Developer Exchange Group

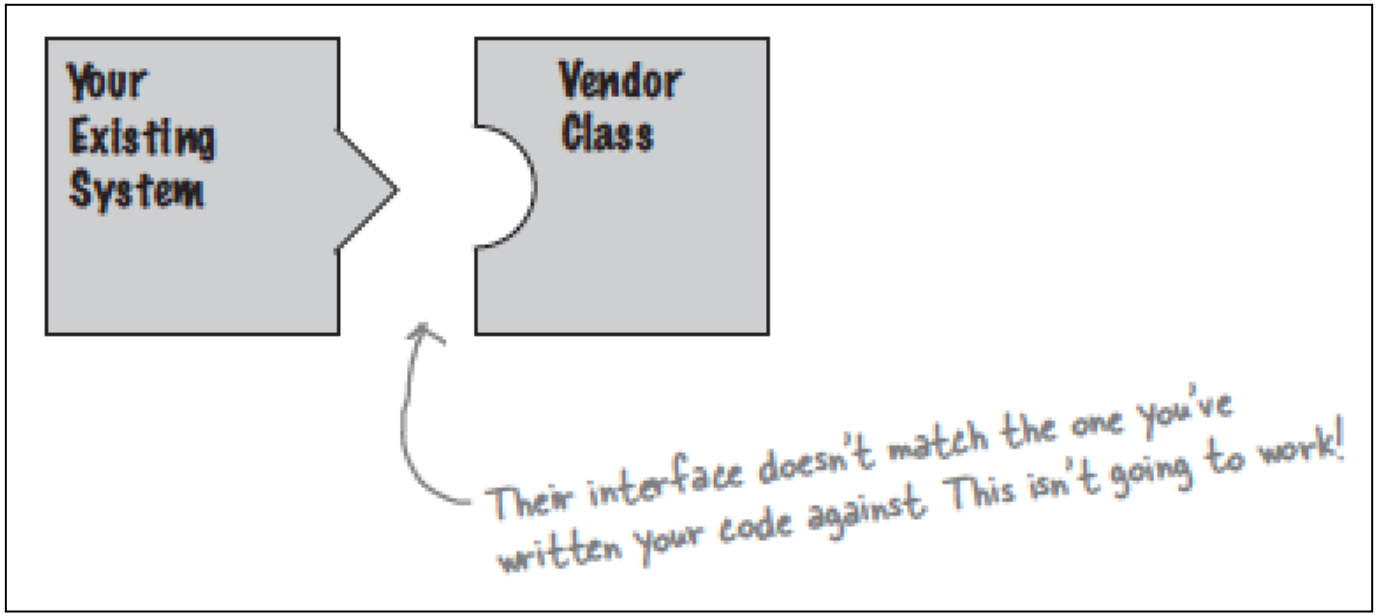
深入淺出設計模式#5:
轉接器模式與表象模式

Prepare by Sean Hsieh

Adapter pattern

Scene

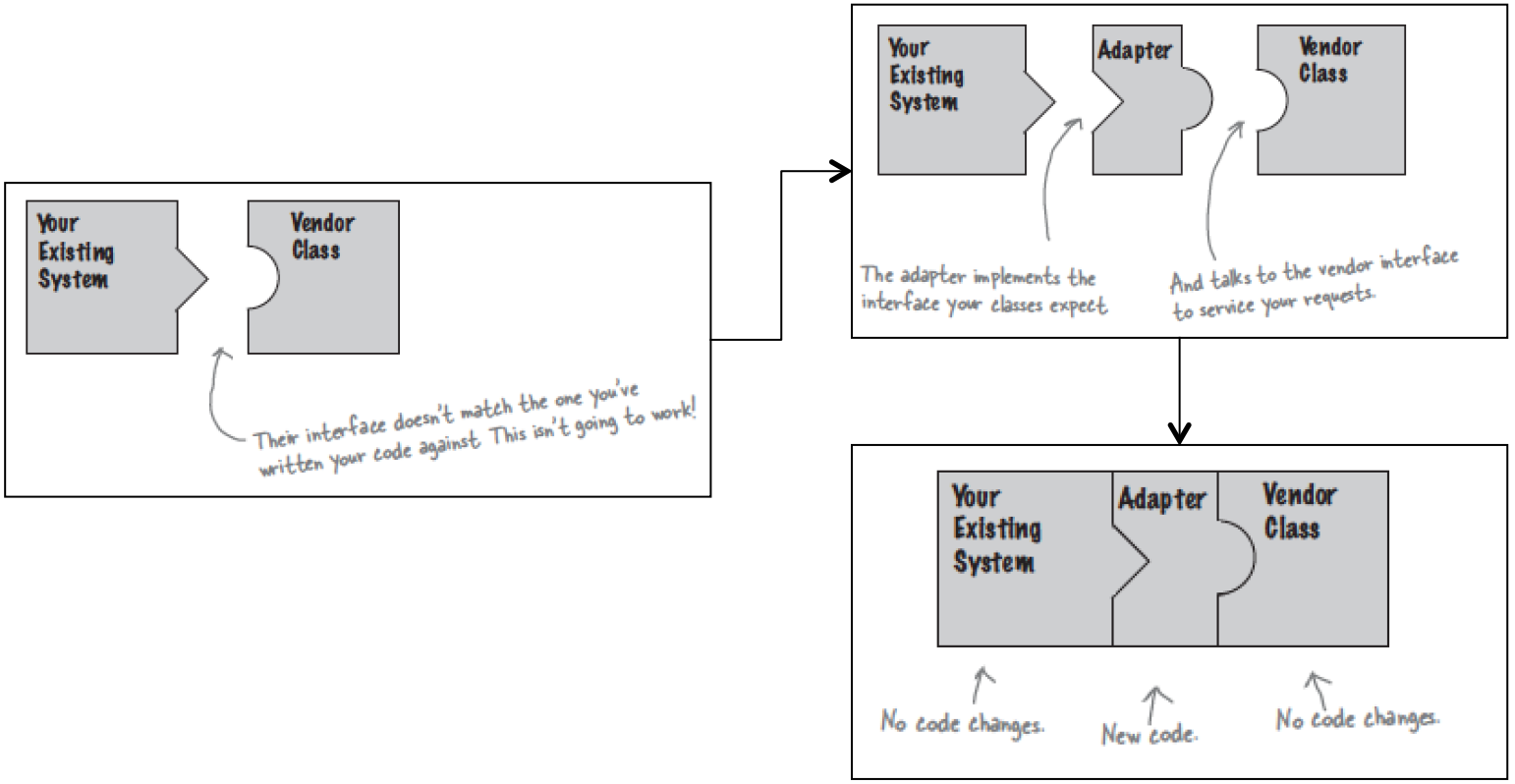
- **New interface doesn't work**
  - *New vendor* designed their interfaces differently than the *last vendor*.



Data source: Head first design pattern

How to use old design?

- **An adapter designed**
  - Designed an interfaces to adapter the old interface.



Data source: Head first design pattern

Example of adapter class

First, you need to implement the interface of the type you're adapting to. This is the interface your client expects to see

```
public class TurkeyAdapter implements Duck {
    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();
        }
    }
}
```

Next, we need to get a reference to the object that we are adapting; here we do that through the constructor.

Now we need to implement all the methods in the interface; the quack() translation between classes is easy: just call the gobble() method.

Even though both interfaces have a fly() method, Turkeys fly in short spurts – they can't do long-distance flying like ducks. To map between a Duck's fly() method and a Turkey's, we need to call the Turkey's fly() method five times to make up for it.

Data source: Head first design pattern

Example of using adapter class

```
File Edit Window Help DontForgetToDuck
%java RemoteControlTest

The Turkey says...

Gobble gobble

I'm flying a short distance

The Duck says...
Quack
I'm flying

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
```

```
public class DuckTestDrive {
    public static void main(String[] args) {
        MallardDuck duck = new MallardDuck();

        WildTurkey turkey = new WildTurkey();
        Duck turkeyAdapter = new TurkeyAdapter(turkey);

        System.out.println("The Turkey says...");
        turkey.gobble();
        turkey.fly();

        System.out.println("\nThe Duck says...");
        testDuck(duck);

        System.out.println("\nThe TurkeyAdapter says...");
        testDuck(turkeyAdapter);
    }

    static void testDuck(Duck duck) {
        duck.quack();
        duck.fly();
    }
}
```

Let's create a Duck... and a Turkey.

And then wrap the turkey in a TurkeyAdapter, which makes it look like a Duck.

Then, let's test the Turkey: make it gobble, make it fly.

Now let's test the duck by calling the testDuck() method, which expects a Duck object.

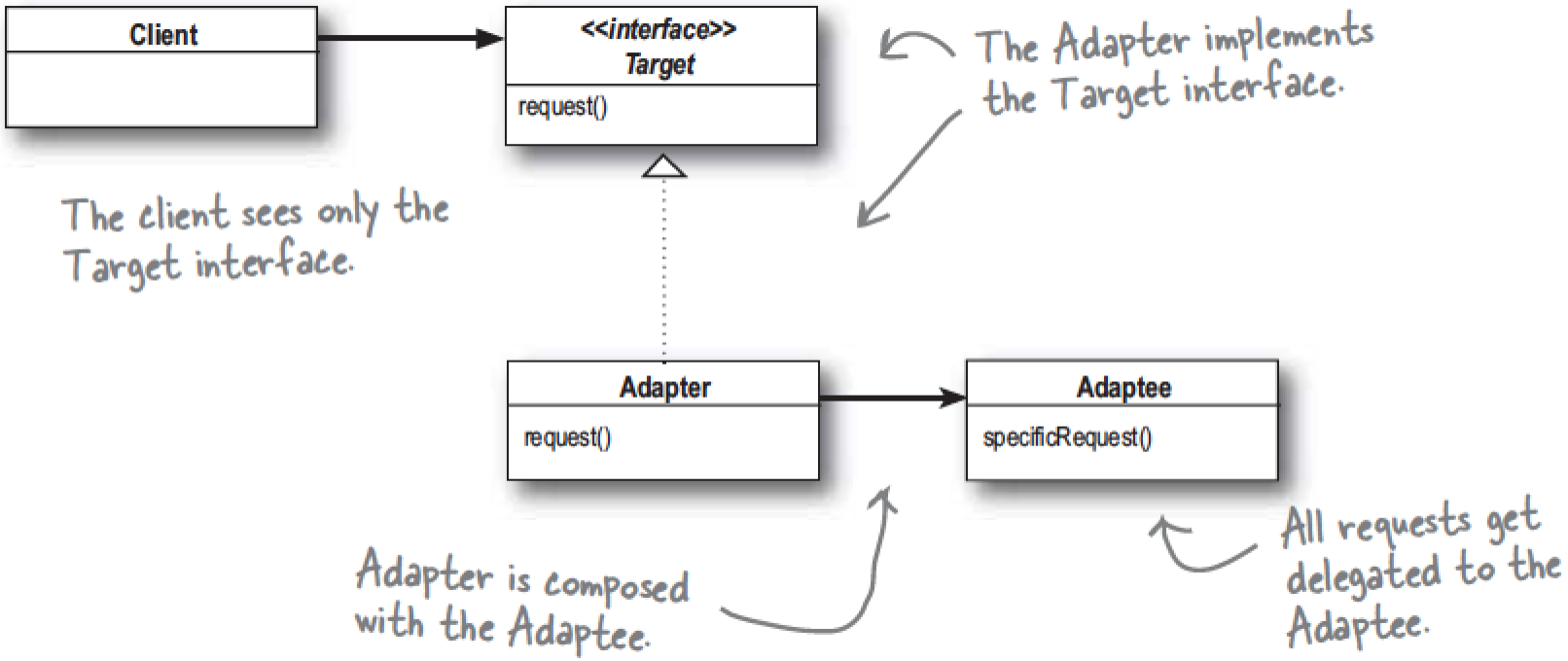
Now the big test: we try to pass off the turkey as a duck...

Here's our testDuck() method; it gets a duck and calls its quack() and fly() methods.

Data source: Head first design pattern

Definition of adapter pattern

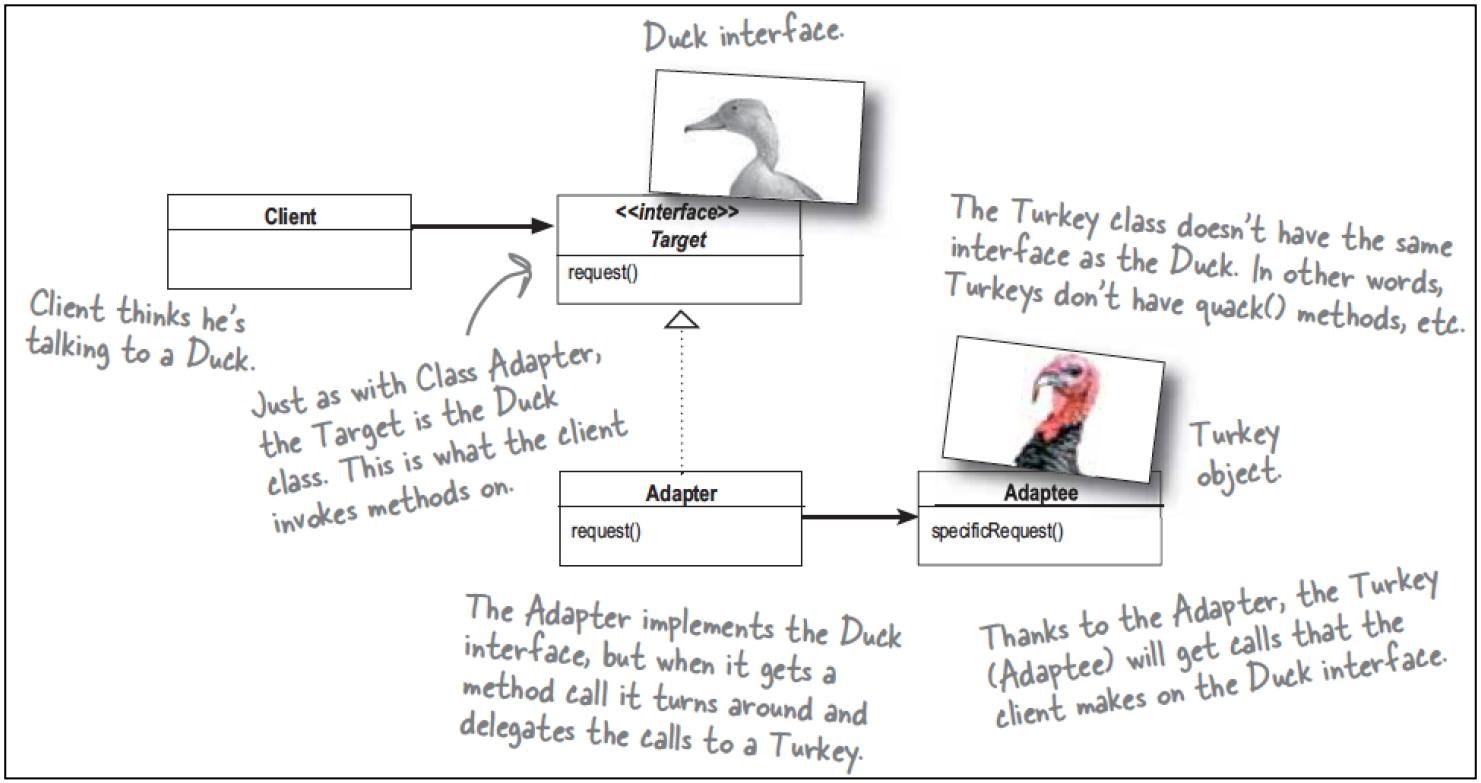
- **Adapter pattern**
  - When you need to use an existing class and interface is not the one you need
  - Usage:
    - converts the interface of a class into another interface the clients except



Data source: Head first design pattern

Kinds of adapter pattern

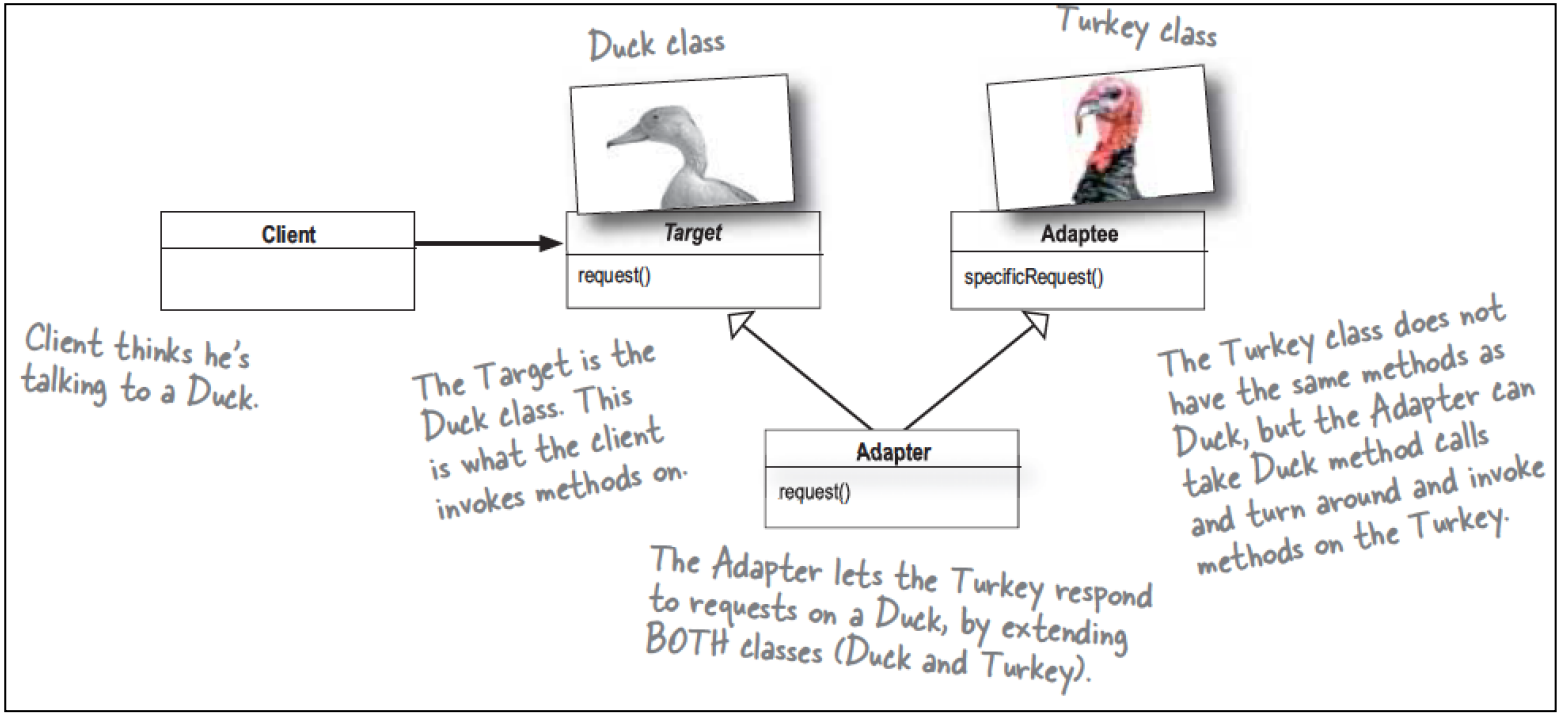
- 1. Object Adapter



Data source: Head first design pattern

Kinds of adapter pattern (cont.)

- 2. Class Adapter
  - Class adapter use multiple inheritance, so you can't do it in **JAVA**



Data source: Head first design pattern

Note

- 1. We could use **adapter** with any subclass of the adaptee.
- 2. Use **adapter** to decouple the client from the implemented interface.
- 3. You can rework all your client-side calls to the interface, which would result in a lot of investigative work and code changes.

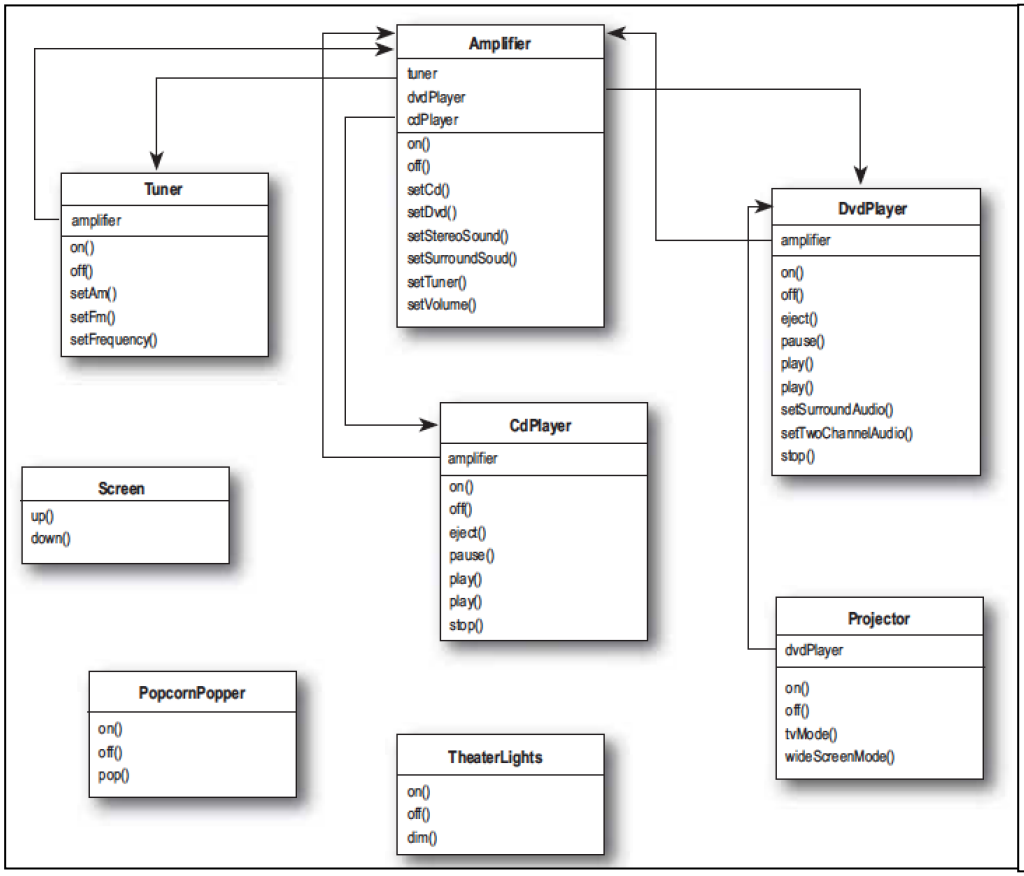
Facade pattern

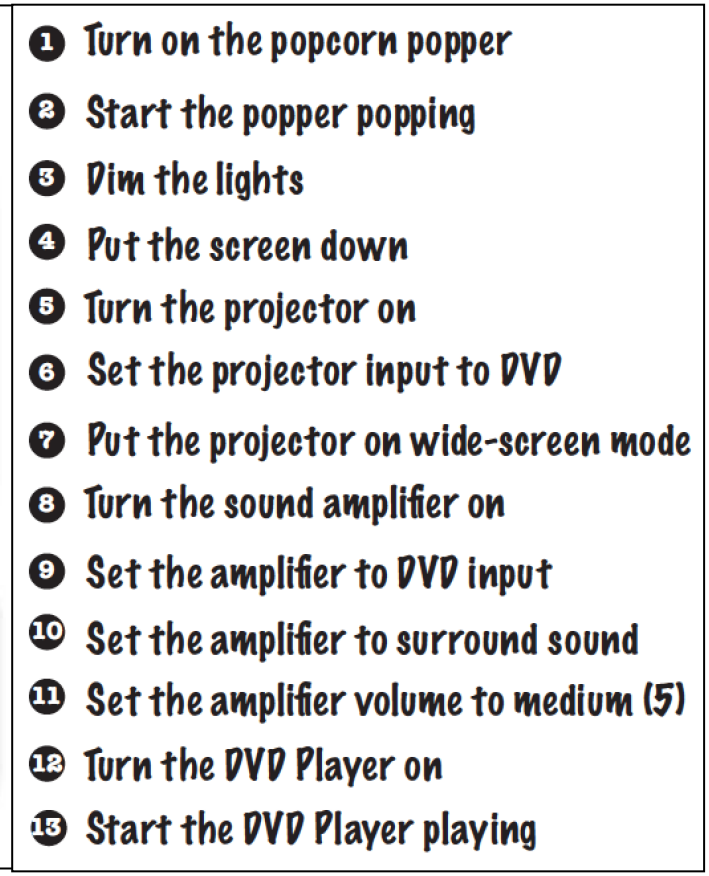
An adapter hold two or more adaptee

Data source: Head first design pattern

scene

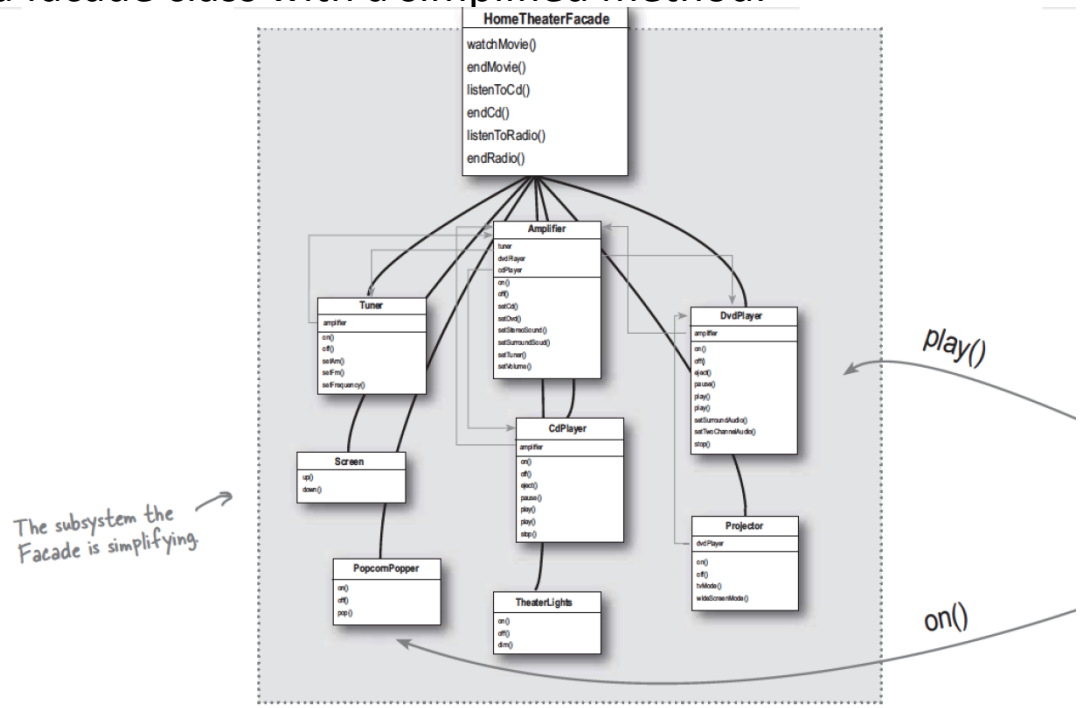
- Home theater
  - We have to do many thing before watching movie





How to simplify these procedure?

- **Facade pattern**
  - When you need to simplify and unify a large interface or complex set of interfaces
  - Usage:
    - Create a facade class with a simplified method.



Data source: Head first design pattern

Example of 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...

Data source: Head first design pattern

Example of facade pattern (cont.)

```
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.

Data source: Head first design pattern

Example of using facade pattern

```
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.

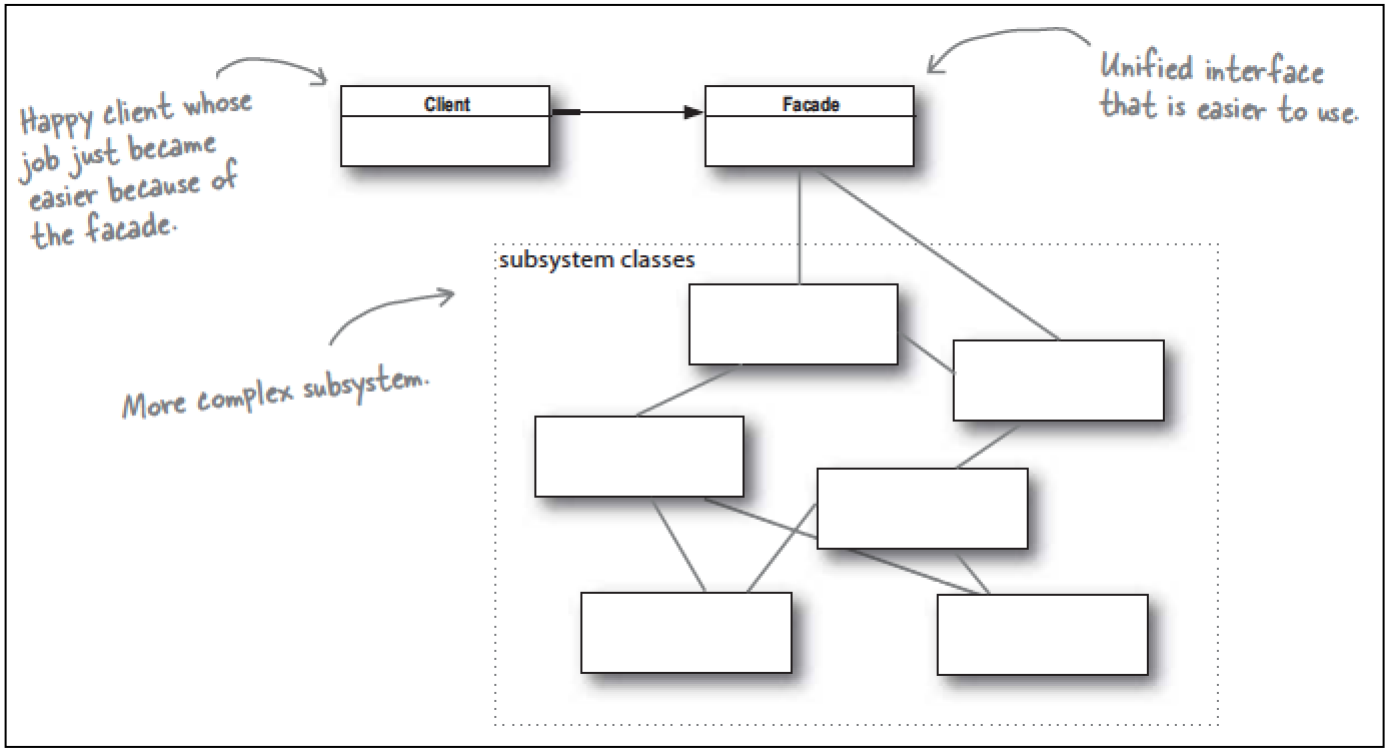
```
File  Edit  Window  Help  SnakesWhyDidHaveToBeSnakes?
%java HomeTheaterTestDrive

Get ready to watch a movie...
Popcorn Popper on
Popcorn Popper popping popcorn!
Theater Ceiling Lights dimming to 10%
Theater Screen going down
Top-O-Line Projector on
Top-O-Line Projector in widescreen mode (16x9 aspect ratio)
Top-O-Line Amplifier on
Top-O-Line Amplifier setting DVD player to Top-O-Line DVD Player
Top-O-Line Amplifier surround sound on (5 speakers, 1 subwoofer)
Top-O-Line Amplifier setting volume to 5
Top-O-Line DVD Player on
Top-O-Line DVD Player playing "Raiders of the Lost Ark"
Shutting movie theater down...
Popcorn Popper off
Theater Ceiling Lights on
Theater Screen going up
Top-O-Line Projector off
Top-O-Line Amplifier off
Top-O-Line DVD Player stopped "Raiders of the Lost Ark"
Top-O-Line DVD Player eject
Top-O-Line DVD Player off
%
```

Data source: Head first design pattern

Definition of facade pattern

- **Facade pattern** provide a unified interface to a set of interfaces in a subsystem.



Data source: Head first design pattern

Note

- 1. **Facade** don't "encapsulate" the subsystem classes; they merely provide a simplified interface to their functionality.
- 2. **Facade** also allows you decouple your client implementation from any one subsystem.

The principle of Least knowledge

- Talk only to your immediate friends.
- Also called **Law of Demeter**
- Ex:

Without the Principle

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

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

With the Principle

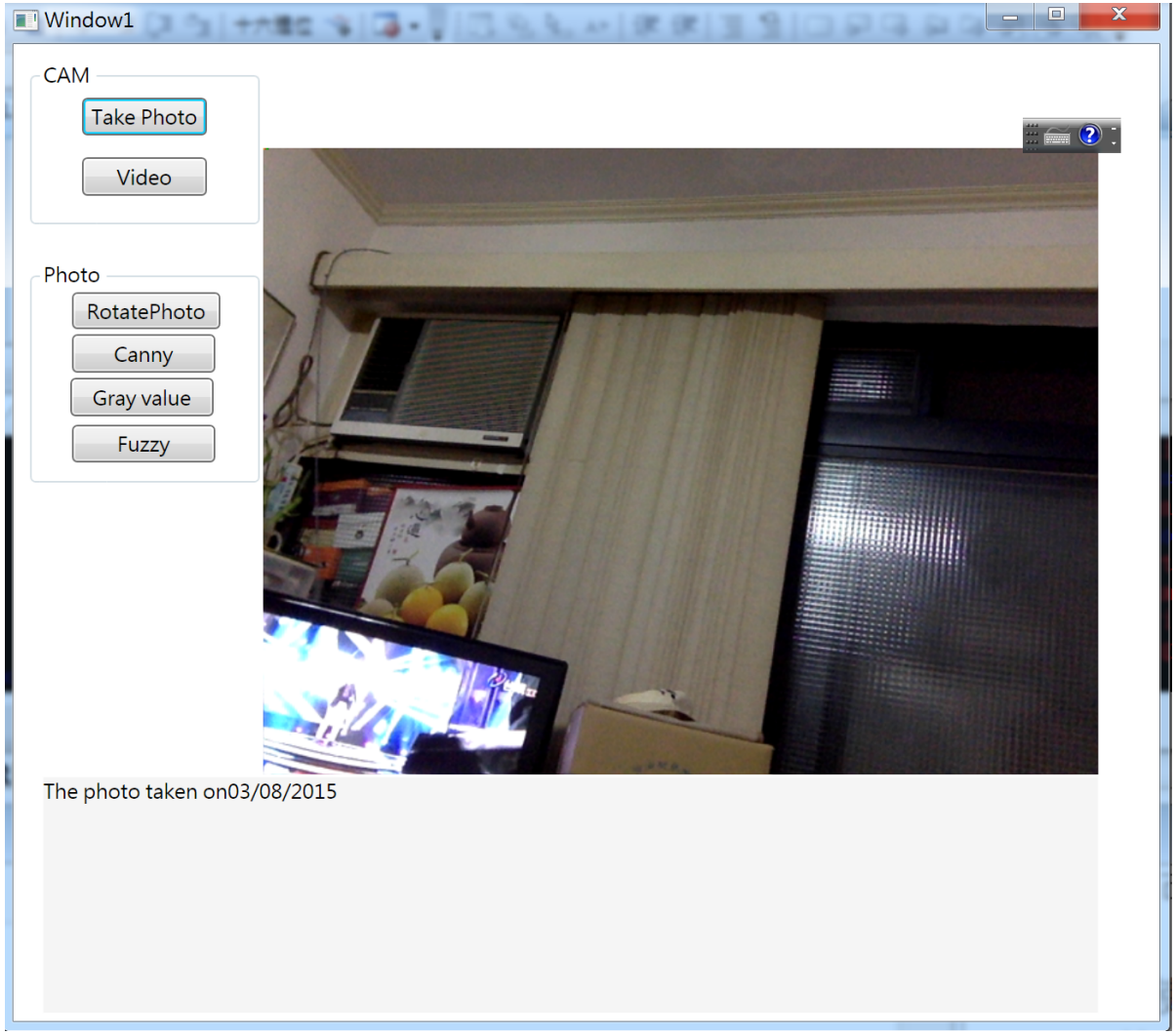
```
public float getTemp() {
    return station.getTemperature();
}
```

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.

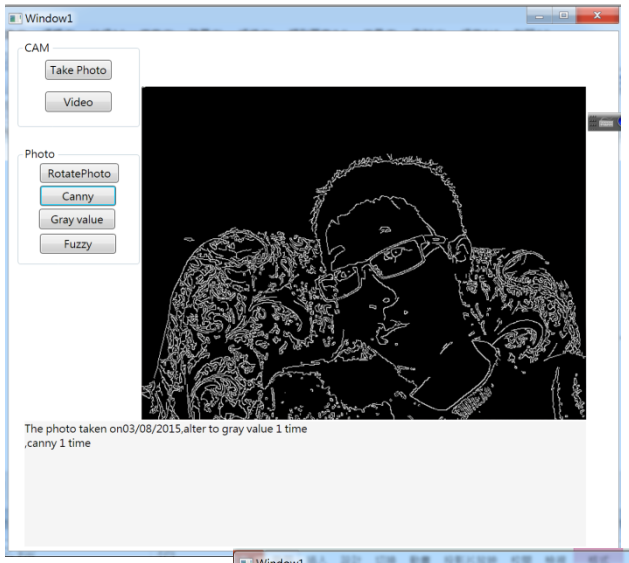
Data source: Head first design pattern

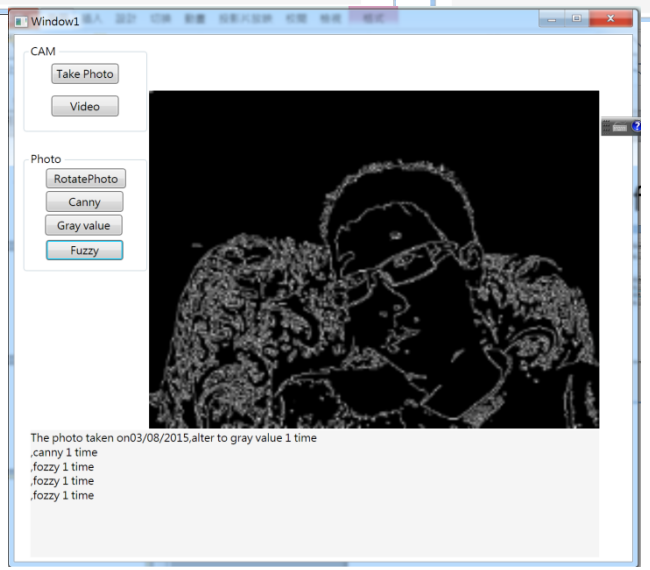
More

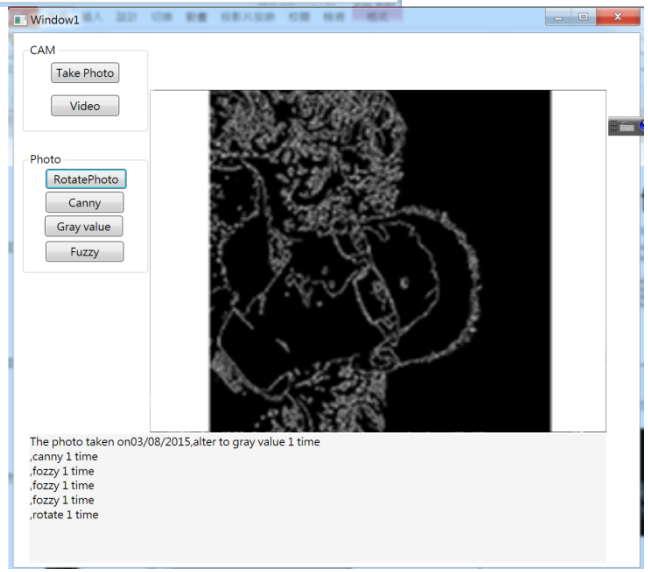
Image editor



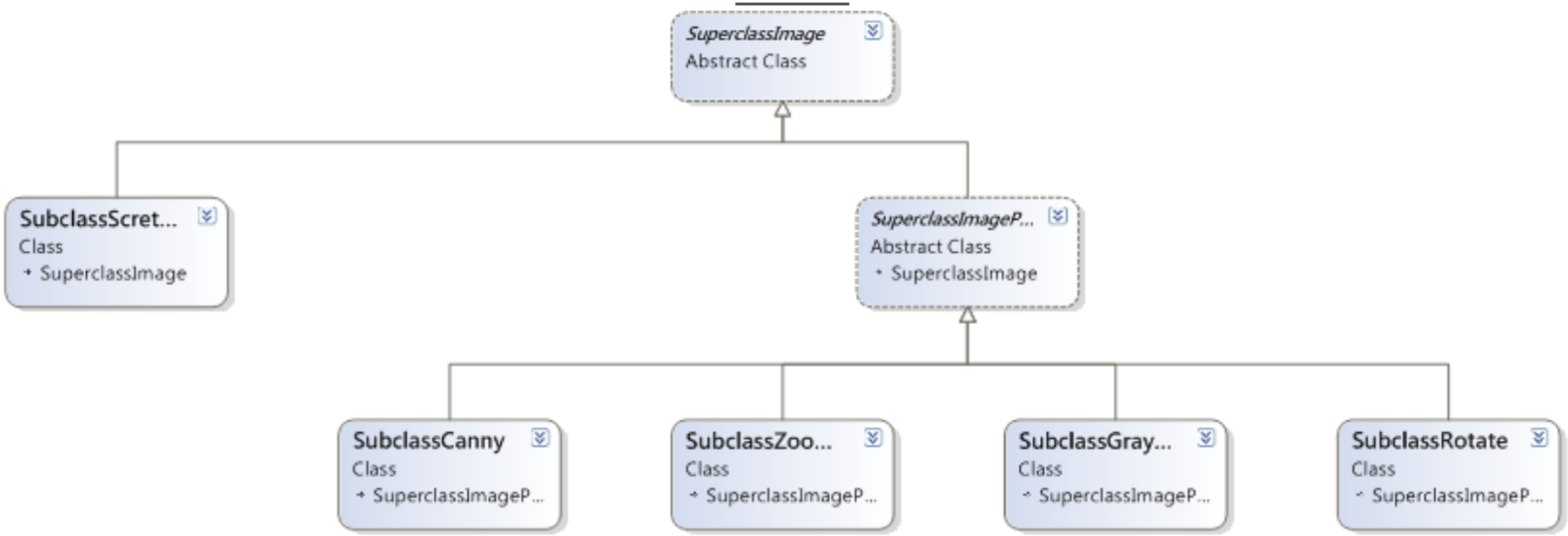
Functions of image editor



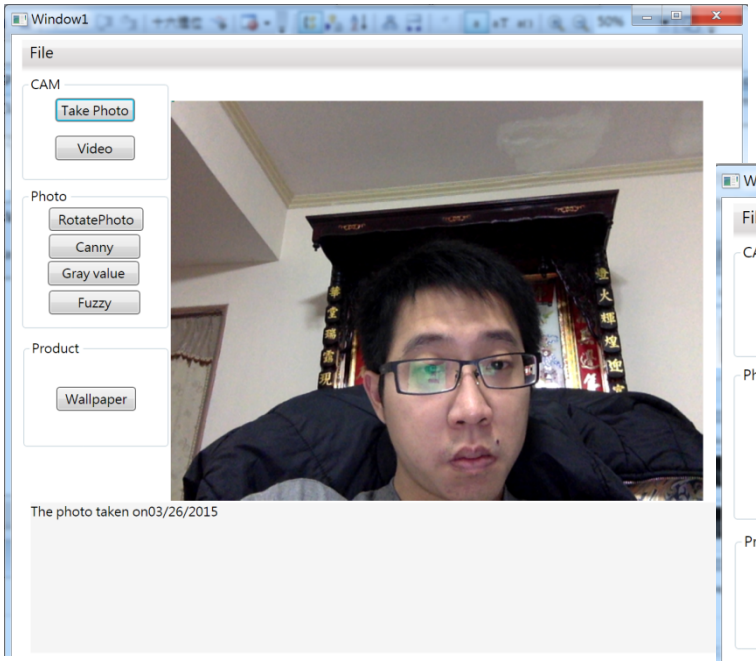


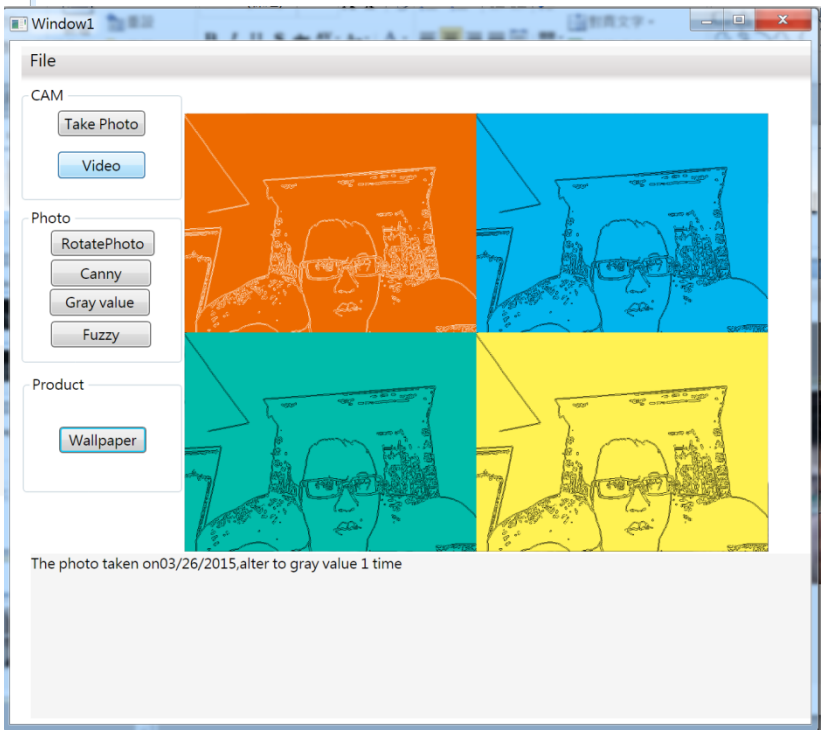


Class diagram of image editor

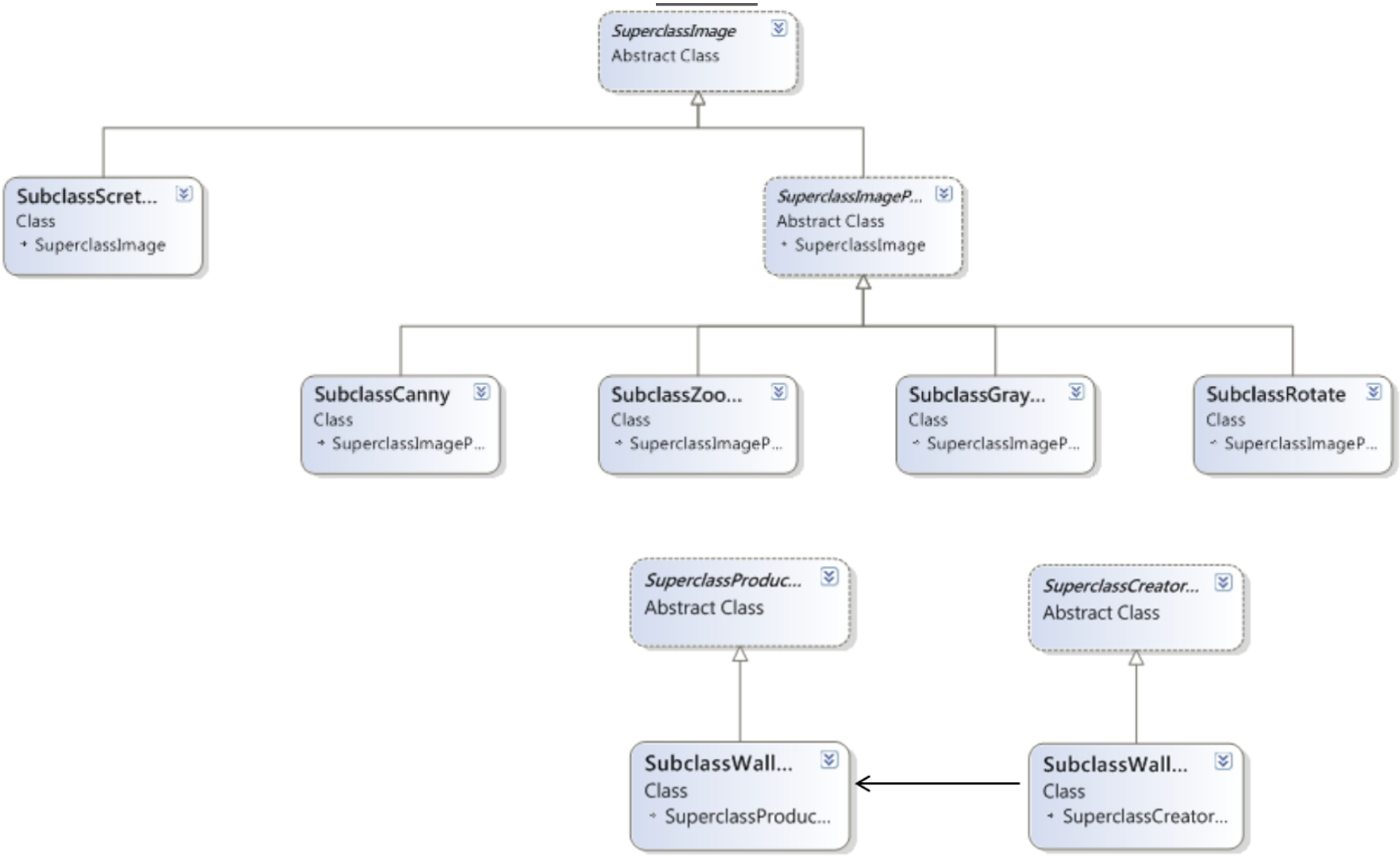


Wallpaper Product

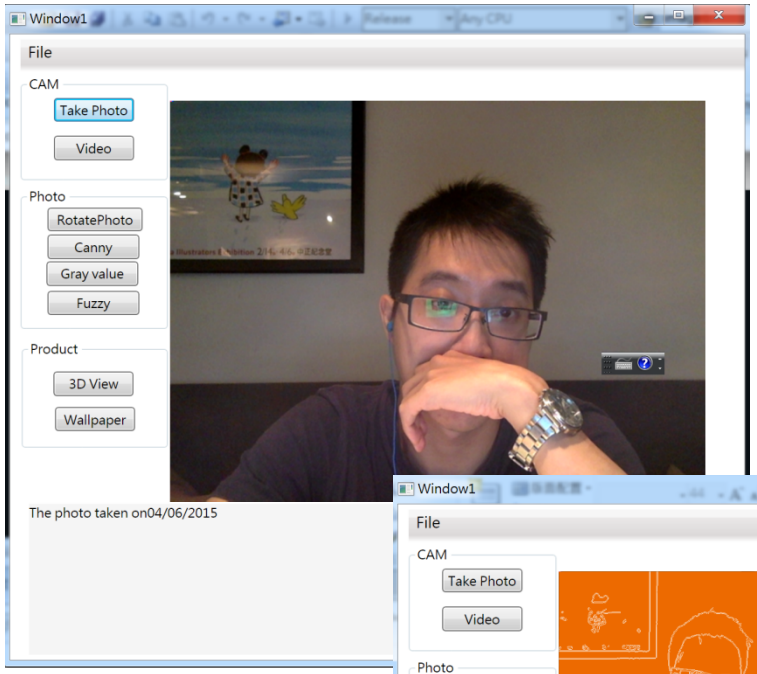


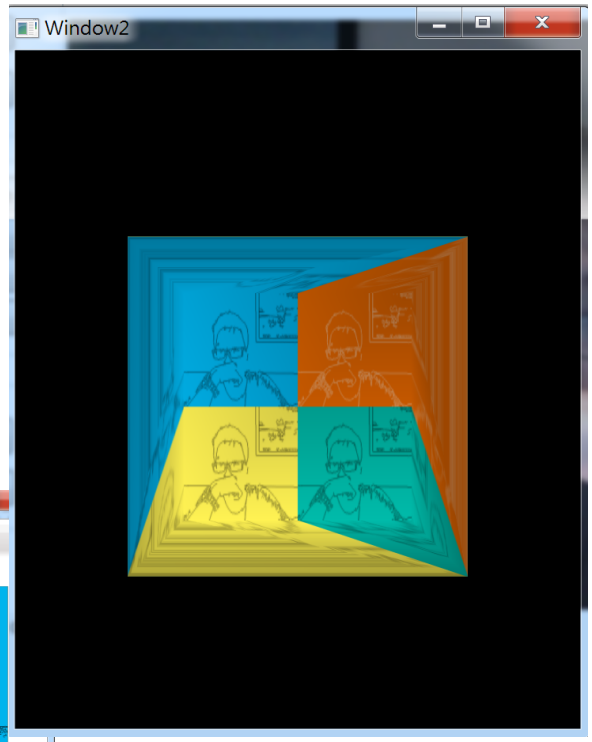


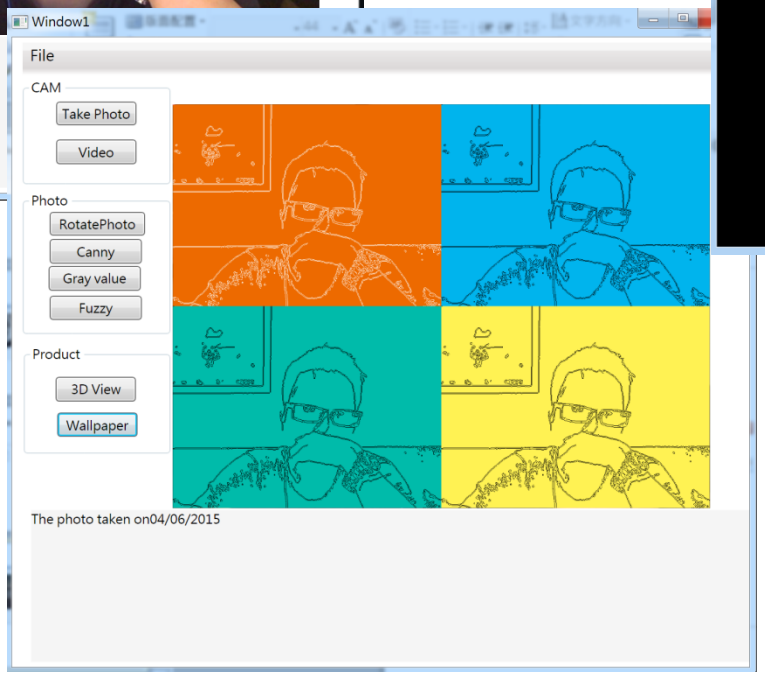
Class diagram of Wallpaper Product



3D View


Window1
File
CAM
Take Photo
Video
Photo
RotatePhoto
Canny
Gray value
Fuzzy
Product
3D View
Wallpaper
The photo taken on04/06/2015


Window2


Window1
File
CAM
Take Photo
Video
Photo
RotatePhoto
Canny
Gray value
Fuzzy
Product
3D View
Wallpaper
The photo taken on04/06/2015

Class diagram of Wallpaper Product