CS 520
Theory and Practice of Software Engineering
Fall 2021

Object Oriented Design Patterns

September 30, 2021
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

- Open/closed principle
- Information hiding (and encapsulation)
- Liskov substitution principle
- Composition/Aggregation over inheritance
  - Can be used to prevent the diamond of death
Inheritance and polymorphism

Example: https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/Stack.html

1) For the Java inheritance, what are the is-a relationships for the Stack class?

2) What kinds of polymorphism are being used by this class?
   - Ad-hoc
   - Subtype
   - Parametric
Open/Closed and information hiding principles

3) What is a Stack field that is open only for extensions?

4) What is a Stack method that is open only for extensions?
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 \( T_1 \) (\( T_2 <: T_1 \)). Any provable property about objects of type \( T_1 \) should be true for objects of type \( T_2 \).

- Supertype \( T_1 \) is Vector \( // \) It is a List.
- Subtype \( T_2 \) is Stack
- One key property of a List is that the List is represented as an ordered sequence of elements.
- Is this principle: satisfied or violated?
Composition/Aggregation over inheritance

Another example:
https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html

- From this class name, the two parts of the functionality are providing Map access and Linked access.

- How is the class implementing the Map access?

- How is the class implementing the Linked access?
Design patterns

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

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>

Reset history button

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

Controller

Reset

Reset history button

Model

Temp. sensor

01/01 -> 0
01/02 -> -5
01/03 -> -10
01/04 -> -4
...

01/01 -> 0
01/02 -> -5
01/03 -> -10
01/04 -> -4
...
What’s a good design for the view?

- 25° F
- -3.9° C
- min: 20° F
- max: 35° F
- 09/01, 12°
- 09/02, 14°
- ...
Weather station: view

How do we need to implement `draw(d:Data)`?
Weather station: view

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

<<interface>>
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)
```

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.

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

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

InputStream
+read():int
+read(buf:byte[]):int

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

```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()

1

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

CompA
+operation()

CompB
+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()
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 $((n/2)+1)$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]) {
                ...  
                swapped = true;
            }
        }
    }
    if (n%2 == 0) {
        return (numbers[(n/2) - 1] + numbers[n/2]) / 2;
    } else {
        return numbers[n/2];
    }
}
```

What’s wrong with this design? How can we improve it?
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[])

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

SimpleMedian

# sort(a:double[])

- The concrete subclass only needs to implement the actual sorting.
One possible solution: template method pattern

AbstractMedian

{abstract}

+ median(a:double[]):double  # sort(a:double[])

SimpleMedian

# sort(a:double[])
Another solution: strategy pattern

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

StrategyMedian
-sortStrategy:Sorter
+median(a:double[]):double
+setSorter(s:Sorter)
```

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

HeapSort
+sort(...)

QuickSort
+sort(...)
```

“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

Client sees

View

Controller uses

Model

updates

manipulates
Model-View-Controller revisited

Design patterns in a MVC architecture

Client sees

Composite

Client uses

Strategy

View updates

Controller manipulates

Model
Model-View-Controller revisited

Design patterns in a MVC architecture

Client

- sees
- uses

View

- updates

Controller

- manipulates

Model

???
Observer pattern

Observer pattern
From Wikipedia, the free encyclopedia

The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.

- 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)

<<interface>>

Observer

+ update()

MyObserver

+ update()

public void stateChanged() {
    for (Observer o : observers) {
        o.update();
    }
}

// For the setState method, use the stateChanged method
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

1) Which is the Observable? Model or View

2) Which is the Observer? Model or View

3) Which class should use the getState method
   Model, View, or Controller

4) Which class should use the setState method?
   Model, View, or Controller
Model-View-Controller revisited

Design patterns in a MVC architecture

Client

View

Controller

Model

Composite

Strategy

Observer