Recap

- Modeling and abstraction
- Software architecture vs. software design

Recap: software architecture examples

- Pipe and filter
- N-tier / client-server
- MVC (Model-View-Controller)

Recap: software architecture principles

- Separation of concerns
- High-level encapsulation
- Manageable complexity
- Well-defined public API
Today

- Specification
- Architecture
- Design
- Source code
- Development process
- Level of abstraction

Today

- OO design principles
  - Information hiding (and encapsulation)
  - Composition/aggregation over inheritance
  - Open/closed principle
  - Liskov substitution principle
- A first design problem (if time permits)

Information hiding

<table>
<thead>
<tr>
<th>MyClass</th>
<th>What does this class do?</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>

Information hiding

<table>
<thead>
<tr>
<th>Stack</th>
<th>What does this class do?</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>

Anything wrong with this Stack implementation?
Information hiding

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:
- Reveal as little information about internals as possible
- Segregate public interface and implementation details
- Reduce complexity

Information hiding vs. visibility

Public

???

Private

Today

- OO design principles
  - Information hiding (and encapsulation)
  - Composition/aggregation over inheritance
  - Open/closed principle
  - Liskov substitution principle
- A first design problem (if time permits)
Aggregation vs. composition

Consider object ownership and lifetime to determine the correct type of association!

Design choice: inheritance or composition?

Which is the right choice?
Think about is-a vs. has-a relationship first.
Design choice: inheritance or composition?

Which is the right choice?
Think about is-a vs. has-a relationship first.
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 (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.

Classes, abstract classes, and interfaces

Recall how default methods (Java 8) fit into this spectrum?
Composition/aggregation over inheritance

The “diamond of death”

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

Which version of `getNum()` should be called?

Can you think of a particular method in Java for which this problem could arise (if Java would allow multiple inheritance)?

Today

- OO design principles
  - Information hiding (and encapsulation)
  - Composition/aggregation over inheritance
  - Open/closed principle
  - Liskov substitution principle
- A first design problem (if time permits)
**Design principles: open/closed principle**

**Software entities** (classes, components, etc.) should be:
- **open** for extensions
- **closed** for modifications

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

**Good or bad?**

1. **Square**
   - `+ drawSquare()`

2. **Circle**
   - `+ drawCircle()`

---

**Today**

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

- **A first design problem (if time permits)**
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?

- Square
- Rectangle

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

Square

Violates the Liskov substitution principle!
A first design problem

Weather station revisited

Current

<table>
<thead>
<tr>
<th>Temperature</th>
<th>30 day history</th>
</tr>
</thead>
<tbody>
<tr>
<td>25° F</td>
<td>Temperature graph</td>
</tr>
<tr>
<td>-3.9° C</td>
<td>min: 20° F, max: 35° F</td>
</tr>
</tbody>
</table>

We focus on the view -> what would be a good design?

Weather station: view

```
public void draw(Data d) {
  for (View v : allViews) {
    v.draw(d);
  }
}
```