CS 520
Theory and Practice of Software Engineering
Fall 2022

Object Oriented (OO) Design Principles

September 27, 2022
Upcoming assignments

- In-class exercise 1 due Thursday 9/29, 11:59 PM (a little before midnight)
- Homework 1 due Tuesday 10/4, 11:59 PM
- Final project group and topic selection due Thursday 10/6, 11:59 PM
- Week 3 Participation Questionnaire due Friday 9/30, 11:59 PM if you decide to submit it.
Today

- UML crash course
- OO design principles
- Homework 1
UML crash course

The main questions

- What is UML?
- Is it useful, why bother?
- When to (not) use UML?
What is UML?

- Unified Modeling Language.
- Developed in the mid 90’s, improved since.
- Standardized notation for modeling OO systems.
- A collection of diagrams for different viewpoints:
  - Use case diagrams
  - Component diagrams
  - Class and Object diagrams
  - Sequence diagrams
  - Statechart diagrams
  - ...
What is UML?

- Unified Modeling Language.
- Developed in the mid 90’s, improved since.
- Standardized notation for modeling OO systems.
- A collection of diagrams for different viewpoints:
  - Use case diagrams
  - Component diagrams
  - Class and Object diagrams
  - Sequence diagrams
  - Statechart diagrams
  - ...
What is UML?

- Unified Modeling Language.
- Developed in the mid 90’s, improved since.
- Standardized notation for modeling OO systems.
- A collection of diagrams for different viewpoints:
  - Use case diagrams
  - Component diagrams
  - **Class and Object diagrams**
  - Sequence diagrams
  - Statechart diagrams
  - ...
Are UML diagrams useful?
Are UML diagrams useful?

Communication

- Forward design (before coding)
  - Brainstorm ideas (on whiteboard or paper).
  - Draft and iterate over software design.

Documentation

- Backward design (after coding)
  - Obtain diagram from source code.

Code generation

- Generating source code from diagrams is challenging.
- Code generation may be useful for skeletons.

In this class, we will use UML class diagrams mainly for visualization and discussion purposes.
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)
  - \( a + b \) \( \Rightarrow \) String vs. int, double, etc.

- Subtype polymorphism (e.g., method overriding)
  - Object obj = ...; \( \Rightarrow \) 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\((int\ index)\) {...}
  }
  \( \Rightarrow \) A 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 and therefore provide a different behavior.

Subtype polymorphism is essential to many OO design principles.
Classes vs. objects

Class
● Grouping of similar objects.
  ○ Student
  ○ Car
● Abstraction of common properties and behavior.
  ○ Student: Name and Student ID
  ○ Car: Make and Model

Object
● Come from the real world.
● Instance of a class
  ○ Student: Juan (4711), Jane (4712), ...
  ○ Car: Audi A6, Honda Civic, Tesla S, ...
UML class diagram: basic notation

MyClass
# UML class diagram: basic notation

<table>
<thead>
<tr>
<th>MyClass</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>- attr1 : type</td>
<td>Attributes</td>
</tr>
<tr>
<td>+ foo() : ret_type</td>
<td>Methods</td>
</tr>
</tbody>
</table>

**Attributes**

<visibility> <name> : <type>

**Methods**

<visibility> <name>(<param>*): <return type>

<param> := <name> : <type>
UML class diagram: basic notation

### MyClass

<table>
<thead>
<tr>
<th>Name</th>
<th>Attributes</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>- attr1 : type</td>
<td>- bar(a:type) : ret_type</td>
</tr>
<tr>
<td></td>
<td># attr2 : type</td>
<td>+ foo() : ret_type</td>
</tr>
<tr>
<td></td>
<td>+ attr3 : type</td>
<td></td>
</tr>
<tr>
<td></td>
<td>~ package-private</td>
<td></td>
</tr>
<tr>
<td></td>
<td># protected</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+ public</td>
<td></td>
</tr>
</tbody>
</table>

#### Visibility

- private
- ~ package-private
# protected
+ public
UML class diagram: basic notation

<table>
<thead>
<tr>
<th>Name</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MyClass</td>
<td>&lt;visibility&gt; &lt;name&gt; : &lt;type&gt;</td>
</tr>
<tr>
<td></td>
<td>Static attributes or methods are underlined</td>
</tr>
<tr>
<td></td>
<td>Methods</td>
</tr>
<tr>
<td></td>
<td>&lt;visibility&gt; &lt;name&gt;(&lt;param&gt;*) : &lt;return type&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;param&gt; := &lt;name&gt; : &lt;type&gt;</td>
</tr>
</tbody>
</table>

Visibility

- private
~ package-private
# protected
+ public

<table>
<thead>
<tr>
<th>MyClass</th>
</tr>
</thead>
<tbody>
<tr>
<td>- attr1 : type</td>
</tr>
<tr>
<td># attr2 : type</td>
</tr>
<tr>
<td>+ attr3 : type</td>
</tr>
<tr>
<td>__bar(a:type) : ret_type</td>
</tr>
<tr>
<td>+ foo() : ret_type</td>
</tr>
</tbody>
</table>
So why bother with UML when you have code?
Classes, abstract classes, and interfaces

MyClass

MyAbstractClass
{abstract}

<<interface>

MyInterface
Classes, abstract classes, and interfaces

MyClass

```java
public class MyClass {
    public void op() {
        ...
    }
    public int op2() {
        ...
    }
}
```

MyAbstractClass

```java
public abstract class MyAbstractClass {
    public abstract void op();
    public int op2() {
        ...
    }
}
```

MyInterface

```java
public interface MyInterface {
    public void op();
    public int op2();
}
```

Level of detail in a given class or interface may vary and depends on context and purpose.
public class SubClass extends SuperClass implements AnInterface
Inheritance: (abstract) classes and interfaces

SequentialList
{abstract}

LinkedList
Inheritance: (abstract) classes and interfaces

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

**LinkedList extends SequentialList**
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

```
<<interface>>
Iterable
```

```
<<interface>>
Collection
```

```
<<interface>>
List
```

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

SequentialList {abstract} extends LinkedList

LinkedList implements Iterable

List extends Deque

Collection extends Iterable
UML class diagram: Aggregation and Composition

**Aggregation**

- Existence of Part does not depend on the existence of Whole.
- Lifetime of Part does not depend on Whole.
- No single instance of Whole is the unique owner of Part (might be shared with other instances of Whole).

**Composition**

- Part cannot exist without Whole.
- Lifetime of Part depends on Whole.
- One instance of Whole is the single owner of Part.
Aggregation or Composition?

Room

Building

Customer

Bank
Aggregation or Composition?

Composition

Room

Building

Aggregation

Customer

Bank

What about class and students or body and body parts?
Inheritance vs. (Aggregation vs. Composition)

**Inheritance**

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

**Aggregation**

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

**Composition**

```java
public class Building {
    Room r;
    public Building(){
        this.r = new Room();
    }
    ...
}
```
UML class diagram: multiplicity

Each A is associated with exactly one B
Each B is associated with exactly one A

Each A is associated with any number of Bs
Each B is associated with exactly one or two As
UML class diagram: navigability

1. Navigability: not specified
   - A 
   - B
   - "can reach B from A"

2. Navigability: unidirectional
   - A 
   - B

3. Navigability: bidirectional
   - A 
   - B
UML class diagram: example
Questions about the UML class diagram example

1. Which classes implement TimedDevice?

2. For class AbstractCGMreceiver:
   a. How many fields?
   b. How many methods?

3. Which class extends AbstractCGMreceiver?

4. What is the relationship between AbstractCGMreceiver and Alert?
Questions about the UML class diagram example

1. Which classes implement TimedDevice?

2. For class AbstractCGMreceiver:
   a. How many fields? 8
   b. How many methods? 5

3. Which class extends AbstractCGMreceiver?

4. What is the relationship between AbstractCGMreceiver and Alert?

   has-a: 1 AbstractCGMreceiver has 3 or more Alerts
OO design principles

- **Single** responsibility
- **Open-closed**
- **Liskov** substitution
- **Interface** segregation
- **Dependency** inversion

[https://en.wikipedia.org/wiki/SOLID]
OO design principles

- **Single responsibility**
- **Open-closed**
- **Liskov substitution**
- **Interface segregation**
- **Dependency inversion**

[https://en.wikipedia.org/wiki/SOLID]
Single responsibility principle

“A class should have one and only one reason to change, meaning that a class should have only one job.”

[https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design]
Single responsibility principle (cont)

**Example:** MVC architecture pattern

- Model is responsible for the data representation
- Views are responsible for visualizing the Model
- Controller is responsible for user interactions
OO design principles

- **Single responsibility**
- **Open-closed**
- **Liskov substitution**
- **Interface segregation**
- **Dependency inversion**

[https://en.wikipedia.org/wiki/SOLID]
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>Square</th>
<th>+ drawSquare()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circle</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
class Square {
    public void draw() {
        // draw square
    }
}
class Circle {
    public void draw() {
        // draw circle
    }
}
class Shape {
    public void draw() {
        // draw shape
    }
}
```

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

```
public static void draw(Shape s) {
    s.draw();
}
```
### Information hiding

#### MyClass

| + nElem : int       | public int nElem; |
| + capacity : int    | public int capacity; |
| + top : int         | public int top; |
| + elems : int[]     | public int[] elems; |
| + canResize : bool  | public boolean canResize; |
| + resize(s:int):void| public void resize(int s){...} |
| + push(e:int):void  | public void push(int e){...} |
| + capacityLeft():int| public int capacityLeft(){...} |
| + getNumElem():int  | public int getNumElem(){...} |
| + pop():int         | public int pop(){...} |
| + getElems():int[]   | public int[] getElems(){...} |
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(){...}
}
```

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>

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(){...}
}
Information hiding:

- Reveal as little information about internals as possible.
- Separate public interface from implementation details.
- Reduce complexity.

### Stack

- `nElem : int`
- `capacity : int`
- `top : int`
- `elems : 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 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

- **Single responsibility**
- **Open-closed**
- **Liskov substitution**
- **Interface segregation**
- **Dependency inversion**

[https://en.wikipedia.org/wiki/SOLID]
Design principles: Liskov substitution principle

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

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

\[
\begin{align*}
\text{int } A &= r.getArea(); \\
\text{int } w &= r.getWidth(); \\
r.setWidth(w \times 2); \\
\text{assertEquals}(A \times 2, r.getArea());
\end{align*}
\]

\( h = 2 \) \hspace{1cm} \(
\begin{array}{c}
\text{w = 2} \\
\text{A = 4}
\end{array}
\)

\( h = 2 \) \hspace{1cm} \(
\begin{array}{c}
\text{w = 4} \\
\text{A = 8}
\end{array}
\)
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());
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

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

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

assertEqual\( s(A \times 2, r \).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.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>
</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>

```csharp
Rectangle

<<interface>>

Shape

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

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

```java
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.
Homework 1: Basic stats app

- Implemented in Java AWT and Swing

- Topics:
  - MVC architecture pattern
  - Manual review (non-functional requirements, design principles, best practices)
  - Proposed extension

- Due: Tuesday October 4, 2022, 11:59 PM (a little before midnight)
Basic stats app: User Interface

![Simple stats interface](image-url)

- **Numbers**: 3
- **Median**: 3.0
- **Mean**: 3.0

Input numbers: 1, 0, 3, 0, 5, 0,
Basic stats app: Git repo (v2.0.0)