Fundamental OOD principles

SOLID: five basic principles of object-oriented (Developed by Robert C. Martin, affectionately known as “Uncle Bob”.)

- **S**ingle responsibility principle
- **O**pen/closed principle
- **L**iskov substitution principle
- **I**nterface segregation principle
- **D**ependency inversion principle

Single Responsibility Principle

Every class should have a single responsibility.

Another way to view this is that a class should only have one reason to change.

But who causes the change?

“*This principle is about people.* … When you write a software module, you want to make sure that when changes are requested, those changes can only originate from a single person, or rather, a single tightly coupled group of people representing a single narrowly defined business function. You want to isolate your modules from the complexities of the organization as a whole, and design your systems such that each module is responsible (responds to) the needs of just that one business function.” [Uncle Bob, The Single Responsibility Principle]

Open/Closed Principle (simplified)

Software entities (classes, modules, functions, etc.) should be **open for extension**, but **closed for modification**.

Add new features not by modifying the original class, but rather by extending it and adding new behaviours, or by adding *plugin* capabilities.

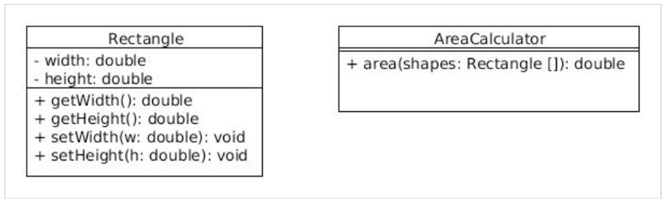
“I’ve heard it said that the OCP is wrong, unworkable, impractical, and not for real programmers with real work to do. The rise of plugin architectures makes it plain that these views are utter nonsense. On the contrary, a strong plugin architecture is likely to be the most important aspect of future software systems.” [Uncle Bob, The Open Closed Principle]

Open/Closed Principle (simplified)

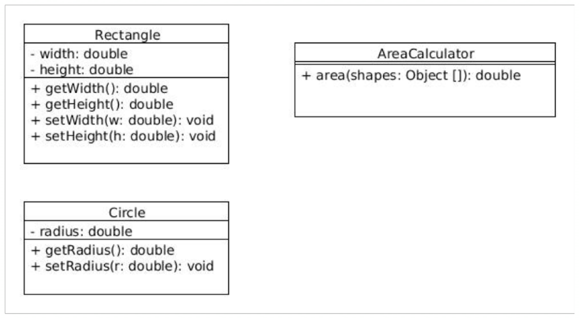
An example, using inheritance:

`area` calculates the area of all `Rectangle`s in the input.

What if we need to add more shapes?

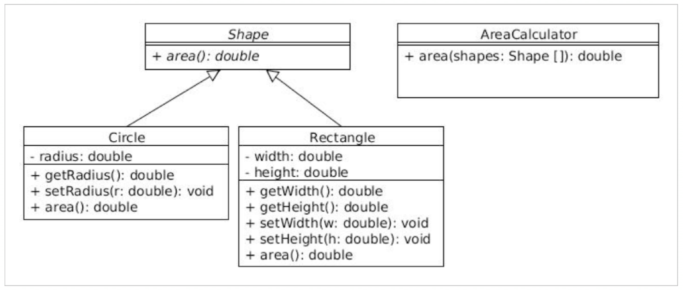


Open/Closed Principle (simplified)



What if we need to add even more shapes?

Open/Closed Principle (simplified)



With this design, we can add any number of shapes (open for extension) and we don't need to re-write the `AreaCalculator` class (closed for modification).

Liskov Substitution Principle (simplified)

If S is a subtype of T, then objects of type S may be substituted for objects of type T, without altering any of the desired properties of the program.

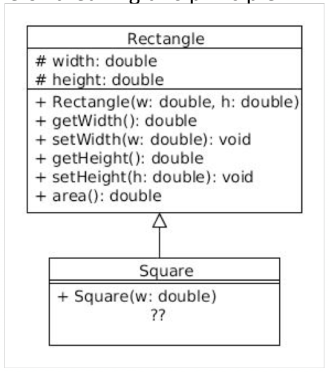
"S is a subtype of T"?

In Java, S is a *child class* of T, or S *implements* interface T.

For example, if C is a child class of P, then we should be able to substitute C for P in our code without breaking it.

Liskov Substitution Principle (simplified)

A classic example of breaking this principle:



Liskov Substitution Principle (simplified)

In OO programming and design, unlike in math, it is not the case that a Square is a Rectangle!

This is because a Rectangle has *more* behaviours than a Square, not less.

The LSP is related to the Open/Closed principle: the subclasses should only extend (add behaviours), not modify or remove them.

Interface Segregation Principle

Here, *interface* means the public methods in a class. (In Java, these are often specified using a Java `interface`, which you'll learn about soon.)

Context: a class that provides a service for other "client" programmers usually requires that the clients write code that has a particular set of features. The service provider says "your code needs to have this interface".

No client should be forced to implement irrelevant methods of an interface. Better to have lots of small, specific interfaces than fewer larger ones: easier to extend and modify the design.

(Uh oh: "The interface keyword is harmful." [Uncle Bob, 'Interface' Considered Harmful])

Dependency inversion principle

When building a complex system, programmers are often tempted to define "low-level" classes first and then build "higher-level" classes that use the low-level classes directly.

But this approach is not flexible! What if we need to replace a low-level class? The logic in the high-level class will need to be replaced — an indication of high coupling.

To avoid such problems, we introduce an *abstraction layer* between low-level classes and high-level classes.

Dependency inversion principle

Goal:

- You want to decouple your system so that you can change individual pieces without having to change anything more than the individual piece.

Two aspects to the dependency inversion principle:

- High-level modules should not depend on low-level modules. Both should depend on abstractions.

- Abstractions should not depend upon details. Details should depend upon abstractions.

Example from Dependency Inversion Principle on OODesign

Example: you have a large system, and part of it has Managers manage Workers. Let's say that the company is restructuring and introducing new kinds of workers, and wants the code updated to reflect this.

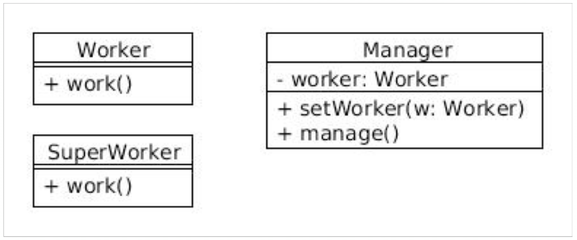
Your code current has a Manager class and a Worker class, and the Manager class has several methods that have Worker parameters.

Now there's a new kind of worker called SuperWorker, and their behaviour and features are separate from regular Workers.

Oh dear …

Example from Dependency Inversion Principle on OODesign

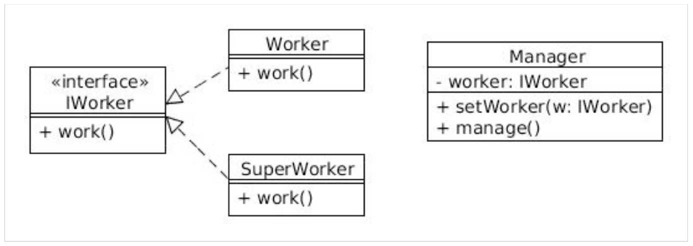
To make Manager work with SuperWorker, we would need to rewrite the code in Manager.



Solution: create an IWorker interface and have Manager use it.

Example from Dependency Inversion Principle on OODesign

In this design, Manager does not know anything about Worker, nor about SuperWorker. It can work with any IWorker, the code in Manager does not need rewriting.


«interface»
IWorker
+ work()
Worker
+ work()
SuperWorker
+ work()
Manager
- worker: IWorker
+ setWorker(w: IWorker)
+ manage()