CS 520
Theory and Practice of Software Engineering
Fall 2020

Object Oriented (OO) Design Principles
September 3, 2020

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

```java
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 getElems(){...}
    public int[] getElems(){...}
}
```

MyClass

```java
+ nElem : int
+ capacity : int
+ top : int
+ elems : int[
+ canResize : bool
+ resize(s:int):void
+ push(e:int):void
+ capacityLeft():int
+ getElems():int[]
```
Information hiding

MyClass
+ nElem : int
+ capacity : int
+ top : int
+ eles : int[]
+ canResize : bool
+ resize(s:int):void
+ push(e:int):void
+ capacityLeft():int
+ getNumElem():int
+ pop():int
+ getElems():int[]

public class MyClass {
    public int nElem;
    public int capacity;
    public int top;
    public int[] eles;
    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(){...}
}

Stack
+ nElem : int
+ capacity : int
+ top : int
+ eles : int[]
+ canResize : bool
+ resize(s:int):void
+ push(e:int):void
+ capacityLeft():int
+ getNumElem():int
+ pop():int
+ getElems():int[]

public class Stack {
    public int nElem;
    public int capacity;
    public int top;
    public int[] eles;
    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(){...}
}

What does MyClass do?

Anything that could be improved in this implementation?

Information hiding

Stack
+ nElem : int
+ capacity : int
+ top : int
+ eles : int[]
+ canResize : bool
+ resize(s:int):void
+ push(e:int):void
+ capacityLeft():int
+ getNumElem():int
+ pop():int
+ getElems():int[]

Information hiding vs. visibility

Public

???

Private

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

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();
  - Can be overridden in subclasses and therefore provide a different behavior.
- Parametric polymorphism (e.g., Java generics)
  - class LinkedList<E> {
    - void add(E); [...]
    - E get(int index) {...}
  }
  - LinkedList can store elements 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 = ...; → toString() can be overridden in subclasses
  - obj.toString(); 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?

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

- `SequentialList` (abstract)
- `LinkedList`

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

**LinkedList** extends **SequentialList**

```
SequentialList (abstract)   <interface> List   <interface> Deque
                     \---------------\
                      |         |         |
                      |         |         |
                      v         v         v
         LinkedList
```

```
SequentialList (abstract)   <interface> List   <interface> Deque
                     \---------------\
                      |         |         |
                      |         |         |
                      v         v         v
         LinkedList
```

```
SequentialList (abstract)   <interface> List   <interface> Deque
                     \---------------\
                      |         |         |
                      |         |         |
                      v         v         v
         LinkedList
```

```
SequentialList (abstract)   <interface> List   <interface> Deque
                     \---------------\
                      |         |         |
                      |         |         |
                      v         v         v
         LinkedList
```

```
SequentialList (abstract)   <interface> List   <interface> Deque
                     \---------------\
                      |         |         |
                      |         |         |
                      v         v         v
         LinkedList
```

```
SequentialList (abstract)   <interface> List   <interface> Deque
                     \---------------\
                      |         |         |
                      |         |         |
                      v         v         v
         LinkedList
```

```
SequentialList (abstract)   <interface> List   <interface> Deque
                     \---------------\
                      |         |         |
                      |         |         |
                      v         v         v
         LinkedList
```

```
SequentialList (abstract)   <interface> List   <interface> Deque
                     \---------------\
                      |         |         |
                      |         |         |
                      v         v         v
         LinkedList
```

```
SequentialList (abstract)   <interface> List   <interface> Deque
                     \---------------\
                      |         |         |
                      |         |         |
                      v         v         v
         LinkedList
```

```
SequentialList (abstract)   <interface> List   <interface> Deque
                     \---------------\
                      |         |         |
                      |         |         |
                      v         v         v
         LinkedList
```

```
SequentialList (abstract)   <interface> List   <interface> Deque
                     \---------------\
                      |         |         |
                      |         |         |
                      v         v         v
         LinkedList
```

```
SequentialList (abstract)   <interface> List   <interface> Deque
                     \---------------\
                      |         |         |
                      |         |         |
                      v         v         v
         LinkedList
```

```
SequentialList (abstract)   <interface> List   <interface> Deque
                     \---------------\
                      |         |         |
                      |         |         |
                      v         v         v
         LinkedList
```

```
SequentialList (abstract)   <interface> List   <interface> Deque
                     \---------------\
                      |         |         |
                      |         |         |
                      v         v         v
         LinkedList
```
Inheritance: (abstract) classes and interfaces

- **List**
  - **AbstractList**
  - **Collection**
  - **Deque**

- **Iterator**

**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

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

Which `getNum()` method should be called?
The “diamond of death”: concrete example

```
Animal
  + canFly(): bool

Bird
  + canFly(): bool

Horse
  + canFly(): bool

Pegasus
```

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?

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

Square

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

Square
```

Is the subtype requirement fulfilled?
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.

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

Square
```

```java
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!
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

Customer
  \--- Bank

Building
  \--- Room

Customer is-a relationship
Bank has-a relationship

Design choice: inheritance or composition?

List <<interface>>
  \--- LinkedList
  \--- Stack

List <<interface>>
  \--- LinkedList
  \--- Stack

public class Stack<E>
  \-- extends LinkedList<E> (no delegation methods required)
  \{ ...

public class Stack<E>
  \-- implements List<E> (reuse of common state and behavior)
  \{ private List<E> l = new LinkedList<>(); ...

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

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

Cons
- Exposure of all inherited methods (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.
- 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

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 documentation)
- More details available here: https://people.cs.umass.edu/~hconboy/class/2020Fall/CS520/finalProject.pdf