Object Oriented Software Development

Week 10

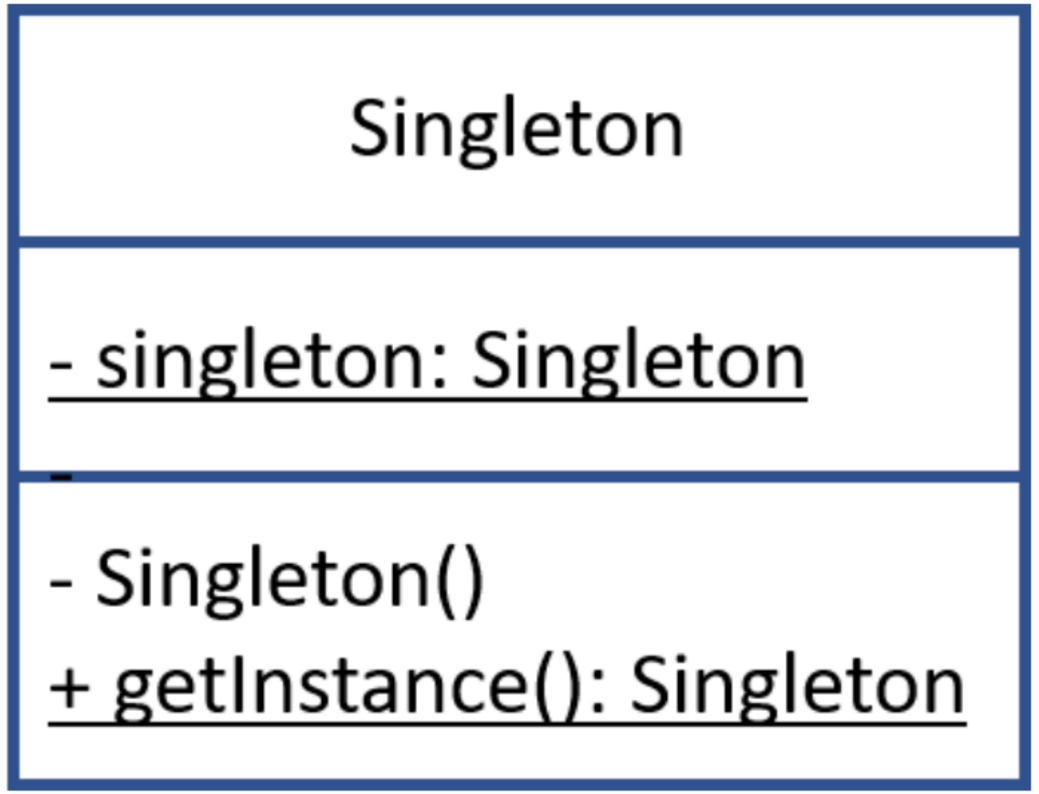
Design Patterns

This week, we are reviewing **design patterns**.

1. Design patterns give you a recipe for solving common problems.
2. They are designed for a specific situation (the **motivation**), come with a specific **structure** (typically a UML diagram), and have **consequences**.
3. The **singleton** design pattern is used to ensure there is a single global instance of an object. Like global variables in C, it is usually best to avoid this pattern.
4. The **factory method** design pattern allows more flexible control over which specific instances are created e.g. of an abstract parent class or interface.
5. The **template method** design pattern allows an implementation of an algorithm to vary in its details using **inheritance**.
6. The **strategy** design pattern allows an implementation of an algorithm to vary in its details using **delegation**.
7. The **observer** design pattern decouples class interactions by notifying an observer when a subject's state changes.

Singleton

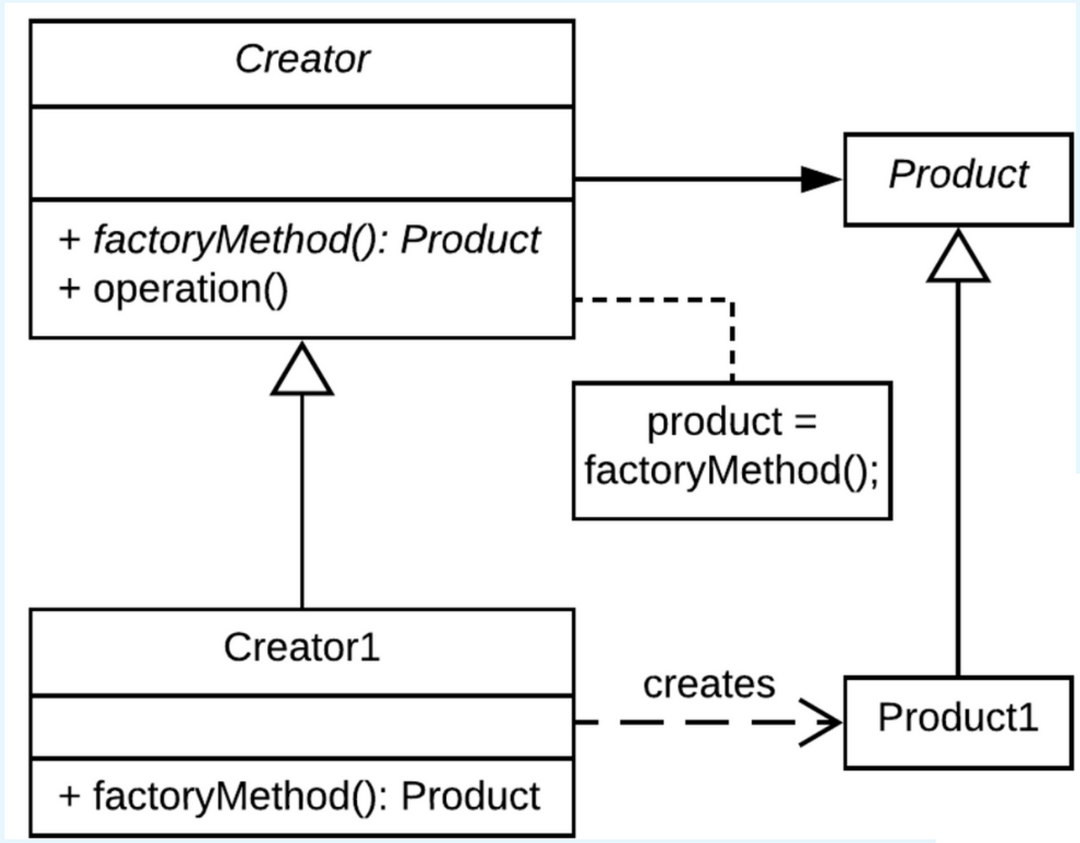
Ensure that a class has only one instance and provide a global point of access to it.

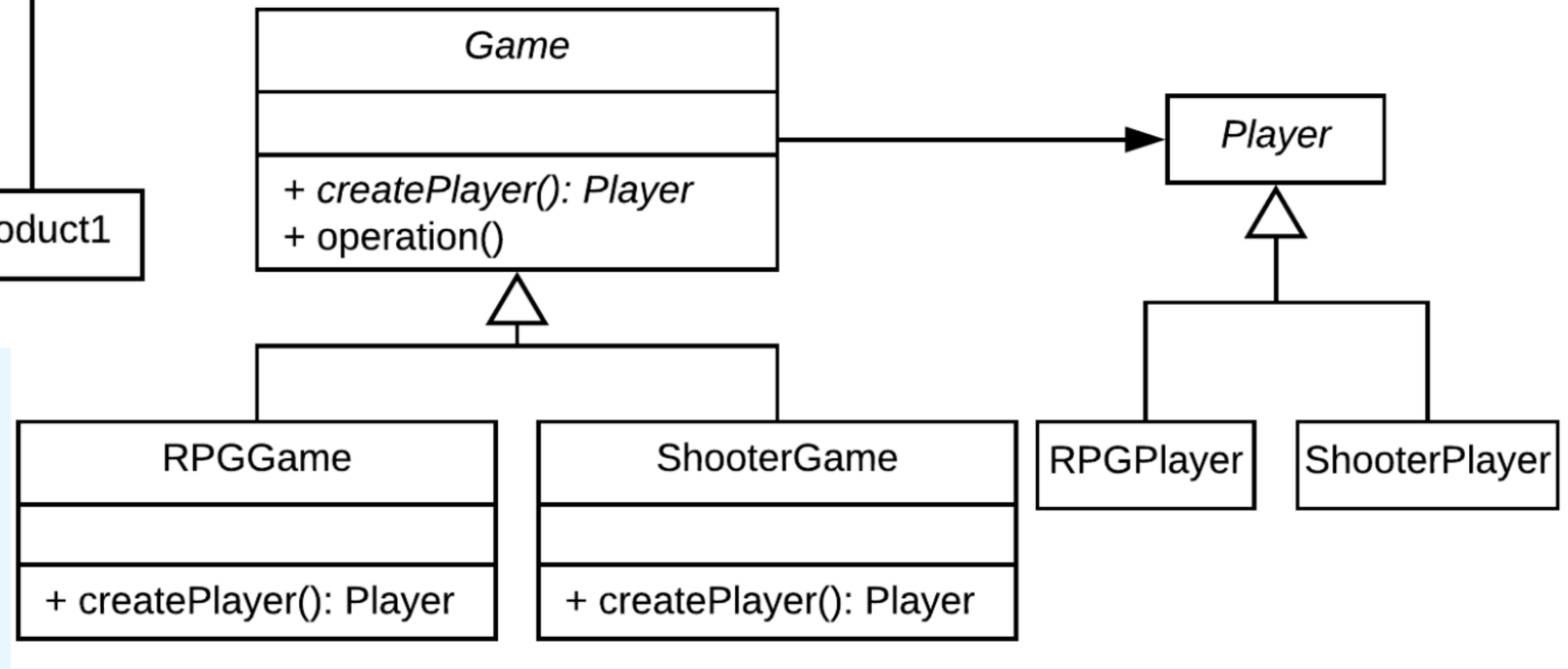


<Notice the variable is private.

<Notice the constructor is private.

Factory method





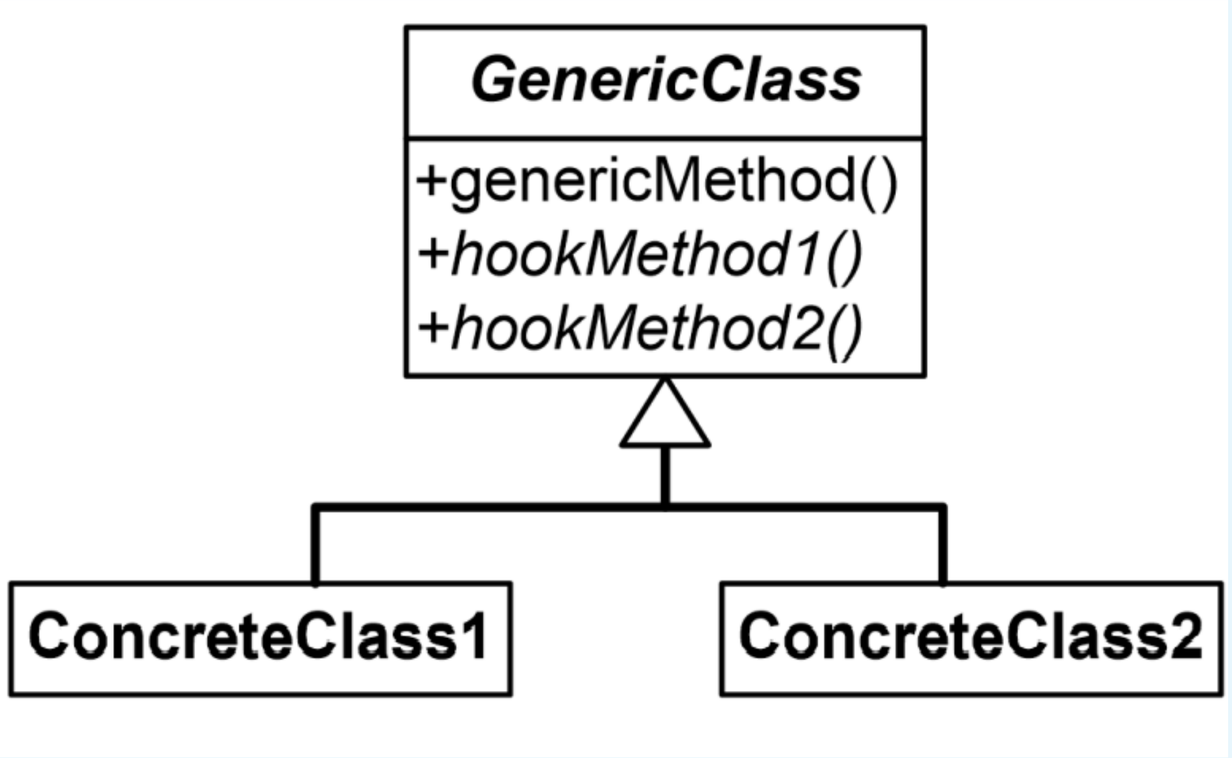
Template Method and *Strategy* are two design patterns that solve the problem of separating a generic algorithm from a detailed design.

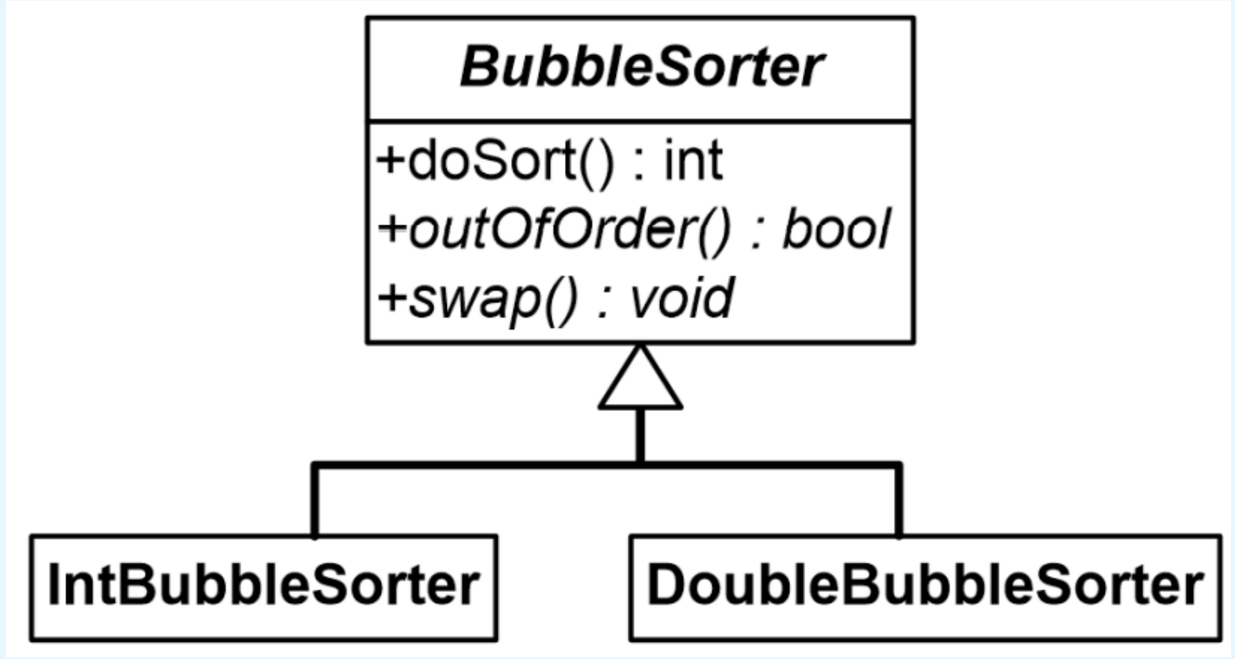
Template Method pattern uses *Inheritance*.

Strategy pattern uses *Delegation*.

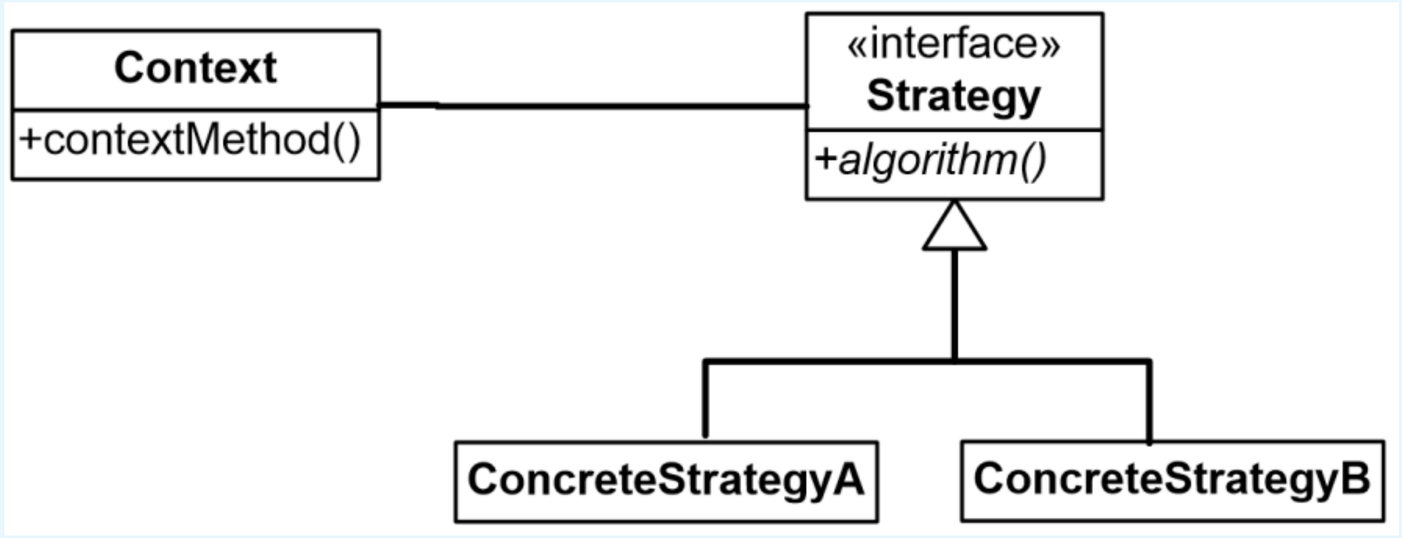
You've seen this in the course in the Collections.sort() method.

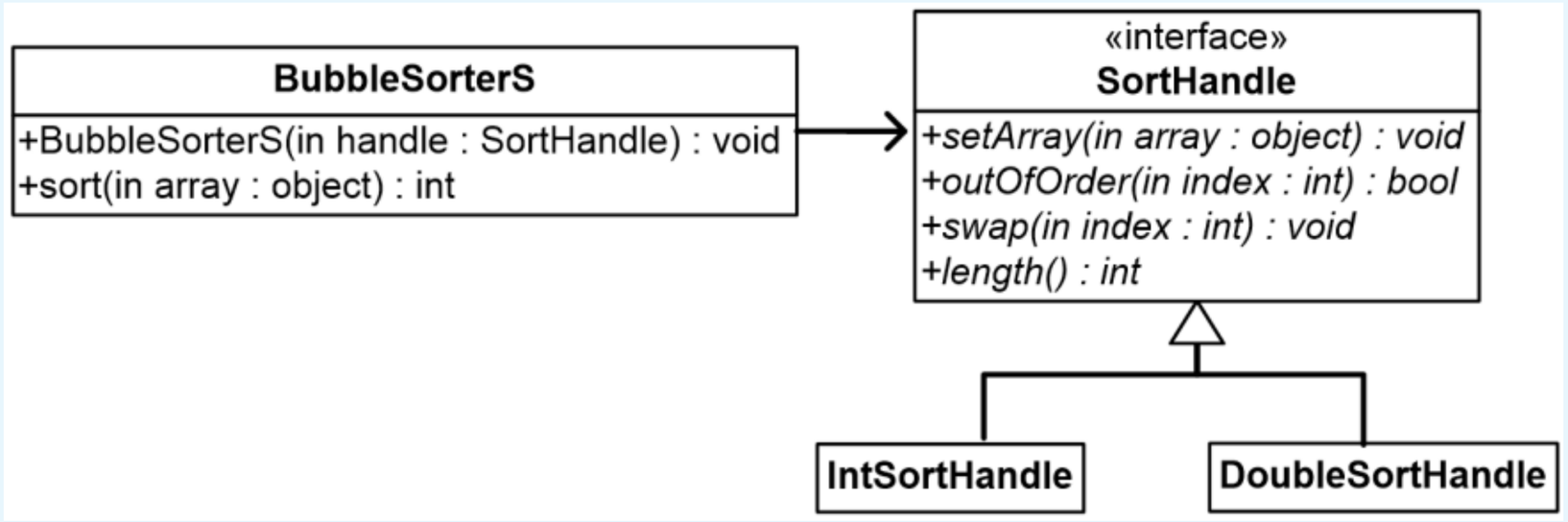
Template method



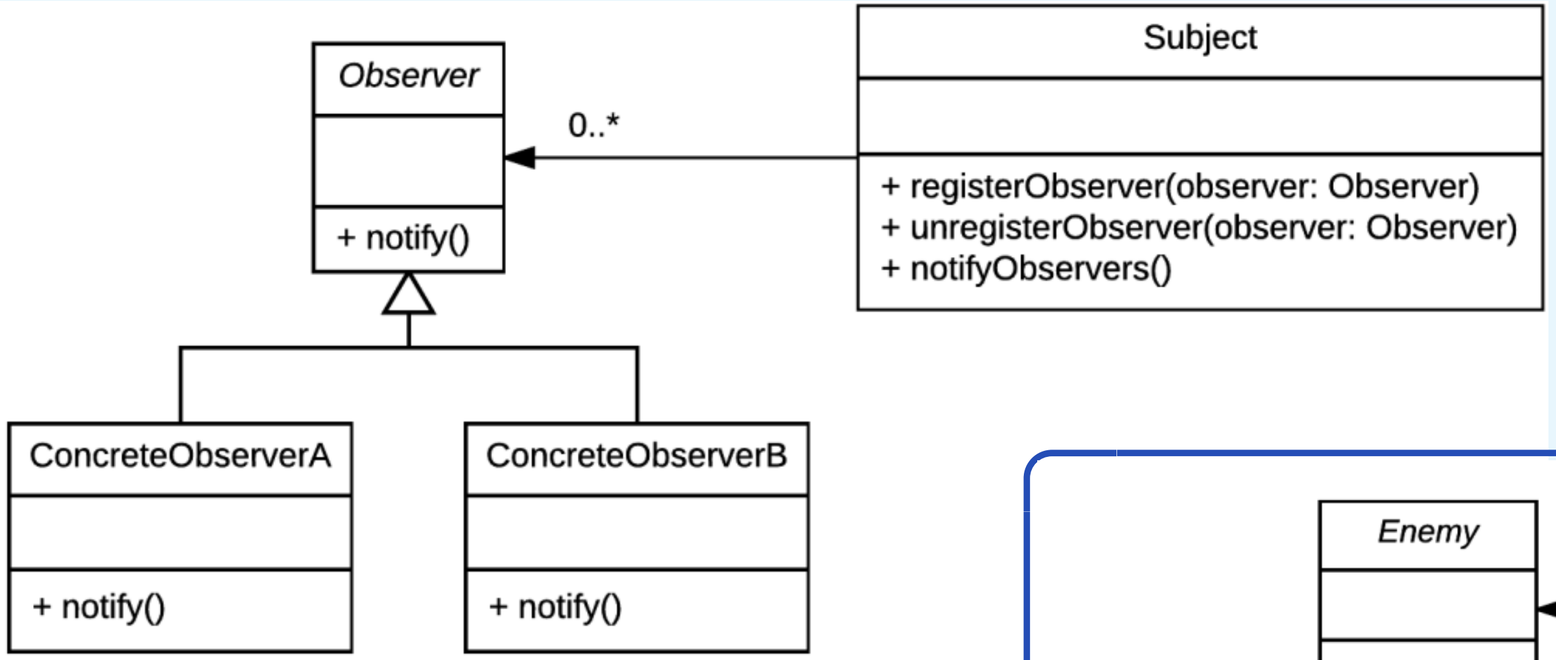


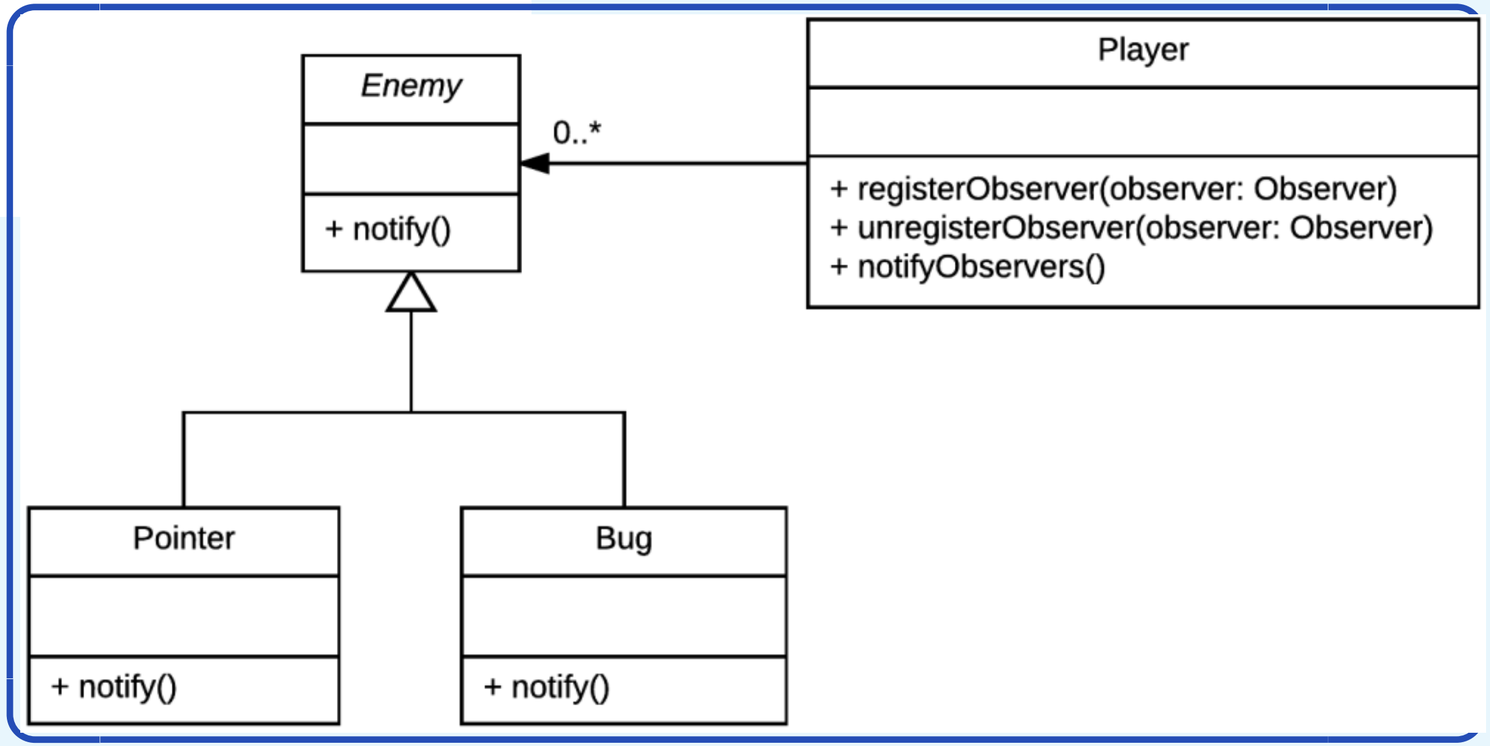
Strategy


Context
+contextMethod()
«interface»
Strategy
+algorithm()
ConcreteStrategyA
ConcreteStrategyB


BubbleSorterS
+BubbleSorterS(in handle : SortHandle) : void
+sort(in array : object) : int
«interface»
SortHandle
+setArray(in array : object) : void
+outOfOrder(in index : int) : bool
+swap(in index : int) : void
+length() : int
IntSortHandle
DoubleSortHandle

Observer


Observer
+ notify()
0..*
Subject
+ registerObserver(observer: Observer)
+ unregisterObserver(observer: Observer)
+ notifyObservers()
ConcreteObserverA
+ notify()
ConcreteObserverB
+ notify()


Enemy
+ notify()
0..*
Player
+ registerObserver(observer: Observer)
+ unregisterObserver(observer: Observer)
+ notifyObservers()
Pointer
+ notify()
Bug
+ notify()

What to know

For all design patterns mentioned here:

- The problem they solve
- How they solve it
- An advantage
- A disadvantage

Pairs or individually

Share a single device