Design Patterns

CS350 Introduction to Software Engineering

Shin Yoo

(for some reason, it is really difficult NOT to talk about architecture in CS350…)

Christopher Alexander

1936 ~ 2022

- British-American Architect / Design Theorist
- Author of “A Pattern Language: Towns, Buildings, Construction”, along with co-authors Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel.

“Suppose you want to design a college campus. You must delegate some of the design to the students and professors, otherwise the Physics building won't work well for the physics people. No architect knows enough about about what physics people need to do it all themselves. But you can't delegate the design of every room to its occupants, because then you'll get a giant pile of rubble.

How can you distribute responsibility for design through all levels of a large hierarchy, while still maintaining consistency and harmony of overall design? This is the architectural design problem Alexander is trying to solve, but it's also a fundamental problem of computer systems development.”

Mark Domains, “Design Patterns Aren’t” (https://perl.plover.com/yak/design/)

“A Pattern Language”

- Alexander breaks down common elements of towns and buildings.
- The common elements become the entries of this “dictionary”.
  - For example, at the beginning of the section for buildings, we have:
    - 95. Building Complex
    - 96. Number of Stories
    - 97. Shielded Parking
    - 98. Circulation Realms
    - 99. Main Building

“A Pattern Language”

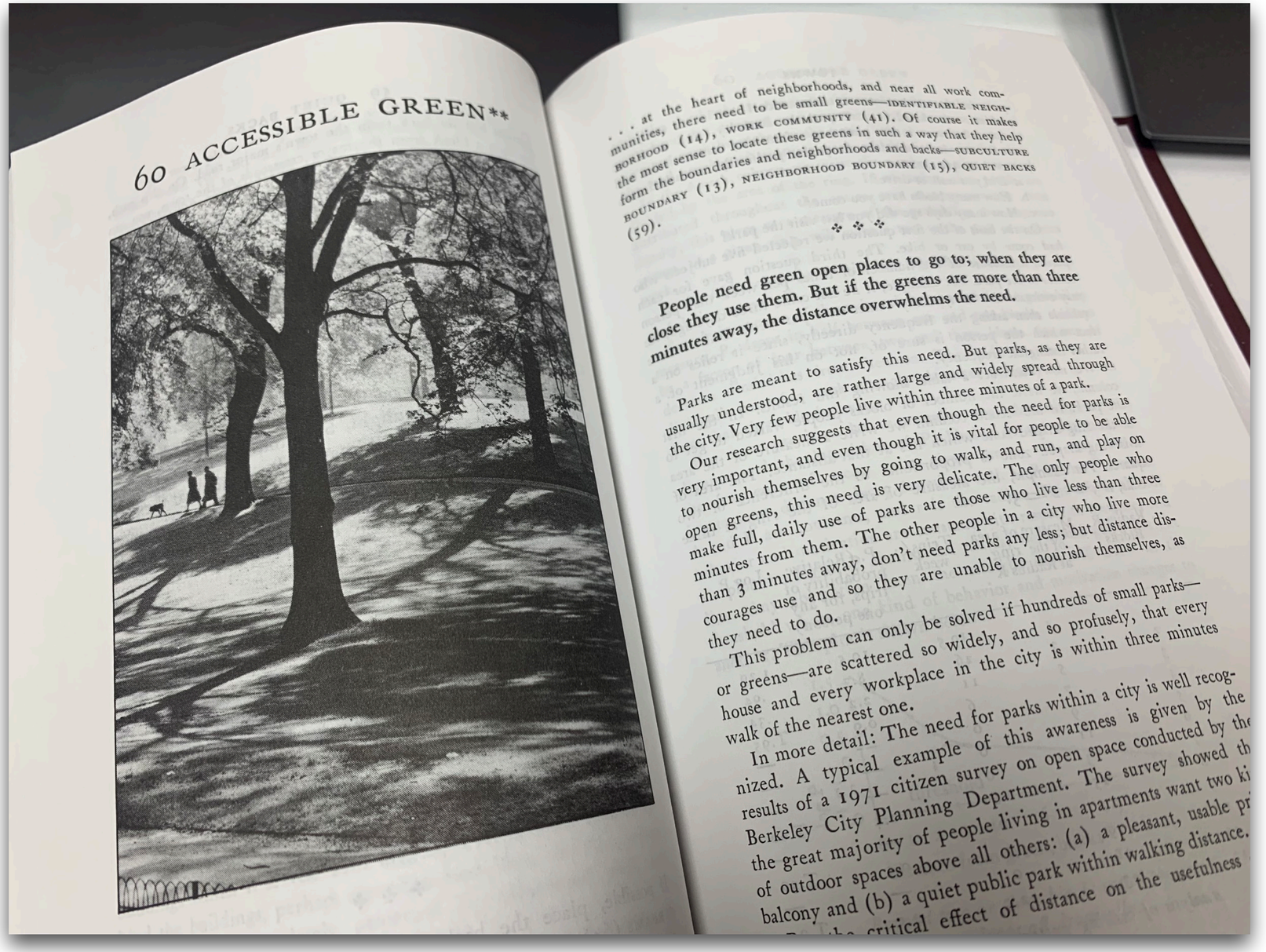
- “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.”

- “(…) we have tried, in each solution, to capture the invariant property common to all places which succeed in solving the problem.

  But of course, we have not always succeeded. The solution we have given to these problems vary in significance. Some are more true, more profound, more certain, than others. To show this clearly we have marked every pattern, in the text itself, with two asterisks, or one asterisk, or no asterisk.”

“A Pattern Language”

- “In the patterns marked with two asterisks, we believe that we have succeeded in stating a true invariant: in short, that the solution we have stated summarises a property common to all possible ways of solving the stated problem.”
- “In the patterns marked with one asterisk, we believe that we have made some progress towards identifying such an invariant: but that with careful work it will certainly be possible to improve on the solution.”
- “In the patterns without an asterisk, we are certain that we have not succeeded in defining a true invariant — that, on the contrary, there are certainly ways of solving the problem different from the one which we have given.”



60 ACCESSIBLE GREEN**

. . . at the heart of neighborhoods, and near all work communities, there need to be small greens—IDENTIFIABLE NEIGHBORHOOD (14), WORK COMMUNITY (41). Of course it makes the most sense to locate these greens in such a way that they help form the boundaries and neighborhoods and backs—SUBCULTURE BOUNDARY (13), NEIGHBORHOOD BOUNDARY (15), QUIET BACKS (59).

❖ ❖ ❖

People need green open places to go to; when they are close they use them. But if the greens are more than three minutes away, the distance overwhelms the need.

Parks are meant to satisfy this need. But parks, as they are usually understood, are rather large and widely spread through the city. Very few people live within three minutes of a park.

Our research suggests that even though the need for parks is very important, and even though it is vital for people to be able to nourish themselves by going to walk, and run, and play on open greens, this need is very delicate. The only people who make full, daily use of parks are those who live less than three minutes from them. The other people in a city who live more than 3 minutes away, don't need parks any less; but distance discourages use and so they are unable to nourish themselves, as they need to do.

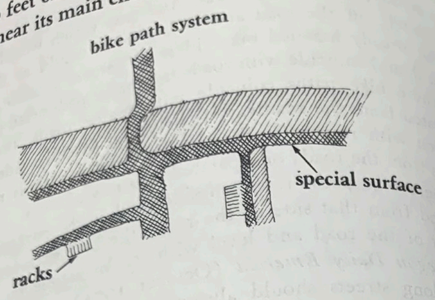
This problem can only be solved if hundreds of small parks—or greens—are scattered so widely, and so profusely, that every house and every workplace in the city is within three minutes walk of the nearest one.

In more detail: The need for parks within a city is well recognized. A typical example of this awareness is given by the results of a 1971 citizen survey on open space conducted by the Berkeley City Planning Department. The survey showed th the great majority of people living in apartments want two ki of outdoor spaces above all others: (a) a pleasant, usable pr balcony and (b) a quiet public park within walking distance.

critical effect of distance on the usefulness

Accessible Green**: people need open places to go to; when they are close they use them. But if the greens are more than three minutes away, the distance overwhelm the need. Therefore, build one open public green within three minutes' walk - about 750 feet - of every house and workplace. This means that the greens need to be uniformly scattered at 1500-foot intervals, throughout the city. Make the greens at least 150 feet across, and at least 60,000 square feet in area.

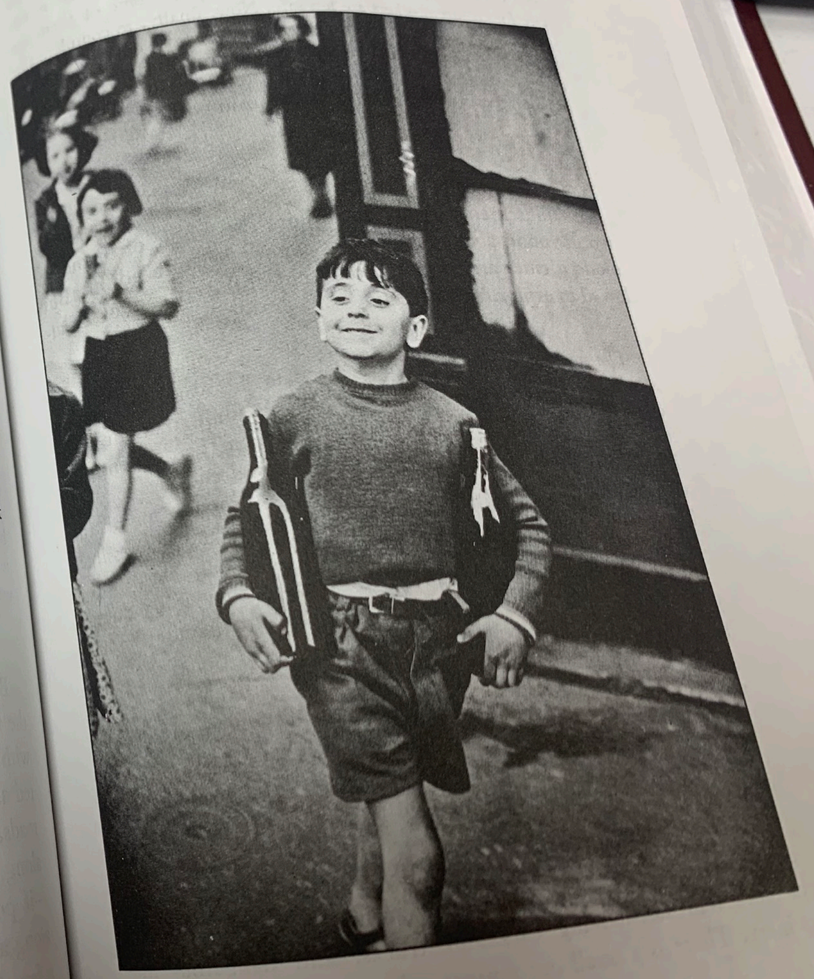
within 100 feet of every building, and give every building a bike rack near its main entrance.



❖ ❖ ❖

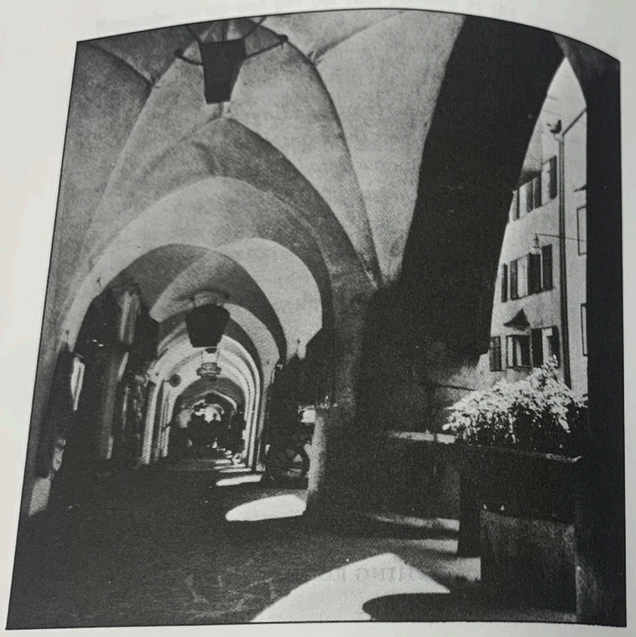
Build the racks for bikes to one side of the main entrance, so that the bikes don't interfere with people's natural movement in and out—MAIN ENTRANCE (110), and give it some shelter, with the path from the racks to the entrance also under shelter—ARCADES (119); keep the bikes out of quiet walks and quiet gardens—QUIET BACKS (59), GARDEN WALL (173). . . .

57 CHILDREN IN THE CITY



Children in the City: If children are not able to explore the whole of the adult world round about them, they cannot become adults. But modern cities are so dangerous that children cannot be allowed to explore them freely. Therefore, as part of the network of bike paths, develop one system of paths that is extra safe - entirely separate from automobiles, with lights and bridges at the crossings, with homes and shops along it, so that there are always many eyes on the path.

119 ARCADES**



. . . the CASCADE OF ROOFS (116) may be completed by arcades. Paths along the building, short paths between buildings, PEDESTRIAN STREET (100), paths between CONNECTED BUILDINGS (108), and parts of CIRCULATION REALMS (98) are all best as arcades. This is one of the most beautiful patterns in the language; it affects the total character of buildings as few other patterns do.

✣ ✣ ✣

Arcades—covered walkways at the edge of buildings, which are partly inside, partly outside—play a vital role in the way that people interact with buildings.

Buildings are often much more unfriendly than they need to be. They do not create the possibility of a connection with the public world outside. They do not genuinely invite the public in; they operate essentially as private territory for the people who are inside.

The problem lies in the fact that there are no strong connections between the territorial world within the building and the purely public world outside. There are no realms between the two kinds of spaces which are ambiguously a part of each—places that are both characteristic of the territory inside and, simultaneously, part of the public world.

The classic solution to this problem is the arcade: arcades create an ambiguous territory between the public world and the private world, and so make buildings friendly. But they need the following properties to be successful.

1. To make them public, the public path to the building must itself become a *place* that is partly inside the building; and this place must contain the character of the inside.

If the major paths through and beside the buildings are genuinely public, covered by an extension of the building, a low arcade, with openings into the building—many doors and windows and half-open walls—then people are drawn into the building; the action is on display, they feel tangentially a part of it. Perhaps they will watch, step inside, and ask a question.

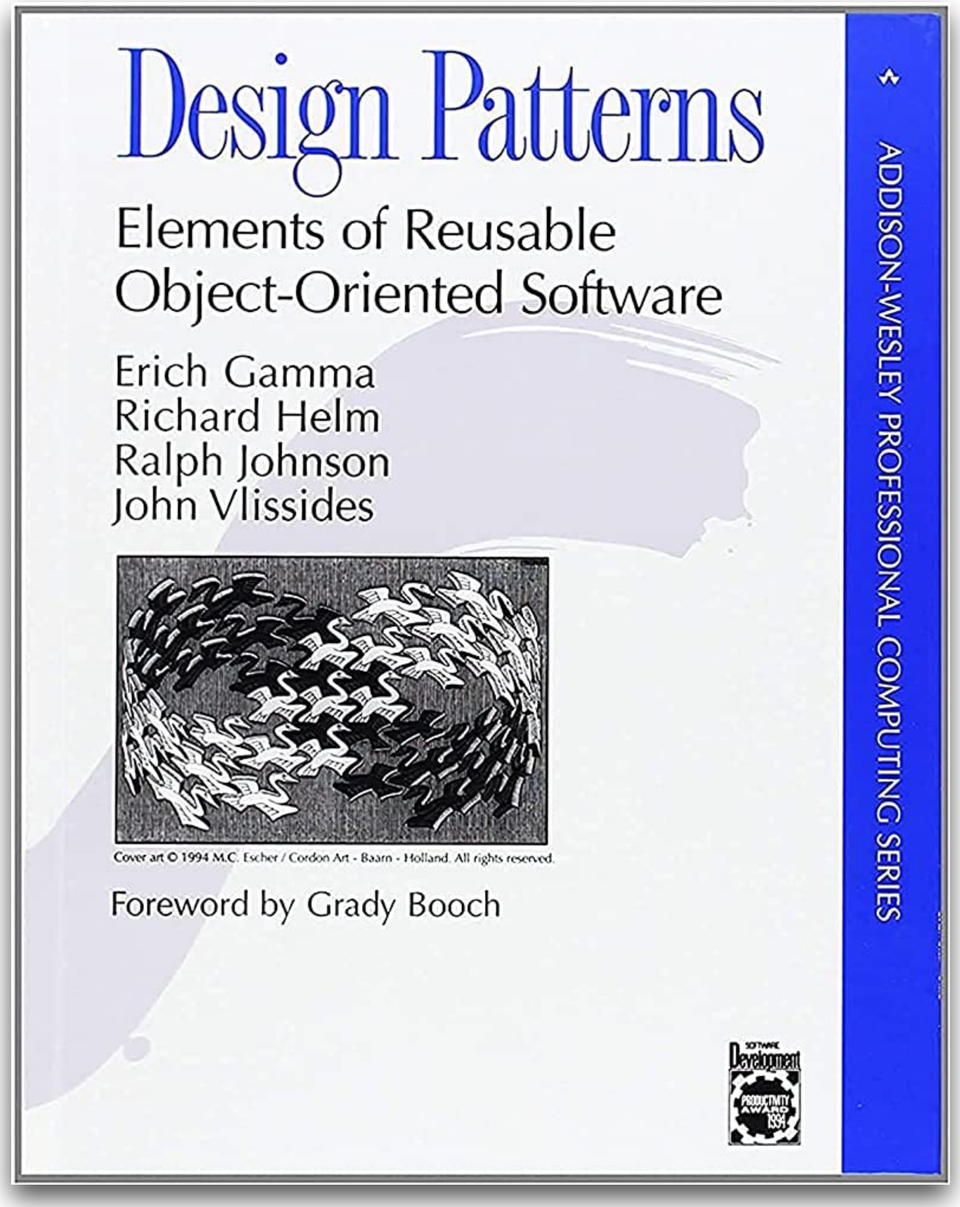
Arcades**: Arcades — covered walkways at the edge of buildings, which are partly inside, partly outside — play a vital role in the way that people interact with buildings. Therefore, wherever paths run along the edge of buildings, build arcades, and use the arcades, above all, to connect up the buildings to one another, so that a person can walk from place to place under the cover of the arcades.

What does this pattern language achieve?

- A set of common design elements AND their names, so that people know what to make decisions about.
- A proposed guidelines (invariants) on the essential property of such design elements, so that good design is achieved.
- Can we do something similar to software design?

Design Patterns

- Structural Patterns that are commonly used to solve “design problems”
  - Algorithms are, in a way, common *computational patterns* for specific functionality (sorting, storing data, etc).
  - Design Patterns are common *structural patterns* for organizing your components.
- Discussed since 70s; made popular by the “Gang of Four”: Eric Gamma, Richard Helm, Ralph Johnson, and John Vissildes via the following book:
  - Design Patterns: Elements of Reusable Object-Oriented Software (1995)

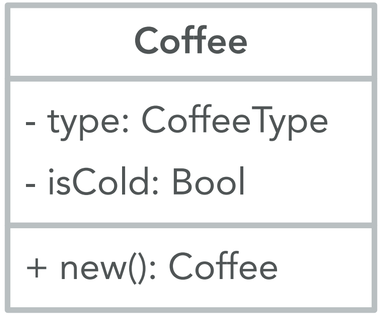


Patterns that we will browse

- Creational Patterns: how should you create objects?
- Structural Patterns: how do you assemble objects while remaining flexible?
- Behavioral Patterns: how do you distribute tasks to objects to implement algorithms?

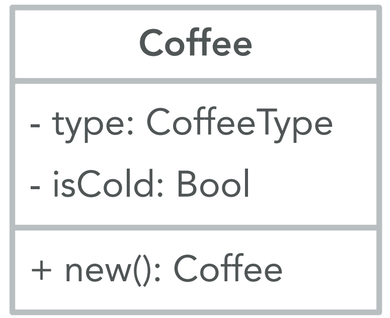
Creational Patterns: Factory

- **Problem:** if you create instances of a class directly using its constructor in multiple locations, you increase the coupling between all these locations and the constructor. Later, if you need to alter the type of objects being created, the high coupling makes the change difficult.
- For example, you built a software for managing a cafe, which initially only sold coffees.



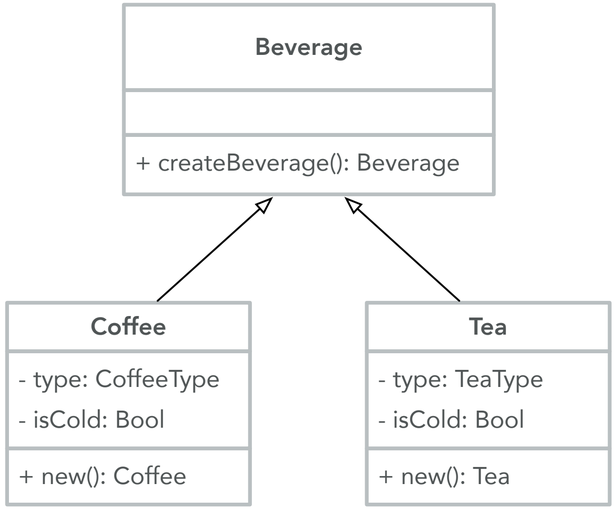
Creational Patterns: Factory

- **Problem:** if you create instances of a class directly using its constructor in multiple locations, you increase the coupling between all these locations and the constructor. Later, if you need to alter the type of objects being created, the high coupling makes the change difficult.
- For example, you built a software for managing a cafe, which initially only sold coffees.
- Later, you decide to sell teas too??

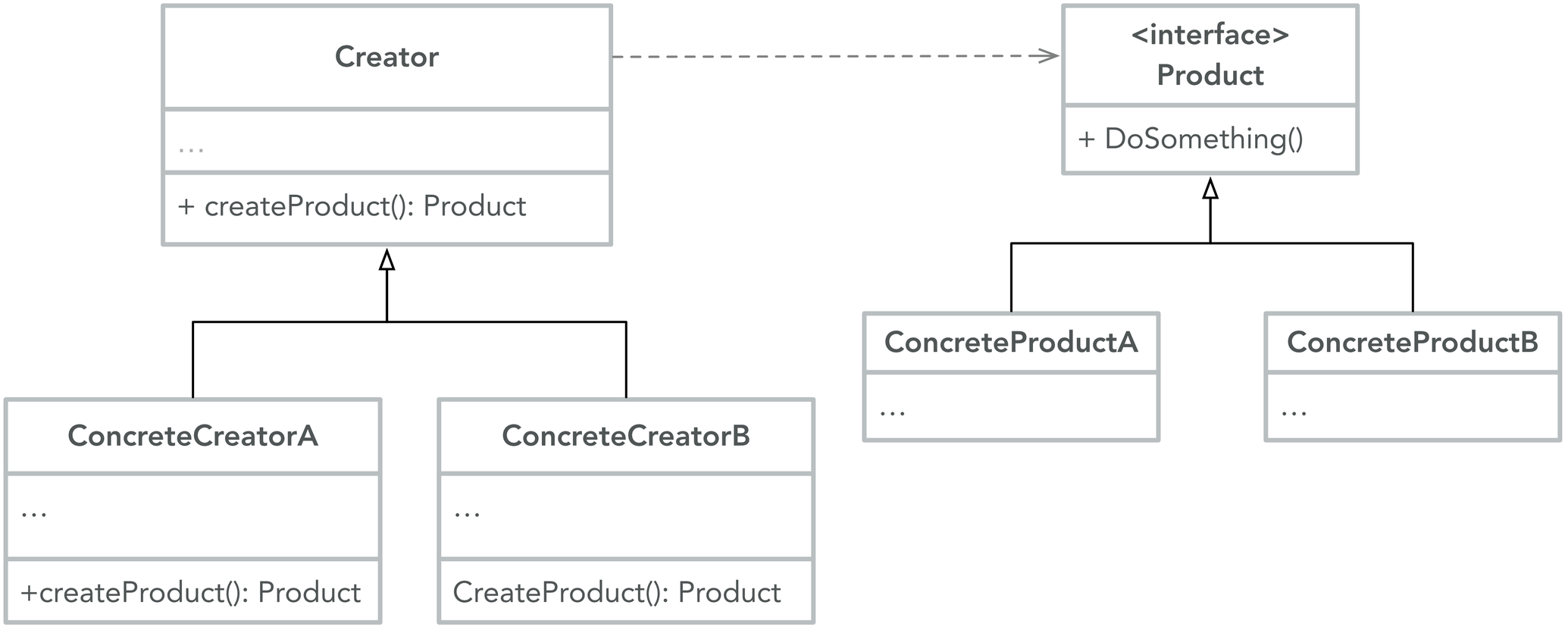


Creational Patterns: Factory

- **Solution:** create a special Factory method, which will take the required inputs (such as “coffee or tea”) and return a valid subclass.
- You always use the factory method to create instances of subclasses.
- Limitations: only works when various objects you want to create has a common base class/interface.



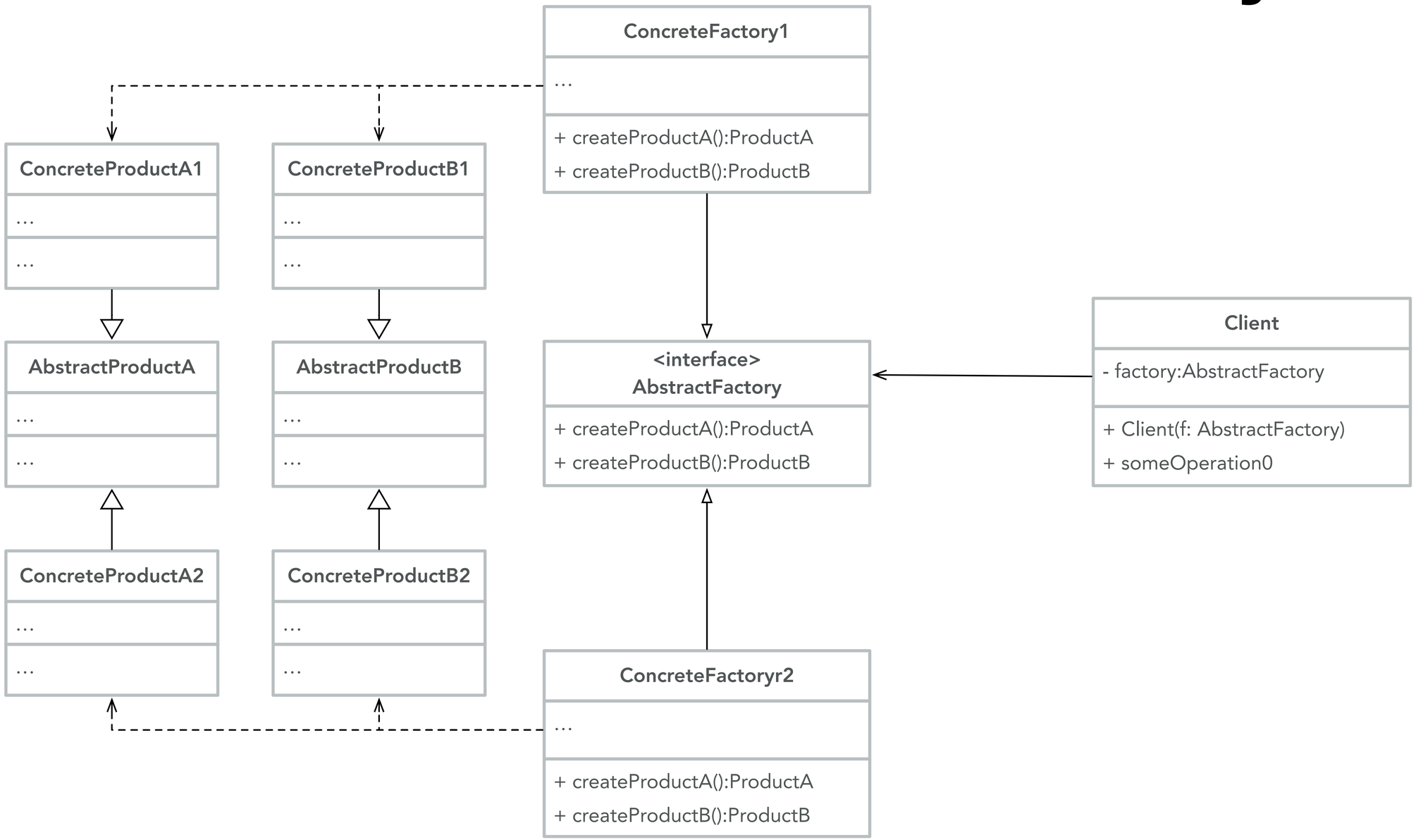
Creational Patterns: Factory



Creational Patterns: Abstract Factory

- What if the whole “line-up” of Product A, B, and C can be varied, but the operations you perform on products are the same?
  - Your cafe chain sells sets of coffees and teas in Korea, but a different line-up of coffees and teas in Italy.
  - What you do to Beverage objects remain the same, regardless of the location of cafes.
- We need different factories for different lineups.

Creational Patterns: Abstract Factory

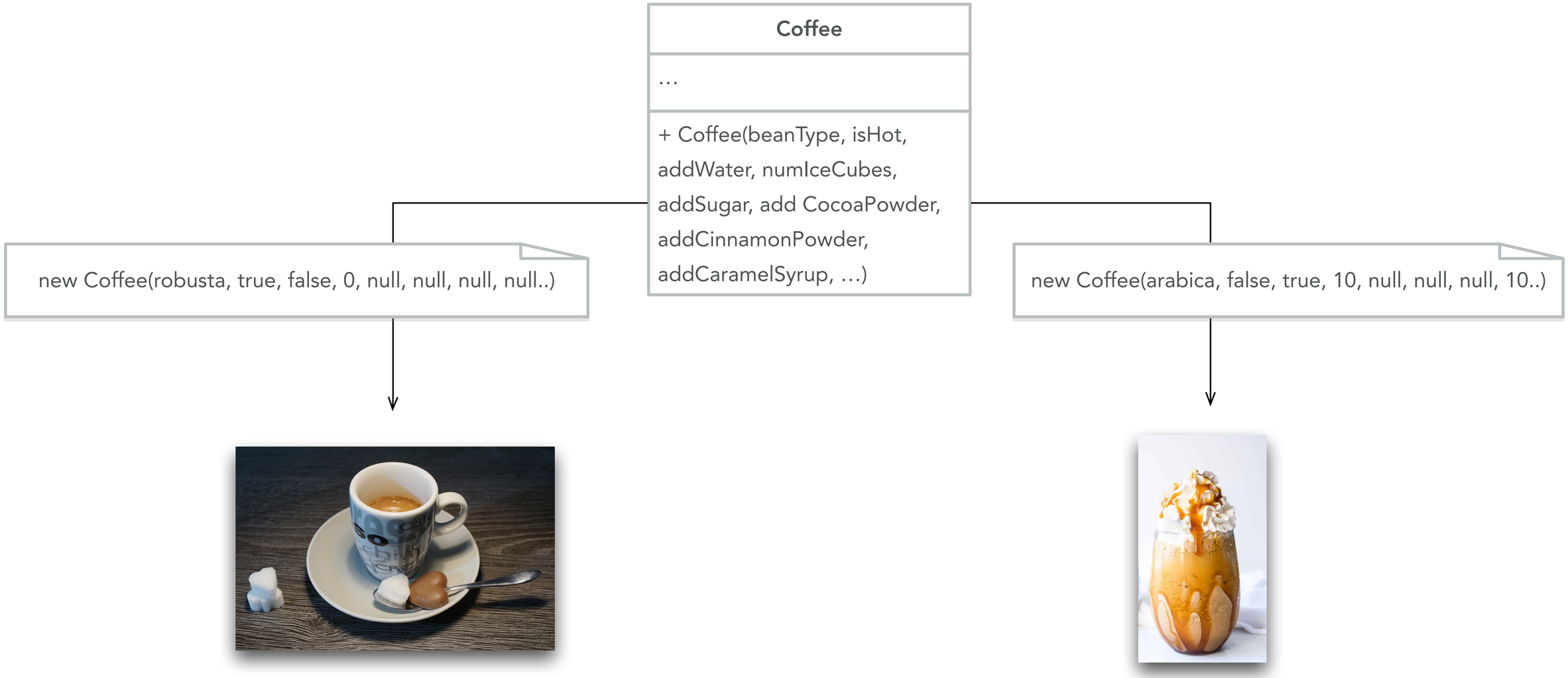


Creational Patterns: Builder

- **Problem**: the object you want to create requires a long list of parameters to be initialised. Further, depending on the combinations in these parameters, you get slightly different objects.
  - Different subclass for different combinations of parameters: complicated type hierarchy, difficult to modify later.
  - A giant constructor with all possible parameters: many parameters will be left unused

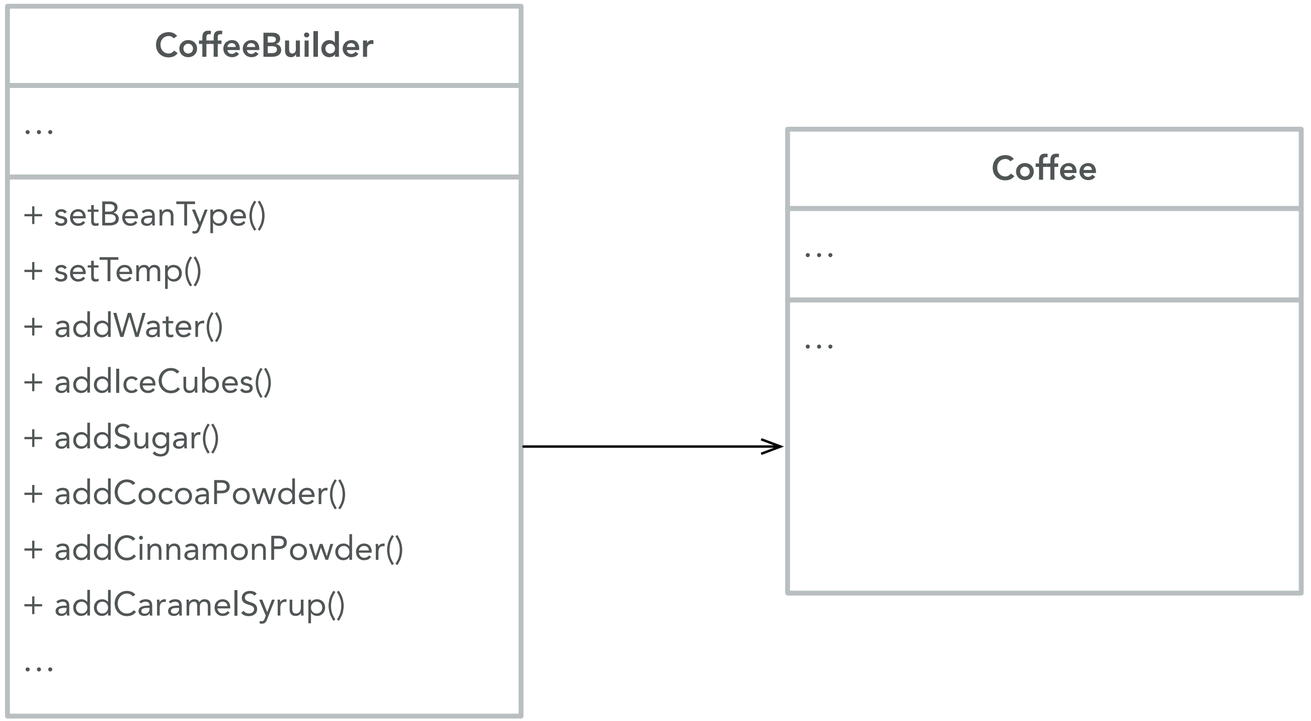
Creational Patterns: Builder

The problem with long constructor calls



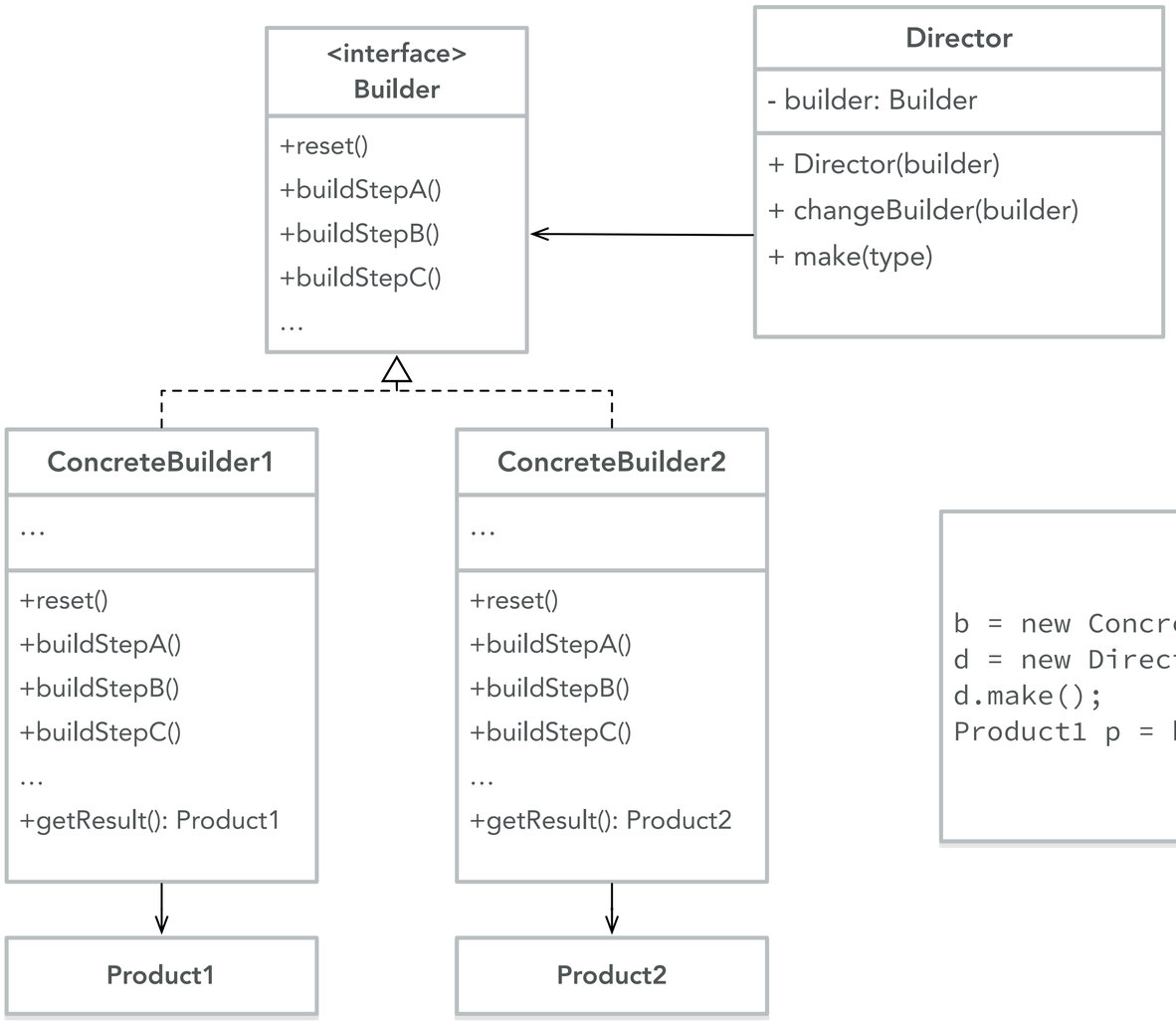
Creational Patterns: Builder

- Extract the object construction to another class; break down the constructor argument into separate steps.



Creational Patterns: Builder

You can even create specialized builders for sub-categories



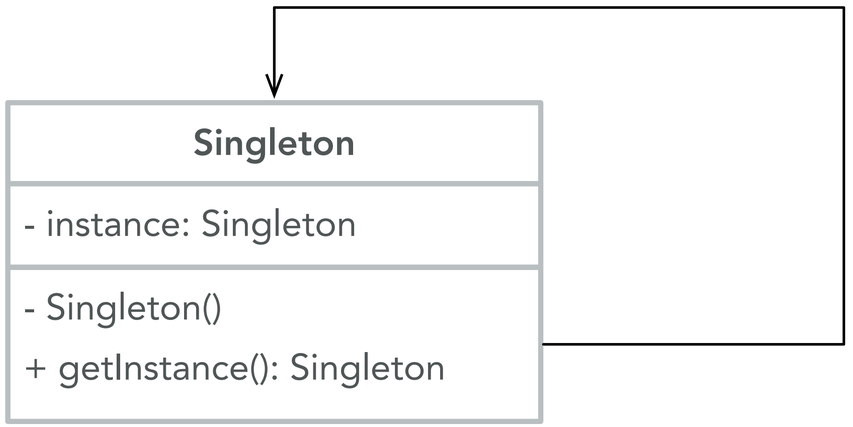
A “Director” that knows how to build (=make) a specific type of product using a builder can be added.

```
b = new ConcreteBuilder1();
d = new Director(b);
d.make();
Product1 p = b.getResult();
```

Creational Patterns: Singleton

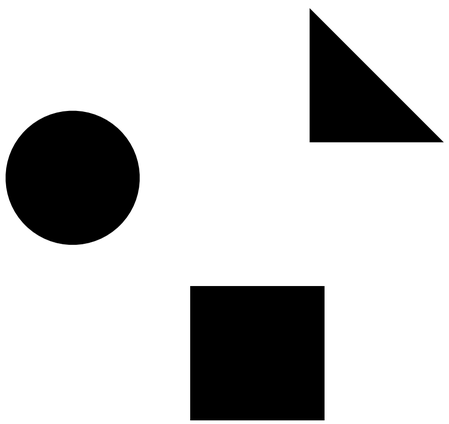
- **Problem**: you want to ensure that a class only ever has a single instance.
  - A class that controls the access to a single shared resource (e.g., DB) or contains a single copy of information that needs to be accessed globally (think of global variables)

```
if (instance==null){
  this.instance = new Singleton();
}
return singleton;
```



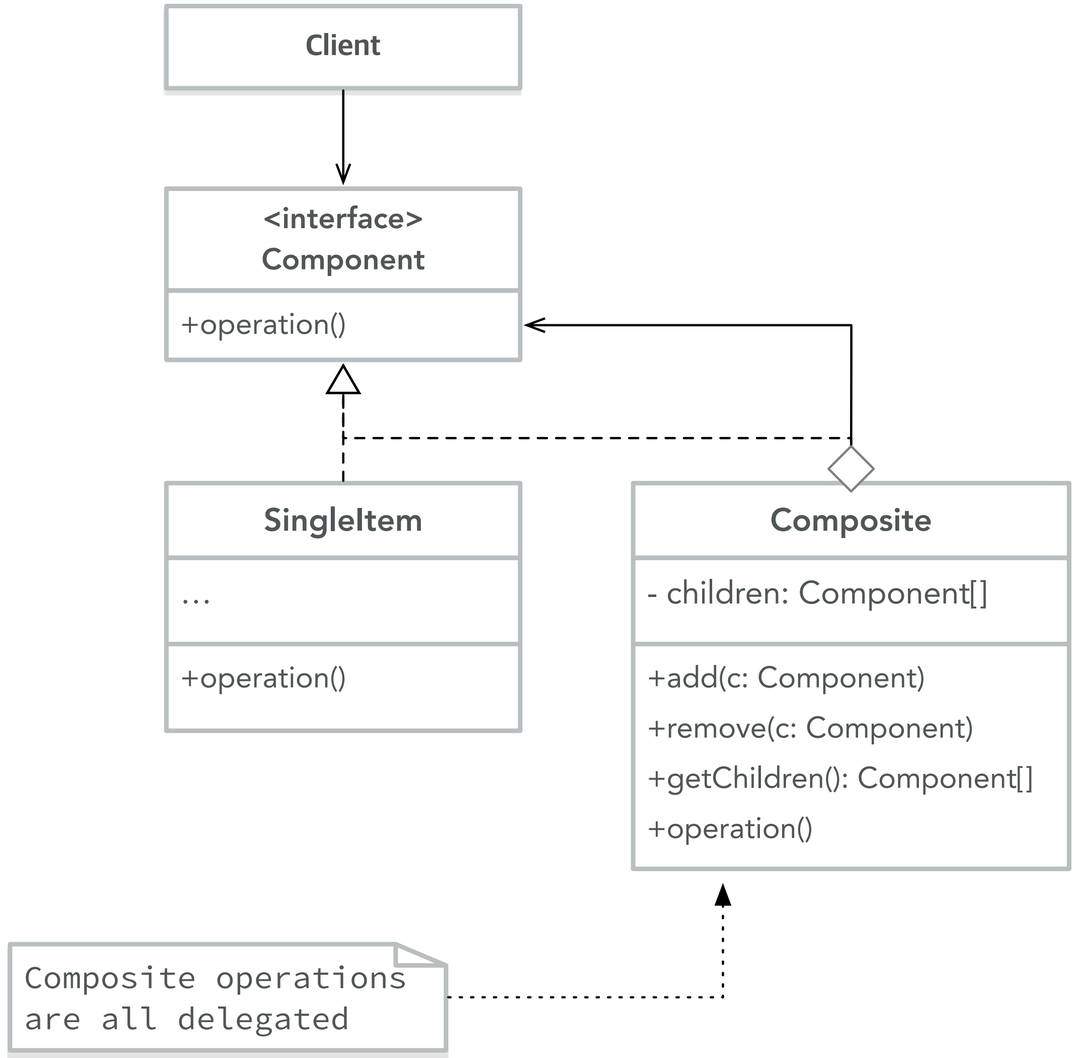
Structural Patterns: Composite

- **Problem:** you need to manipulate both individual objects and groups of them - think of grouped shapes in PowerPoint/Keynote. Any operation we can perform to individual objects (e.g., resize) should also work for the composite object. How do we do this efficiently?



Structural Patterns: Composite

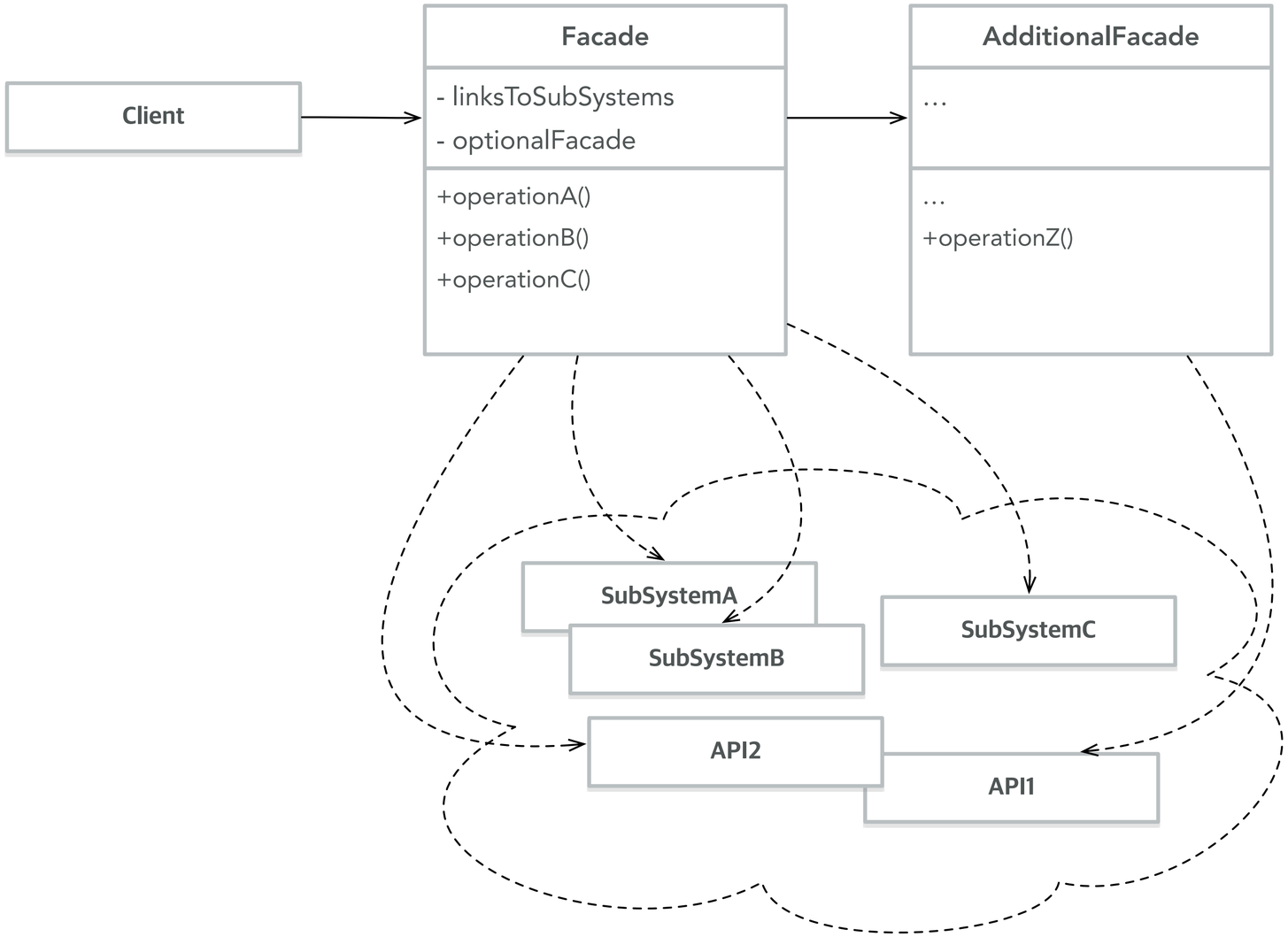
- Operations are designed so that anything done to a composite can be delegated into individual children.



Structural Patterns: Facade

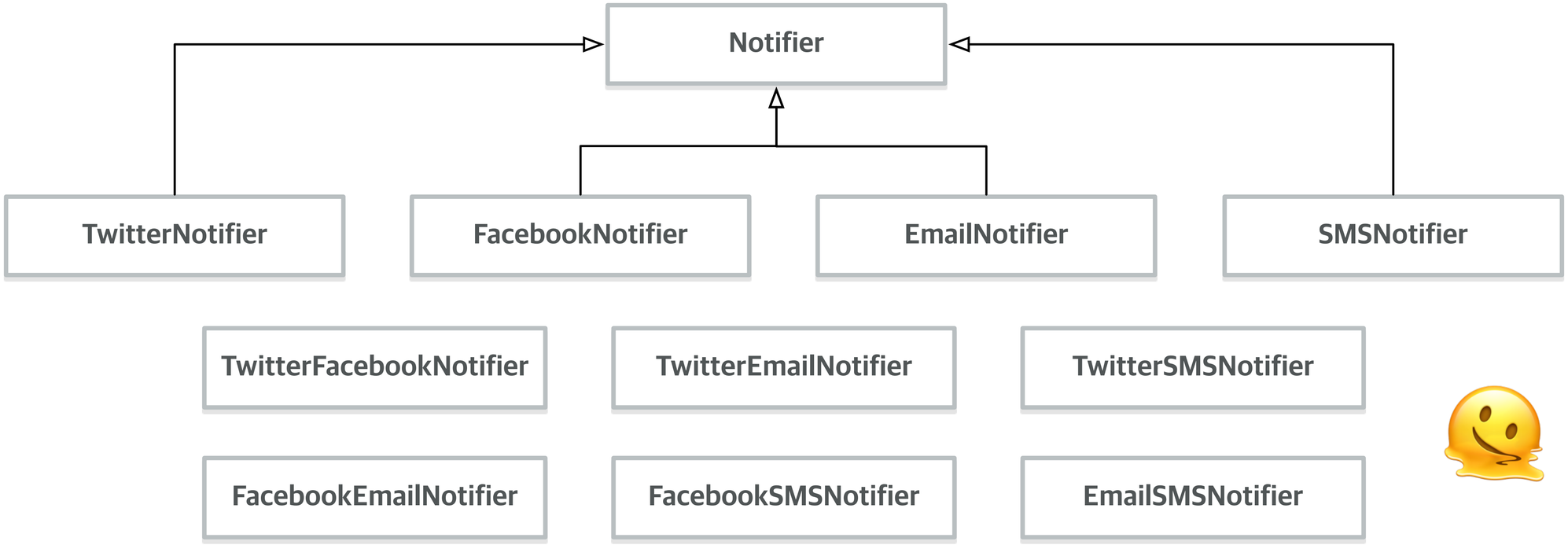
- **Problem**: your system depends on a complex, behind-the-scene libraries and frameworks. However, directly connecting your business logic to multiple libraries and frameworks will introduce unnecessary coupling.
  - Comprehension becomes more challenging.
  - Changing one of the libraries later becomes more difficult.
- Implement a facade (front/face of a building) that hides the complexity

Structural Patterns: Facade



Structural Patterns: Decorator (Wrapper)

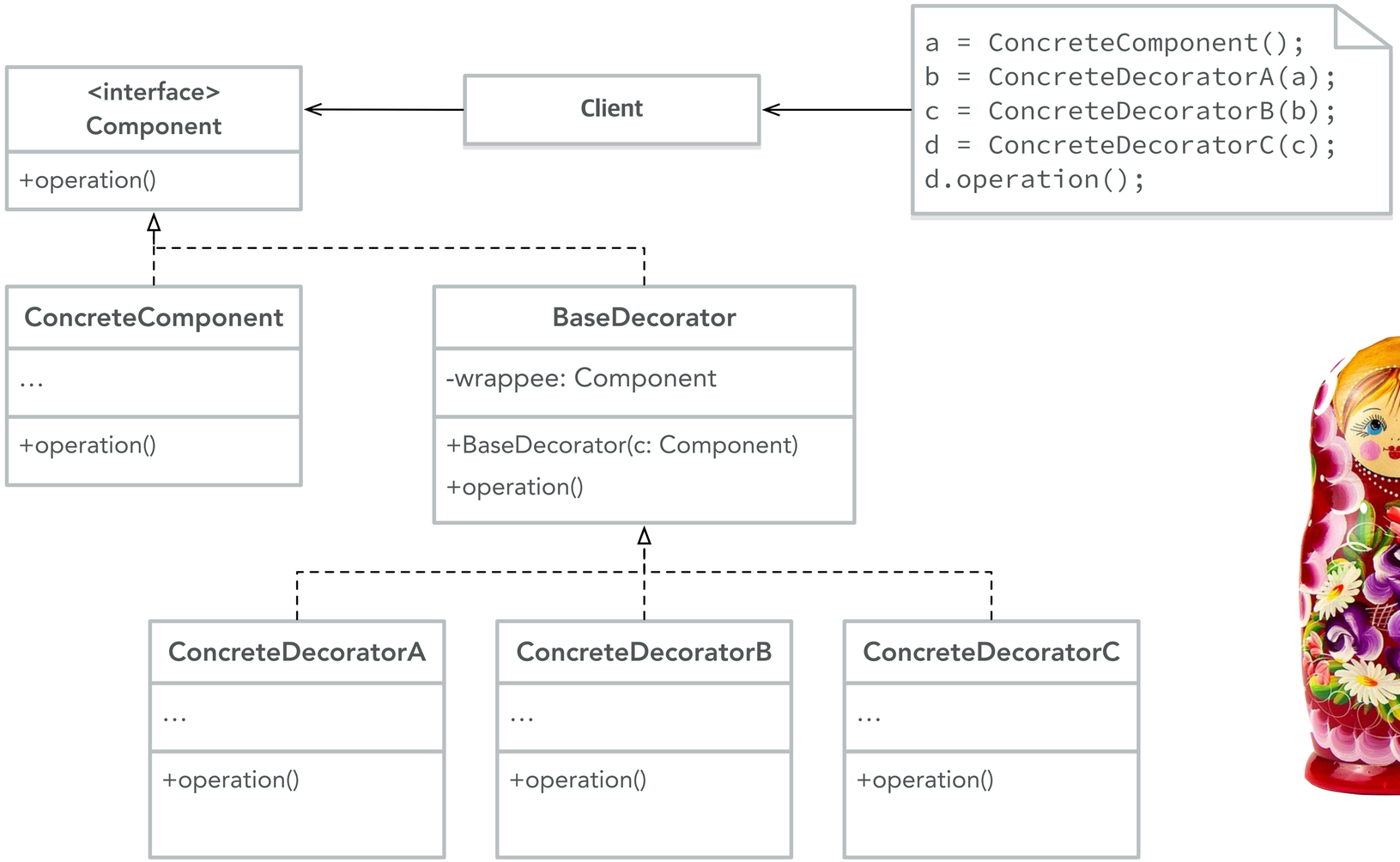
- **Problem**: you need to modify the behaviour of a specific class in a number of ways. However, if you implement it via inheritance, you end up with too many subclasses.



Structural Patterns: Decorator (Wrapper)

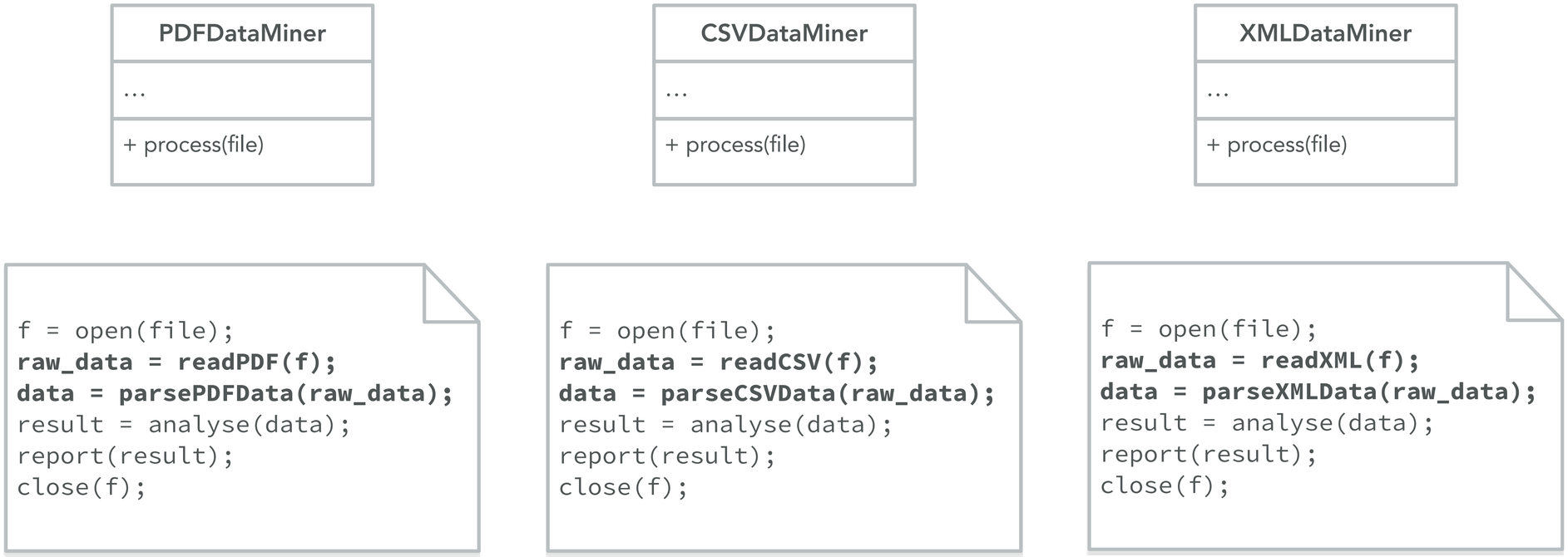
- In general, handling variation always with inheritance is cumbersome; instead, try using aggregation (i.e., chaining objects with variations)+ delegation (call each of the chained objects).
- Each object “wraps” another object that performs a similar operation: before/after calling the target method, each object does a little bit of its own thing.

Structural Patterns: Decorator (Wrapper)



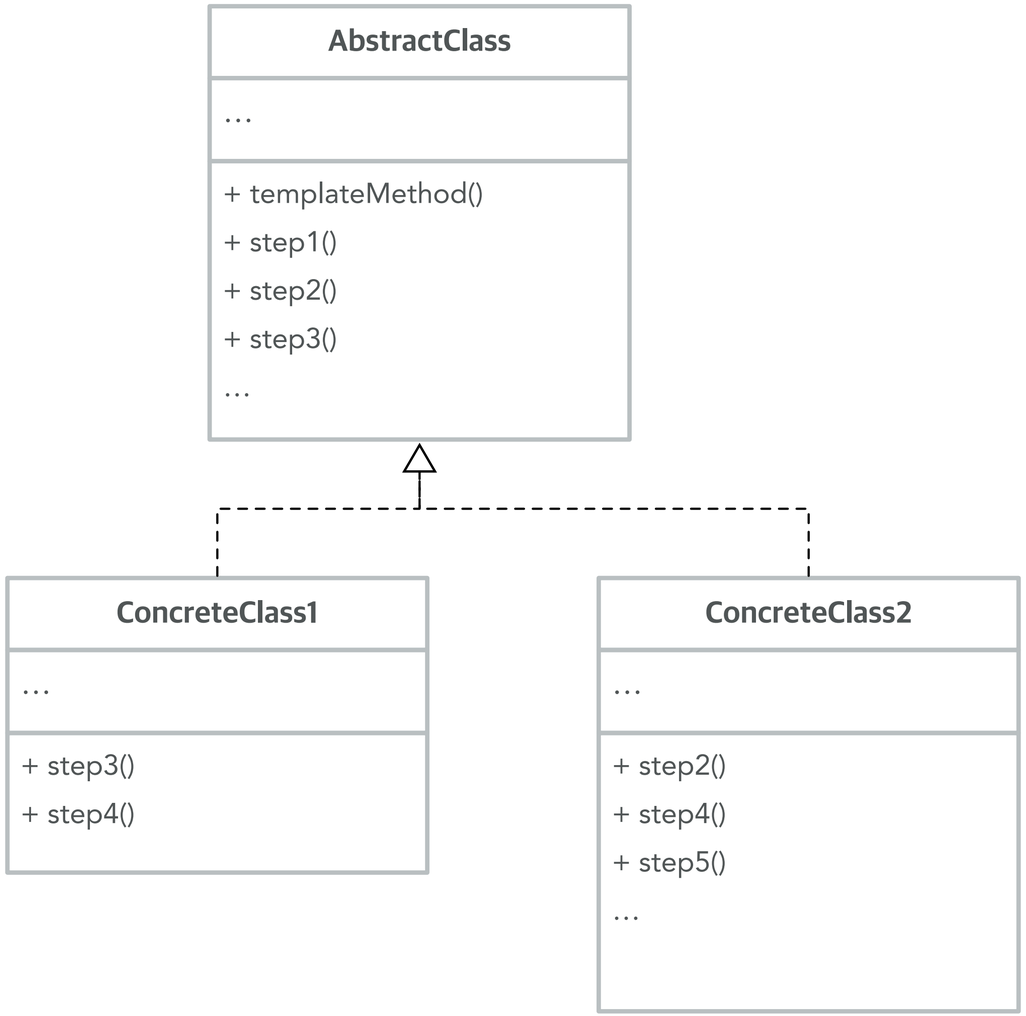
Behavioural Patterns: Template

- **Problem**: you have various classes that perform variations of the same task.



Behavioural Patterns: Template

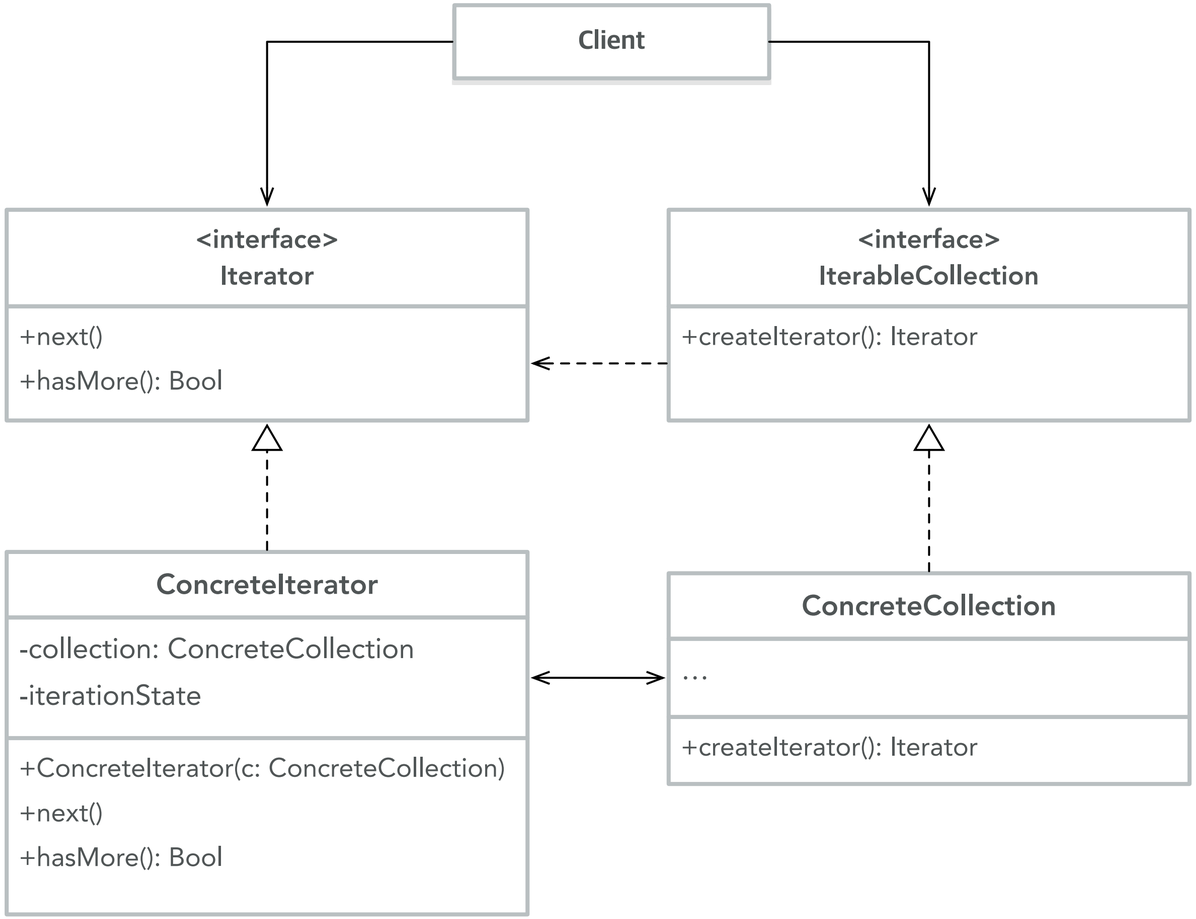
Subclasses have to override what is needed only



Behavioural Patterns: Iterator

- You should be able to iterate over a collection, without being concerned with the lower level data structure.
    - If you decide that your underlying data structure should change from a linked list to a binary tree, your algorithm should not be affected.

Behavioural Patterns: Iterator

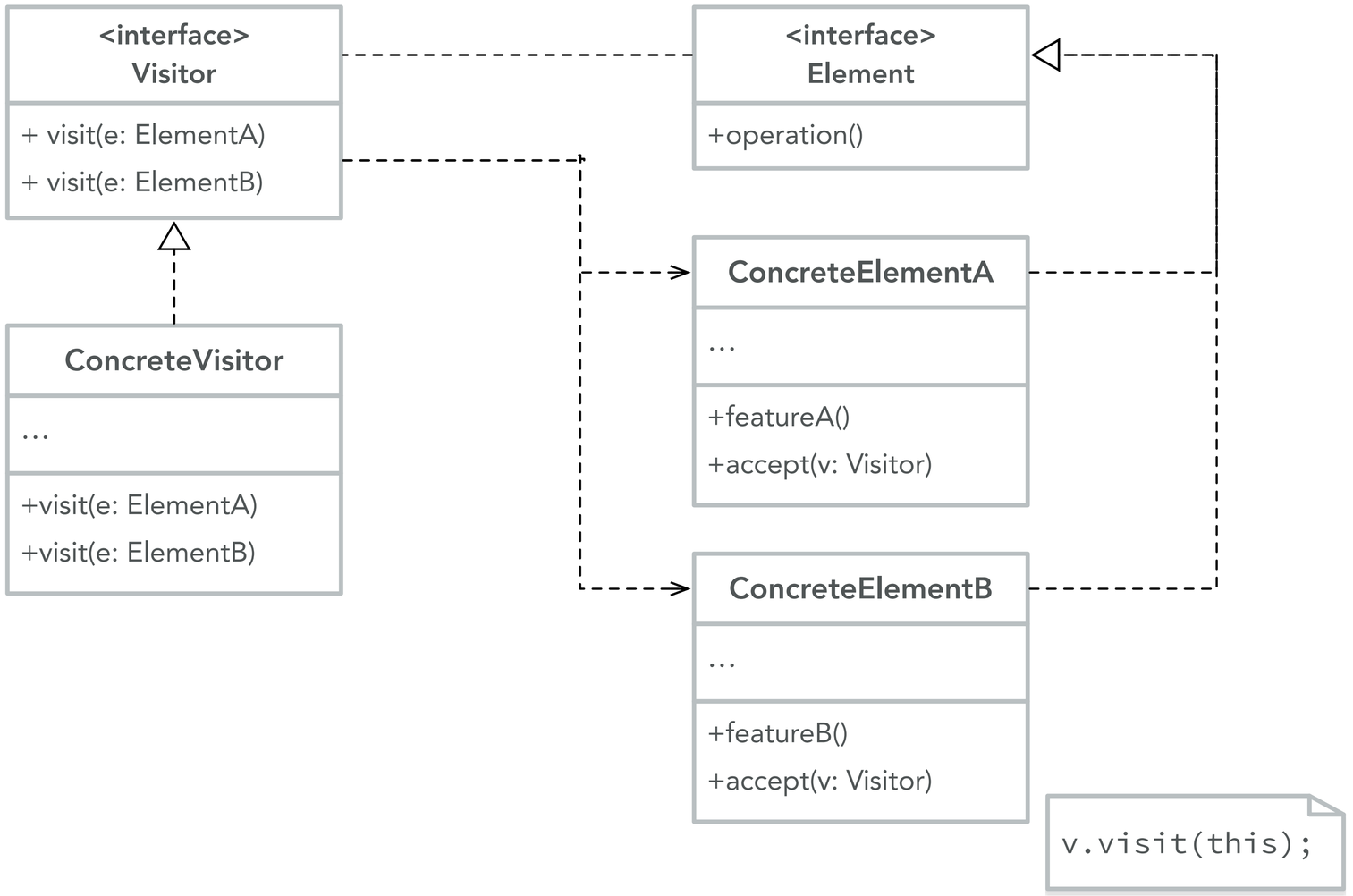



```
ArrayList<String> c = ...

Iterator<String> it = c.getIterator();
while(it.hasNext()){
    System.out.println(it.next());
}
```

Behavioural Patterns: Visitor

- Problem: you want to separate a data structure and an algorithm that operates on the data structure.
  - For example, you have parsed your source code into an AST, which is a tree. Now you want to analyse or manipulate the tree using an algorithm.
  - Do you implement what needs to be done per node type into the class that represents the tree nodes?

Behavioural Patterns: Visitor

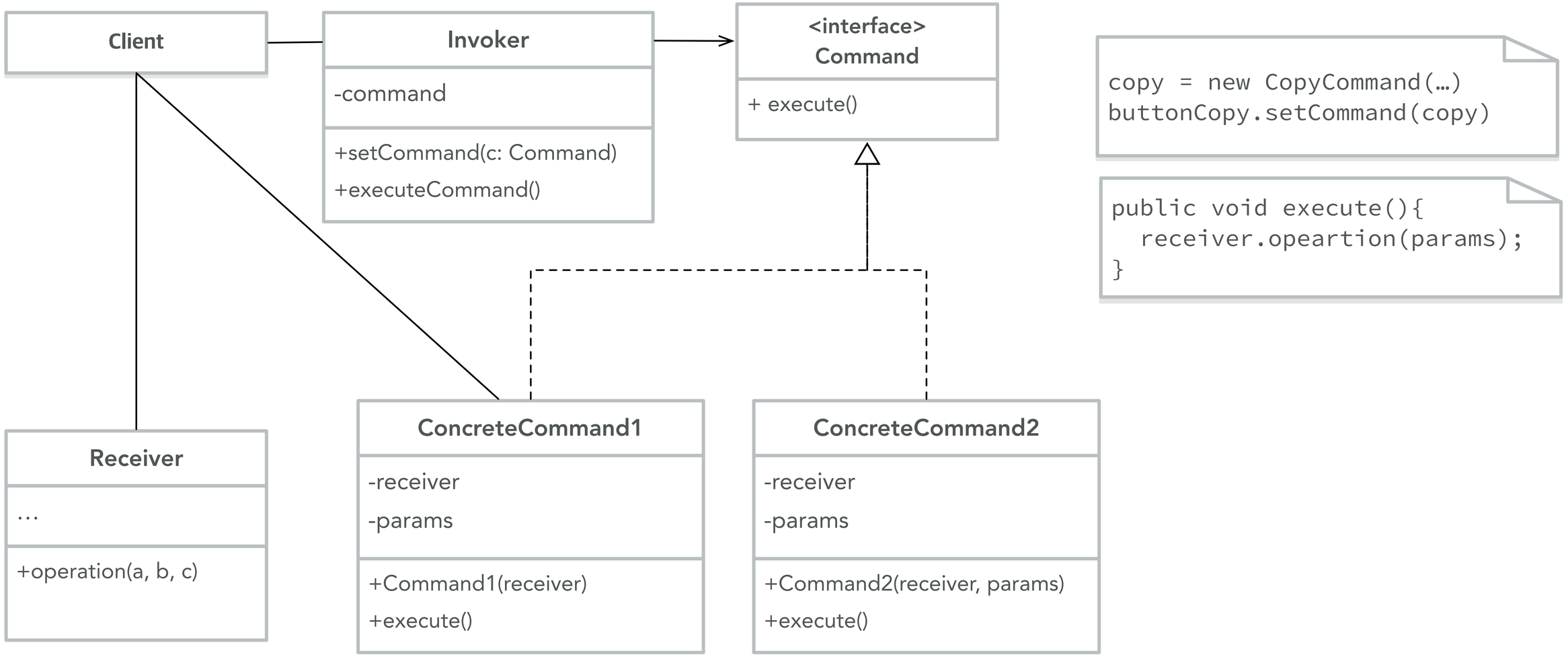



```
left = node.getLeftChild();
left.accept(this);

some_action(node);

right = node.getRightChild();
right.accept(this)
```

Behavioural Patterns: Command

- Problem: you are implementing GUI - each GUI element should invoke the corresponding business logic. However, if you add the corresponding logic to event handler, you duplicate a lot of code.
  - “Saving the file” can be invoked from a short-cut key (KeyEventHandler), a Save button in the menu-bar (OnClick event handler of the button class), and a menu item (OnClick event handler of the menu item).

Behavioural Patterns: Command



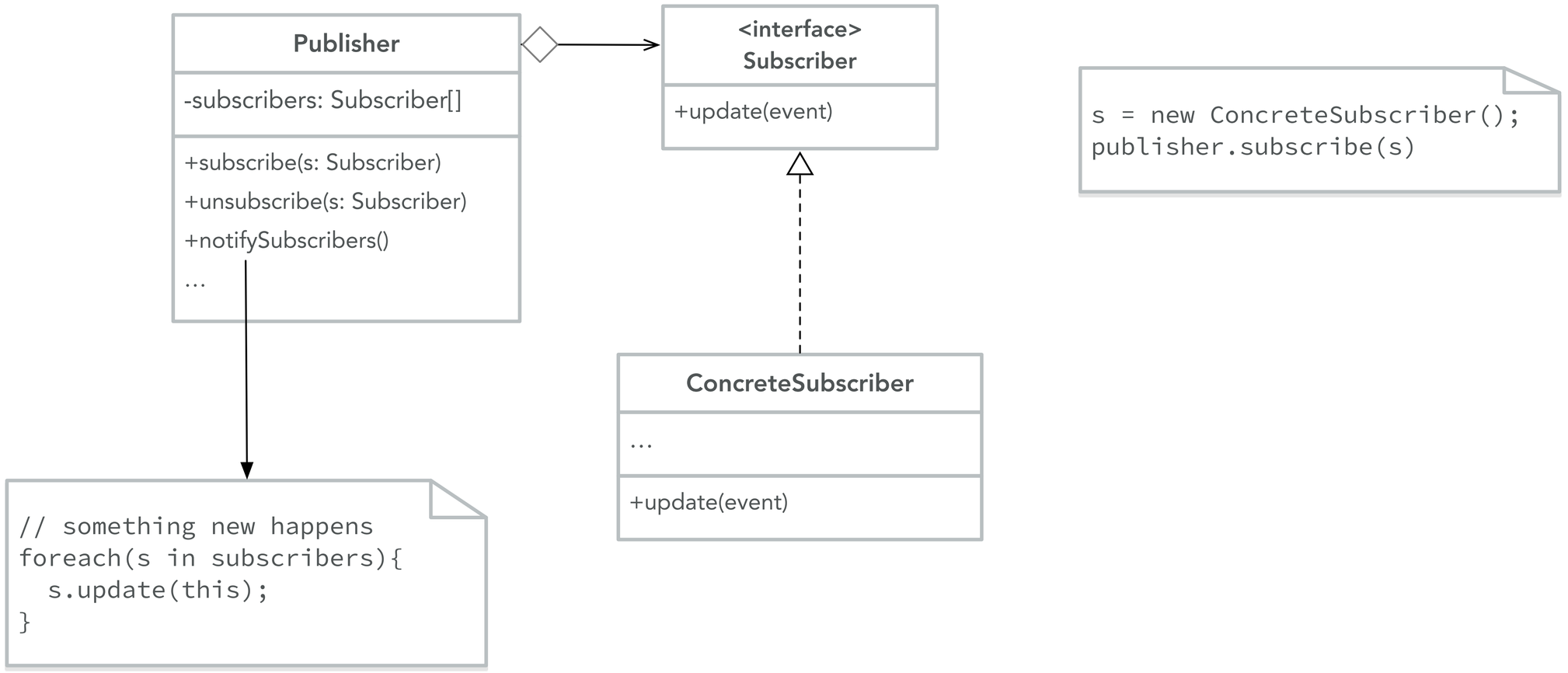

```
copy = new CopyCommand(…)
buttonCopy.setCommand(copy)
```

```
public void execute(){
  receiver.opeartion(params);
}
```

Behavioural Patterns: Observer

- **Problem**: how do you wait for an event, so that you can react?
  - A naive approach would be polling, i.e., regularly checking whether the event has happened.
  - This is 1) mostly wasteful, and 2) requires a timer mechanism.
  - Instead, the source of the event becomes a “Publisher”, and the interested parties become “Subscribers”. You register your interest at the beginning; if something happens, the publisher notifies the subscribers.
  - Can we describe this structure in UML?

Behavioural Patterns: Observer



Essence of Design Patterns

- These are Object Oriented Best Practices, in some sense.
  - There are clear “patterns” (pun intended) here: caution against excessive subclass hierarchy, delegation by creating a third-party object, reducing coupling, increasing cohesion…
  - For those relatively inexperienced in design level thinking, these guidelines do make you think about the structures in your software.

Critiques of Design Patterns

- The vision of Alexander is that we will have a common language (patterns) with which we will examine the design problems: while it gives the invariant properties, it does not suggest actual blueprints.
- Critiques of GoF/Design Patterns argue that, instead of “thinking” about these design problems, the book and the patterns simply reduced the problems into copying/pasting a lot of boilerplates.
- Another common criticism is that some of the patterns are only necessary when the programming language is not powerful enough.

Summary

- Design patterns are an attempt to extract repeatedly used common structures from OOP design: there are definitely recurring problems, and there are also good solutions for them.
- Good solutions/designs may share some patterns. However, using those patterns does not automatically result in good designs.
- “If you are holding a hammer, everything looks like a nail”
- Again: it is the ideas and thoughts that count, not the fixed rules.