COMP2511

WEEK 4

Would you rather give up bread or rice?

IMPORTANT NOTICE

- Assignment-i is due NEXT WEEK!
- Week 5 Wednesday, 3pm (15th October 2025)
- Maximum 5 day late penalty
- Worth 15% of total grade

ASSIGNMENT TIPS

- Use .equal() for String (or Class) instead of ==
- Avoid magic numbers, use final variables
- Use the super constructor to set variables
- Use instanceof for type comparison
- Polymorphism is preferred over type checking to perform a specific action

ASSIGNMENT TIPS

```
if (s.getClass().equals(Imposter.class)) { // v1: kinda ok
    // do something only on exactly the Imposter class
}
if (s.getType().equals("Imposter")) { // v2: very bad
    // do something on all imposters
}
if (s instanceof Imposter) { // v3: good
    // do something on all imposters
}
```

ASSIGNMENT TIPS

```java
// Example: What not to do
public abstract class Shape {
    public abstract String getType();
}
public class Rectangle extends Shape {}

public class Square extends Rectangle {
    public static void main(String[] args) {
        List<Shape> shapes = new ArrayList<>();
        shapes.add(new Rectangle());
        shapes.add(new Square());
        for (Shape s : shapes) {
            if (s.getType().equals("Rectangle")) {
                // calculate the area this way
            } else if (s.getType().equals("Square")) {
                // calculate the area a different way
            }
        }
    }
}
```

```java
// Example: What to do
public abstract class Shape {
    public abstract String getType();
    public abstract double area();
    // ^ declare method in superclass
}

public class Rectangle extends Shape {
    public double area() {
        // calculate the area this way
    }
}

public class Square extends Rectangle {
    public double area() {
        // calculate the area a different way
    }
    public static void main(String[] args) {
        List<Shape> shapes = new ArrayList<>();
        shapes.add(new Rectangle());
        shapes.add(new Square());
        for (Shape s : shapes) {
            s.area(); // no more type checking
        }
    }
}
```

A G E N D A

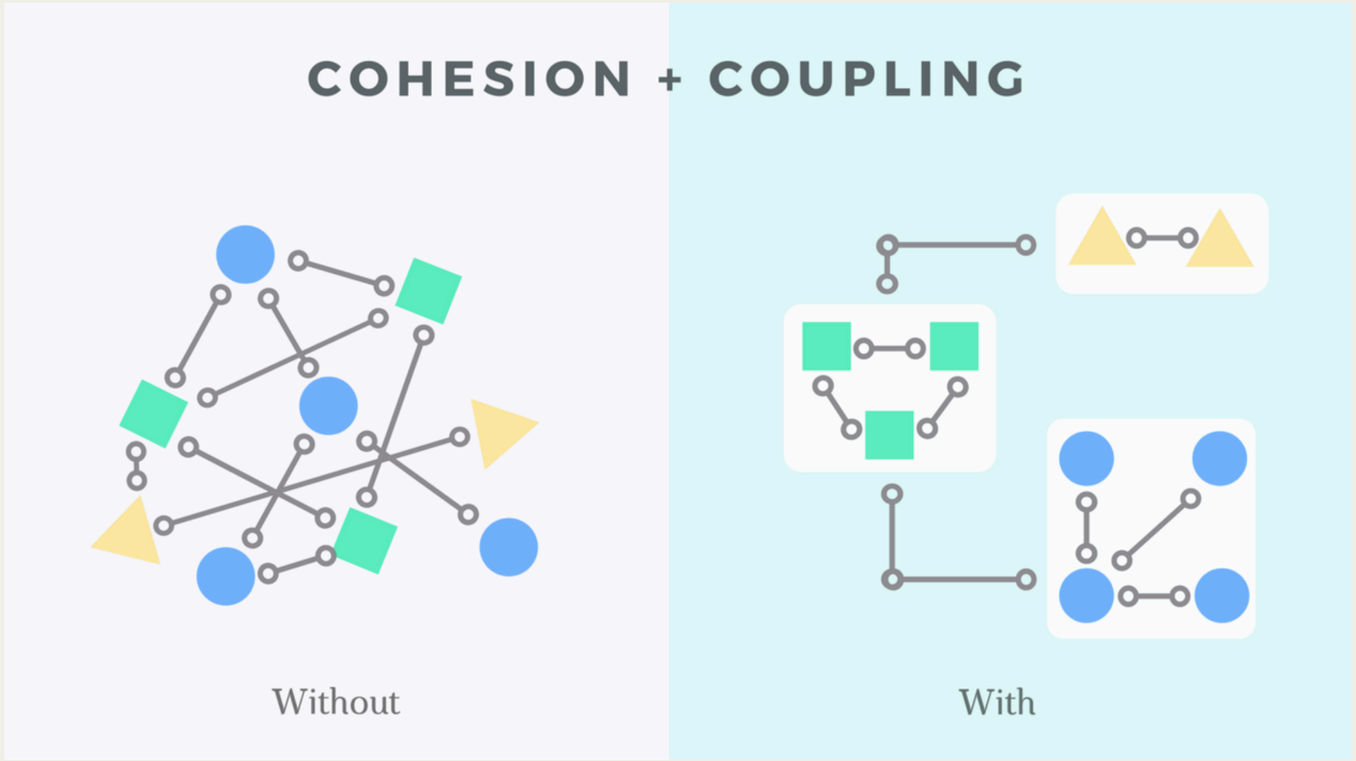
- Design Principles
- Composite Pattern
- Factory Pattern

Law of Demeter

“Principle of least knowledge”

LAW OF DEMETER

Law of Demeter (aka principle of least knowledge) is a **design guideline** that says that an **object** should **assume as little as possible knowledge** about the structures or properties of other objects.



LAW OF DEMETER

A method in an object should only invoke methods of:

- The object itself
- The object passed in as a parameter to the method
- Objects instantiated within the method
- Any component objects
- And not those of objects returned by a method

E.g., don't do this

```
object.get(name).get(thing).remove(node)
```

*Caveat is that sometimes this is unavoidable

LAW OF DEMETER

In the **unsw.training** package there is some skeleton code for a training system.

- Every employee must attend a whole day training seminar run by a qualified trainer
- Each trainer is running multiple seminars with no more than 10 attendees per seminar

In the **TrainingSystem** class there is a method to book a seminar for an employee given the dates on which they are available. This method violates the principle of least knowledge (Law of Demeter).

1. How and why does it violate this principle?
2. In violating this principle, what other properties of this design are not desirable?
3. Refactor the code so that the principle is no longer violated. How has this affected other properties of the design?
4. More generally, are getters essentially a means of violating the principle of least knowledge? Does this make using getters bad design?

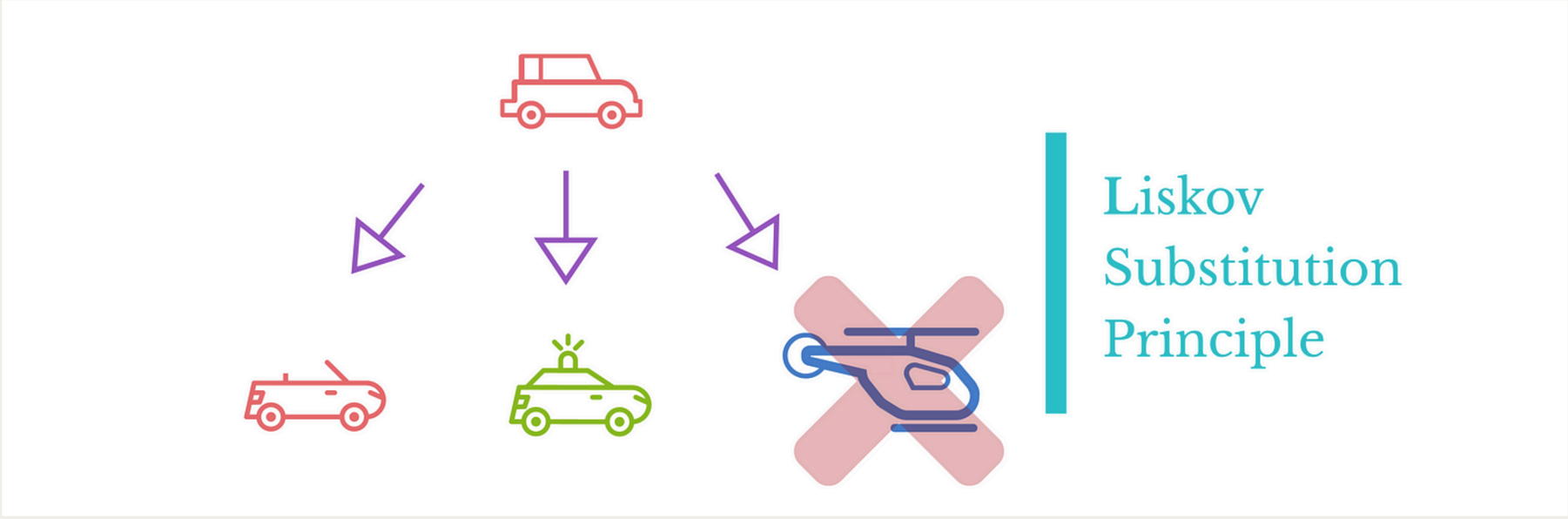
LAW OF DEMETER

- Getters which pass objects (not primitives) let clients use its methods
  - Violating principle of least knowledge
  - However, makes classes more reusable
  - What if it doesn’t have that functionality? Then you are using it to extend functionality. So good?
- Another example: having **getAttendees** in the seminar class.
  - Any client is able to modify this to have more than 10 attendees (everything is an object and passed by reference)
  - Unfortunately Java offers no good solutions to this problem
    - **getAttendees** has to create a copy of the list or
      - Needlessly copies data
    - Use **Collections.unmodifiableList(...)**
      - Still has the **add(...)** method but using it causes an exception
  - Other languages resolve this problem by having proper immutable or read-only lists.

LISKOV SUBSTITUTION PRINCIPLE

Liskov Substitution Principle (LSP) states that objects of a **superclass** should be **replaceable** with objects of its **subclasses without breaking the application**.

*inheritance arrows are the other way around



LISKOV SUBSTITUTION PRINCIPLE

Solve the problem without inheritance

- Delegation - delegate the functionality to another class
- Composition - reuse behaviour using one or more classes with composition

Design principle: Favour composition over inheritance.

If you favour composition over inheritance, your software will be more flexible, easier to maintain, extend.

LISKOV SUBSTITUTION PRINCIPLE

Look at the **OnlineSeminar** class. How does this violate the Liskov Substitution Principle?

LISKOV SUBSTITUTION PRINCIPLE

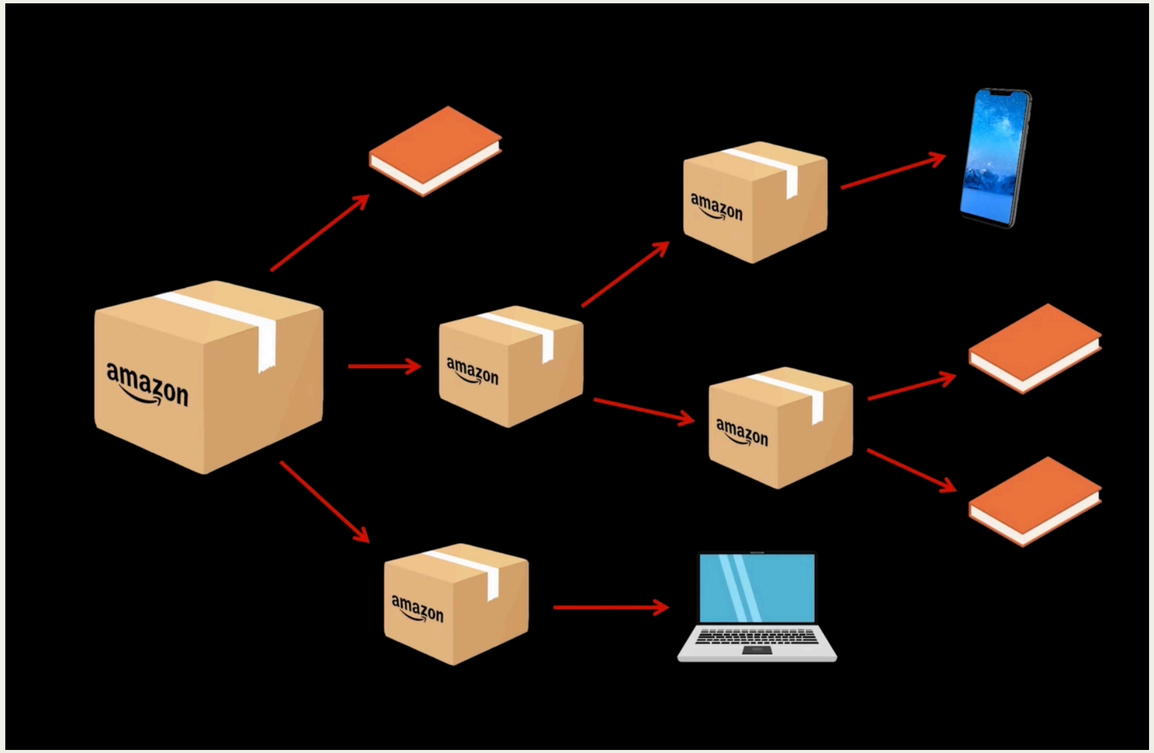
Look at the **OnlineSeminar** class. How does this violate the Liskov Substitution Principle?

- **OnlineSeminar** does not require a list of attendees
- Would expect it to be “booked” like a **Seminar**
- Has “Is-A” relationship but doesn’t make sense here
  - But invalid inheritance once you take into account what the classes do and represent.

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.

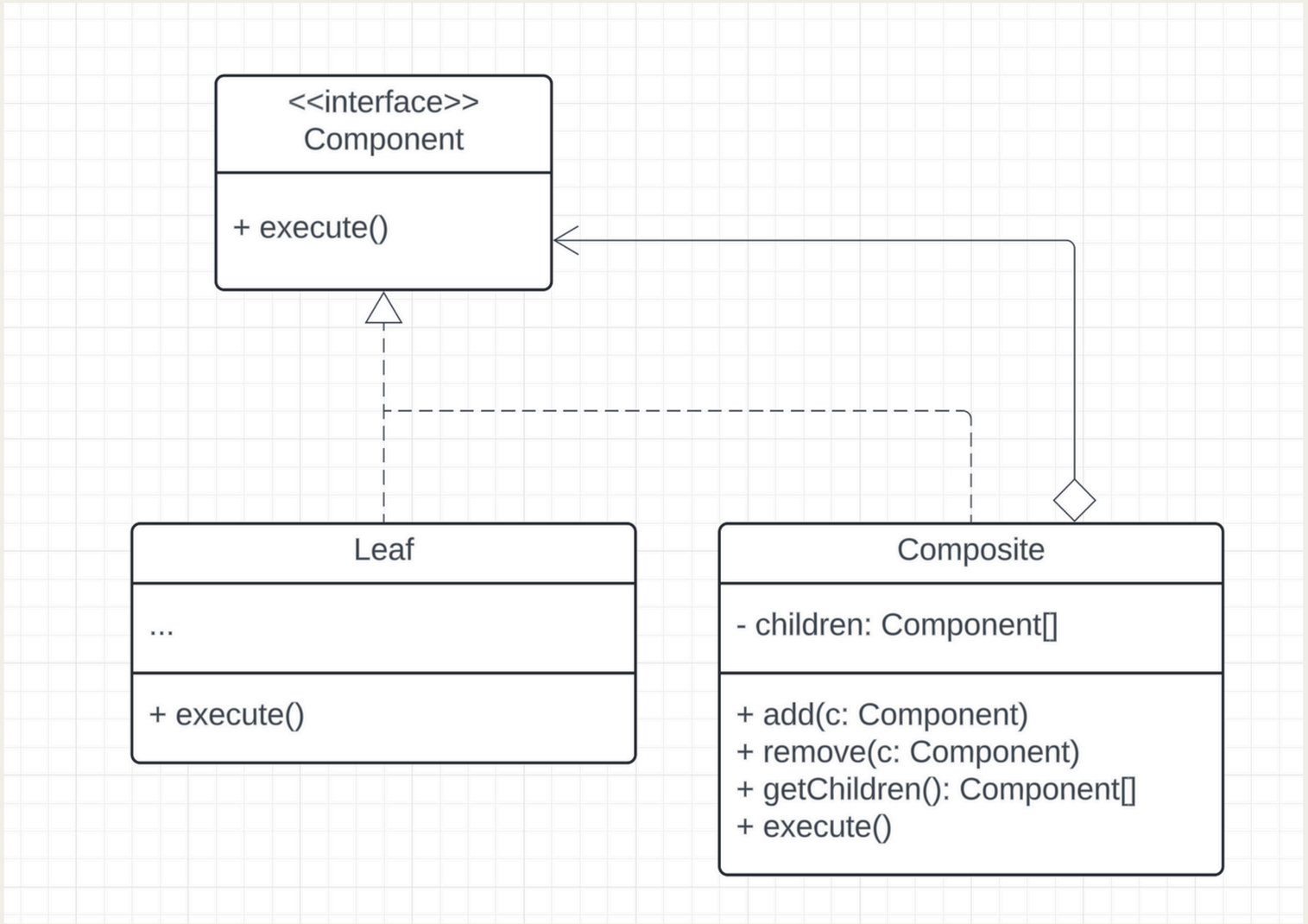


COMPOSITE PATTERN

Structural design pattern are patterns that ease the design by identifying a simple way to realise 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

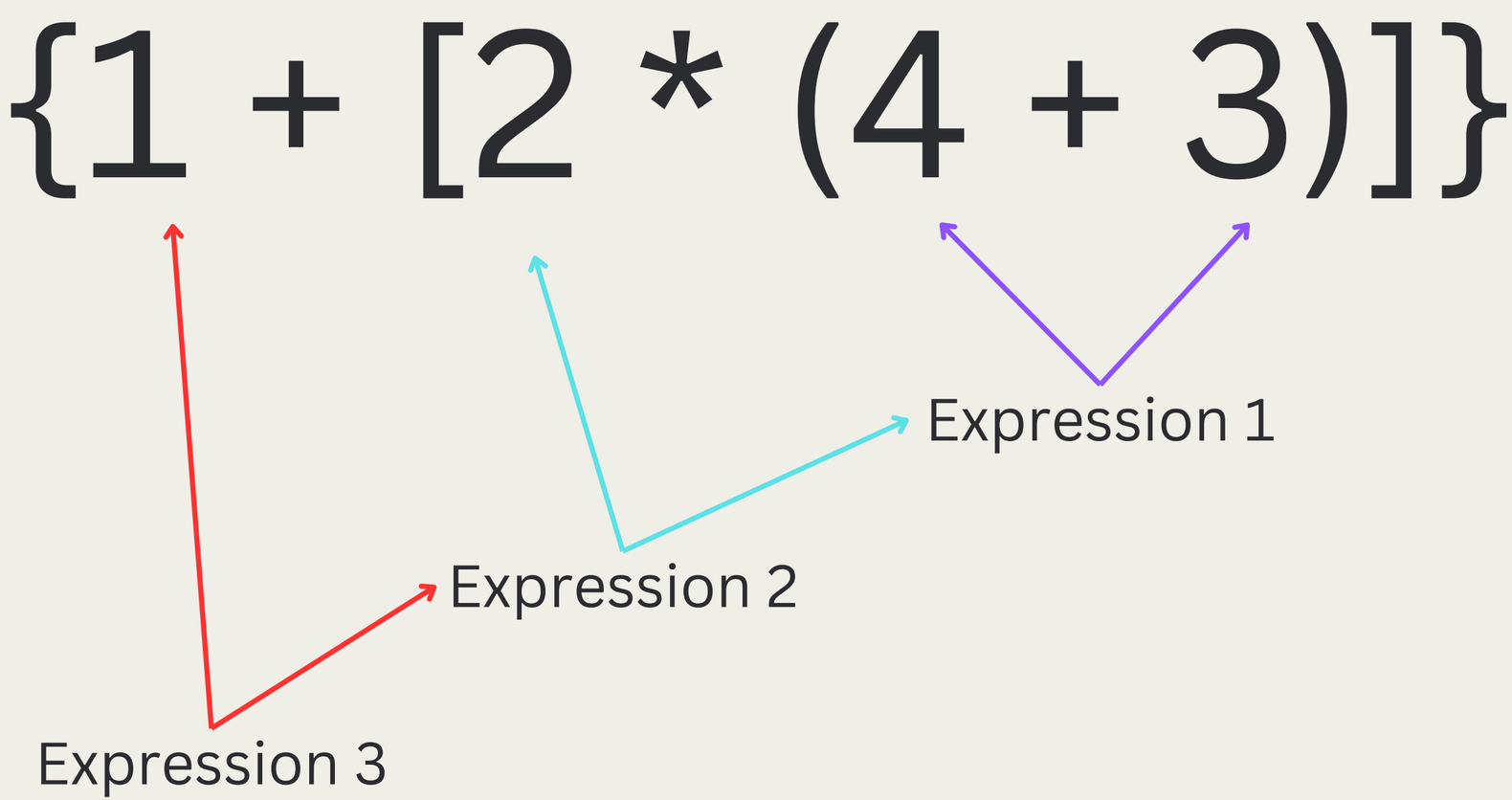
There should be a **Calculator** class as well which can have an expression passed in, and calculate that expression.

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

COMPOSITE PATTERN

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

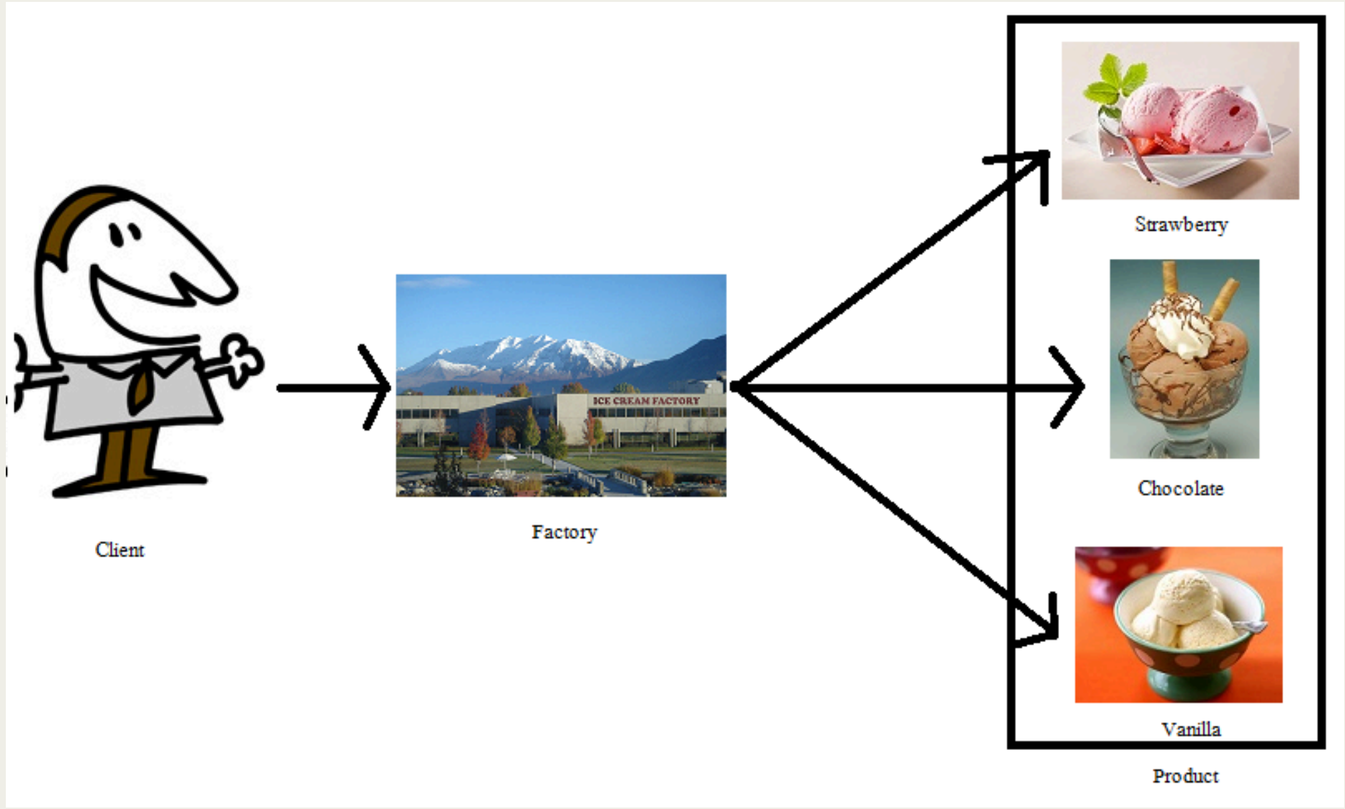
COMPOSITE PATTERN



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.

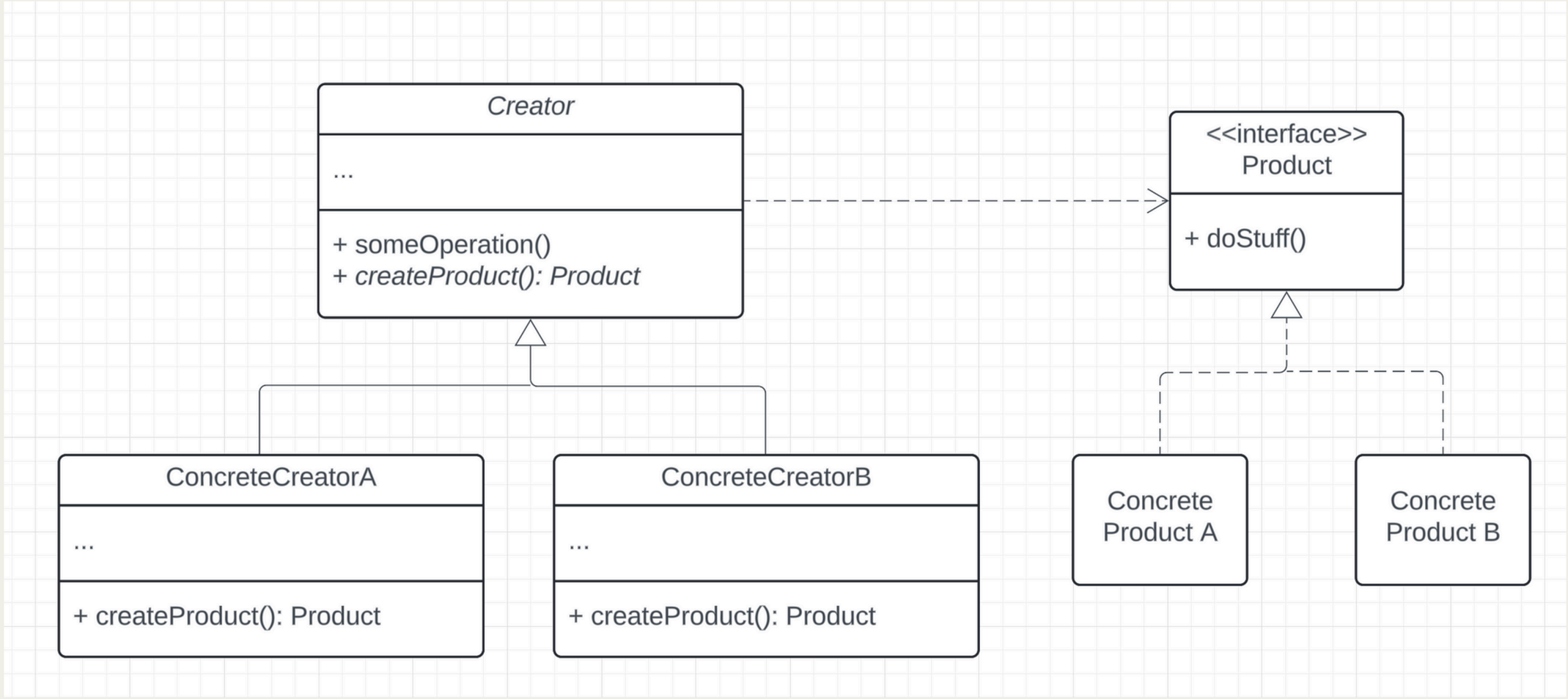
```
1  public Programmer createProgrammer(String team) {
2    return new Programmer(DEFAULT_PROGRAMMER_SALARY, team, makeNewUniqueID());
3  }
```

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  }
```

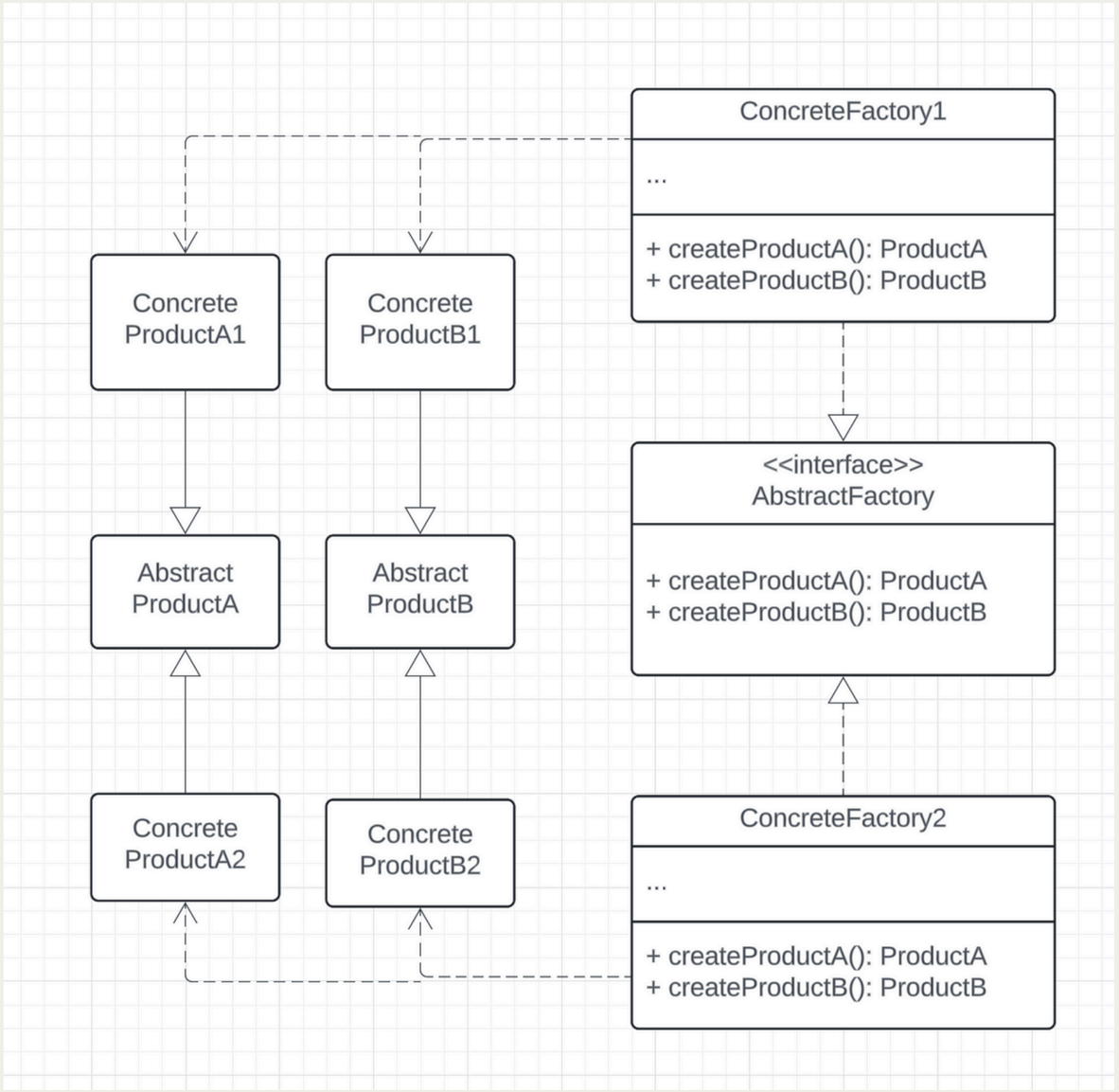
FACTORY PATTERN- ABSTRACT FACTORY PATTERN

- 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.

LABORATORY

MARKING SESSION