CS 520
Theory and Practice of Software Engineering
Spring 2021

Object Oriented (OO) Design Principles

February 11, 2021
Recap: Some best programming practices

- Decomposition into modules/packages, classes, and methods
- Encapsulation of methods and fields
- Pre- and post-conditions for methods (e.g., defensive programming techniques, run-time assertions)
- Type safety for methods (e.g., enum types instead of ints)
- Readability/Understandability:
  - Naming conventions
  - Documentation (e.g., internal comments, README files)
Triangle class snippet

```java
public class Triangle extends Shape {
    public int b;
    public int h;

    public Triangle(int b, int h) {
        super();
        this.b = b;
        this.h = h;
    }

    public int getB() { return this.b; }

    ...
}
```
Code review

Satisfies best practices: Violates best practices:
Code review

Satisfies best practices:

- Decomposition
- Naming conventions

Violates best practices:

- Encapsulation
- Pre- and post-conditions
- Documentation
Today

● **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
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

<table>
<thead>
<tr>
<th>MyClass</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ nElem : int</td>
</tr>
<tr>
<td>+ capacity : int</td>
</tr>
<tr>
<td>+ top : int</td>
</tr>
<tr>
<td>+ elems : int[]</td>
</tr>
<tr>
<td>+ canResize : bool</td>
</tr>
<tr>
<td>+ resize(s:int):void</td>
</tr>
<tr>
<td>+ push(e:int):void</td>
</tr>
<tr>
<td>+ capacityLeft():int</td>
</tr>
<tr>
<td>+ getNumElem():int</td>
</tr>
<tr>
<td>+ pop():int</td>
</tr>
<tr>
<td>+ getElems():int[]</td>
</tr>
</tbody>
</table>

public class MyClass {
    public int nElem;
    public int capacity;
    public int top;
    public int[] elems;
    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

<table>
<thead>
<tr>
<th>MyClass</th>
<th>public class MyClass {</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ nElem : int</td>
<td>public int nElem;</td>
</tr>
<tr>
<td>+ capacity : int</td>
<td>public int capacity;</td>
</tr>
<tr>
<td>+ top : int</td>
<td>public int top;</td>
</tr>
<tr>
<td>+ elems : int[]</td>
<td>public int[] elems;</td>
</tr>
<tr>
<td>+ canResize : bool</td>
<td>public boolean canResize;</td>
</tr>
<tr>
<td>+ resize(s:int):void</td>
<td>...</td>
</tr>
<tr>
<td>+ push(e:int):void</td>
<td>public void resize(int s){...}</td>
</tr>
<tr>
<td>+ capacityLeft():int</td>
<td>public void push(int e){...}</td>
</tr>
<tr>
<td>+ getNumElem():int</td>
<td>public int capacityLeft();</td>
</tr>
<tr>
<td>+ pop():int</td>
<td>public int getNumElem();</td>
</tr>
<tr>
<td>+ getElems():int[]</td>
<td>public int pop(){...}</td>
</tr>
<tr>
<td></td>
<td>public int[] getElems(){...}</td>
</tr>
</tbody>
</table>

What does MyClass do?
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

Public

???

Private
Information hiding vs. visibility

Public

???

Private

- 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 = ...; ⇒ toString() can be overridden in subclasses
    obj.toString(); and therefore provide a different behavior.

- Parametric polymorphism (e.g., Java generics)
  - class LinkedList<E> {
    void add(E) {...}  ⇒ A LinkedList can store elements
    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)
  
  ```java
  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

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

Good or bad design?
Open/closed principle

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

- **open** for extensions
- **closed** for modifications

```java
public 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!
Open/closed principle

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

- **open** for extensions
- **closed** for modifications

```java
public static void draw(Object s) {
    if (s instanceof Shape) {
        s.draw();
    } else {
        ...
    }
}
```

```java
public static void draw(Shape s) {
    s.draw();
}
```
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
}

LinkedList
Inheritance: (abstract) classes and interfaces

**LinkedList** extends **SequentialList**
Inheritance: (abstract) classes and interfaces

**LinkedLIst** extends SequentialLIst

```
SequentialList {abstract}
```
```
<<interface>> List
```
```
<<interface>> Deque
```
Inheritance: (abstract) classes and interfaces

**LinkedList** extends **SequentialList** implements **List, Deque**
Inheritance: (abstract) classes and interfaces

`<<interface>> Iterable`

`<<interface>> Collection`

`<<interface>> List`
Inheritance: (abstract) classes and interfaces

List extends Iterable, Collection
Inheritance: (abstract) classes and interfaces

- <<interface>> Iterable
- <<interface>> Collection
- SequentialList {abstract}
- <<interface>> List
- <interface> Deque
- LinkedList
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

...  
A a = new D();  
int num = a.getNum();  
...
The “diamond of death”: the problem

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

Which `getNum()` method should be called?

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

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

Which design below should be used?
Design principles: Liskov substitution principle

Rectangle $r = \text{new Rectangle}(2, 2)$;

int $A = r.getArea()$;
int $w = r.getWidth()$;
r.setWidth($w \times 2$);

assertEquals($A \times 2$, r.getArea());

A = 4
w = 2
h = 2

A = 8
w = 4
h = 2
Design principles: Liskov substitution principle

Rectangle r =
new Rectangle(2,2);

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

A = 4
w = 2
h = 2

A = 8
w = 4
h = 2
Design principles: Liskov substitution principle

Rectangle \( r = \) new Rectangle\((2,2)\);
new Square\((2)\);

int \( A = r \).getArea();
int \( w = r \).getWidth();
r.setWidth\((w \times 2)\);

assertEquals\((A \times 2, r \).getArea());
Design principles: Liskov substitution principle

Rectangle r = new Rectangle(2, 2);
new Square(2);

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

Violates the Liskov substitution principle!

```
Rectangle r = new Rectangle(2, 2);
new Square(2);
int A = r.getArea();
int w = r.getWidth();
r.setWidth(w * 2);
assertEquals(A * 2, r.getArea());
```
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>
</tr>
</thead>
<tbody>
<tr>
<td>+ width :int</td>
</tr>
<tr>
<td>+ height:int</td>
</tr>
<tr>
<td>+ setWidth(w:int)</td>
</tr>
<tr>
<td>+ setHeight(h:int)</td>
</tr>
<tr>
<td>+ getArea():int</td>
</tr>
</tbody>
</table>

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

<<interface>>
Shape

Rectangle --> Square
```
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)

- **Person**
  - **Student**: is-a relationship

- **Customer**
  - **Bank**: has-a relationship

- **Building**
  - **Room**: has-a relationship

```
public class Student extends Person{
    public Student(){
        ...
    }
}
```

```
public class Bank {
    Customer c;
    public Bank(Customer c){
        this.c = c;
    }
    ...
}
```

```
public class Building {
    Room r;
    public Building(){
        this.r = new Room();
    }
    ...
}
```
Design choice: inheritance or composition?

Hmm, both designs seem valid -- what are pros and cons?

```java
public class Stack<E> extends LinkedList<E> {
    ...
}
```

```java
public class Stack<E> implements List<E> {
    private List<E> l = new LinkedList<>();
    ...
}
```
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.**

**Pros**
- Highly flexible and configurable: no additional subclasses required for different compositions.

**Cons**
- All interface methods need to be implemented -> delegation methods required, even for code reuse.
OO design principles: summary

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