Recap

- Design principles
  - Separation of concerns
  - Information hiding (and encapsulation)
  - Composition/aggregation over inheritance
  - Open/closed principle
  - Liskov substitution principle
Today

- Homework 1: scope and questions.
- Design problems & potential solutions.
- Design patterns
  - What is a design pattern?
  - Categories of design patterns.
A little refresher

So wait, how can I tell the difference between inheritance, aggregation, and composition?

Polymorphism means what again and why is it important?

Why do I care about this composed view thing? What actual problem does it solve?
Inheritance vs. (aggregation vs. composition)

**is-a relationship**

**Stack**
- public class Stack extends LinkedList{
  public Stack(){
    ...  
  }
}

**has-a relationship**

**Stack**
- public class Stack {
  LinkedList l;
  public Stack(LinkedList l){
    this.l = l;
  }
  ...
}

**Stack**
- public class Stack {
  LinkedList l;
  public Stack(){
    this.l = new LinkedList();
  }
  ...
}
Inheritance vs. (aggregation vs. composition)

**is-a relationship**

- **LinkedList**
  - **Stack**

**has-a relationship**

- **public class Stack**
  ```java
  public class Stack {
    LinkedList l;
    public Stack() {
      this.l = new LinkedList();
    }
    public Stack(LinkedList l) {
      this.l = l;
    }
    ...
  }
  ```

**What about class and students?**

**What about body and body parts?**
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> {`  $\Rightarrow$ The list can store elements regardless of their type but still
    `void add(E) {...} E get(int index) {...}`  provide full type safety.
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.

We are focusing on subtype polymorphism.
Why use Polymorphism?

```java
... 
LinkedList<Integer> list1 = new LinkedList<>();
LinkedList<Integer> list2 = new LinkedList<>();
... // add elements to the lists

Integer min1 = getMin(list1);
Integer min2 = getMin(list2);
System.out.println("Min list1: " + min1);
System.out.println("Min list2: " + min2);
}
public Integer getMin(LinkedList<Integer> list) {
    sort(list);
    return list.get(0);
}
public void sort(LinkedList<Integer> list) {
    ... // sort the list
}
Why use Polymorphism?

```java
...  
ArrayList<Integer> list1 = new LinkedList<>();  
ArrayList<Integer> list2 = new LinkedList<>();  
... // add elements to the lists

Integer min1 = getMin(list1);  
Integer min2 = getMin(list2);  
System.out.println("Min list1: "+ min1);  
System.out.println("Min list2: "+ min2);
}

public Integer getMin(LinkedList<Integer> list) {
    sort(list);
    return list.get(0);
}

public void sort(LinkedList<Integer> list) {
    ... // sort the list
}

What if we want to use ArrayLists instead?
```
Why use Polymorphism?

```java
... LinkedList<Integer> list1 = new LinkedList<>();
ArrayList<Integer> list2 = new ArrayList<>();
... // add elements to the lists

Integer min1 = getMin(list1);
Integer min2 = getMin(list2);
System.out.println("Min list1: "+min1);
System.out.println("Min list2: "+min2);
}
public Integer getMin(LinkedList<Integer> list) {
    sort(list);
    return list.get(0);
}
public void sort(LinkedList<Integer> list) {
    ... // sort the list

What if we want to use an ArrayList and a LinkedList?
Why use Polymorphism?

We can solve these problems with subtype polymorphism.
A first design problem

Weather station revisited

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

Temp. sensor

What’s a good design for the view?
What’s a good design for the view?

Client

25° F
-3.9° C

min: 20° F
max: 35° F

Reset button

Temp. sensor

Reset history

sees
uses
updates
manipulates

09/01, 12°
09/02, 14°
...

Weather station: view

<<interface>>
View
+draw(d:Data)

SimpleView
+draw(d:Data)

GraphView
+draw(d:Data)

...View
+draw(d:Data)

ComplexView
-views:List<View>
+draw(d:Data)
+addView(v:View)

25° F
-3.9° C
min: 20° F
max: 35° F

How do we need to implement draw(d:Data)?
public void draw(Data d) {
    for (View v : views) {
        v.draw(d);
    }
}
Design pattern: Composite

Component

+operation()

CompA

+operation()

CompB

+operation()

Composite

-comps:Collection<Component>

+operation()

+addComp(c:Component)

+removeComp(c:Component)
Design pattern: Composite

<<interface>>

Component

+operation()

CompA

+operation()

CompB

+operation()

Composite

-comps:Collection<Component>

+operation()
+addComp(c:Component)
+removeComp(c:Component)

1..n

Iterate over all composed components (comps), call operation() on each, and potentially aggregate the results.
What is a design pattern?

- Addresses a recurring, common design problem.
- Provides a generalizable solution.
- Provides a common terminology.
What is a design pattern?

- Addresses a recurring, common design problem.
- Provides a generalizable solution.
- Provides a common terminology.

Pros
- Improves communication and documentation.
- “Toolbox” for novice developers.
What is a design pattern?

- Addresses a recurring, common design problem.
- Provides a generalizable solution.
- Provides a common terminology.

Pros
- Improves communication and documentation.
- “Toolbox” for novice developers.

Cons
- Risk of over-engineering.
- Potential impact on system performance.

More than just a name for common sense and best practices.
Design patterns: categories

1. Structural

2. Behavioral

3. Creational
Design patterns: categories

1. Structural
   - Composite
   - Decorator
   - ...

2. Behavioral
   - Template method
   - Visitor
   - ...

3. Creational
   - Singleton
   - Factory (method)
   - ...
Design patterns: categories

1. **Structural**
   - Composite
   - Decorator
   - ...

2. **Behavioral**
   - Template method
   - Visitor
   - ...

3. **Creational**
   - Singleton
   - Factory (method)
   - ...
Another design problem: I/O streams

InputStream is =
   new FileInputStream(...);

int b;
while((b=is.read()) != -1) {
   // do something
}

FileInputStream
+read():int
+read(buf:byte[]):int
Another design problem: I/O streams

... InputStream is =
    new FileInputStream(...);

int b;
while((b=is.read()) != -1) {
    // do something
}
...

Problem: filesystem I/O is expensive
Another design problem: I/O streams

Problem: filesystem I/O is expensive
Solution: use a buffer!

Why not simply implement the buffering in the client or subclass?
Another design problem: I/O streams

...InputStream is =
   new BufferedInputStream(  
       new FileInputStream(...));
int b;
while((b=is.read()) != -1) {
   // do something
}
...
Design pattern: Decorator

[Diagram showing the Decorator design pattern with classes Component, CompA, CompB, and Decorator, and their relationships with operations and decorated objects.]
Composite vs. Decorator

<<interface>>
Component
+operation()

Composite
- comp: Collection<Component>
+operation()
+ addComp(c: Component)
+ removeComp(c: Component)

CompA
+operation()

Decorator
- decorated: Component
+Decorator(d: Component)
+operation()