# Announcements

- CATME Midterm Evaluation Out
- Midterm Anonymous survey Out
- Extra Credit Opportunity in today's class
- Project Progress Report due 6/15
- Sit with your teammates today in class for an activity
- Final Project Presentation, Code and Report Assignments Out

CS3300 Introduction to Software Engineering

L8: Design Patterns; Intro to testing

Dr. Nimisha Roy   ▸ nroy9@gatech.edu

What are Design Patterns?
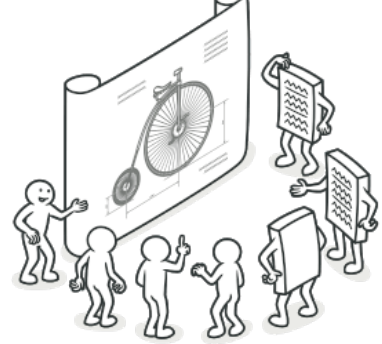
- **Typical solutions to common problems in software design**. Each pattern is like a blueprint that you can customize to solve a particular design problem in your code.

- Patterns define industry standard strategies for solving common problems.

- By using design patterns, you can make your code more flexible, reusable, and easier to maintain.

What are Design Patterns?
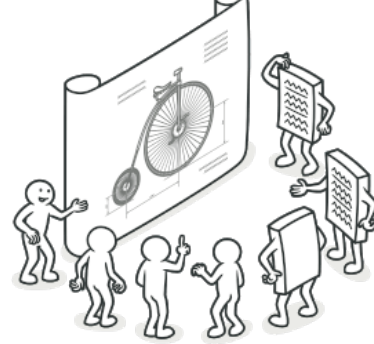
- Design pattern is a problem & solution in context - solution/strategy reuse
- Goals:
  - To support reuse, of
    - Successful designs
    - Existing code (though less important)
  - To facilitate software evolution
    - Add new features easily, without breaking existing ones
  - Reduce implementation dependencies between elements of software system.

Design Patterns: Origin

Erich Gamma
Richard Hem
Ralph Johnson
John Vlissides
(gang of four)

Book "Design Patterns: Elements of Reusable OO Software"

Patterns Catalogue

Fundamental Patterns

Delegation pattern

Interface pattern

Proxy pattern

…

Creational Patterns

Abstract Factory pattern

Factory Method pattern

Lazy Initialization pattern

Singleton pattern

…

Structural Patterns

Adapter pattern

Bridge pattern

Decorator pattern

…

Behavioral Patterns

Chain of responsibility pattern

Iterator pattern

Observer pattern

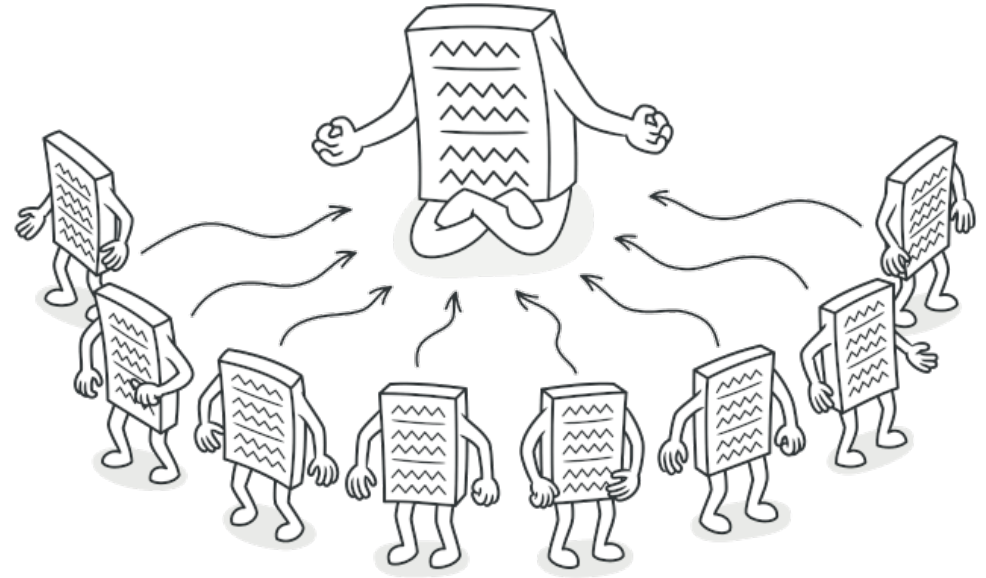State Pattern

Strategy pattern

Visitor pattern

…

Concurrency Patterns

Active object pattern

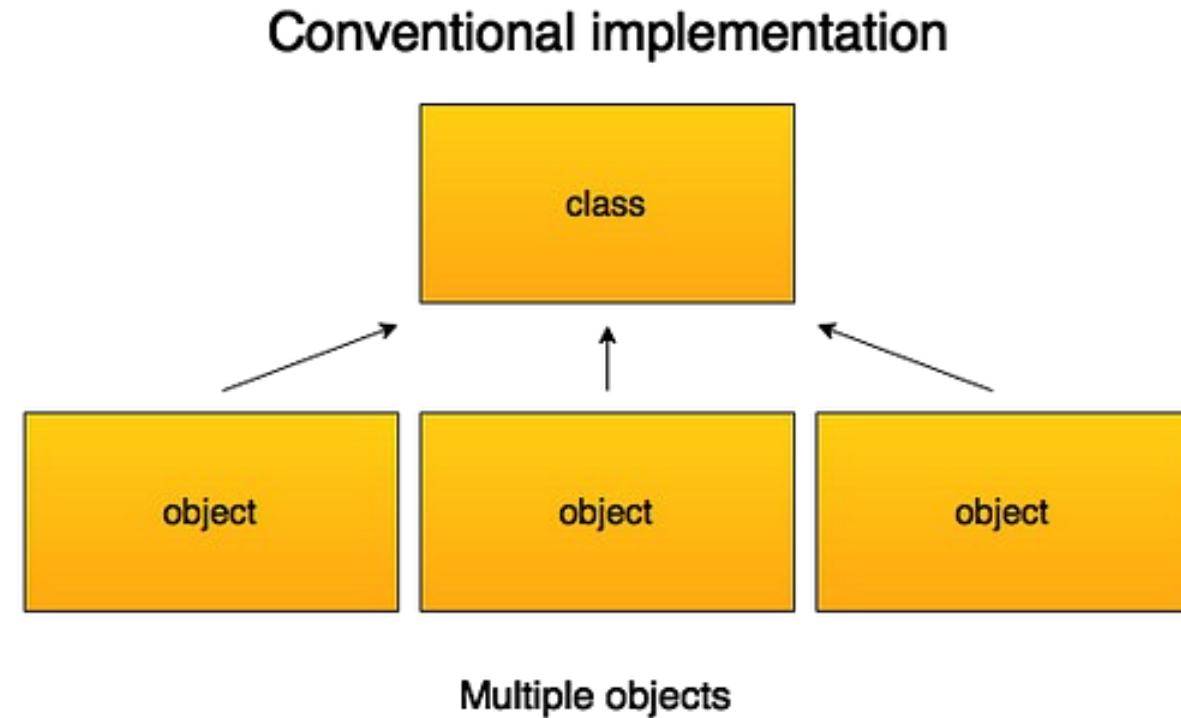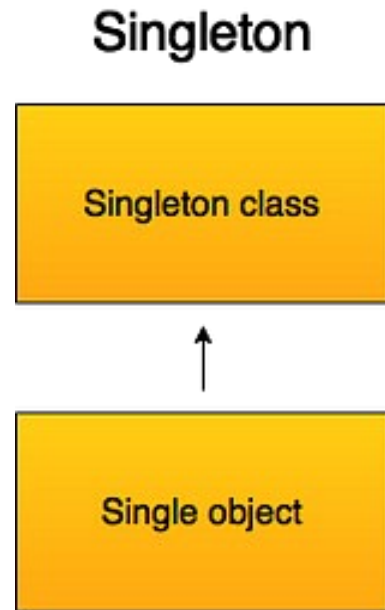Monitor object pattern

Thread pool pattern

…

Singleton Design Pattern

Singleton Pattern

- A creational design pattern

- It is used to **ensure that only one instance of a particular class ever gets created** and that **there is just one (global) way to gain access to that instance**

Singleton

Singleton class
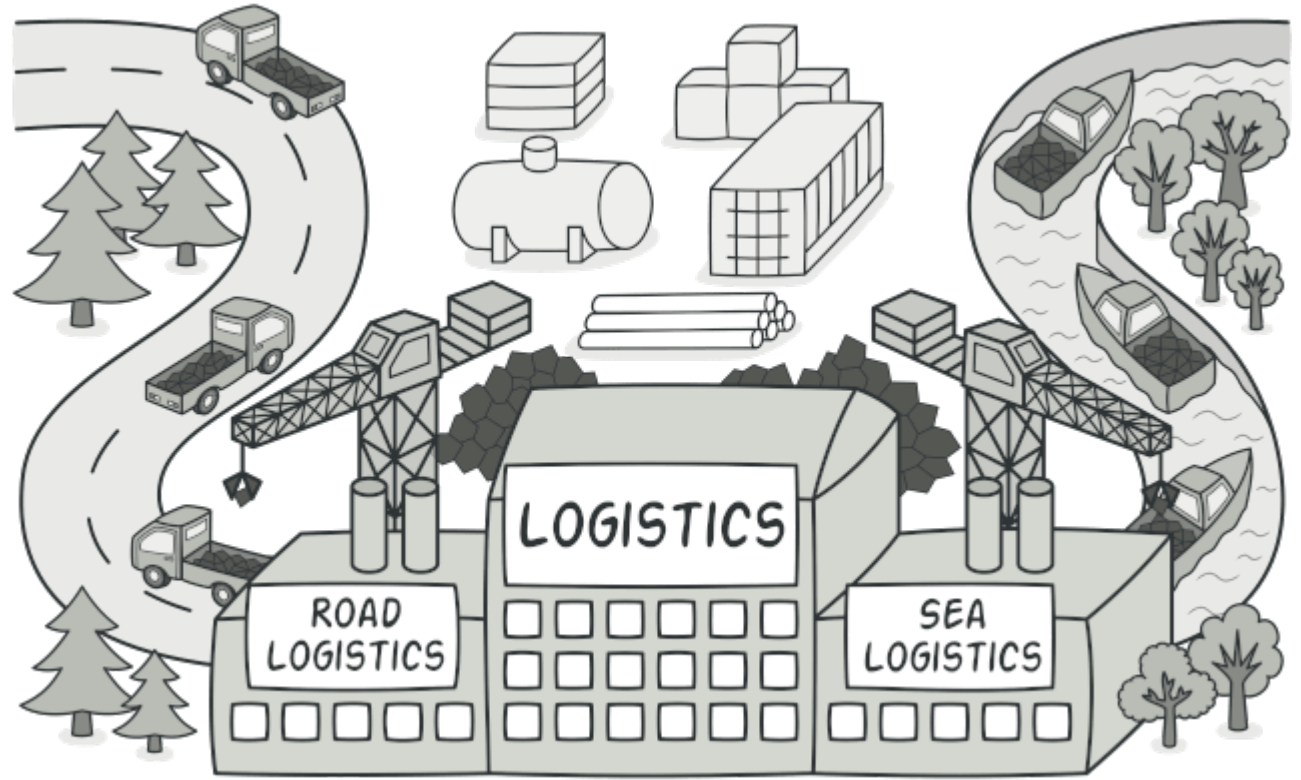
↑

Single object

Conventional implementation

class

object   object   object

Multiple objects

Singleton Class - Structure

```java
public class Singleton {

    private static Singleton uniqueInstance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }
}
```

- **private constructor** (to prevent other classes in creating new instance)
- **private static instance variable** (to store one instance)
- **public static method** to gain access to instance
  - creates object if needed; returns it

Real World Examples of Singleton Class

- **Logging systems**: Singleton logger class ensures that all log entries are written to a single instance of the logger. It provides a central point of access for logging information from different parts of the application.

- **Database connections**: In applications that require database access, a Singleton database connection class can be used to manage a single connection instance throughout the system. This ensures that multiple connections are not unnecessarily established, improving performance and resource utilization.

- **Configuration managers**: Singleton pattern is often utilized in configuration management systems where global access to configuration settings is needed. A Singleton configuration manager class provides access to configuration parameters and ensures that the settings remain consistent across the application.

Factory Method Pattern

You have a burger restaurant, and you need to create a delivery application that delivers burgers


```java
public class Restaurant {
    public ???? orderBurger(String request) {
        if ("BEEF".equals(request)) {
            BeefBurger burger = new BeefBurger();
            burger.prepare();
            return burger;
        } else if ("VEGGIE".equals(request)) {
            VeggieBurger burger = new VeggieBurger();
            burger.prepare();
            return burger;
        }
    }
}
```

Not closed for modification !

BeefBurger
- productId : int
- angus : boolean
- addOns : String
+ prepare()

VeggieBurger
- productId : int
- combo : boolean
- addOns : String
+ prepare()

We need 1 interface/abstract class that creates objects of a type and creator subclasses that instantiate specific objects.

```java
public abstract class Restaurant {
    public Burger orderBurger() {
        Burger burger = createBurger();
        burger.prepare();
        return burger;
    }
                          Factory method

    public abstract Burger createBurger();
}
```

```java
public class BeefBurgerRestaurant extends Restaurant {
    @Override
    public Burger createBurger() {
        return new BeefBurger();
    }
}
```

```java
public class VeggieBurgerRestaurant extends Restaurant {
    @Override
    public Burger createBurger() {
        return new VeggieBurger();
    }
}
```

```java
public interface Burger {
    void prepare();
}
```

```java
public class BeefBurger
              implements Burger {

    @Override
    void prepare() {
        // prepare beef
        // burger code
    }
}
```

```java
public class VeggieBurger
              implements Burger {

    @Override
    void prepare() {
        // prepare veggie
        // burger code
    }
}
```

Users of the restaurant can now directly (dynamically) invoke the concrete restaurant class implementation they need and the correct prepared burger will be returned to them.

```java
public abstract class Restaurant {
    public Burger orderBurger() {
        Burger burger = createBurger();
        burger.prepare();
        return burger;
    }

    public abstract Burger createBurger();
}
```

```java
public static void main(String[] args) {

    Restaurant beefResto = new BeefBurgerRestaurant();
    Burger beefBurger = beefResto.orderBurger();

    Restaurant veggieResto = new VeggieBurgerRestaurant();
    Burger veggieBurger = veggieResto.orderBurger();

}
```

Factory Method Pattern
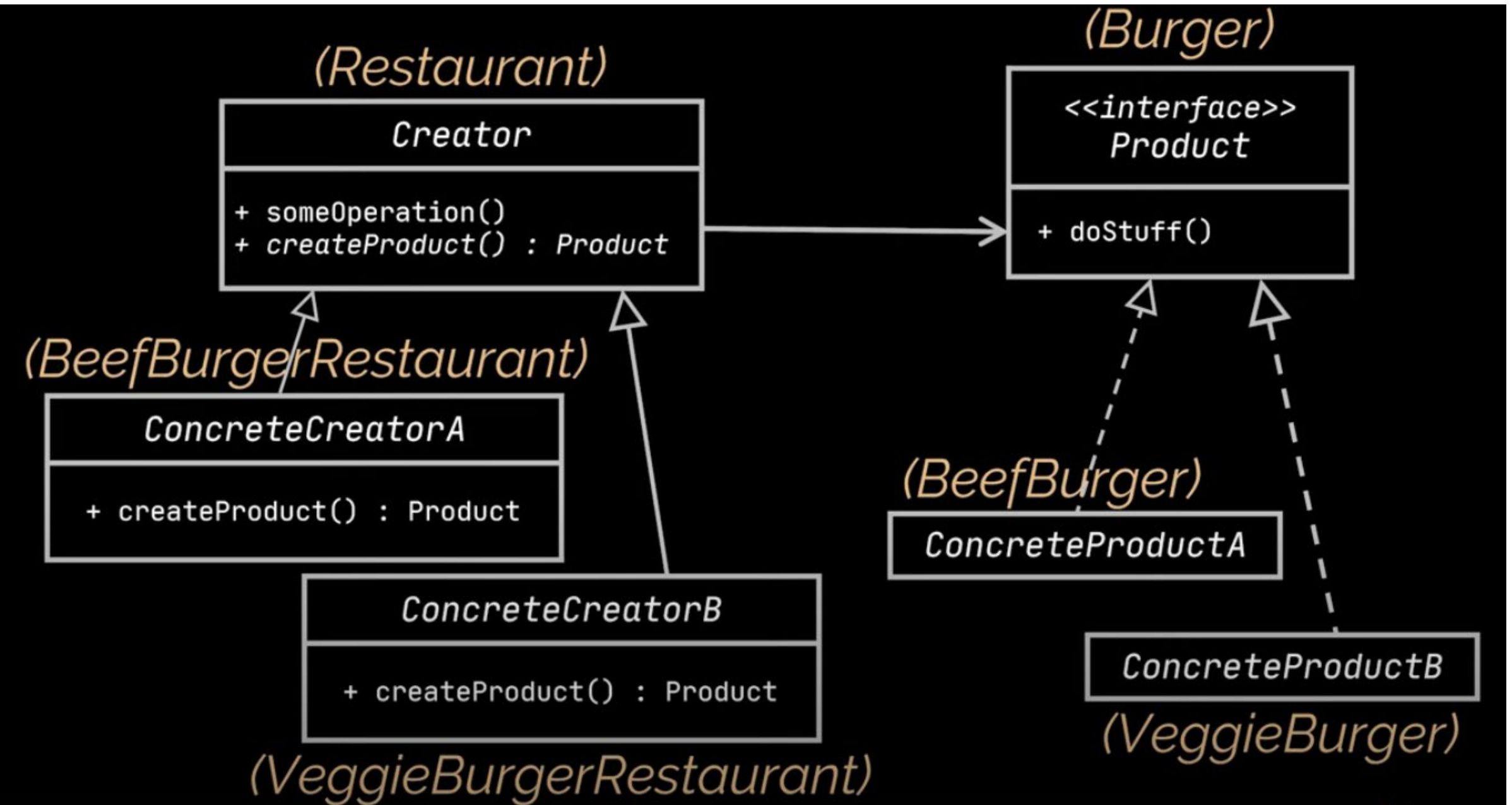
When to use a factory method pattern

- When you don't know ahead of time what class object you need to instantiate OR there is some logic associated to object instantiation
- When all of the potential classes are in the same subclass hierarchy
- To centralize class selection code
- To encapsulate object creation

centralizes the product creation code in one place in the program

use it if you have no idea of the exact types of the objects your code will work with
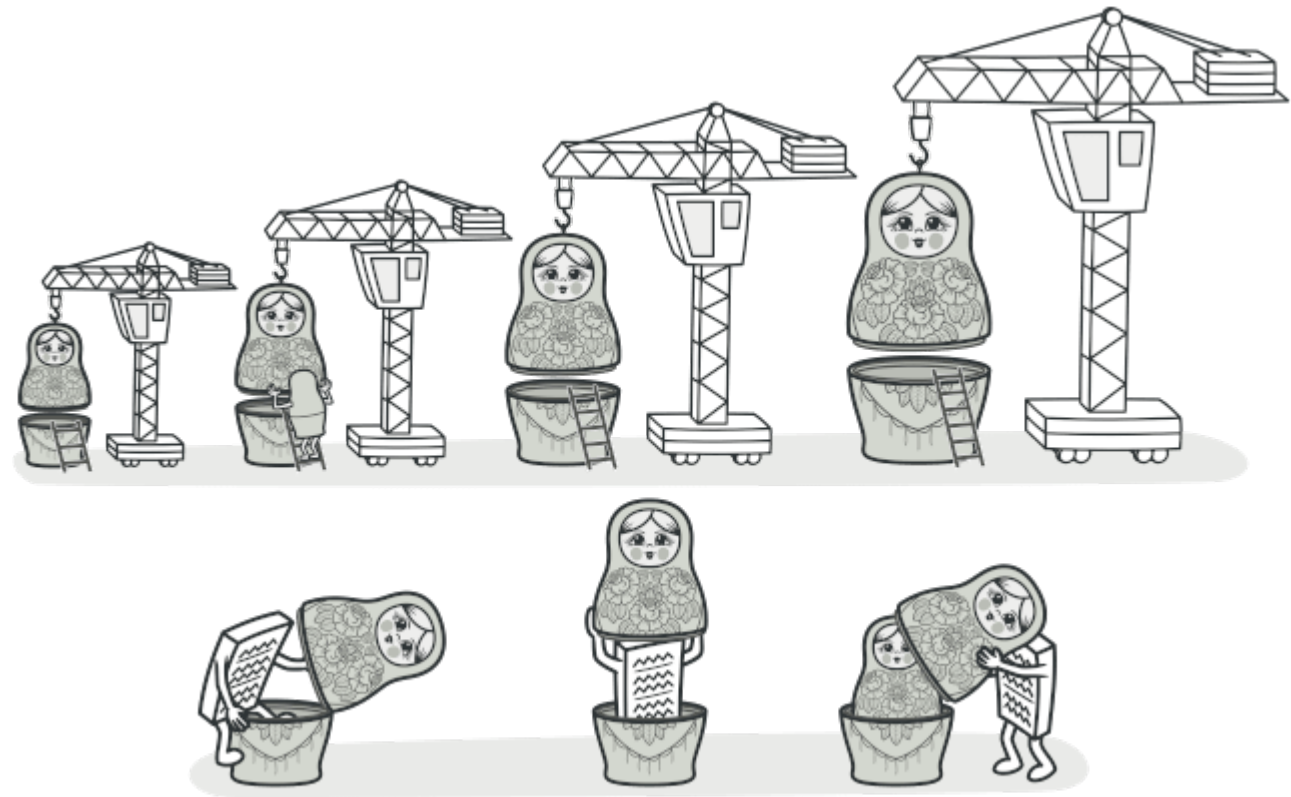
Factory Method

allows introducing new products without breaking existing code

makes it easy to extend the product construction code independently from the rest of the application

Decorator Pattern

Decorator Pattern

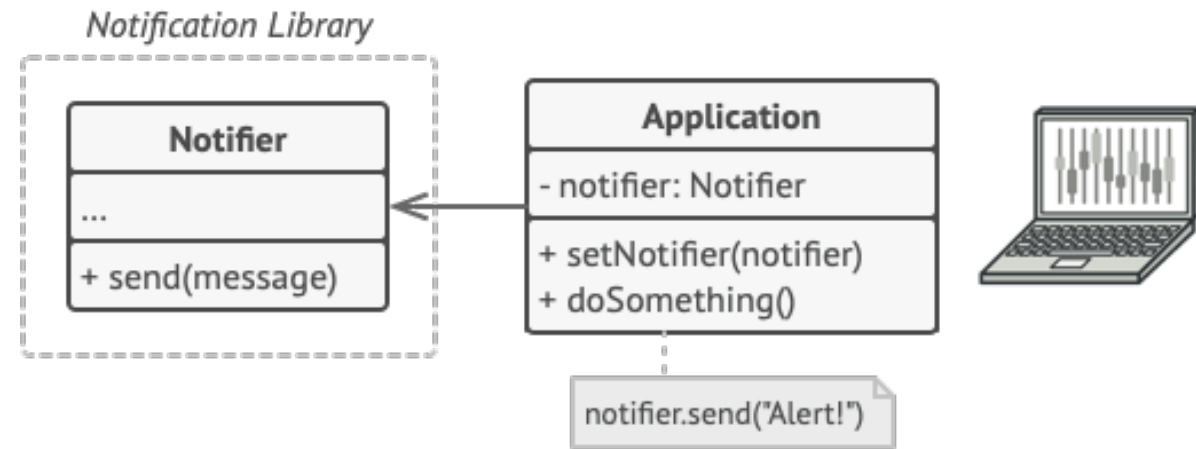- Decorator is a **structural** design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

- These new behaviors are added to the object dynamically using wrapping.

- Wrapping is just a fancy way of saying "delegation" but with the added twist that the delegator and the delegate both implement the same interface
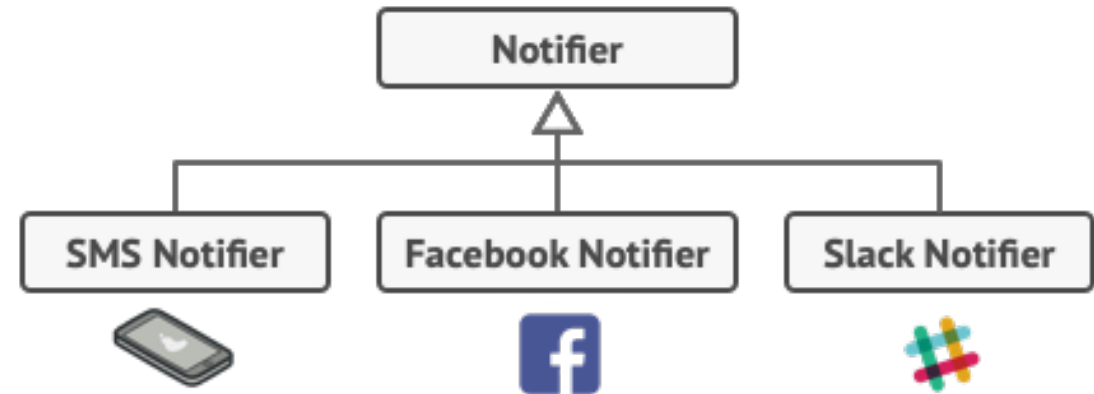
Problem

- Imagine that you're working on a notification library which lets other programs notify their users about important events.

- The initial version of the library was based on the Notifier class that had only a few fields, a constructor and a single send method.

- The method could accept a message argument from a client and send the message to a list of emails that were passed to the notifier via its constructor. A third-party app which acted as a client was supposed to create and configure the notifier object once, and then use it each time something important happened.

Notification Library

Notifier
…
+ send(message)

| Application |
| --- |
| - notifier: Notifier |
| + setNotifier(notifier)
+ doSomething() |

notifier.send("Alert!")

Problem

- You realize that users of the library expect more than just email notifications. Many of them would like to receive an SMS about critical issues. Others would like to be notified on Facebook and, of course, the corporate users would love to get Slack notifications.

- You extended the Notifier class and put the additional notification methods into new subclasses. Now the client was supposed to instantiate the desired notification class and use it for all further notifications.

- "Why can't you use several notification types at once? If your house is on fire, you'd probably want to be informed through every channel."

Problem

- You tried to address that problem by creating special subclasses which combined several notification methods within one class. However, it quickly became apparent that this approach would bloat the code immensely, not only the library code but the client code as well.

- You have to find some other way to structure notifications classes
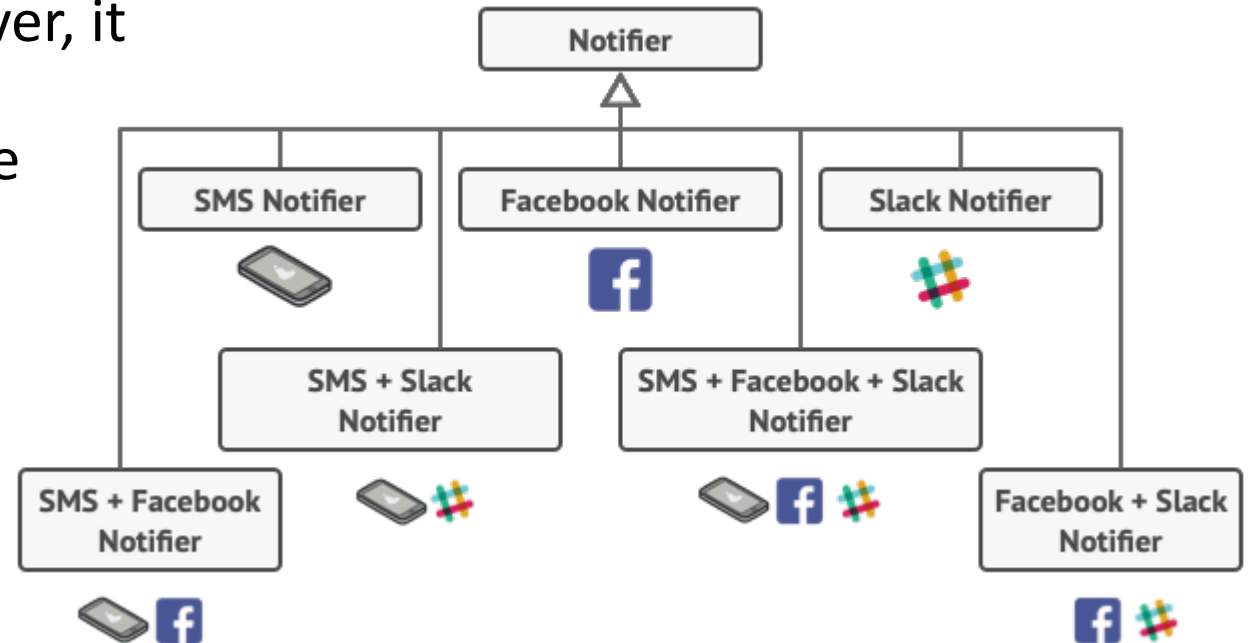
Problem with Inheritance

- Extending a class is the first thing that comes to mind when you need to alter an object's behavior. However, inheritance has several serious caveats that you need to be aware of.
  - **Inheritance is static**. You can't alter the behavior of an existing object at runtime. You can only replace the whole object with another one that's created from a different subclass.
  - **Subclasses can have just one parent class**. In most languages, inheritance doesn't let a class inherit behaviors of multiple classes at the same time.
- One of the ways to overcome these caveats is by using *Aggregation or Composition* instead of Inheritance.
  - key principle behind many design patterns, including Decorator.

Decorator Pattern

Component: This is an interface or abstract class that defines the common behavior of the objects that can be decorated. Concrete components and decorators will implement or extend this interface.

Concrete Component: Class that implements the Component interface or extends the abstract Component class. It represents the base object that can be decorated with additional behavior.
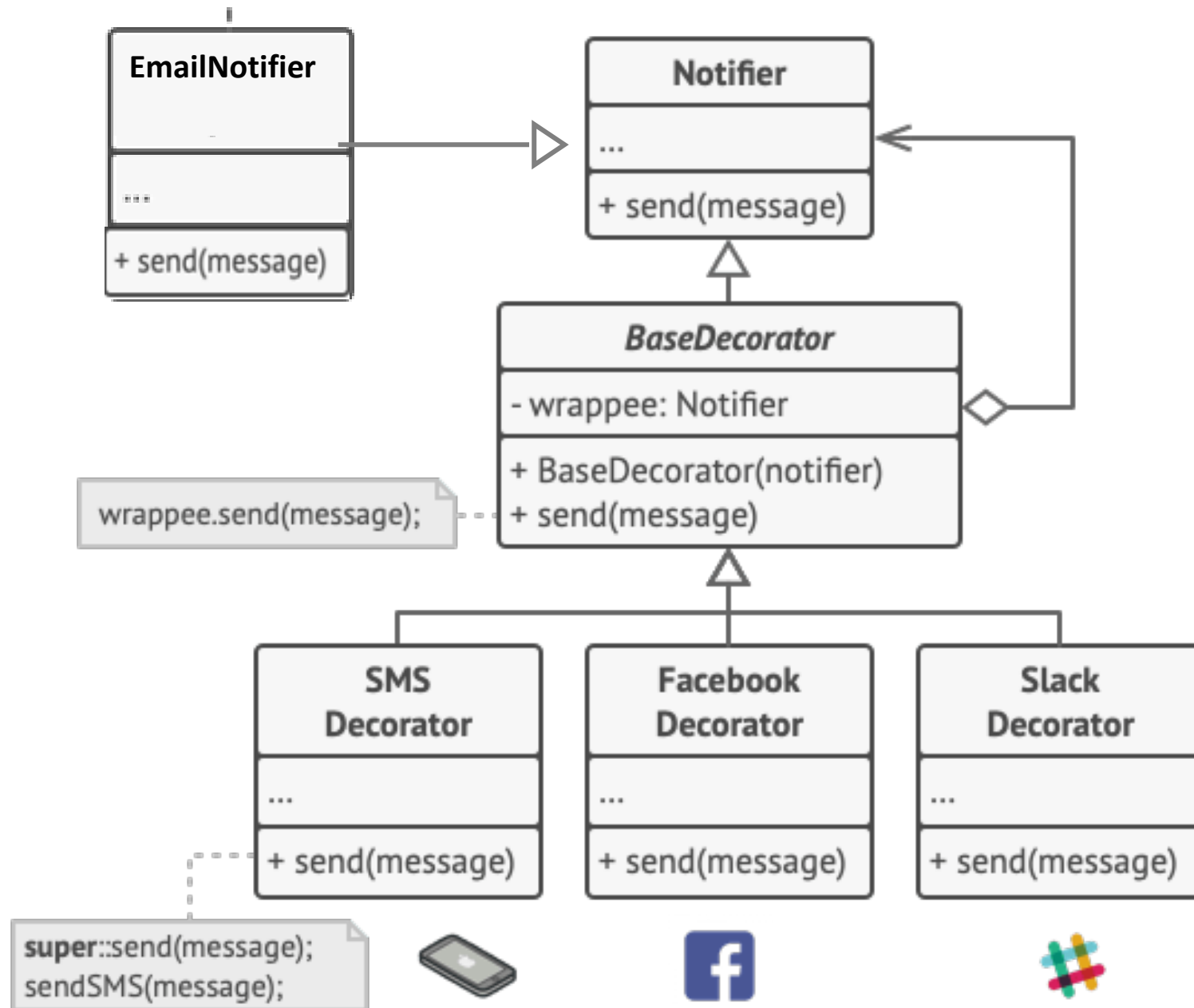
Concrete Decorators define extra behaviors that can be added to components dynamically. Concrete decorators override methods of the base decorator and execute their behavior either before or after calling the parent method.
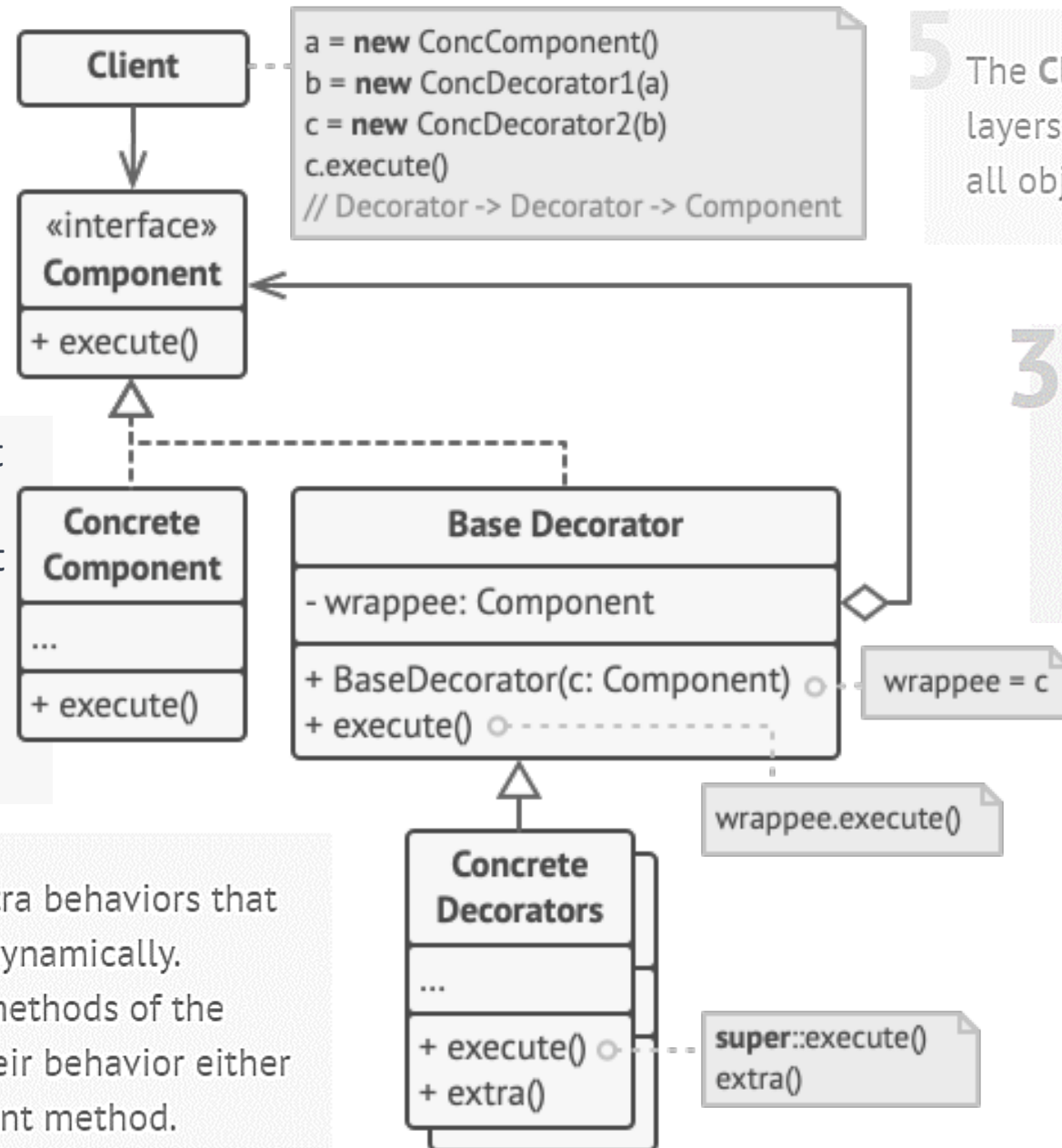
The **Client** can wrap components in multiple layers of decorators, as long as it works with all objects via the component interface.

Decorator: Abstract class that also implements the Component interface or extends the abstract Component class. It has a reference to a Component object, which represents the object it decorates. The Decorator class forwards requests to the Component it decorates and can add or modify behavior before or after forwarding the request.

Client

```
a = new ConcComponent()
b = new ConcDecorator1(a)
c = new ConcDecorator2(b)
c.execute()
// Decorator -> Decorator -> Component
```

«interface» Component

+ execute()

Concrete Component

...

+ execute()

Base Decorator

- wrappee: Component

+ BaseDecorator(c: Component)   wrappee = c
+ execute()

wrappee.execute()

Concrete Decorators

...

+ execute()   **super**::execute()
+ extra()   extra()

Solution: Decorator Pattern

EmailNotifier

...

+ send(message)

Notifier

...

+ send(message)

BaseDecorator

- wrappee: Notifier

+ BaseDecorator(notifier)
+ send(message)

wrappee.send(message);

SMS Decorator

...

+ send(message)

Facebook Decorator

...

+ send(message)

Slack Decorator

...

+ send(message)

super::send(message);
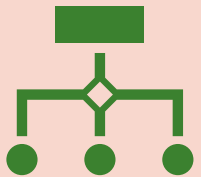sendSMS(message);
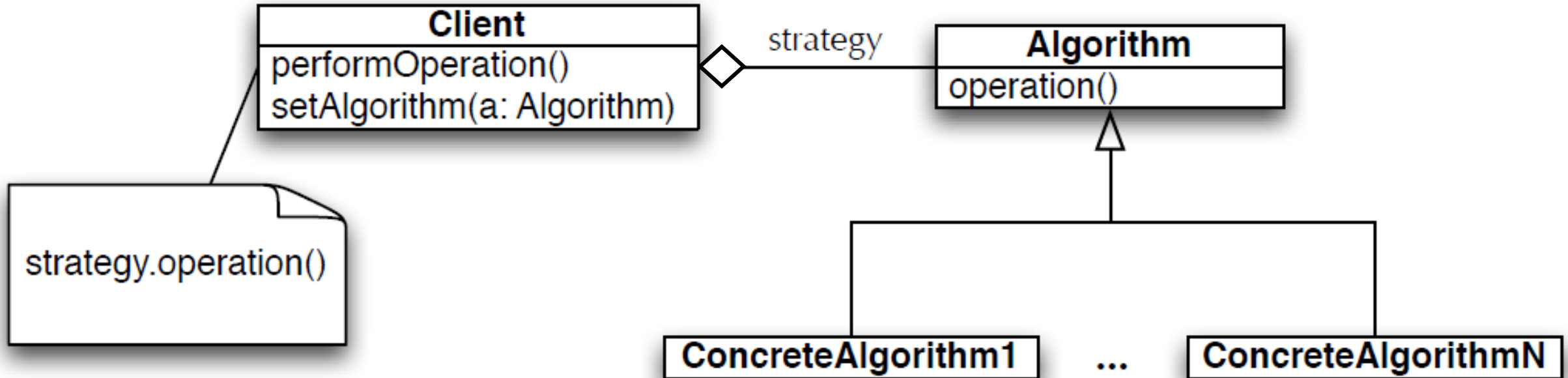
The Strategy Pattern

The Strategy Pattern

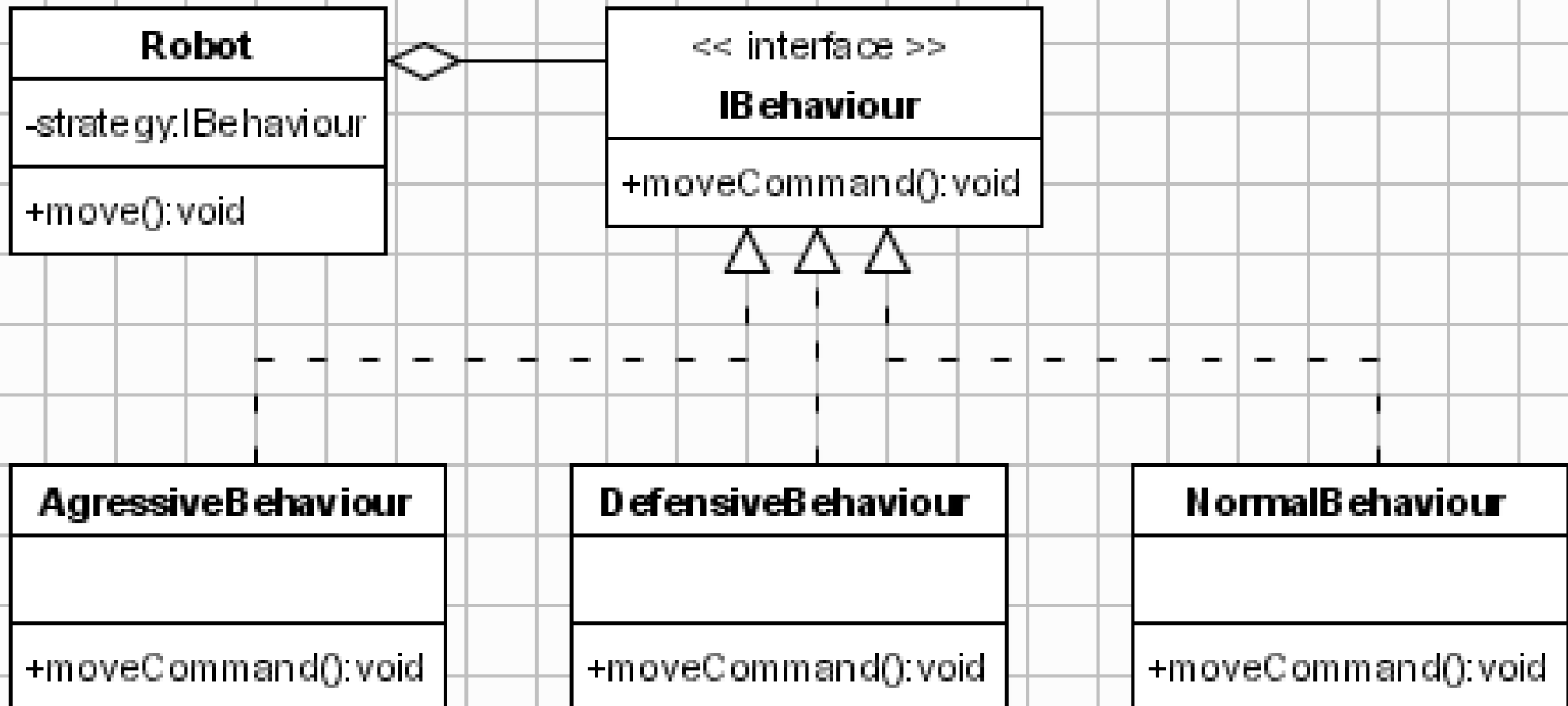Allows for switching between different behaviors for accomplishing a task

The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Structure of Strategy

- Algorithm is pulled out of Client. Client only makes use of the public interface of Algorithm and is not tied to concrete subclasses.
- Client can change its behavior by switching among the various concrete algorithms

Robot

-strategy:IBehaviour

+move():void

<< interface >>
IBehaviour

+moveCommand():void

AgressiveBehaviour

+moveCommand():void

DefensiveBehaviour

+moveCommand():void

NormalBehaviour

+moveCommand():void

Strategy Pattern example in Game development

- **Character Movement**: Strategy – Movement. Concrete Strategies – walk, fly, swim, teleport….

- **Weapon Fight Behavior**: Strategy – Fight. Concrete Strategies – Sword, Bow, Axe….

- **AI Behavior**: Strategy – Action & Reaction. Concrete Strategies - attack aggressively, attack from range, avoid opponent….

When to use Strategy Pattern?

- The Strategy pattern can be used with classes that do something specific with different strategies
  - E.g., different attack behaviors, different robot functionalities
- It splits the different strategies (by using an interface) from the context
- The code in the context is unchanged if
  - A strategy contains a bug and is fixed
 - A new strategy is added
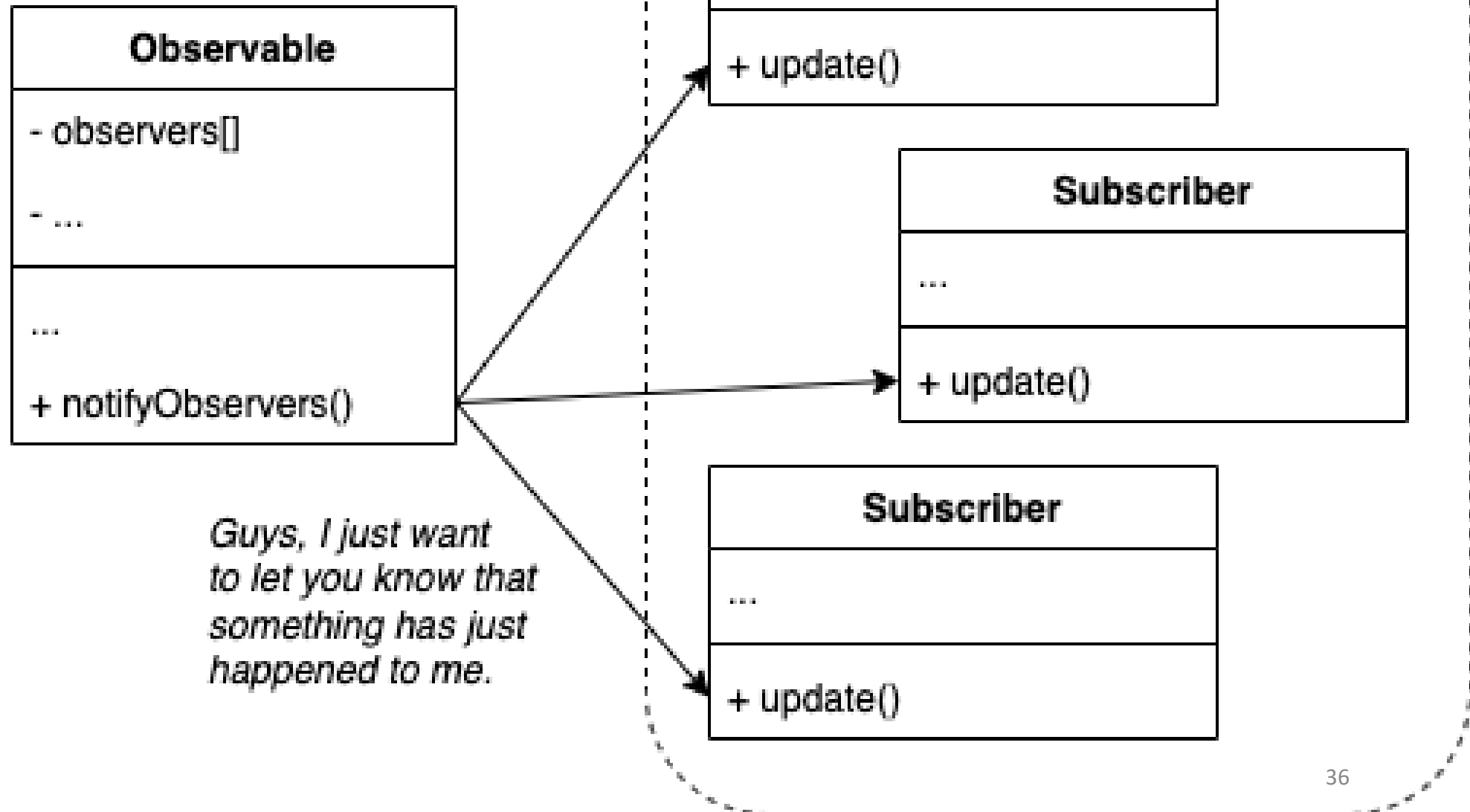
The Observer Pattern
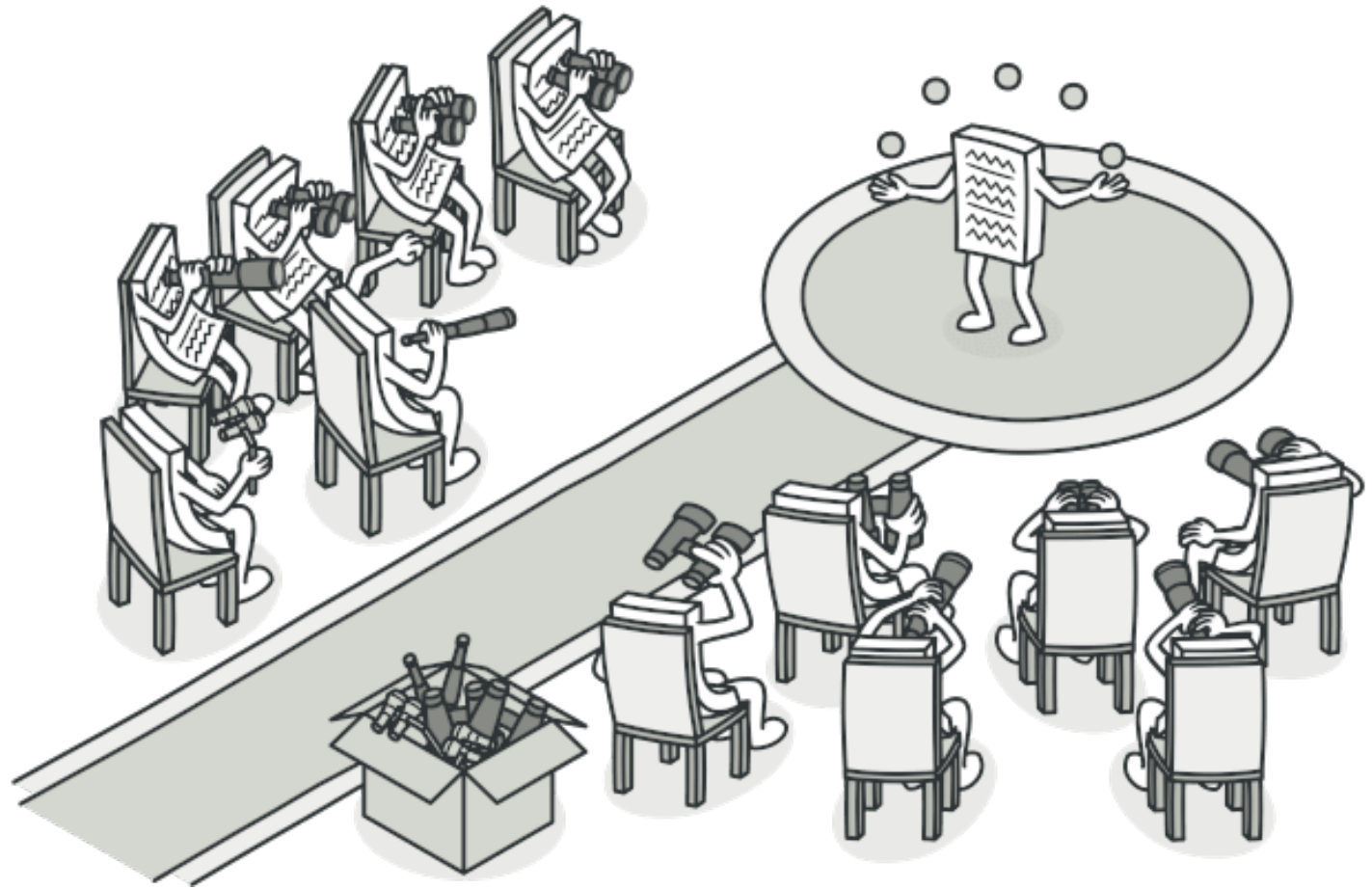
Observer Pattern

- Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing

- The observer pattern allows objects to keep other objects informed about events occurring within a software system (or across multiple systems).

- It's dynamic in that an object can choose to receive or not receive notifications at run-time

- Observer happens to be one of the most heavily used patterns in the Java Development Kit
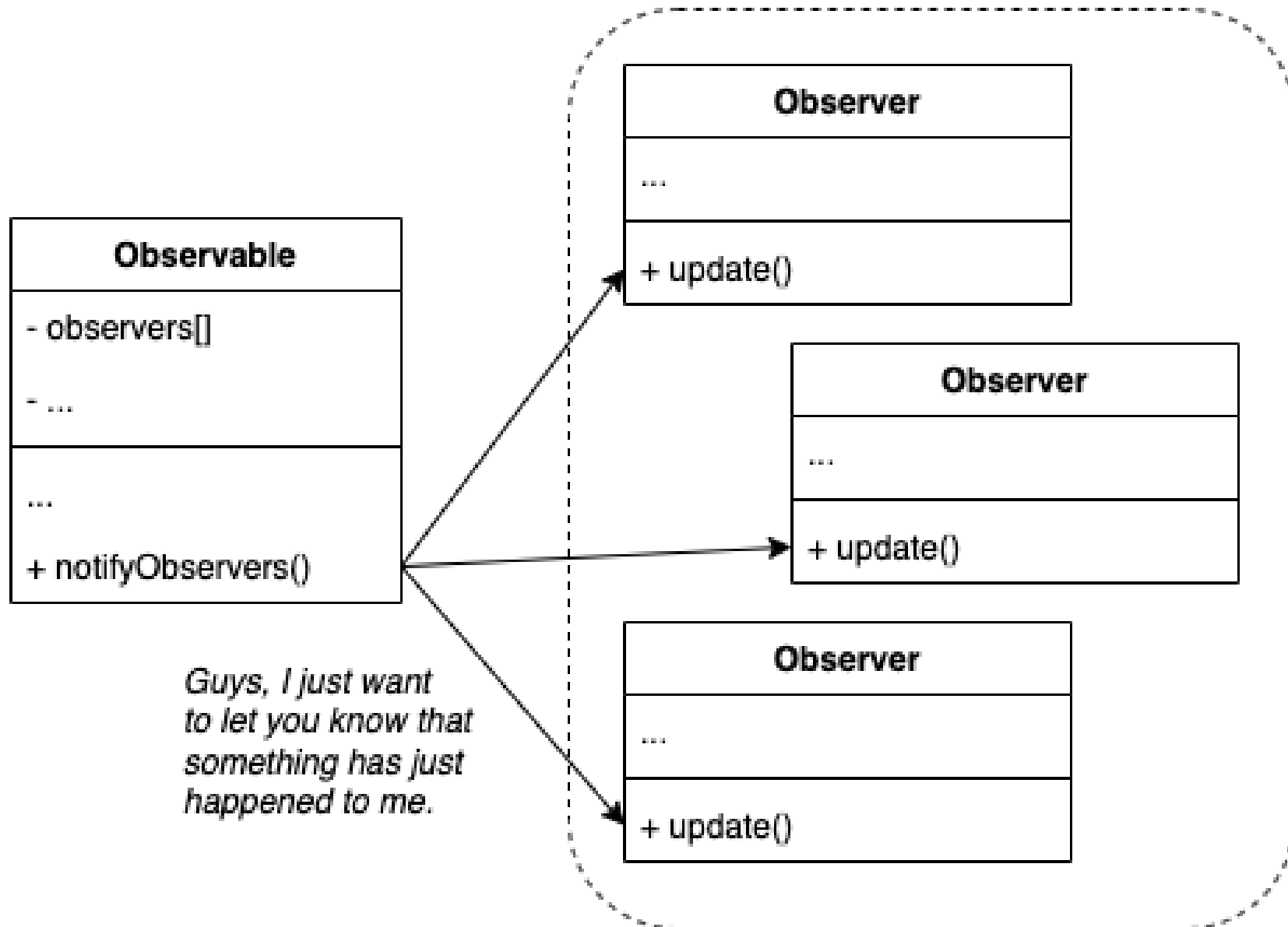
Observer Pattern
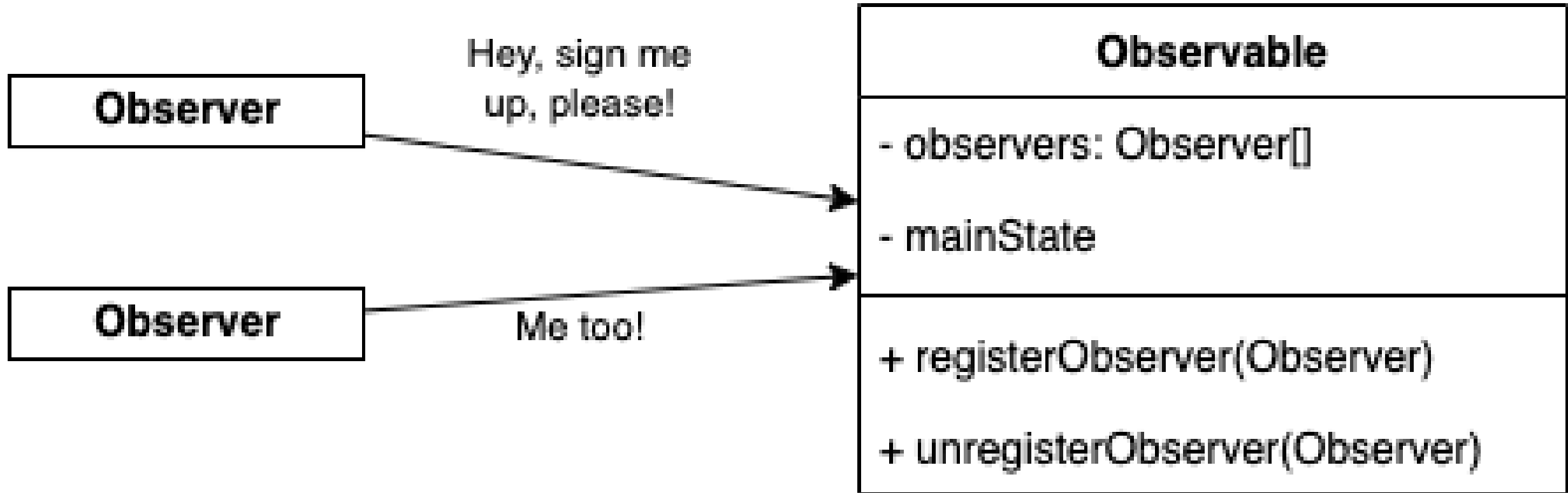
Observable/Publisher consists of 1) an array field for storing a list of references to observer/subscriber objects and 2) several public methods which allow adding observers to and removing them from that list.

Observer Pattern

Observable

- observers[]

- ...

...

+ notifyObservers()

Guys, I just want to let you know that something has just happened to me.

Subscriber

...

+ update()

Subscriber

...

+ update()

Subscriber

...

+ update()

Observer Pattern

- You wouldn't want to couple the observable to all those observer classes

- It is crucial that all observers implement the same interface, and that the observable communicates with them via that interface

Observable

- observers[]

- ...

...

+ notifyObservers()

Guys, I just want to let you know that something has just happened to me.

Observer

...

+ update()

Observer
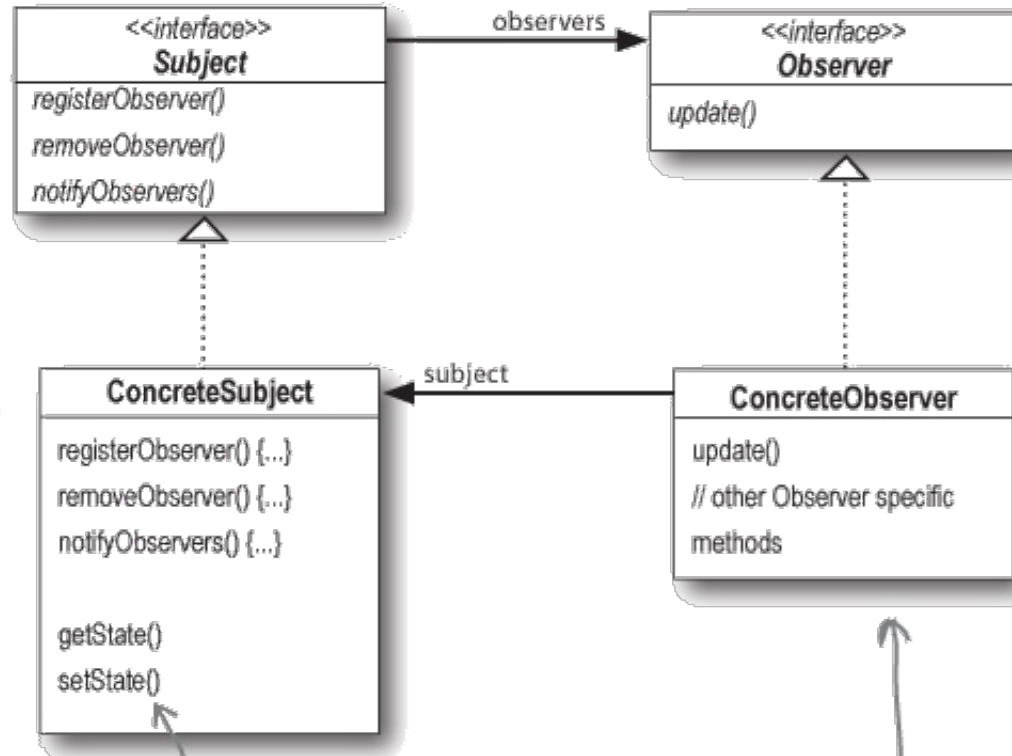
...

+ update()

Observer

...

+ update()

Observer Pattern

Here's the Subject interface. Objects use this interface to register as observers and also to remove themselves from being observers.

Each subject can have many observers.

All potential observers need to implement the Observer interface. This interface just has one method, update(), that gets called when the Subject's state changes.

```
            <<interface>>
              Subject
    ─────────────────────────
    registerObserver()
    removeObserver()
    notifyObservers()
```

observers →

```
           <<interface>>
             Observer
    ─────────────────────
    update()
```

A concrete subject always implements the Subject interface. In addition to the register and remove methods, the concrete subject implements a notifyObservers() method that is used to update all the current observers whenever state changes.

```
          ConcreteSubject
    ───────────────────────────
    registerObserver() {...}
    removeObserver() {...}
    notifyObservers() {...}

    getState()
    setState()
```

← subject

```
         ConcreteObserver
    ───────────────────────────
    update()
    // other Observer specific
    methods
```

The concrete subject may also have methods for setting and getting its state (more about this later).

Concrete observers can be any class that implements the Observer interface. Each observer registers with a concrete subject to receive updates.

Applicability

- When changes to the state of one object may require changing other objects, and the actual set of objects is unknown beforehand or changes dynamically
- When some objects in your app must observe others, but only for a limited time or in specific cases

Choosing a Pattern

Approach
- Understand your design context
- Examine the patterns catalogue
- Identify and study related patterns
- Apply suitable pattern

Pitfalls
- Selecting wrong patterns
- Abusing patterns

You will incorporate atleast 3 design patterns in your project

Activity with design patterns – code maintainability

- E-commerce Project Example - The project simulates an e-commerce order processing system. It includes classes for handling products, orders, payment processing, notifications, and discounts. However, the current implementation is tightly coupled, hard to extend, and lacks modularity due to the absence of several design patterns and presence of design flaws.

Activity with design patterns – code maintainability

- Inspect the project as a group for 10 mins and list code maintainability issues
  - List atleast 1 issue per .java class
- List design patterns that could help make maintainability better
  - List atleast 3 design patterns
- Submit your findings via assignment on Canvas

Software Testing
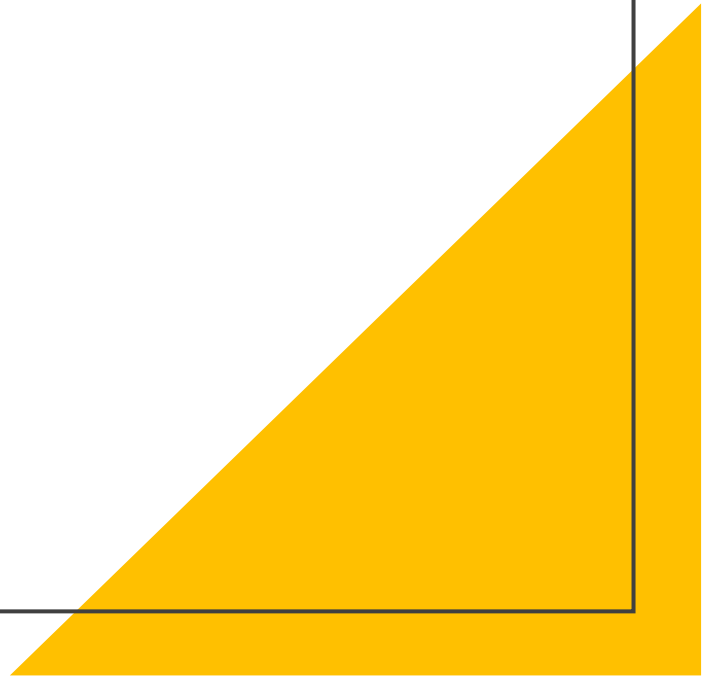
Testing is a part of Verification and Validation...

Requirements

Engineering

Design

Implementation

Maintenance

Verification &

Validation

Software is Buggy!

- Cost of bugs: $ 60 B/year

- On average, 1-5 errors per 1K LOC

- Windows 10
    - 50M LOC
    - 63,000 known bugs at the time of release
    - 1.25 per 1,000 lines

- For mass market software 100% correct SW development is infeasible, but

- We must verify the SW as much as possible

Failure, Fault, Error

Failure: Observable incorrect behavior of a program. Conceptually related to the behavior of the program, rather than its code.
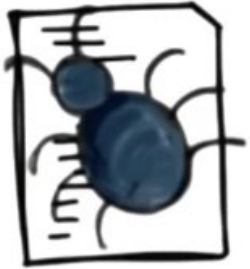
Fault (bug): Related to the code. Necessary (not sufficient!) condition for the occurrence of a failure.

Error: Cause of a fault. Usually a human error (conceptual, typo, etc.)

Failure, Fault, Error: Example

1. double doubleValue(int param) {
2.   double result;
3.   result = (double) param * param;
4.   return(result);
5. }

A call to doubleValue(3) returns 9. What is this?

The result 9 is a failure- it is an observable behavior

Where is the fault?

Line 3

What is the error that caused the fault?

N/A. Maybe typo, erroneous copy paste, or conceptual. Only the developer knows.

Approaches to Verification

- **Testing** (dynamic verification): exercising software to try and generate failures

- **Static analysis**: identify (specific) problems statically, that is, considering all possible executions

- **Inspections/reviews/walkthroughs**: systematic group review of program text to detect faults

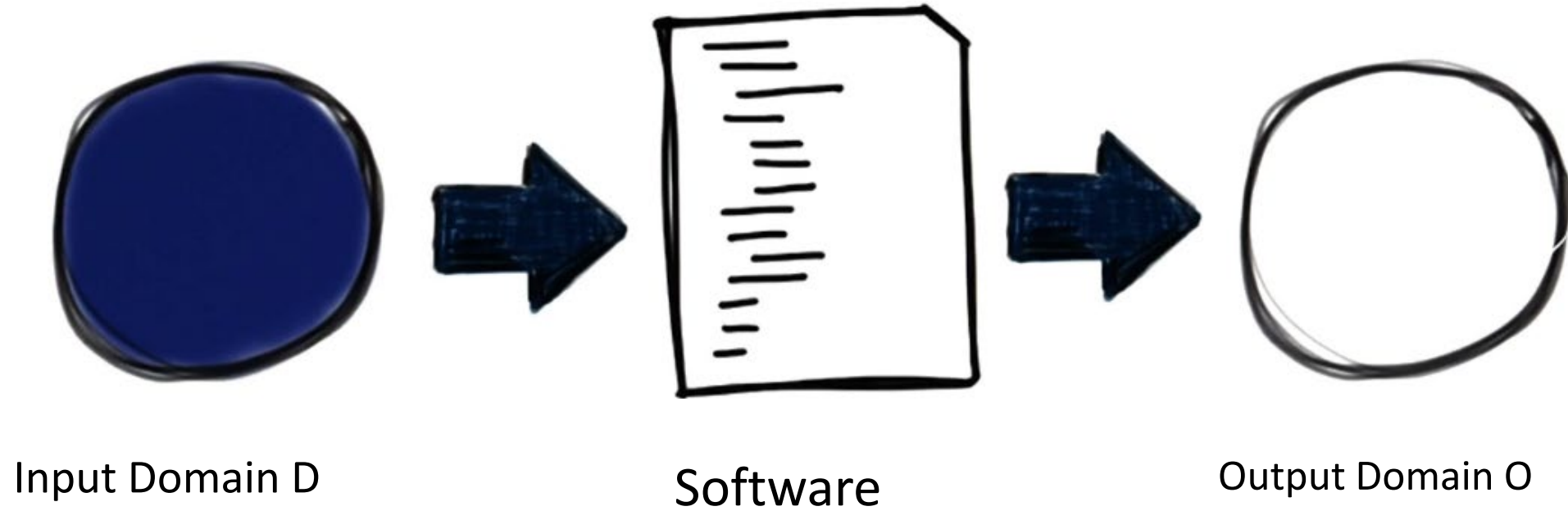- **Formal verification** (proof of correctness): proving that the program implements the program specification
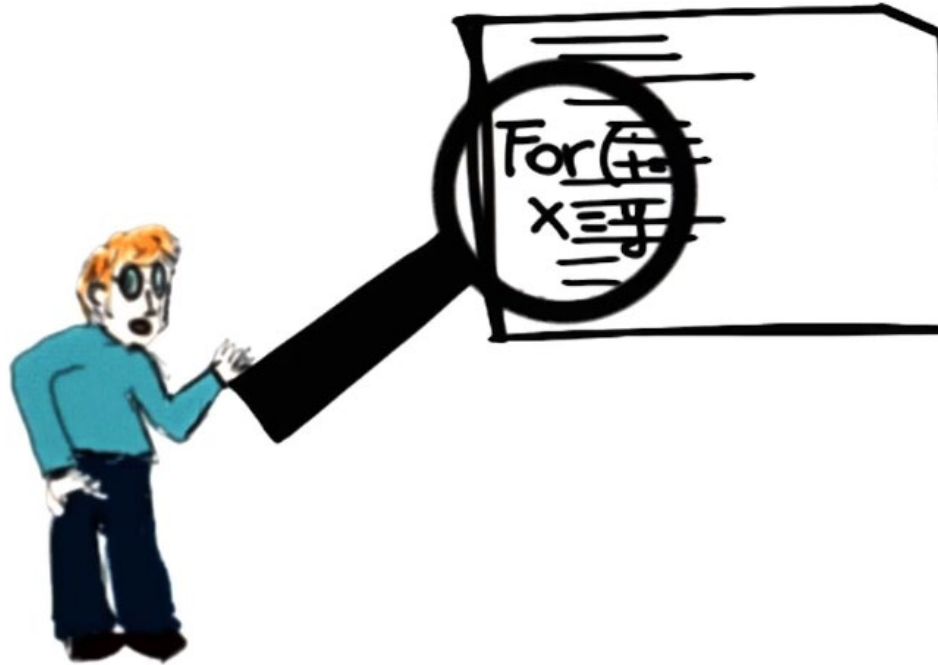
Testing

Input Domain D ⟶ Software ⟶ Output Domain O
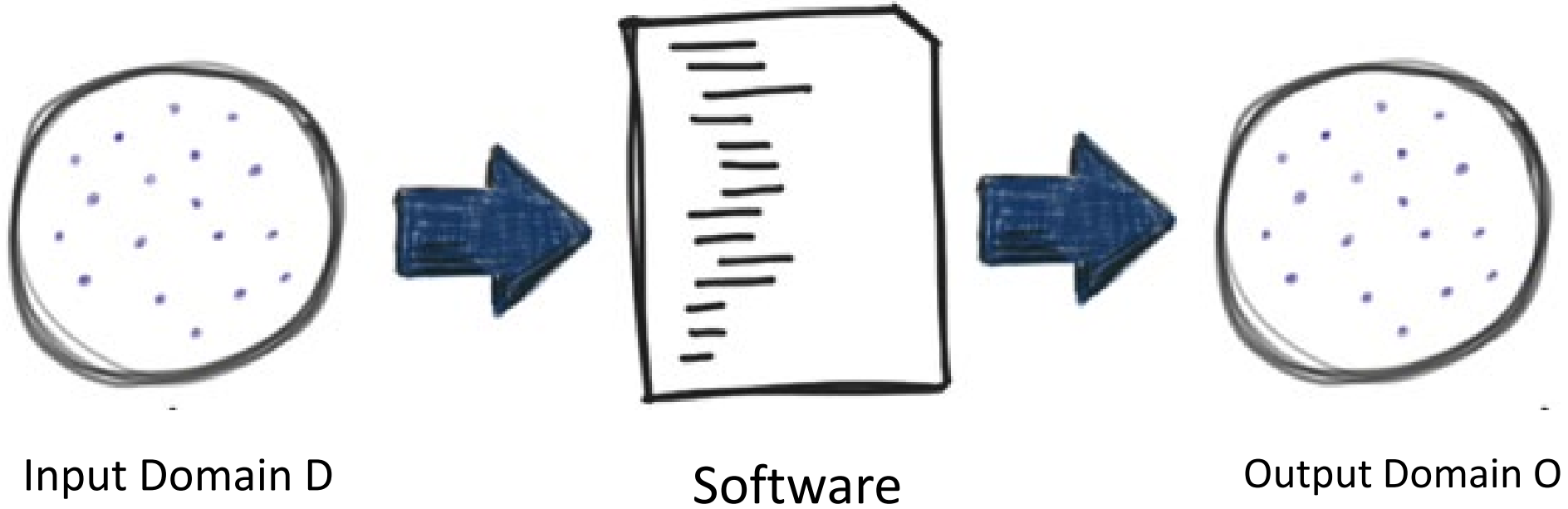
Test Case: {i ∈ D, o ∈ O}

Test Suite: A set of Test Cases

Static Verification

Input Domain D   Software   Output Domain O

Considers all possible inputs
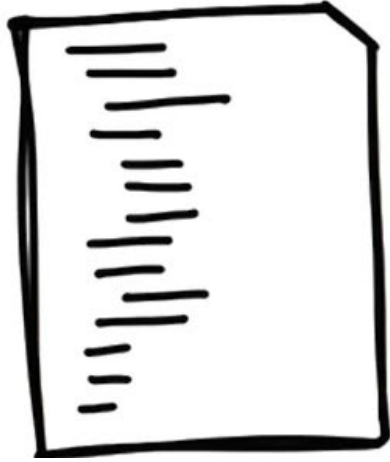(execution/behaviors)

Inspections/Reviews/Walkthroughs

Human intensive activity

Manual

Group activity

Inspect defects in the artifacts by identifying faults

Formal Proof (Of correctness)

Program

Specification

Given a formal specification, checks that the code corresponds to such specification

Sophisticated mathematical analysis

Comparison among the 4 techniques

	PROS	**CONS**
Testing	No False Positives	Highly Incomplete
Static Verification	Considers all program behaviors, Complete	False Positives, Expensive
Inspections	Systematic, Thorough	Informal, Subjective
Formal Proofs of Correctness	Strong Guarantees	Complex, Expensive to build/prove a mathematical basis

Today, Quality Assurance (Verification) is mostly Testing

"50% of my company employees are testers, and the rest spend 50% of their time testing".

- Bill Gates

What is Testing?

Testing == To execute a program with a sample of the input data

- Dynamic technique: program must be executed

- Optimistic approximation:

  - The program under test is exercised with a (very small) subset of all the possible input data

  - We **assume** that the behavior with any other input is consistent with the behavior shown for the selected subset of input data

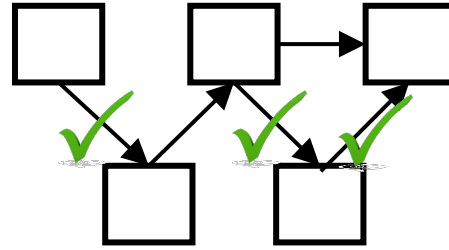# Successful Tests

"A test is successful if the program fails"

-Goodenough and Gerhart (1985). "Towards a Theory of Test data selection". *IEEE Transactions of Software Engineering,* Jan 1985

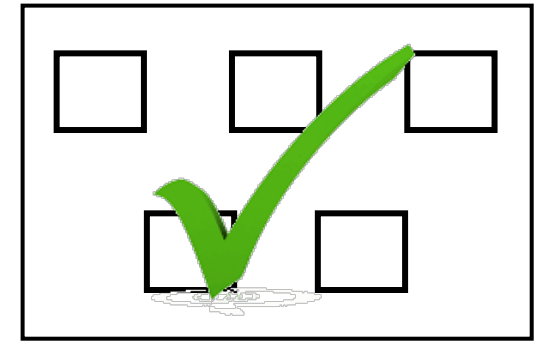# Testing Granularity Levels

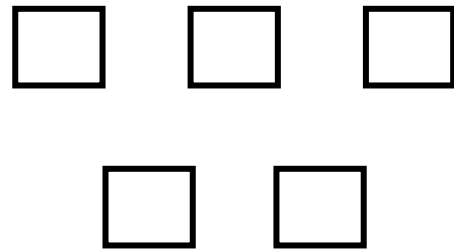Unit Testing
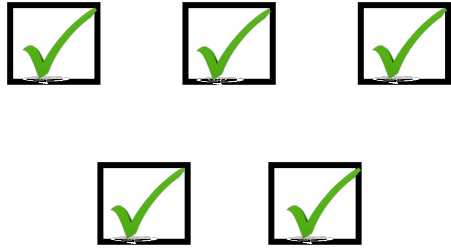
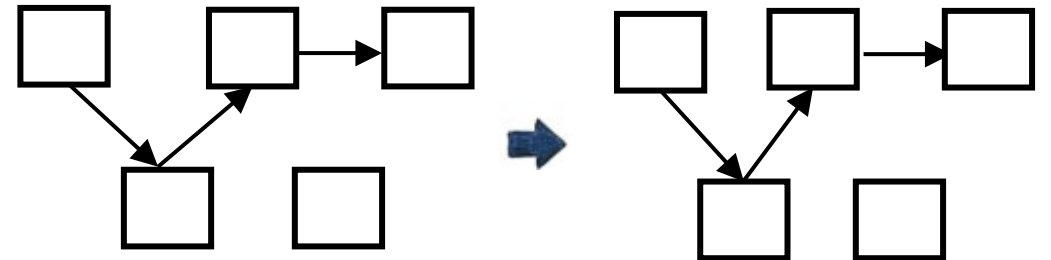Integration

Big Bang

System

Functional/Non-functional

Acceptance Testing

Customer

Regression Testing
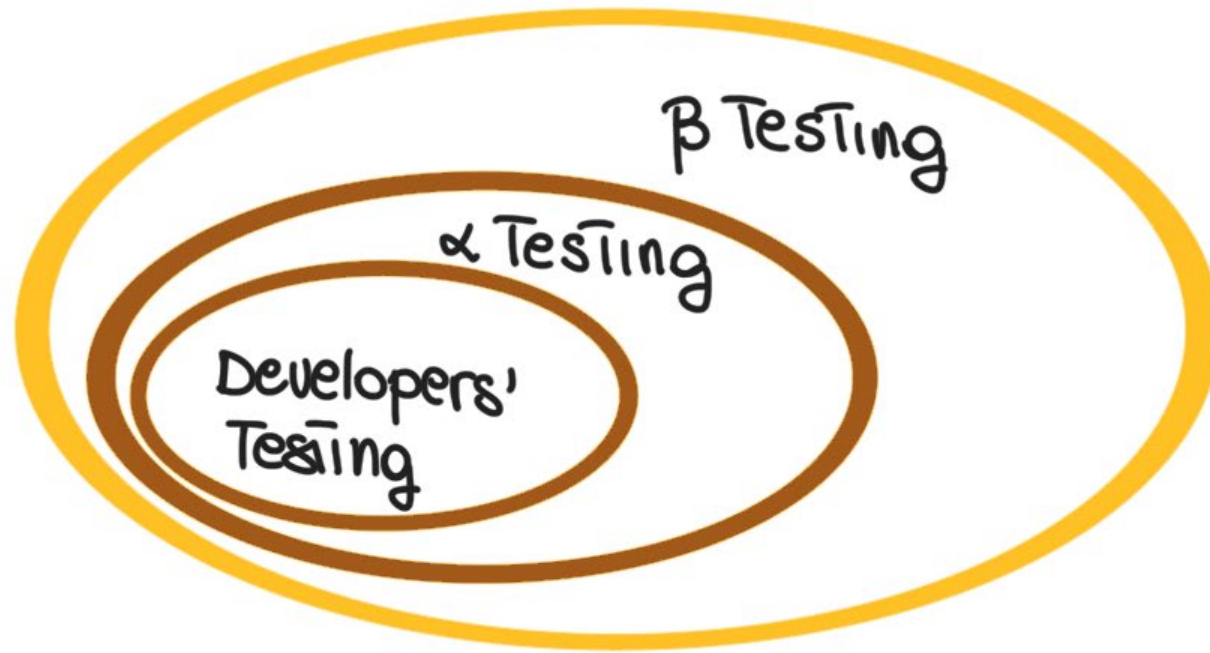
Testing Stages

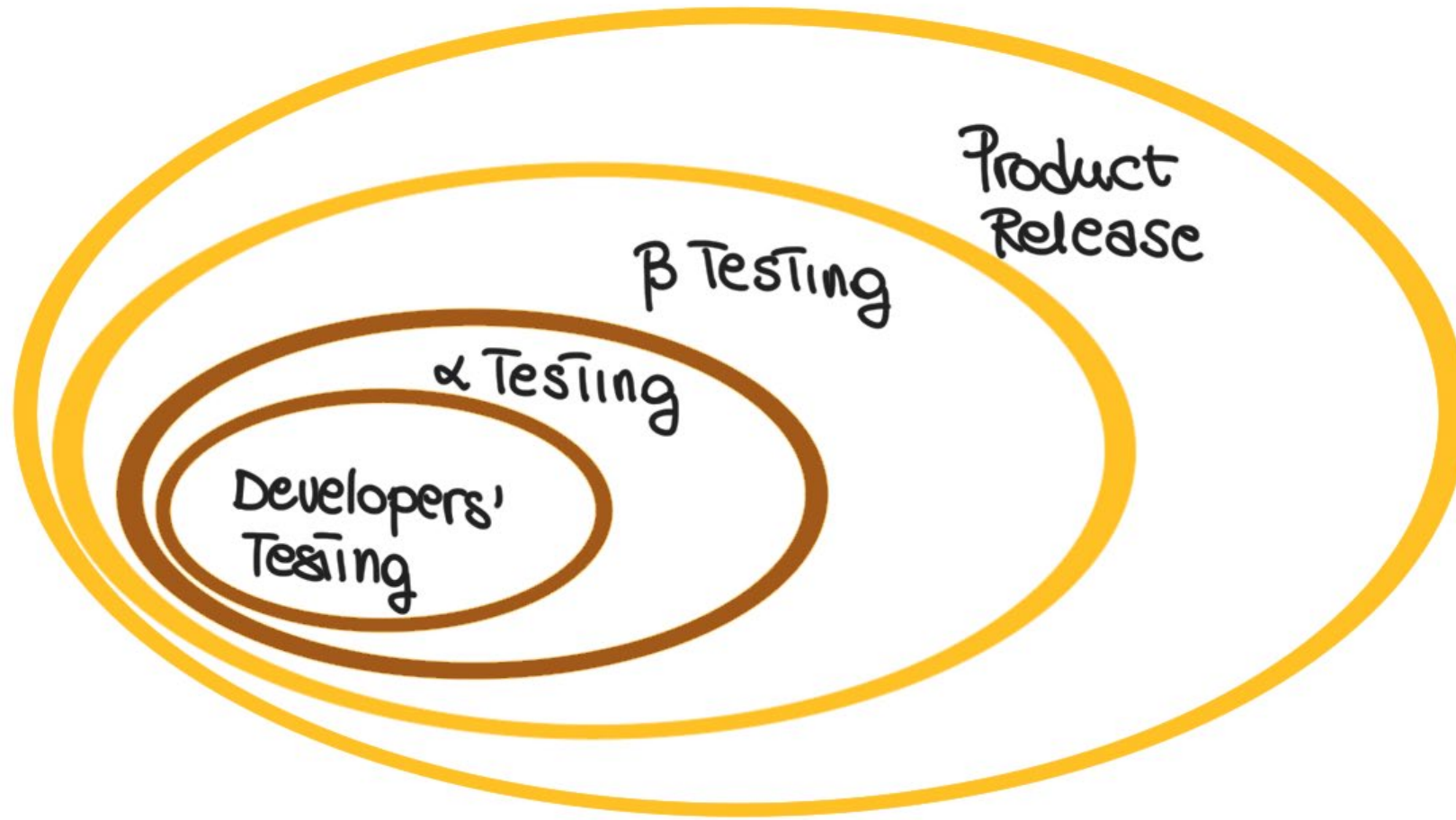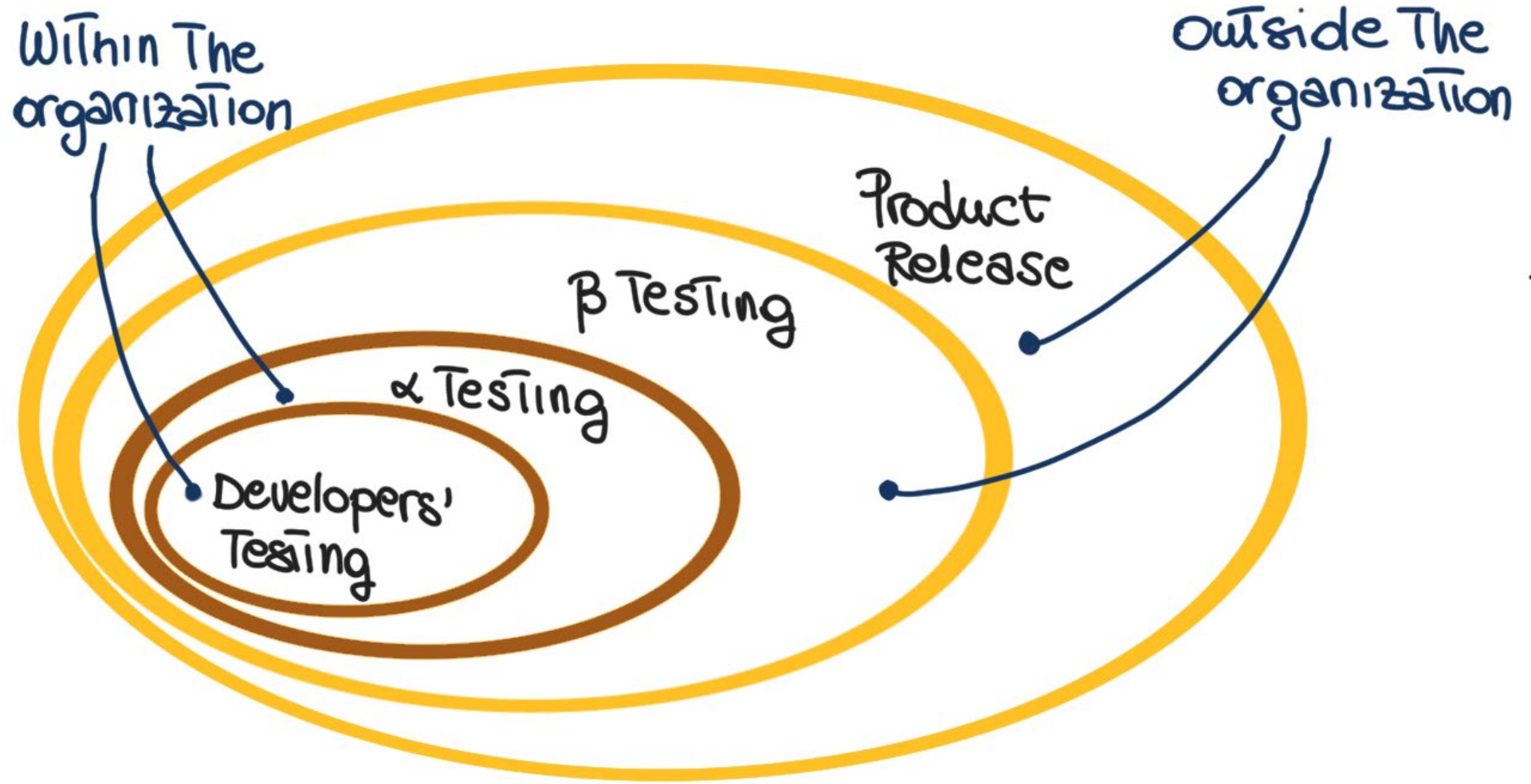# Testing Stages

α Testing

Developers'
Testing

Testing Stages

Testing Stages

Testing Stages

Testing Techniques

There are several techniques

- Different processes
- Different artifacts
- Different approaches

There are no perfect techniques

- Testing is a best-effort activity

There is no <u>best</u> technique

- Different contexts
- Complementary strengths and weaknesses
- Trade-offs

Testing Techniques

BLACK BOX TESTING

- Based on a description of the software (specification)
- Cover as much specified behavior as possible
- Cannot reveal errors due to implementation details

WHITE BOX TESTING

- Based on the code
- Cover as much coded behavior as possible
- Cannot reveal errors due to missing paths

Black-Box Testing Example

Specification: Inputs an integer and prints it

```
1.  void printNumBytes (param){
2.     if (param < 1024) printf("%d", param);
3.     else  printf('%d KB" ,  param/124);
4.  }
```

Blackbox testing attempts: Inputs +, -, and 0

Will blackbox testing be able to catch the failure?   **Most likely Not**

White-Box Testing Example

Specification: Inputs an integer and prints it

```
1.  void printNumBytes (param){
2.      if (param < 1024) printf("%d", param);
3.    else  printf('%d KB" ,  param/124);
4.  }
```

Whitebox testing attempts: Cover all 4 statements or the 2 paths.
So, <1024, =1024, >1024

Will whitebox testing be able to catch the failure? **Most likely Yes**

Black-box Testing Example: more effective

User Interface (UI) Testing

- Imagine a simple login form for a web application with the following fields: Username (text field), Password (text field), Login Button
- The form is designed to authenticate users based on their input. The requirements specify that: A username must be between 5 to 15 characters; The password must be at least 8 characters long; Special characters are allowed in the password but not in the username.

Why Black-Box Testing Could Be More Effective:

- **Focus on User Behavior**: Black-box testing is effective here because it focuses on how a user interacts with the form rather than the internal logic behind the input validation.
- **Testing Inputs and Outputs**: It can test various scenarios like valid and invalid usernames, short and long passwords, special characters, and empty fields, verifying that the user gets the correct messages like "Username must be at least 5 characters" or "Password is too short."
- **Real-World Use Cases**: Black-box testing ensures that the form behaves as expected in real-world use cases and catches issues like improper error messages, incorrect form submissions, or failures to handle certain types of input.

Black-box Testing Example: more effective

Why White-Box Testing May Be Less Effective:

- **Focus on Code Paths, Not User Experience**: White-box testing in this scenario would focus on the internal validation logic—checking the functions that verify the username length, password length, and special character restrictions.
- **Potential to Miss UI Issues**: It may miss issues related to the user experience, such as how error messages are displayed or whether the form reloads when it shouldn't.
- **Less Emphasis on Usability**: White-box testing might ensure that the functions for validation work perfectly according to the code, but it could overlook how well the user interface guides the user through input errors or displays feedback.

Quizizz