CS 351
Design of Large Programs
Adapter Pattern and Facade Pattern

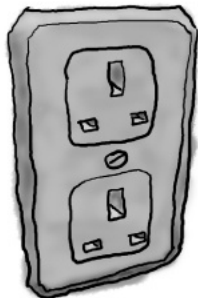
Brooke Chenoweth

University of New Mexico

Spring 2024

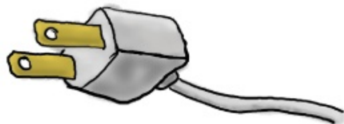
The Adapter Analogy

British Wall Outlet



AC Power Adapter

Standard AC Plug

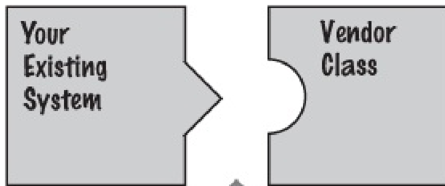


The British wall outlet exposes one interface for getting power.

The adapter converts one interface into another.

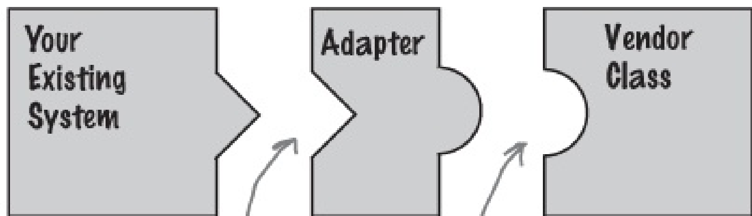
The US laptop expects another interface.

The OO Adapter

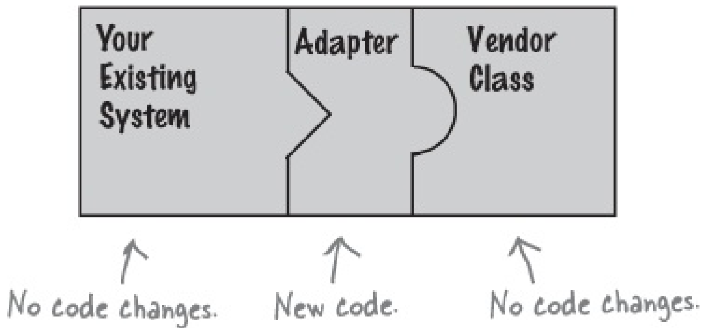


Their interface doesn't match the one you've written your code against. This isn't going to work!

The OO Adapter



The OO Adapter



Writing an Adaptor

Back to Ducks. . . but with interfaces this time:

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

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

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

A New Fowl. . .

```
public interface Turkey {
  public void gobble();
  public 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 amount!");
  }
}
```

Here's my problem. . .

- I need ducks
- But what I have are turkeys
- Is there any way to turn a turkey into a duck?

The TurkeyAdapter

```
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();
    }
  }
}
```

The TurkeyAdapter

```
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();
    }
  }
}
```

Implement the target interface

The TurkeyAdapter

```
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();
    }
  }
}
```

Implement the target interface

reference to object we're adapting

The TurkeyAdapter

```
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();
    }
  }
}
```

Implement the target interface

reference to object we're adapting

Gobble is like a quack, right?

The TurkeyAdapter

```
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();
    }
  }
}
```

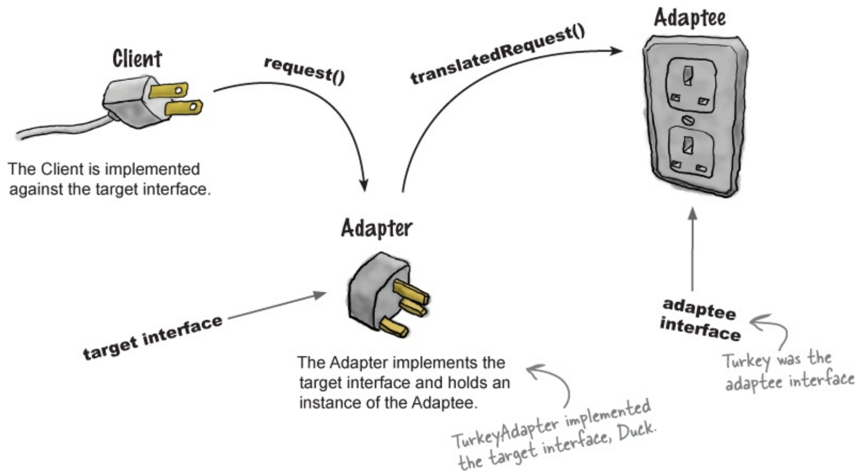
Implement the target interface

reference to object we're adapting

Gobble is like a quack, right?

Turkey's fly is different than duck's

The Adapter Pieces



Questions

How much adapting can be done in an adapter?

Questions

How much adapting can be done in an adapter?

- That really depends on the particular situation and the particular interfaces. It could be just basic translation or massive amounts of work

Questions

How much adapting can be done in an adapter?

- That really depends on the particular situation and the particular interfaces. It could be just basic translation or massive amounts of work

Does an adapter always wrap only one class?

Questions

How much adapting can be done in an adapter?

- That really depends on the particular situation and the particular interfaces. It could be just basic translation or massive amounts of work

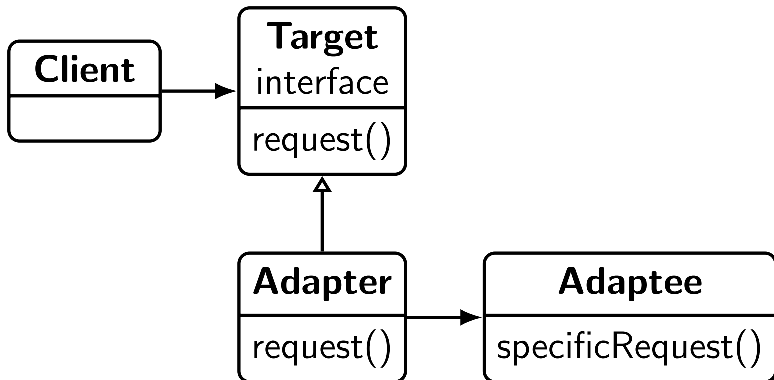
Does an adapter always wrap only one class?

- The real world can be messier; an adapter could wrap two or more adaptees needed to implement the target interface
- However, the Adapter always converts one interface to another (the point is just that the definition of "interface" may not be limited to a single class)

The Adapter Pattern

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

The Adapter Class Diagram



Adapter Pattern and Good OO Design

- Favor composition
  The adaptee is wrapped with an altered interface using composition
- Programming to an interface not an implementation
  The client is only aware of the target interface

An Example Adapter in the Wild

- Old World Enumerations...
  Early Java "collections" types implemented an `elements()` method that you could then step through without knowing how the collection was implemented
- New World Iterators...
  The Collections classes use an `Iterator` that implements a similar capability but also allows item removal
- Never the twain shall meet
  Oh, wait...

Backwards Compatibility

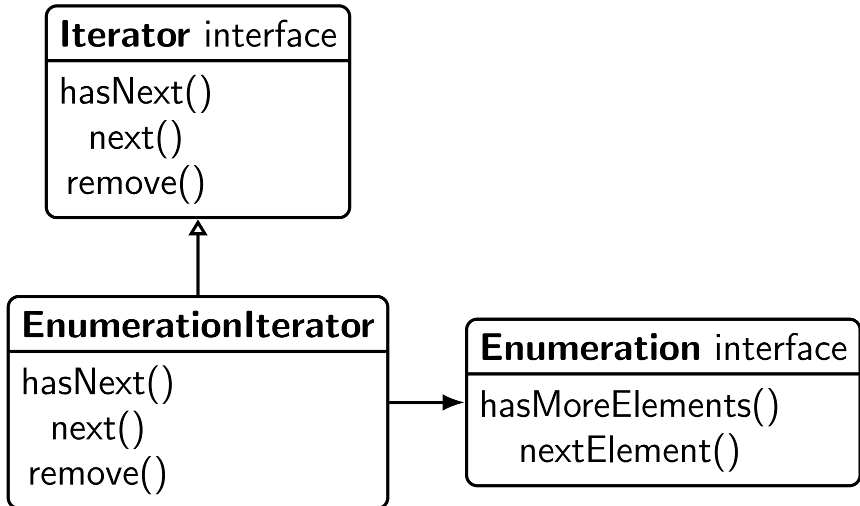
- You know what that means, right?
- We often have to work with legacy code that does it the old way.
- But our clients insist (and they should) on using the new way.

Let's Examine the Interfaces

Iterator interface
hasNext()
next()
remove()

Enumeration interface
hasMoreElements()
nextElement()

The Class Diagram



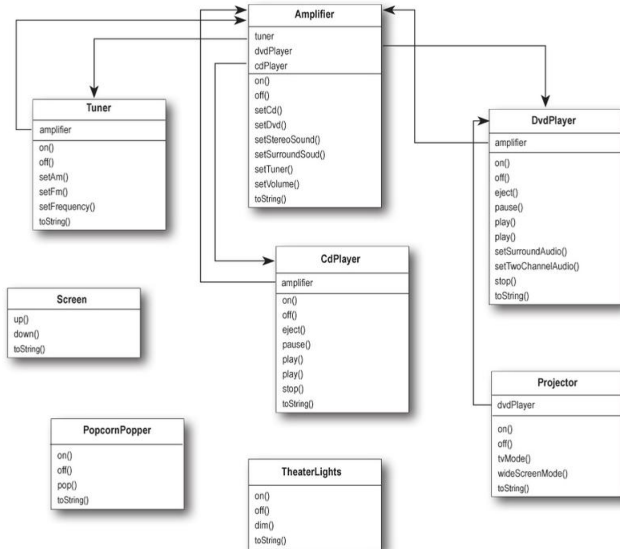
How to Handle the `remove()` method

- Enumeration simply doesn't support remove; it's "read only"
- We can, however, throw a runtime exception if someone tries to call `remove()` on an `EnumerationIterator`
  The Iterator class supports this; it's remove method supports throwing an `UnsupportedOperationException`
- In the end, the adapter isn't perfect; clients will still have to deal with potential exceptions

The EnumerationIterator Adapter

```
public class EnumerationIterator implements Iterator {
  private Enumeration en;

  public EnumerationIterator(Enumeration en) {
    this.en = en;
  }

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

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

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

A Home Theater



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

Watching a Movie

Sit back, relax, and…

Watching a Movie

Sit back, relax, and. . .

1. Turn on the popcorn popper
2. Start the popper popping
3. Dim the lights
4. Put the screen down
5. Turn the projector on
6. Set the projector input to DVD
7. Put the projector in wide-screen mode
8. Turn the sound amplifier on
9. Set the amplifier to DVD input
10. Set the amplifier to surround sound
11. Set the amplifier volume to medium (5)
12. Turn the DVD player on

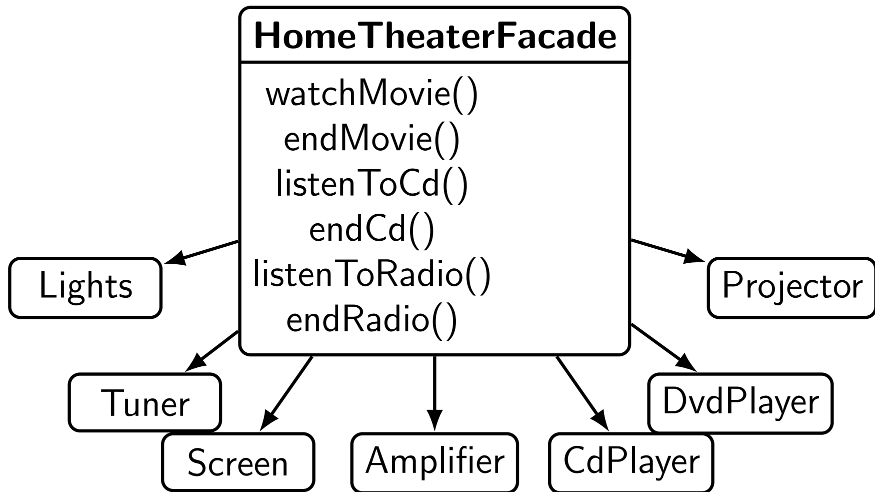
Further Complications. . .

- When the movie finishes, you have to do it all in reverse!
- Doing a slightly different task (e.g., listen to streaming audio) is equally complex
- When you upgrade your system, you have to learn a slightly different procedure

Home Theater Facade

- Time to create a Facade for the home theater system.
- We create a new class `HomeTheaterFacade`, which exposes a few simple methods such as `watchMovie()`
- The Facade class treats the home theater components as a subsystem, and calls on the subsystem to implement its `watchMovie()` method

Home Theater Facade



Sidebar: Facade vs Adapter

- A facade not only simplifies an interface, but it also decouples a client from a subsystem of components
- Facades and adapters may wrap multiple classes
  - A facade's intent is to *simplify*
  - An adapter's intent is to *convert* the interface into something different
- A facade does not encapsulate; it just provides a simplified interface

The Home Theater Facade

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

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

The Home Theater Facade

```
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);
}
```

The Home Theater Facade

```
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();
}
```

The Facade Pattern

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.

A Sidebar: The Principle of Least Knowledge

As a general design principle, you should reduce the interactions between objects to just a few "close friends"

- Be careful of the number of classes an object interacts with and also how it comes to interact with those classes
- Prevents creating designs that have a very high degree of coupling among classes (these systems are much more fragile)

A Sidebar: The Principle of Least Knowledge

General guidelines:

- Only invoke methods that belong to
  - The object itself
  - Objects passed as parameters to the method
  - Any object the method creates or instantiates
  - Any components of the object
- Do *not* invoke methods on objects that were returned from calling other methods!

An Example

Without the Principle

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

We get the thermometer object from the station and call the method ourselves.

With the Principle

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

We add a method to the Station class that makes a request to the thermometer for us, reducing the number of classes we're dependent on.

Principle of Least Knowledge

Advantages

- Reduces dependencies between objects, which has been demonstrated to reduce maintenance costs

Disadvantages

- Results in more "wrapper" classes to avoid method calls to other components
- This can result in increased complexity and development time

Examples of Good Practice

```
public class Car {
  private Engine engine;
  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
  }
}
```

Exercise

Does this violate the Principle of Least Knowledge?

```
public House {
  private WeatherStation station;

  // other methods and constructor

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

Exercise

Does this violate the Principle of Least Knowledge?

```
public House {
  private WeatherStation station;

  // other methods and constructor

  public float getTemp() {
    Thermometer therm = station.getThermometer();
    return getTempHelper(therm);
  }

  public float getTempHelper(Thermometer therm) {
    return therm.getTemperature();
  }
}
```

Facade and the Principle of Least Knowledge

- The client only has one friend, the HomeTheaterFacade.
- The HomeTheaterFacade manages all the subsystem components for the client, keeping the client simple and flexible.
- We can upgrade the home theater components without affecting the client
- Try to keep subsystems adhering to the Principle of Least Knowledge as well.
  If it gets too complex, we can add more facades to form layers of subsystems.