الجامعة السورية الخاصة
SYRIAN PRIVATE UNIVERSITY

المحاضرة 5

كلية الهندسة المعلوماتية

مقرر تصميم نظم البرمجيات

Design Patterns: Template Method Pattern, State Pattern, Decorator

د. رياض سنبل

Design Patterns

- **Creational Patterns** *(abstracting the object-instantiation process)*

Factory Method	Abstract Factory	Singleton
Builder	Prototype	

- **Structural Patterns** *(how objects/classes can be combined)*

Adapter	Bridge	Composite
Decorator	Facade	Flyweight
Proxy		

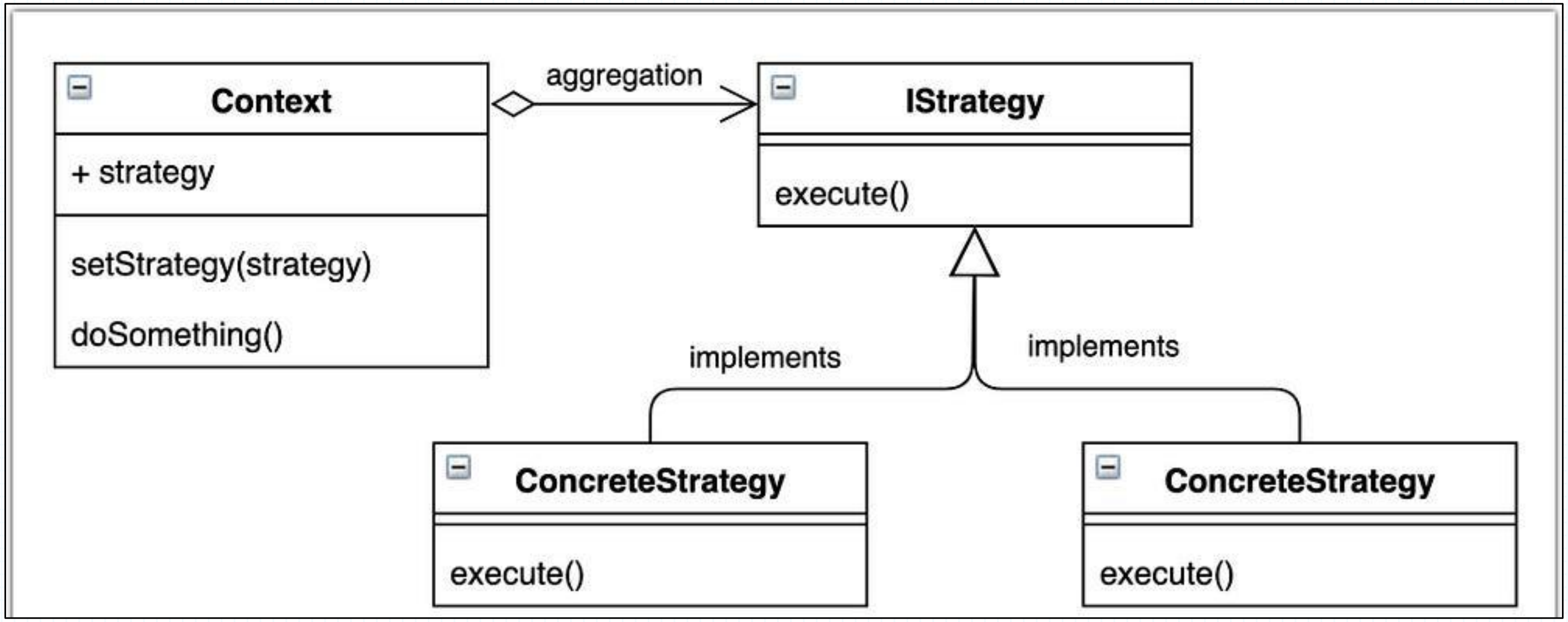
- **Behavioral Patterns** *(communication between objects)*

Command	Interpreter	Iterator
Mediator	Observer	State
Strategy	Chain of Responsibility	Visitor
Template Method		

Template Method Pattern

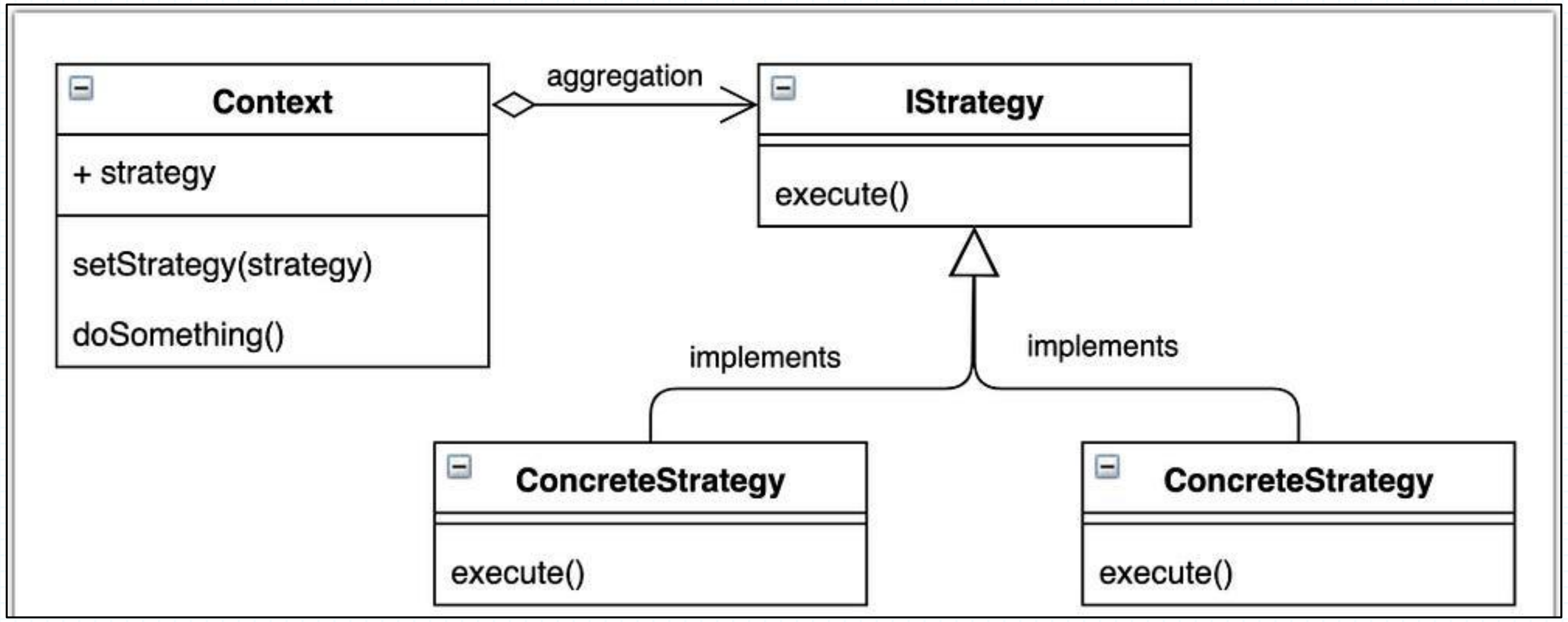
TEMPLATE METHOD PATTERN

Strategy Pattern



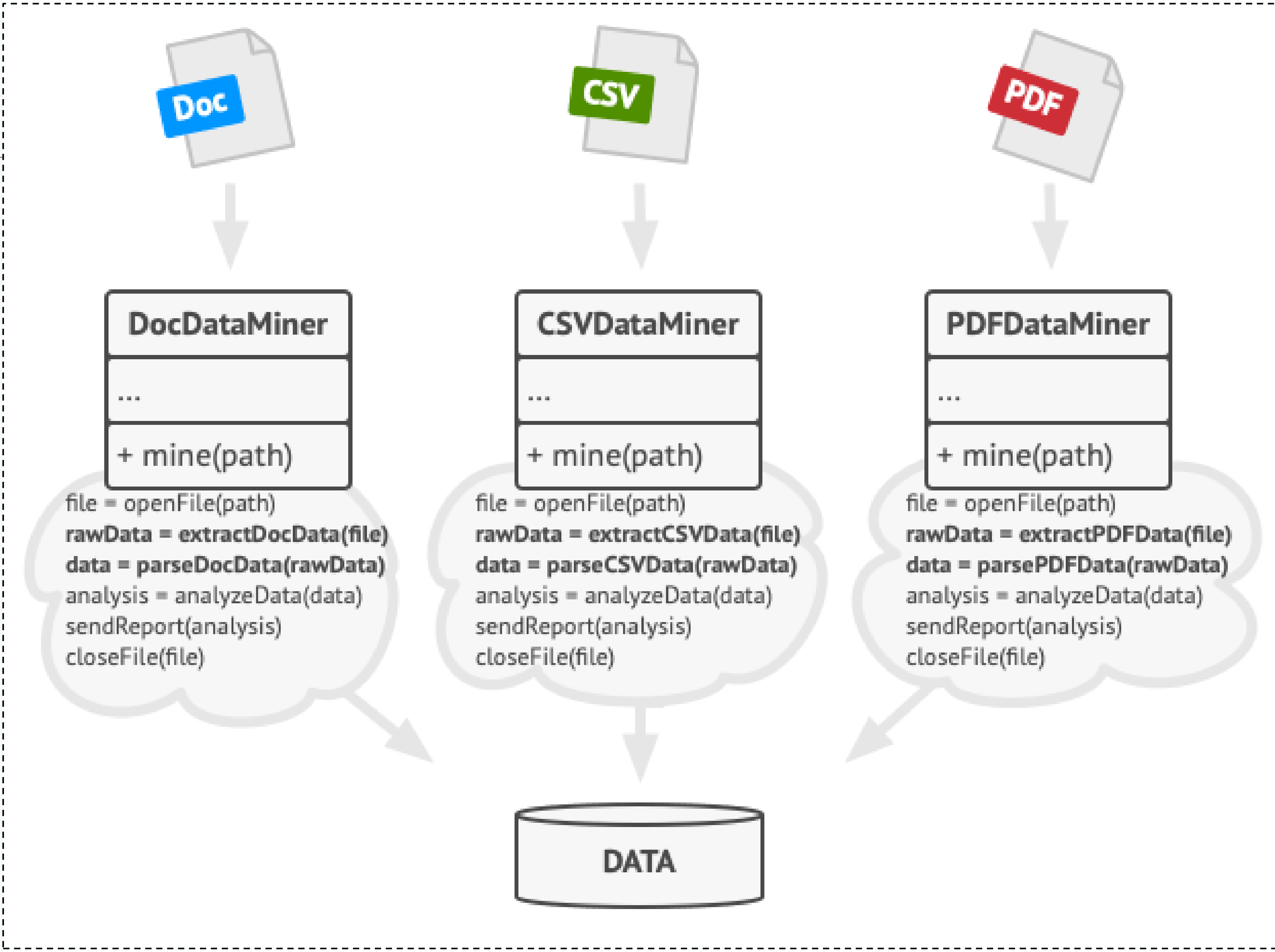
Strategy Pattern.. What if?

- In Strategy patter, IStrategy is an Interface
- i.e. each concreate Stategy has its own implementation..
- What if these strategies have some **<u>common</u>** ("shared") steps!



Example

- Imagine that you’re creating a data mining application that analyzes corporate documents. Users feed the app documents in various formats (PDF, DOC, CSV), and it tries to extract meaningful data from these docs in a uniform format.



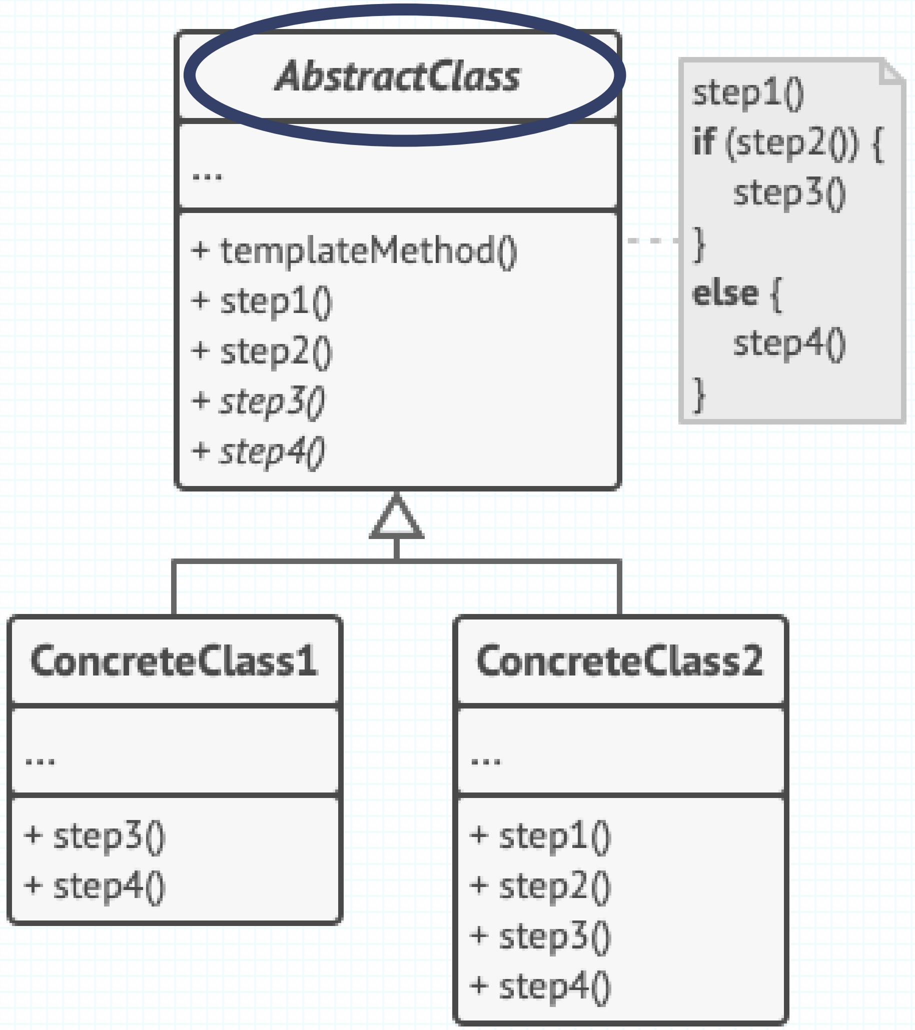
```
file = openFile(path)
rawData = extractDocData(file)
data = parseDocData(rawData)
analysis = analyzeData(data)
sendReport(analysis)
closeFile(file)
```

```
file = openFile(path)
rawData = extractCSVData(file)
data = parseCSVData(rawData)
analysis = analyzeData(data)
sendReport(analysis)
closeFile(file)
```

```
file = openFile(path)
rawData = extractPDFData(file)
data = parsePDFData(rawData)
analysis = analyzeData(data)
sendReport(analysis)
closeFile(file)
```

Template Method Pattern

- **Template Method** is a behavioral design pattern that defines the skeleton of an algorithm in the superclass but lets **subclasses override** specific steps of the algorithm without changing its structure.



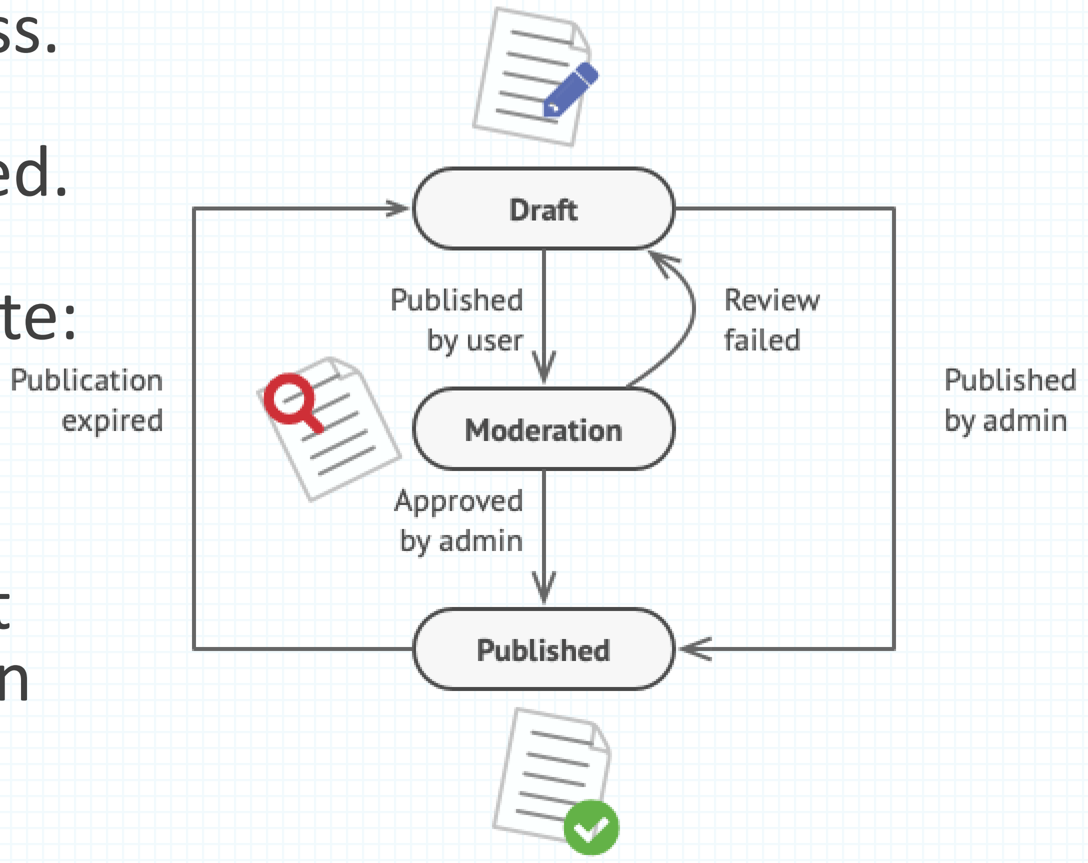
State Pattern

The Problem

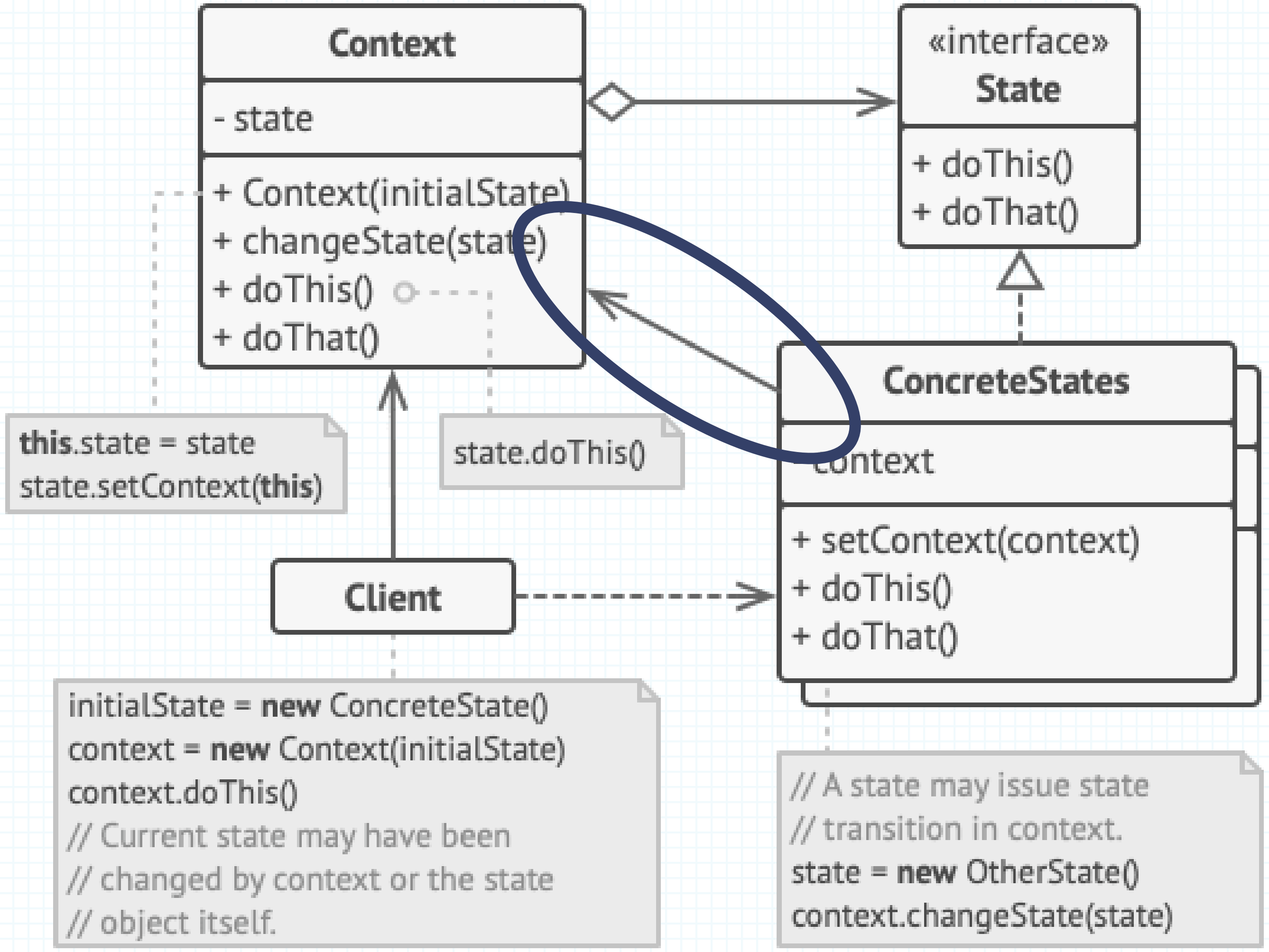
- An object's behavior changes based on its internal state!
- Operations have large, multipart conditional statements that depend on the object’s state.
- This state is usually represented by one or more enumerated constants.
- Often, several operations will contain this same conditional structure.
- The State pattern puts each branch of the conditional in a separate class

Example

- Imagine that we have a Document class. A document can be in one of three states: Draft, Moderation and Published. The publish method of the document works a little bit differently in each state:
- In Draft, it moves the document to moderation.
- In Moderation, it makes the document public, but only if the current user is an administrator.
- In Published, it doesn't do anything at all.



State Pattern



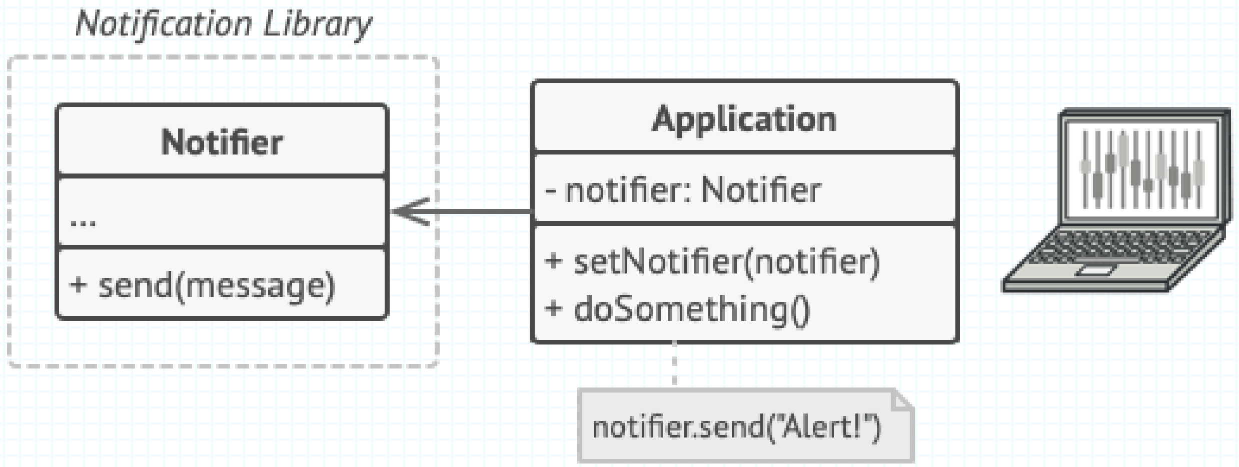
```
initialState = new ConcreteState()
context = new Context(initialState)
context.doThis()
// Current state may have been
// changed by context or the state
// object itself.
```

```
// A state may issue state
// transition in context.
state = new OtherState()
context.changeState(state)
```

Decorator Pattern

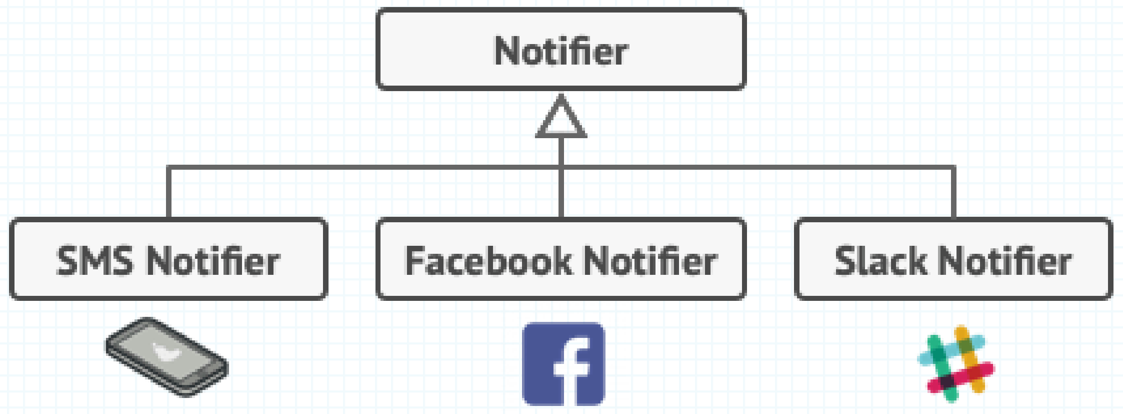
Again.. Let us start with the problem ☺

- you’re working on a notification library which lets other programs notify their users about important events.
- The initial version of the library was based on:
  - the *Notifier* class that had only a few fields, a constructor and a single *send* method. The method could accept a message argument from a client and send the message to a list of emails that were passed to the notifier via its constructor.



Req. Changed:

The library expect more than just email notifications. They would like to receive an SMS about critical issues. Others would like to be notified on Facebook and, of course, the corporate users would love to get Slack notifications.

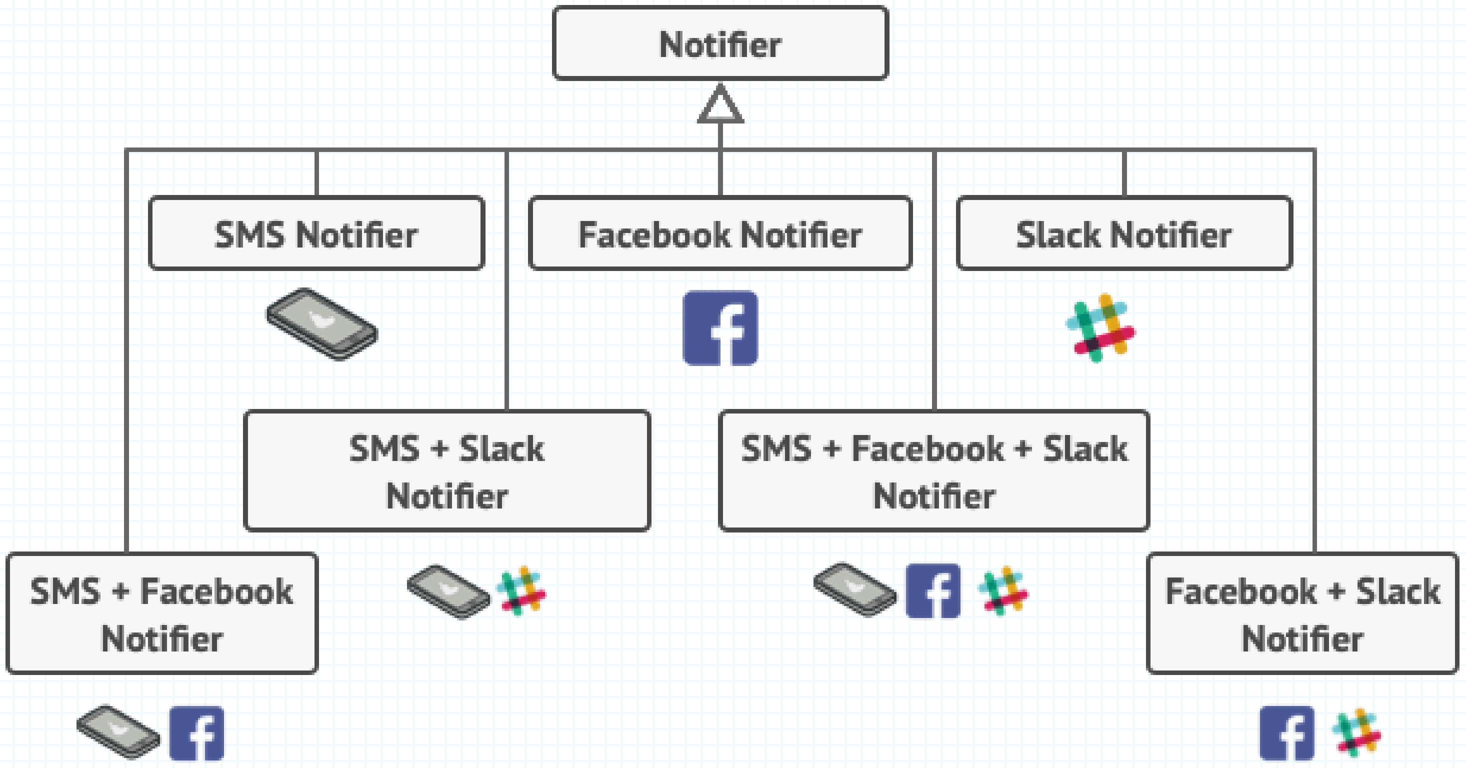


Solution:

- ✓ Extended the Notifier class and put the additional notification methods into new subclasses.
- ✓ Now the client was supposed to instantiate the desired notification class and use it for all further notifications.

New Change ☹

Why can't we use several notification types at once?
If your house is on fire, you'd probably want to be informed through every channel.

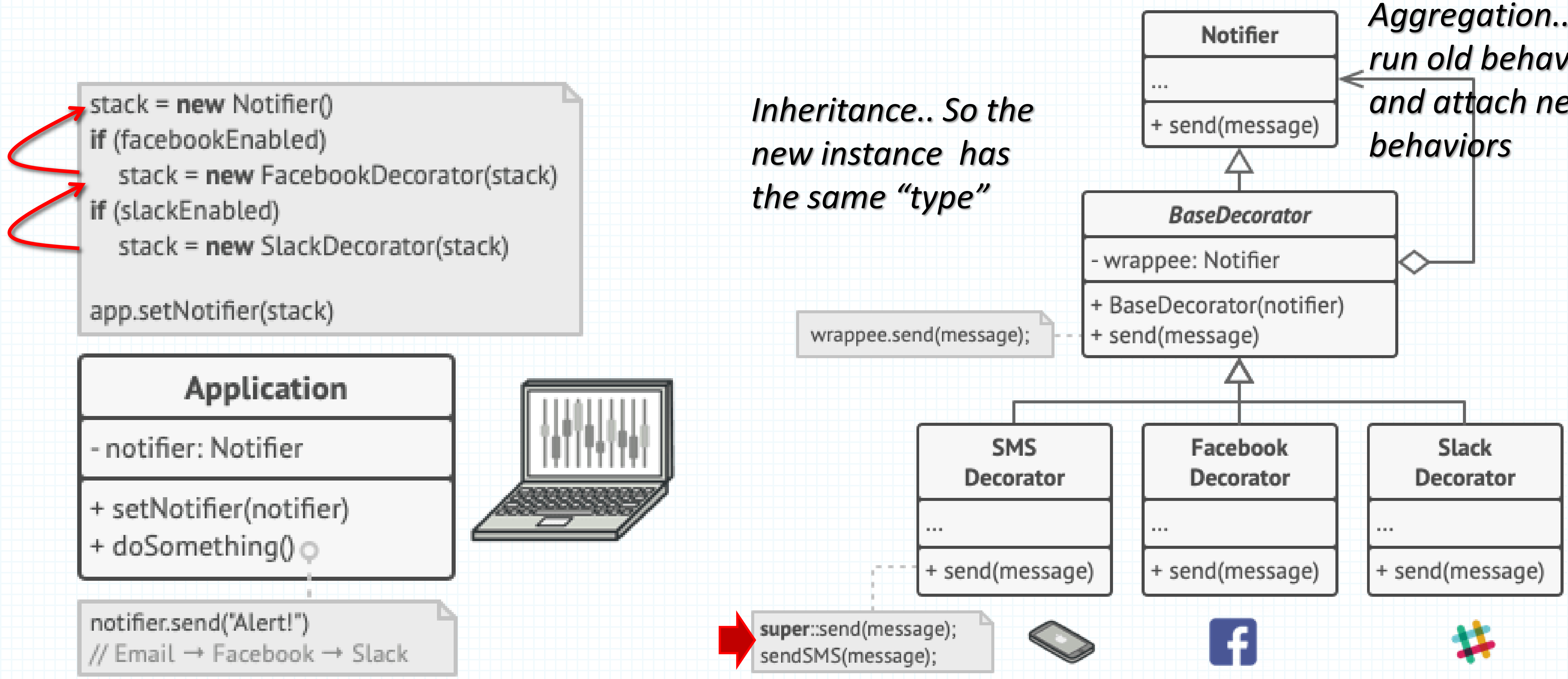


BAD Solution

Solution (Decorator)

Inheritance.. So the new instance has the same "type"

Aggregation.. To run old behaviors and attach new behaviors



```
stack = new Notifier()
if (facebookEnabled)
    stack = new FacebookDecorator(stack)
if (slackEnabled)
    stack = new SlackDecorator(stack)

app.setNotifier(stack)
```

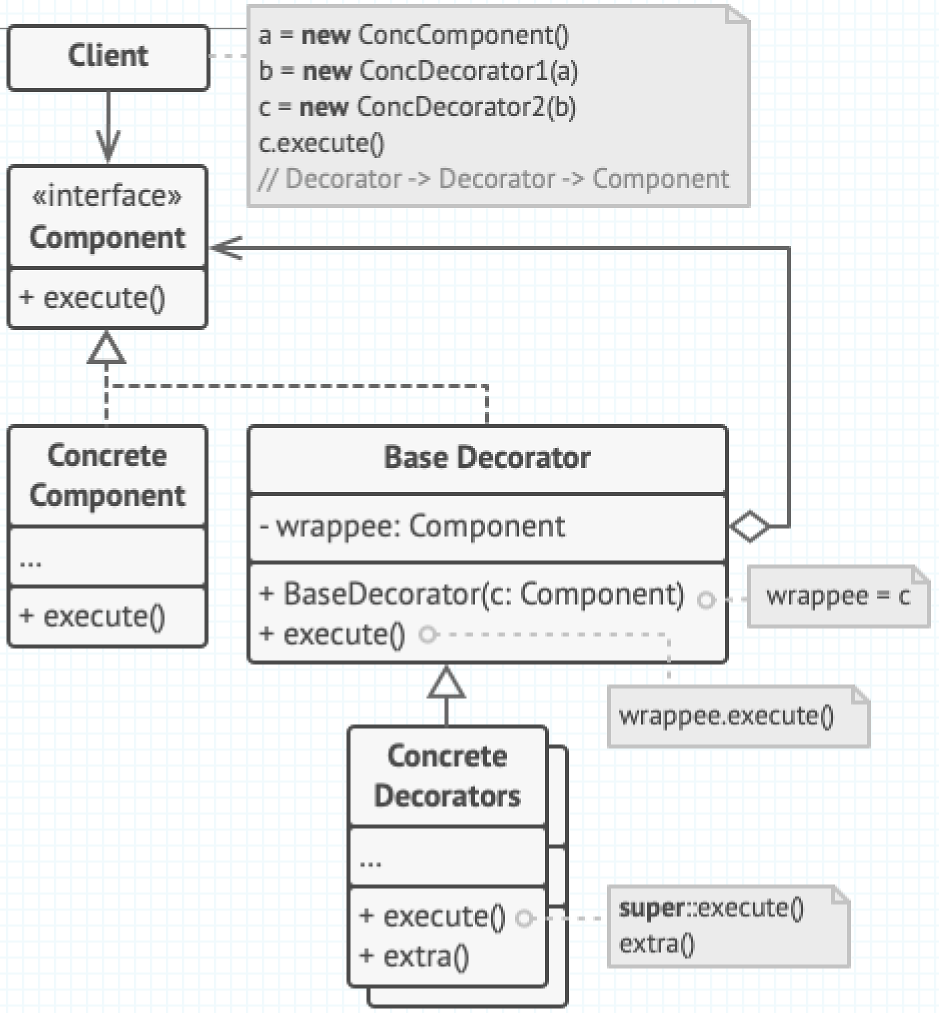
```
notifier.send("Alert!")
// Email → Facebook → Slack
```

```
wrappee.send(message);
```

```
super::send(message);
sendSMS(message);
```

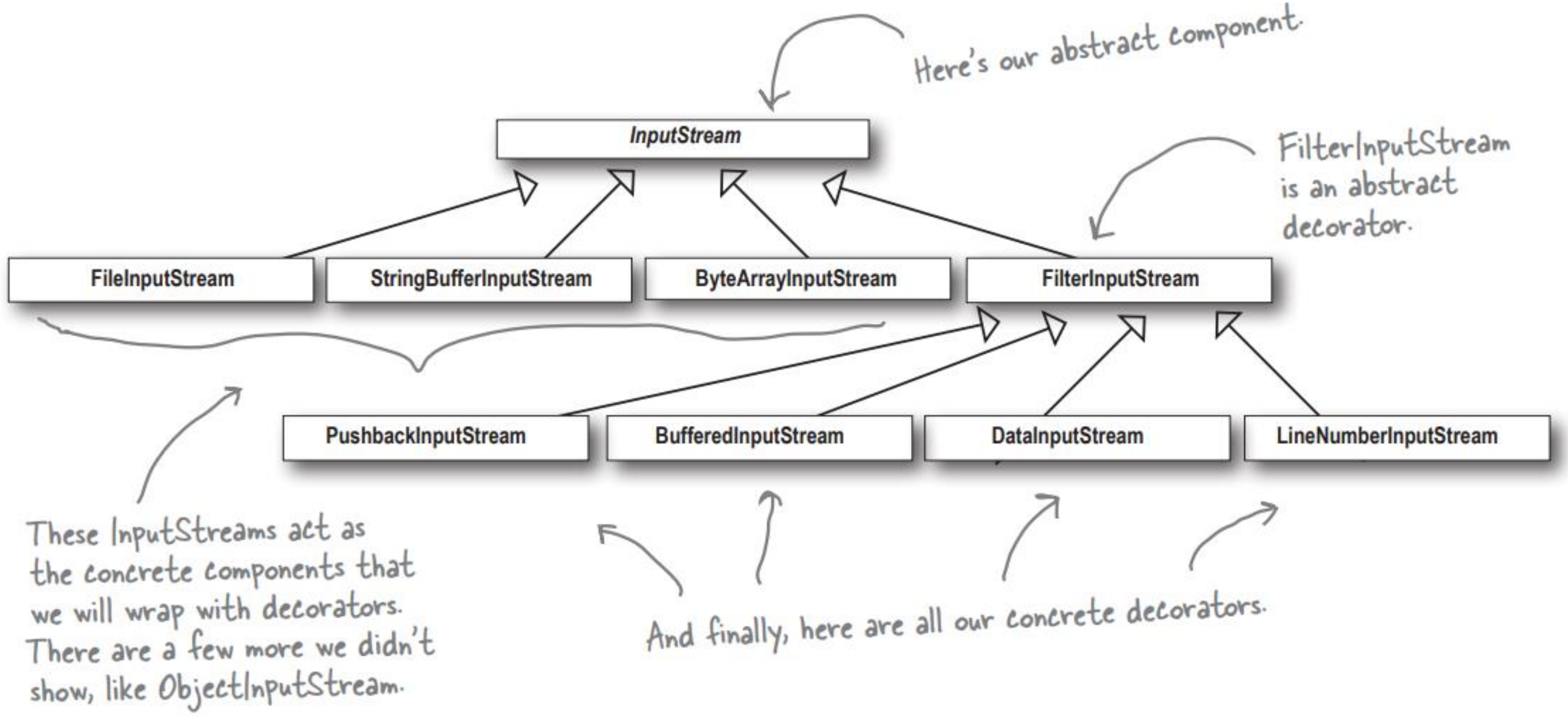
Decorator Pattern

- Decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.
- Use the Decorator pattern when you need to be able to assign extra behaviors to objects at <u>runtime</u> without breaking the code that uses these objects.

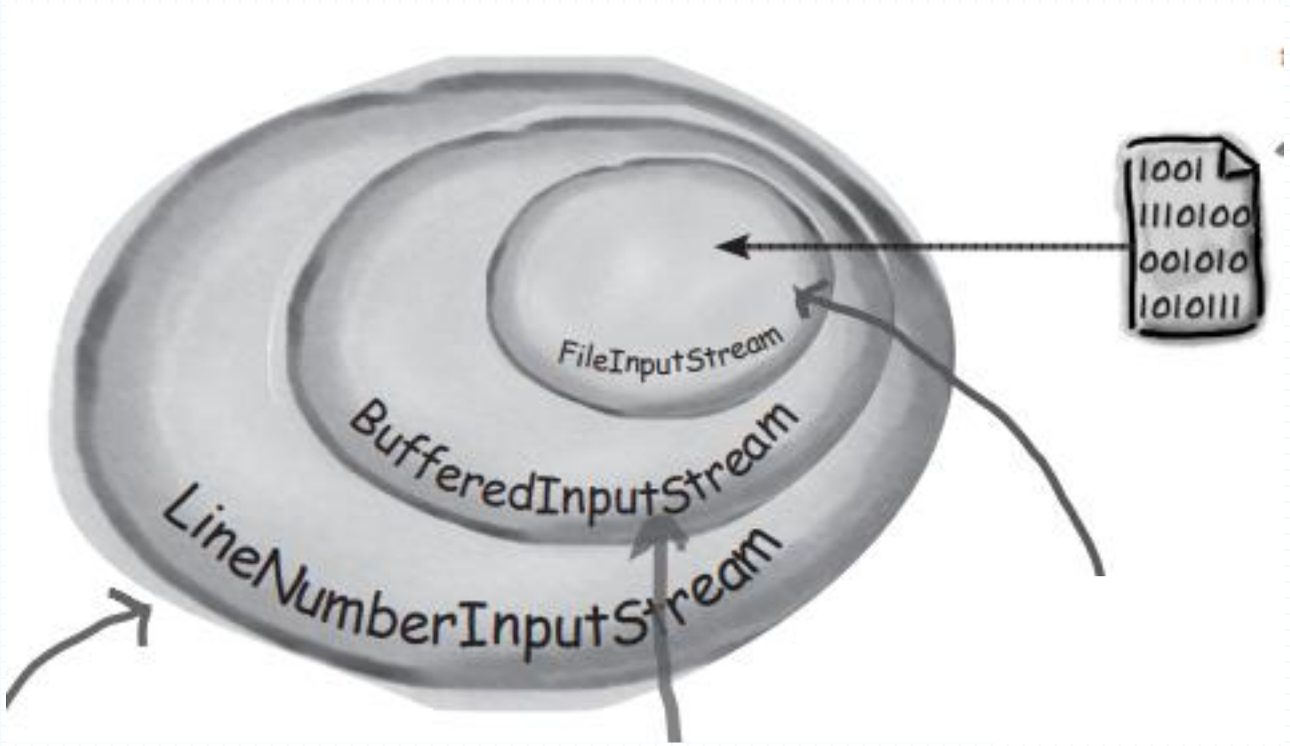


```
a = new ConcComponent()
b = new ConcDecorator1(a)
c = new ConcDecorator2(b)
c.execute()
// Decorator -> Decorator -> Component
```

Another Example.. From Java.io


Here's our abstract component.
InputStream
FilterInputStream is an abstract decorator.
FileInputStream
StringBufferInputStream
ByteArrayInputStream
FilterInputStream
PushbackInputStream
BufferedInputStream
DataInputStream
LineNumberInputStream
These InputStreams act as the concrete components that we will wrap with decorators. There are a few more we didn't show, like ObjectInputStream.
And finally, here are all our concrete decorators.

Another Example.. From Java.io


1001
1110100
001010
1010111
FileInputStream
BufferedInputStream
LineNumberInputStream