CS 520
Theory and Practice of Software Engineering
Fall 2017

OO design patterns

September 28, 2017
Logistics

Homework 1

- **deadline:** 10/17/2017.

Paper reading 1 *(Practical guide to statistical tests)*

- **deadline:** 10/04/2017.
- 2 students per group *(group self-selection)*.

Paper reading 2 *(Internal and external validity)*

- **deadline:** 10/11/2017.
- 2 students per group *(group self-selection)*

Course website, Moodle, and Piazza

Deadlines and assignments are posted on course website and Moodle.
Logistics

Homework 1

● **deadline:** 10/17/2017.
● **goal:** Code review, (re)design, and (re)implementation.
Today

- Recap: Composite vs. decorator pattern.
- Behavioral design patterns
  - Template method pattern
  - Strategy pattern
  - Observer pattern
- Live coding.
Recap: Composite vs. Decorator

<<interface>>
Component
+operation()

1..n

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

CompA
+ operation()

1

Decorator
- decorated: Component
+ operation()
+ Decorator(d: Component)
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 \( n \)  \hspace{1cm} \text{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]) {
                ...   
            swapped = true;
        }
    }
    if (n%2 == 0) {
        return (numbers[(n>>1) - 1] + numbers[n>>1]) / 2;
    } else {
        return numbers[n>>1];
    }
}
public static void main(String ... args) {
    System.out.println(median(1,2,3,4,5));
}

class MedianComputation {
    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>>1) - 1] + numbers[n>>1]) / 2;
        } else {
            return numbers[n>>1];
        }
    }
}

What’s wrong with this design? How can we improve it?
Live coding: naïve solution

See code examples (online)

- naive
  - 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.

Use `ant` to compile and test the code:

- `$ant -p` => list all targets
- `$ant compile` => compile the code
- `$ant test` => run all tests
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.
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.

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

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

Should the median method be final?
Live coding: template method pattern

See code example (online)

- template
  - Abstract superclass with template method and abstract method `sort`.
  - Two subclasses with concrete implementations for the method `sort`.

Use `ant` to compile and test the code:

- `$ant -p` => list all targets
- `$ant compile` => compile the code
- `$ant test` => run all tests (note the run-time differences between SimpleMedianTest and QuickMedianTest!)
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(...)
```
Another solution: strategy pattern

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

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

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

```
HeapSort
+sort(...)
```

```
QuickSort
+sort(...)
```

"median" delegates the sorting of the array to a "sortStrategy"
Live coding: strategy pattern

See code example (online)

- **strategy**
  - Interface *Sorter* for sorting strategies that defines the method *sort*.
  - Two implementations of this interface (*BubbleSort* and *QuickSort*).
  - *StrategyMedian* delegates the sorting to a sorting strategy, which can be configured and changed at run time.

Use `ant` to compile and test the code:

$ ant -p => list all targets
$ ant compile => compile the code
$ ant test => run all tests (note how testSwapSorter in StrategyMedianTest changes the sorter at run time!)
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.
- Don’t call us, we’ll call you.
- Template method is usually final.

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

Design patterns in an MVC architecture

View sees Controller uses
View updates Model manipulates
MVC revisited

Design patterns in an MVC architecture

- **Client**
  - sees
  - uses

- **View**
  - updates

- **Controller**
  - manipulates

- **Model**

- **Composite**

- **Strategy**
MVC revisited

Design patterns in an MVC architecture

Client

View

Controller

Model

sees

uses

updates

manipulates

???
Observer pattern

- Models a “one to many” dependency.
- Decouples state and action:
  Notify registered observer(s) about state change.
Observer pattern

**Observable**

{abstract}

- # observers:Set<Observer>
- + register(o:Observer)
- + unregister(o:Observer)
- + stateChanged()

**MyObservable**

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

**Observer**

- + update()

**MyObserver**

- + update()
Observer pattern

**Observable**

{abstract}

- # observers: Set<Observer>
- + register(o: Observer)
- + unregister(o: Observer)
- + stateChanged()

**MyObservable**

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

**Observer**

<<interface>>

- + update()

**MyObserver**

- + update()

Variation: pass incremental changes or the state to update method.
Live coding: observer pattern

See code example (online)

- observer
  - One observable data model (*ObservableModel*).
  - Two observers (*PrintNumbers* and *NegateNumbers*).
  - A state change in the data model leads to a notification of all registered observers.

Use `ant` to compile and test the code:

- `$ant -p` => list all targets
- `$ant compile` => compile the code
- `$ant run` => simulate state changes (note how adding and removing observers changes the output!)