

CS 520

Theory and Practice of Software Engineering
Spring 2021

Object Oriented Design Patterns

February 16, 2021

Logistics

- **Office hours:** Mondays 1 – 2 PM, Thursdays 30 minutes after lecture, Fridays 9:30 – 10 AM
- **Second participation questionnaire** posted and due this Thursday if you decide to submit it
- **First homework assignment** applying architecture, design, and best programming practices will be posted this Thursday

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

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? **Vector, Serializable, Cloneable, Iterable, Collection, List, RandomAccess**
- 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?

Open/Closed and information hiding principles

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

`capacityIncrement`, `elementCount`, `elementData`, `modCount`

4) What is a Stack method that is open only for extensions?

`removeRange`

Liskov substitution principle



Subtype requirement

Let object x be of type $T1$ and object y be of type $T2$. Further, let $T2$ be a subtype of $T1$ ($T2 <: T1$). Any provable property about objects of type $T1$ should be true for objects of type $T2$.

- Supertype $T1$ is Vector // It is a List.
- Subtype $T2$ 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**?

Liskov substitution principle

Subtype requirement

Let object x be of type $T1$ and object y be of type $T2$. Further, let $T2$ be a subtype of $T1$ ($T2 <: T1$). Any provable property about objects of type $T1$ should be true for objects of type $T2$.

- Supertype $T1$ is Vector // It is a List.
- Subtype $T2$ 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?

Composition/Aggregation over inheritance

Another example:

<https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html>

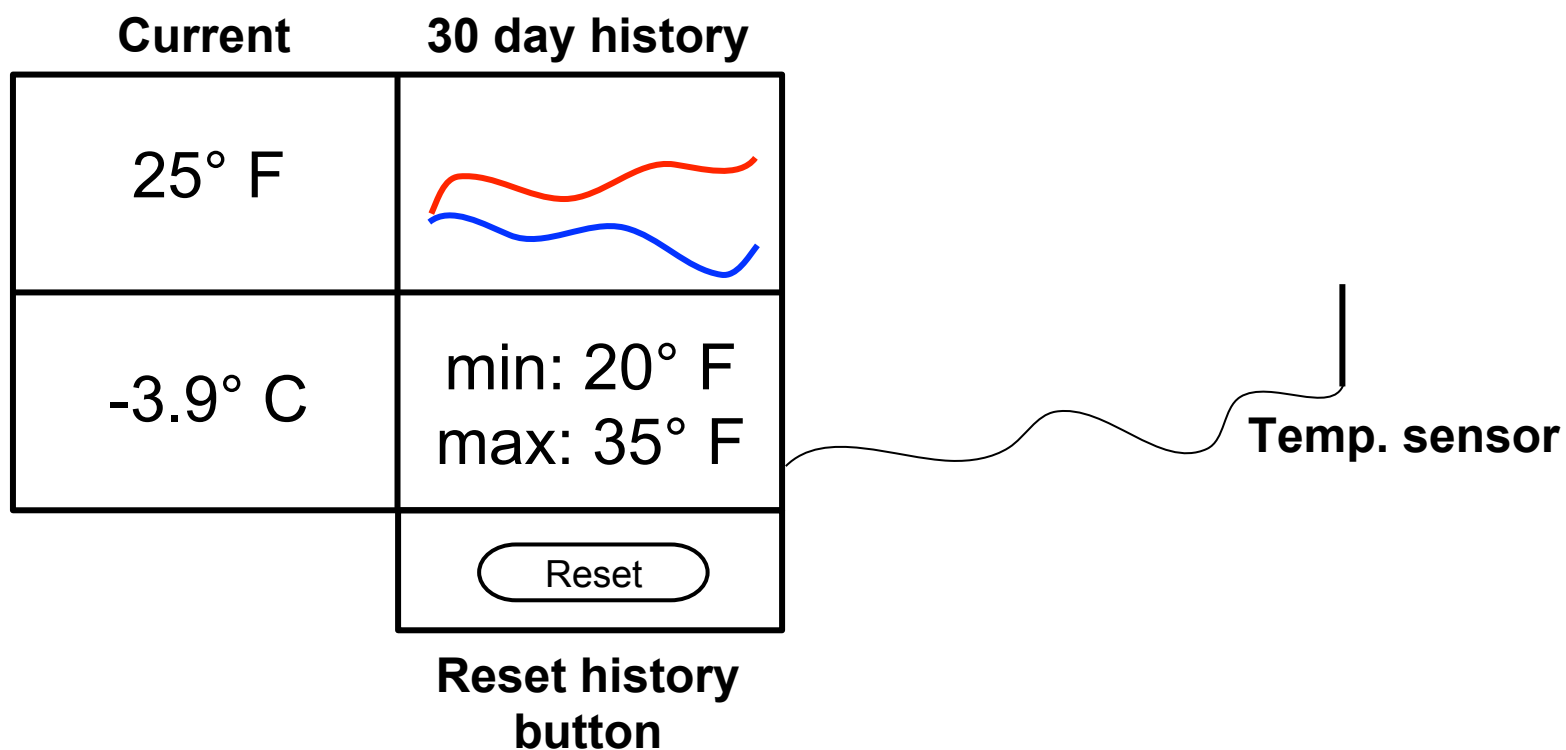
- From this class name, the two parts of the functionality are providing Map access and Linked access.
- How is the class implementing the Map access?
Extends HashMap (i.e. is-a HashMap)
- How is the class implementing the Linked access?
Declares LinkedList fields (i.e. has-a LinkedList)

Design patterns

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

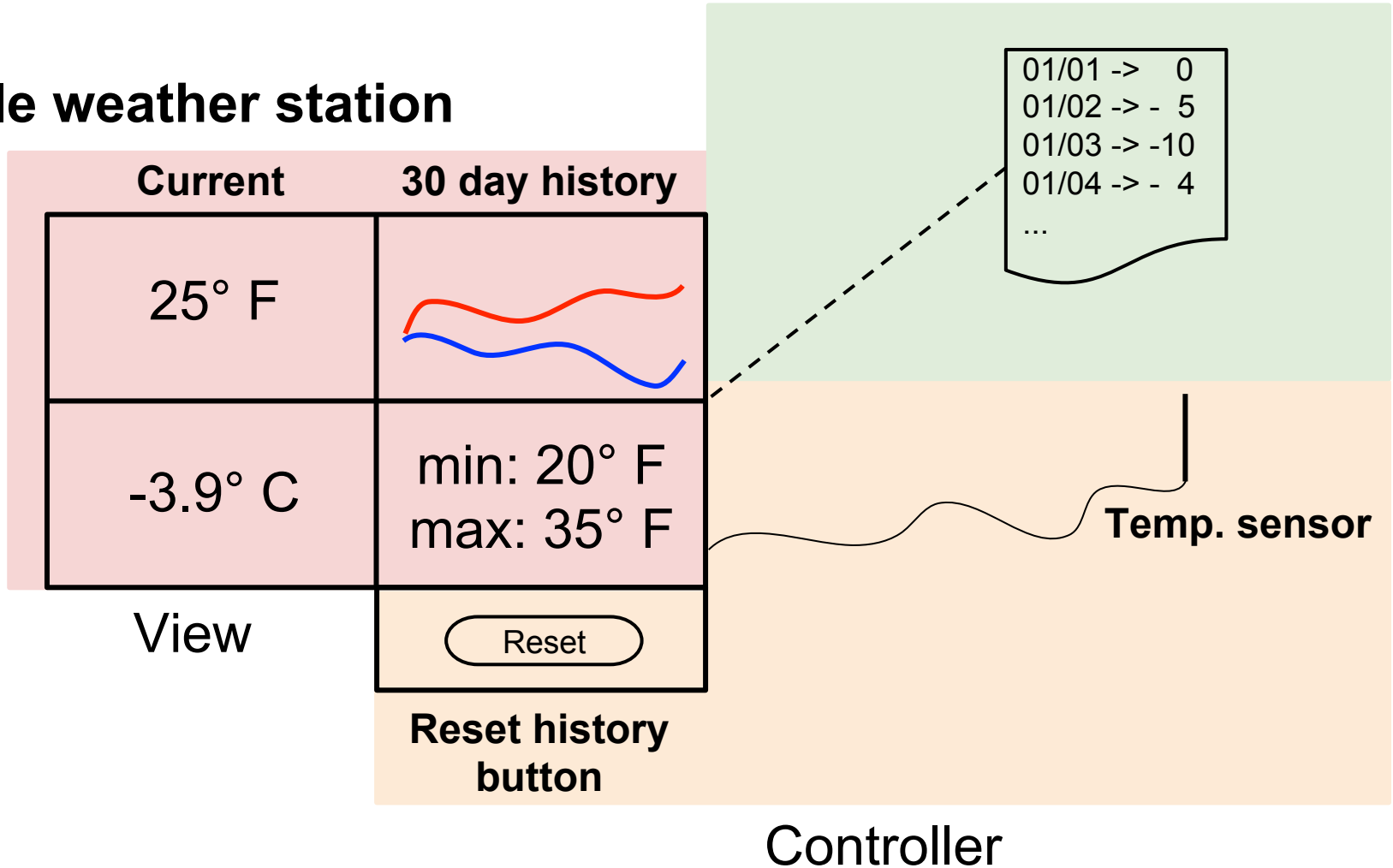
A first design problem

Weather station revisited

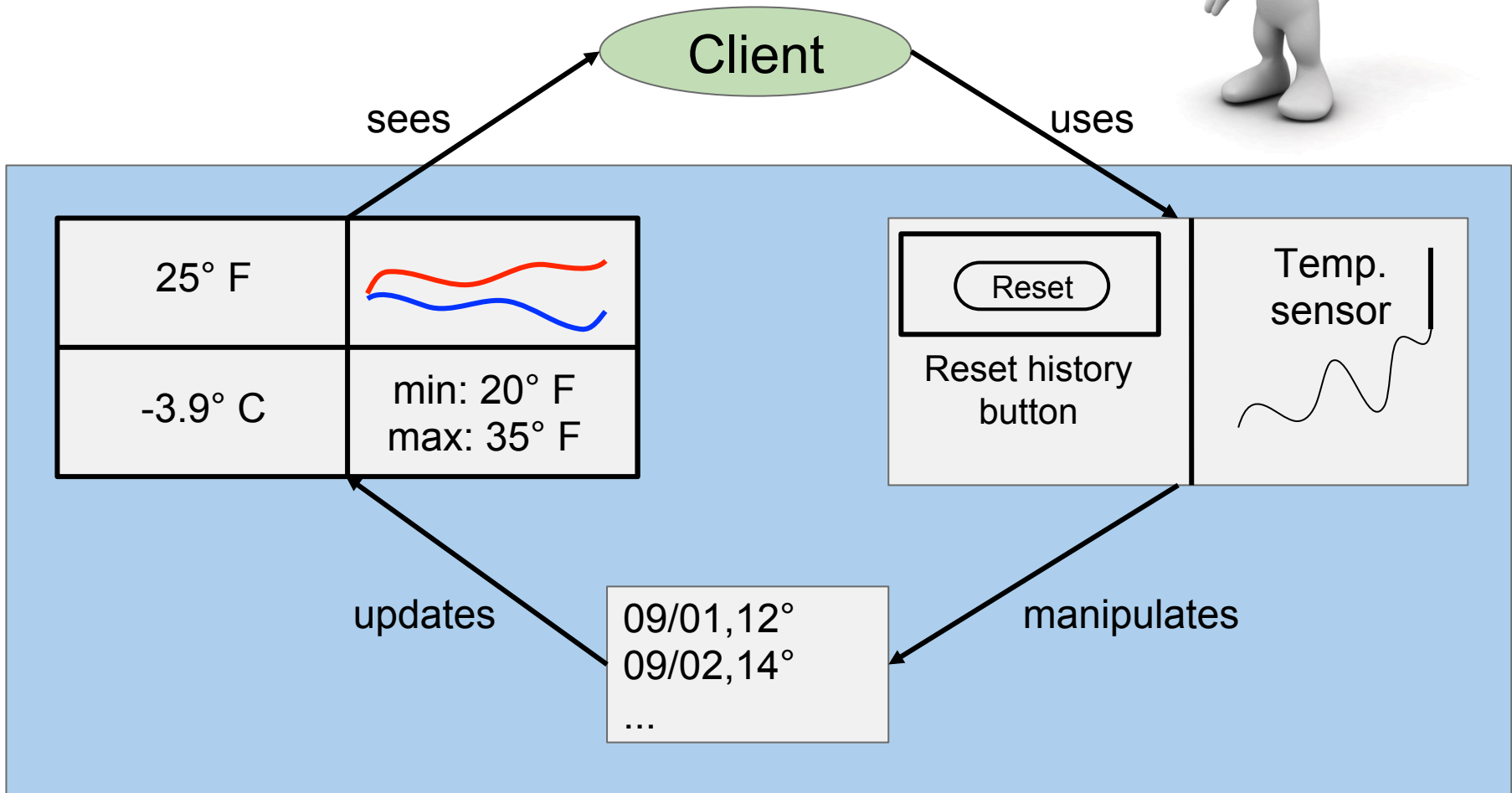


Model View Controller: example

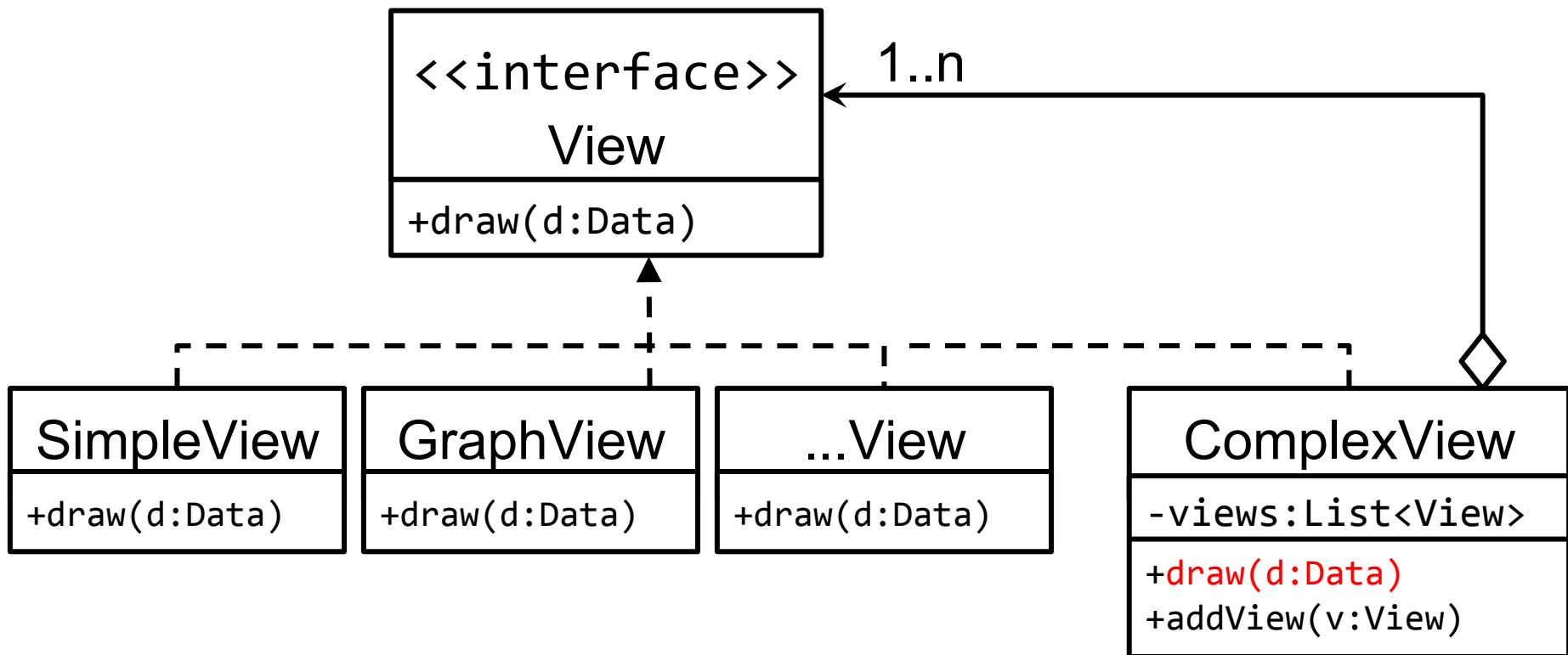
Simple weather station




What's a good design for the view?



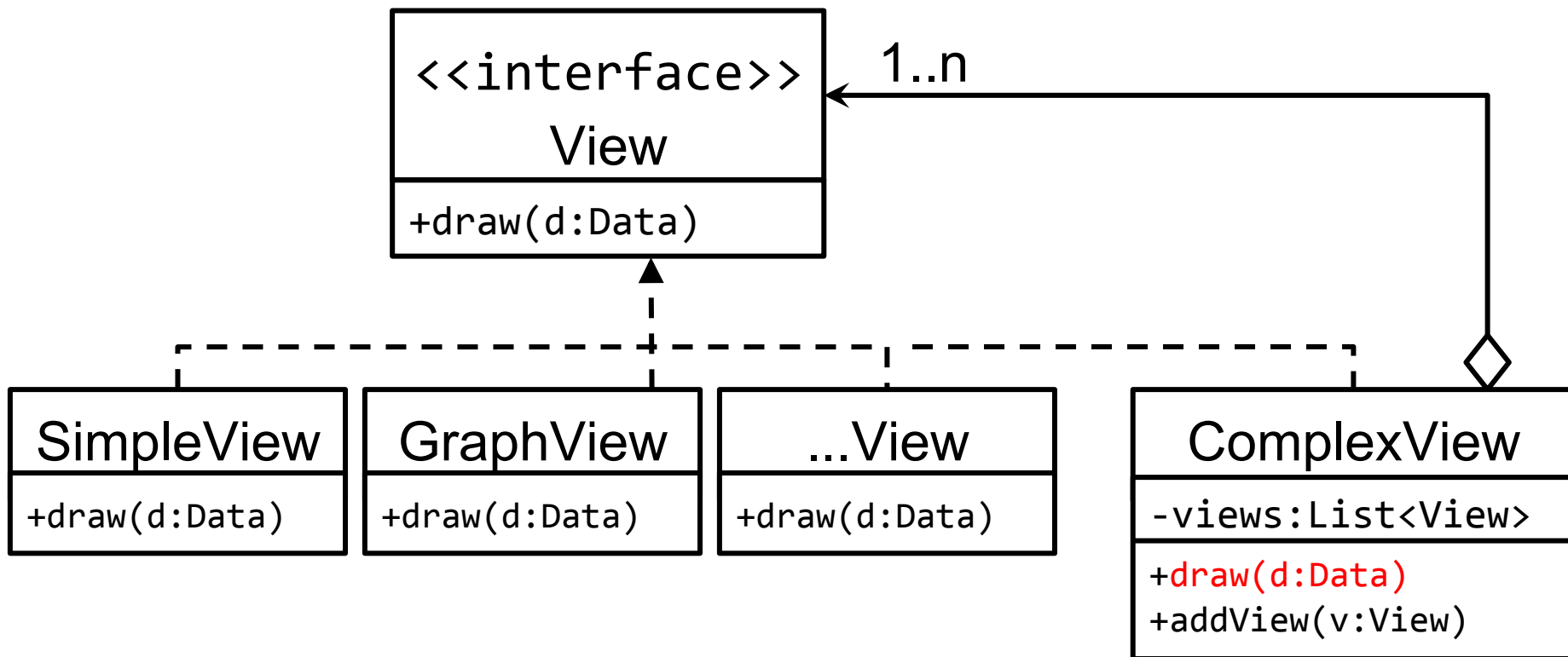
Weather station: view




25° F	
-3.9° C	min: 20° F max: 35° F

How do we need to implement draw(d:Data)?

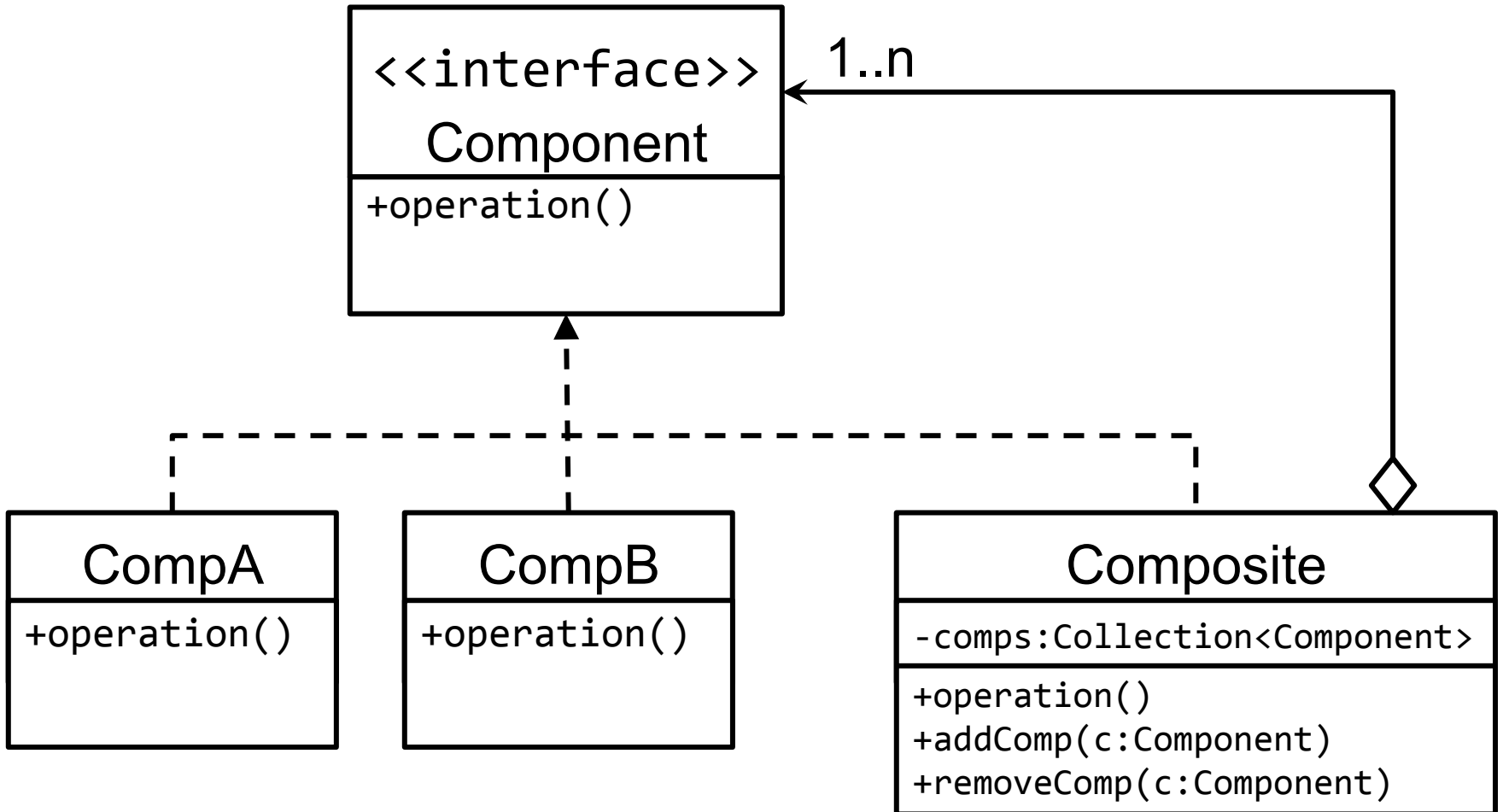
Weather station: view



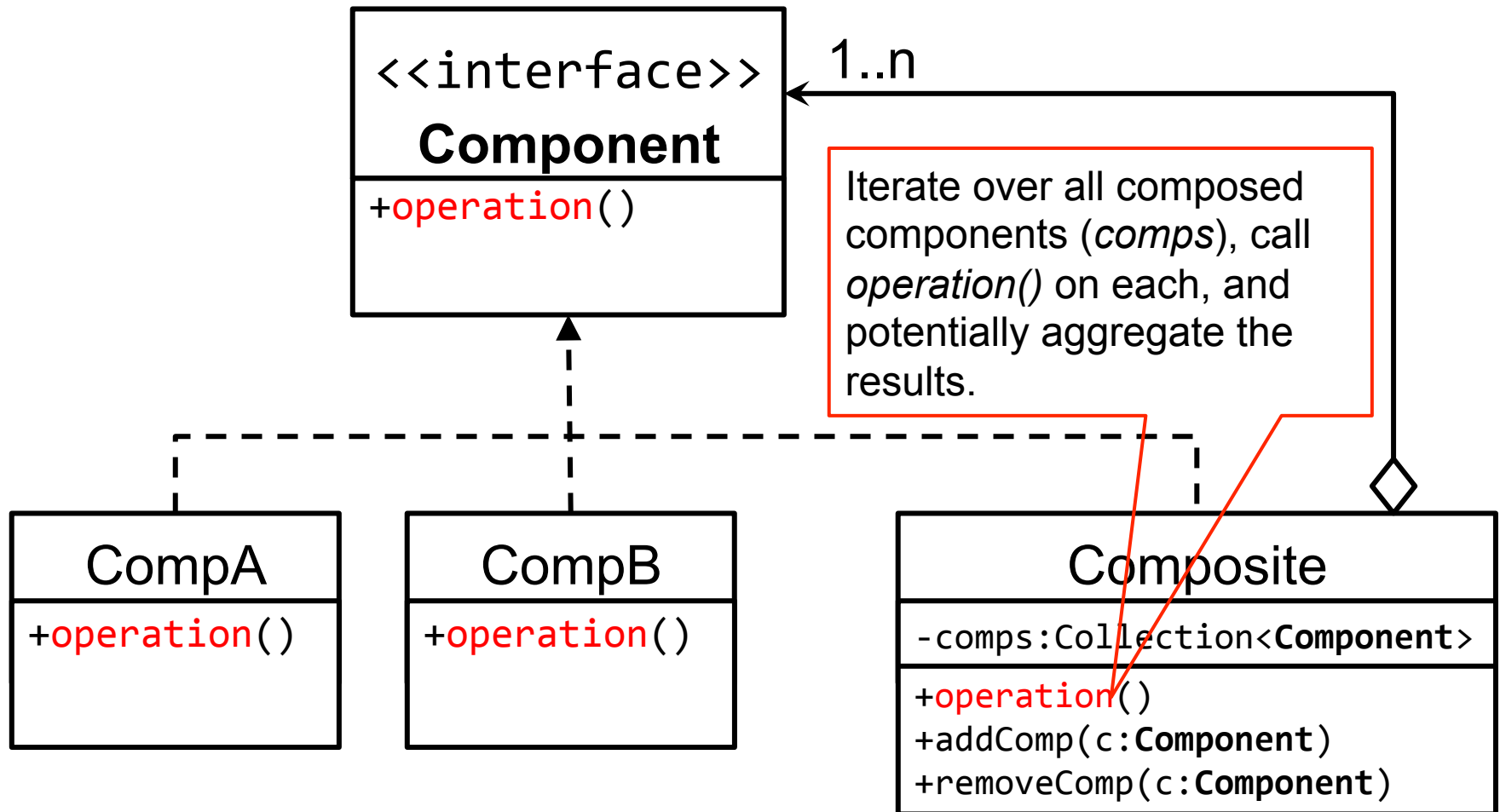
25° F	
-3.9° C	min: 20° F max: 35° F

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

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
- ...

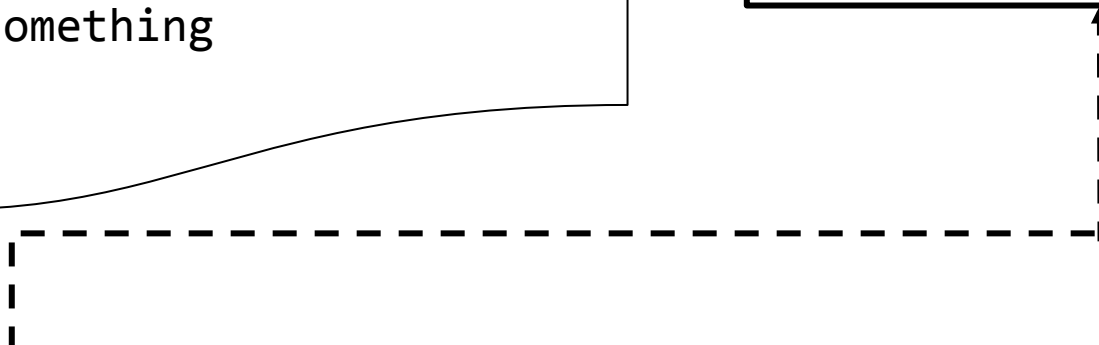
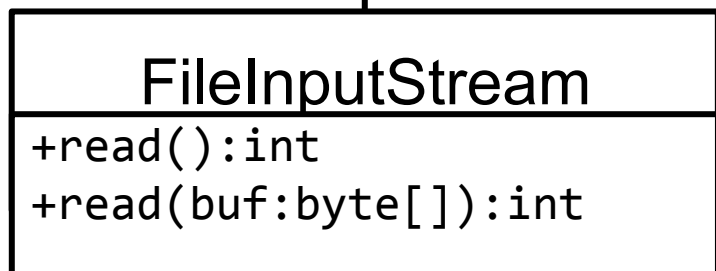
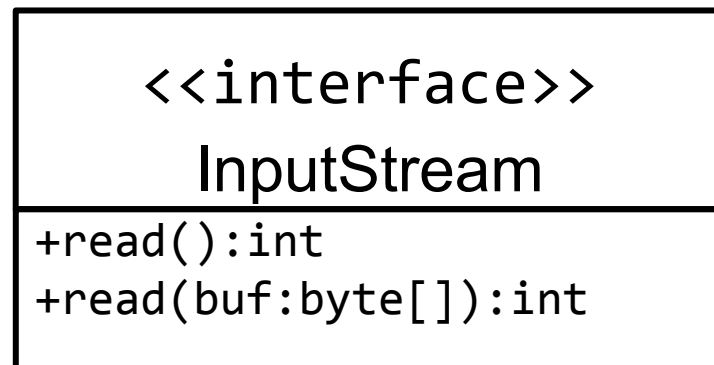
3. Creational

- Singleton
- Factory (method)
- ...

Another design problem: I/O streams

```
...
InputStream is =
    new FileInputStream(...);

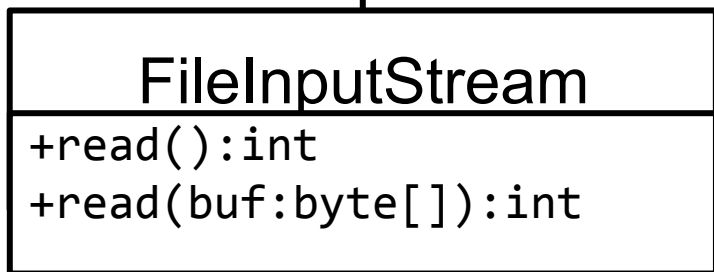
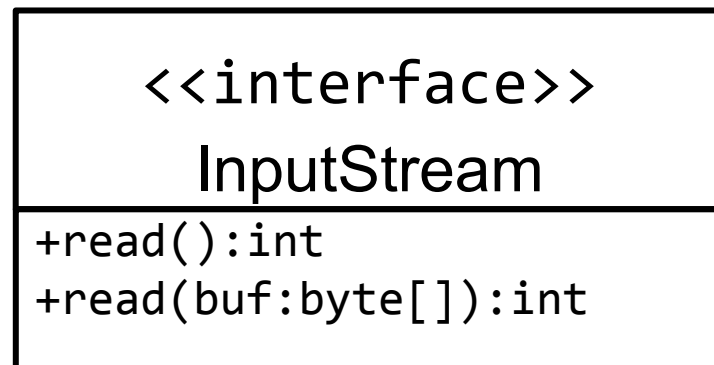
int b;
while((b=is.read()) != -1) {
    // do something
}
...
```



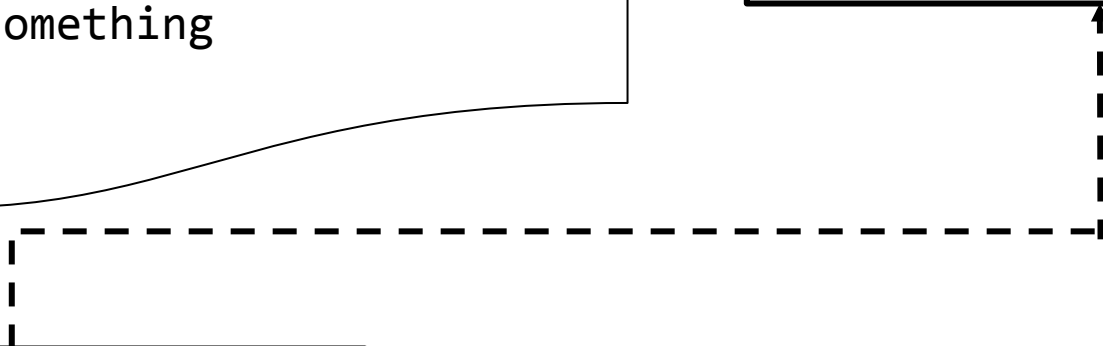
Another design problem: I/O streams

```
...
InputStream is =
    new FileInputStream(...);

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



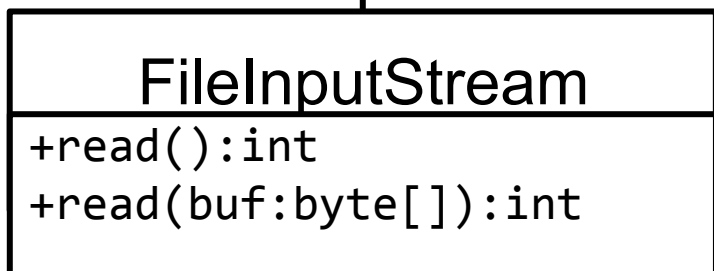
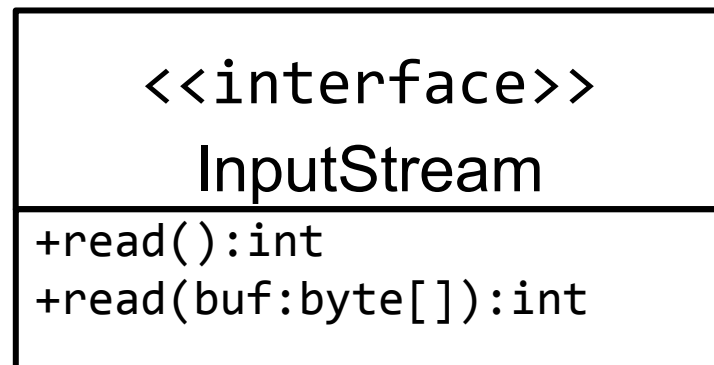
Problem: filesystem I/O is expensive



Another design problem: I/O streams

```
...
InputStream is =
    new FileInputStream(...);

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

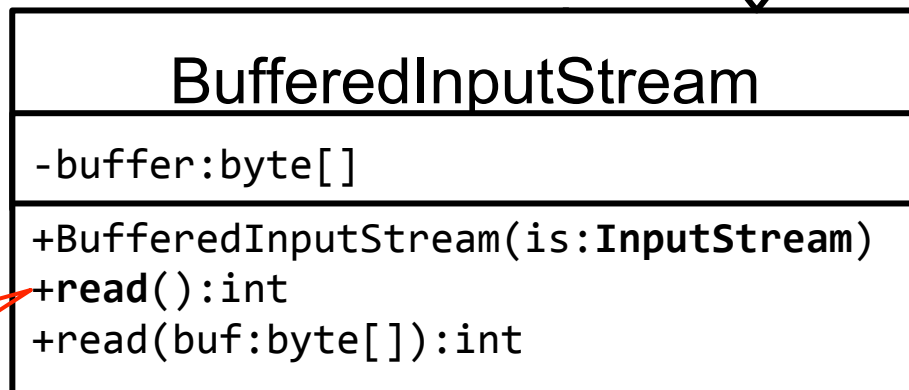
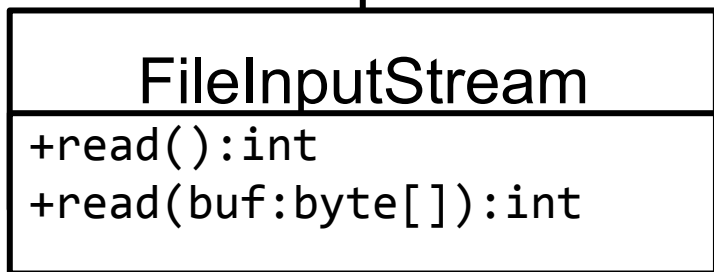
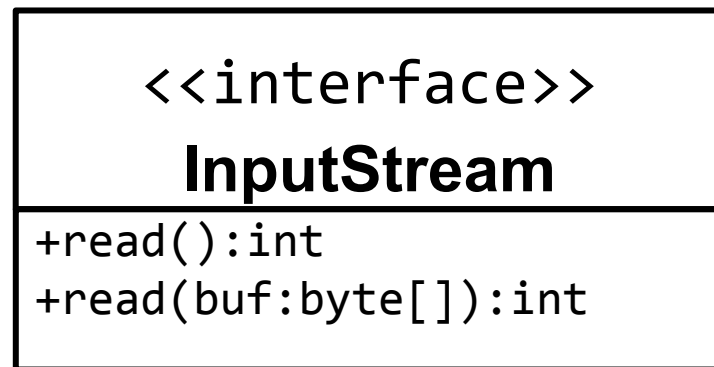


Problem: filesystem I/O is expensive
Solution: use a buffer!

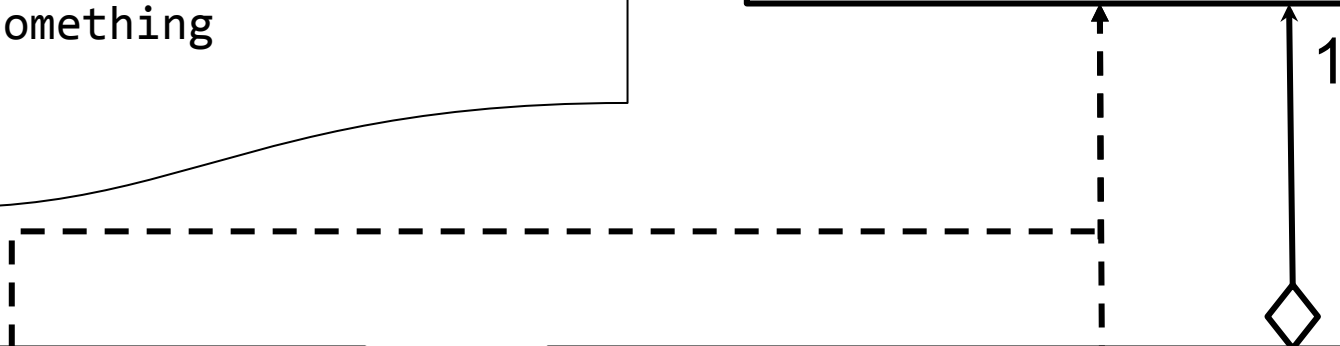
Why not simply implement the buffering in the client or subclass?

Another design problem: I/O streams

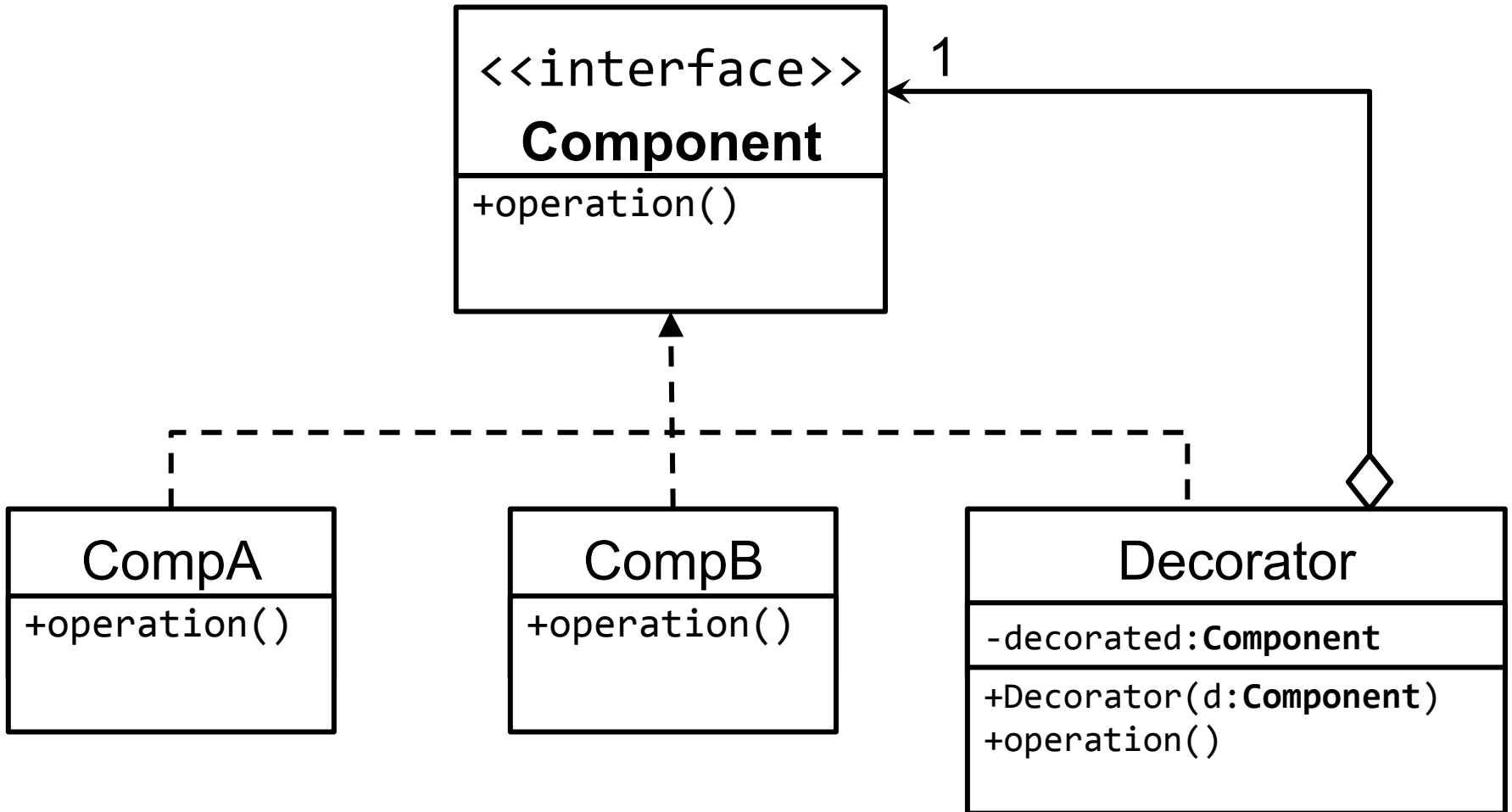
```
...
InputStream is =
    new BufferedInputStream(
        new FileInputStream(...));
int b;
while((b=is.read()) != -1) {
    // do something
}
...
```



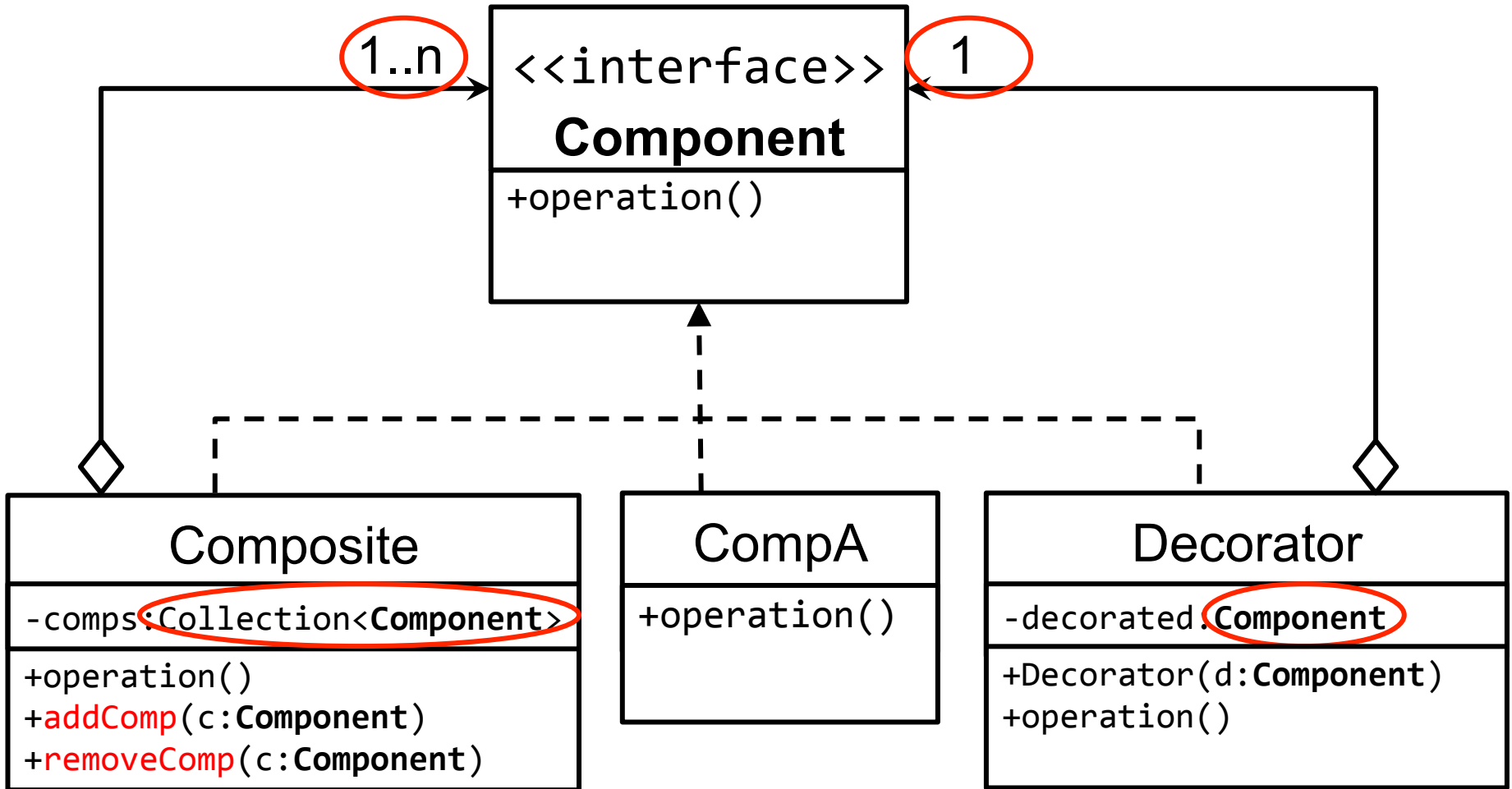
Still returns one byte (int) at a time, but from its buffer, which is filled by calling read(buf:byte[]).



Design pattern: Decorator



Composite vs. Decorator



Find the median in an array of doubles

Examples:

- $\text{median}([1, 2, 3, 4, 5]) = ???$
- $\text{median}([1, 2, 3, 4]) = ???$



Find the median in an array of doubles



Examples:

- $\text{median}([1, 2, 3, 4, 5]) = 3$
- $\text{median}([1, 2, 3, 4]) = 2.5$

Algorithm

Input: *array* of length n **Output:** median

Find the median in an array of doubles

Examples:

- $\text{median}([1, 2, 3, 4, 5]) = 3$
- $\text{median}([1, 2, 3, 4]) = 2.5$

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));
}

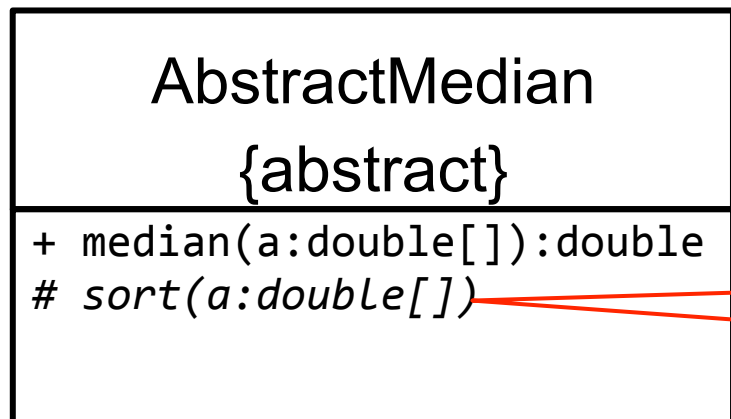
public static double median(double ... numbers) {
    int n = numbers.length;
    boolean swapped = true;
    while(swapped) {
        swapped = false;
        for (int i = 1; i<n; ++i) {
            if (numbers[i-1] > numbers[i]) {
                ...
                swapped = true;
            }
        }
    }
    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

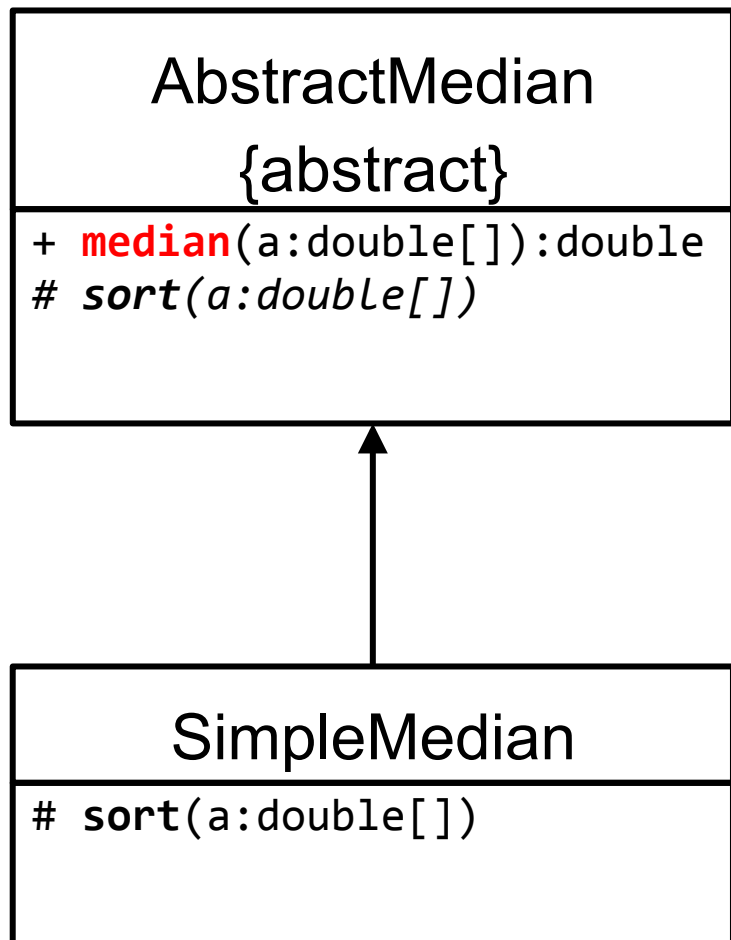
- 1: Monolithic version, static context.
- 2: Extracted sorting method, non-static context.
- 3: Proper package structure and visibility, extracted main method.
- 4: Proper testing infrastructure and build system.

One possible solution: **template method pattern**



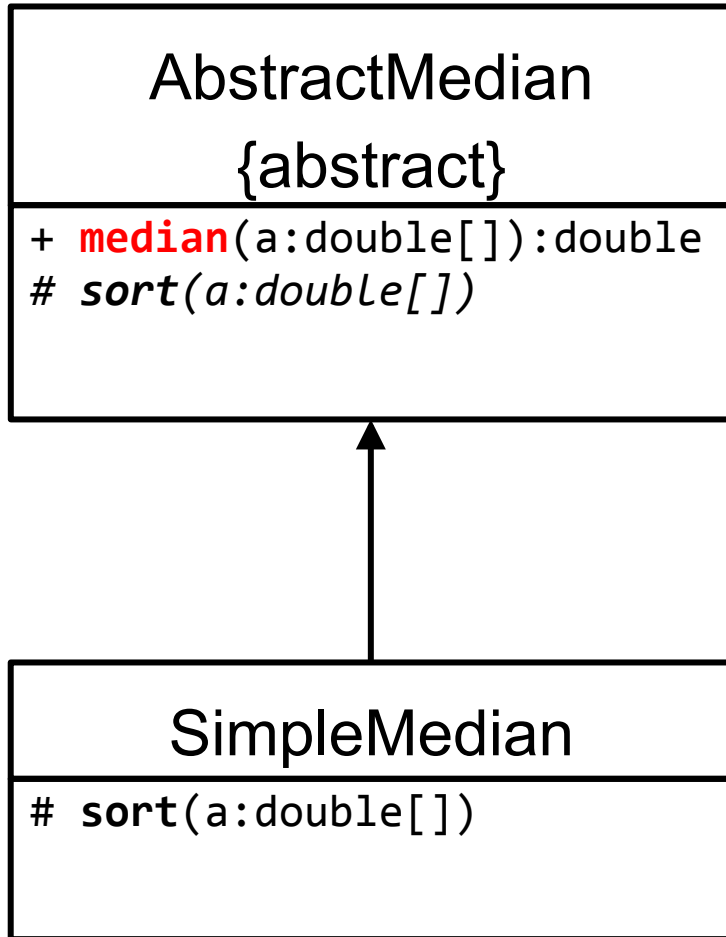
Italics indicate an abstract method.

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**.

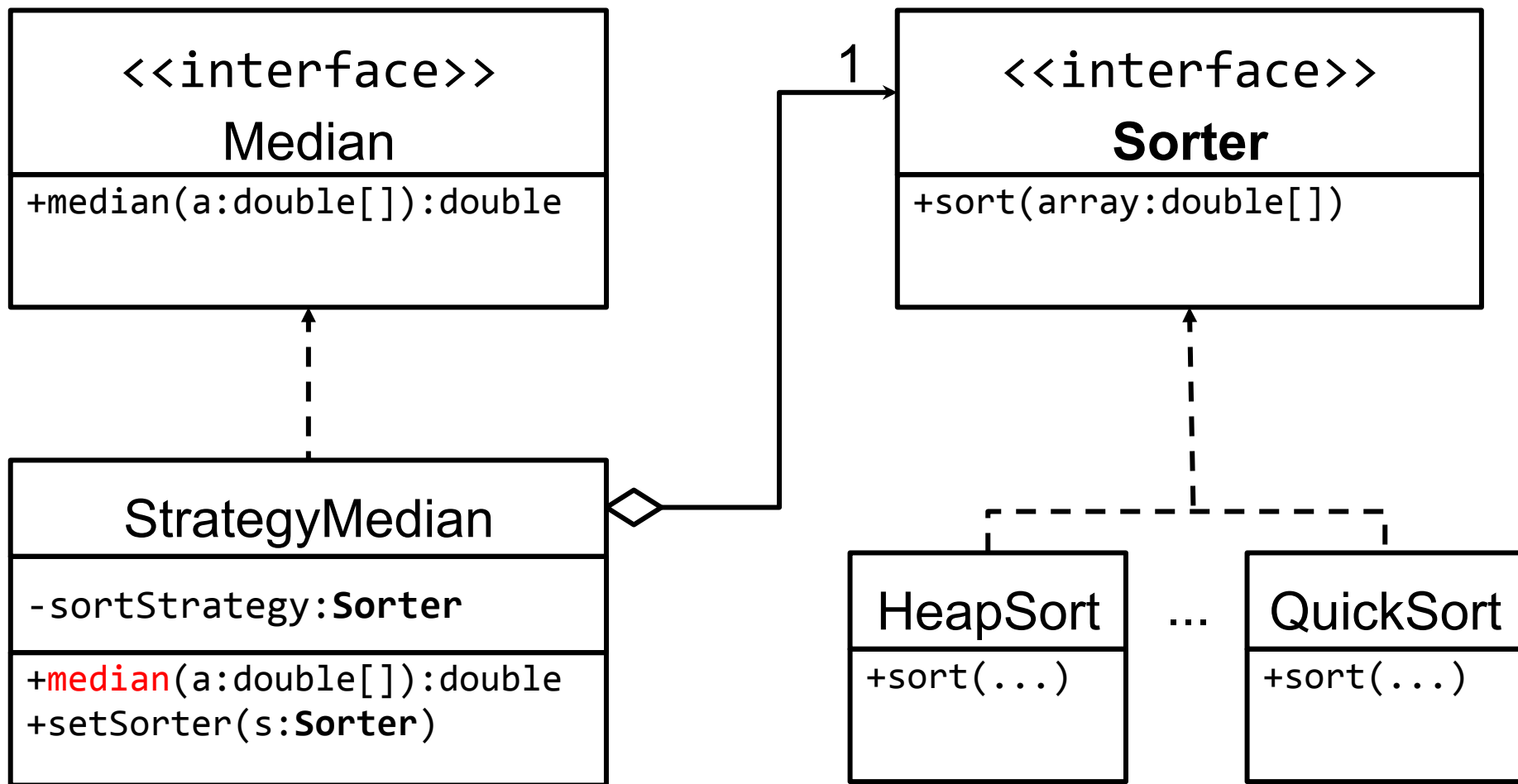
One possible solution: template method pattern



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

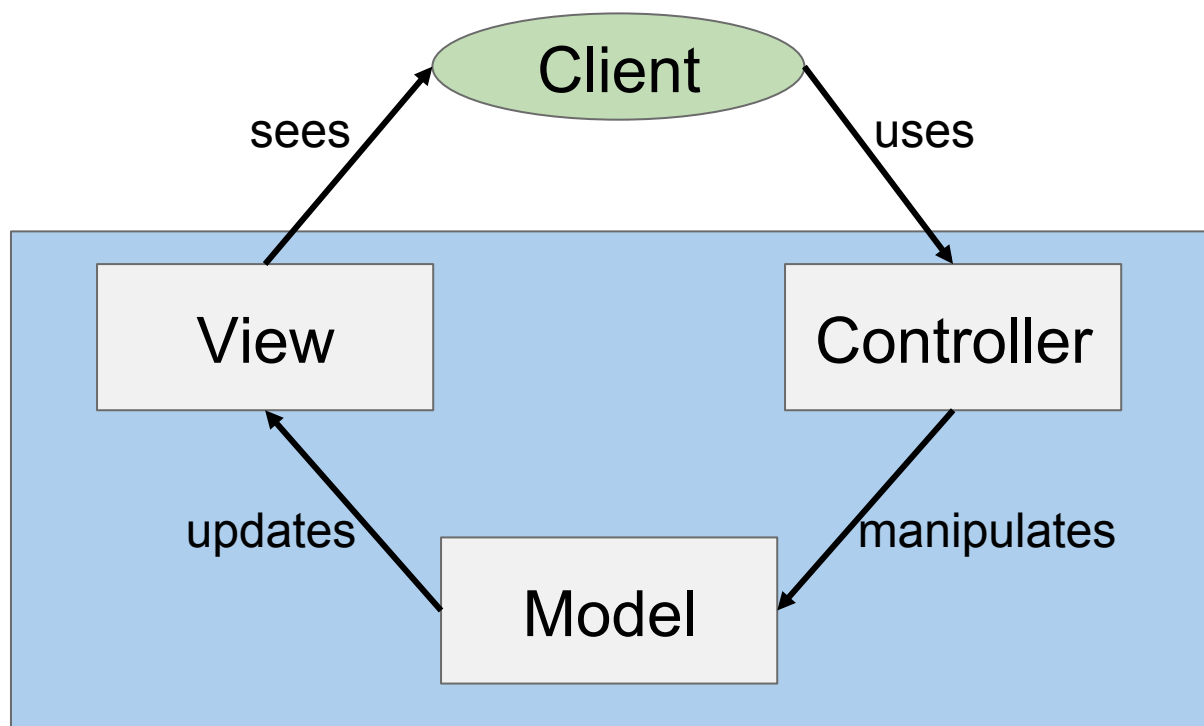
- Behavior selected at compile time.
- Template method is usually final.

Strategy

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

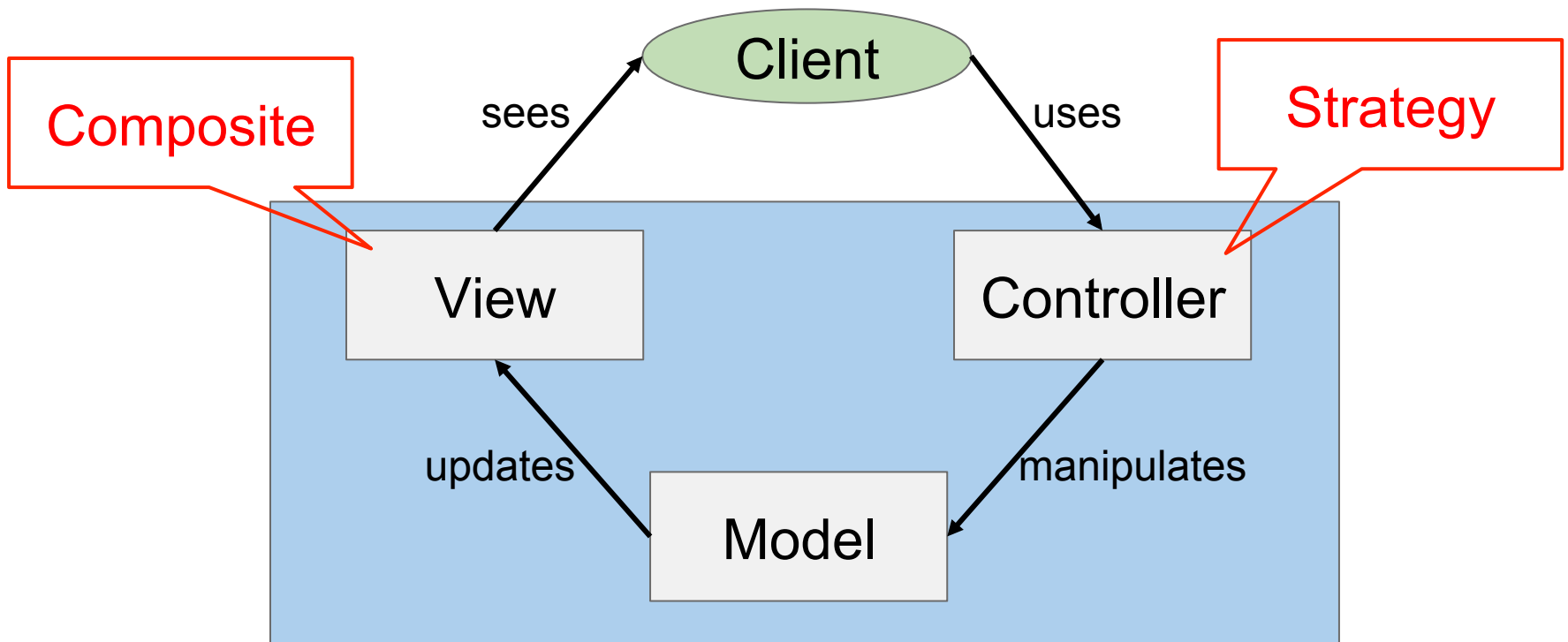
Model-View-Controller revisited

Design patterns in a MVC architecture



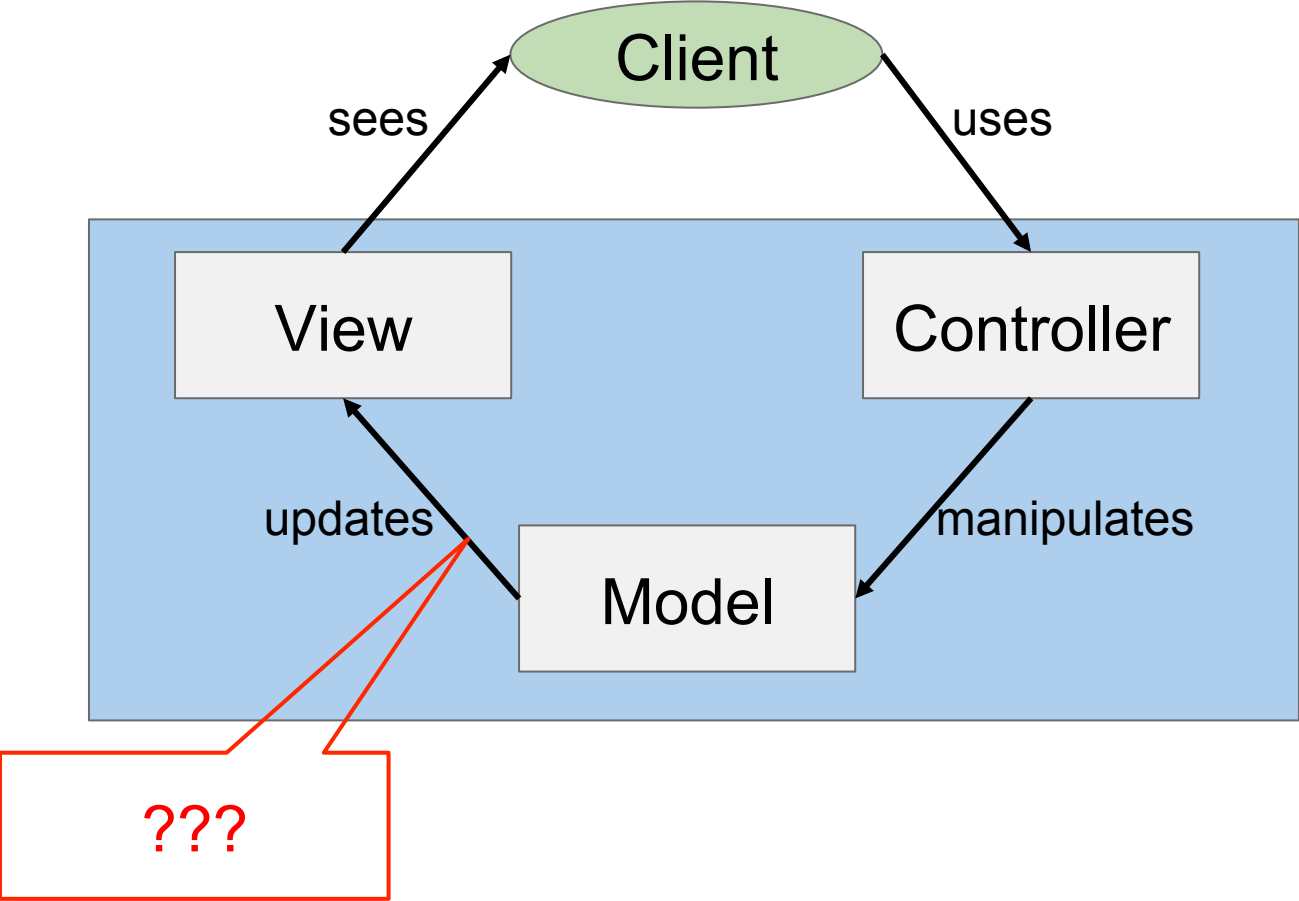
Model-View-Controller revisited

Design patterns in a MVC architecture



Model-View-Controller revisited

Design patterns in a MVC architecture



Observer pattern

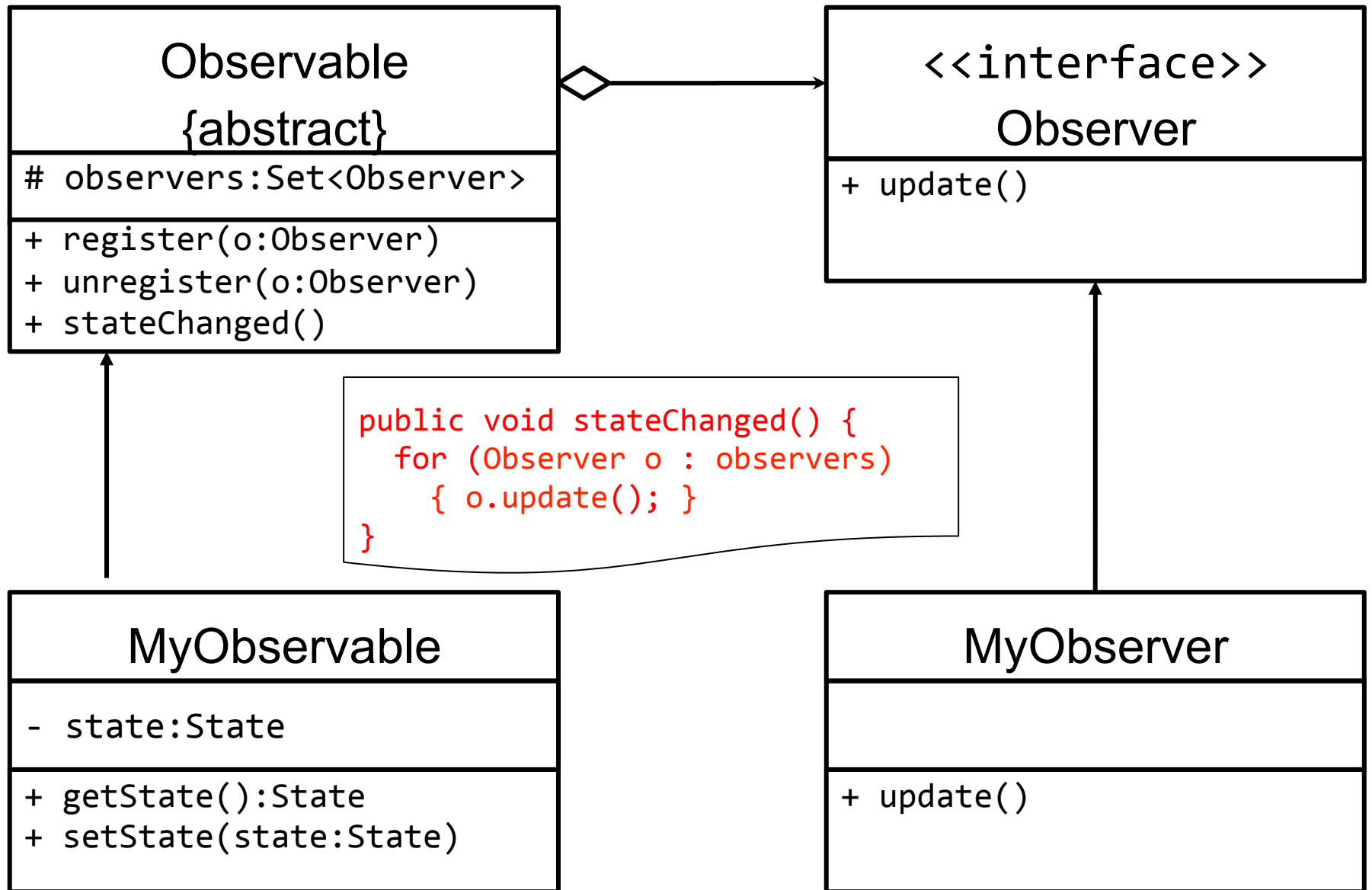
Observer pattern

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



// 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

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

