CS 520
Theory and Practice of Software Engineering
Spring 2020

Object Oriented (OO) Design Principles
January 30, 2020

Today

- Code review and (re)design of an MVC application
- OO design principles
  - Information hiding (and encapsulation)
  - Polymorphism
  - Open/closed principle
  - Inheritance in Java
  - The diamond of death
  - Liskov substitution principle
  - Composition/aggregation over inheritance

Let's review the code of the following application

Source code available on the course web site

OO design principles

- Information hiding (and encapsulation)
- Polymorphism
- Open/closed principle
- Inheritance in Java
- The diamond of death
- Liskov substitution principle
- Composition/aggregation over inheritance
Information hiding

MyClass

public class MyClass {
    public int nElem;
    public int capacity;
    public int top;
    public int[] elesm;
    public boolean canResize;
    ...
    public void resize(int s){...}
    public void push(int e){...}
    public int capacityLeft(){...}
    public int getNumElem(){...}
    public int pop(){...}
    public int[] getElems(){...}
}

Information hiding

Stack

public class Stack {
    public int nElem;
    public int capacity;
    public int top;
    public int[] elesm;
    public boolean canResize;
    ...
    public void resize(int s){...}
    public void push(int e){...}
    public int capacityLeft(){...}
    public int getNumElem(){...}
    public int pop(){...}
    public int[] getElems(){...}
}

Anything that could be improved in this implementation?

Information hiding:
- Reveal as little information about internals as possible.
- Separate public interface from implementation details.
- Reduce complexity.
Information hiding vs. visibility

<table>
<thead>
<tr>
<th>Public</th>
<th>Private</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>???</td>
</tr>
</tbody>
</table>

Information hiding vs. visibility

<table>
<thead>
<tr>
<th>Public</th>
<th>Private</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>???</td>
</tr>
</tbody>
</table>

- Protected, package-private, or friend-accessible (C++).
- Not part of the public API.
- Implementation detail that a subclass/friend may rely on.

OO design principles

- Information hiding (and encapsulation)
- **Polymorphism**
- Open/closed principle
- Inheritance in Java
- The diamond of death
- Liskov substitution principle
- Composition/aggregation over inheritance

A little refresher: what is Polymorphism?
A little refresher: what is Polymorphism?

An object’s ability to provide different behaviors.

Types of polymorphism

● Ad-hoc polymorphism (e.g., operator overloading)
  ○ a + b = String vs. int, double, etc.

● Subtype polymorphism (e.g., method overriding)
  ○ Object obj = ...;
  ○ obj.toString();
  ○ toString() can be overridden in subclasses
  ○ and therefore provide a different behavior.

● Parametric polymorphism (e.g., Java generics)
  ○ class LinkedList<E> {
  ○ A LinkedList can store elements
  ○ void add(E e) {...}
  ○ E get(int index) {...}
  ○ regardless of their type but still
  ○ provide full type safety.

https://www.destroyallsoftware.com/talks/wat

A little refresher: what is Polymorphism?

An object’s ability to provide different behaviors.

Types of polymorphism

● Subtype polymorphism (e.g., method overriding)
  ○ Object obj = ...;
  ○ obj.toString();
  ○ toString() can be overridden in subclasses
  ○ and therefore provide a different behavior.

Subtype polymorphism is essential to many OO design principles.

OO design principles

● Information hiding (and encapsulation)
● Polymorphism
● Open/closed principle
● Inheritance in Java
● The diamond of death
● Liskov substitution principle
● Composition/aggregation over inheritance

Open/closed principle

Software entities (classes, components, etc.) should be:

● open for extensions
● closed for modifications

Good or bad design?
Open/closed principle

Software entities (classes, components, etc.) should be:
- open for extensions
- closed for modifications

```
pubic static void draw(Object o) {
    if (o instanceof Square) {
        drawSquare((Square)o);
    } else if (o instanceof Circle) {
        drawCircle((Circle)o);
    } else {
        ...
    }
}
```

Violates the open/closed principle!

OO design principles

- Information hiding (and encapsulation)
- Polymorphism
- Open/closed principle
- Inheritance in Java
- The diamond of death
- Liskov substitution principle
- Composition/aggregation over inheritance

Inheritance: (abstract) classes and interfaces

```
SequentialList (abstract)
```

```
Inheritance: (abstract) classes and interfaces

**LinkedList** extends **SequentialList**

- **SequentialList** (abstract)
- **LinkedList**

Inheritance: (abstract) classes and interfaces

**LinkedList** extends **SequentialList**

- **SequentialList** (abstract)
- **List**
- **Deque**
- **LinkedLIst**

Inheritance: (abstract) classes and interfaces

**LinkedList** extends **SequentialList** implements **List, Deque**

- **SequentialList** (abstract)
- **List**
- **Deque**
- **LinkedLIst**
Inheritance: (abstract) classes and interfaces

List extends Iterable, Collection

Inheritance: (abstract) classes and interfaces

OO design principles

- Information hiding (and encapsulation)
- Polymorphism
- Open/closed principle
- Inheritance in Java
- The diamond of death
- Liskov substitution principle
- Composition/aggregation over inheritance

The “diamond of death”: the problem
The “diamond of death”: the problem

```
A a = new D();
int num = a.getNum();
...
```

Which getNum() method should be called?

The “diamond of death”: concrete example

Can this happen in Java? Yes, with default methods in Java 8.

OO design principles

- Information hiding (and encapsulation)
- Polymorphism
- Open/closed principle
- Inheritance in Java
- The diamond of death
- Liskov substitution principle
- Composition/aggregation over inheritance

Design principles: Liskov substitution principle

Motivating example
We know that a square is a special kind of a rectangle. So, which of the following OO designs makes sense?

```
Square
  Rectangle

Rectangle
  Square
```
Design principles: Liskov substitution principle

**Subtype requirement**
Let object x be of type T1 and object y be of type T2. Further, let T2 be a subtype of T1 (T2 <: T1). Any provable property about objects of type T1 should be true for objects of type T2.

```
Rectangle
+ width : int
+ height : int
+ setWidth(w : int)
+ setHeight(h : int)
+ getArea() : int
```

Is the subtype requirement fulfilled?

```
Rectangle r = new Rectangle(2, 2);
int A = r.getArea();
int w = r.getWidth();
r.setWidth(w * 2);
assertEquals(A * 2, r.getArea());
```

Violates the Liskov substitution principle!
Design principles: Liskov substitution principle

Subtype requirement

Let object \( x \) be of type \( T_1 \) and object \( y \) be of type \( T_2 \). Further, let \( T_2 \) be a subtype of \( T_1 \) (\( T_2 \) <: \( T_1 \)). Any provable property about objects of type \( T_1 \) should be true for objects of type \( T_2 \).

<table>
<thead>
<tr>
<th>Rectangle</th>
<th>&lt;&lt;interface&gt;&gt;</th>
<th>Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ width:int</td>
<td>+ height:int</td>
<td></td>
</tr>
<tr>
<td>+ setWidth(w:int)</td>
<td>+ setHeight(h:int)</td>
<td>+ getArea():int</td>
</tr>
</tbody>
</table>

OO design principles

- Information hiding (and encapsulation)
- Polymorphism
- Open/closed principle
- Inheritance in Java
- The diamond of death
- Liskov substitution principle
- Composition/aggregation over inheritance

Inheritance vs. (Aggregation vs. Composition)

<table>
<thead>
<tr>
<th>Inheritance</th>
<th>(Aggregation vs. Composition)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person</td>
<td>Customer</td>
</tr>
<tr>
<td>Student</td>
<td>Room</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>is-a relationship</th>
<th>has-a relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bank</td>
<td>Building</td>
</tr>
<tr>
<td>Customer</td>
<td>Stack</td>
</tr>
</tbody>
</table>

Design choice: inheritance or composition?

<table>
<thead>
<tr>
<th>List</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>LinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
</tr>
<tr>
<td>Stack</td>
</tr>
</tbody>
</table>

Hmm, both designs seem valid – what are pros and cons?
Design choice: inheritance or composition?

Pros
- No delegation methods required.
- Reuse of common state and behavior.

Cons
- Exposure of all inherited methods (a client might rely on this particular superclass -> can't change it later).
- Changes in superclass are likely to break subclasses.

Composition/aggregation over inheritance allows more flexibility.

OO design principles: summary

- Information hiding (and encapsulation)
- Open/closed principle
- Liskov substitution principle
- Composition/aggregation over inheritance

Final project description

- Each team of 4-5 will carry out one of the following projects:
  - MSR 2020 Mining Challenge
  - Replication Study
  - Model Inference for Inferring Processes
  - EleNa: Elevation-based Navigation
- The key phases of the project are: topic selection, mid-point presentation, final presentation (and document)
- More details available here:
  [https://people.cs.umass.edu/~hconboy/class/2020Spring/CS520_finalProject.pdf](https://people.cs.umass.edu/~hconboy/class/2020Spring/CS520_finalProject.pdf)