Facade Pattern (门面, Structural Pattern)

Kai SHI

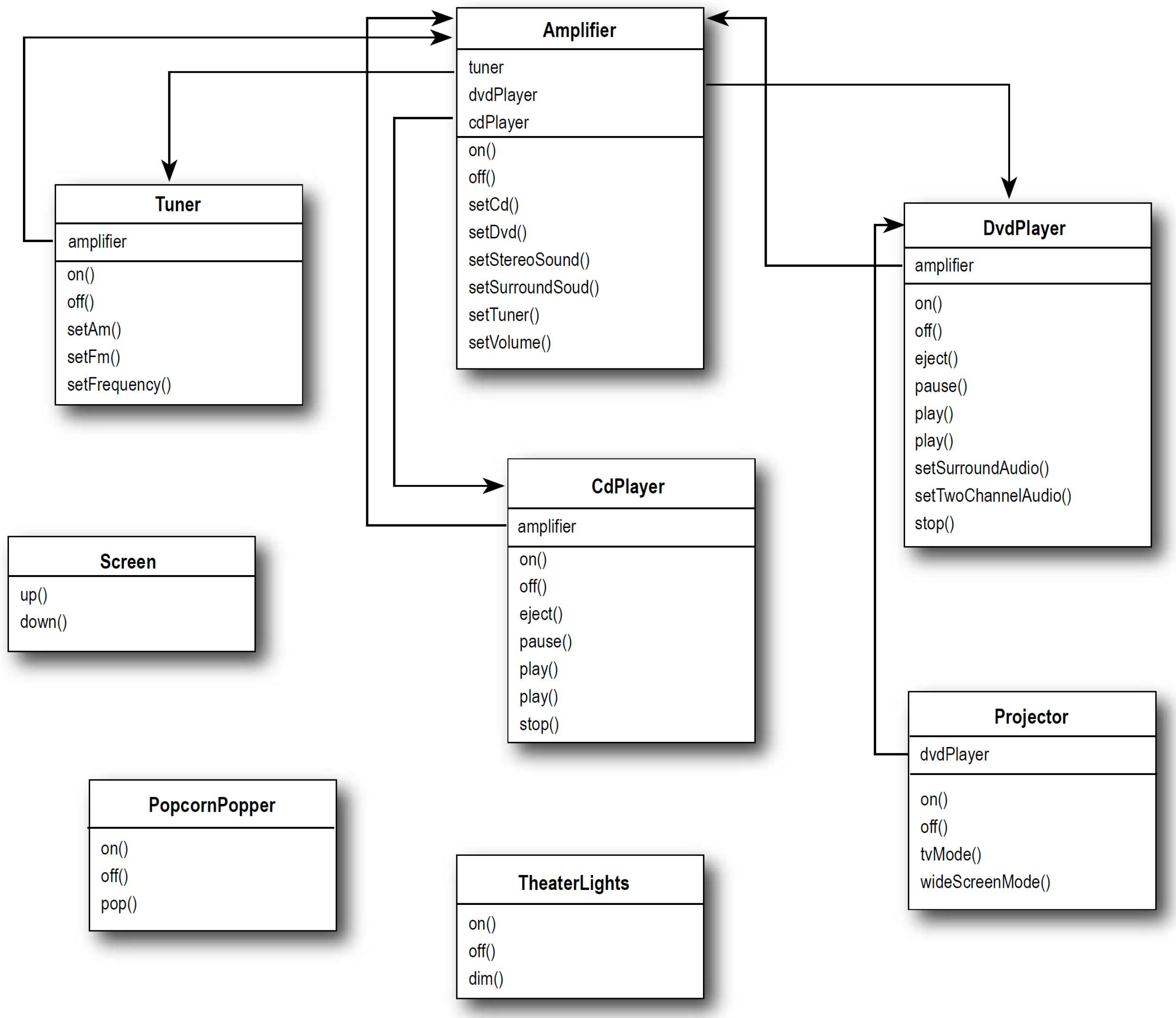
Too Many Interfaces, Too Complex. It Breaks Law of Demeter

You Just Want A Unified Interface

✓

✗

Problem: Home Theater (1/3)



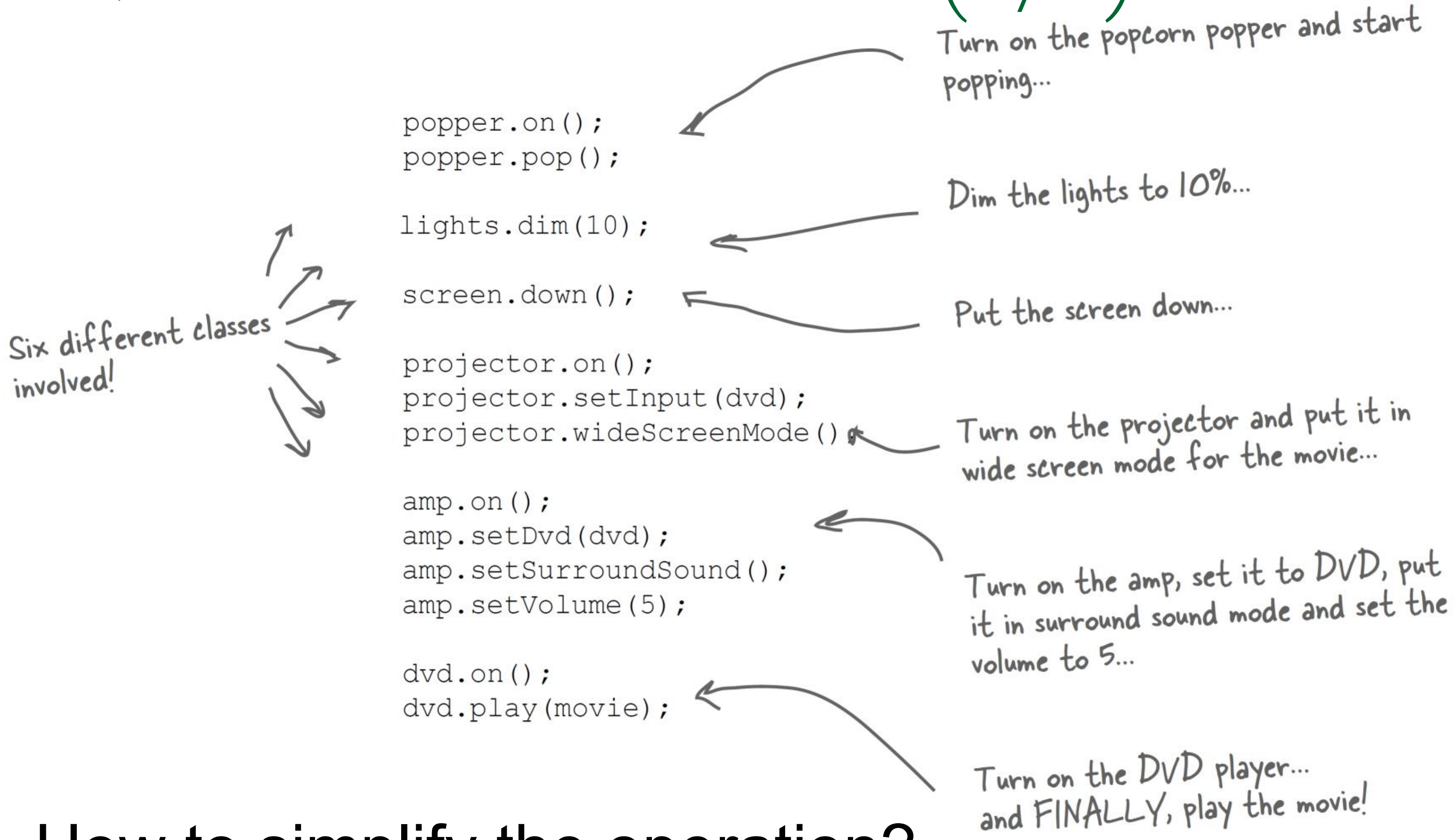
That's a lot of classes, a lot of interactions, and a big set of interfaces to learn and use

Problem: Home Theater (2/3) Watching a movie (the hard way)

- Turn on the popcorn popper
- Start the popper popping
- Dim the lights
- Put the screen down
- Turn the projector on
- Set the projector input to DVD
- 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
- Turn the DVD Player on
- Start the DVD Player playing

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

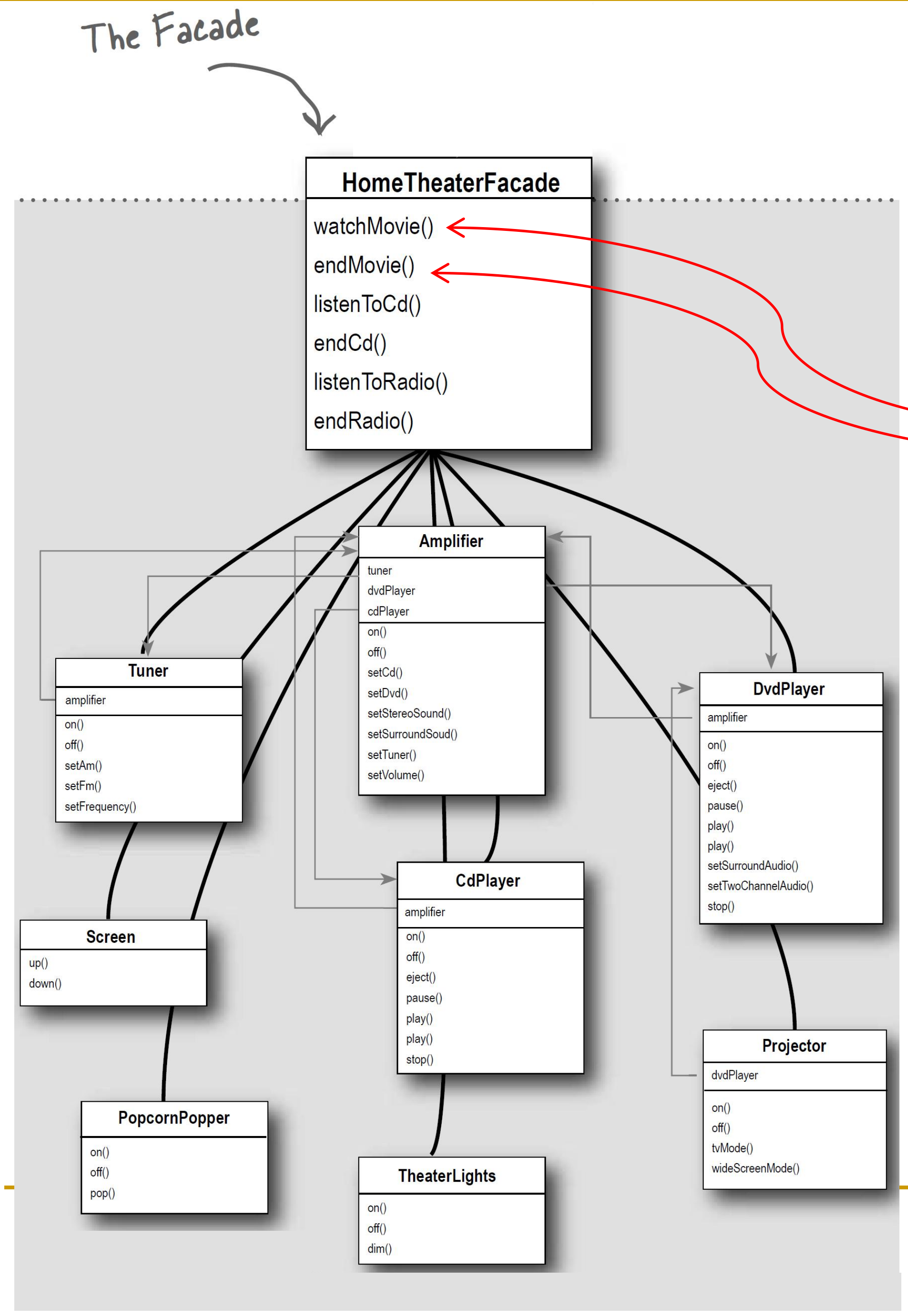
Problem: Home Theater (3/3)

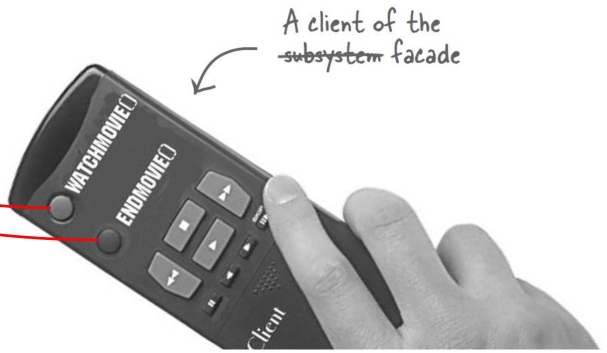


```
popper.on();
popper.pop();

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!

- How to simplify the operation?
- Draw class diagram.





The Facade still leaves the subsystem accessible to be used directly.

Home Theater Facade (1/3)

```
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...

Home Theater Facade (2/3)

```
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.

Home Theater Facade (3/3)

```
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.

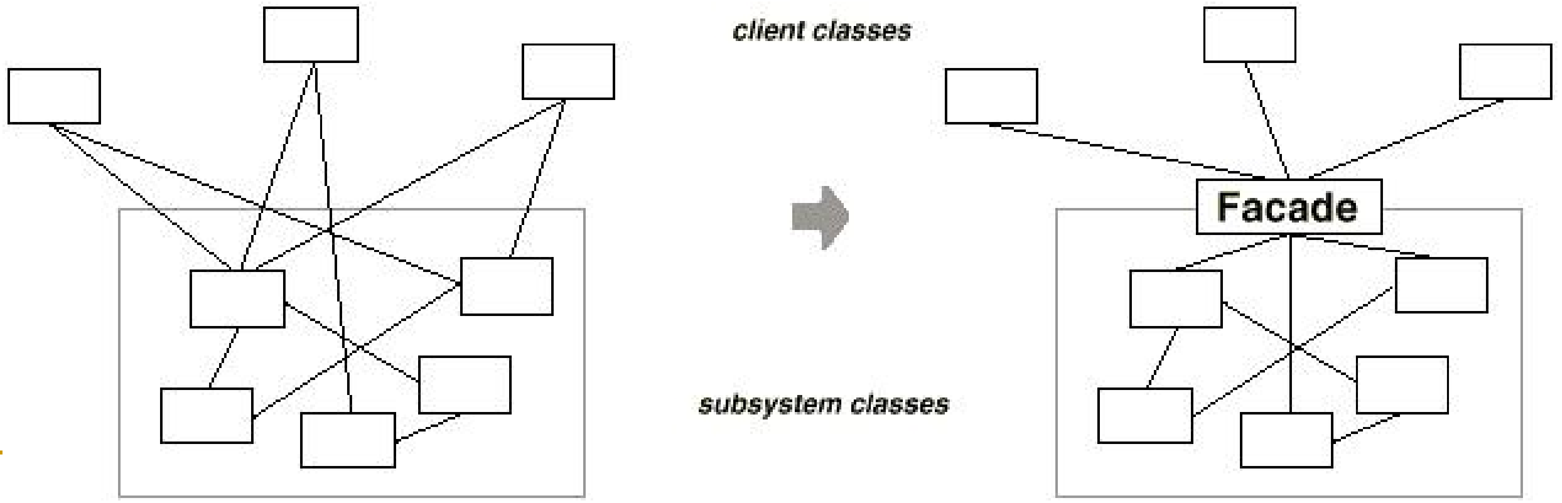
Code: net.dp.facade.hometheater.HomeTheaterTestDrive

Facade Pattern

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

Motivation

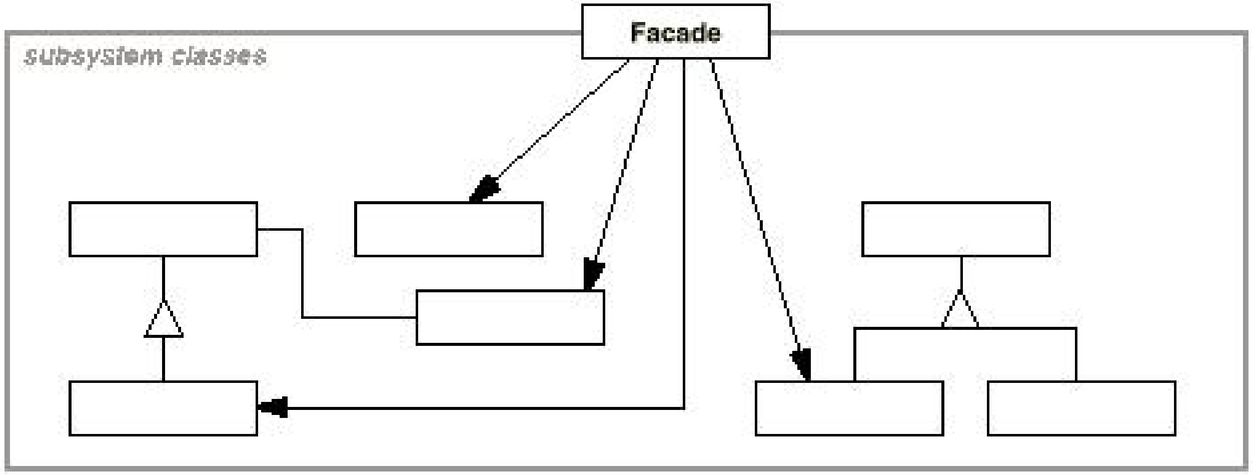
- Structuring a system into subsystems helps reduce complexity.
- A common design goal is to minimize the communication and dependencies between subsystems.
- One way to achieve this goal is to introduce a facade object that provides a single, simplified interface to the more general facilities of a subsystem.



Applicability:
Use the Facade pattern when

- you want to provide a simple interface to a complex subsystem.
- there are many dependencies between clients and the implementation classes of an abstraction. Introduce a facade to decouple the subsystem from clients and other subsystems, thereby promoting subsystem independence and portability.
- you want to layer your subsystems. Use a facade to define an entry point to each subsystem level.

Structure



Participants

- Facade
  - Knows which subsystem classes are responsible for a request.
  - Delegates client requests to appropriate subsystem objects.
- Subsystem classes
  - Implement subsystem functionality.
  - Handle work assigned by the Facade object.
  - Have no knowledge of the facade; that is, they keep no references to it.

Consequences

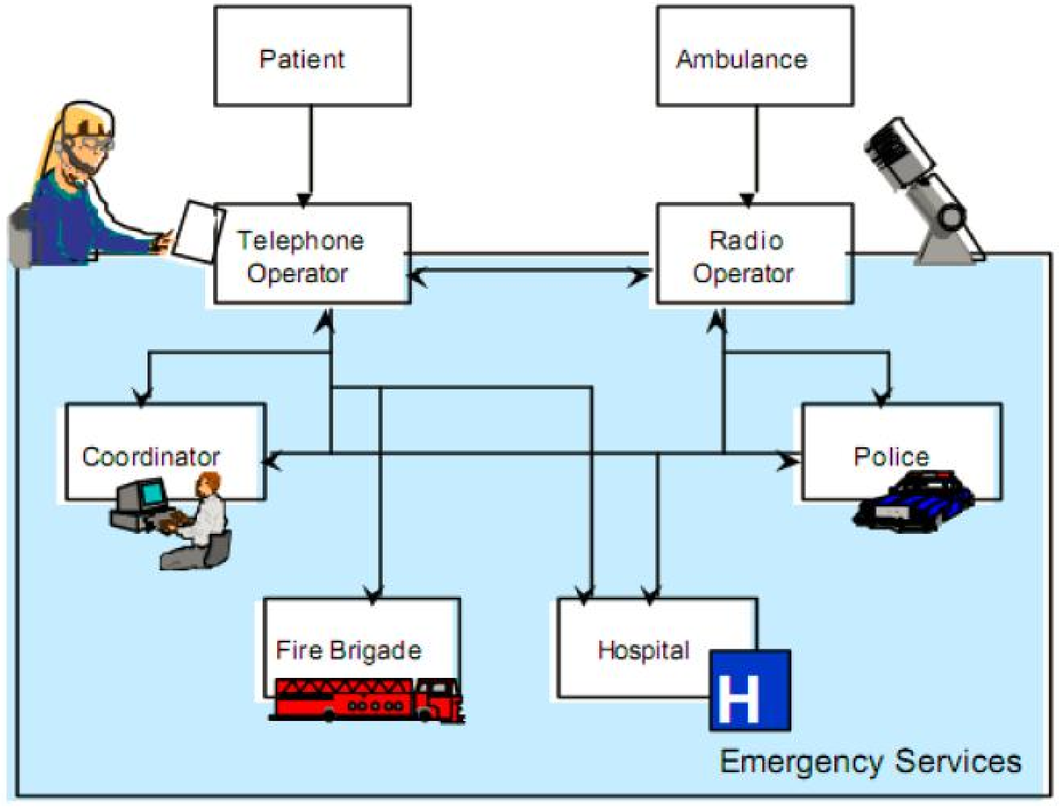
- It shields clients from subsystem components.
- It promotes weak coupling between the subsystem and its clients.
- It doesn't prevent applications from using subsystem classes if they need to.

Design Principle

- Principle of Least Knowledge (Law of Demeter)
  - talk only to your immediate friends.

Implementation Issue: Multiple Facades

- In general, a subsystem only need one facade. But in some situation, multiple facades should be considered.



Implementation Issue: No new behaviors introduced in Facades

- **DO NOT** introduce new behaviors to the facade.
  - If such behaviors is implemented by subsystem, then extend the facade.
  - If such behaviors is not implemented by subsystem yet, then extend the subsystem and facade both.

Implementation Issue: Reducing client-subsystem coupling

- Making Facade an abstract class with concrete subclasses for different implementations of a subsystem. This abstract coupling keeps clients from knowing which implementation of a subsystem is used. (使clients与具体子系统解耦)
- An alternative to subclassing is to **configure** a Facade object with different subsystem objects. To customize the facade, simply replace one or more of its subsystem objects.

Implementation Issue: Public versus private subsystem classes

- A subsystem is analogous to a class in that both have interfaces (NOT Java interface here), and both encapsulate something
  - A class encapsulates state and operation
  - A subsystem encapsulates classes.
- Think about the public and private interface of a class,
- Think about the public and private interface of a subsystem.
  - The public interface to a subsystem consists of classes that all clients can access;
  - The private interface is just for subsystem extenders.
- The Facade class is part of the public interface.

Summary (1/2)

- From the client's point of view, the facade model not only simplifies the interface of the entire component system, but also achieves a "decoupling" effect to a certain extent for internal components and external client programs. Any changes in the subsystems will **not affect** the changes of the Facade interface.
- Facade model pays more attention to the entire system from the architectural level, rather than the level of a single class. Facade is more of an architectural design pattern.

Summary (2/2)

- The corresponding components of the facade should be a series of components with coupling relationship, rather than a collection of simple functions.
- Facade focuses on simplifying interfaces; Adapter focuses on converting interfaces; Decorator focuses on extending functions for objects on the premise of stable interfaces.