# CS 520

Theory and Practice of Software Engineering
Fall 2019

**Object Oriented Design Patterns**

September 19, 2019

## Thursday (September 26)

- First in-class exercise
- On using git (Tuesday will be a prelude with useful info)
- Form 4-person teams
  - Use moodle to self-select a team; can do it before Thursday or on Thursday
- At least one person per team needs to bring a laptop

**BRING A LAPTOP!**

## Today

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

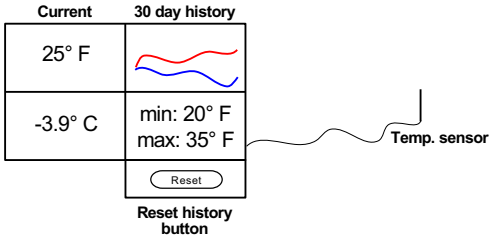
## Recap

**Object oriented design principles**

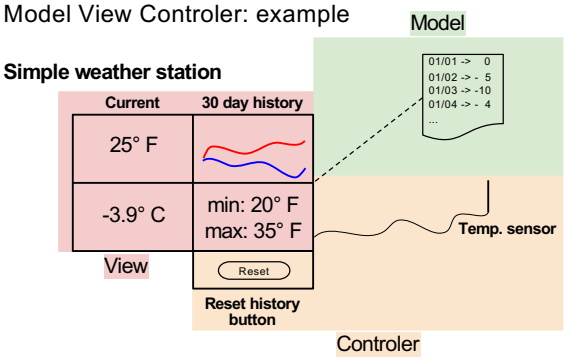
- Information hiding (and encapsulation)
- Open/closed principle
- Liskov substitution principle
- Composition/aggregation over inheritance

## A first design problem

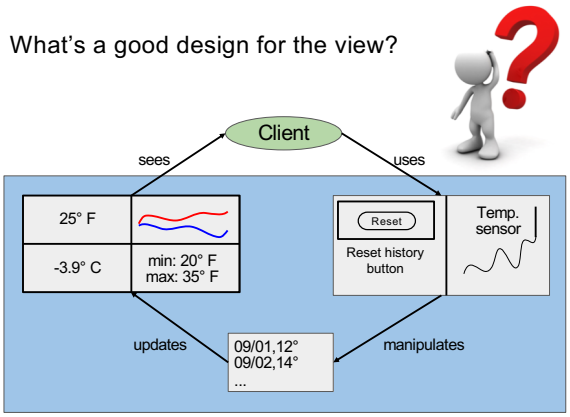
**Weather station revisited**



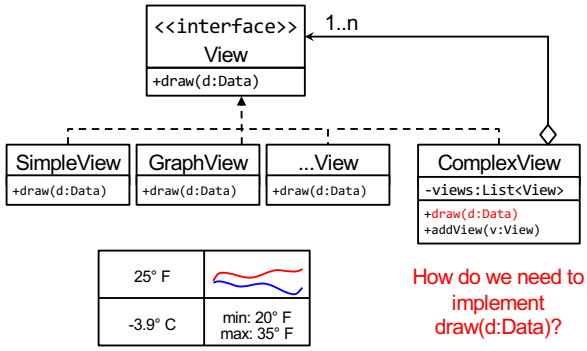

## Model View Controler: example



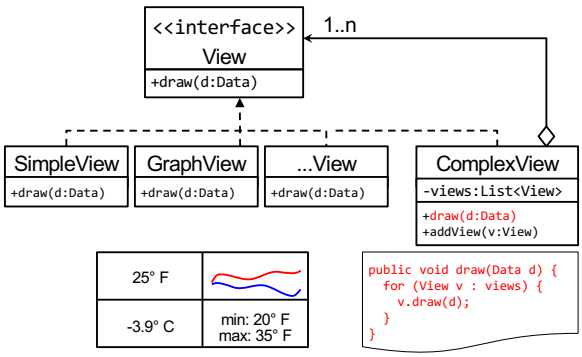
## What's a good design for the view?



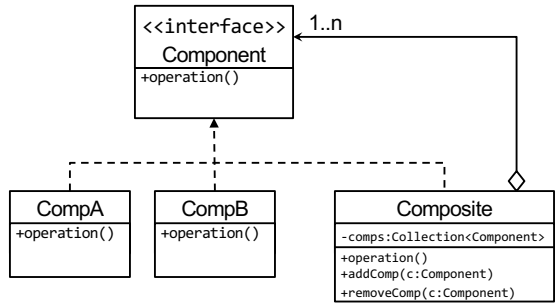

## Weather station: view



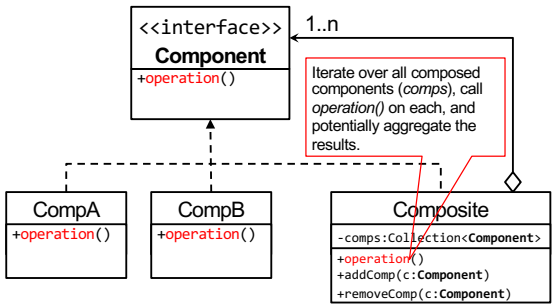

## Weather station: view




## 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.

## 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**

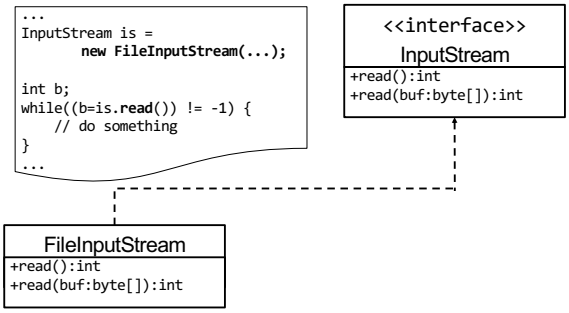
- Risk of over-engineering.
- Potential impact on system performance.

More than just a name for common sense and best practices.

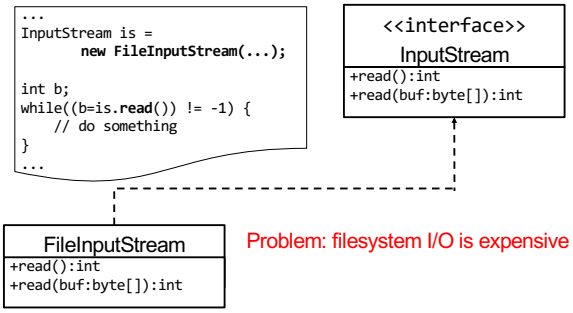
## Design patterns: categories

1. Structural
   - Composite
   - Decorator
   - ...
2. Behavioral
   - Template method
   - Visitor
   - ...
3. Creational
   - Singleton
   - Factory (method)
   - ...

## Another design problem: I/O streams

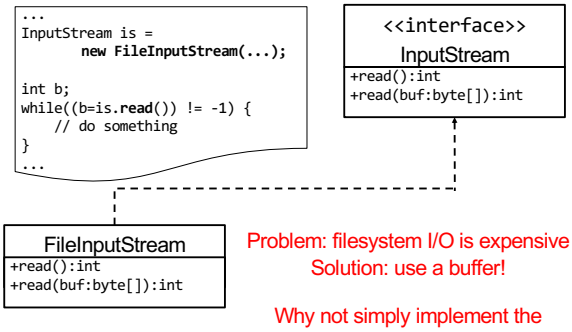



## Another design problem: I/O streams




Problem: filesystem I/O is expensive

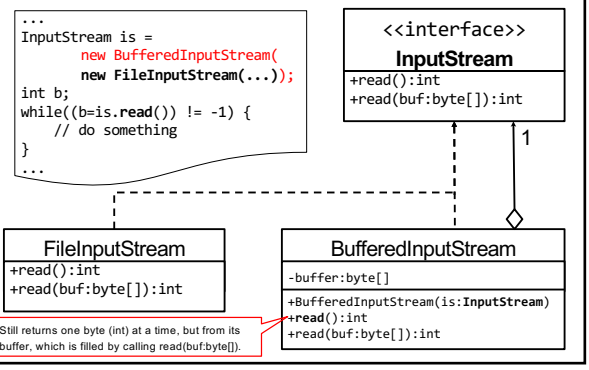
## Another design problem: I/O streams



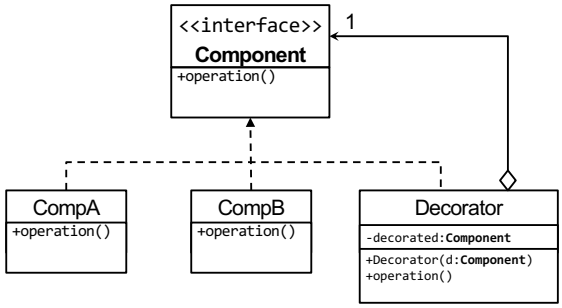

Problem: filesystem I/O is expensive
Solution: use a buffer!

Why not simply implement the buffering in the client or subclass?

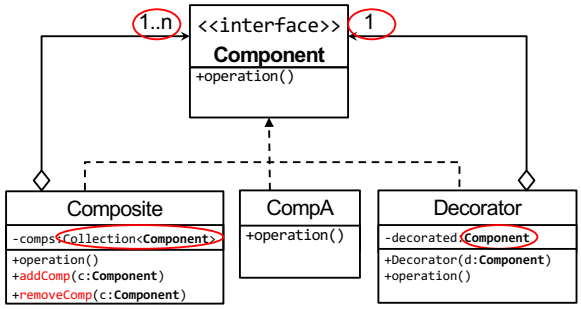
## Another design problem: I/O streams



## 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

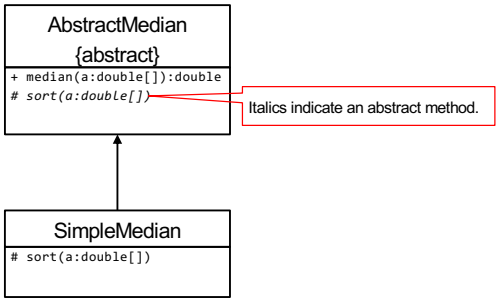
```
public static void main(String ... args) {
        System.out.println(median(1,2,3,4,5));
}

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?

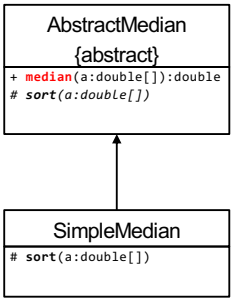
## Ways to improve

- 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

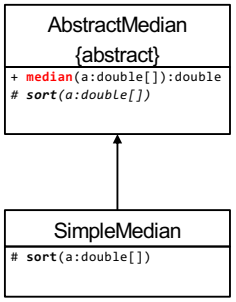



## One possible solution: template method pattern




- 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**.

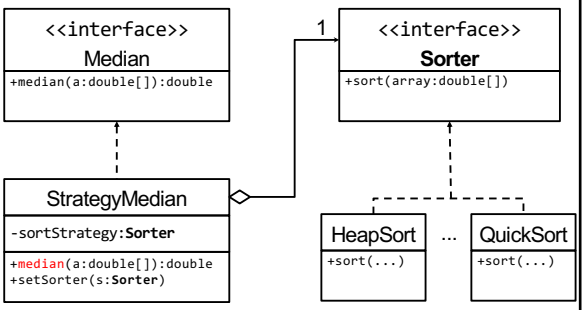
## One possible solution: template method pattern




Should the median method be final?

- 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




"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

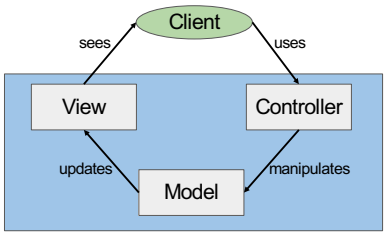
**Two solutions to the same problem**

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

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

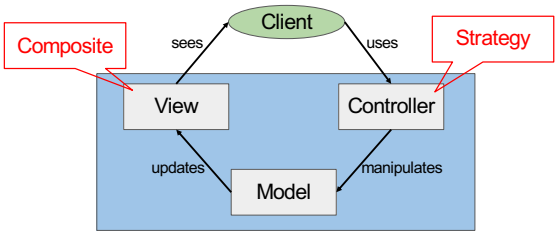
## Model-View-Controller revisited

**Design patterns in a MVC architecture**



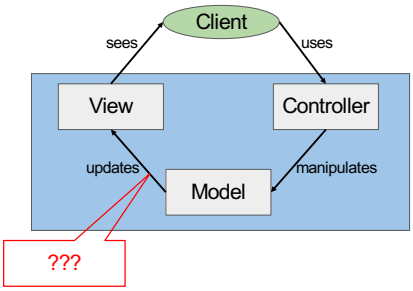

## Model-View-Controller revisited

**Design patterns in a MVC architecture**




## Model-View-Controller revisited

**Design patterns in a MVC architecture**




## Observer pattern

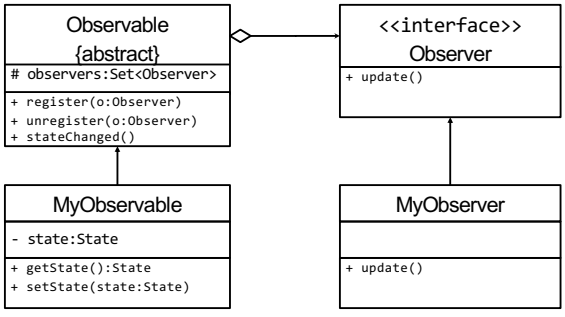
Observer pattern

From Wikipedia, the free encyclopedia

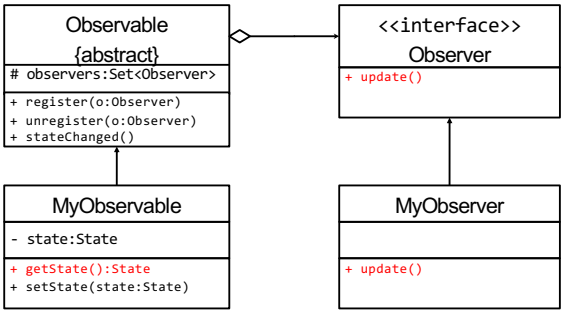
The **observer pattern** is a software design pattern in which an object, called the **subject**, maintains a list of its dependents, called **observers**, and notifies them automatically of any state changes, usually by calling one of their methods.

- 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



## Observer pattern




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

## Model-View-Controller revisited

**Design patterns in a MVC architecture**