

# Object-oriented Design Principles

CSC301 Fall 2014



# OO Design Principles

First five basic principles of object-oriented design.

# SOLID

# SOLID

- S** Single responsibility principle
- O** Open/closed principle
- L** Liskov substitution principle
- I** Interface segregation principle
- D** Dependency inversion principle

# S: Single Responsibility Principle

- every class should have a single responsibility
- responsibility should be entirely encapsulated by the class
- all class services should be aligned with that responsibility

Why?

- makes the class more robust
- makes the class more reusable

Note the terminology clash here: in CRC cards “responsibility” is what we call “service” here.

# O: 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.
- The derived class may or may not have the same interface as the original class.

# O: Open/Closed Principle (simplified)

Example:

area calculates the area of all Rectangles in the input.

What if we need to add more shapes?

Rectangle
- width: double - height: double
+ getWidth(): double + getHeight(): double + setWidth(w: double): void + setHeight(h: double): void

AreaCalculator
+ area(shapes: Rectangle []): double

# O: Open/Closed Principle (simplified)

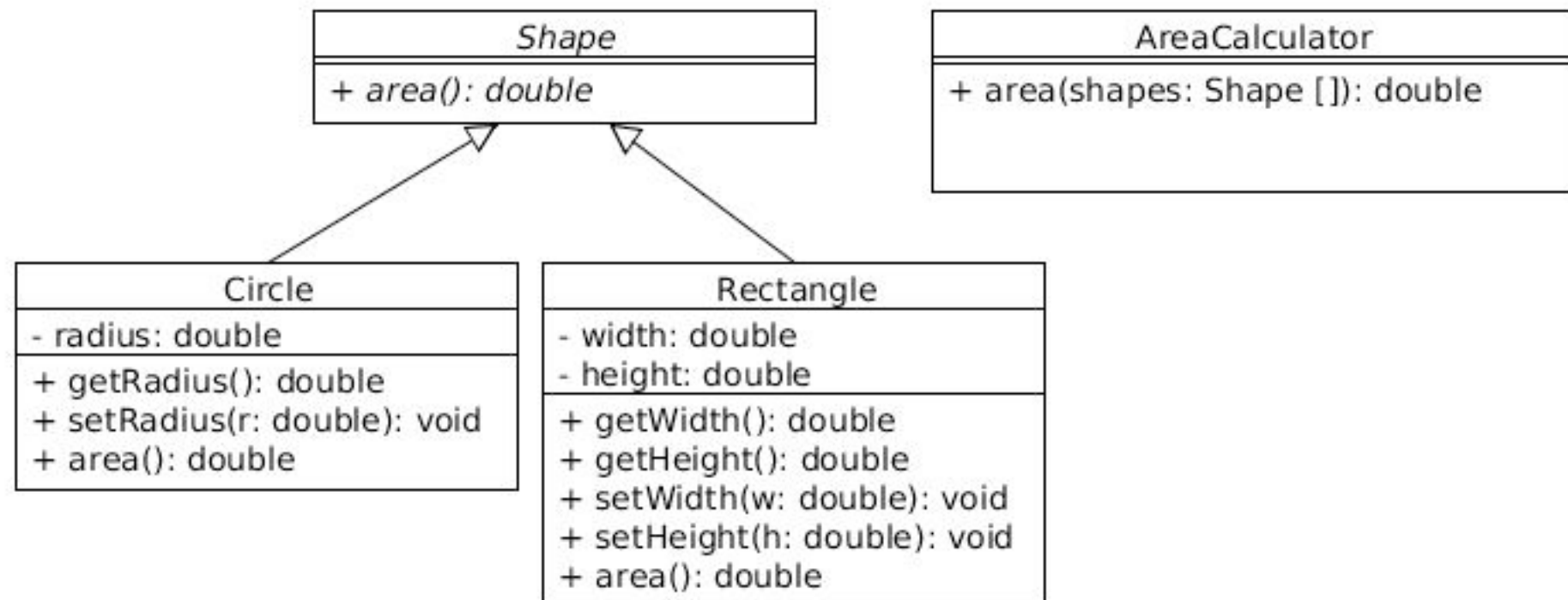
Rectangle
- width: double - height: double
+ getWidth(): double + getHeight(): double + setWidth(w: double): void + setHeight(h: double): void

AreaCalculator
+ area(shapes: Object []): double

Circle
- radius: double
+ getRadius(): double + setRadius(r: double): void

What if we need to add even more shapes?

# O: 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).



# L: Liskov Substitution Principle (simplified)

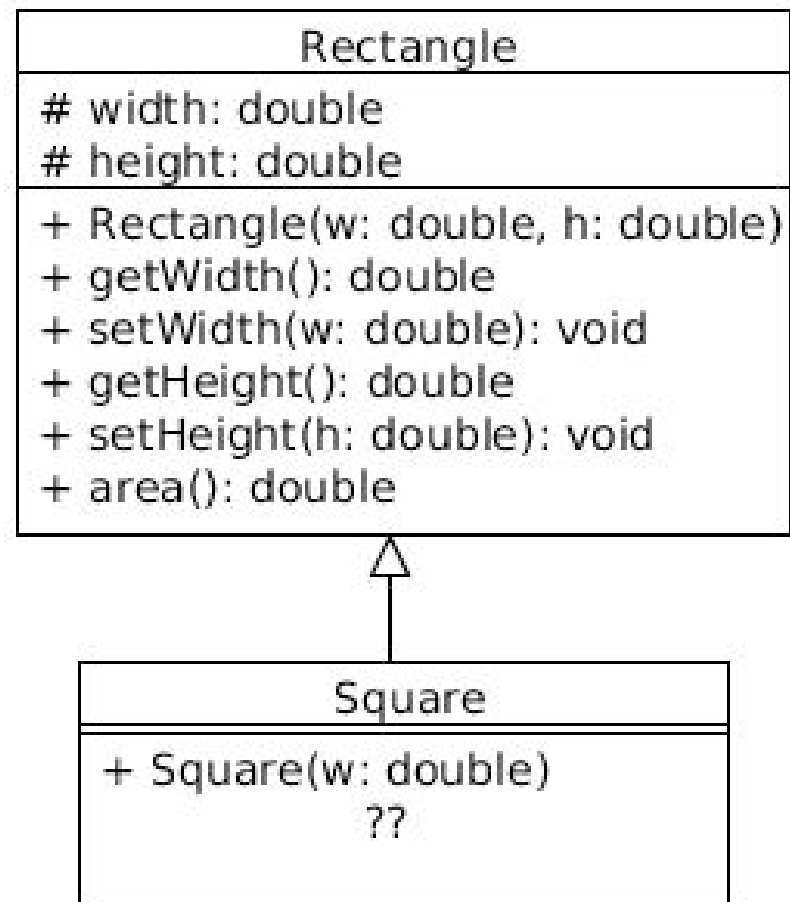
- If  $S$  is a subtype of  $T$ , then objects of type  $S$  may be substituted for objects of type  $T$ , without altering any of the desired properties of the program.
- “ $S$  is a subtype of  $T$ ”?

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.

# L: Liskov Substitution Principle (simplified)

A classic example of breaking this principle:



# L: 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/Close principle: the sub classes should only extend (add behaviours), not modify or remove them.

# I: Interface Segregation Principle

- No client should be forced to depend on methods it doesn't use.
- Better to have lots of small, specific interfaces than fewer larger ones.
- Easier to extend and modify the design.

# D: Dependency inversion principle

- When building a complex system, we may be tempted to define the “low-level” classes first and then build the “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.
- To avoid such problems, we can introduce an abstraction layer between low-level classes and high-level classes.

# D: Dependency inversion principle

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

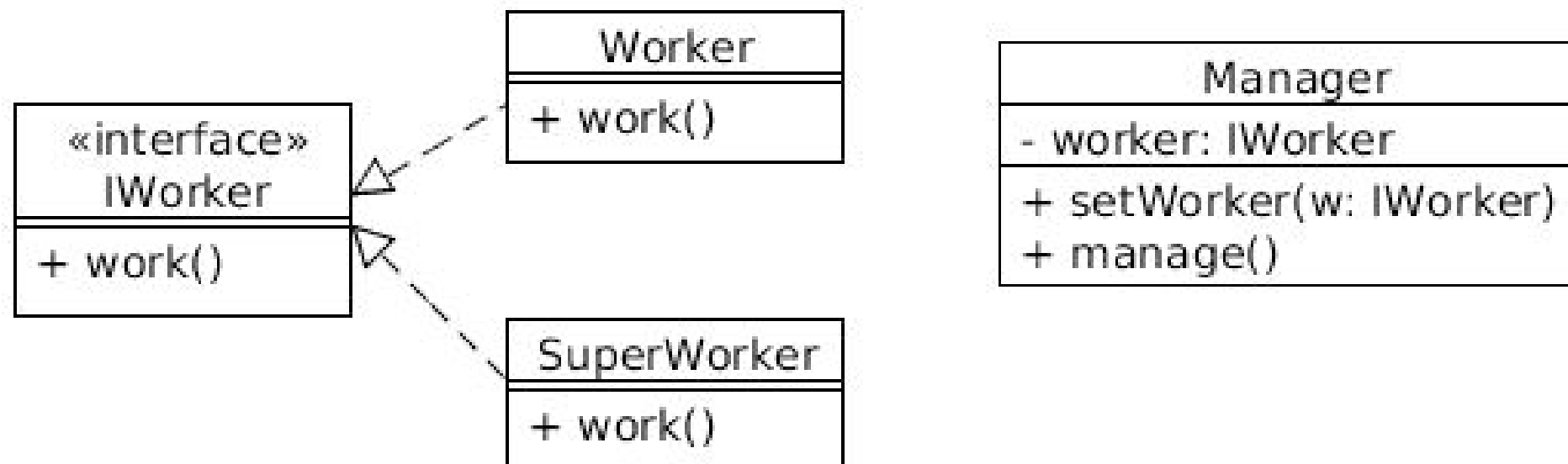
Worker
+ work()

SuperWorker
+ work()

Manager
- worker: Worker
+ setWorker(w: Worker)
+ manage()

# D: Dependency inversion principle

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



# SOLID

Many Design Patterns follow the SOLID principles of object-oriented Design.

Can you identify any of these principles in any of the design patterns we saw?



# D: Dependency inversion principle

- Two aspects:
  - 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.
- When building a complex system, we may be tempted to define the “low-level” classes first and then build the “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.
- To avoid such problems, we can introduce an abstraction layer between low-level classes and high-level classes.