CS 520
Theory and Practice of Software Engineering
Spring 2020
Object Oriented Design Patterns
February 4, 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

**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 $T_1$ ($T_2 \subseteq T_1$). Any provable property about objects of type $T_1$ should be true for objects of type $T_2$.

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

```
Square
```

Is the subtype requirement fulfilled?

```java
Rectangle r = new Rectangle(2,2);
int A = r.getArea();
int w = r.getWidth();
r.setWidth(w * 2);
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 \( T_1 \) (\( T_2 \in \text{ subtype of } T_1 \)). 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>&lt;&lt;interface&gt;&gt;</th>
<th><strong>Shape</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>+ width : int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ height : int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ setWidth(w : int)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ setHeight(h : int)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ getArea() : int</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rectangle</th>
</tr>
</thead>
</table>

| Square |

---

**Design patterns**

- What is a design pattern?
- Categories of design patterns
- Structural design patterns

---

**A first design problem**

**Weather station revisited**

Current 30 day history

<table>
<thead>
<tr>
<th>25° F</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>-3.9° C</th>
</tr>
</thead>
</table>

| min: 20° F |
| max: 35° F |

Temp. sensor

Reset history button

---

**Model View Controller: example**

**Simple weather station**

Current 30 day history

<table>
<thead>
<tr>
<th>25° F</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>-3.9° C</th>
</tr>
</thead>
</table>

| min: 20° F |
| max: 35° F |

Temp. sensor

Reset history button

---
What's a good design for the view?

Weather station: view

Weather station: view

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)
   - ...

Design pattern: Composite

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

```
CompA
  +operation()

CompB
  +operation()
```

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

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.

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.
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?
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

```
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] + 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.

**Median computation: naive solution**

What's wrong with this design? How can we improve it?

One possible solution: template method pattern

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

SimpleMedian
    # sort(a:double[])
```
One possible solution: template method pattern

- 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.

Should the median method be final?

Another solution: strategy pattern

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

**Observer pattern**

```
observable (abstract)
+ observers:Set<Observer>
+ register(o:Observer)
+ unregister(o:Observer)
+ stateChanged()

myobservable
- state:State
+ getState():State
+ setState(state:State)

myobserver
- update()
```

Variation: pass incremental changes or the state to update method.

**Model-View-Controller revisited**

**Design patterns in a MVC architecture**

```
Composite
  + sees
  + uses

Strategy
  + sees
  + uses

Observer
  + sees
  + uses

View
  + updates
  + sees

Controller
  + manipulates
  + uses

Model
```

Client
  + sees
  + uses
Homework 1: Code review

- Three in a Row game
  - Implemented in Java
- Topics:
  - Code review
  - Architecture & design patterns
  - Implementation
  - Testing
- Due: Thursday February 27, 2020 9 AM EDT