Principles of Object-Oriented Design

M Saeed Siddik
IIT, University of Dhaka

SOLID Principle

Single-responsibility principle

class should only have a single responsibility, that is, only changes to one part of the software's specification should be able to affect the specification of the class.

Open–closed principle

"Software entities ... should be open for extension, but closed for modification."

Liskov substitution principle

"Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program." See also design by contract.

Interface segregation principle

"Many client-specific interfaces are better than one general-purpose interface."

Dependency inversion principle

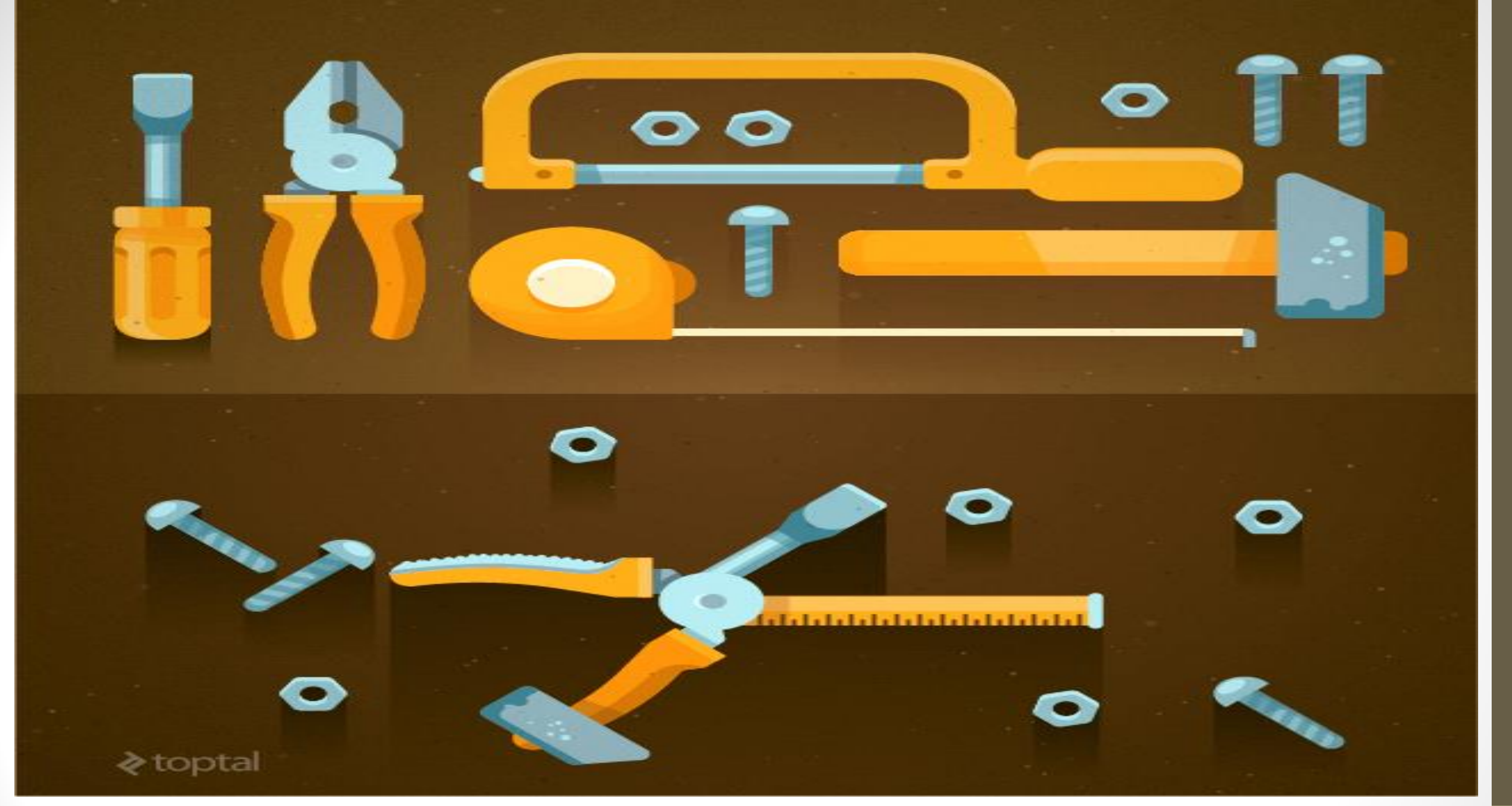
One should "depend upon abstractions, [not] concretions

Single-responsibility principle

- A class should have one, and only one, reason to change.
- Every module or class should have responsibility over a single part of the functionality
- Responsibility should be entirely encapsulated by the class, module or function

Single Responsibility Principle

Just because you *can* doesn't mean you *should*.



Single-responsibility principle

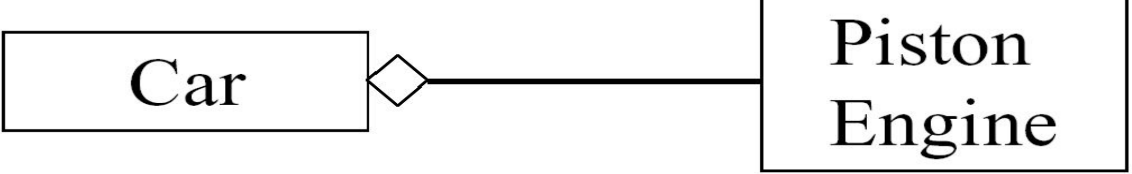
Open-Closed Principle (OCP)

- "*Software Systems change during their life time*"
  - both better designs and poor designs have to face the changes;
  - good designs are stable

> *"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."* **B. Meyer**, 1988 / quoted by **R. Martin**, 1996

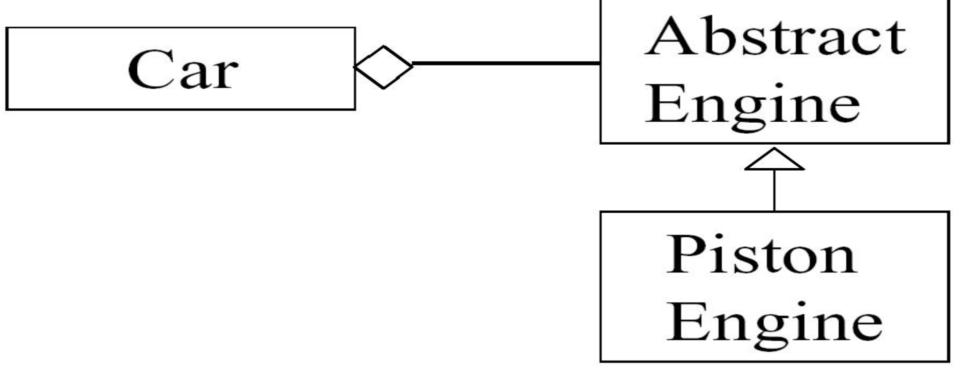
- Be open for extension
  - module's behavior can be extended
- Be closed for modification
  - source code for the module must not be changes
- *Modules should be written so they can be extended without requiring them to be modified*

Open the door ...



- How to make the Car run efficiently with a TurboEngine?
- Only by changing the Car!
  - ...in the given design

... But Keep It Closed!



- A class must not depend on a concrete class!
- It must depend on an abstract class ...
- ...using polymorphic dependencies (calls)

Strategic Closure

"*No significant program can be 100% closed* "

R.Martin, *“The Open-Closed Principle,” 1996*

- Closure not *complete* but *strategic*

- Use abstraction to gain explicit closure
  - provide class methods which can be dynamically invoked
    - to determine *general* policy decisions
    - e.g. draw Squares before Circles
  - design using abstract ancestor classes
- Use "Data-Driven" approach to achieve closure
  - place volatile policy decisions in a separate location
    - e.g. a file or a separate object
  - minimizes future change locations

Liskov Substitution Principle (LSP)

- The principle defines that objects of a superclass shall be replaceable with objects of its subclasses without breaking the application.
- That requires the objects of your subclasses to behave in the same way as the objects of your superclass.

> *Inheritance should ensure that any property proved about supertype objects also holds for subtype objects*
>
> **B. Liskov**, 1987

Liskov Substitution Principle (LSP)

- An overridden method of a subclass needs to accept the same input parameter values as the method of the superclass.
- That means you can implement less restrictive validation rules, but you are not allowed to enforce stricter ones in your subclass.
- Otherwise, any code that calls this method on an object of the superclass might cause an exception, if it gets called with an object of the subclass.
- Similar rules apply to the return value of the method. The return value of a method of the subclass needs to comply with the same rules as the return value of the method of the superclass.

LSP Example

```
BasicCoffeeMachine
-Map<CoffeeSelection, Configuration> configMap
-Map<CoffeeSelection, GroundCoffee> groundCoffee
-BrewingUnit brewingUnit

+BasicCoffeeMachine(Map<CoffeeSelection, GroundCoffee> coffee)
+CoffeeDrink brewCoffee(CoffeeSelection selection)
-CoffeeDrink brewFilterCoffee()
+void addCoffee(CoffeeSelection sel, GroundCoffee newCoffee)
```

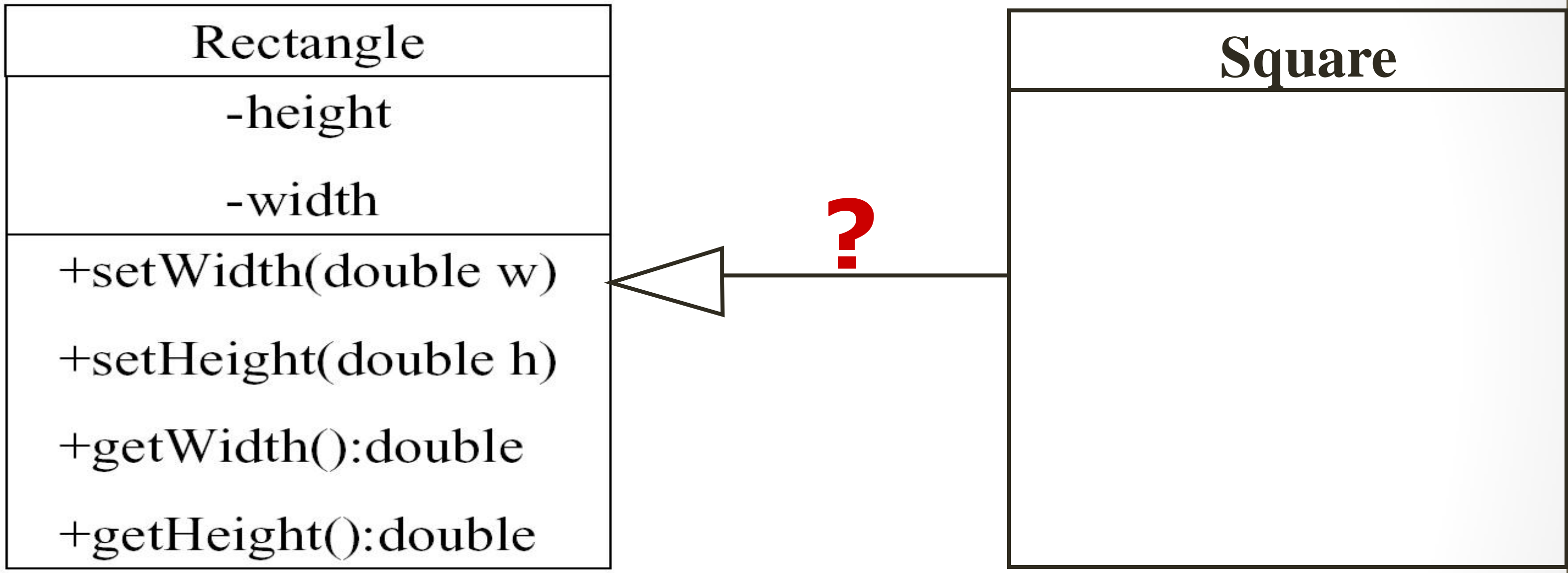
```
PremiumCoffeeMachine
-Map<CoffeeSelection, Configuration> configMap
-Map<CoffeeSelection, CoffeeBean> beans
-Grinder grinder
-BrewingUnit brewingUnit

+PremiumCoffeeMachine(Map<CoffeeSelection, CoffeeBean> beans)
+CoffeeDrink brewCoffee(CoffeeSelection selection)
-CoffeeDrink brewEspresso()
-CoffeeDrink brewFilterCoffee()
+void addCoffee(CoffeeSelection sel, CoffeeBean newBeans)
```

*Source: https://stackify.com/solid-design-liskov-substitution-principle/

LSP Example

- The *BasicCoffeeMachine* can only brew filter coffee. So, the *brewCoffee* method checks if the provided *CoffeeSelection* value is equal to *FILTER_COFFEE* before it calls the private *brewFilterCoffee* method to create and return a *CoffeeDrink* object.
- The premium coffee machine has an integrated grinder, and the internal implementation of the *brewCoffee* method is a little more complex. But you don't see that from the outside. The method signature is identical to the one of the *BasicCoffeeMachine* class.

Square IS-A Rectangle?



- Should I inherit Square from Rectangle?

The Answer is ...

- Override **`setHeight`** and **`setWidth`**
  - duplicated code...
  - static binding (in C++)
    - **`void f(Rectangle& r) { r.setHeight(5); }`**
    - change base class to set methods **`virtual`**

- The real problem

```
void g(Rectangle& r) {
  r.setWidth(5); r.setHeight(4);
  // How large is the area?
}
```

  - 20! ... Are you sure? ;-)

- IS-A relationship must refer to the **behavior** of the class!

Interface segregation principle

- The **interface-segregation principle** (**ISP**) states that no client should be forced to depend on methods it does not use
- ISP splits interfaces that are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them.
- ISP is intended to keep a system decoupled and thus easier to refactor, change, and redeploy. ISP is similar to the High Cohesion Principle.

Dependency Inversion Principle

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

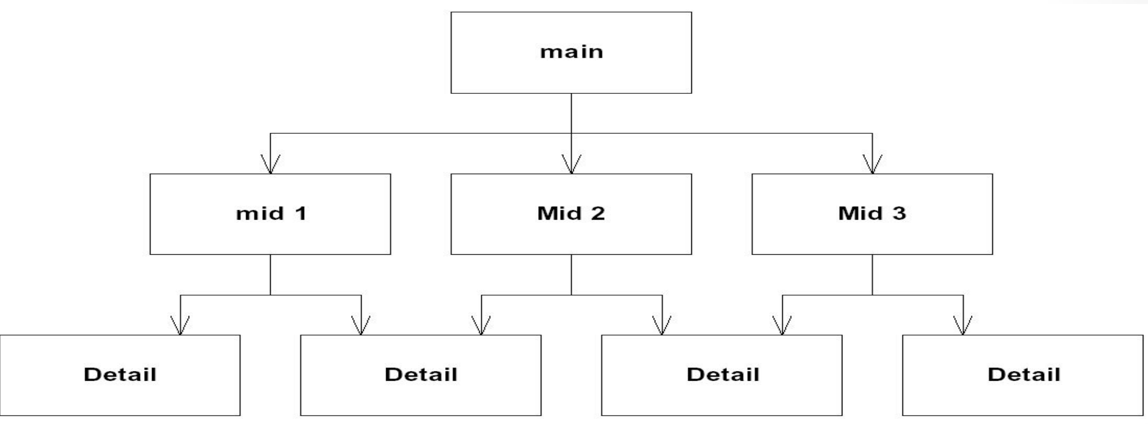
II. Abstractions should not depend on details.
   Details should depend on abstractions

R. Martin, 1996

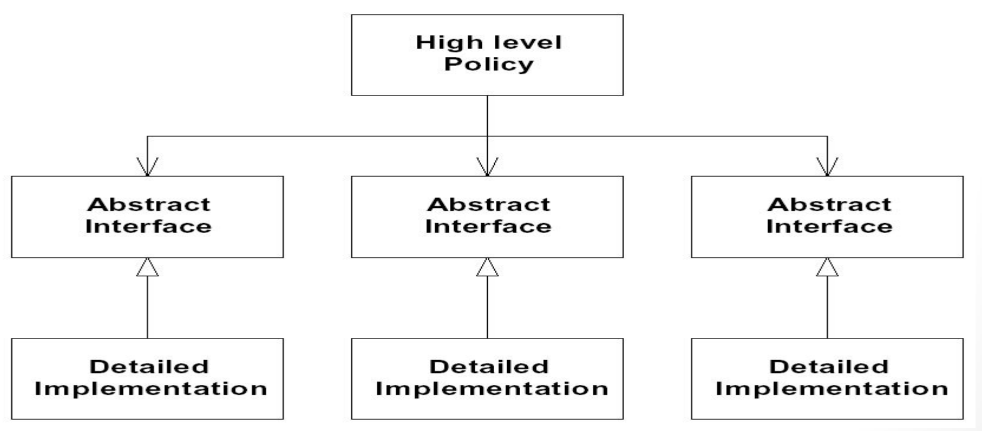
- OCP states the goal; DIP states the mechanism
- A base class in an inheritance hierarchy should not know any of its subclasses
- Modules with detailed implementations are not depended upon, but depend themselves upon abstractions

Procedural vs. OO Architecture

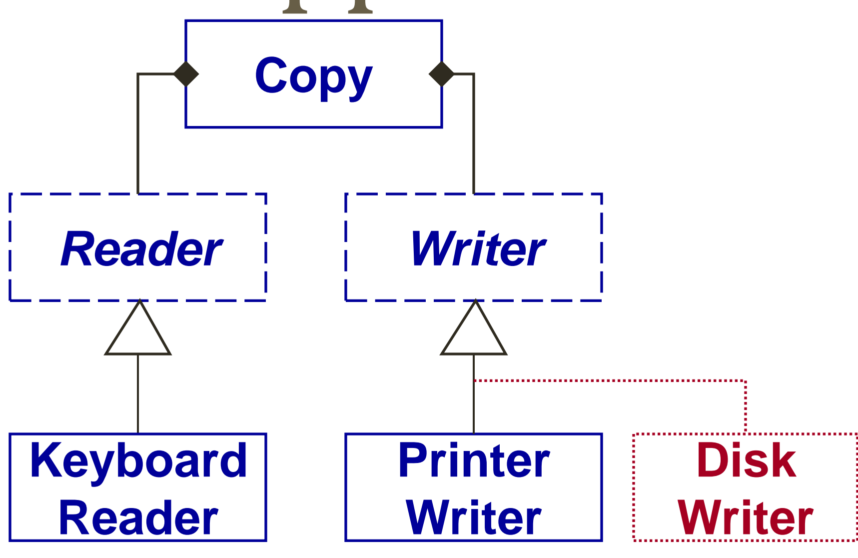
Procedural Architecture



Object-Oriented Architecture



DIP Applied on Example

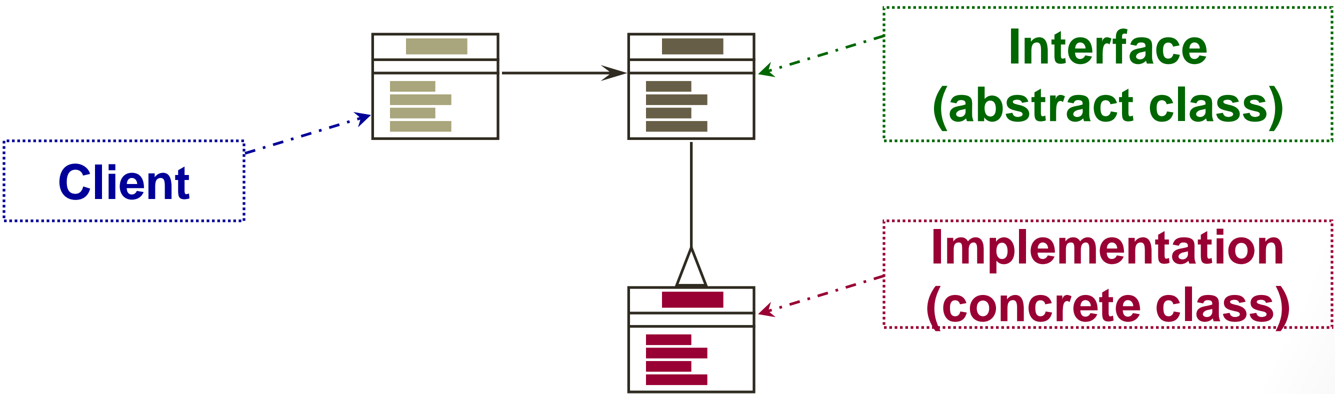



```
class Reader {
  public:
    virtual int read()=0;
};

class Writer {
  public:
    virtual void write(int)=0;
};

void Copy(Reader& r, Writer& w){
   int c;
   while((c = r.read()) != EOF)
     w.write(c);
}
```

DIP Related Heuristic

Design to an interface, not an implementation!

- Use inheritance to avoid direct bindings to classes:



Reference

- https://stackify.com/interface-segregation-principle/
- https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design
- https://en.wikipedia.org/wiki/SOLID
- https://www.geeksforgeeks.org/solid-principle-in-programming-understand-with-real-life-examples/

End of SOLID Principle