CS 520
Theory and Practice of Software Engineering
Fall 2017

OO design principles

September 14, 2017
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

<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>

```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 getNumElem(){...}
    public int pop(){...}
    public int[] getElems(){...}
}
```
Information hiding

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(){...}
}

What does MyClass do?
Information hiding

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

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

```java
public class Stack {
    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(){...}
}
```

Anything that could be improved in this implementation?
Information hiding

- Reveal as little information about internals as possible.
- Segregate public interface and implementation details.
- Reduces 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
OOP 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($E$) {...}
    $E$ get(int index) {...} } ⇒ A LinkedList can store elements regardless of their type but still provide full type safety.
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 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 {
        ...
    }
}
```

<table>
<thead>
<tr>
<th></th>
<th>Square</th>
<th>Circle</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+ drawSquare()</td>
<td>+ drawCircle()</td>
</tr>
</tbody>
</table>

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();
}
```

![Diagram of Shape hierarchy with interface and classes](attachment:image.png)
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
```

```
extends
```

LinkedLIst
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

- SequentialList {abstract}
- LinkedList
- List
- Deque
- Collection

Relationships:
- SequentialList extends LinkedList
- LinkedList implements Iterable
- LinkedList implements List
- List implements Collection
- Collection extends Iterable
- LinkedList extends Deque
- Deque extends List
- List extends Collection
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();
...
```
The “diamond of death”: the problem

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

Which getNum() method should be called?

Diagram showing the inheritance hierarchy with A, B, C, and D classes, each with a getNum() method.
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?*
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 $T_2$.

Is the subtype requirement fulfilled?
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 $T_2$.

```java
Rectangle r = new Rectangle(2,2);
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 $T_2$.

```java
Rectangle r = new Rectangle(2,2);
new Square(2);
int A = r.getArea();
int w = r.setWidth();
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

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>

<<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

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

is-a relationship

Customer

Bank

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

has-a relationship

Building

Room

public class Building {
    Room r;
    public Building(){
        this.r = new Room();
    }
    ...
}

public class Customer {
    public Customer(String name){
        this.name = name;
    }
    ...
}
Design choice: inheritance or composition?

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.

**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