COMP2511

WEEK 7

iPad Update???

IMPORTANT NOTICE

- Assignment-i marks manual marks out.
  - Auto-marks early next week or over the weekend
- WRITE YOUR BLOGS. (Including MRs).
- Flag any group issues to me ASAP
- Any contribution issue should be raised at the end of each week (either in person, or via teams/email is fine)

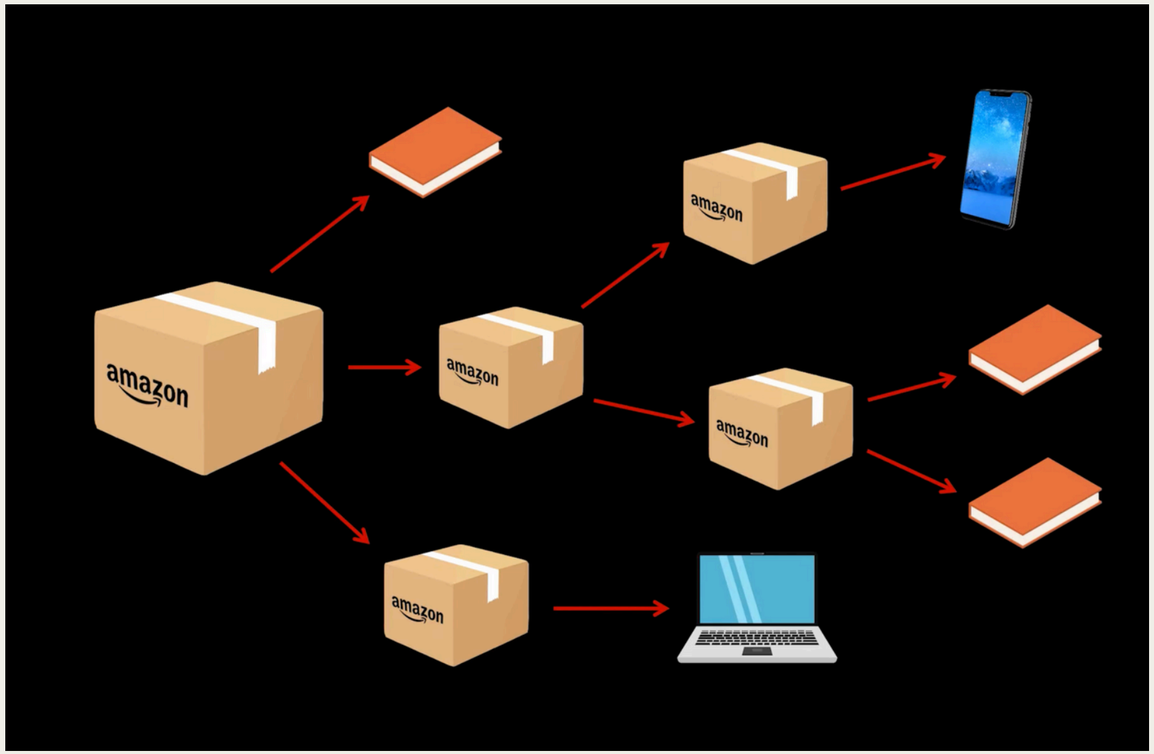
A G E N D A

- Composite Pattern
- Factory Pattern
- Decorator Pattern

Composite Pattern

COMPOSITE PATTERN

Composite is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.

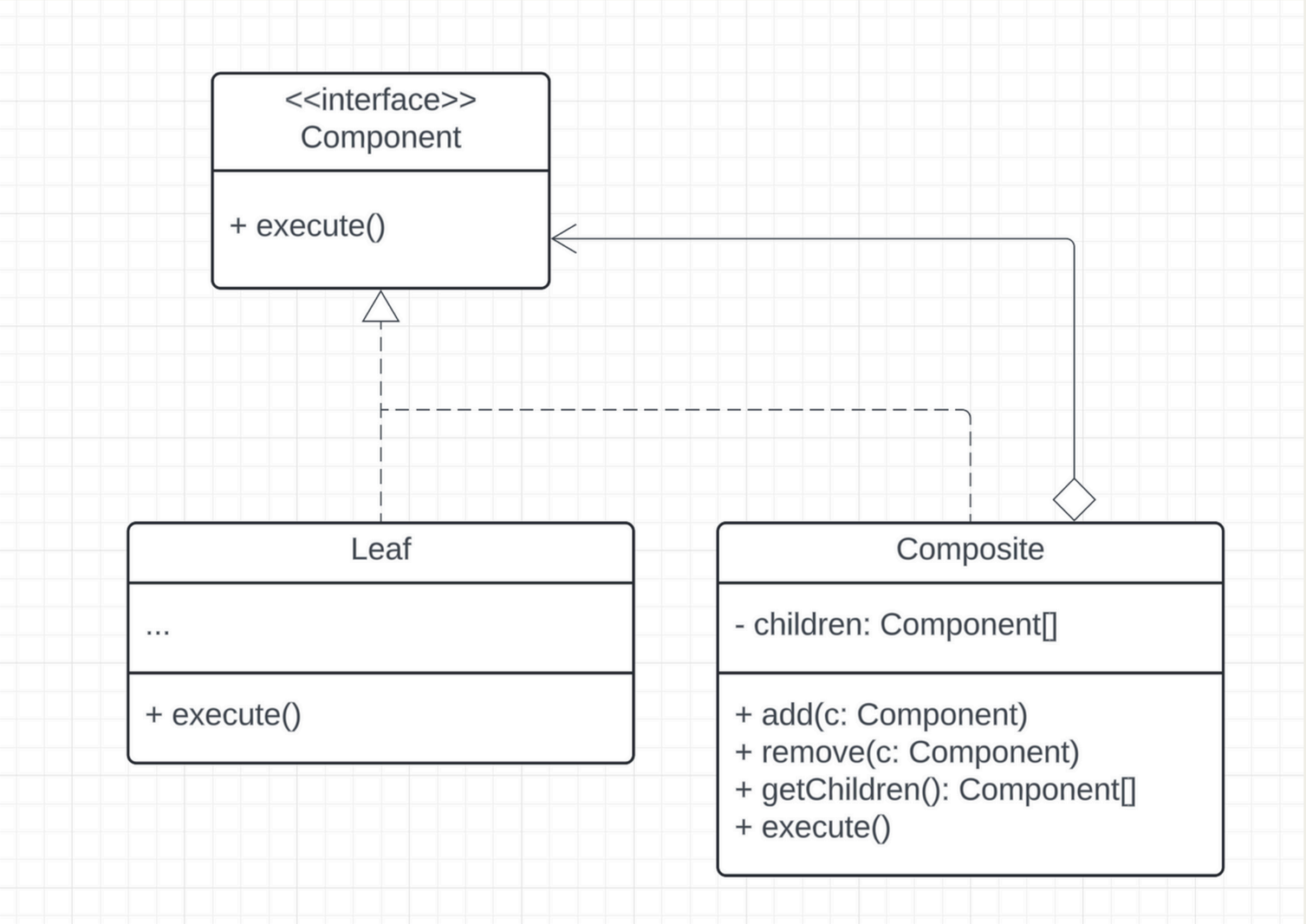


COMPOSITE PATTERN

Structural design pattern are patterns that ease the design by identifying a simple way to realize relationships among entities.

They explain how to assemble objects and classes into large structures, while keeping structures flexible and efficient.

COMPOSITE PATTERN



COMPOSITE PATTERN

Inside **src/calculator**, use the Composite Pattern to write a simple calculator that evaluates an expression. Your calculator should be able to:

- Add two expressions
- Subtract two expressions
- Multiply two expressions
- Divide two expressions

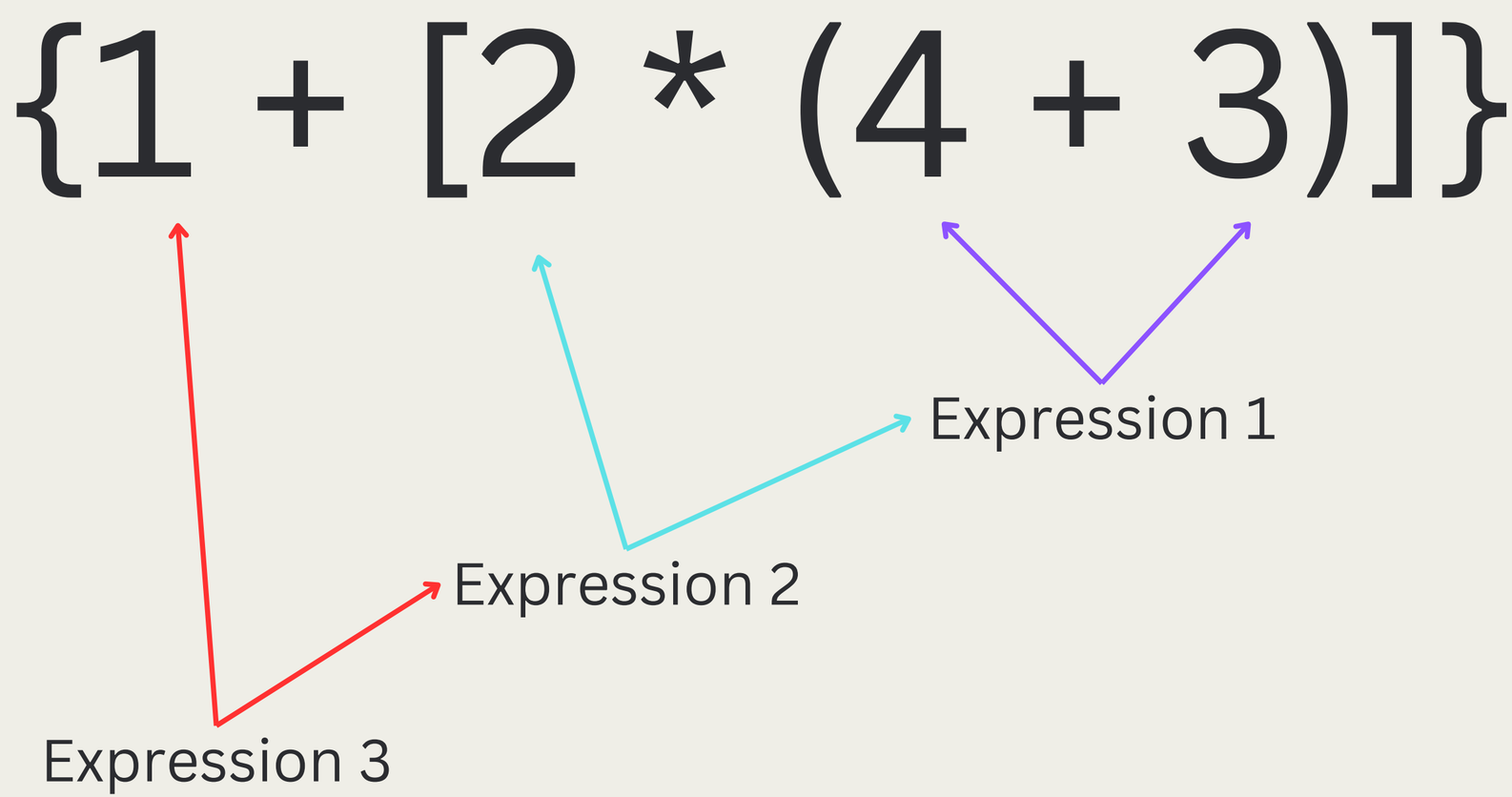
There should be a **Calculator** class as well which can have an expression passed in, and calculate that expression.

Design a solution, create stubs, write failing unit tests, then implement the functions.

COMPOSITE PATTERN

{1 + [2 * (4 + 3)]}

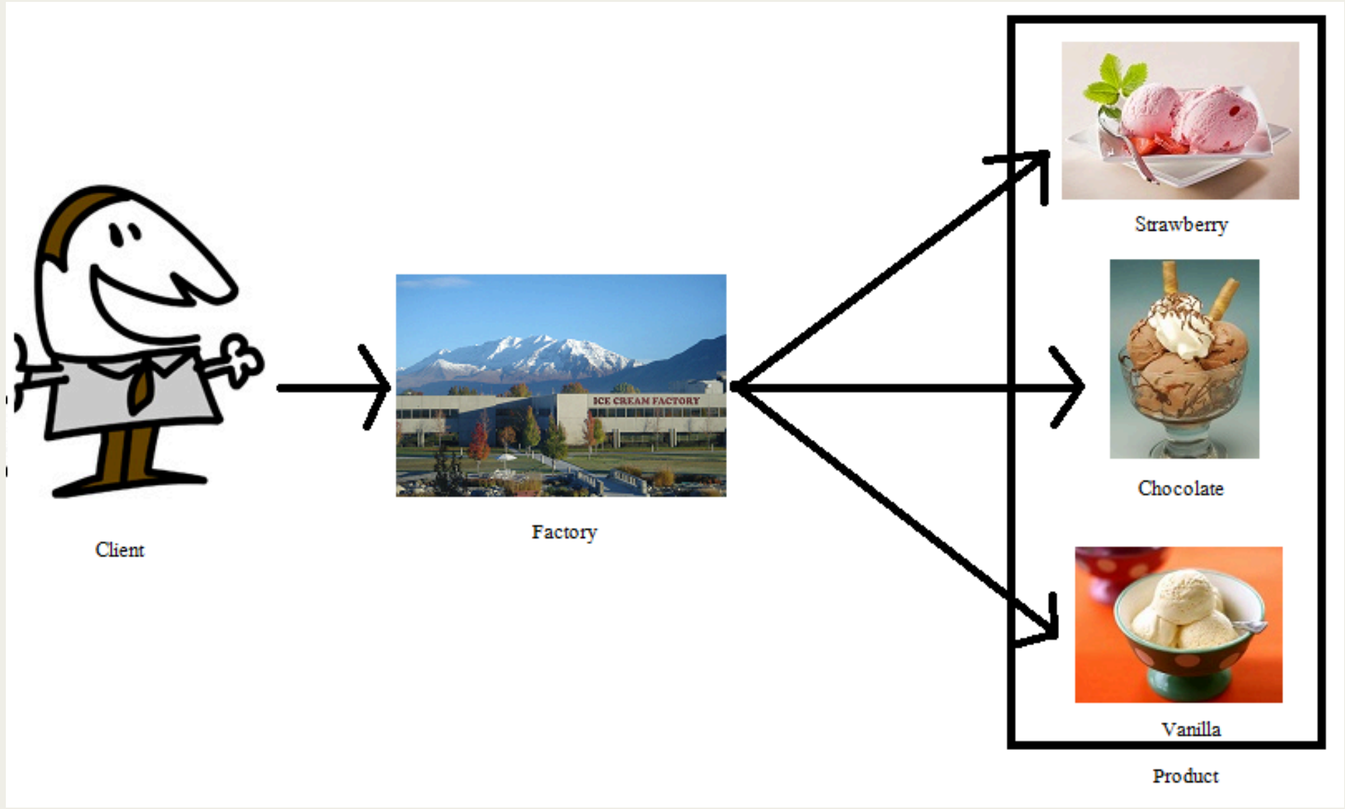
COMPOSITE PATTERN



Factory Pattern

FACTORY PATTERN

Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.



FACTORY PATTERN

Creational patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.

Factory method provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. Reduces coupling and shotgun surgery, as all classes are created using the same method.

We let our factory become responsible for creational logic abiding to single responsibility principle and promoting abstraction.

FACTORY PATTERN- CREATIONAL METHODS

A creational method is one that creates objects, and often is a wrapper around a constructor.

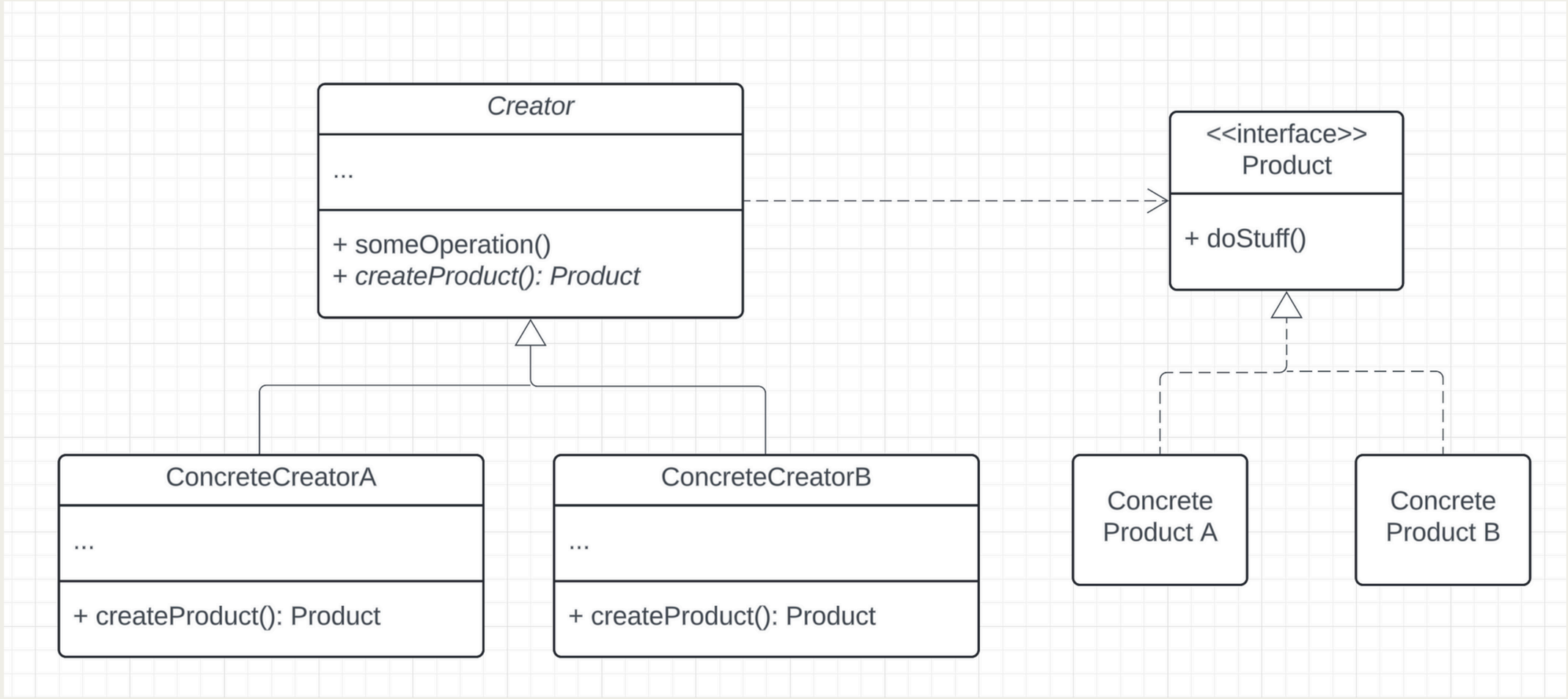
```
1  public Programmer createProgrammer(String team) {
2    return new Programmer(DEFAULT_PROGRAMMER_SALARY, team, makeNewUniqueID());
3  }
```

FACTORY PATTERN- SIMPLE FACTORY PATTERN

A simple factory pattern describes a class that has one big creation method with conditionals to determine what to construct.

```
class AccountantFactory {
    public static Accountant createAccountant(int experience, boolean hasCertification) {
        if (experience > 5 && hasCertification) {
            return new SeniorAccountant();
        } else if (experience > 5) {
            return new JuniorAccountant();
        } else if (experience >= 1) {
            return new EntryLevelAccountant();
        } else {
            return new InternAccountant();
        }
    }
}
```

FACTORY PATTERN



FACTORY PATTERN- FACTORY METHOD

The factory method is when we have some general logic that we want to group, but let subclasses decide which object will be created.

```java
public abstract class Department {
    private List<Employee> employees;

    public abstract Employee createEmployee(String name);

    public void fireEmployee(String id) {
        Employee toBeFired = getEmployee(id);
        toBeFired.paySalary();
        employees.remove(toBeFired);
    }
}

class ITDepartment extends Department {
    @Override
    public Employee createEmployee(String name); {
        return new Programmer(name);
    }
}

class AccountingDepartment extends Department {
    @Override
    public Employee createEmployee(String name); {
        return new Accountant(name);
    }
}
```

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  }
```

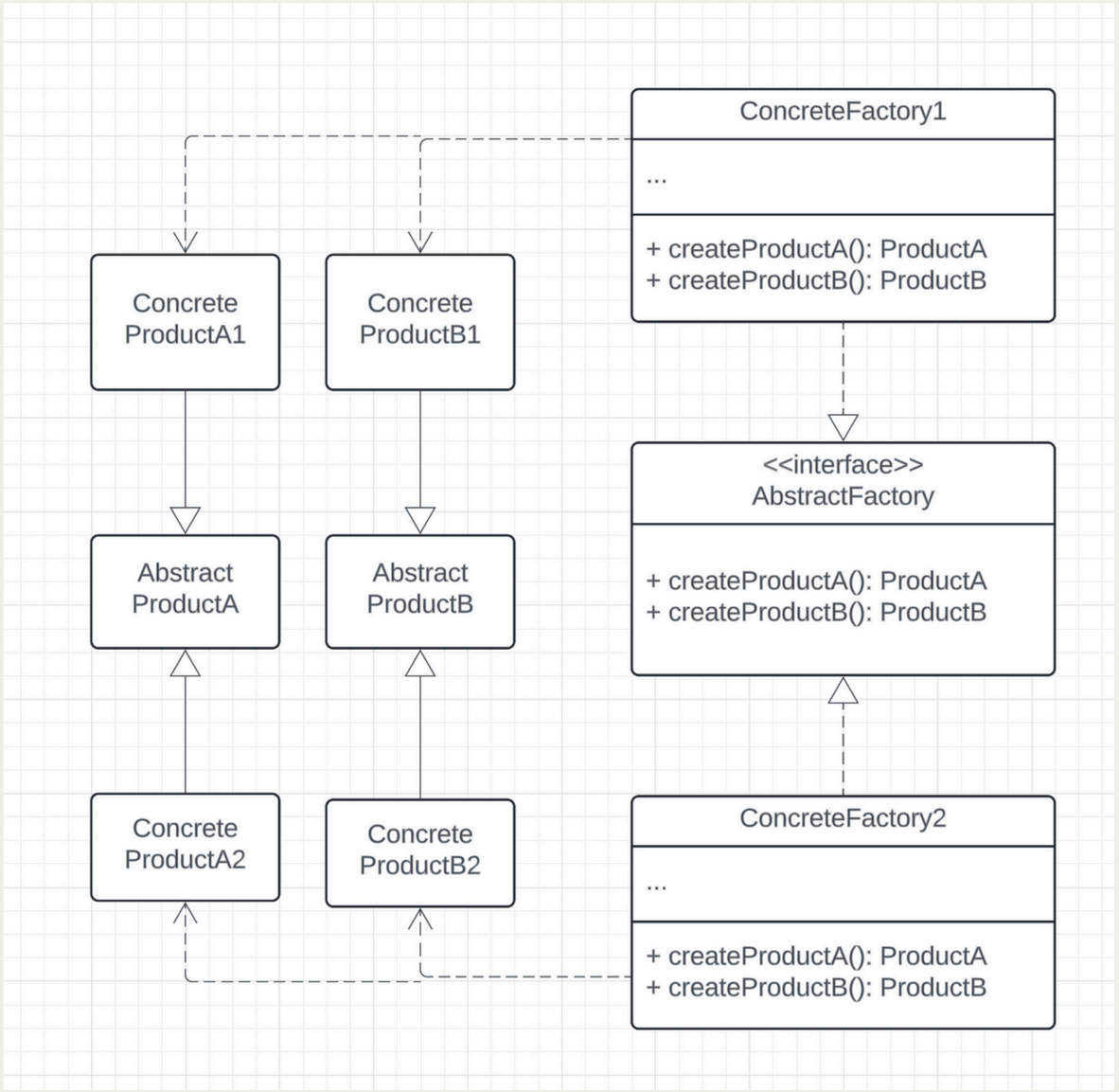
FACTORY PATTERN- ABSTRACT FACTORY PATTERN

- 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- ABSTRACT FACTORY PATTERN

- This is definitely a lot of different types of factories!
- It can be absolutely be confusing at first, but they all involve some abstraction of how an object or a family of objects is created.

- There are some pretty good resources online about the differences between factories

https://refactoring.guru/design-patterns/factory-comparison

https://stackoverflow.com/questions/13029261/design-patterns-factory-vs-factory-method-vs-abstract-factory

FACTORY PATTERN

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

We want to refactor the code so that when the characters are created, they are put in a random location in a grid of length 5.

1. How does the Factory Pattern (AKA Factory Method) allow us to abstract construction of objects, and how will it improve our design with this new requirement?
2. Use the Factory Pattern to create a series of object factories for each of the character types, and change the main method of Game.java to use these factories.

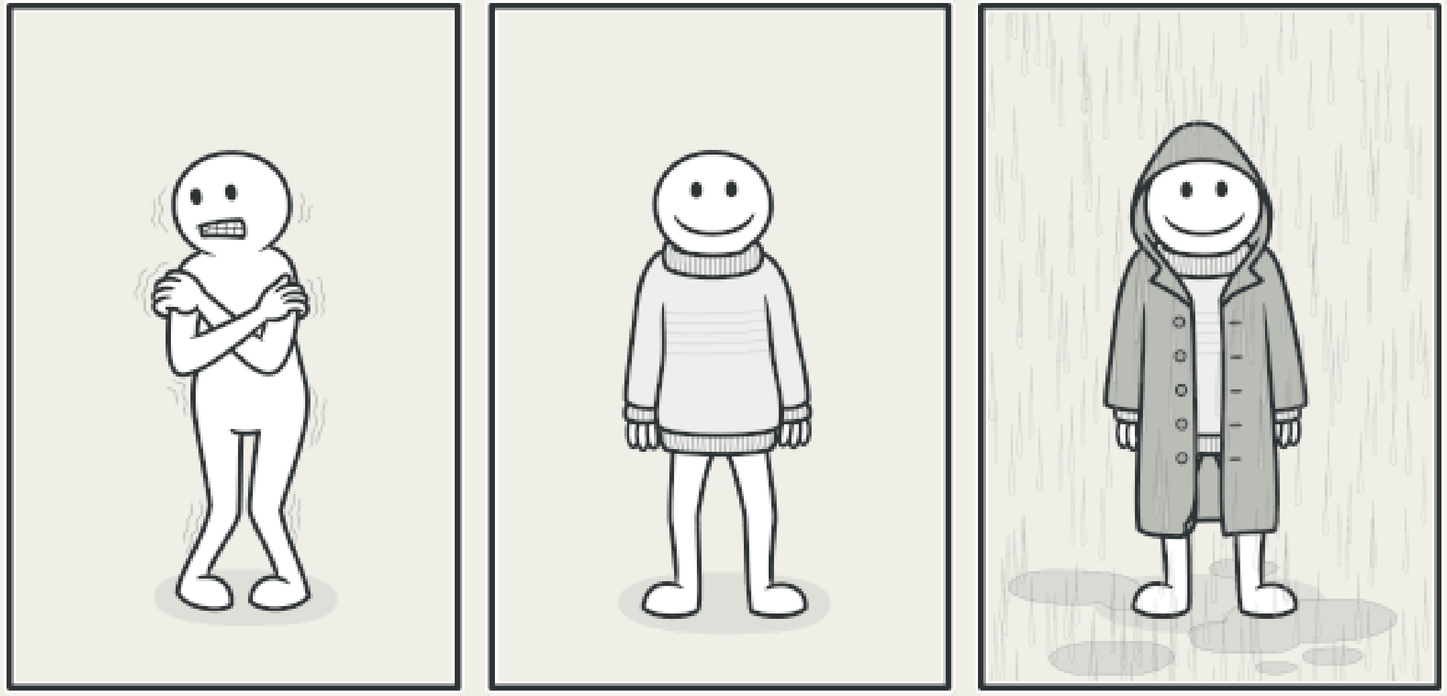
FACTORY PATTERN

1. Abstract the construction of the character objects. We don't deal with the constructor, instead call a general factory method that handles the number.

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.

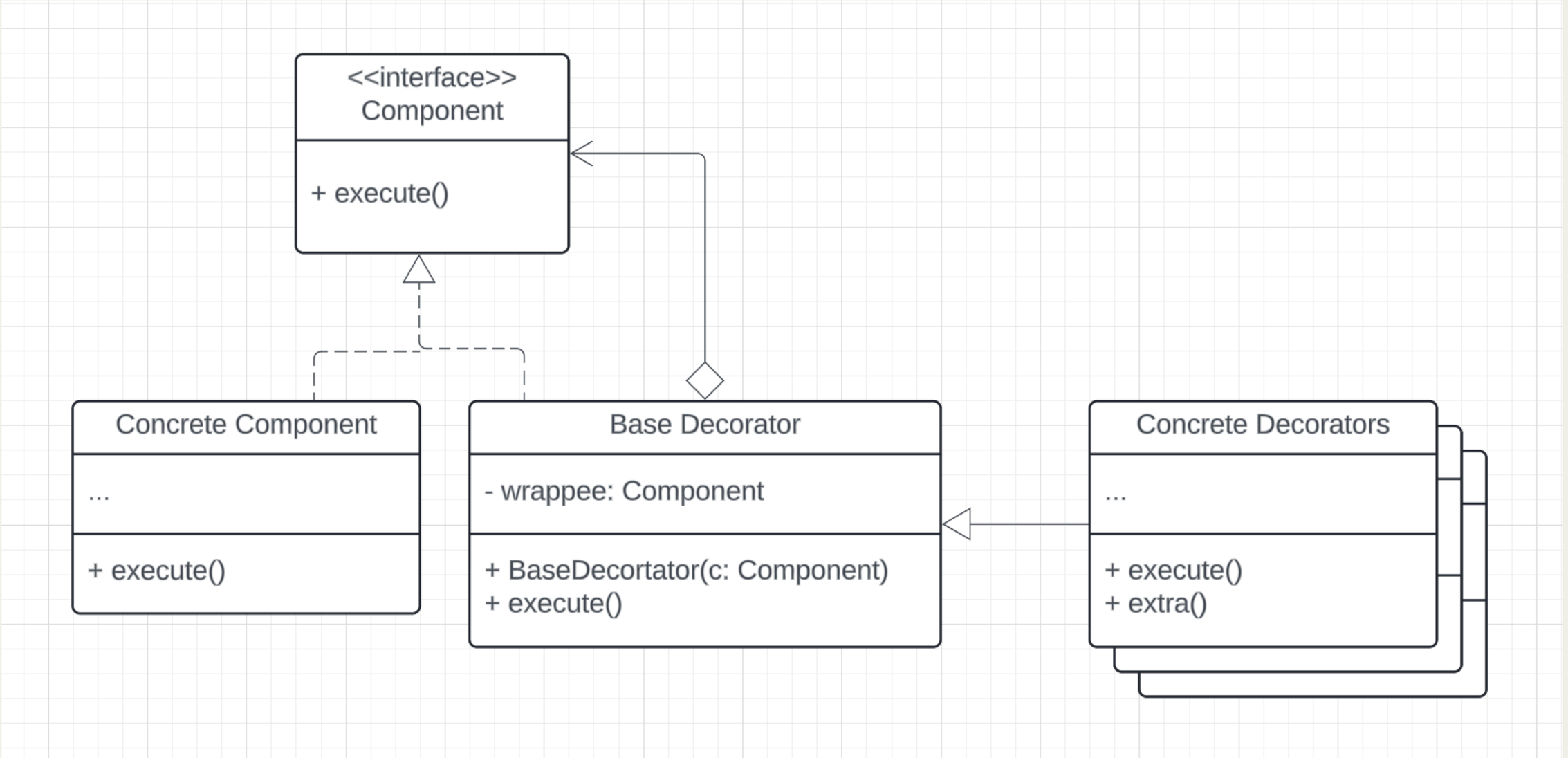


DECORATOR PATTERN

- 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 introduce that necessitated support for 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