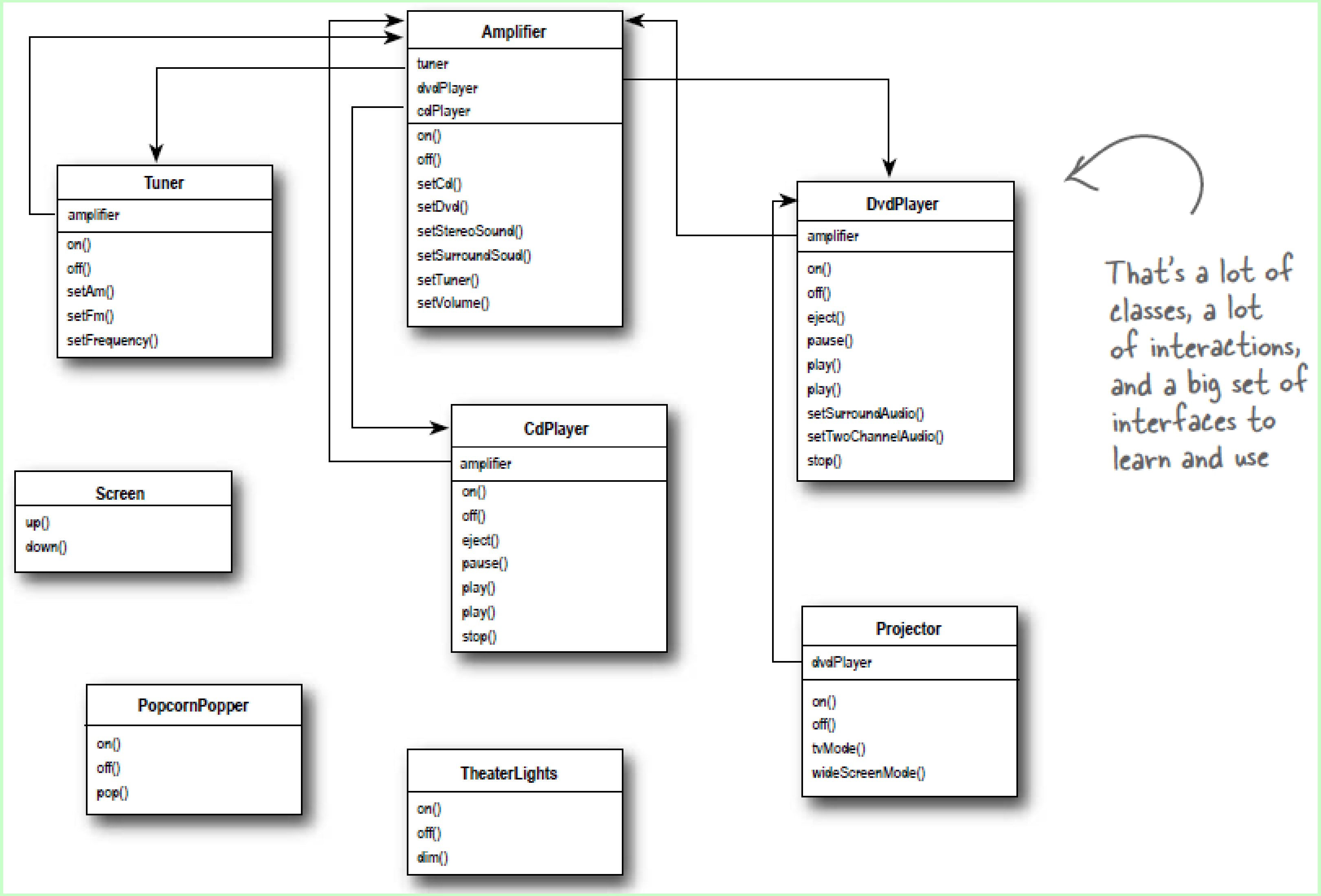
Facade Pattern

Home Theater

- You’ ve done your research and you’ ve assembled a killer system complete with
  - a DVD player,
  - a projection video system,
  - an automated screen,
  - surround sound and
  - even a popcorn popper.

Home Theater

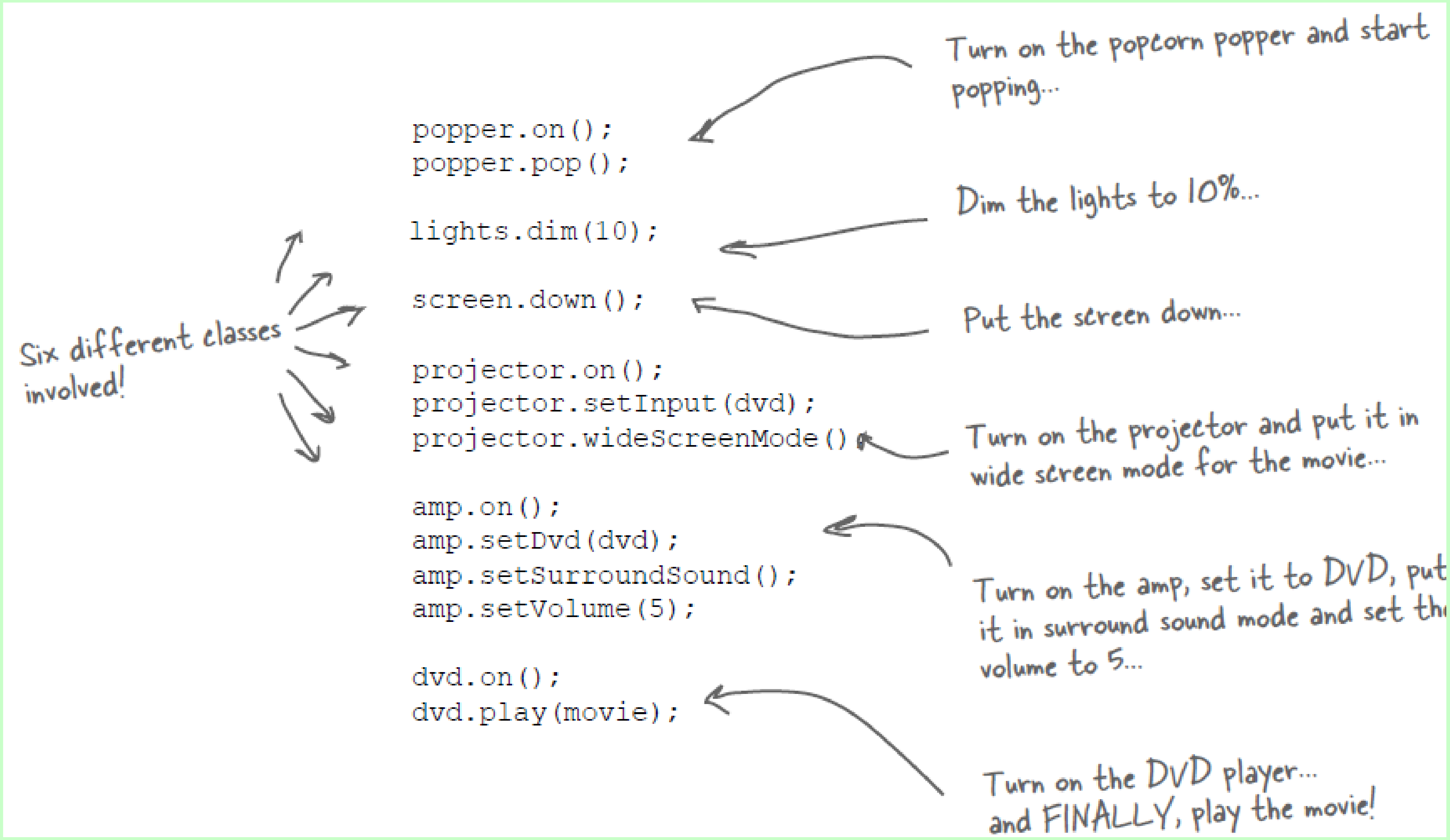


Watching a movie

- Pick out a DVD, relax, and get ready for movie magic.
- There’s just one thing – to watch the movie, you need to perform a few tasks:

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

In terms of OO Programming



```
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 th volume to 5...

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

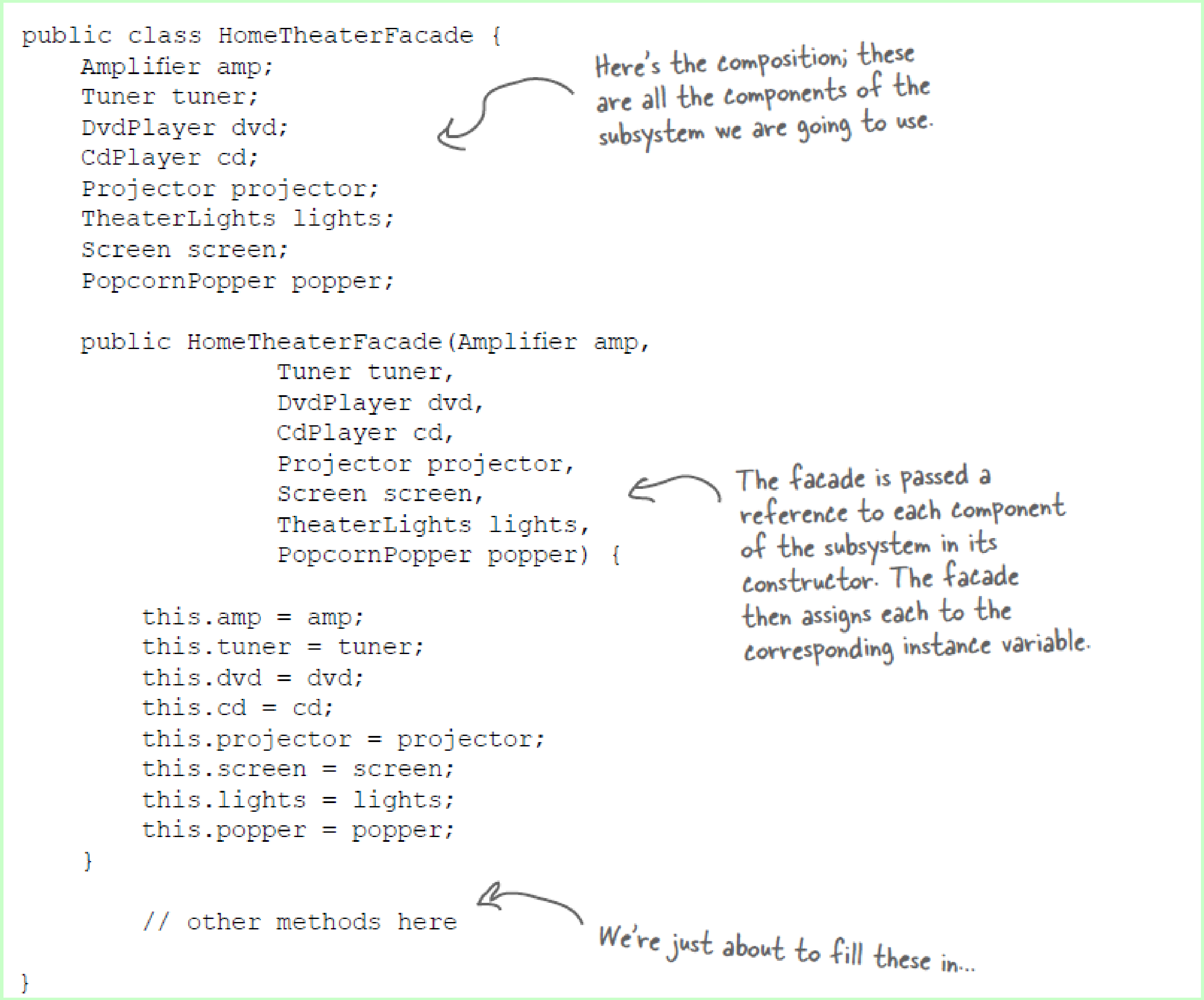
But there’s more...

- 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.

Facade Pattern

- 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.
- To do this 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.

Facade Pattern



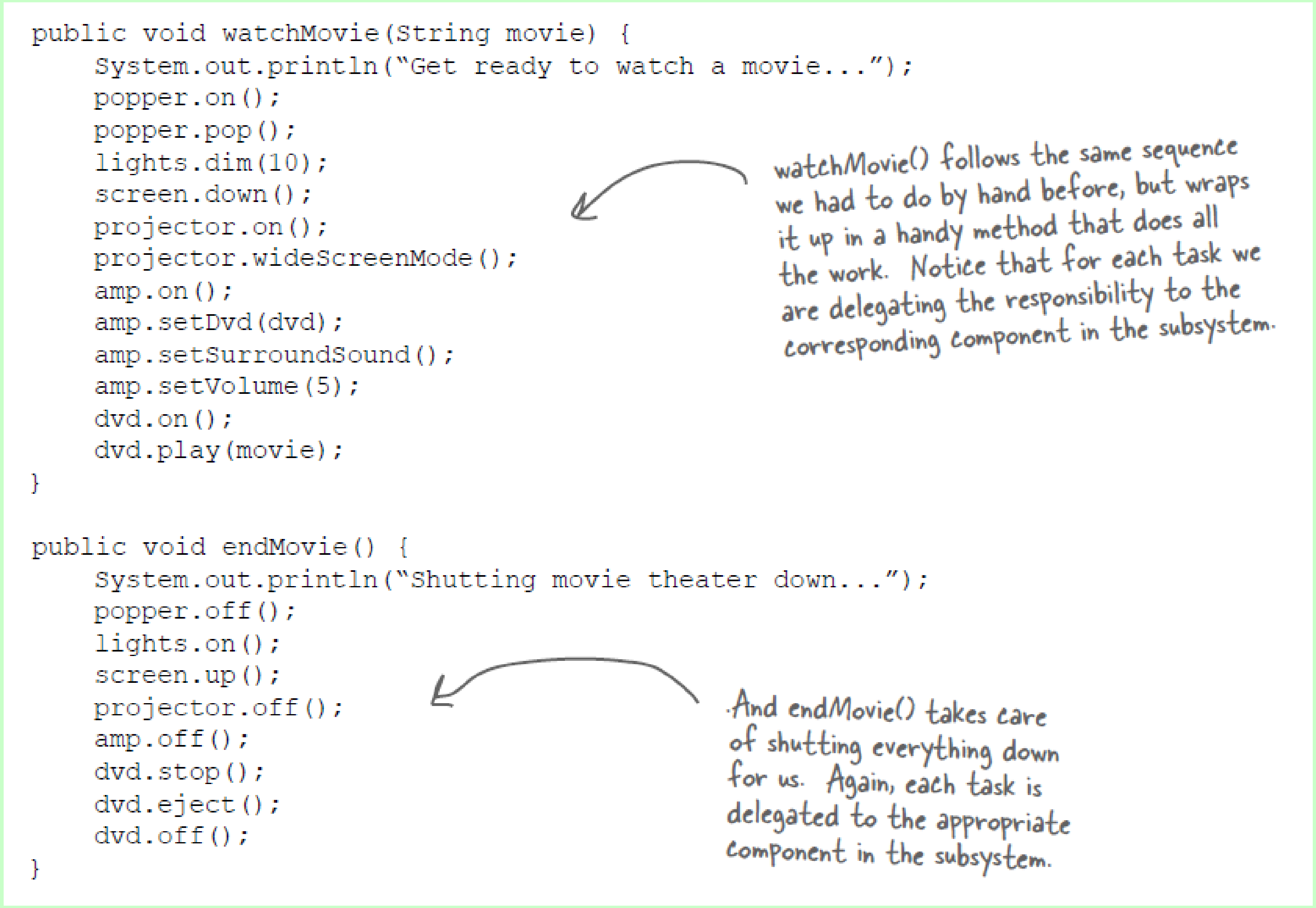
```
public class HomeTheaterFacade {
    Amplifier amp;
    Tuner tuner;
    DvdPlayer dvd;
    CdPlayer cd;
    Projector projector;
    TheaterLights lights;
    Screen screen;
    PopcornPopper popper;

    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
}
```

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

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...

Facade Pattern



```
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);
}

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();
}
```

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.

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

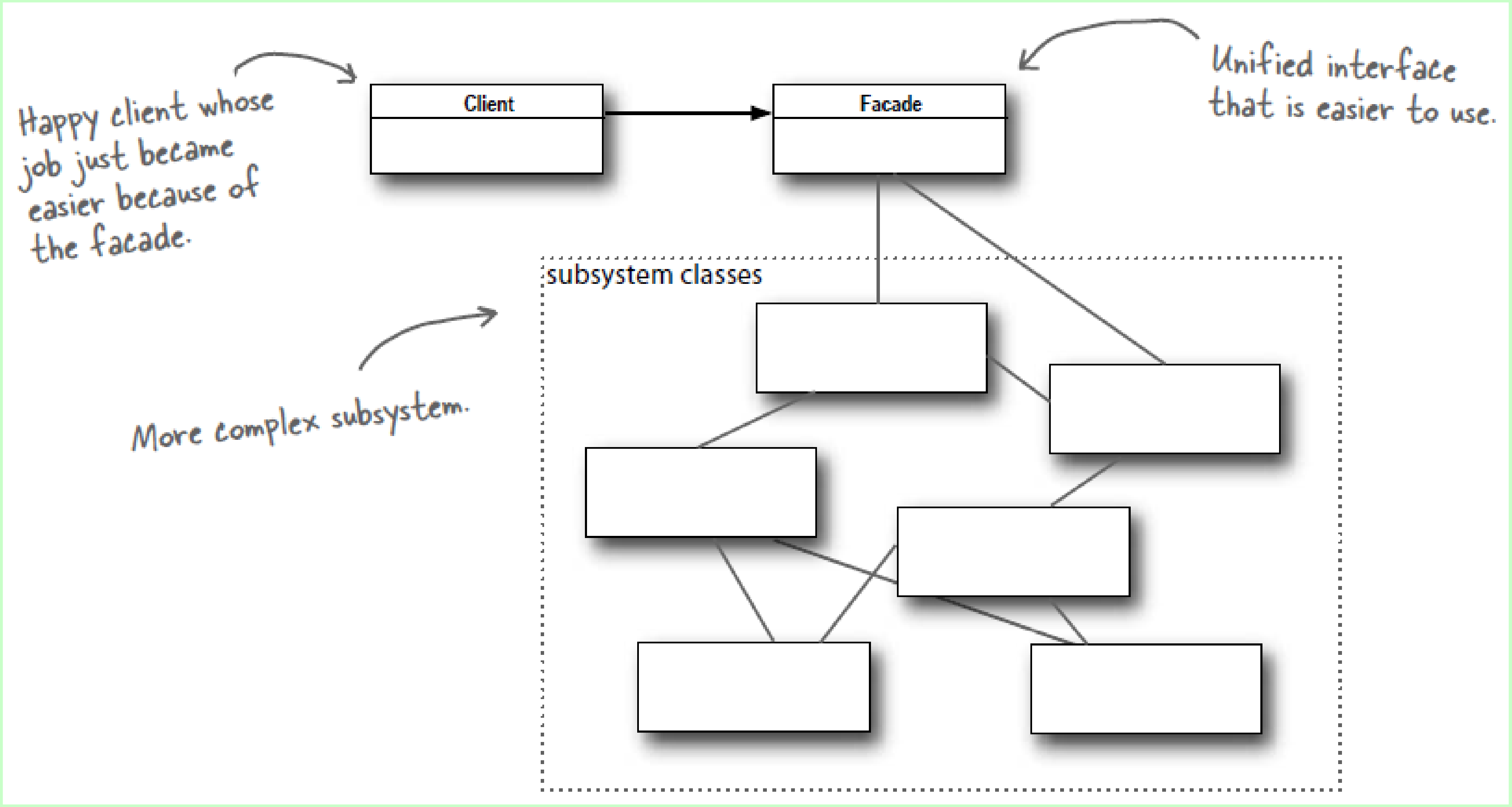
TestDrive

```
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();
    }
}
```

Facade Pattern

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

Class Diagram


Happy client whose job just became easier because of the facade.
Client
Facade
Unified interface that is easier to use.
subsystem classes
More complex subsystem.

The Principle of Least Knowledge

- **Design Principle** : *Principle of Least Knowledge* - talk only to your immediate friends.

- The Principle of Least Knowledge guides us to reduce the interactions between objects to just a few close “friends.”

The Principle of Least Knowledge

- 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.

- 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.

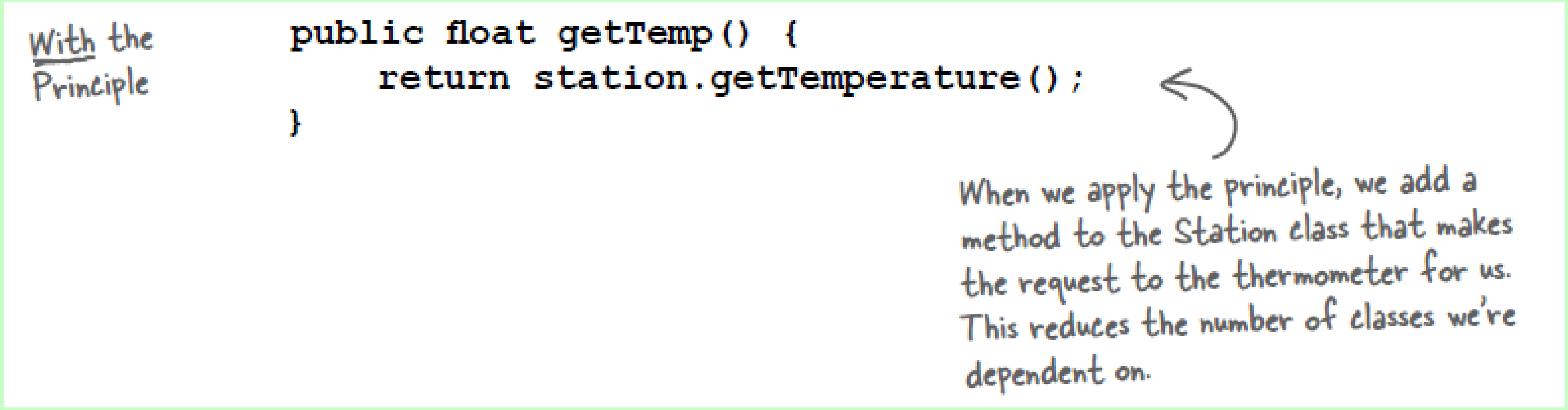
How many classes is this code coupled to?

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

is equivalent to

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

Guidelines for the Principle

- take any class; 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

Applying the Principle

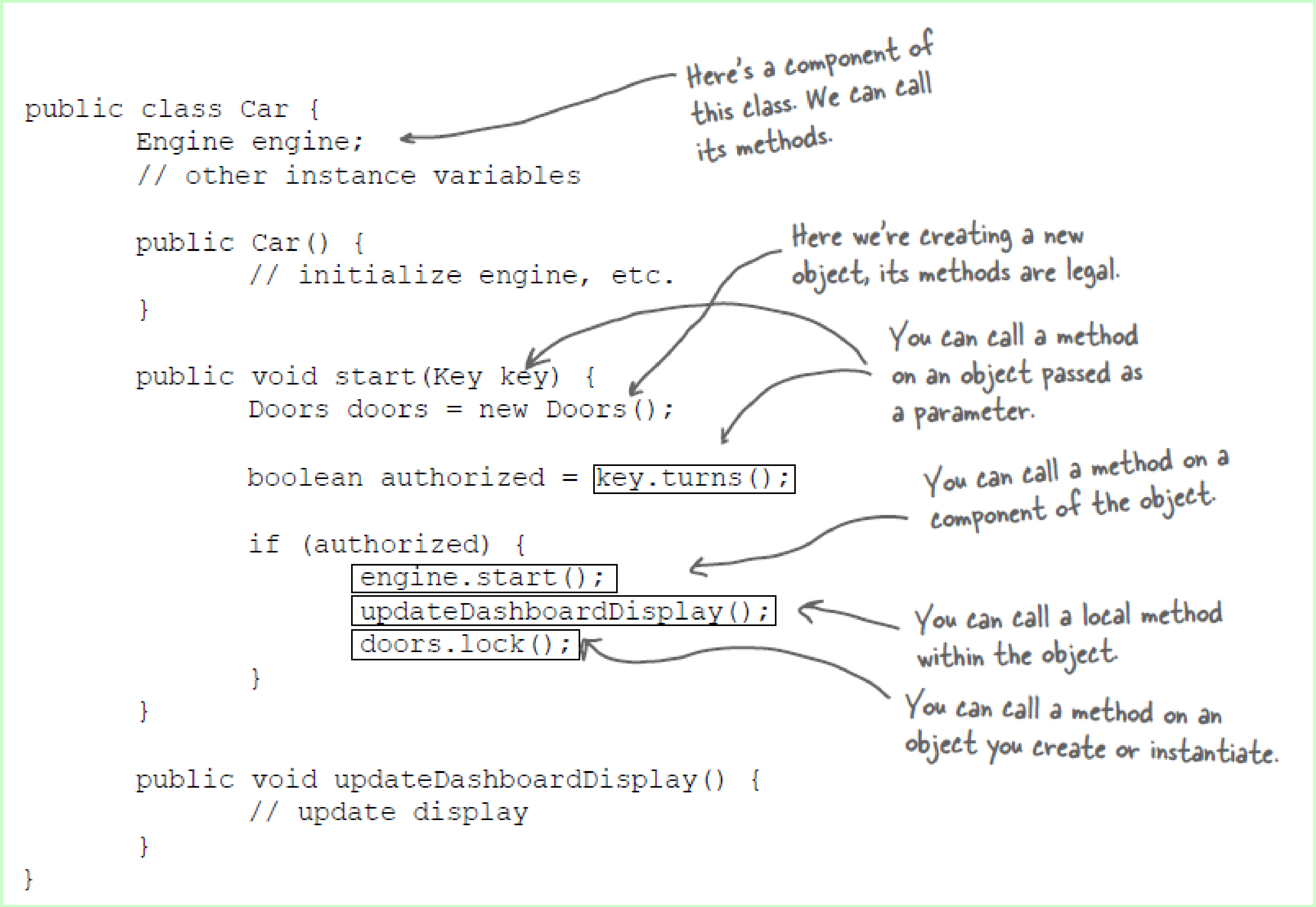


With the Principle

```
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.

Applying the Principle



```
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.

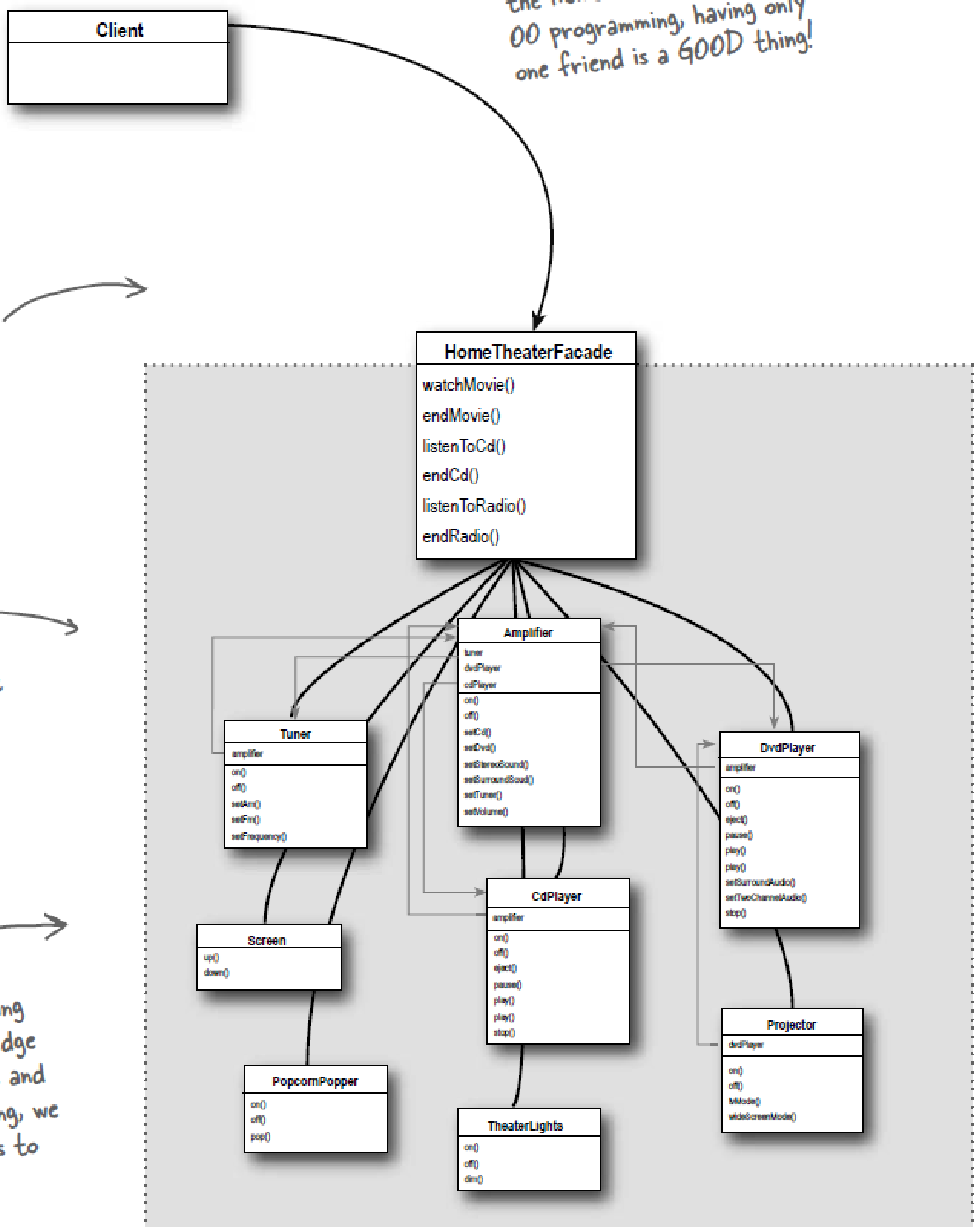
You can call a method on an object passed as a parameter.

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.

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.

References

- Design Patterns: Elements of Reusable Object-Oriented Software By Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). Addison-Wesley. ISBN 0-201-63361-2.
- **Head First Design Patterns** By Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates
  First Edition October 2004
  ISBN 10: 0-596-00712-4