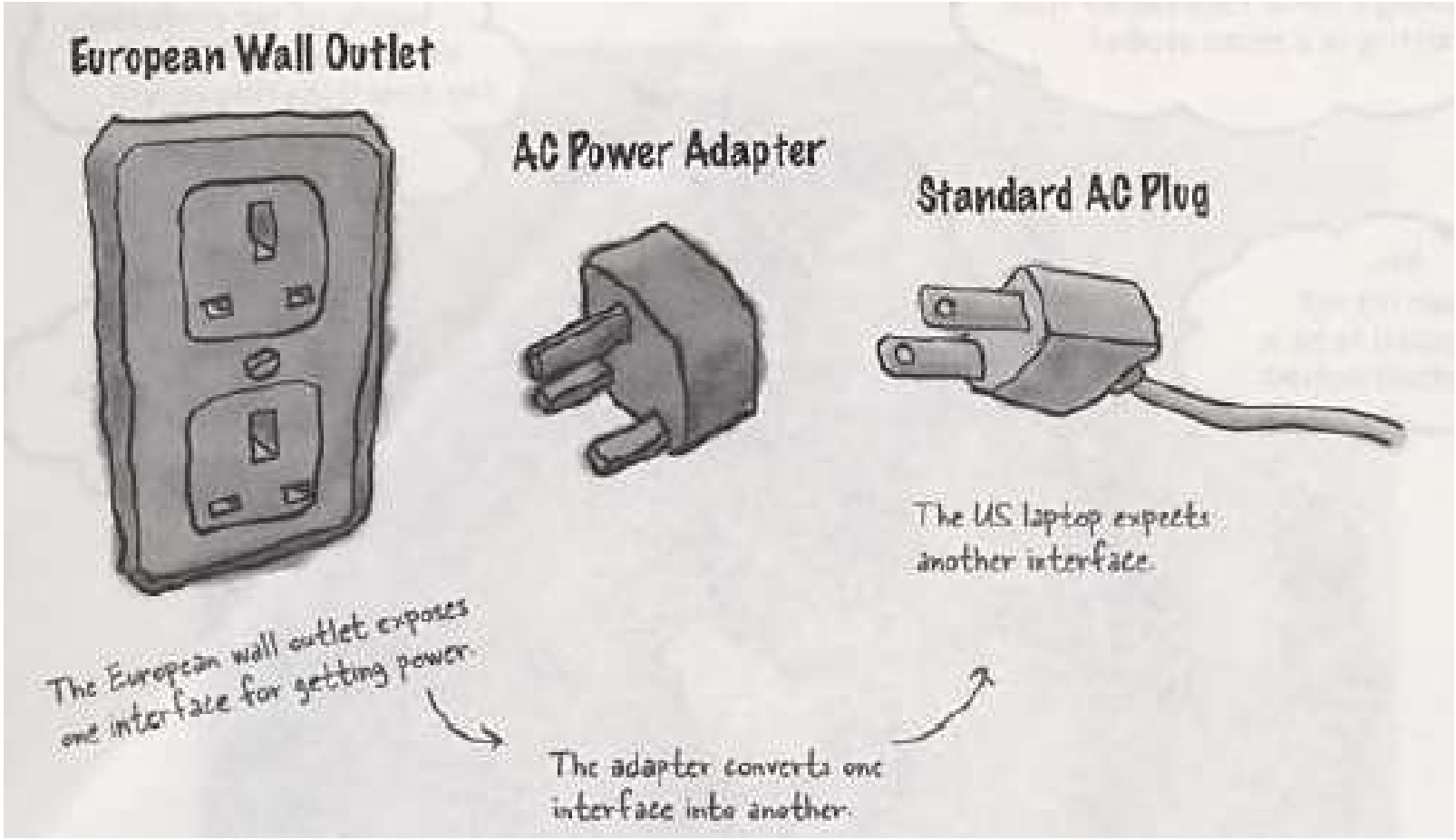
The Adapter Pattern

Putting a Square Peg in a Round Hole!

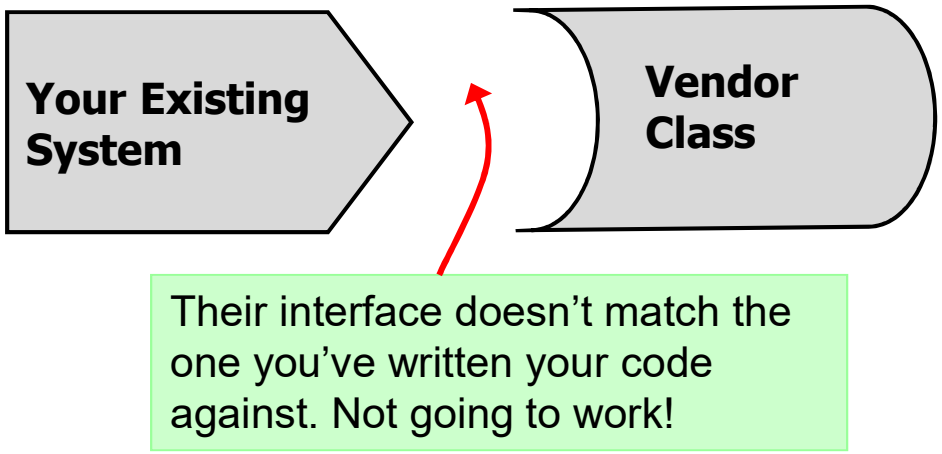
What is Adapters

- Real world is full of them!



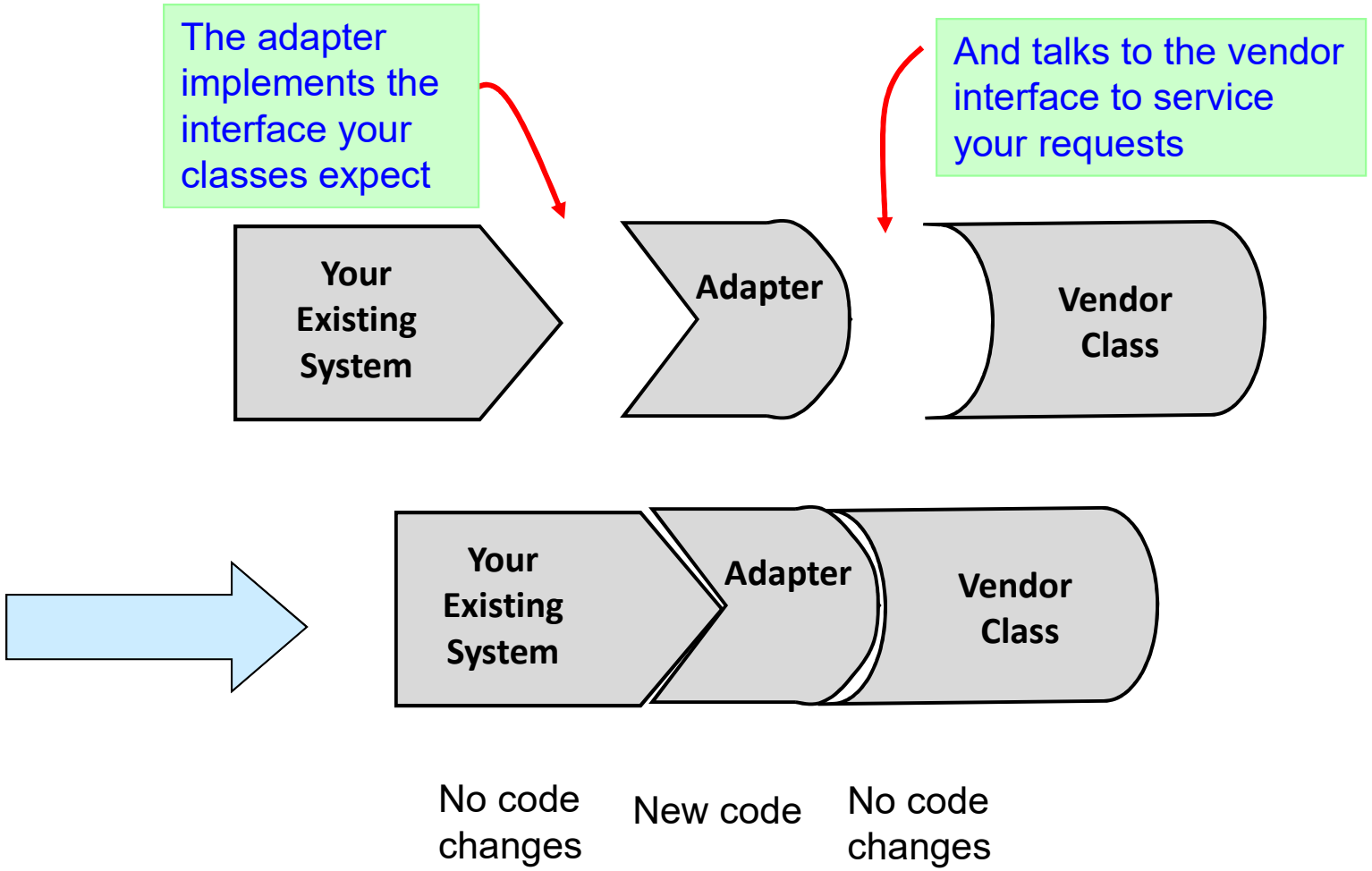
Object oriented adapters

- Scenario:
  - you have an existing software system that you need to work a new vendor library into, but the new vendor designed their interfaces differently than the last vendor.



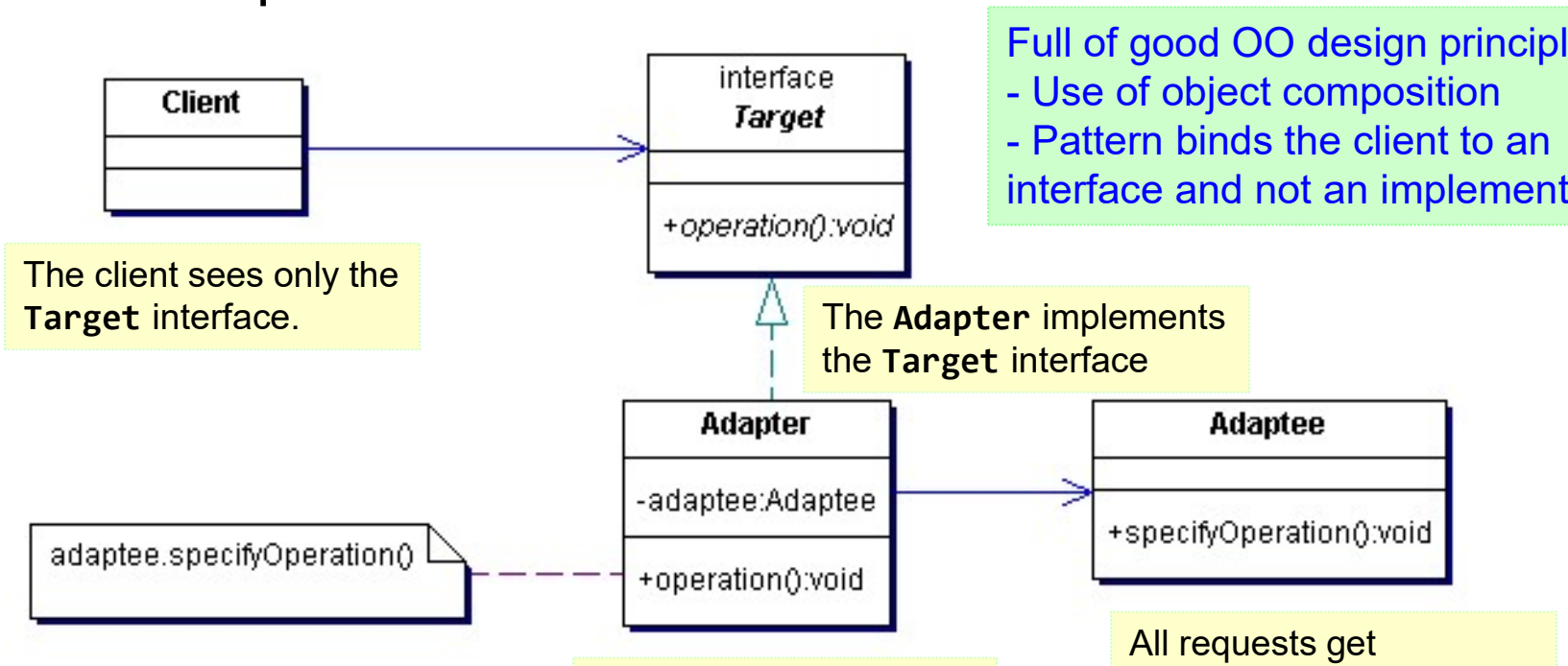
- What to do? Write a class that adapts the new vendor interface into the one you're expecting.

Object oriented adapters



The Adapter Pattern - Intent

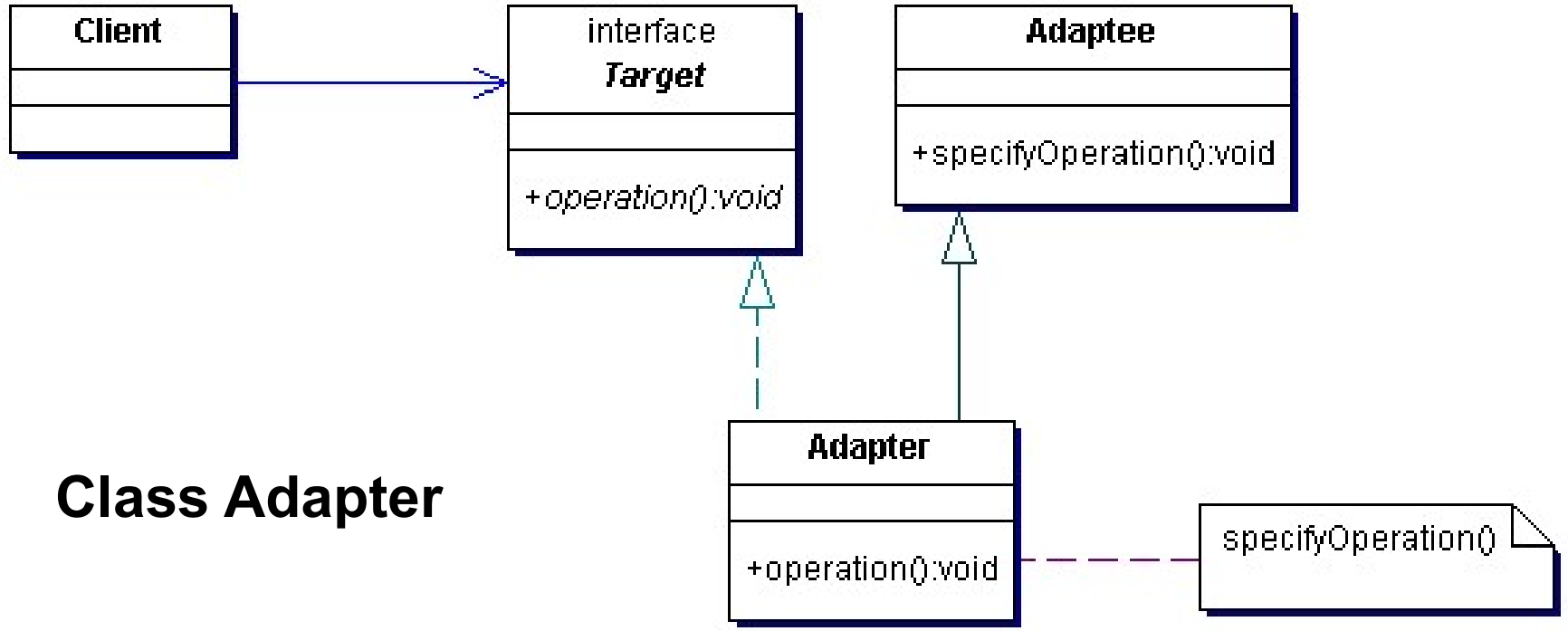
- The Adapter Pattern **converts the interface** of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.



Object Adapter

Object and Class Adapters

- There are two types of Adapters
  - ***Object* Adapter** : use composition to adaptive the adaptee.
  - ***Class* Adapter** : use inheritance.



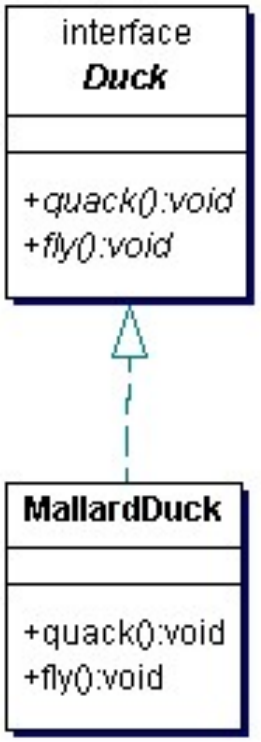
Class Adapter

Applicability

- Use the Adapter pattern when
  - want to use an existing class, and its interface does not match the one you need.
  - want to create a reusable class that cooperates with unrelated or unforeseen classes that don't necessarily have compatible interfaces.
- Class and object adapters have different trade-offs.
  - A class adapter won't work when we want to adapt a class and all its subclasses.
  - An object adapter lets a single Adapter work with the Adaptee itself and all of its subclasses (if any).

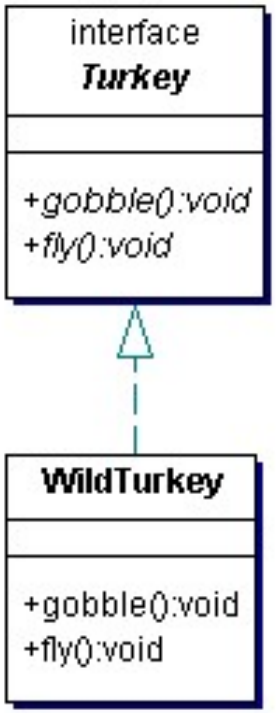
Example

Target interface

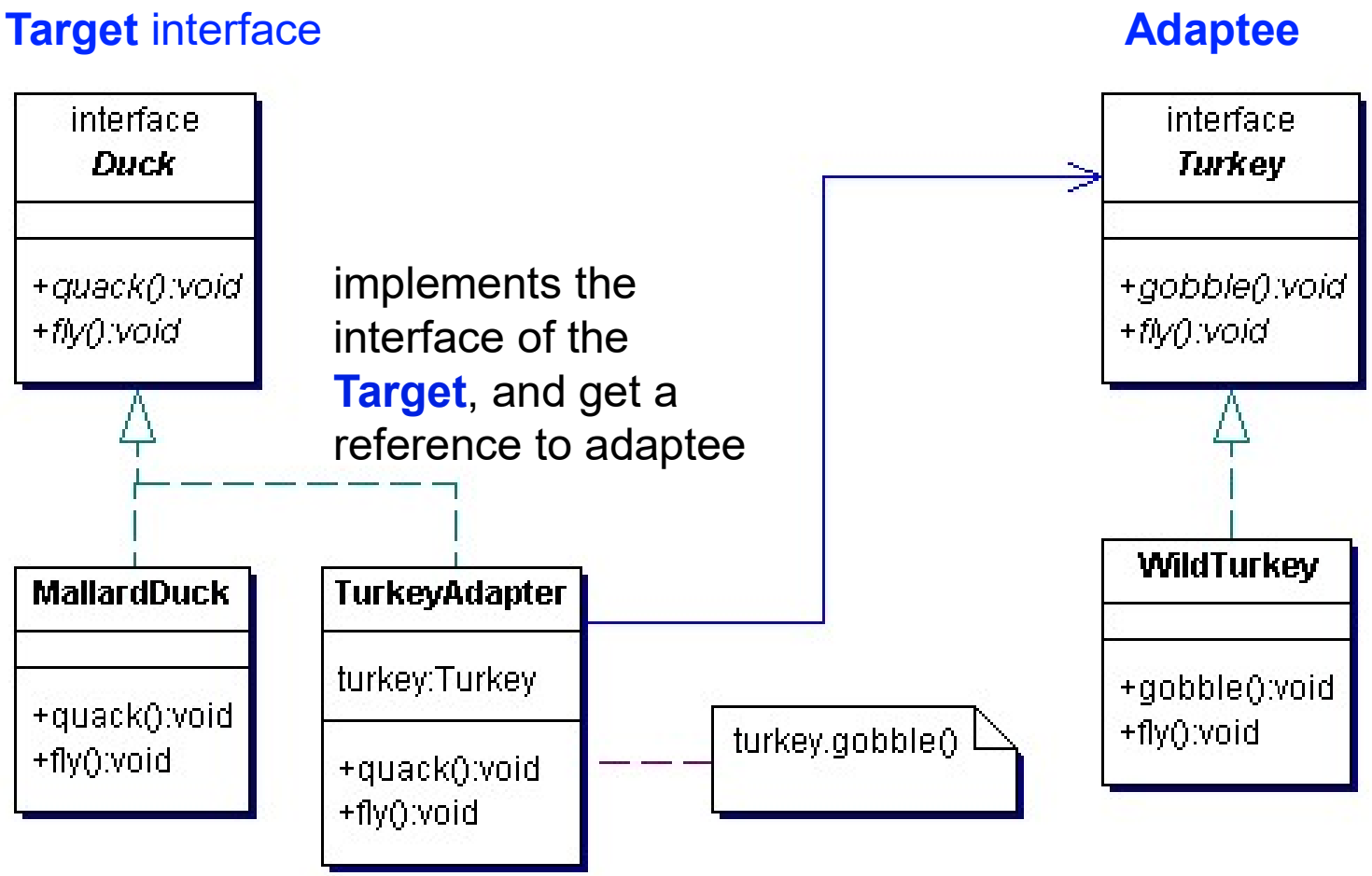


Turkey has a incompatible interface with Duck. We'd like to use some Turkey as Duck

Adaptee



Write Adapter



Turkey world code

```
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 distance");
   }
}
```

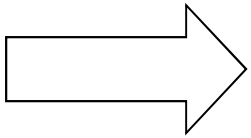
Duck and TurkeyAdapter

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

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

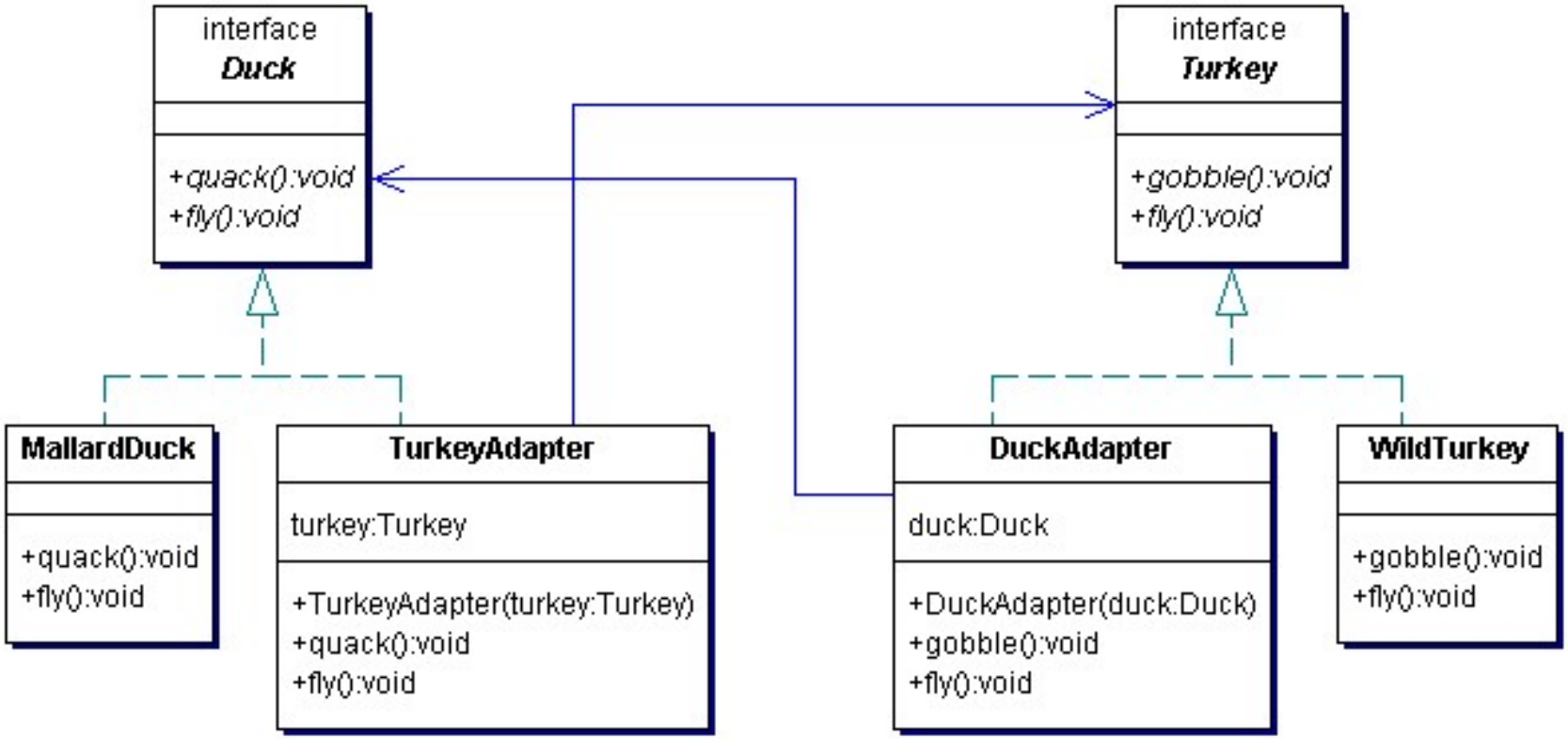
Test Drive

```
public class DuckTestDrive {
  public static void main(String[] args) {
    Turkey turkey = new WildTurkey();
    Duck turkeyAdapter = new TurkeyAdapter(turkey);

    System.out.println("The Turkey says...");
    turkey.gobble();
    turkey.fly();
    System.out.println("\nThe TurkeyAdapter says...");
    turkeyAdapter.quack();
    turkeyAdapter.fly();
  }
}
```



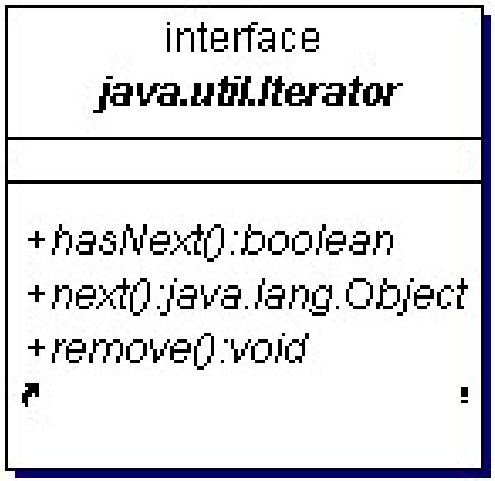
```
The Turkey says...
Gobble gobble
I'm flying a short distance

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
```

Using two-way adapters to provide transparency

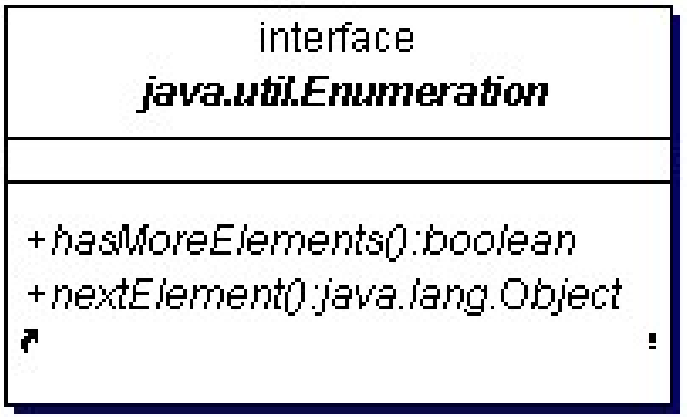


Example: Adapting an `Enumeration` to an `Iterator`

Target interface

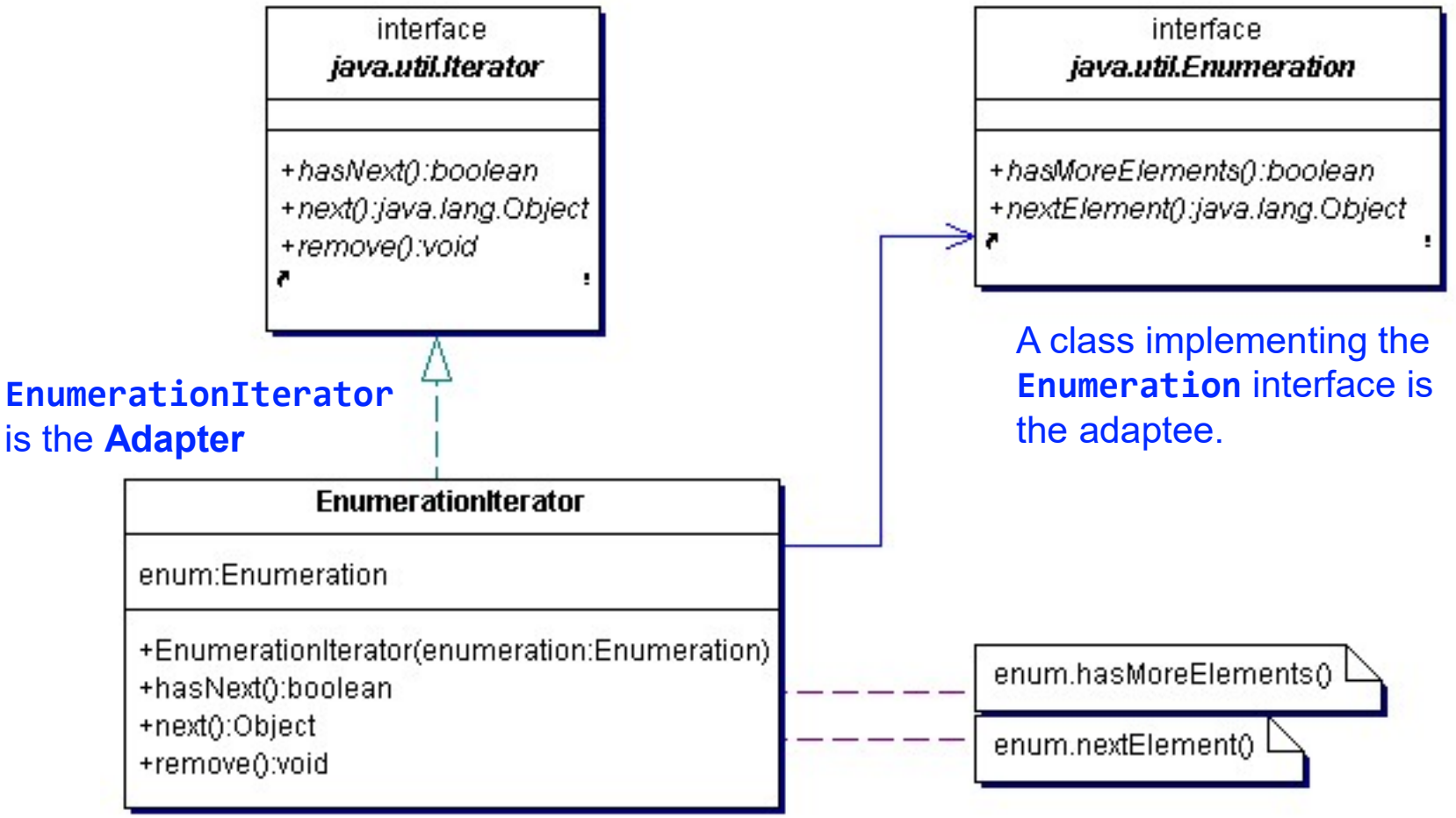


Adaptee interface



We are making the `Enumeration` in your old code look like `Iterator` for your new code.

Example
Adapting an Enumeration to an Iterator



Summary

- When you need to use an existing class and its interface is not the one you need, use an adapter.
- An adapter changes an interface into one a client expects.
- Implementing an adapter may require little work or a great deal of work depending on the size and complexity of the target interface.
- There are two forms of adapter patterns: object and class adapters. Class adapters require multiple inheritance.
- An adapter wraps an object to change its interface, a decorator wraps an object to add new behaviors and responsibilities.

The Façade Pattern

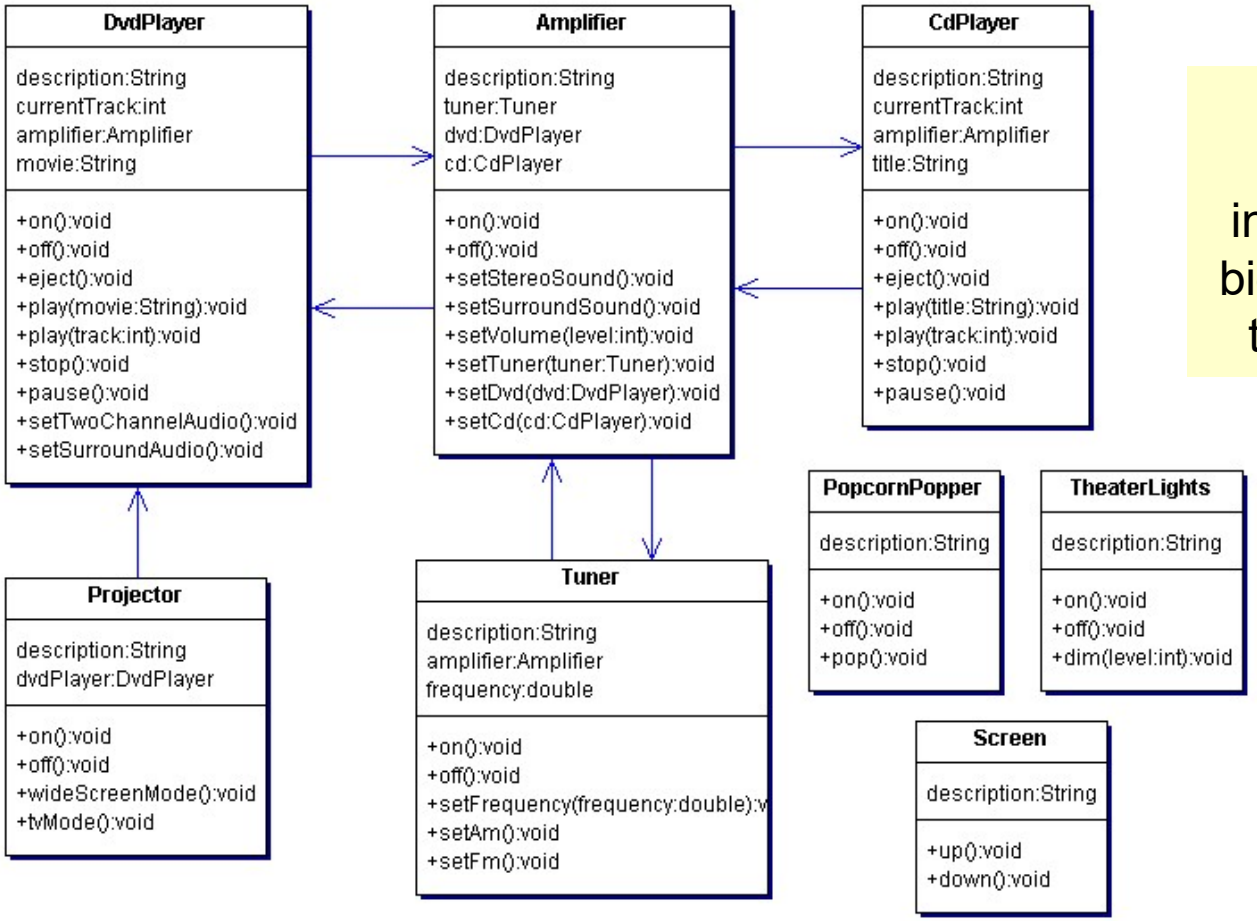
Simplify, simplify, simplify!

Façade

- Another pattern that wraps objects!
- For a different reason - to simplify the interface
- Aptly named as this pattern hides all the complexity of one or more classes behind a clean, well-lit façade!

Sweet Home Theater

- Building your own home theater - check out the components that you have/need to put together.



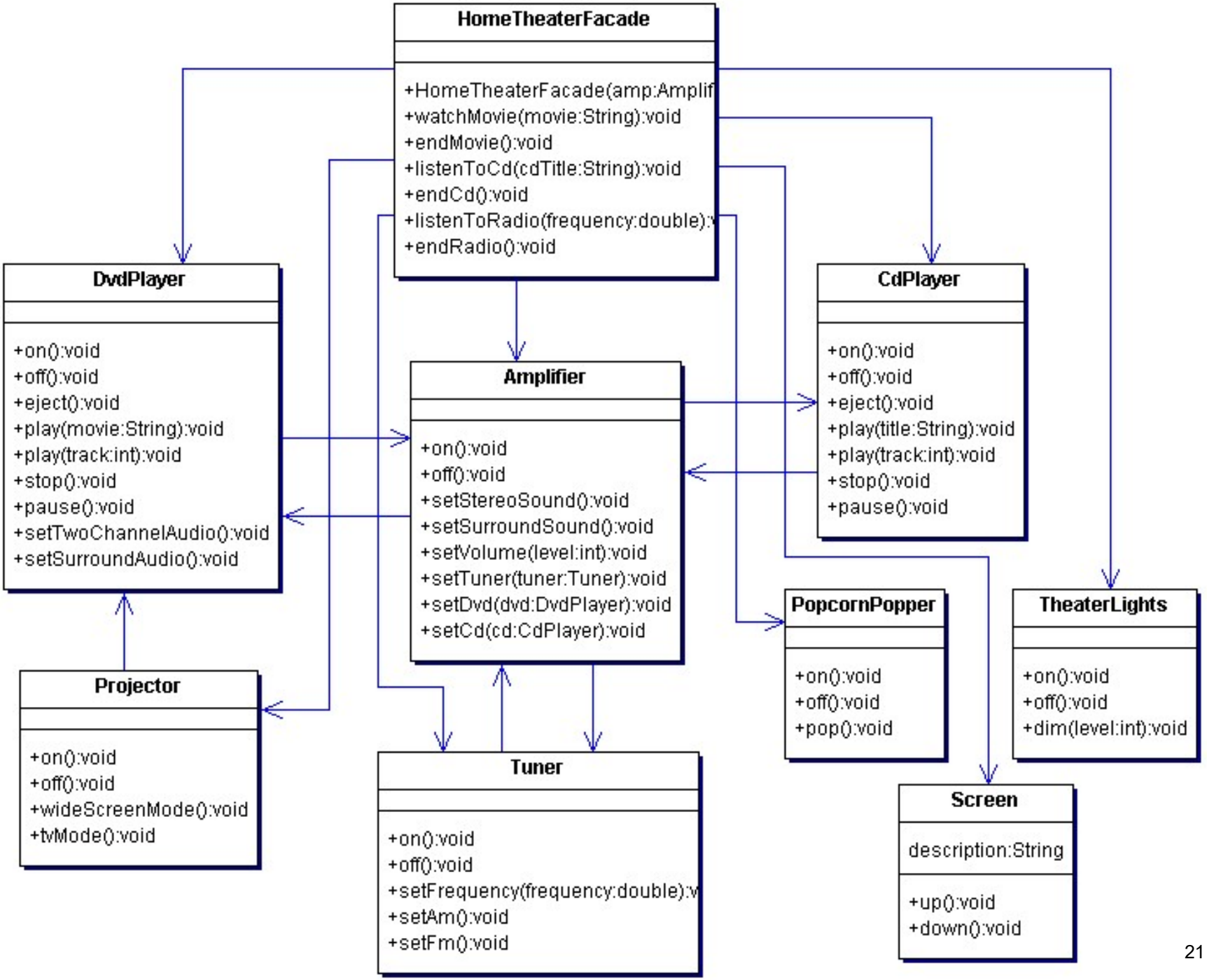
That's a lot of classes, a lot of interactions, and a big set of interfaces to learn and use.

Watching a Movie the Hard Way!

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 on 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
13. Start the DVD player playing.
14. Whew!

But there's more!
When the movie is done,
- How do you turn everything off?
- Do you reverse all the steps?

Façade to the Rescue!!


HomeTheaterFacade
+HomeTheaterFacade(amp:Amplif
+watchMovie(movie:String):void
+endMovie():void
+listenToCd(cdTitle:String):void
+endCd():void
+listenToRadio(frequency:double):
+endRadio():void
DvdPlayer
+on():void
+off():void
+eject():void
+play(movie:String):void
+play(track:int):void
+stop():void
+pause():void
+setTwoChannelAudio():void
+setSurroundAudio():void
CdPlayer
+on():void
+off():void
+eject():void
+play(title:String):void
+play(track:int):void
+stop():void
+pause():void
Amplifier
+on():void
+off():void
+setStereoSound():void
+setSurroundSound():void
+setVolume(level:int):void
+setTuner(tuner:Tuner):void
+setDvd(dvd:DvdPlayer):void
+setCd(cd:CdPlayer):void
PopcornPopper
+on():void
+off():void
+pop():void
TheaterLights
+on():void
+off():void
+dim(level:int):void
Projector
+on():void
+off():void
+wideScreenMode():void
+tvMode():void
Tuner
+on():void
+off():void
+setFrequency(frequency:double):
+setAm():void
+setFm():void
Screen
description:String
+up():void
+down():void

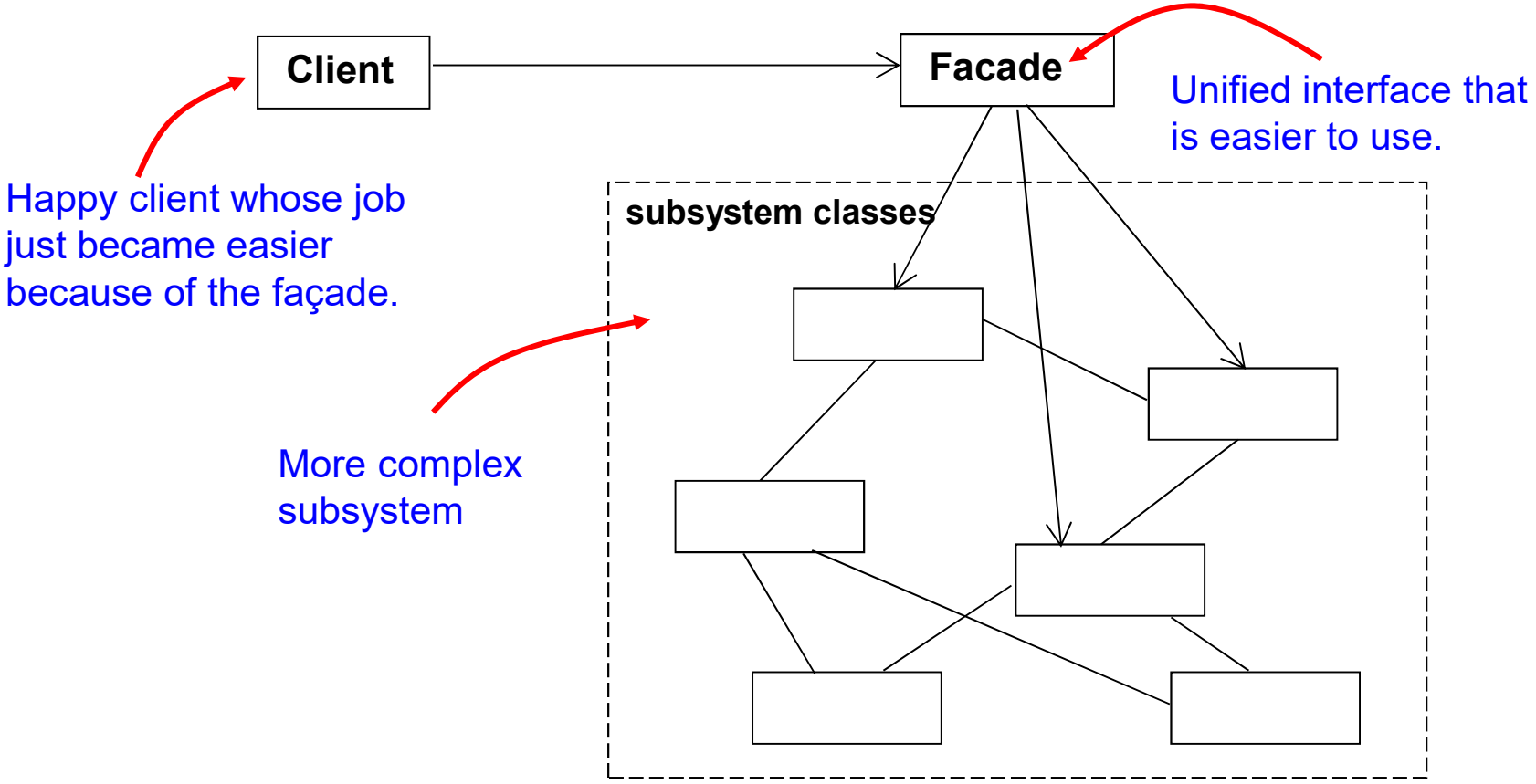
Example explain

- Create a **`Façade`** for the HomeTheater which exposes a few simple methods such as **`watchMovie()`**
- The **`Façade`** treats the home theater components as its subsystem, and calls on the subsystem to implement its **`watchMovie()`** method.
- The **`Client`** now calls methods on the façade and not on the subsystem.
- The **`Façade`** still leaves the subsystem accessible to be used directly.

⇨**`HomeTheaterFacade`** manages all those subsystem components for the client. It keeps the client simple and flexible.

The Facade Pattern – Key Features

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



The Facade Pattern – Key Features

- **Applicability**: Use the Facade pattern when
  - want to provide a simple interface to a complex subsystem.
  - decouple the subsystem from the dependencies of clients and other subsystems, thereby promoting subsystem independence and portability.
  - want to layer your subsystems. Use a facade to define an entry point to each subsystem level.
- **Consequences**: offers the following benefits:
  - shields clients from subsystem components,
  - promotes weak coupling between the subsystem and its clients.
  - help layer a system and the dependencies between objects. They can eliminate complex or circular dependencies.

A New Design Principle

- Principle of Least Knowledge –

  Talk only to your immediate friends!

- What does it mean?
  - When 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.
- 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 the 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

- The principle provides some guidelines - 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

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

a "component" is any object that is referenced by an instance variable (HAS_A relationship).

Without principle

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

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

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

With principle

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 numebr of classes we're dependent on.

Keeping your method calls in bounds….

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

   public Car() {
      // initialize enginer here
   }
   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 are creating a new object, its methods are legal.

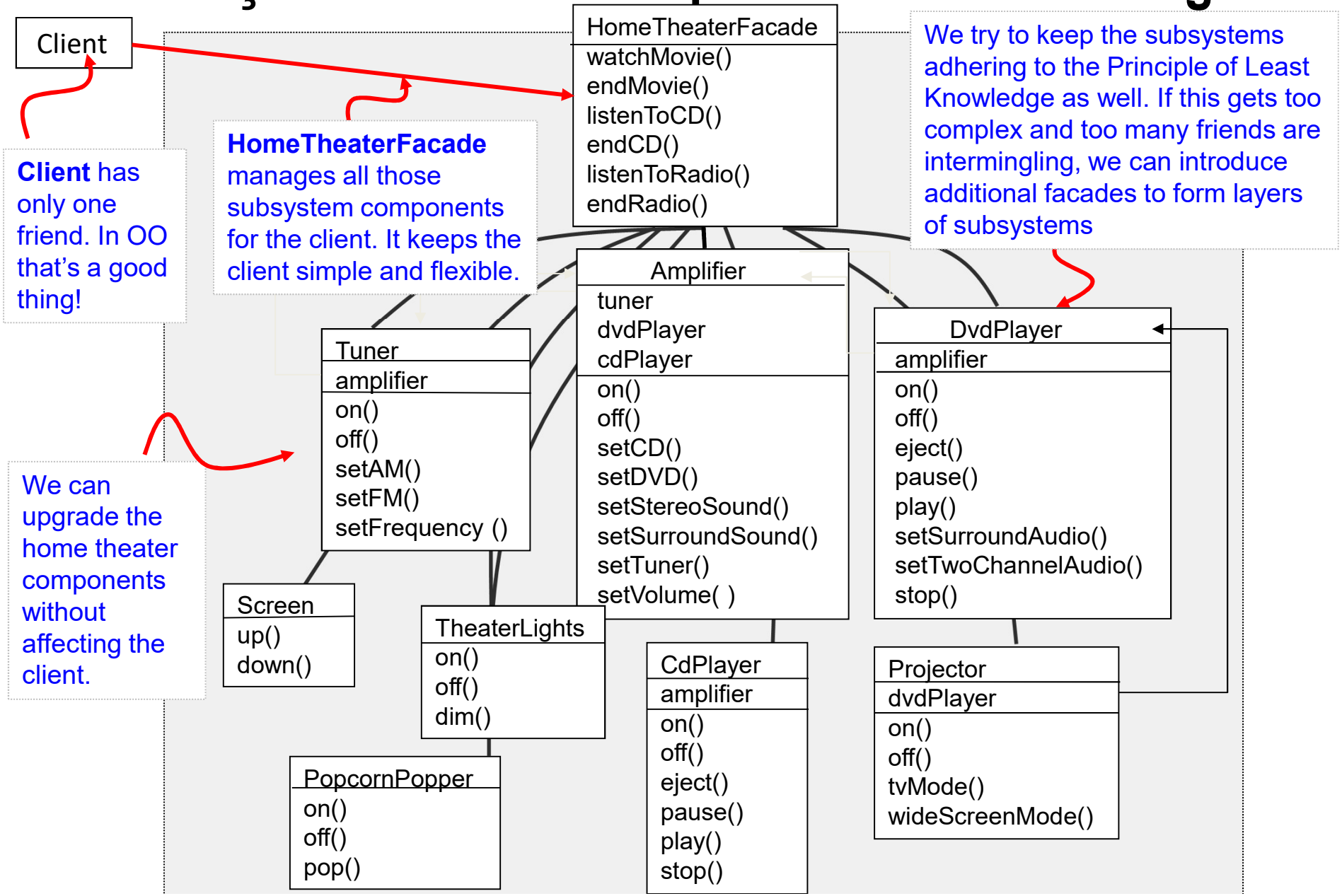
You can call a method on an object passed as a parameter

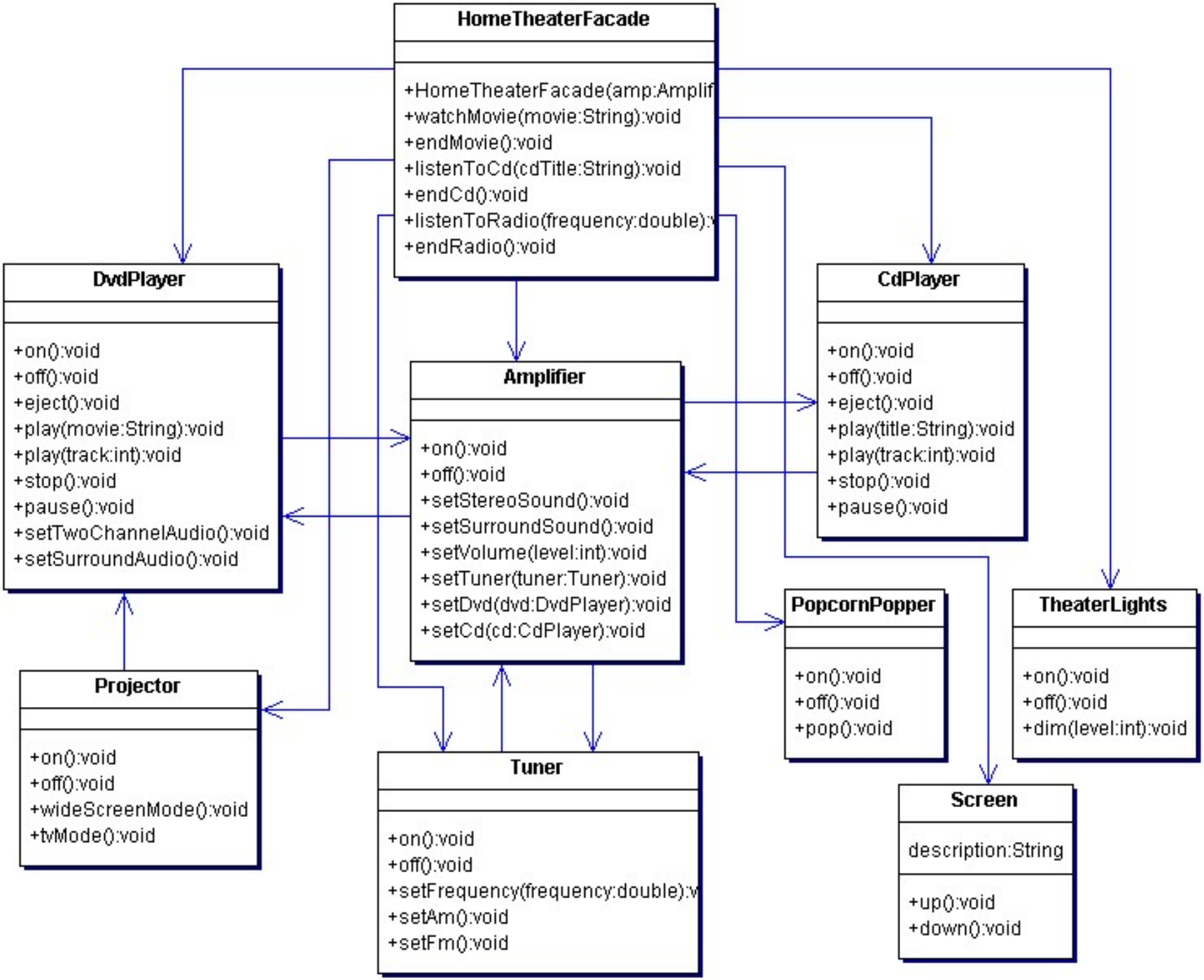
You can call a method on a component of the object.

You can call a local method within the object.

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

The Façade and the Principle of Least Knowledge




HomeTheaterFacade
+HomeTheaterFacade(amp:Amplif
+watchMovie(movie:String):void
+endMovie():void
+listenToCd(cdTitle:String):void
+endCd():void
+listenToRadio(frequency:double):
+endRadio():void
DvdPlayer
+on():void
+off():void
+eject():void
+play(movie:String):void
+play(track:int):void
+stop():void
+pause():void
+setTwoChannelAudio():void
+setSurroundAudio():void
Amplifier
+on():void
+off():void
+setStereoSound():void
+setSurroundSound():void
+setVolume(level:int):void
+setTuner(tuner:Tuner):void
+setDvd(dvd:DvdPlayer):void
+setCd(cd:CdPlayer):void
CdPlayer
+on():void
+off():void
+eject():void
+play(title:String):void
+play(track:int):void
+stop():void
+pause():void
PopcornPopper
+on():void
+off():void
+pop():void
TheaterLights
+on():void
+off():void
+dim(level:int):void
Projector
+on():void
+off():void
+wideScreenMode():void
+tvMode():void
Tuner
+on():void
+off():void
+setFrequency(frequency:double):v
+setAm():void
+setFm():void
Screen
description:String
+up():void
+down():void

Summary

- When you need to simplify and unify a large interface or a complex set of interfaces, use a façade.
- A façade decouples the client from a complex subsystem.
- Implementing a façade requires that we compose the façade with its subsystem and use delegation to perform the work of the façade.
- You can implement more than one façade for a subsystem.
- A façade “wraps” a set of objects to simplify!