CS342: Software Design

Oct 9, 2017
Outline

Facade pattern
Observer pattern
Homework 1 classes

Human player replaces a card:
1. Main --o UserPlayer
2. Main -> CardPile
3. CardPile -> Card -> Main
4. Main -> Card -> UserPlayer
Project 1 class redesigned

Player Base class: common fields and methods for user and computer
- Add a card to hand
- Discard a card
- Has flus, has four suit, has etc
- Calculate rank
- Cal # of cards u can discard
- Get hand

Card Pile
- Shuffle
- Deal a card

User Player
- Interactive selection

Computer Player
- AI function

Game Session: main logic and workflow
- Set up pile
- Instantiate Players
- Deal to Players
- Players replace cards
- Decide results and winners

Five CardDraw (Main): UI and orchestration
- Prompt # of players
- Trigger dealing
- User's turn
- AI's turn
- Display results

Card - Number, suit, string
Home theater components
To watch a movie, you need to...

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

```javascript
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);
```
Introducing HomeTheaterFacade

Treat components as a subsystem
- Orchestrates functions of components to perform a higher level task
- Provide a simplified interface to client.
- Integrate, not encapsulate: subsystem still available to client
- Decouples client from components
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;
    }

    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();
    }
}
Facade pattern defined

Provides a unified interface to a set of interfaces in a subsystem.

- Avoid tight coupling between client and subsystem
- Better structured and easier to read code
The Principle of Least Knowledge

Talk only to your immediate friends:
- Reduce interactions between objects
- If lots of objects have dependencies with each other, it’s hard to make changes, and systems becomes fragile

Guideline: an object’s method only calls methods that belong to
- The object itself
- Objects passed in as a parameter
- Any objects the method creates
- Any components of the object

Implying don’t call methods of an object returned by calling other methods
- Otherwise, we increase the number of objects that the class direct knows
Do’s and Don’t’s

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

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

```java
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
    }
}
```
Facade and Principle of least knowledge

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 interfering, we can introduce additional facades to form layers of subsystems.
Weather monitoring application
public class WeatherData {

    // instance variable declarations

    public void measurementsChanged() {
        float temp = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();

        currentConditionsDisplay.update(temp, humidity, pressure);
        statisticsDisplay.update(temp, humidity, pressure);
        forecastDisplay.update(temp, humidity, pressure);
    }

    // other WeatherData methods here
}
Issues with the implementation

Coding to concrete implementation

- When we need to add, remove, modify display features, will need to change client
- Display objects have the same method signature, but different behaviors (sounds familiar?)
- Encapsulate them -- but how?
- Observation pattern
One to many dependency between objects

- When source/subject/publisher state changes, all dependents/observer/subscribers are notified and updated automatically.
- There are many different ways to implement.
Loosely coupled objects

- Subjects and observers barely know each other
- A subject class only implements the interface, it doesn’t know or care who subscribe to it
- Observer can be added any time
- Reuse subjects and observers
- Changing either doesn’t affect the other
Weather station class design

This display element shows the current measurements from the WeatherData object.

WeatherData now implements the Subject interface.

Developers can implement the Observer and Display interfaces to create their own display element.

This one keeps track of the min/avg/max measurements and displays them.
public interface Subject {
    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}

public class WeatherData implements Subject {
    private ArrayList observers;
    private float temperature;
    private float humidity;
    private float pressure;
    public WeatherData() {
        observers = new ArrayList();
    }
    public void registerObserver(Observer o) {
        observers.add(o);
    }
    public void removeObserver(Observer o) {
        int i = observers.indexOf(o);
        if (i >= 0) {
            observers.remove(i);
        }
    }
    public void notifyObservers() {
        for (int i = 0; i < observers.size(); i++) {
            Observer observer = (Observer)observers.get(i);
            observer.update(temperature, humidity, pressure);
        }
    }
    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }
}

// other WeatherData methods here

public interface Observer {
    public void update(float temp, float humidity, float pressure);
}

public interface DisplayElement {
    public void display();
}
Observer 1: current condition display

```java
public interface Observer {
    public void update(float temp, float humidity, float pressure);
}

public interface DisplayElement {
    public void display();
}

public class CurrentConditionsDisplay implements Observer, DisplayElement {
    private float temperature;
    private float humidity;
    private Subject weatherData;
    public CurrentConditionsDisplay(Subject weatherData) {
        this.weatherData = weatherData;
        weatherData.registerObserver(this);
    }

    public void update(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        display();
    }

    public void display() {
        System.out.println("Current conditions: " + temperature + "+F degrees and " + humidity + "% humidity");
    }
}
```
Main function (client)

```java
public class WeatherStation {
    public static void main(String[] args) {
        WeatherData weatherData = new WeatherData();
        CurrentConditionsDisplay currentDisplay = new CurrentConditionsDisplay(weatherData);
        StatisticsDisplay statisticsDisplay = new StatisticsDisplay(weatherData);
        ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData);
        weatherData.setMeasurements(80, 65, 30.4f);
        weatherData.setMeasurements(82, 70, 29.2f);
        weatherData.setMeasurements(78, 90, 29.2f);
    }
}
```

%java WeatherStation

Current conditions: 80.0F degrees and 65.0% humidity
Avg/Max/Min temperature = 80.0/80.0/80.0
Forecast: Improving weather on the way!
Current conditions: 82.0F degrees and 70.0% humidity
Avg/Max/Min temperature = 81.0/82.0/80.0
Forecast: Watch out for cooler, rainy weather
Current conditions: 78.0F degrees and 90.0% humidity
Avg/Max/Min temperature = 80.0/82.0/78.0
Forecast: More of the same
We have to repeat some code across all classes implementing “Subject”

- registerObserver, removeObserver, notifyObserver
- Can we create some framework so we don’t copy and paste them?
Java’s built-in observer pattern

The Observable class keeps track of all your observers and notifies them for you.

Observable is not an interface, so WeatherData extends Observable.

This doesn’t look familiar! Hold tight, we’ll get to this in a sec...

Here's our Subject, which we can now also call the Observable. We don't need the register(), remove() and notifyObservers() methods anymore; we inherit that behavior from the superclass.

We left out the DisplayElement interface, but all the displays still implement it too.

There will be a few changes to make to the update() method in the concrete Observers, but basically it's the same idea... we have a common Observer interface, with an update() method that's called by the Subject.
Built-in vs. home-brew code for subject

```java
import java.util.Observable;
import java.util.Observer;

public class WeatherData extends Observable {
    private float temperature;
    private float humidity;
    private float pressure;
    public WeatherData() {
        measurementsChanged();
        notifyObservers();
    }
    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }
    public float getTemperature() {
        return temperature;
    }
    public float getHumidity() {
        return humidity;
    }
    public float getPressure() {
        return pressure;
    }
}
```

```java
public interface Subject {
    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}

public class WeatherData implements Subject {
    private ArrayList observers;
    private float temperature;
    private float humidity;
    private float pressure;
    public WeatherData() {
        observers = new ArrayList();
    }
    public void registerObserver(Observer o) {
        observers.add(o);
    }
    public void removeObserver(Observer o) {
        int i = observers.indexOf(o);
        if (i >= 0) {
            observers.remove(i);
        }
    }
    public void notifyObservers() {
        for (int i = 0; i < observers.size(); i++) {
            Observer observer = (Observer) observers.get(i);
            observer.update(temperature, humidity, pressure);
        }
    }
}
```
Observer using built-in support

```java
import java.util.Observable;
import java.util.Observer;

public class CurrentConditionsDisplay implements Observer, DisplayElement {
    Observable observable;
    private float temperature;
    private float humidity;
    public CurrentConditionsDisplay(Observable observable) {
        this.observable = observable;
        observable.addObserver(this);
    }
    public void update(Observable obs, Object arg) {
        if (obs instanceof WeatherData) {
            WeatherData weatherData = (WeatherData)obs;
            this.temperature = weatherData.getTemperature();
            this.humidity = weatherData.getHumidity();
            display();
        }
    }
    public void display() {
        System.out.println("Current conditions: "+ temperature + "F degrees and " + humidity + "% humidity");
    }
}
```
Built-in: no guarantee order of observer notification

Is it too much to ask the order to be consistent?
Enterprise Service Bus (ESB)

Implements a communication system between mutually interacting software applications in a service-oriented architecture (SOA)

- Distributed computing
- A service does something and publishes an event on the bus
- Subscribing services picks up the event and take actions
- Typical Observer pattern
- Agile, loosely couple services