Eliminating Conditional Complexity

Andrejs Doronins
TEST AUTOMATION ENGINEER

Conditional Code

(Branching)

App1.java

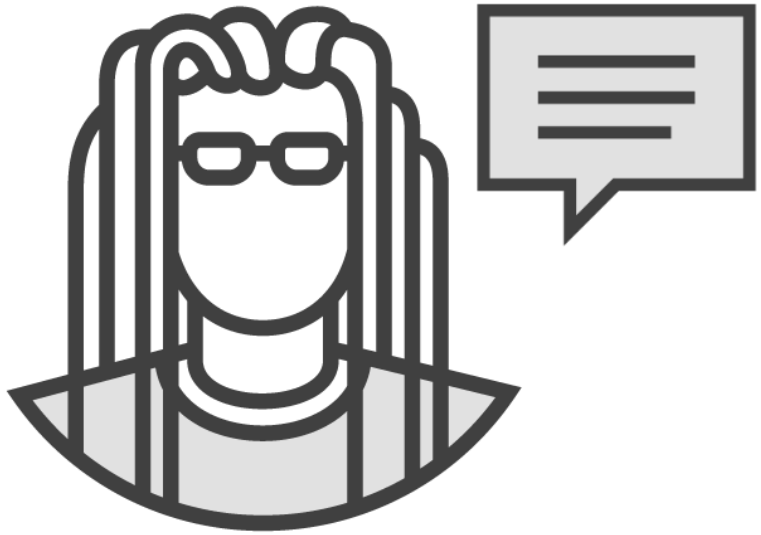
```
if(condition1) {

} else if (condition2) {

}
```

App2.java

```
switch(value){
    case v1:
    case v2:
    default:
}
```

Conditional code
is not bad in itself

Conditional code breaks OCP!



```
if(condition1) {
    if(condition3 && condition4){
        if(condition5){...}
    }
} else if(condition2) {
    if(!condition6){
        switch(val){...}
    }
}
```

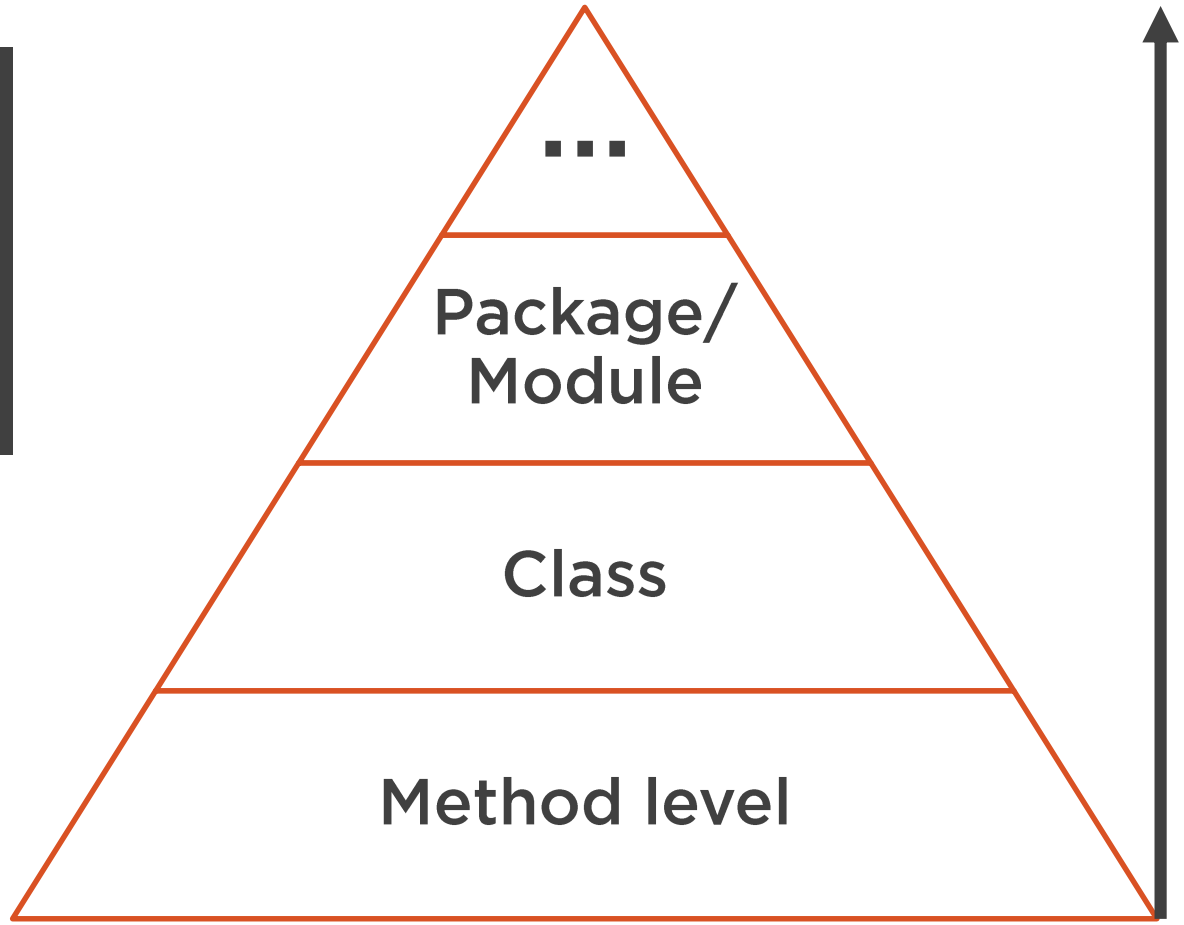
Fragile and complex

End goal:

Reduce conditional complexity ✓

Fit code into a pattern ✗

Refactoring Pyramid



Overview

- Reduce level of code nestedness
- Replace conditional with polymorphism
- Strategy pattern
- Functional programming alternative

Reducing Nested Code

1. Guard Clause
2. Extract Conditional
3. Java 8 Streams

Use Guard Clauses

```
if(val1 != null) {
    if(val2 != null){
        if(val3 != null){
            // do the thing
        }
    }
}
throw new SomeException("...");
```

```
if(val1 == null || val2 == null ||
    val3 == null ){
    throw new SomeException("...");
}
// do the thing
```

Extract Conditional

```
Integer val = ...;

if(val == null && val < 0){
    throw new SomeException("...");
}
```

```
Integer val = ...;

if(check(val)){
    throw new SomeException("...");
}

boolean check(Integer val){
    // check
}
```

Imperative (How?)
vs.
Declarative (What?)

ImperativeStyle.java

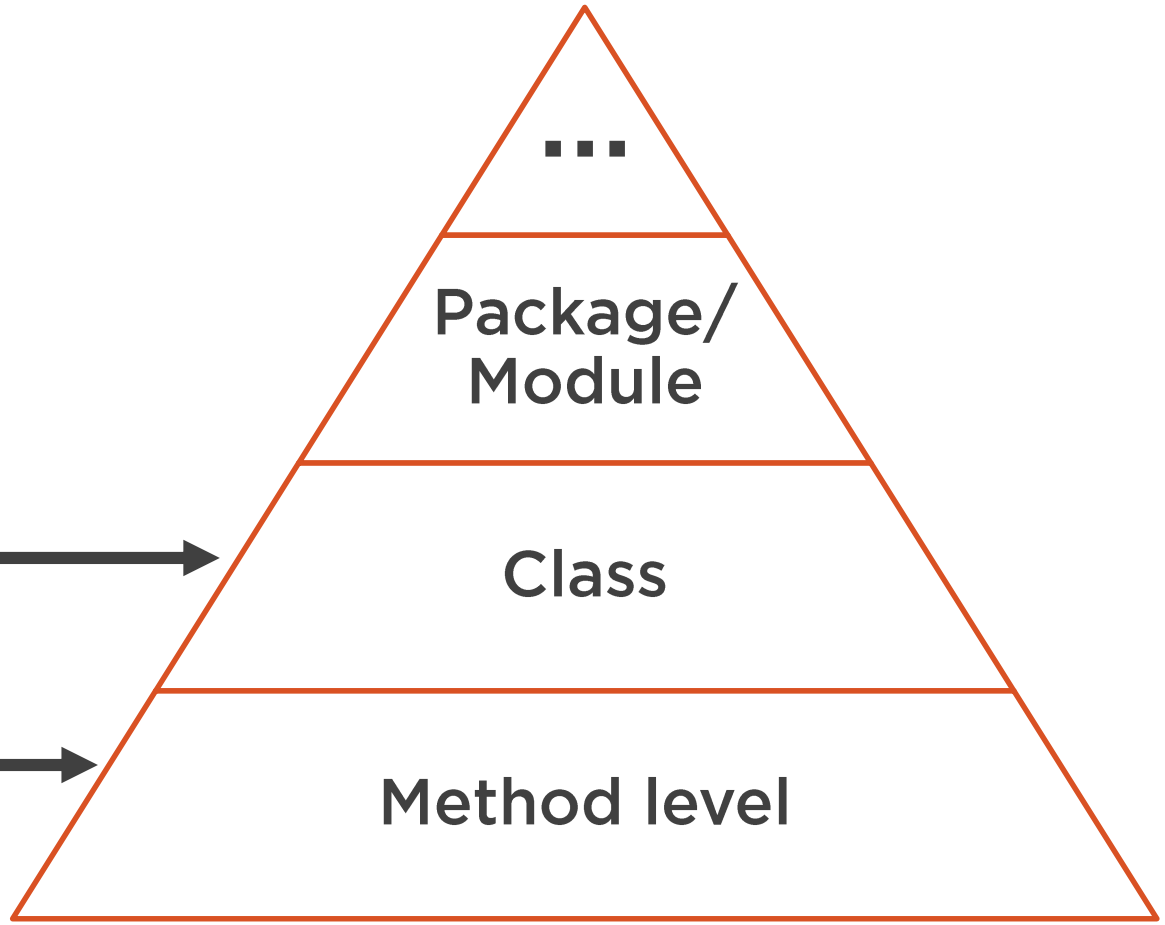
```
for(...){
    if(...) {
        for(...){
            if(...){ }
        }
    }
}
```

DeclarativeStyle.java

```
things.stream()
    .filter()
    .map()
    .distinct()
    .collect();
```

Refactoring Pyramid

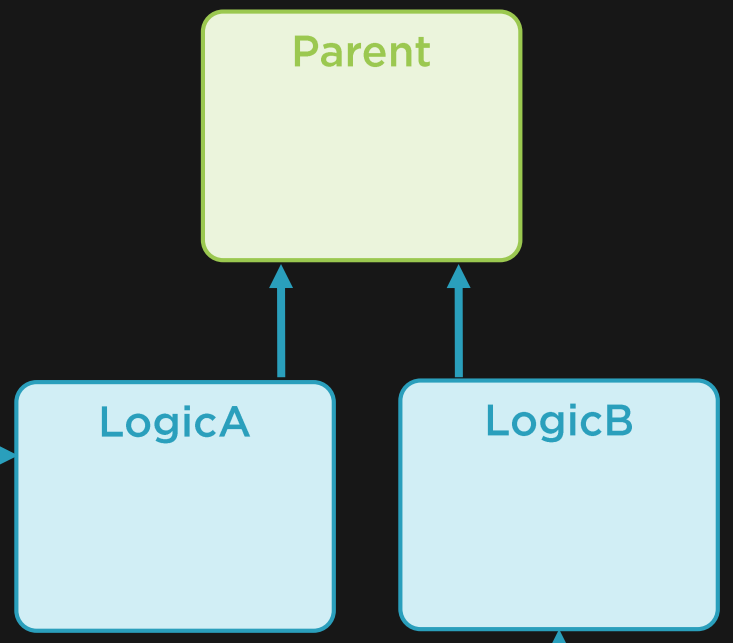
1. Guard Clause
2. Extract Conditional
3. Java 8 Streams



Simple but large conditional

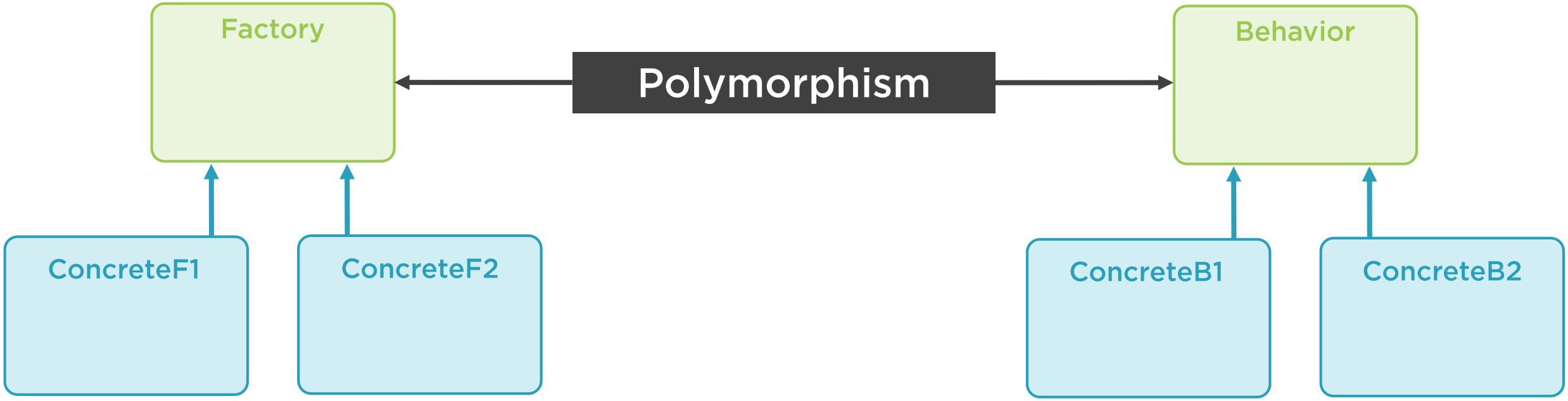
SomeFile.java

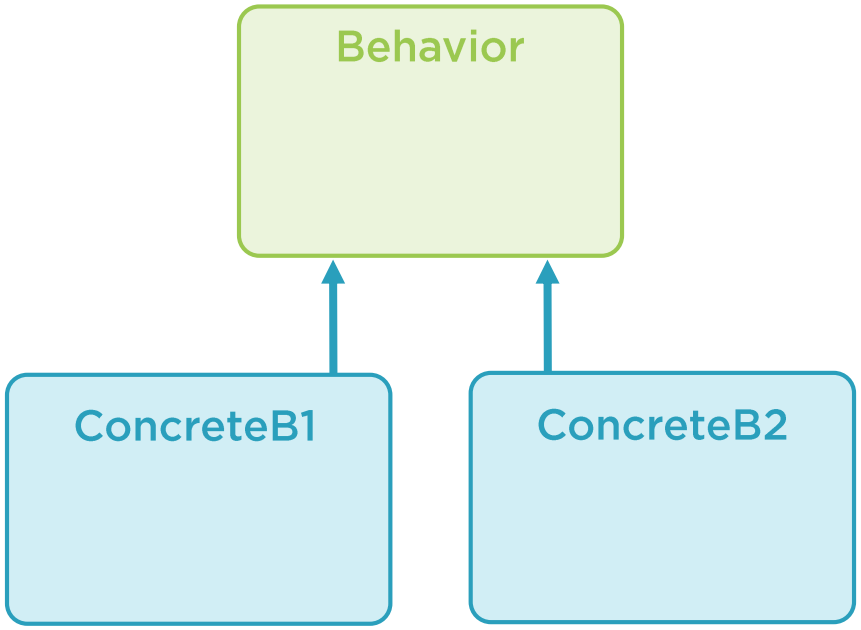
```
if(condition1){
    // lots of code
    // lots of code
    // lots of code
} else {
    // lots of code
    // lots of code
    // lots of code
}
```

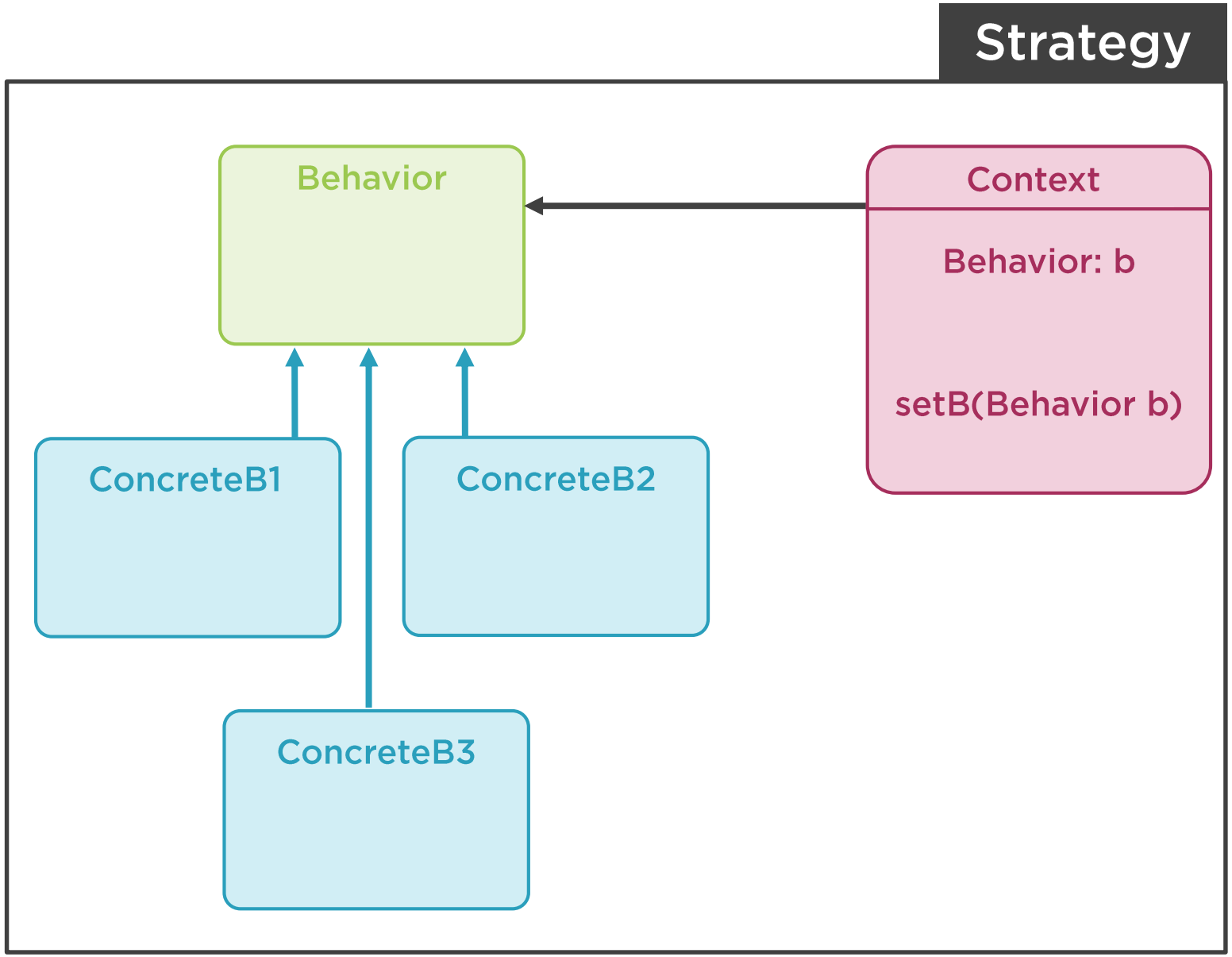


Encapsulating behavior

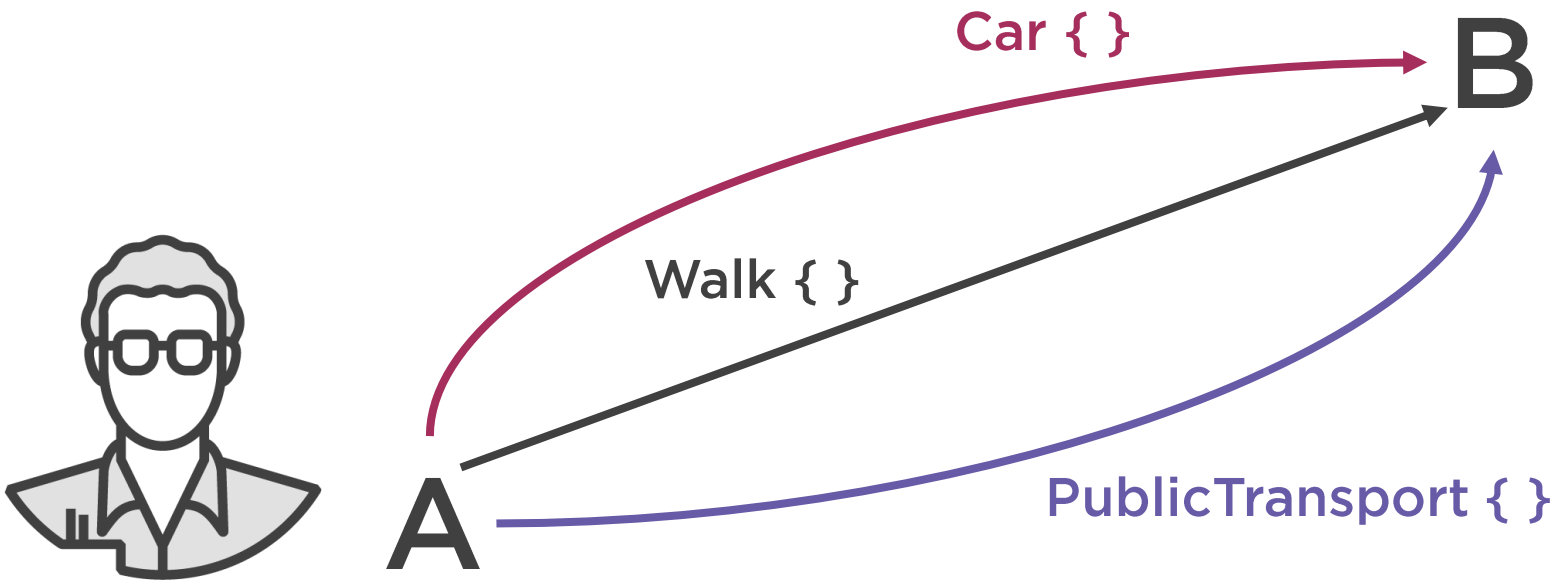
Consider **replacing** with **polymorphism** before introducing a behavioral pattern


Factory
Polymorphism
Behavior
ConcreteF1
ConcreteF2
ConcreteB1
ConcreteB2


Behavior
ConcreteB1
ConcreteB2

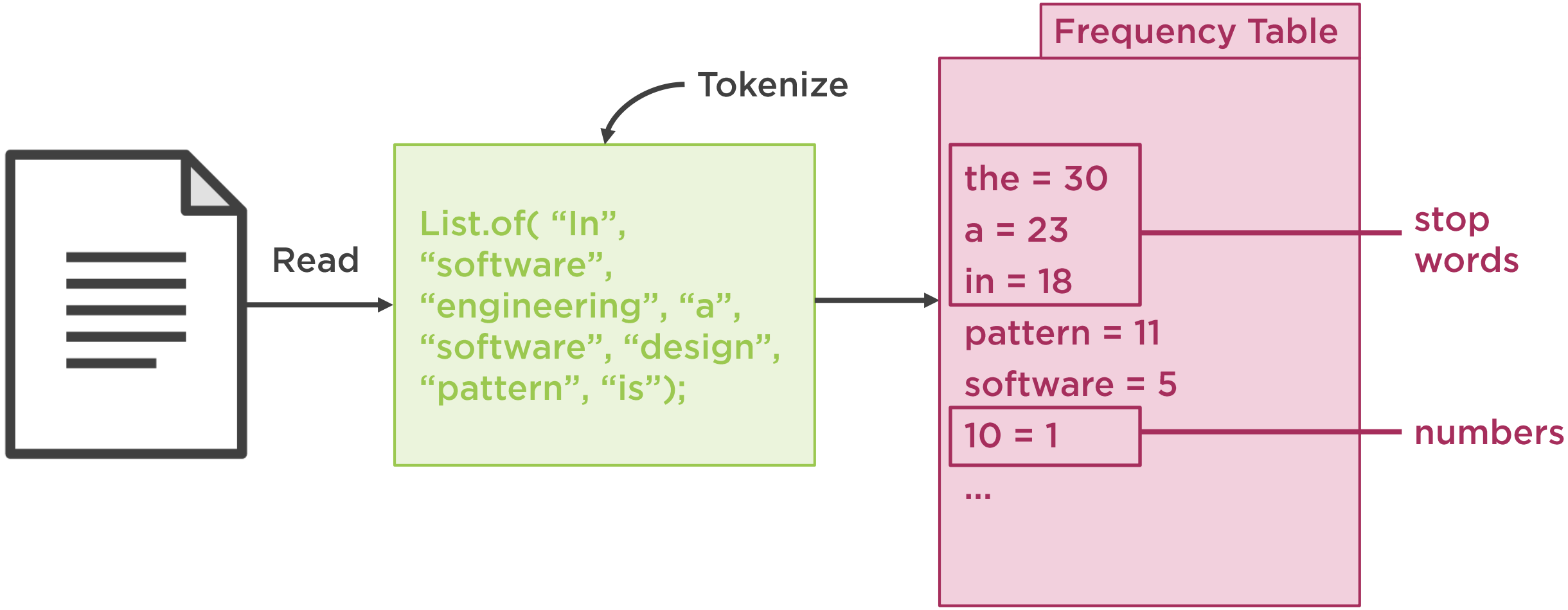



```
Context c = new Context();
c.doStuff();


c.setB(new ConcreteB2());
c.doStuff(); // behavior changed
```

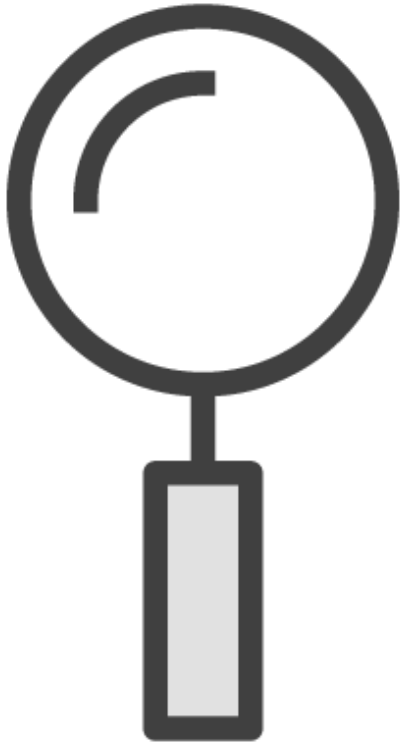

Car { }
B
Walk { }
A
PublicTransport { }

Future Requirements

Text Statistics!


Read
Tokenize
List.of(“In”, “software”, “engineering”, “a”, “software”, “design”, “pattern”, “is”);
Frequency Table
the = 30
a = 23
in = 18
pattern = 11
software = 5
10 = 1
...
stop words
numbers

Strategy Pattern Benefits and Drawbacks



Pros:

- No branching code
- Improved encapsulation
- Swap behavior at runtime
- OCP is respected

Cons:

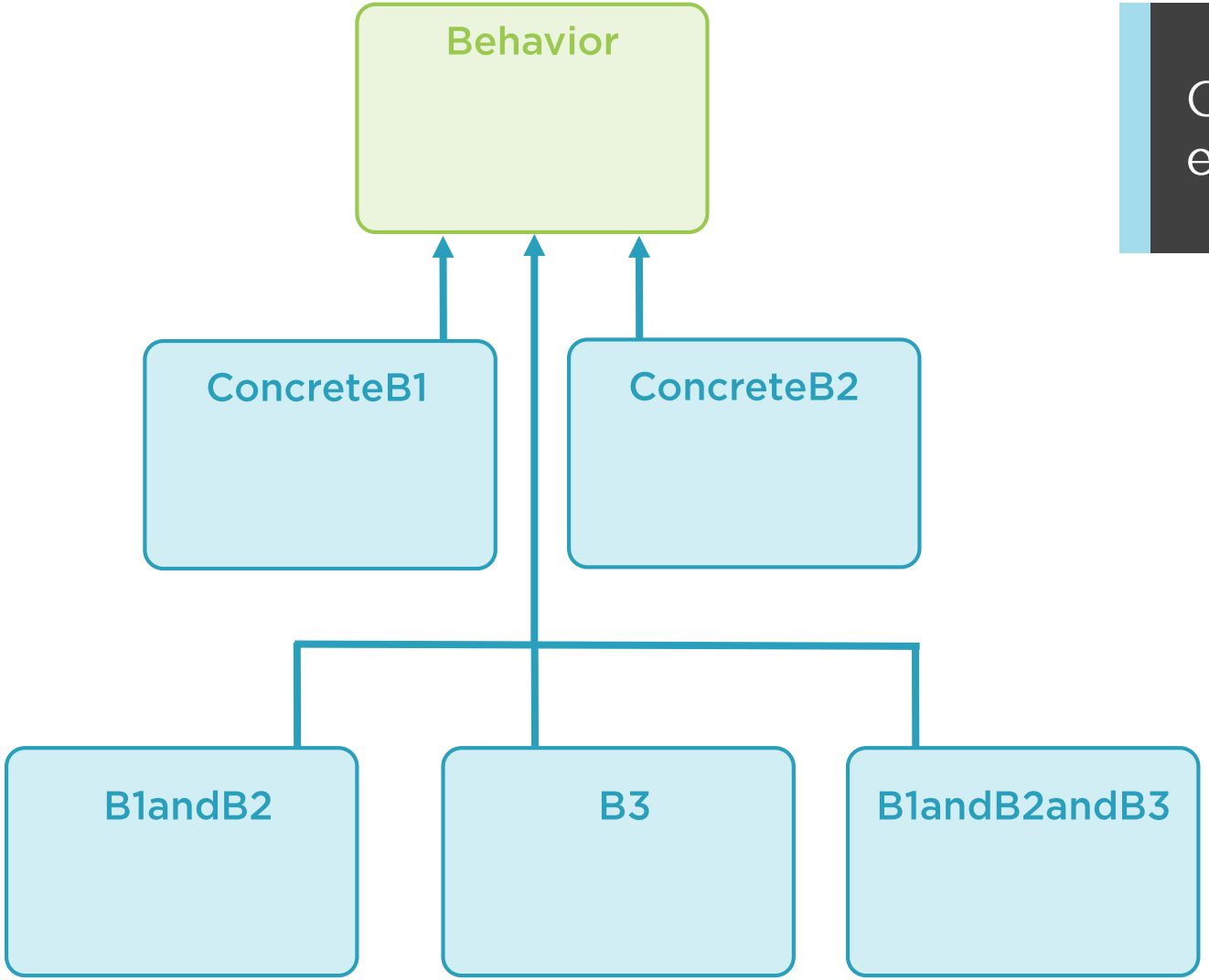
- More complicated design (more classes)

```
Behavior b1 = // no stop words
Behavior b2 = // no numbers
Behavior b3 = // b1 && b2
Behavior b4 = // b1 && b2 && other
```

Strategy is no longer an adequate solution

Use decorator for mix’n’match situations

Class (combinatorial) explosion



Decorator Example

```
open("f.txt", "r")
```

Python

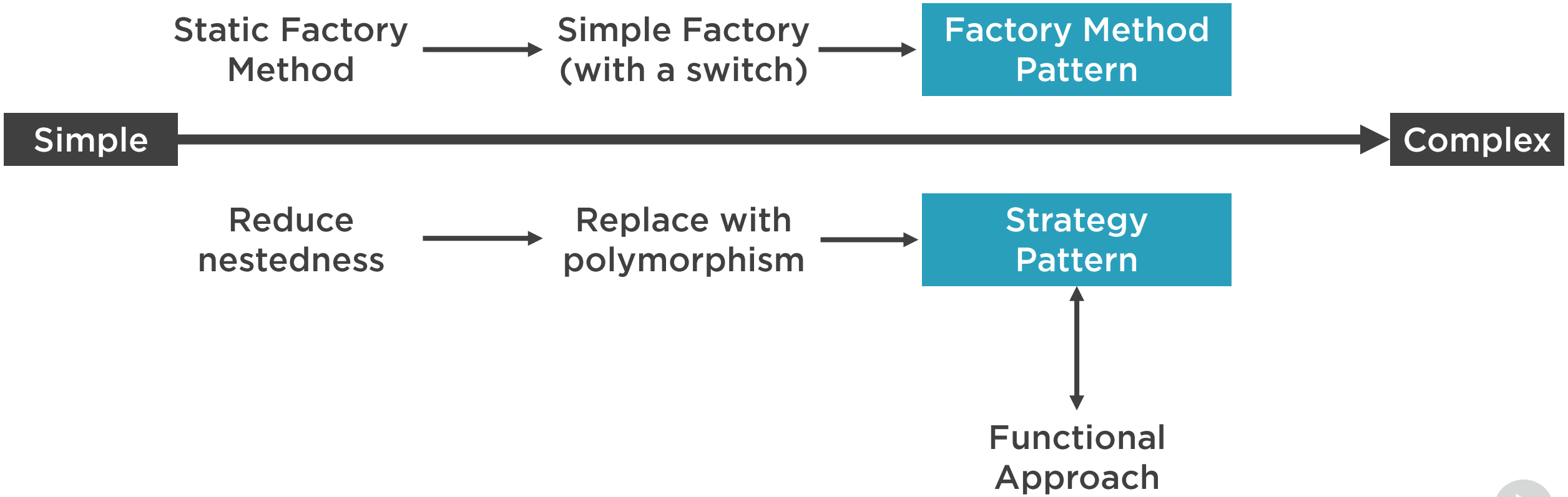
```
new BufferedReader(new FileReader(new File("f.txt")));
```

Since
Java 7

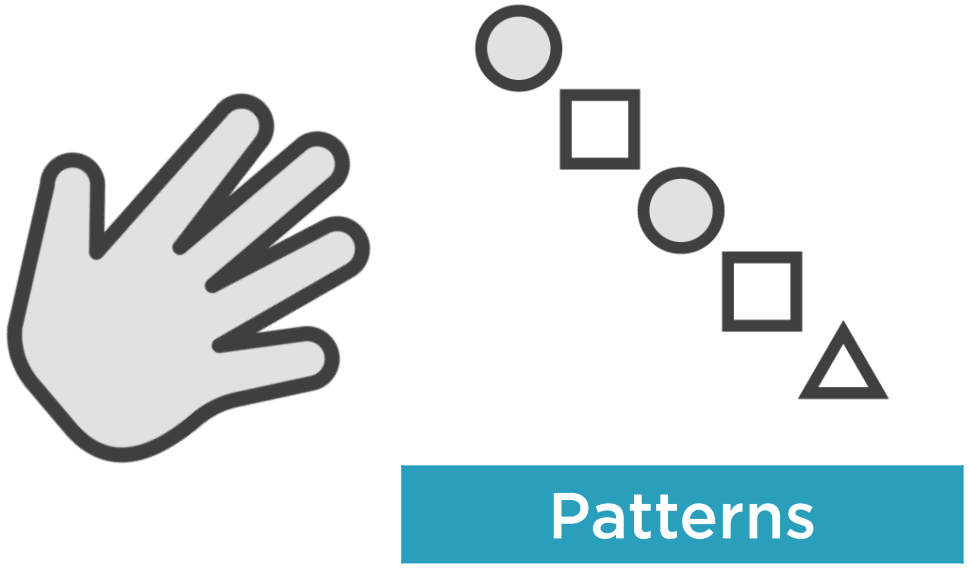
```
Files.readString(path, UTF_8);
```

Let's try
Functional Programming

~~new NoNumbers(new NoStopWords(new AllWords()));~~


Static Factory Method
Simple Factory (with a switch)
Factory Method Pattern
Simple
Complex
Reduce nestedness
Replace with polymorphism
Strategy Pattern
Functional Approach

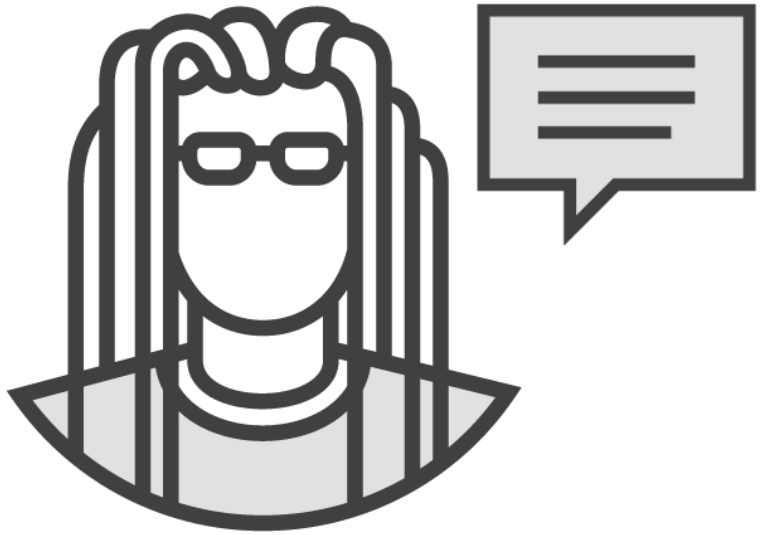
Handle with Care!

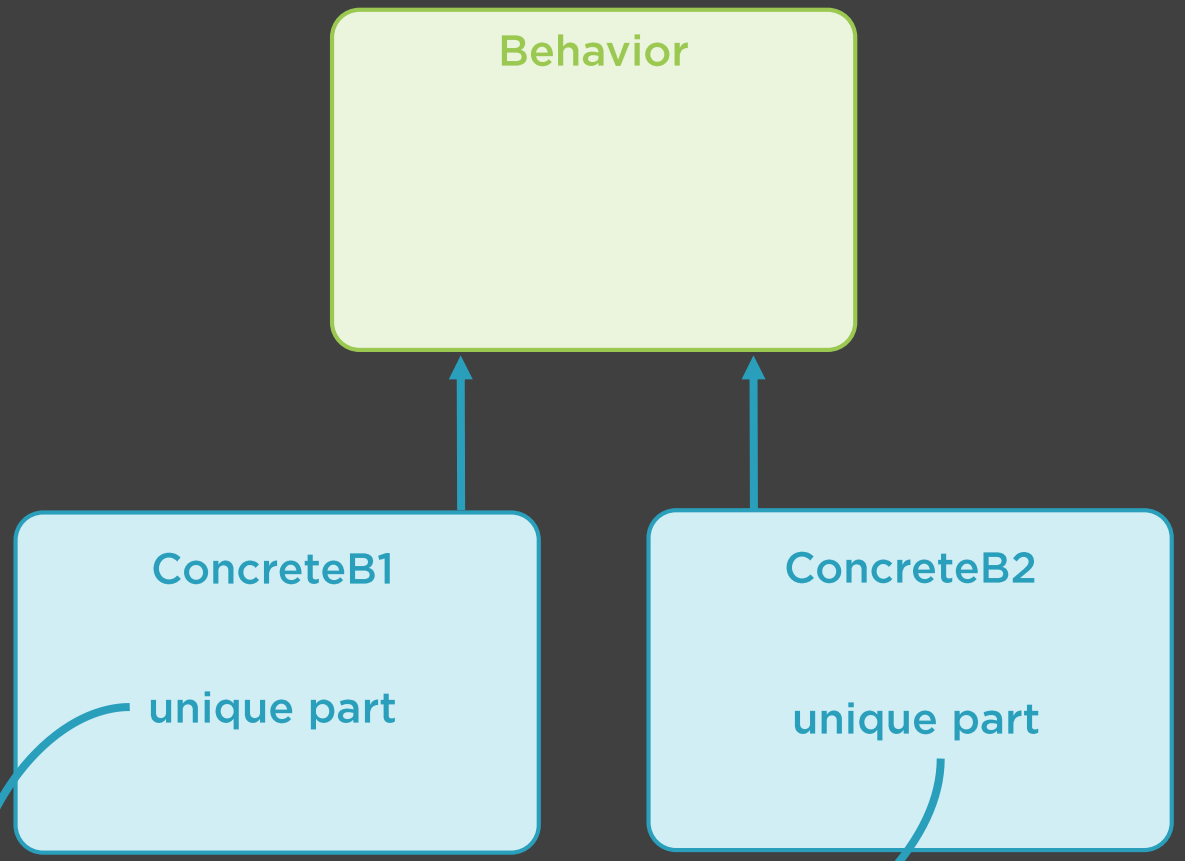


Over-application of design patterns leads to **overcomplicated** code

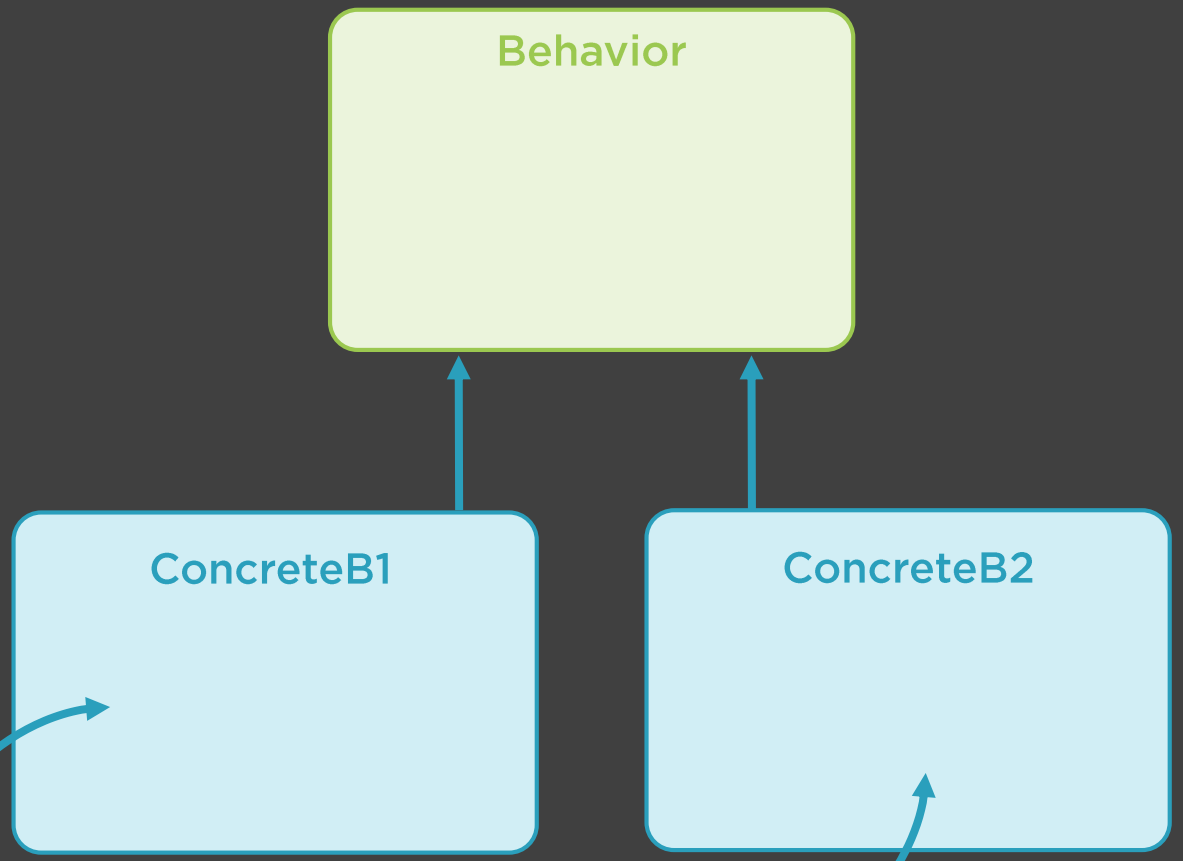
Further Material

Depends... I use both





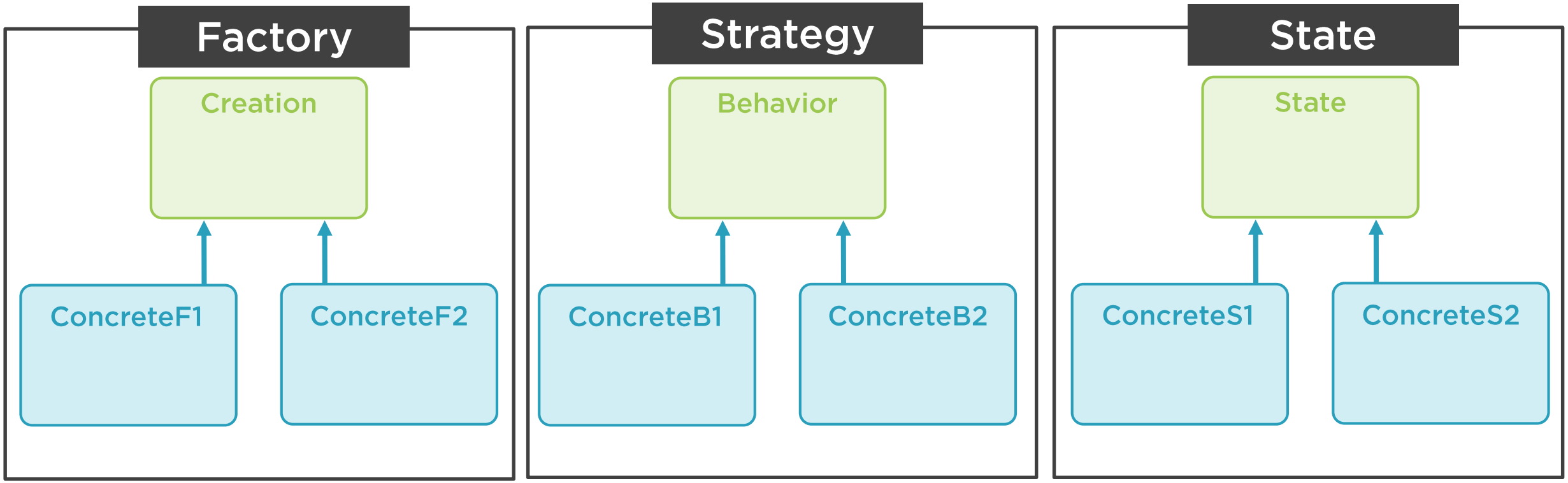

```
Predicate<T> p1 = // 1-2 line lambda
Predicate<T> p2 = // 1-2 line lambda
(or a Function<T,R>)
```



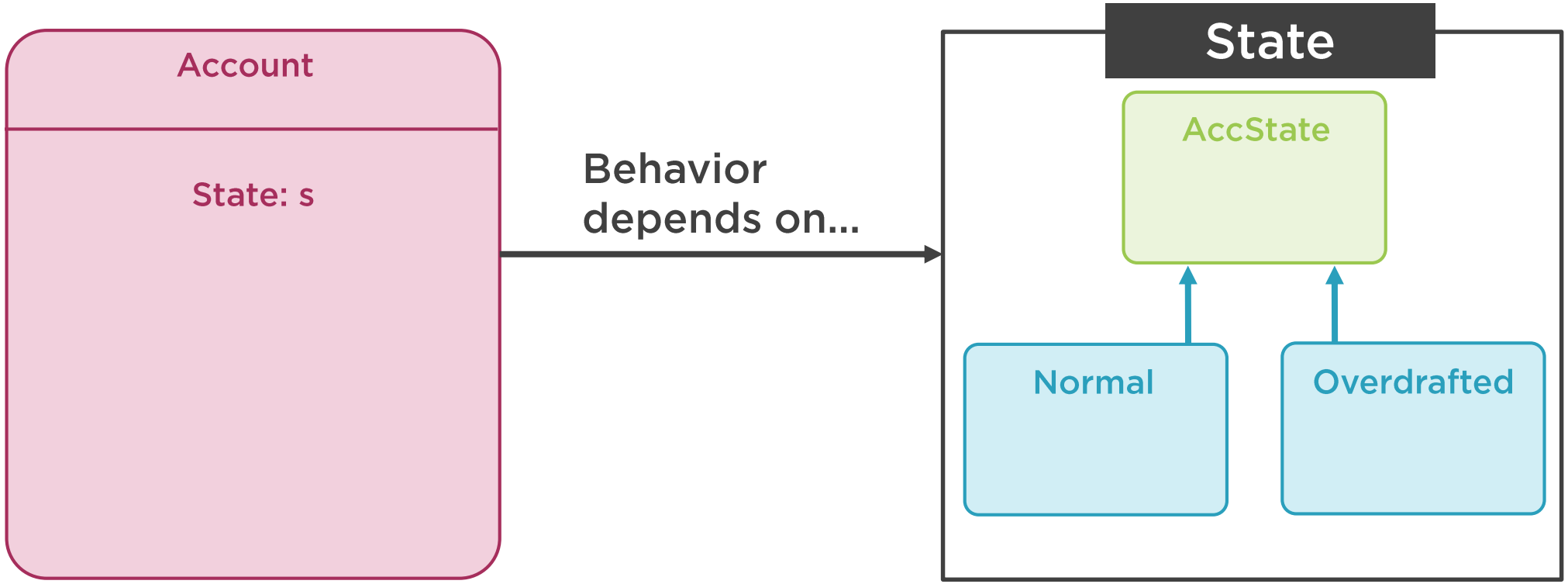

```
Function<T,R> f1 = // complex 5+ lines
Function<T,R> f2 = // complex 10+ lines
```

Long and complex? Class!

Short and simple? Lambda!



Polymorphism everywhere!


Account
State: s
Behavior
depends on...
State
AccState
Normal
Overdrafted

Summary

Many ways to reduce conditional complexity

Try to use simple techniques first

Replace branching with polymorphism is often enough

Strategy (behavior) or State (state)

Object-Oriented vs. Functional

- Can be complementary, not opposing
- Right tool for the right job