The Façade Pattern

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.

Toni Sellarès
Universitat de Girona

Façade: Motivation

A subsystem could consist of a large number of classes.

Clients of a subsystem may need to interact with a number of subsystem classes for their needs.

This leads to a high degree of coupling between the client objects and the subsystem.

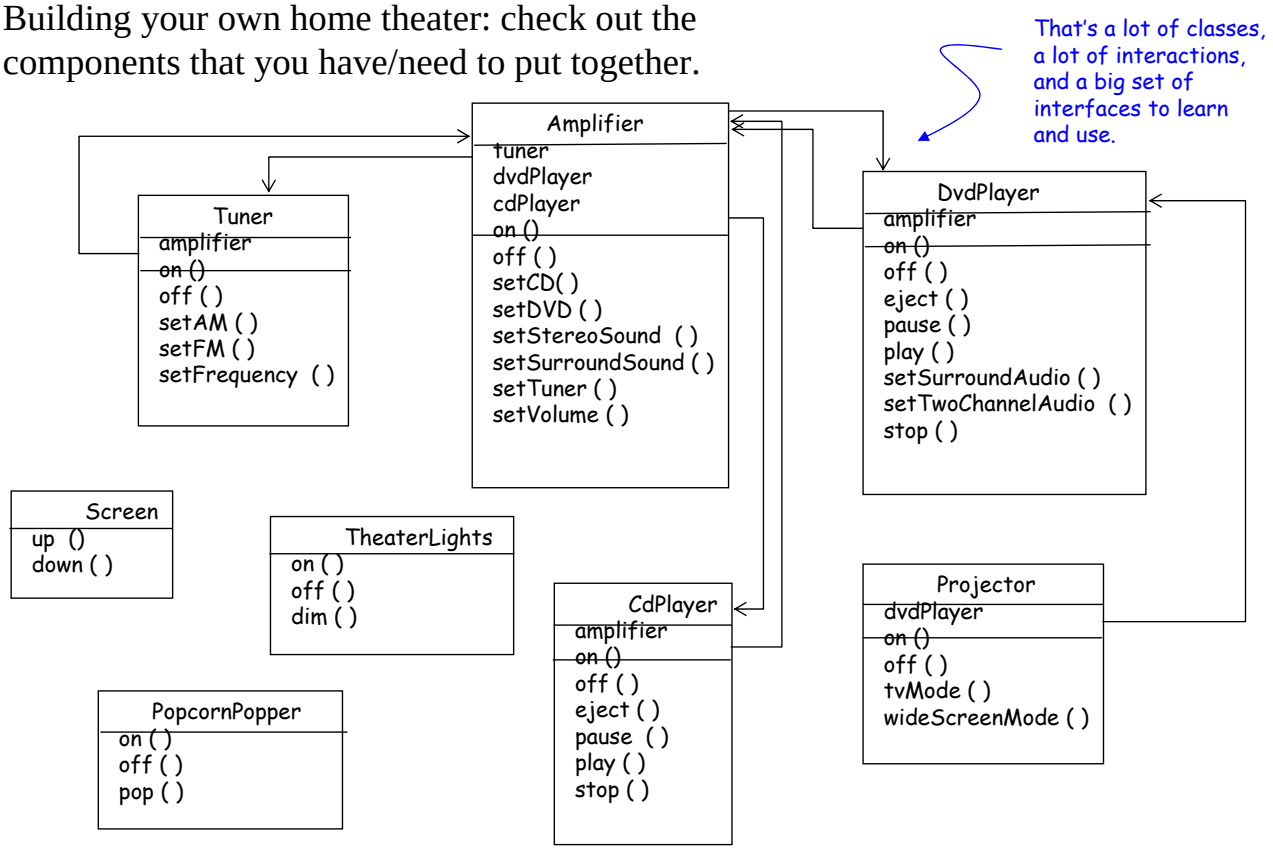
The Façade pattern provides a higher level, simplified interface for a subsystem resulting in reduced complexity and dependency.

Clients interact with the Façade object to deal with the subsystem instead of interacting directly with subsystem classes.

Even though clients interact with the Façade, when needed, a client will be able to access subsystem components directly through the lower level interfaces of the subsystem.

Façade: Example - Home Sweet Home Theater

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



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?

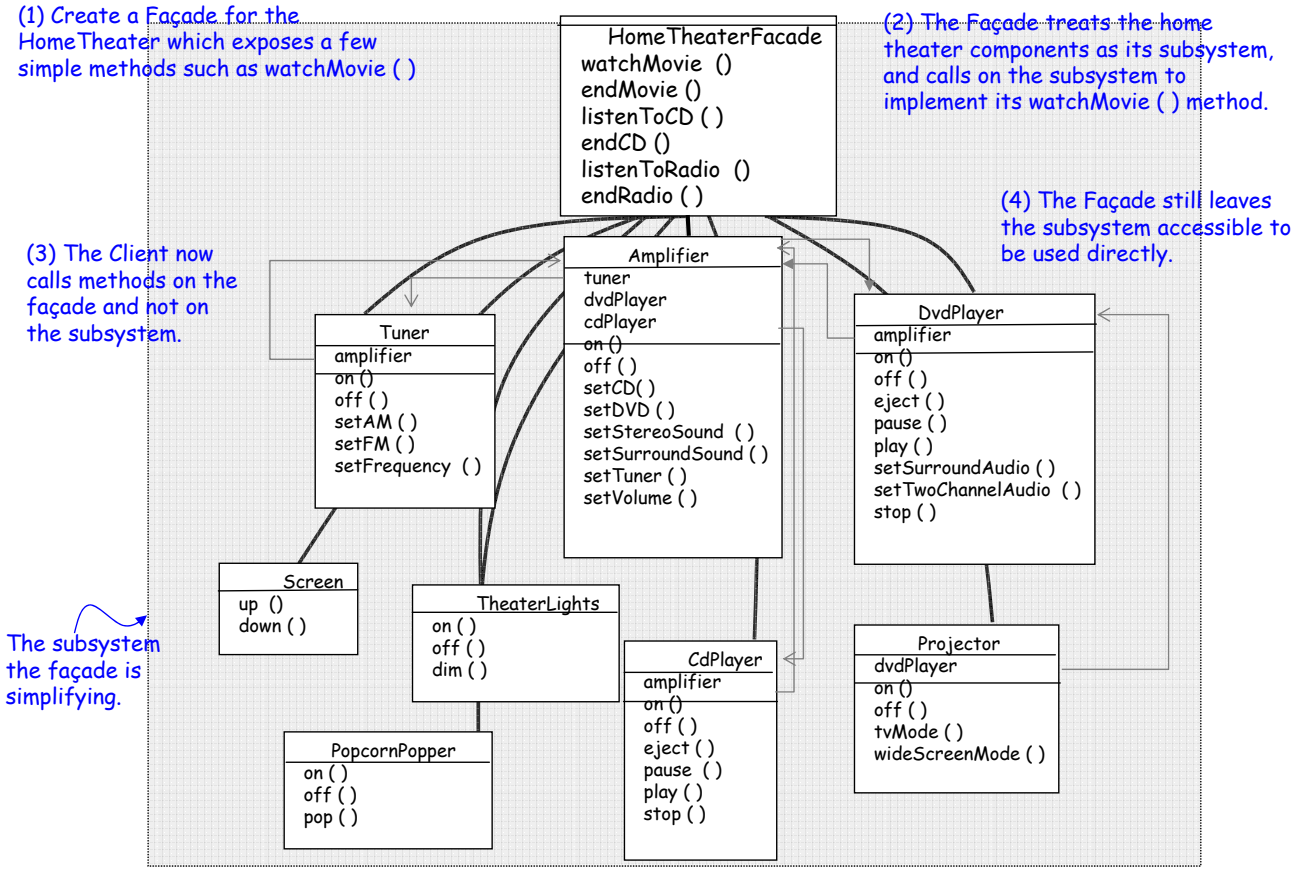
Do you reverse all the steps?

Wouldn't it be just as complex to listen to a CD or radio?

If you decide to upgrade your system, you're probably going to have to learn a slightly different procedure!

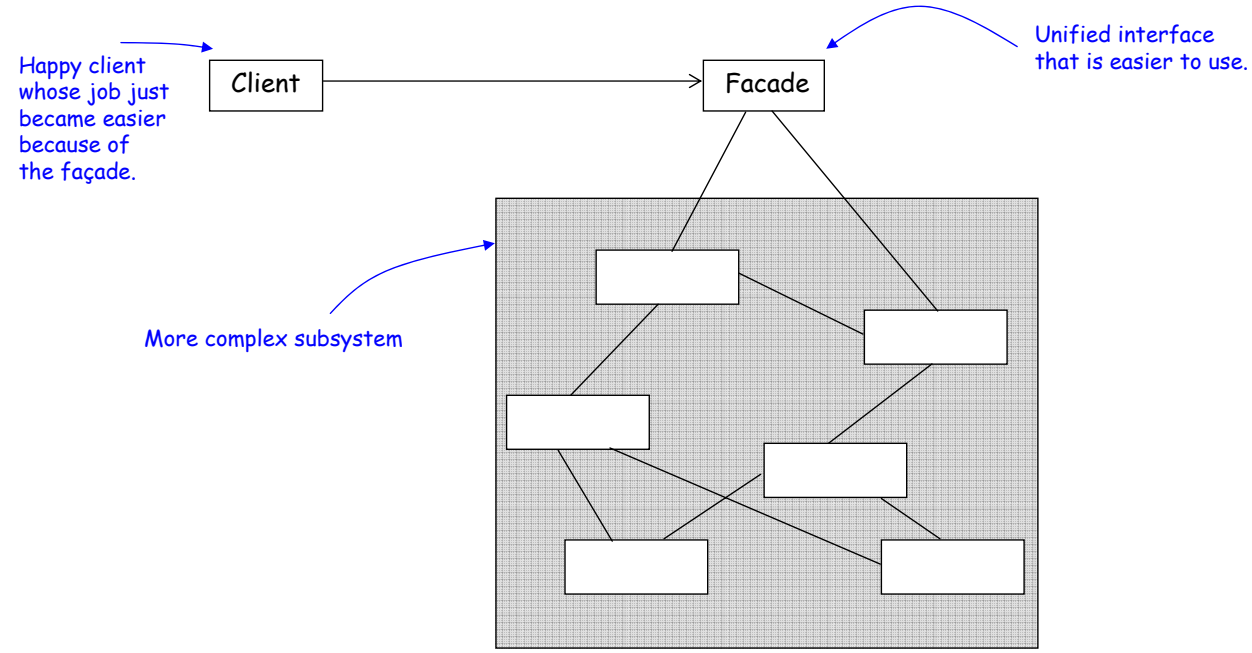
Façade to the Rescue!!

Façade!



The Façade Pattern Defined

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 Façade Pattern: Structural Code

```
/**
 * Test driver for the pattern.
 */
public class Test {
    public static void main( String arg[] ) {
         Facade facade = new Facade();
         facade.go();
         }
}

/**
 * Implement subsystem functionality. Handle work assigned by the Facade object.
 * Have no knowledge of the facade; that is, they keep no references to it.
 */
public class Subsystem1 {
         public void doWork()
         }
}
```

```
public class Subsystem2 {
          public void doWork()
          }
}


public class Subsystem3 {
          public void doWork()
          }
}



/**
 * Knows which subsystem classes are responsible for a request.
 * Delegates clients requests to appropriate subsystem objects.
 */
public class Facade {
          private Subsystem1 sub1 = new Subsystem1();
          private Subsystem2 sub2 = new Subsystem2();
          private Subsystem3 sub3 = new Subsystem3();
          public void go() {
                    sub1.doWork();
                    sub2.doWork();
                    sub3.doWork();
          }
}
```

Least Knowledge Design Principle (LKDP)

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!

LKDP

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

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

Without principle

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

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

With principle

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.

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

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'are dependent on.

```
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.

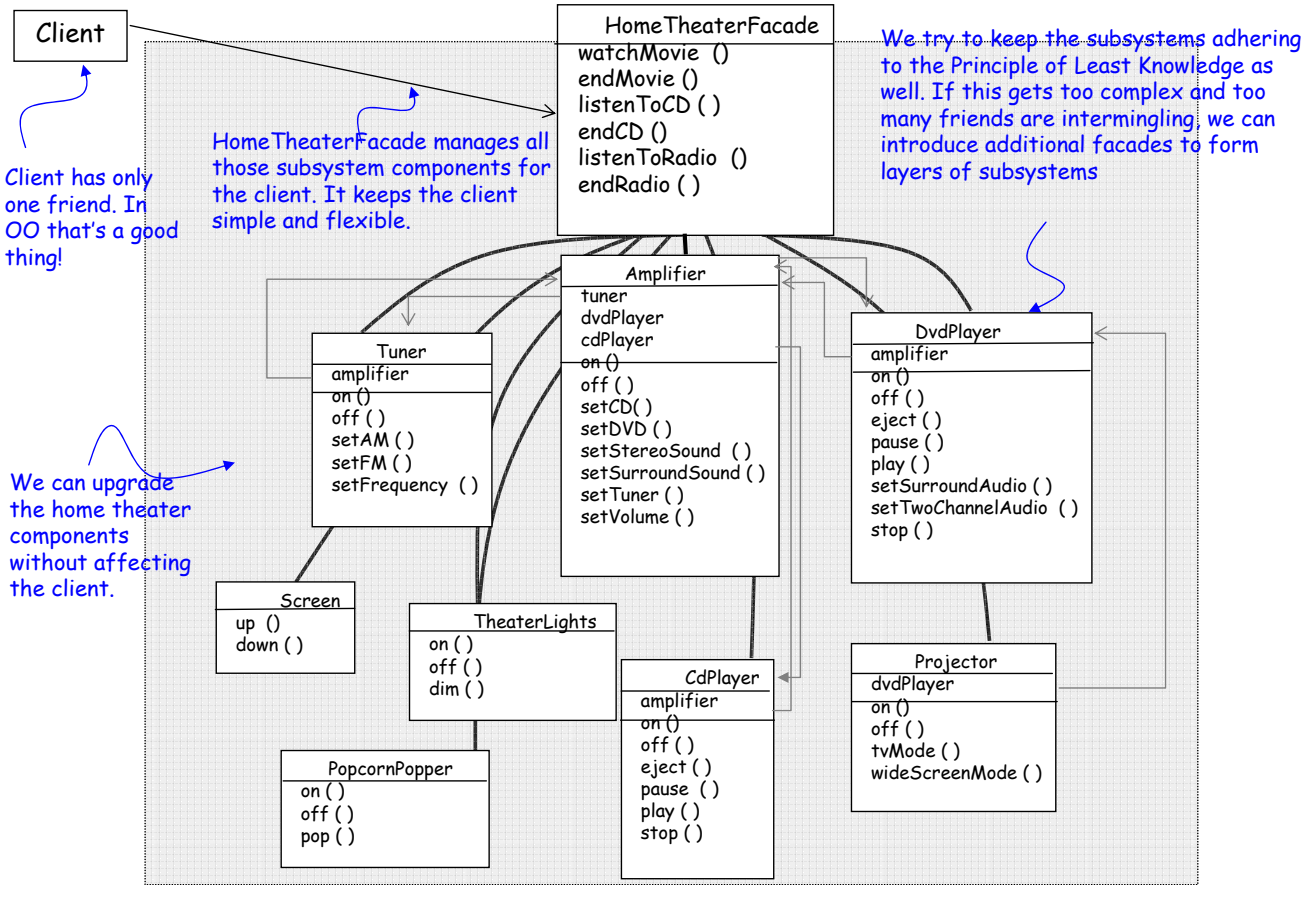
You can call a method on an object passed as a parameter.

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



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!