Object Oriented Design Patterns
September 8, 2020

Today
- Recap: Object oriented design principles
- Design problems & potential solutions
- Design patterns:
  - What is a design pattern?
  - Categories of design patterns
  - Structural design patterns

Recap
Object oriented design principles
- Information hiding (and encapsulation)
- Open/closed principle
- Liskov substitution principle
- Composition/aggregation over inheritance
  - Can be used to prevent the diamond of death
Design principles: Liskov substitution principle

Rectangle
+ width : int
+ height : int
+ setWidth(w : int)
+ getHeight(h : int)
+ getArea() : int

Square
<<interface>>
Shape

Which design below should be used?

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

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 \( T_1 \) (\( T_2 \subset T_1 \)). Any provable property about objects of type \( T_1 \) should be true for objects of type \( T_2 \).
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 \( T_1 \) (\( T_2 \leq T_1 \)). 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>Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ width: int</td>
<td>&lt;&lt;interface&gt;&gt;</td>
</tr>
<tr>
<td>+ height: int</td>
<td>Shape</td>
</tr>
<tr>
<td>+ setWidth(w:int)</td>
<td></td>
</tr>
<tr>
<td>+ setHeight(h:int)</td>
<td></td>
</tr>
<tr>
<td>+ getArea(): int</td>
<td>Rectangle</td>
</tr>
<tr>
<td></td>
<td>Square</td>
</tr>
</tbody>
</table>

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>

Model View Controller: example

Simple weather station

<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>
Weather station: view

Design pattern: Composite
Design pattern: Composite

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)
   ○ …
Another design problem: I/O streams

```java
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?

```java
BufferedInputStream is = new BufferedInputStream(new FileInputStream(...));
```

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

Composite vs. Decorator

Find the median in an array of doubles

Examples:
- median([1, 2, 3, 4, 5]) = ???
- median([1, 2, 3, 4]) = ???

Find the median in an array of doubles

Examples:
- median([1, 2, 3, 4, 5]) = 3
- median([1, 2, 3, 4]) = 2.5

Algorithm
Input: array of length n  Output: median
Find the median in an array of doubles

Examples:
- median([1, 2, 3, 4, 5]) = 3
- median([1, 2, 3, 4]) = 2.5

Algorithm

Input: array of length n  Output: median

1. Sort array
2. if n is odd return ((n+1)/2)th element
   otherwise return arithmetic mean of (n/2)th element and ((n/2)+1)th element

Median computation: naive solution

```java
public static void main(String ... args) {
    System.out.println(median(1,2,3,4,5));
}

public static double median(double ... numbers) {
    int n = numbers.length;
    boolean swapped = true;
    while(swapped) {
        swapped = false;
        for (int i = 1; i < n; ++i) {
            if (numbers[i-1] > numbers[i]) {
                swap(numbers, i-1, i);
                swapped = true;
            }
        }
    }
    if (n%2 == 0) {
        return (numbers[(n/2)-1] + numbers[n/2]) / 2;
    } else {
        return numbers[n/2];
    }
}
```

Ways to improve

- 1: Monolithic version, static context.
- 2: Extracted sorting method, non-static context.
- 3: Proper package structure and visibility, extracted main method.
- 4: Proper testing infrastructure and build system.

One possible solution: template method pattern

```
AbstractMedian
  (abstract)
  + median(a:double[]):double
    # sort(a:double[])

SimpleMedian
  # sort(a:double[])
```

Italics indicate an abstract method.
One possible solution: template method pattern

AbstractMedian (abstract)
- median(a:double[]):double
  # sort(a:double[])

SimpleMedian
# sort(a:double[])

● The template method (median) implements the algorithm but leaves the sorting of the array undefined.

● The concrete subclass only needs to implement the actual sorting.

Another solution: strategy pattern

<<interface>>
Median
+median(a:double[]):double

<<interface>>
Sorter
+sort(array:double[])

StrategyMedian
- sortStrategy:Sorter

"median" delegates the sorting of the array to a "sortStrategy"

Template method pattern vs. strategy pattern

Two solutions to the same problem

What are the differences, pros, and cons?
Template method pattern vs. strategy pattern

Two solutions to the same problem

Template method
- Behavior selected at compile time.
- Template method is usually final.

Strategy
- Behavior selected at runtime.
- Composition/aggregation over inheritance.

Model-View-Controller revisited

Design patterns in a MVC architecture
**Observer pattern**

- **Problem solved:**
  - A one-to-many dependency between objects should be defined without making the objects tightly coupled.
  - When one object changes state, an open-ended number of dependent objects are updated automatically.
  - One object can notify an open-ended number of other objects.

---

**Variations of the Observer update method**

- `update(state:State)`
  - Alternatively, could decompose the State into pieces

- `update(observable:Observable)`
  - Use the Observable `getState` method(s)

---

**Example: Observer pattern for MVC**

- **Which is the Observable?** Model or View
- **Which is the Observer?** Model or View
- **Which class should use the `setState` method?** Model, View, or Controller
Example: Observer pattern for MVC

- Which is the Observable? Model or View
- Which is the Observer? Model or View
- Which class should use the setState method? Model, View, or Controller

Model-View-Controller revisited

Design patterns in a MVC architecture