Upcoming assignments

- In-class exercise 1 due Thursday 2/17, 11:59 PM (a little before midnight) [Submit on Moodle]
- Homework 1 due Thursday 2/24, 11:59 PM [Submit on Gradescope]
- Final project group and topic selection due Thursday 2/24, 11:59 PM [Submit on Moodle]
- There is no Week 3 Participation Questionnaire.
Today

- UML crash course
- OO design principles
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.
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

MyClass

- attr1 : type

+ foo() : ret_type

Name

Attributes

\(<visibility> \ <name> \ : \ <type>\)

Methods

\(<visibility> \ <name>(<param>*) \ : \ <return\ type>\)
\(<param> \ := \ <name> \ : \ <type>\)
### UML class diagram: basic notation

#### MyClass

<table>
<thead>
<tr>
<th>Visibility</th>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>attr1</td>
<td>type</td>
</tr>
<tr>
<td>#</td>
<td>attr2</td>
<td>type</td>
</tr>
<tr>
<td>+</td>
<td>attr3</td>
<td>type</td>
</tr>
<tr>
<td>~</td>
<td>bar(a:type)</td>
<td>ret_type</td>
</tr>
<tr>
<td>+</td>
<td>foo()</td>
<td>ret_type</td>
</tr>
</tbody>
</table>

#### Name

#### Attributes

<visibility> <name> : <type>

#### Methods

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

#### Visibility

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

<table>
<thead>
<tr>
<th>MyClass</th>
</tr>
</thead>
</table>
| - attr1 : type  
  # attr2 : type  
  + attr3 : type  |
| ~ bar(a:type) : ret_type  
  + foo() : ret_type |

Name

Attributes
<visibility> <name> : <type>

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

Visibility
- private  
~ package-private  
# protected  
+ public

Static attributes or methods are underlined
public class Person {
    ...
}

public class Student extends Person {
    private int id;
    
    public Student(String name, int id) {
        ...
    }
    
    public int getId() {
        return this.id;
    }
}

So why bother with UML when you have code?
Classes, abstract classes, and interfaces

- MyClass
- MyAbstractClass
  - {abstract}
- 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
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?
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
   - “can reach B from A”

2. Navigability: unidirectional
   - “can reach B from A”

3. Navigability: bidirectional
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?
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

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

A stack can be implemented by hiding the following members:

- `nElem`, `capacity`, `top`, `elems`, and `canResize` are private members of the class.
- The methods `resize`, `push`, `capacityLeft`, `getNumElem`, `pop`, and `getElems` are public methods.

```java
public class Stack {
    private int nElem;
    private int capacity;
    private int top;
    private int[] elems;
    private 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

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 and therefore provide a different behavior.

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

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

```
<<interface>>
Shape
+ draw()
```

```
Square
Circle
...
```
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
```

Linked List
Inheritance: (abstract) classes and interfaces

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

```
<<interface>>
Iterable
```
```
<<interface>>
Collection
```
```
<<interface>>
List
```
List extends Iterable, Collection
Inheritance: (abstract) classes and interfaces

```
SequentialList  {abstract}    List
      |   \       |   \    |
      |    \      \   |
      |     \  extends  extends|
      |      \        |
      \     v        \
        LinkedList

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

- SequentialList is an abstract class that extends SequentialList and implements Iterable and List.
- LinkedList extends SequentialList and implements List.
- List extends Iterable and implements Collection.
- Deque also 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

```
...  
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?
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 = 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);

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
Design principles: Liskov substitution principle

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

int $A = r.getArea();$
int $w = r.getWidth();$
$r.set Width(w * 2);$

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

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

\[ \text{int } A = r.getArea(); \]
\[ \text{int } w = r.getWidth(); \]
\[ r.setWidth(w \times 2); \]
\[ \text{assertEquals}(A \times 2, r.getArea()); \]
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 $T_2$.

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

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>
</tbody>
</table>

```
+ setWidth(w:int)
+ setHeight(h:int)
+ getArea():int
```

```
<<interface>>
Shape
```

```
Rectangle
Square
```

Rectangle

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**

**public class Bank**

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

**has-a relationship**

**public class Customer**

**Bank**

**public class Building**

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

**Room**
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.