CS 520

Theory and Practice of Software Engineering
Fall 2017

OO design patterns

September 26, 2017

Today

- MAP.
- Recap: OO design principles.
- Design problems & potential solutions.
- Design patterns:
  - What is a design pattern?
  - Categories of design patterns.
  - Structural design patterns.

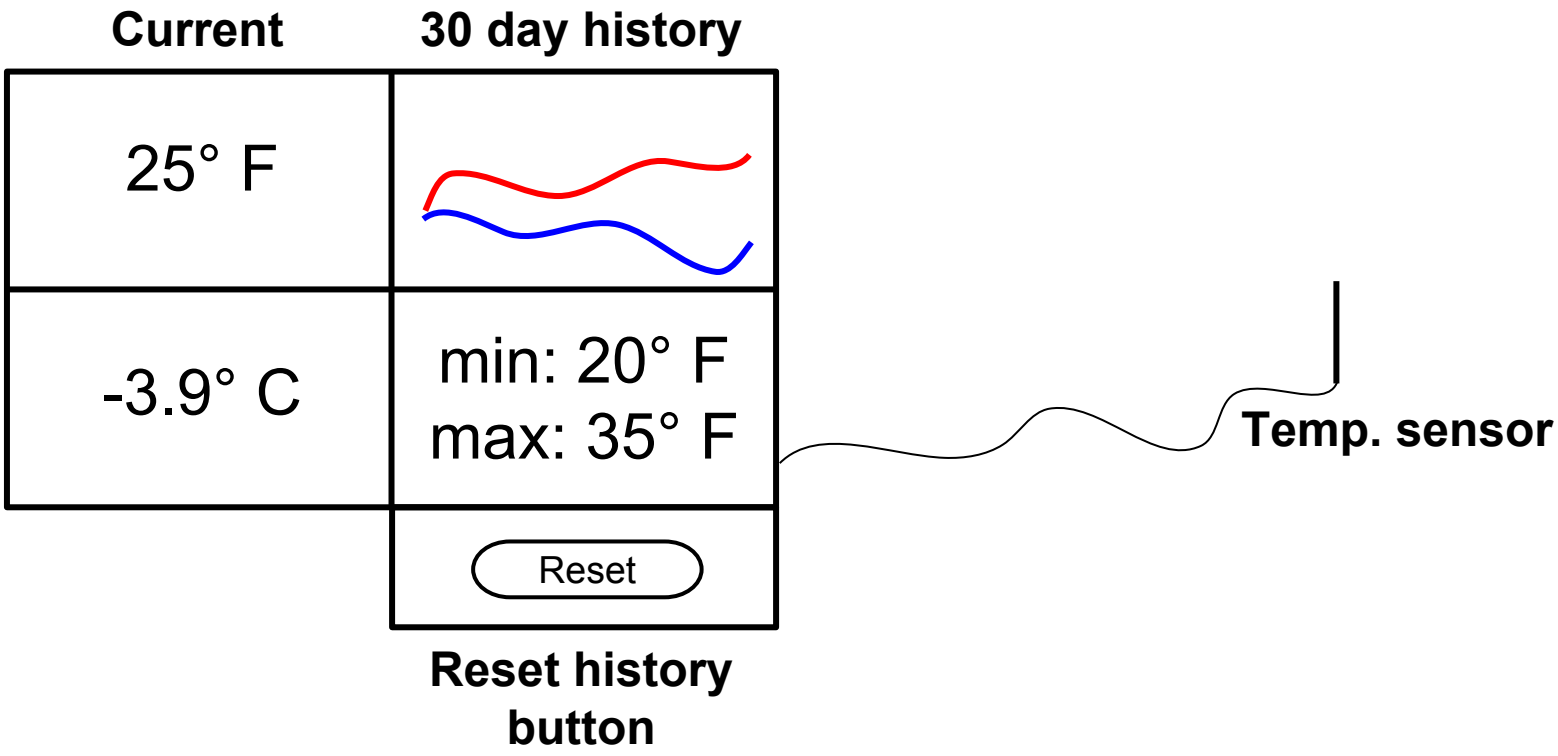
Recap

OO design principles

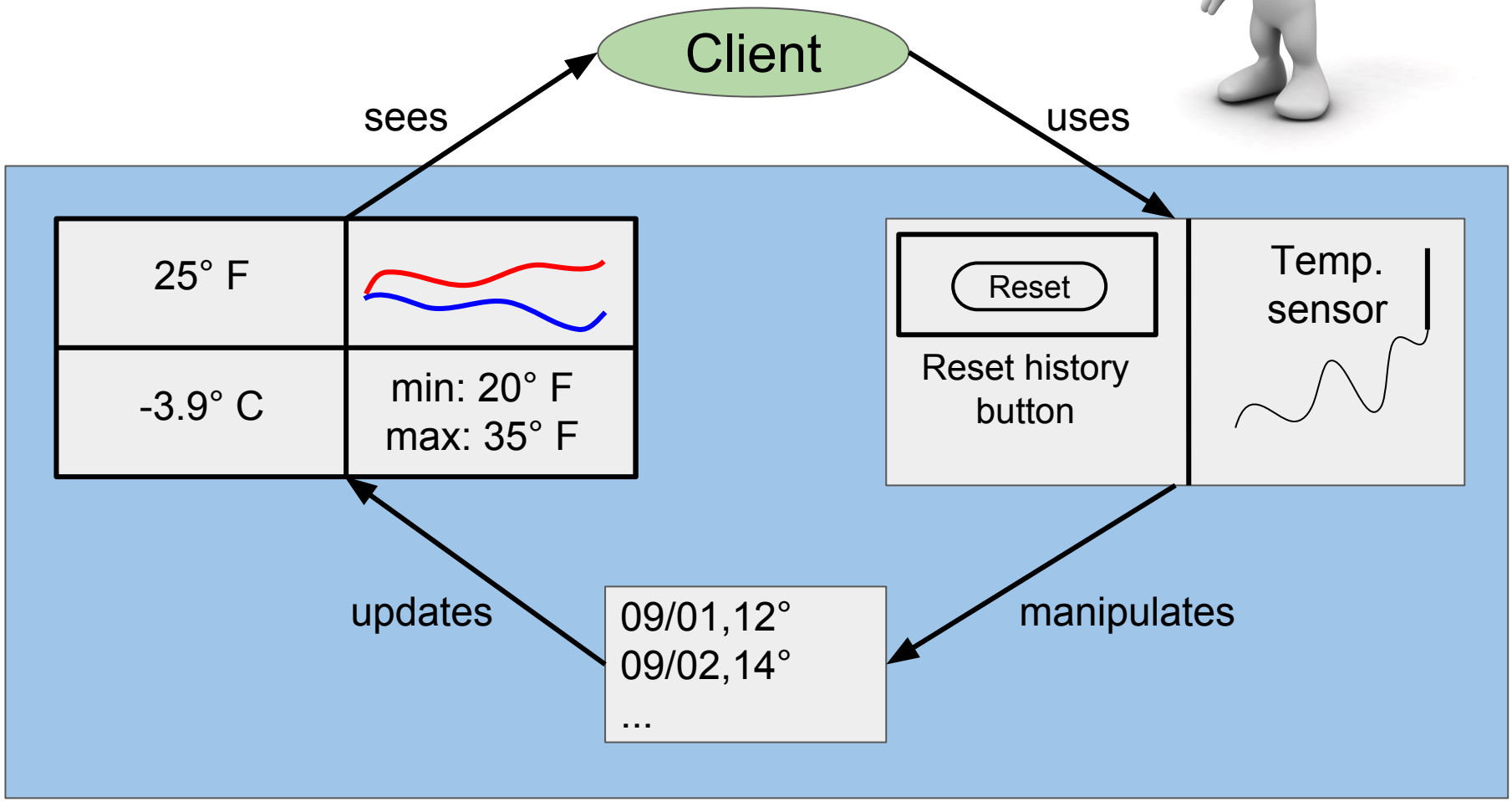
- Information hiding (and encapsulation)
- Open/closed principle
- Liskov substitution principle
- Composition/aggregation over inheritance

A first design problem

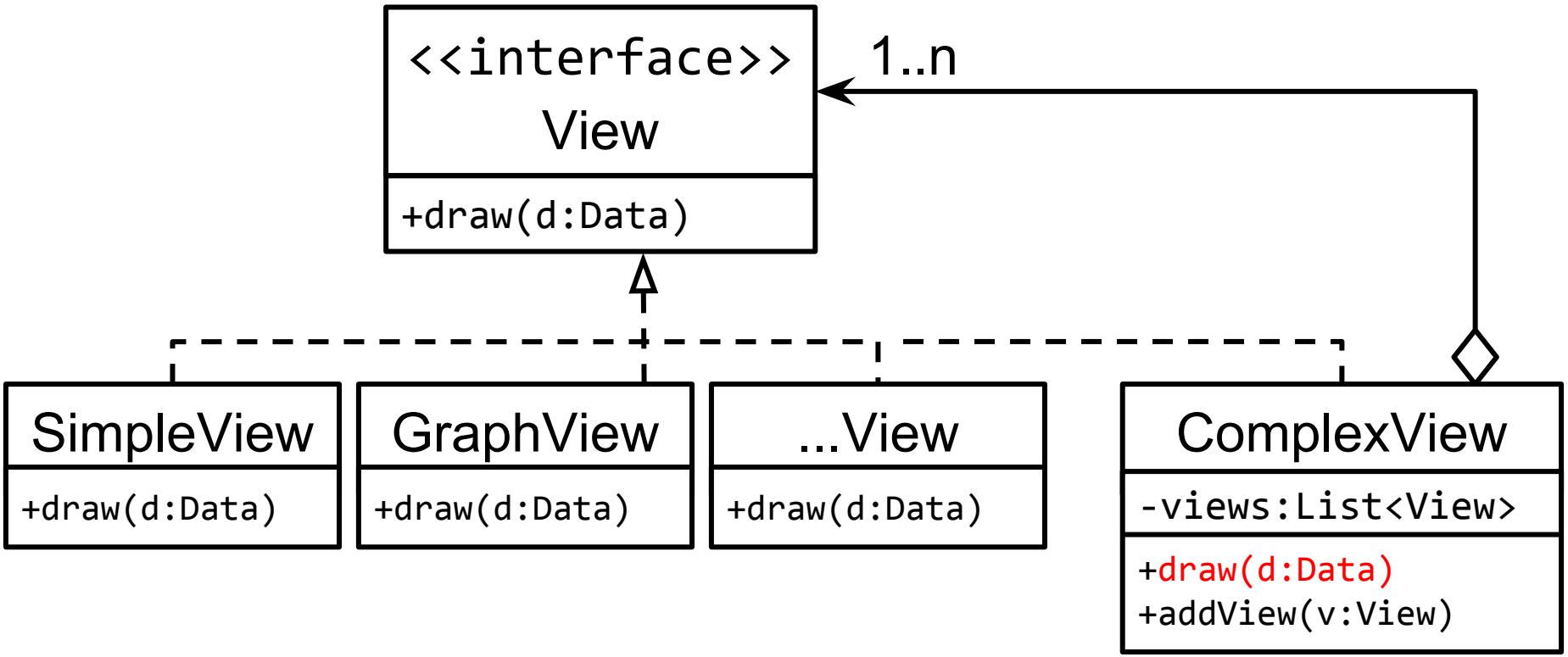
Weather station revisited

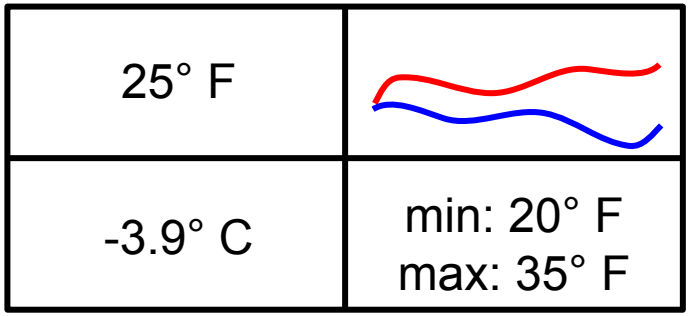


What's a good design for the view?



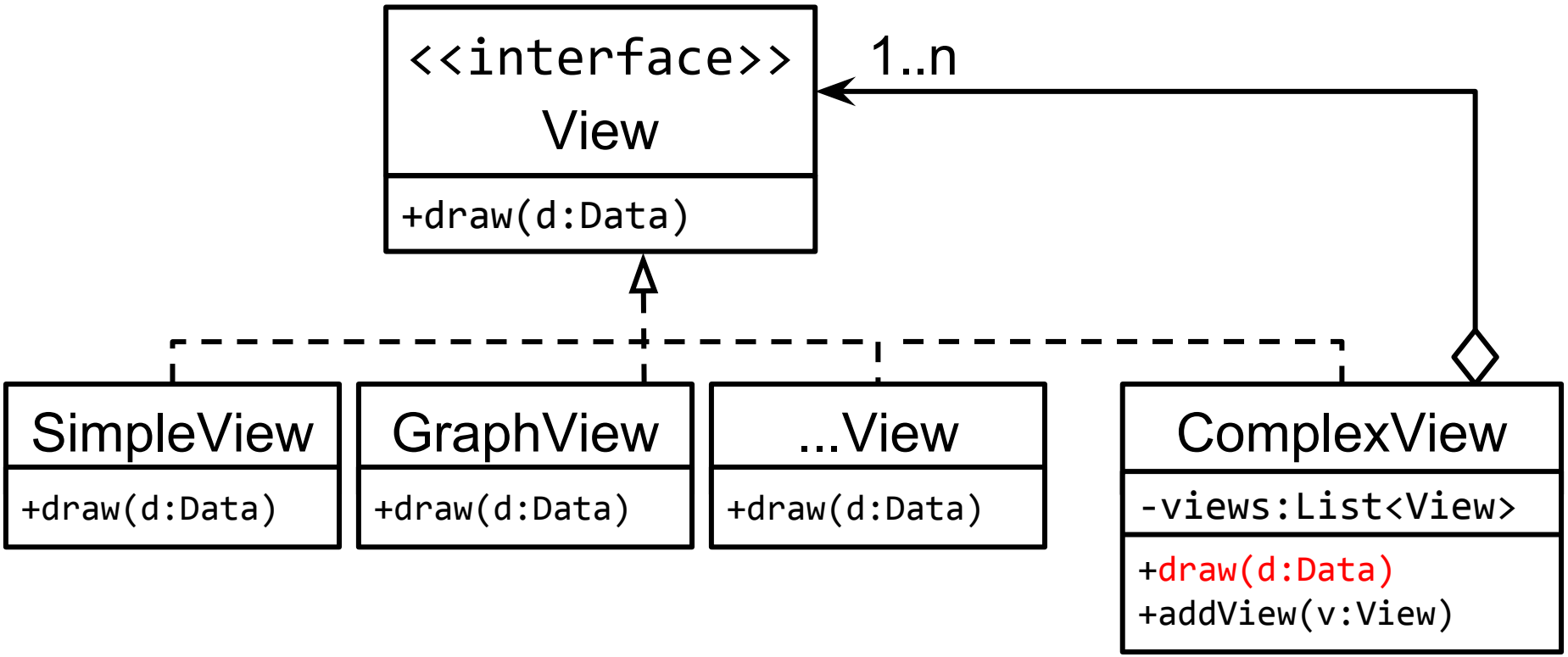
Weather station: view

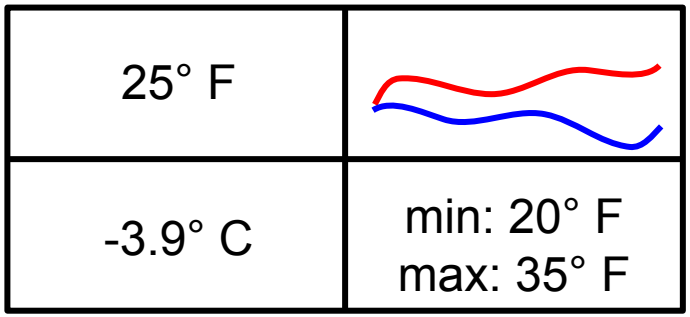




How do we need to implement draw(d:Data)?

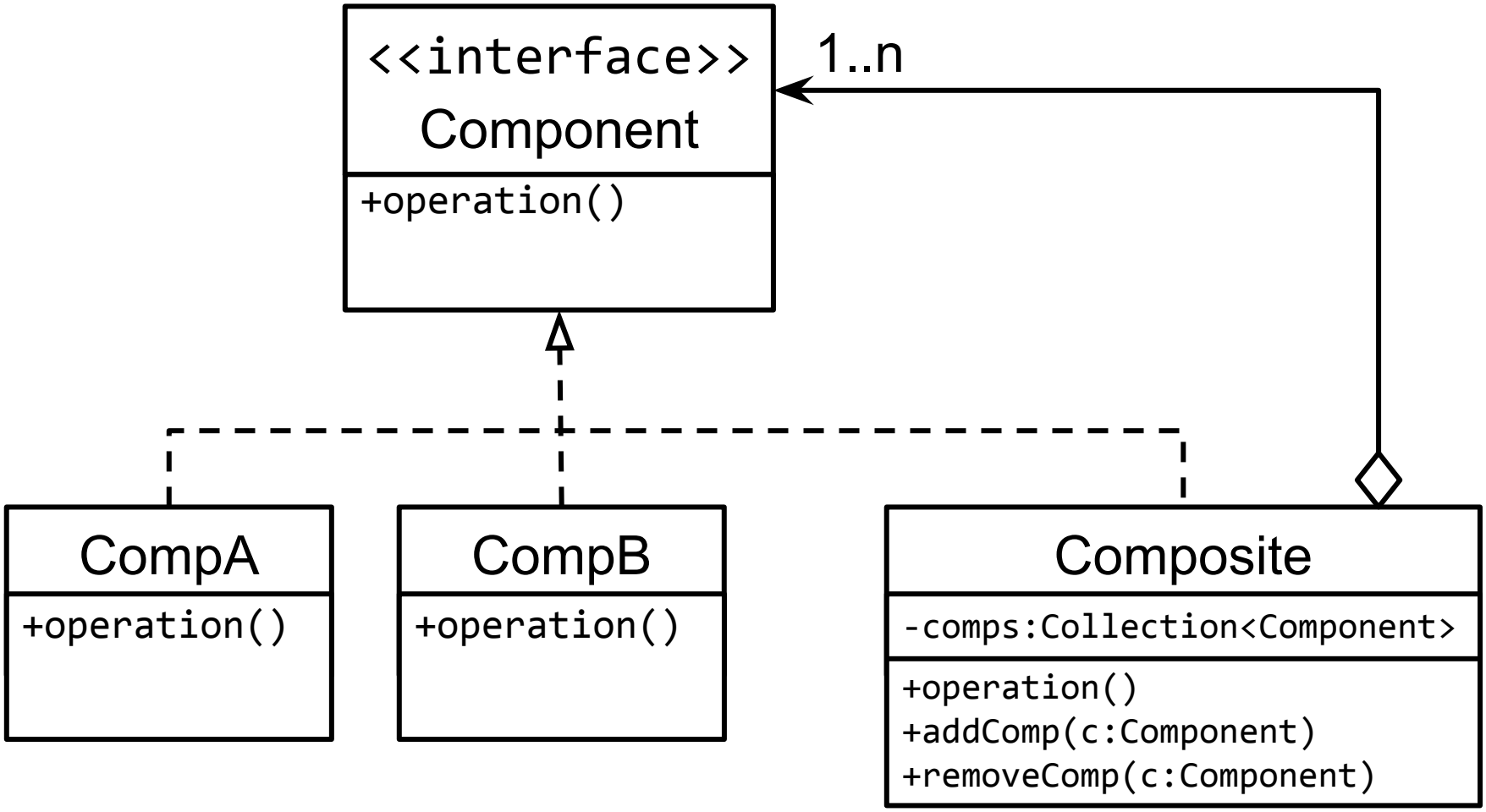
Weather station: view



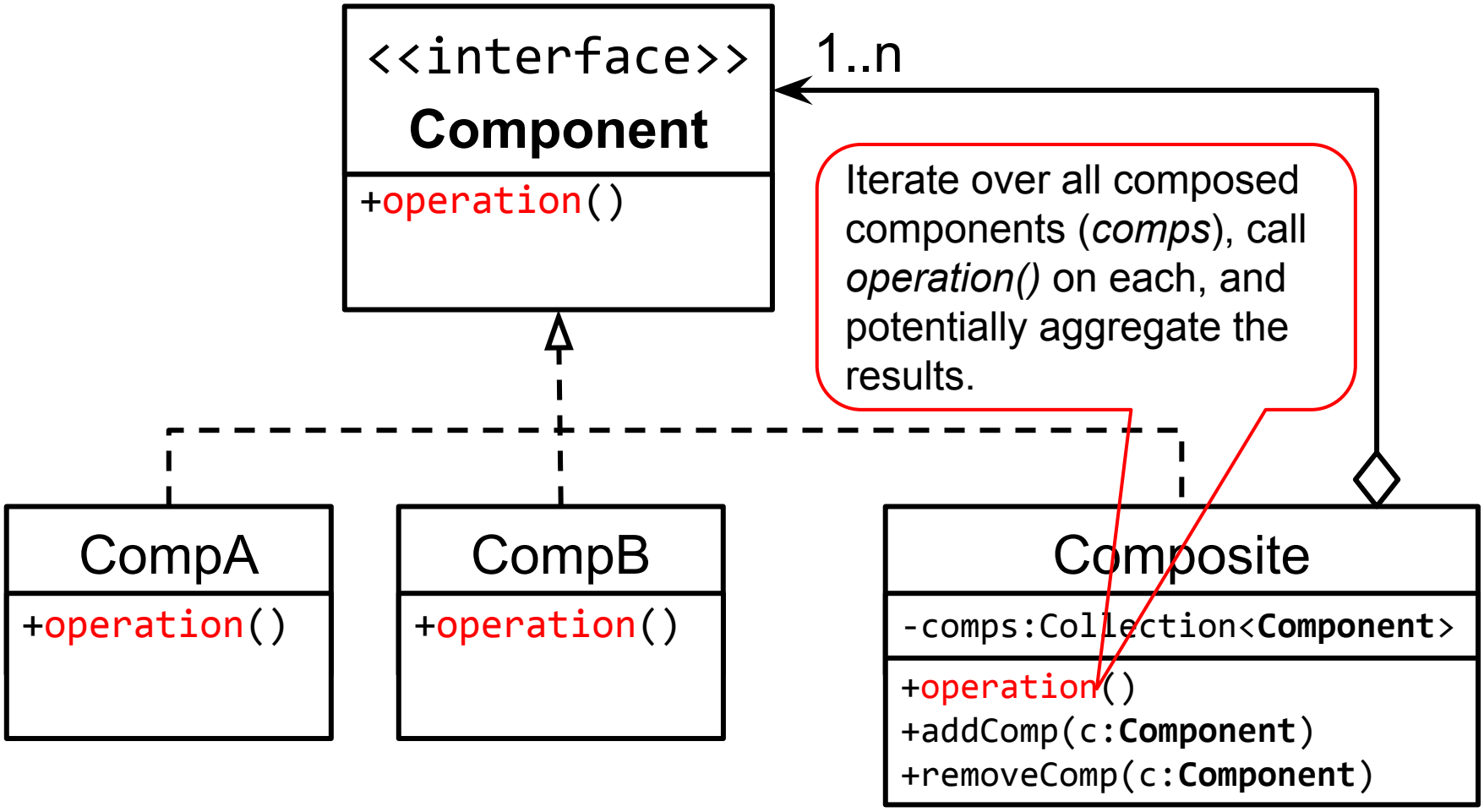



```
public void draw(Data d) {
  for (View v : views) {
    v.draw(d);
  }
}
```

Design pattern: Composite



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

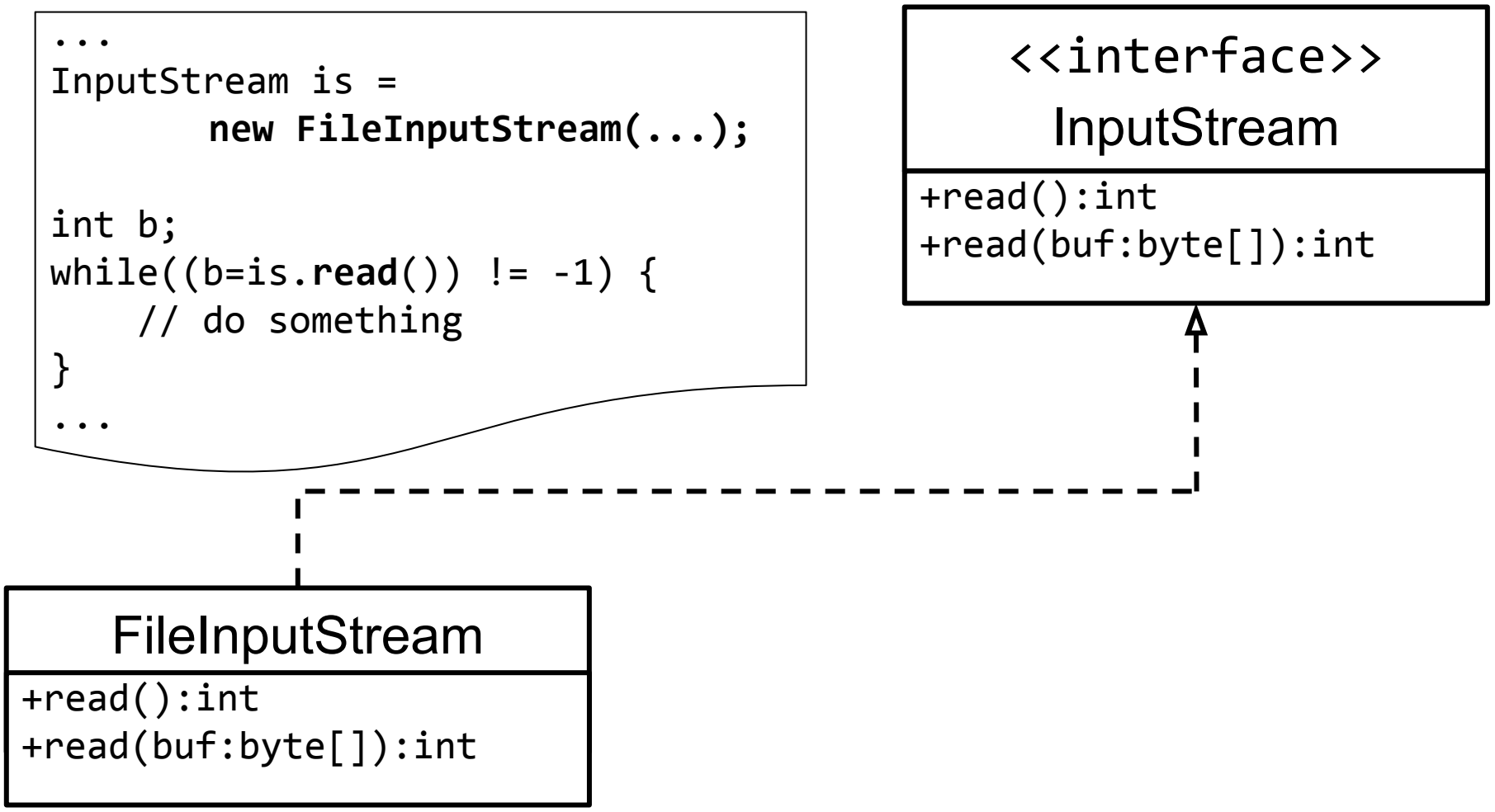
1. Structural
   - Composite
   - Decorator
   - ...
2. Behavioral
   - Template method
   - Visitor
   - ...
3. Creational
   - Singleton
   - Factory (method)
   - ...

Design patterns: categories

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

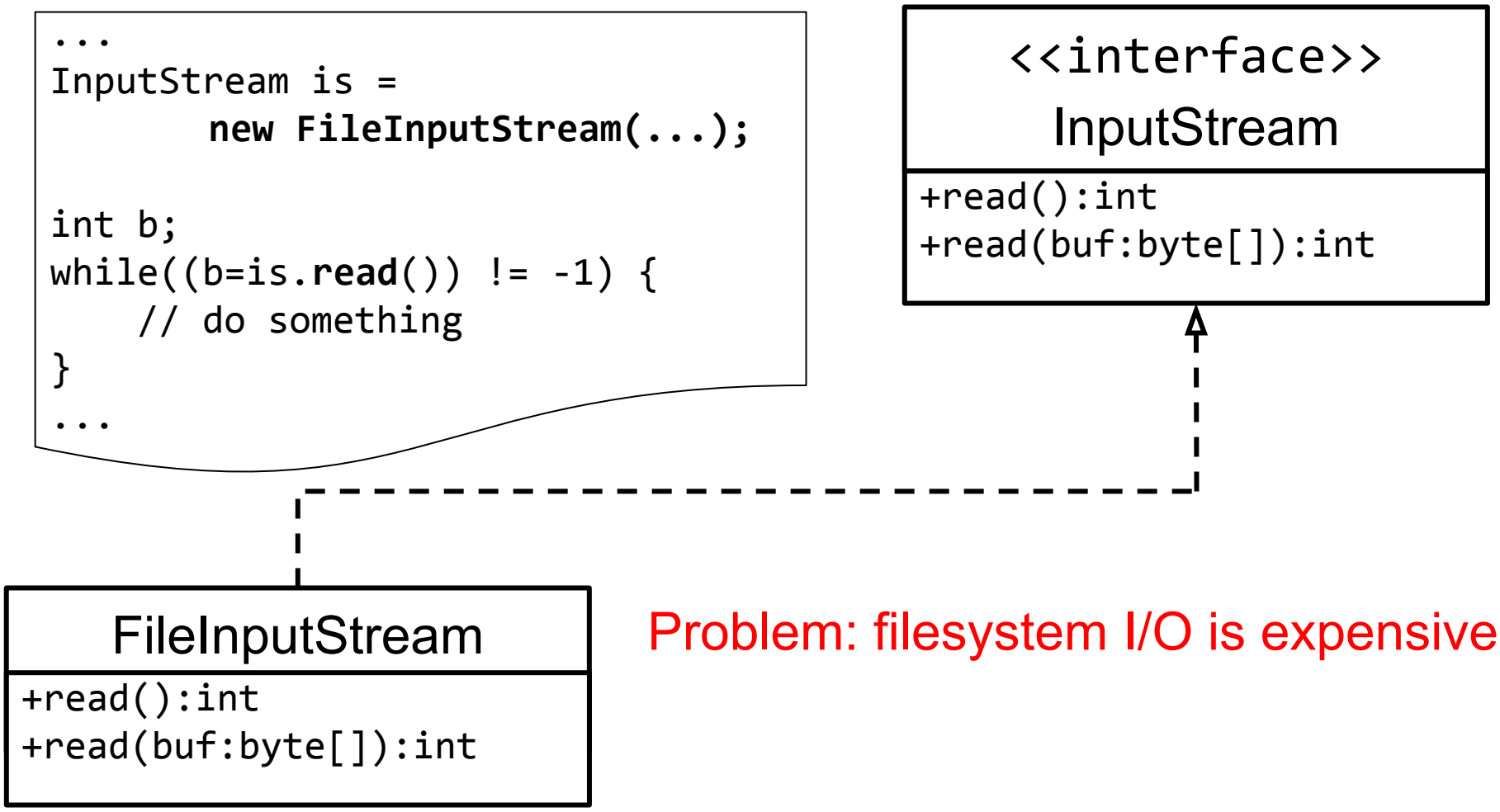
Another design problem: I/O streams

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

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



Another design problem: I/O streams

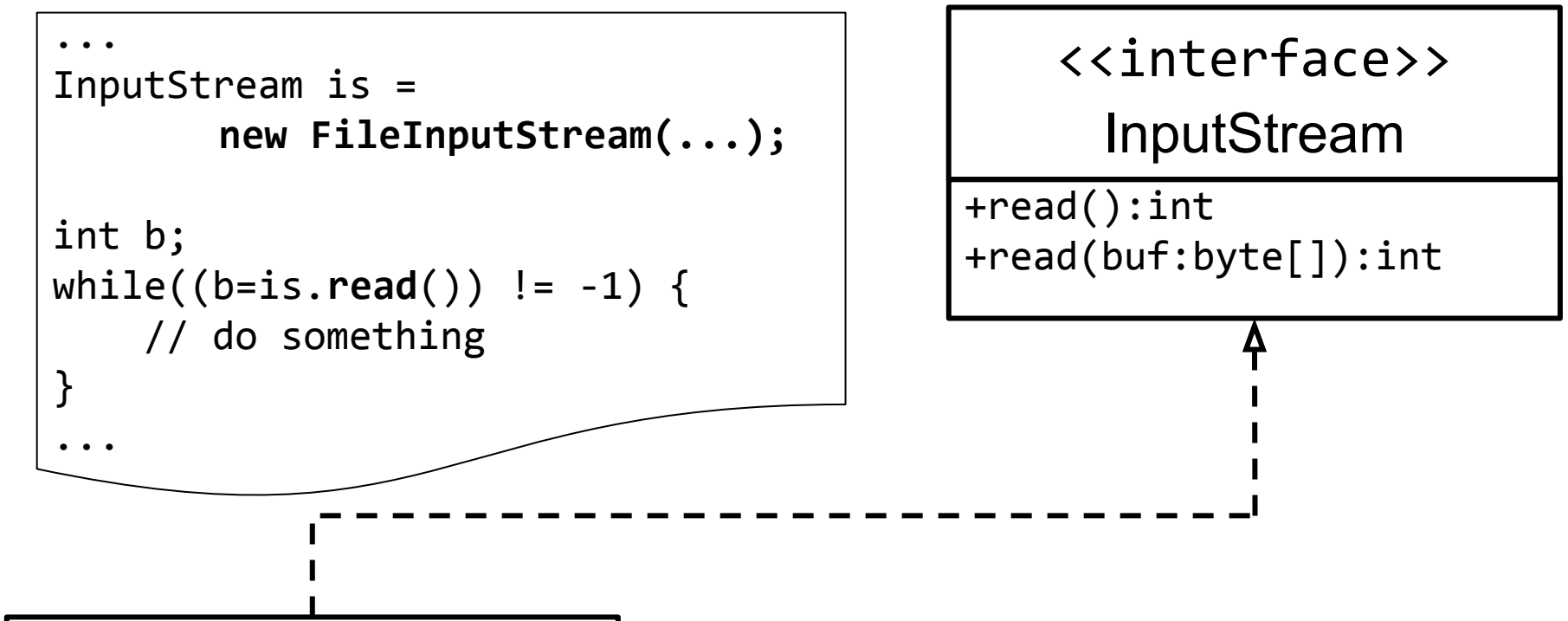
```
...
InputStream is =
        new FileInputStream(...);

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
}
...
```

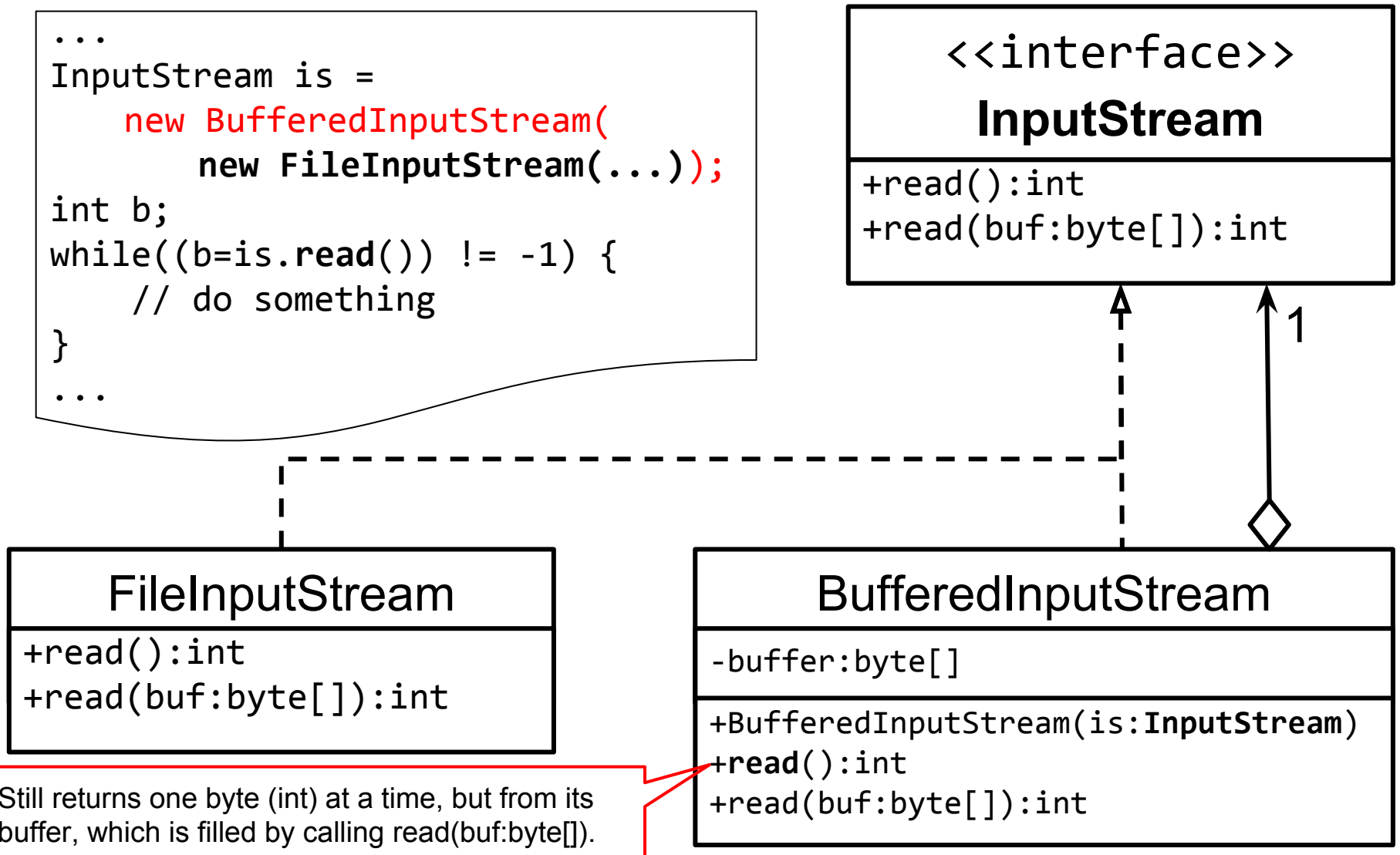


Problem: filesystem I/O is expensive
Solution: use a buffer!

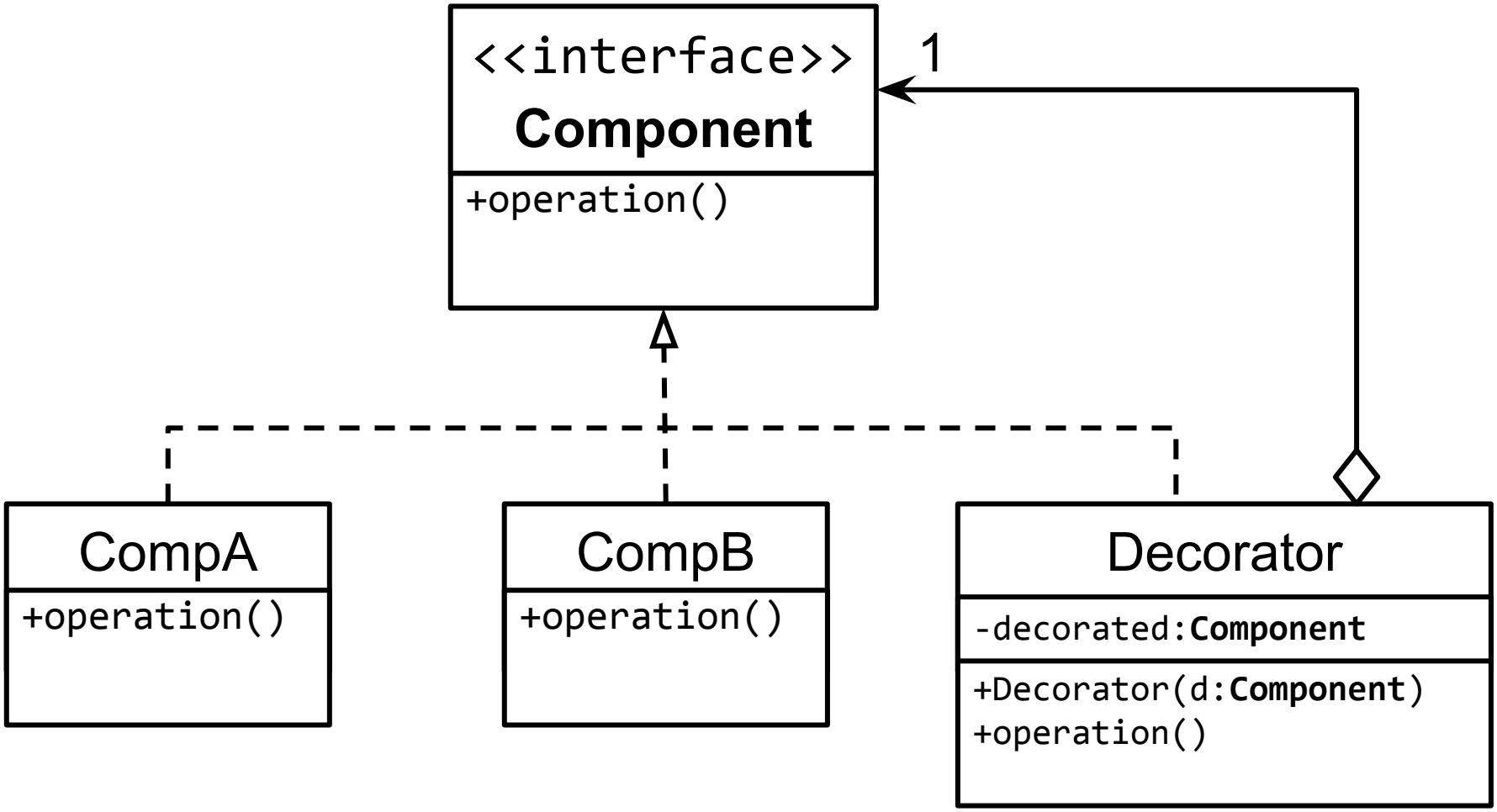
Why not simply implement the buffering in the client or subclass?

Another design problem: I/O streams

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



Design pattern: Decorator



Composite vs. Decorator