SOLID

Rajeev gupta
Java trainer
Rgupta.mtech@gmail.com

SOLID

S Single responsibility principle

O Open/closed principle

L Liskov substitution principle

I Interface segregation principle

D Dependency inversion principle

S: Single Responsibility

- every class should have a single responsibility
- responsibility should be entirely encapsulated by the class
- all class services should be 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.

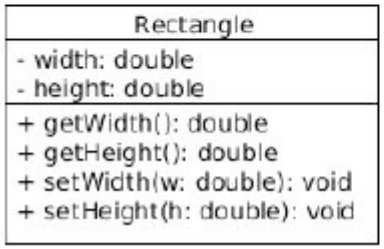
Open Closed Principle

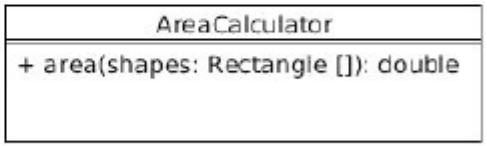
- Software entities (classes, modules, functions, etc.) should be **open for extension**, but **closed for modification**.
- Add new features not by modifying the original class, but rather by extending it and adding new behaviours.
- The derived class may or may not have the same interface as the original class.

Example:

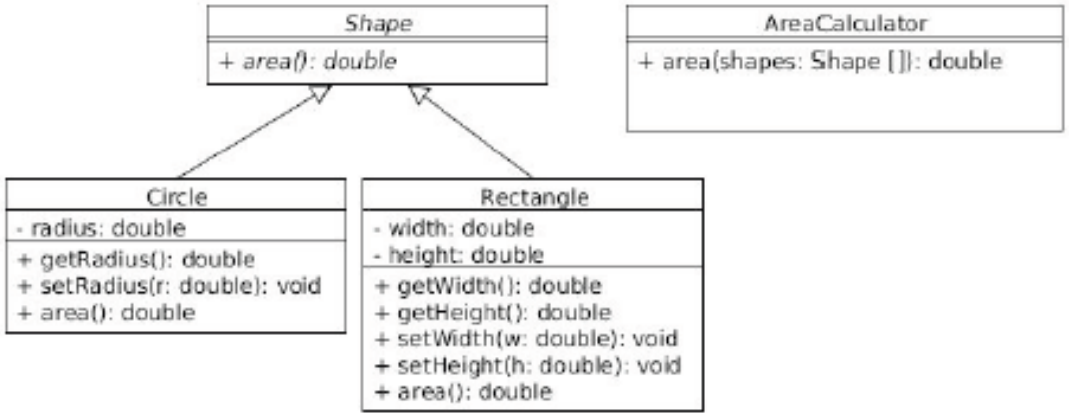
`area` calculates the area of all `Rectangle`s in the input.

What if we need to add more shapes?





Open Closed Principle



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).

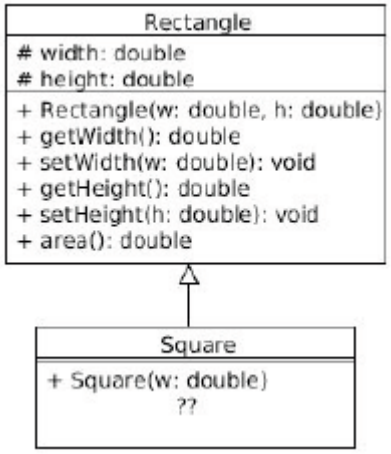
Liskov Substitution Principle

- 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

- 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.



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.

Dependency Inversion Principle

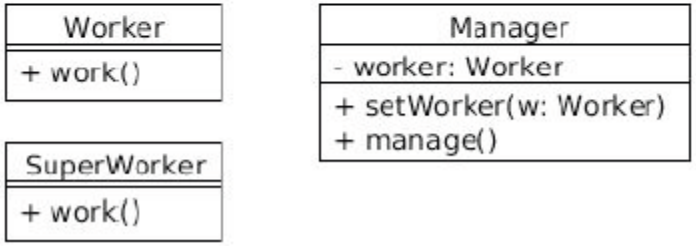
- When building a complex system, we may be tempted to define the "low-level" classes first and then build the "higher-level" classes that use the low-level classes directly.
- But this approach is not flexible! What if we need to replace a low-level class? The logic in the high-level class will need to be replaced.
- To avoid such problems, we can introduce an abstraction layer between low-level classes and high-level classes.

Dependency Inversion Principle

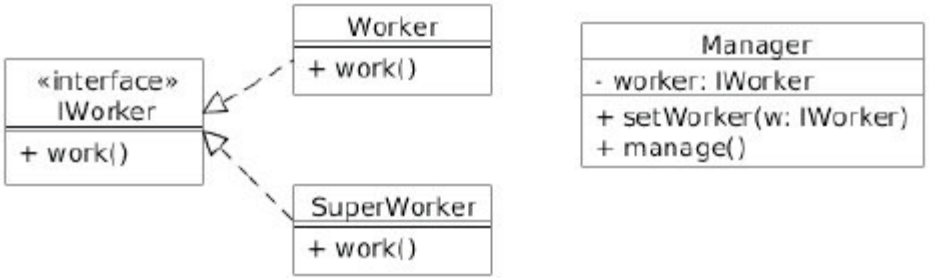
- Two aspects:
  - High-level modules should not depend on low-level modules. Both should depend on abstractions.
  - Abstractions should not depend upon details. Details should depend upon abstractions.
- When building a complex system, we may be tempted to define the "low-level" classes first and then 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.

Dependency Inversion Principle

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



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.



Reference

- http://cafe.elharo.com/programming/a-square-is-not-a-rectangle/
- http://www.cdf.toronto.edu/~csc207h/fall/stg/lectures/eve/w10/
- http://zeroturnaround.com/rebellabs/object-oriented-design-principles-and-the-5-ways-of-creating-solid-applications/