COMP2511

WEEK 7

A G E N D A

- Concurrency Overview
- Singleton Pattern, Multithreading, and Data Races
- Abstract Factory Pattern
- Evolution of Requirements (Decorator Pattern)

Concurrency Overview

CONCURRENCY

- Concurrency is the ability of a program to execute different sections of code seemingly simultaneously.
- It is the basis behind all of modern computing, and it is how your computer is able to do so much all at once.
    - It's how your clock is able to tick while you watch a youtube video, and it's how your operating system allows multiple programs to actively coexist.
- There are many ways to achieve concurrency. For this tutorial, and for when we discuss software architectures, we will look a little closer at multithreading and asynchronous programming.

CONCURRENCY

- Single-threading: Up until now, most of the code you have written is single-threaded; it executes line after line, without interruption.
  - Think of this like having one chef in a kitchen, who cooks a steak, and waits for it to be finished before moving onto the potatoes, then the veggies. At the end, only the veggies are warm and we wasted time sitting around, as we waited for each prior dish to be finished before continuing.
- Multi-threading: This is where different threads run different parts of your code. Creating multiple threads in your programs allows you to execute lines simultaneously, instead of one after the other.
  - Think of this like having multiple chefs to prepare different dishes at the same time. One can prepare the steak, another the potatoes, and another the veggies. The benefit, is all the food can come out warm and in a timely manner.

CONCURRENCY

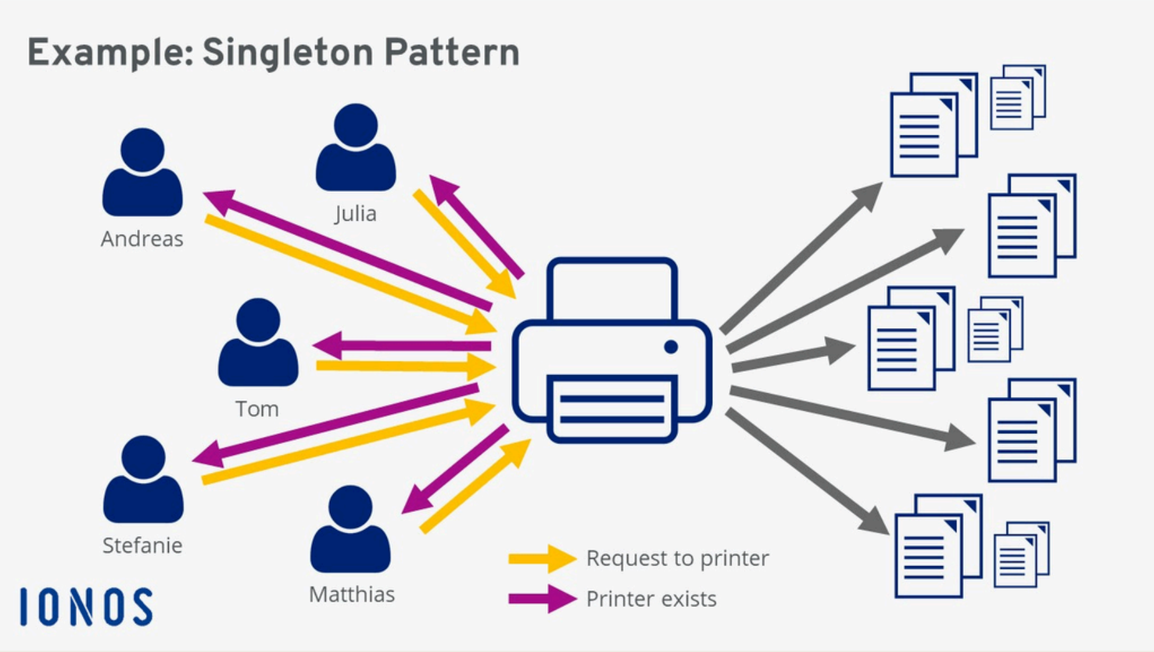
- Asynchronous: This is a little different, but still achieves the same idea. Instead of spawning multiple threads to run code at the same time, we have one thread that manages tasks. The thread manages it's tasks by having them yield control when they are busy doing something else. Once the task is ready to run again, we continue.
    - Think of this like having one chef, that starts cooking the steak, then, while it's on the grill, moves on to boiling the potatoes. While the potatoes boil, the chef can start preparing the veggies. The chef then moves around the kitchen to the different tasks as they require attention. The benefit is we make better use of our time, and the dish comes out warm and timely.
- It's important to know that, under the hood, the operating system delegates execution time to your program as it sees fit. It chooses which programs should run, and when, at the millisecond scale. As such, this "context-switching" makes it seem like programs are executing code at the same time, when in reality the OS is alternating who gets to run their code very, very quickly.

Singleton Pattern, Multithreading, and Data Races

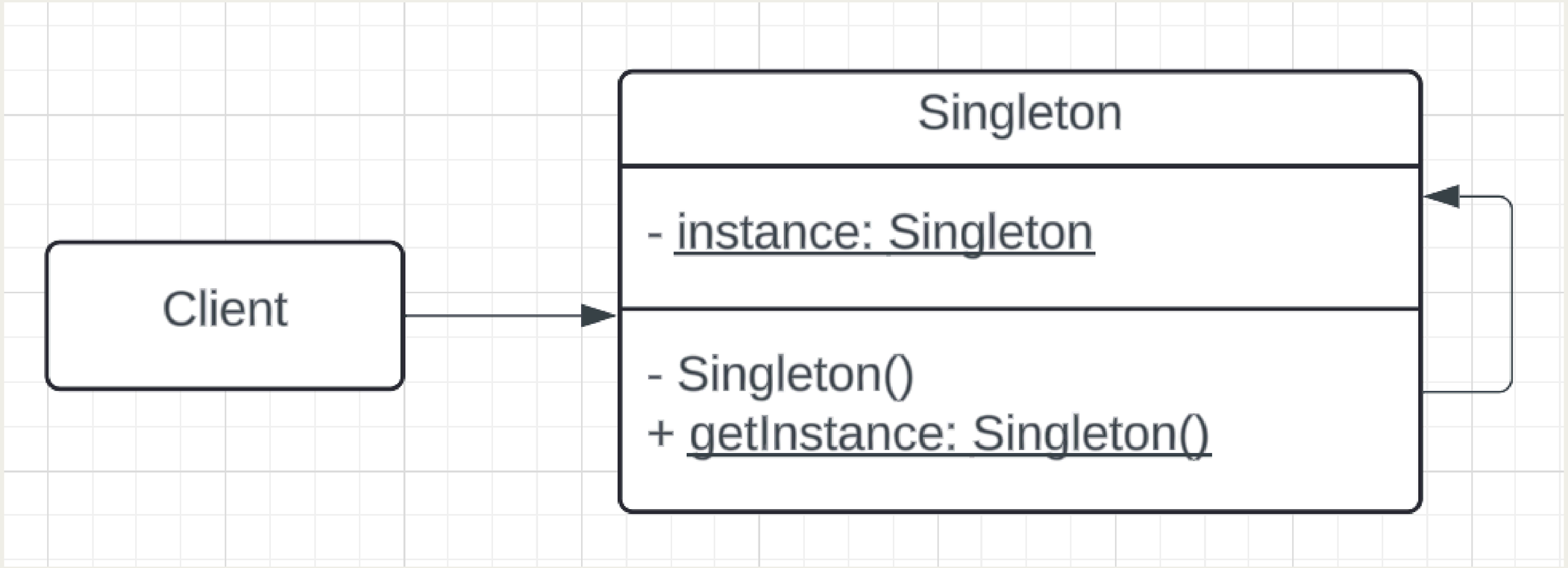
SINGLETON PATTERN

Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

It helps avoid initialisation overhead when only 1 copy of an instance is needed.



SINGLETON PATTERN



EXAMPLE

- Inside src/overcooked there is some code to model a very busy kitchen. In particular, it's September 17th (the most common birthday in Australia), and all of our chefs are pushing and shoving to get birthday cakes out of the oven and ready to be decorated. In order to safely access the oven, each Chef needs to use the Oven Mitts. But there seems to be some strange behaviour where because each Chef has their own mitts, they grab cakes willy-nilly....
- In this codebase we have the following classes:
  - Chef. This object extends the Thread class to be able to work alongside other Chefs.
  - OvenMitts. This is what Chefs use to safely access the Oven class.
  - Oven. This represents the oven that our kitchen has, it allows Chefs to take out cakes.
  - Kitchen. This is where the mayhem occurs.
- Currently when you run the code, you will find that each Chef accesses the oven at the same time (which doesn't make sense). They also seem to just take out as many cakes as they can until the oven has none left... or somehow... has a negative amount left:

EXAMPLE

- The above is an example of a data-race, which occurs when two threads are competing for a shared resource. We'll need to fix this problem in order to bring peace to our kitchen.
- Use the Singleton Pattern to ensure that only one Chef can access the oven at a time. If we control the amount of Mitts in the kitchen, we can control when Chefs access the oven. You should only need to change src/overcooked/Chef.java and src/overcooked/OvenMitts.java.

```
Preheating oven to 180°C...
Loaded the oven with 10 cakes...

---

Jamie puts on the oven mitts and opens the oven...
Julia puts on the oven mitts and opens the oven...
Gordon puts on the oven mitts and opens the oven...
Jamie took a cake out! (9 left)
Julia took a cake out! (8 left)
Gordon took a cake out! (7 left)
Julia took a cake out! (6 left)
Gordon took a cake out! (4 left)
Jamie took a cake out! (5 left)
Julia took a cake out! (3 left)
Jamie took a cake out! (1 left)
Gordon took a cake out! (2 left)
Gordon closes the oven and takes off the mitts.
Jamie took a cake out! (-1 left)
Julia took a cake out! (0 left)
Julia closes the oven and takes off the mitts.
Jamie found the oven empty!
Jamie closes the oven and takes off the mitts.

---

Cakes left in oven: -1
```

Abstract Factory Pattern

FACTORY PATTERN- ABSTRACT FACTORY PATTERN

- Commonly, people mistake an abstract factory as being a simple factory class that is just declared abstract. This is not correct.
- The abstract factory pattern is where we have some interface that declares creational methods that are related to each other.

```
1  public interface TransportFactory {
2      public Transport createTransport();
3      public Engine createEngine();
4      public Wheel createWheel();
5  }
```

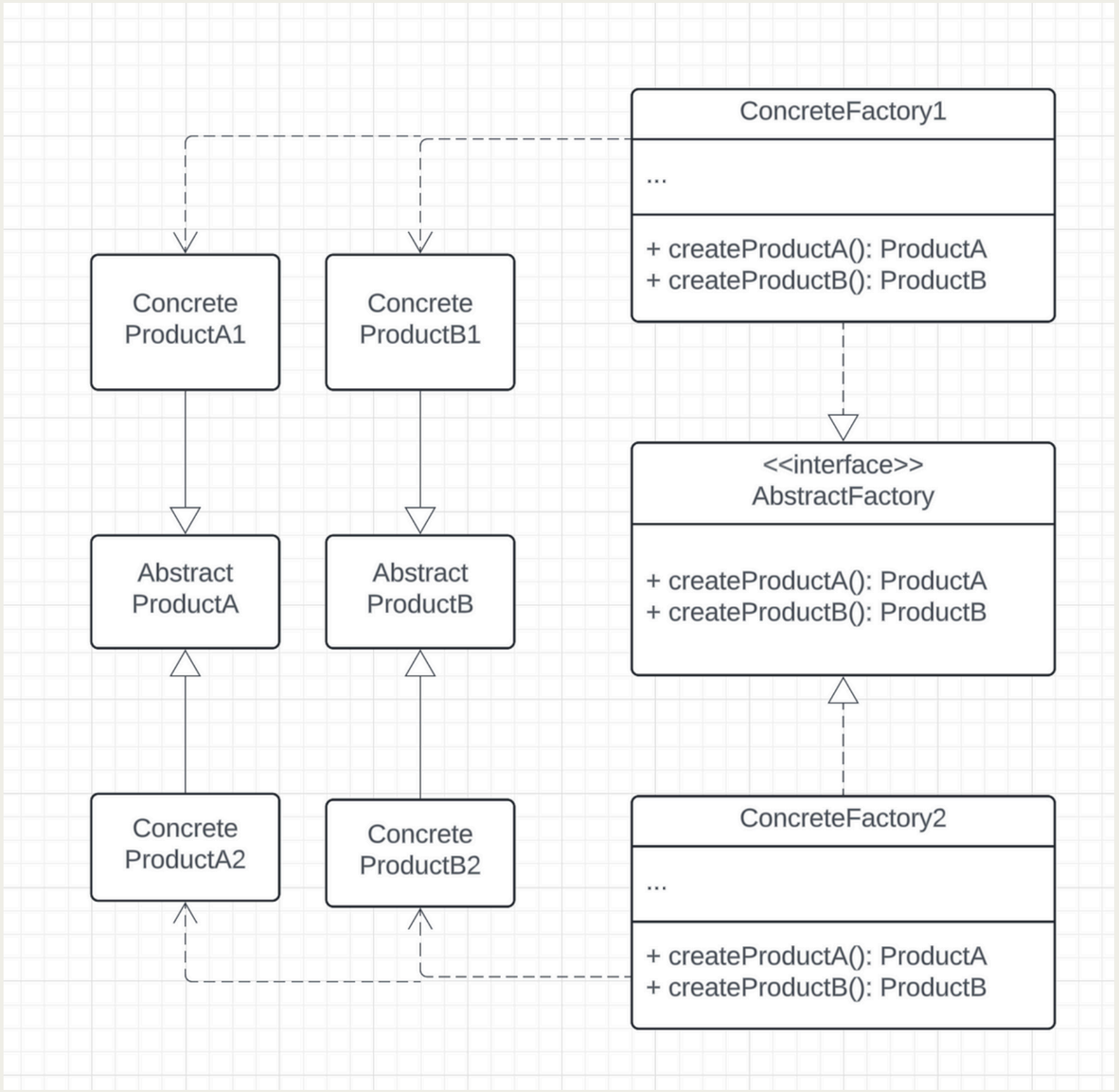
FACTORY PATTERN- ABSTRACT FACTORY PATTERN

- Classes that implement this interface (or inherit it if we were to use an abstract class) must implement these creational methods (methods that create objects).
- Reduces coupling between the client and the factory, promotes appropriate encapsulation of families of objects.

```
class PlaneFactory implements TransportFactory {
    public Transport createTransport() {
        return new Plane();
    }

    public Engine createEngine() {
        return new JetEngine();
    }

    public Controls createControls() {
        return new Yoke();
    }
}
```

```
class CarFactory implements TransportFactory {
    public Transport createTransport() {
        return new Car();
    }

    public Engine createEngine() {
        return new CombustionEngine();
    }

    public Controls createControls() {
        return new SteeringWheel();
    }
}
```

FACTORY PATTERN- ABSTRACT FACTORY PATTERN



FACTORY PATTERN- PREVIOUS EXAMPLE

Inside src/thrones, there is some code to model a simple chess-like game. In this game different types of characters move around on a grid fighting each other. When one character moves into the square occupied by another they attack that character and inflict damage based on random chance. There are four types of characters:

- A king can move one square in any direction (including diagonally), and always causes 8 points of damage when attacking.
- A knight can move like a knight in chess (in an L shape), and has a 1 in 2 chance of inflicting 10 points of damage when attacking.
- A queen can move to any square in the same column, row or diagonal as she is currently on, and has a 1 in 3 chance of inflicting 12 points of damage and a 2 out of 3 chance of inflicting 6 points of damage.
- A dragon can only move up, down, left or right, and has a 1 in 6 chance of inflicting 20 points of damage.

FACTORY PATTERN- NEW REQUIREMENTS

We now want to simulate the board game with the added change that pieces can be made from different materials, either Wood, Plastic or Metal. We want to be able to pick and choose which material we create a board with. Each material has its own construction requirements:

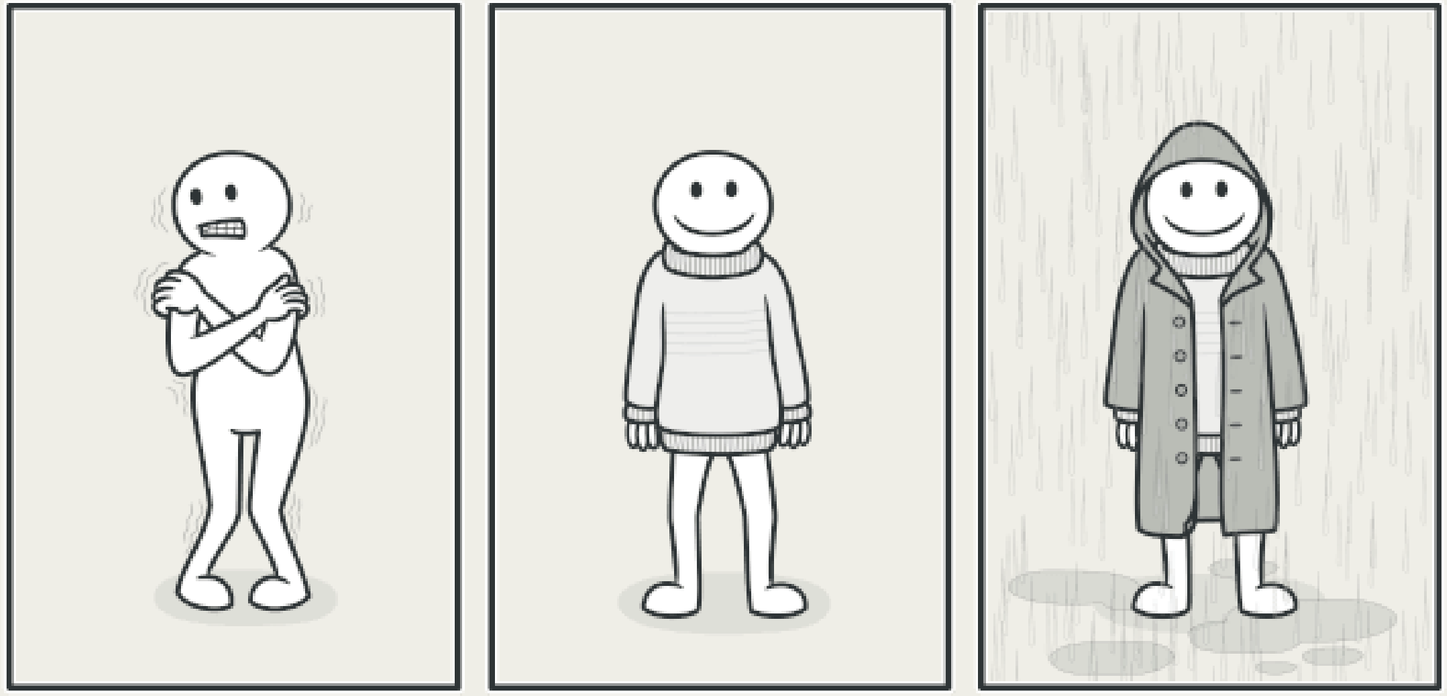
- Metal pieces are heavy and hence can only be set with 0 <= x <= 5, and y = 0
- Plastic pieces are lightweight and can be placed anywhere within a given bound (provided to the factory)
- Wooden pieces are large and bulky, and take up a nxn grid, meaning they can only be placed on tiles with x and y values which divide n (n is provided to the factory)

Utilise the Abstract Factory pattern to solve these requirements.

Decorator Pattern

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.

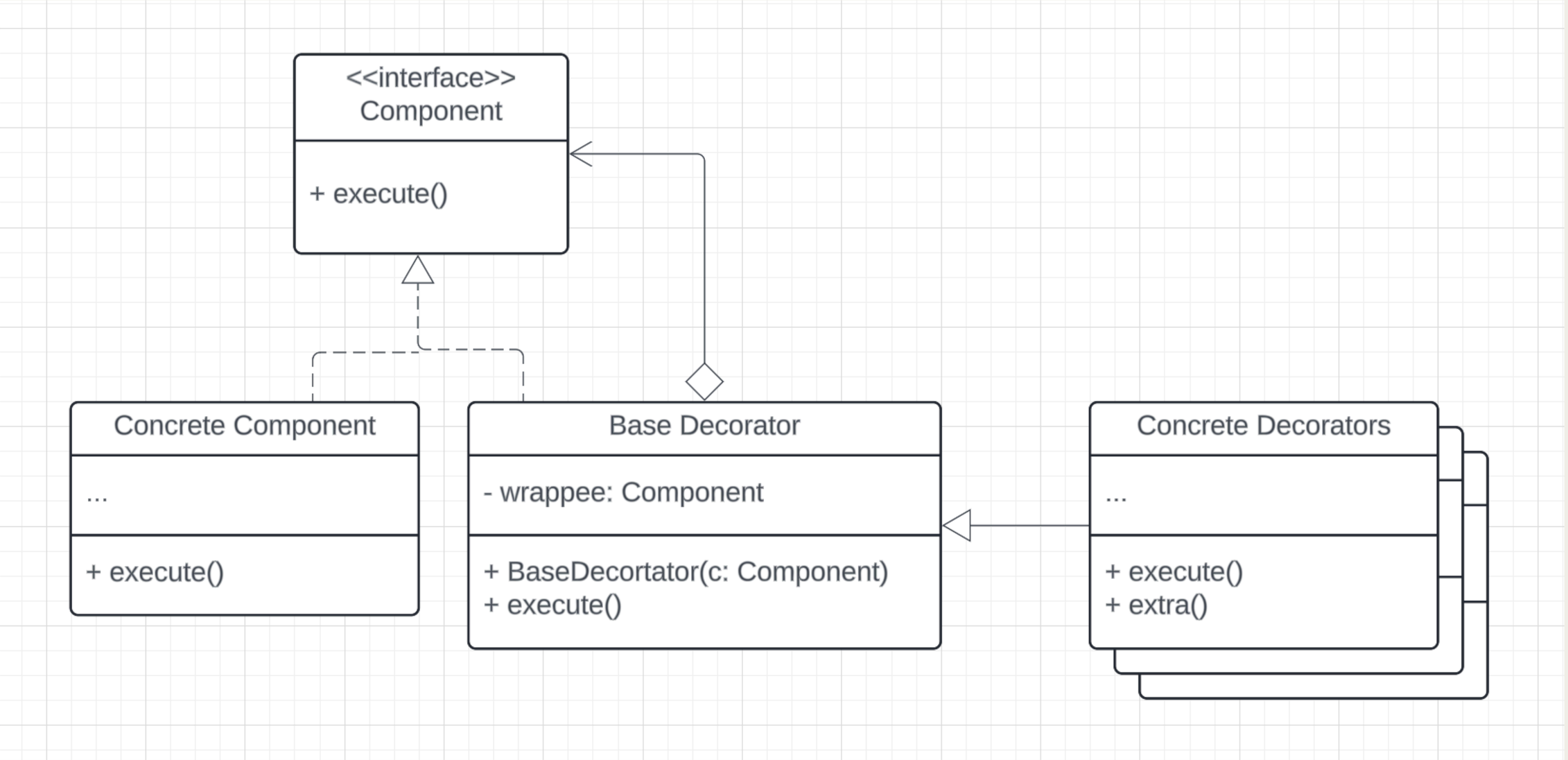


DECORATOR PATTERN

- Adds functionality to a class at run-time. Used when subclassing would result in an exponential rise in new classes
- Attaches additional responsibilities to an object dynamically
- Avoids implementing all possible functionality in one complex class
- Prefers composition over inheritance

Adding behaviour to an object, without opening the object up (i.e., rewriting its contents) and changing it

DECORATOR PATTERN



DECORATOR PATTERN

Suppose a requirements change was introduced that necessitated support for any character to wear different sorts of armour.

- A helmet reduces the amount of damage inflicted upon a character by 1 point.
- Chain mail reduces the amount of damage by half (rounded down).
- A chest plate caps the amount of damage to 7, but also slows the character down. If the character is otherwise capable of moving more than one square at a time then this armour restricts each move to distances of 3 squares or less (by manhattan distance).

Use the Decorator Pattern to realise these new requirements. Assume that, as this game takes place in a virtual world, there are no restrictions on the number of pieces of armour a character can wear and that the "order" in which armour is worn affects how it works. You may need to make a small change to the existing code.

Time for da lab

YEEEE