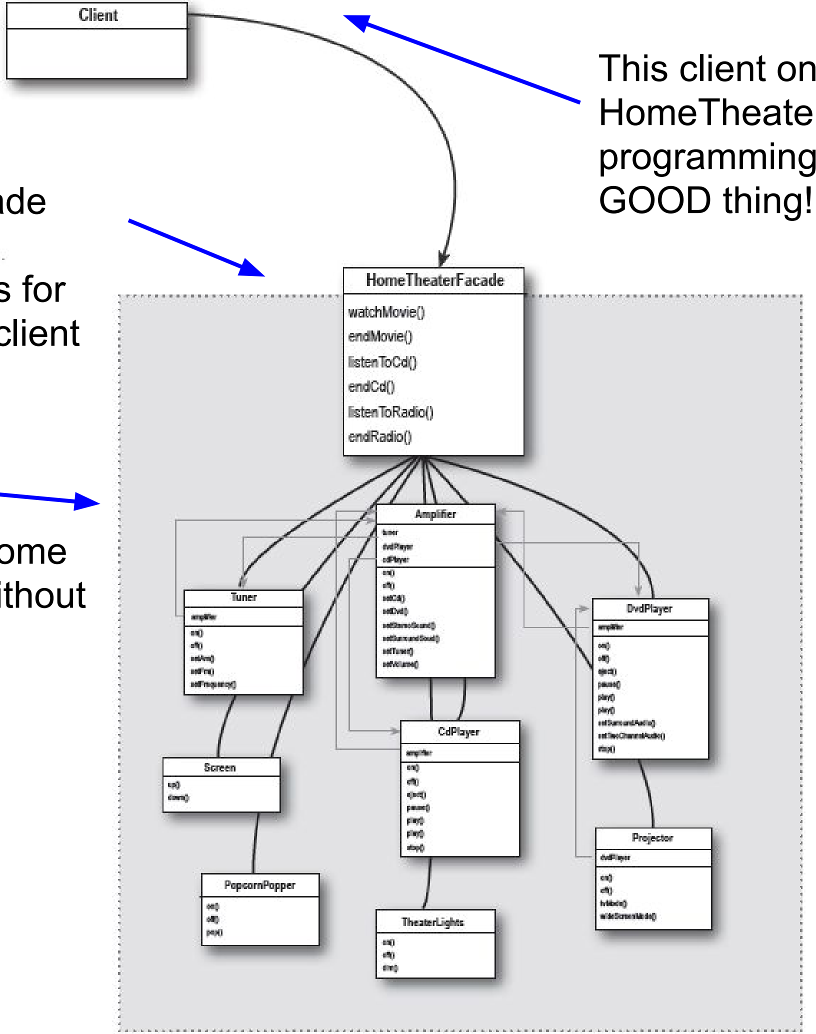
The Facade and the Principle of Least Knowledge

This client only has one friend: the HomeTheaterFacade. In OO programming, having only one friend is a GOOD thing!

The HomeTheaterFacade manages all those subsystem components for the client. It keeps the client simple and flexible.

We can upgrade the home theater components without affecting the client.

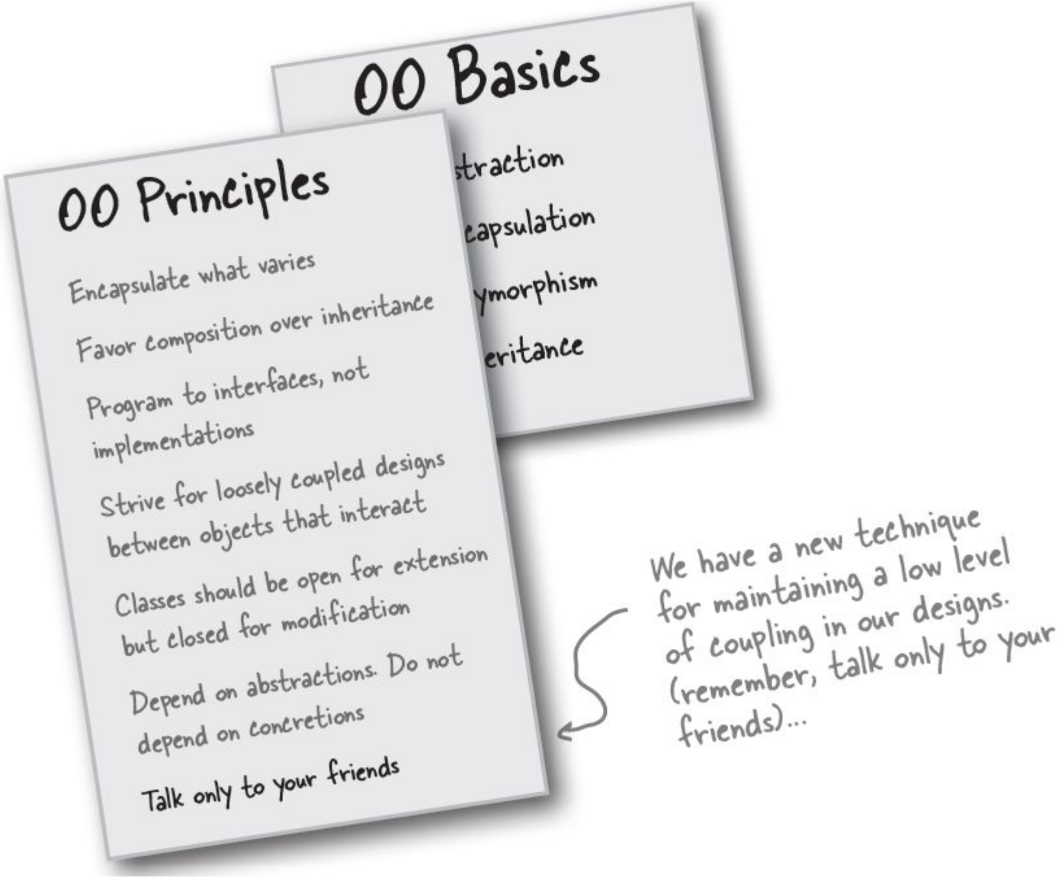
We try to keep subsystems adhering to the Principle of Least Knowledge as well. If this gets too complex and too many friends are intermingling, we can introduce additional facades to form layers of subsystems.



Some Bullet Points

- When you need to use an existing class and its interface is not the one you need, use an adapter.
- When you need to simplify and unify a large interface or complex set of interfaces, use a facade.
- An adapter changes an interface into one a client expects.
- A facade decouples a client from a complex subsystem.
- You can implement more than one facade for a subsystem.
- An adapter wraps an object to change its interface, a decorator wraps an object to add new behaviors and responsibilities, and a facade “wraps” a set of objects to simplify.

Tools for your Design Toolbox



OO Basics

straction

capsulation

ymorphism

eritance

OO Principles

Encapsulate what varies

Favor composition over inheritance

Program to interfaces, not implementations

Strive for loosely coupled designs between objects that interact

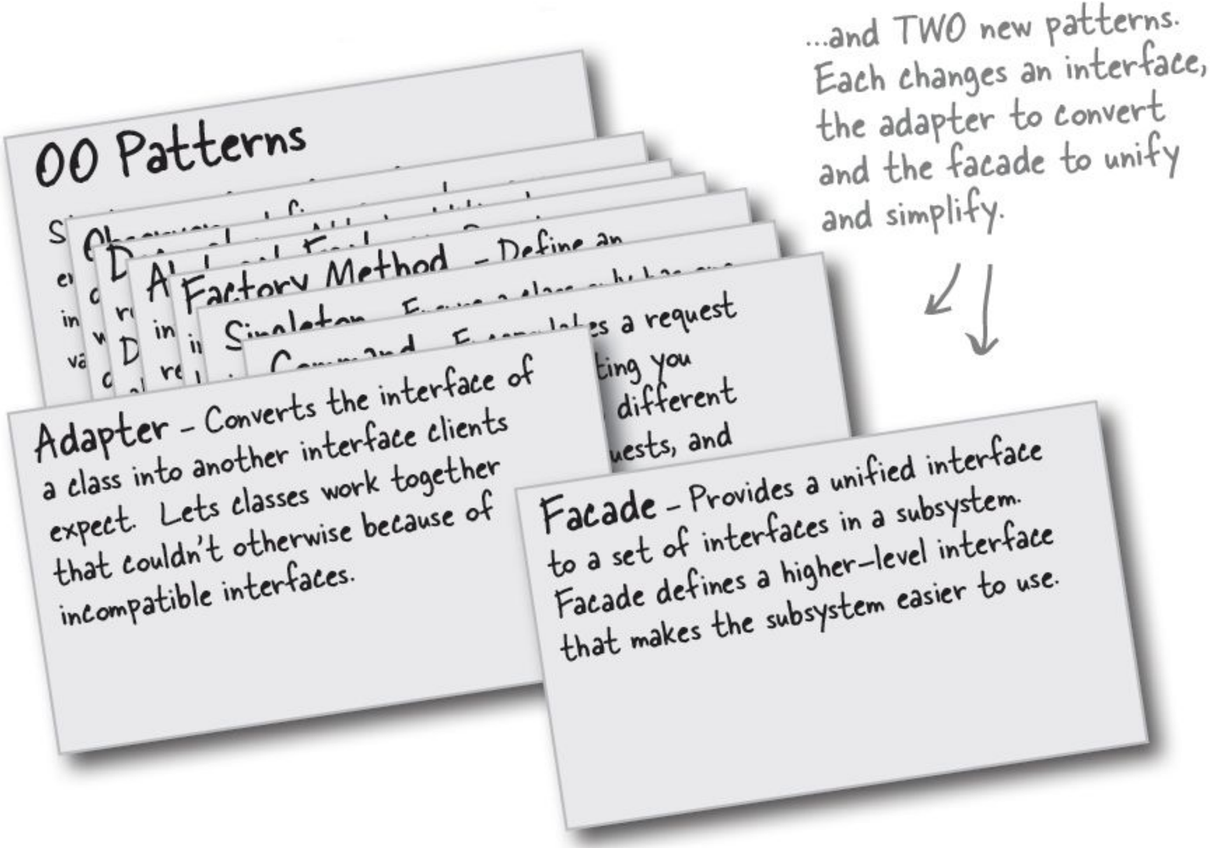
Classes should be open for extension but closed for modification

Depend on abstractions. Do not depend on concretions

Talk only to your friends

We have a new technique for maintaining a low level of coupling in our designs. (remember, talk only to your friends)...

Tools for your Design Toolbox (cont.)



...and TWO new patterns. Each changes an interface, the adapter to convert and the facade to unify and simplify.

Adapter – Converts the interface of a class into another interface clients expect. Lets classes work together that couldn't otherwise because of incompatible interfaces.

Facade – 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.

References

Material in this lecture is taken from Freeman, E., Robson, E., Bates, B., & Sierra, K., *Head First Design Patterns: A Brain-Friendly Guide*, O'Reilly Media, Inc., 2014.