Today

- MAP.
- Recap: OO design principles.
- Design problems & potential solutions.
- Design patterns:
  - What is a design pattern?
  - Categories of design patterns.
  - Structural design patterns.

Recap

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

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 max: 35° F</td>
</tr>
</tbody>
</table>

Temp. sensor

Reset history button
What’s a good design for the view?

Weather station: view

Design pattern: Composite

Client

- sees
- uses
- manipulates
- updates

25° F
-3.9° C
min: 20° F
max: 35° F

09/01, 12°
09/02, 14°
...

Temp.
Reset
Reset history
button

How do we need to implement draw(d:Data)?

SimpleView
GraphView
... View
ComplexView

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

public void draw(Data d) {
  for (View v : views) {
    v.draw(d);
  }
}
Design pattern: Composite

![Composite design pattern diagram]

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
   - 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
}
```

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

```java
InputStream is = new BufferedInputStream(new FileInputStream(...));
int b;
while((b=is.read()) != -1) {
    // do something
}
```

Still returns one byte (int) at a time, but from its buffer, which is filled by calling read(buf:byte[]).

Design pattern: Decorator

```
<<interface>>
Component
+operation()
```

```
CompA CompB
+operation()
+operation()
```

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

Composite vs. Decorator

```
<<interface>>
Component
+operation()
```

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

```
CompA
+operation()
```

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