CS342: Software Design

Oct 3, 2017
Agenda

Introduction to design patterns

- Class inheritance could be risky
- Repeated code resulted from bad use of interface
- Encapsulation and composition
- Strategy pattern

Observer pattern

Extra topic: Enterprise service bus
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

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

User Player
- Interactive selection

Computer Player
- AI function

Card Pile
- Shuffle
- Deal a card

Card
- Number, suit, string
Player classes

```java
public class UserPlayer extends BasePlayer {
    public int ScanAndValidateCardToDiscard() {
        boolean validInput = false;
        while (!validInput) {
            // scan for input
            // if input is "X", break and return -1
            // if input is between 1 and 5, break and return input
            // otherwise repeat
        }
    }
}

public class OpponentPlayer extends BasePlayer {
    public int AISelectCardToDiscard() {
        // 1. First check if the computer player has "chips"
        // 2. If the hand evaluates to "HighCard", determine
        // 3. Next determine if the user has 4 cards
        // 4. Next if the user has an Ace,discard the other
        // 5. Otherwise, keep the two highest cards
    }
}
```
Things could get much more complicated
Character Base class: common fields and methods for character
- Weight, height, skin/hair/eye... colors, armor, weapon…, HP, CP...
- GainHP(), LoseHP(), Destroyed()...

- Elf
  - Night Elf
  - Blood Elf - Sword
  - Dark Elf - Ride

- Orc
  - Rifleman - Shoot

- Dwarf
  - WildHammer - Hammer

- Dragon
  - Deathwing
  - Malygos
  - Farie

- Can shoot fire
- Can Fly
- Can shoot arrow
Let’s do some inheritance

```java
public class Dragon extends BaseCharacter {
    private int wingWidth;
    private int tailLength;
    // other....

    public void combate () {
        // implement
    }
}

public class Dwarf extends BaseCharacter {
    private int armor;
    private int beardLength;
    // other....

    public void combate () {
        // implement
    }
}

public class Elf extends BaseCharacter {
    private EarShape earShape;
    // other....

    public void combate () {
        // implement
    }
}

public class BaseCharacter {
    // Weight, height, skin/hair/eye... colors
    // , armor, weapon..., HP, CP...
    // - Walk, sleep, eat, combat....

    public void Fly() {
        System.out.println("I believe i can fly");
    }

    public void Fire() {
        System.out.println("Spit a fire ball");
    }

    public void Archer() {
        System.out.println("Shoot an arrow with a bow");
    }
    // other public methods
}
```
Character Base class: common fields and methods for character
- Weight, height, skin/hair/eye... colors, armor, weapon..., HP, CP...
- GainHP(), LoseHP(), Destroyed()...
- Fly(), Fire(), Archer()
How about implement an interface?

```java
public Interface Flyable {
    public void Fly ();
}

public abstract class BaseCharacter implements Flyable Fireable Archerable {
    public void Fly() { }
    public void Fire() { }
    public void Archer() { }
    //other public methods
}

public class Dragon extends BaseCharacter {
    public void Fly () { }
    public void Fire() {
        System.out.println("I don't know how");
    }
}

public class Faerie extends Dragon {
    public void Fly () {
        System.out.println("I don't know how");
    }
    public void Fire () {
        System.out.println("I don't know how");
    }
}

public class DeathWing extends Dragon {
    public void Fly () {
        System.out.println("Fly with wings");
    }
    public void Fire () {
        System.out.println("Spit a fire ball");
    }
}

public class Malygos extends Dragon {
    public void Fly () {
        System.out.println("I believe i can fly");
    }
    public void Fire () {
        System.out.println("Spit a fire ball");
    }
}
```
Character Base class: common fields and methods for character
- Weight, height, skin/hair/eye... colors, armor, weapon…, HP, CP...
- GainHP(), LoseHP(), Destroyed()...
- implements Flyable Fireable Archerable

Night Elf
- Fly(), Fire(), Archer()

Blood Elf
- Fly(), Fire(), Archer()

Dark Elf
- Fly(), Fire(), Archer()

Rifleman
- Fly(), Fire(), Archer()

WildHammer
- Fly(), Fire(), Archer()

Deathwing
- Fly(), Fire(), Archer()

Malygos
- Fly(), Fire(), Archer()

Farie
- Fly(), Fire(), Archer()

Can shoot fire
Can Fly
Can shoot arrow
Repeated and unnecessary code everywhere
The better a software is, the more is’t subject to changes, enhancements, and upgrades

- Thus it will become more and more complicated
- So take enhancement request as a compliment!

So what can we do?

- Encapsulation
- Composition
Encapsulation

Take the parts that vary and encapsulate them, so that later you can alter or extend the parts that vary without affecting those that don’t

- Separating what changes from what stays the same

Now take another look at Warcraft

- What are the fields/behaviors that don’t change?
- What changes?
Identify aspects of your application that vary and separate them from what stay the same

- Fly(), fire(), archer() behaviors vary across different
- Encapsulate them from rest of the system
- Interface “FlyBehavior”, and implemented by a set of classes
- The base class doesn’t need to know implementation of behaviors
Duck class example in textbook - inheritance

```
Duck
quack()
swim()
display()
fly()
// OTHER duck-like methods...

MallardDuck
display() {
  // looks like a mallard
}

RedheadDuck
display() {
  // looks like a redhead
}

RubberDuck
quack()
  // overridden to Squeak
} 
display() {
  // looks like a rubberduck
}
```
Separate what vary and encapsulate them

The Duck class is still the superclass of all ducks, but we are pulling out the fly and quack behaviors and putting them into another class structure.

Now flying and quacking each get their own set of classes.

Various behavior implementations are going to live here.

Pull out what varies:

Duck class

Flying Behaviors

Quacking Behaviors

Duck Behaviors
Duck and its behaviors

The behavior variables are declared as the behavior INTERFACE type.

These methods replace fly() and quack().

Instance variables hold a reference to a specific behavior at runtime.

Duck

FlyBehavior flyBehavior
QuackBehavior quackBehavior

performQuack()
swim()
display()
performFly()
// OTHER duck-like methods...
Abstract duck class and its behaviors

```java
public interface FlyBehavior {
    public void fly();
}

public interface QuackBehavior {
    public void quack();
}

public abstract class Duck {
    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior;

    public Duck() {}  
    public abstract void display();
    public void performFly() {
        flyBehavior.fly();
    }

    public void performQuack() {
        quackBehavior.quack();
    }

    public void swim() {
        System.out.println("All ducks float, even decoys!");
    }
}

public class FlyWithWings implements FlyBehavior {
    public void fly() {
        System.out.println("I am flying with my wings");
    }
}

public class FlyNoWay implements FlyBehavior {
    public void fly() {
        System.out.println("I can’t fly");
    }
}

public class Quack implements QuackBehavior {
    public void quack() {
        System.out.println("Quack");
    }
}

public class MuteQuack implements QuackBehavior {
    public void quack() {
        System.out.println("<< Silence >>");
    }
}

public class Squeak implements QuackBehavior {
    public void quack() {
        System.out.println("Squeak");
    }
}
```
MallardDuck and RubberDuck classes

```java
public class MallardDuck extends Duck {
    public MallardDuck() {
        quackBehavior = new Quack();
        flyBehavior = new FlyWithWings();
    }

    public void display() {
        System.out.println("I'm a real Mallard duck");
    }
}
```

```java
public class RubberDuck extends Duck {
    public RubberDuck() {
        quackBehavior = new MuteQuack();
        flyBehavior = new FlyNoWay();
    }

    public void display() {
        System.out.println("I'm a rubber duck");
    }
}
```

```java
public class MiniDuckSimulator {
    public static void main(String[] args) {
        Duck mallard = new MallardDuck();
        mallard.performQuack();
        mallard.performFly();
    }
}
```
Setting behavior dynamically

Each duck object has FlyBehavior and QuackBehavior, and it delegates flying and quacking

- Composition: an object gets its behavior by being composed with the right object
- Behavior can be changed at runtime
- Favor composition inheritance
- Remember the Thread vs. Runnable example?

```java
public abstract class Duck {
    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior;

    public void setFlyBehavior(FlyBehavior fb) {
        flyBehavior = fb;
    }

    public void setQuackBehavior(QuackBehavior qb) {
        quackBehavior = qb;
    }
}
```

```java
public class MiniDuckSimulator {
    public static void main(String[] args) {
        Duck rubberDuck = new RubberDuck();
        rubberDuck.performQuack();
        rubberDuck.performFly();
        rubberDuck.setFlyBehavior(new FlyWithWings());
        rubberDuck.performFly();
    }
}
```
Big picture

Client makes use of an encapsulated family of algorithms for both flying and quacking.

Encapsulated fly behavior

Encapsulated quack behavior

Think of each set of behaviors as a family of algorithms.

Has-a vs. Is-a
Strategy pattern

Defines a family of algorithms, encapsulates each one, and makes them interchangeable.

Strategy lets the algorithm vary independently from clients that use it.
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
}
Introduction to observer pattern

Defines a one-to-many dependency between objects, so that when one object changes state, all of its dependents are notified and updated automatically

Publisher + Subscribers

Terms: Subject & Observers
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