SOLID Principles

SOLID Principles

1. Single responsibility principle
2. Open close principle
3. Liskov’s substitution principle
4. Interface segregation principle
5. Dependency inversion principle

Single Responsibility Principle

A module (class or function) should focus on a single entity or task

Violations

```
searchAndSort(...)

class StudentAndTA {...}

class Emp { string street[N]; string city[N]; … }
```

Open Close Principle

A software entity (class, function, etc) should be open for extension but closed for modification

The programmer should be able to add new features without changing the existing code

We can replace a switch statement with polymorphism

```
void foo(void* shape, int type) {
    switch(type) {
        case ‘C’:
            Circle* cir = (Circle*) shape;
            cir->draw();
            break;
        case ‘R’:
            Rect* rec = (Circle*) shape;
            rec->draw();
            break;
        …
    }
}
```

```
void foo(Shape* shape) {

    shape->draw();

}
```

Liskov’s Substitution Principle

We should be able to replace a super-type reference with a sub-type reference without affecting program’s correctness

Interface Segregation Principle

Many small specific interfaces are better than a single large interface

```
class Circle {
    … scale(...);
    … rotate(...);
    … shear(...);

    … circum(...);
    … area(...);
    … dia(...);
}
```

```
class Parent1 {
    … scale(...) = 0;
    … rotate(...) = 0;
    … shear(...) = 0;
}
```

```
class Parent2 {
    … circum(...) = 0;
    … area(...) = 0;
    … dia(...) = 0;
}
```

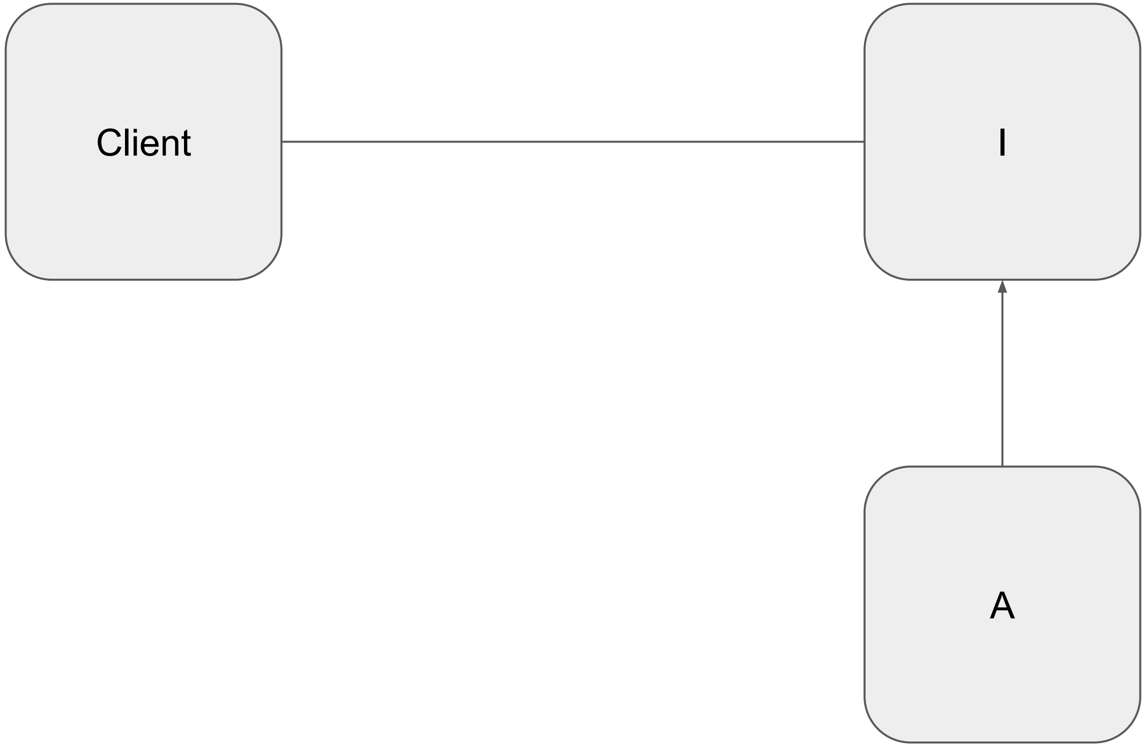
```
class Circle : public Parent1, public Parent2 {
    … scale(...);
    … rotate(...);
    … shear(...);

    … circum(...);
    … area(...);
    … dia(...);
}
```

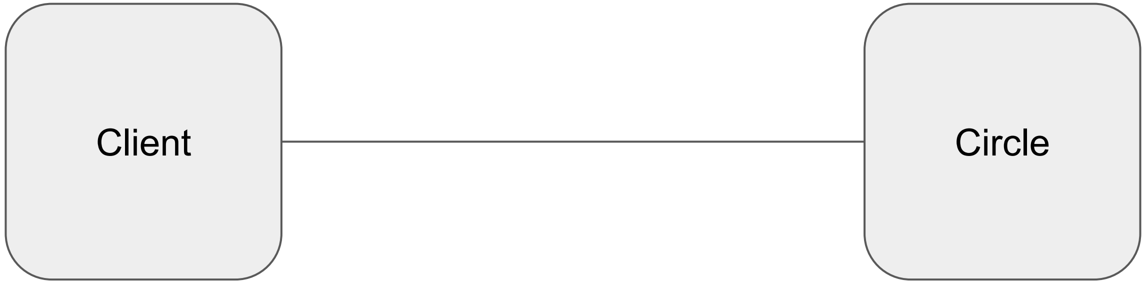
```
class Client1 {
    Circle* cir;
    void foo() {
        cir->scale(25);
    }
}

class Client2 {
    Circle* cir;
    void bar() {
        cout << cir->area();
    }
}
```

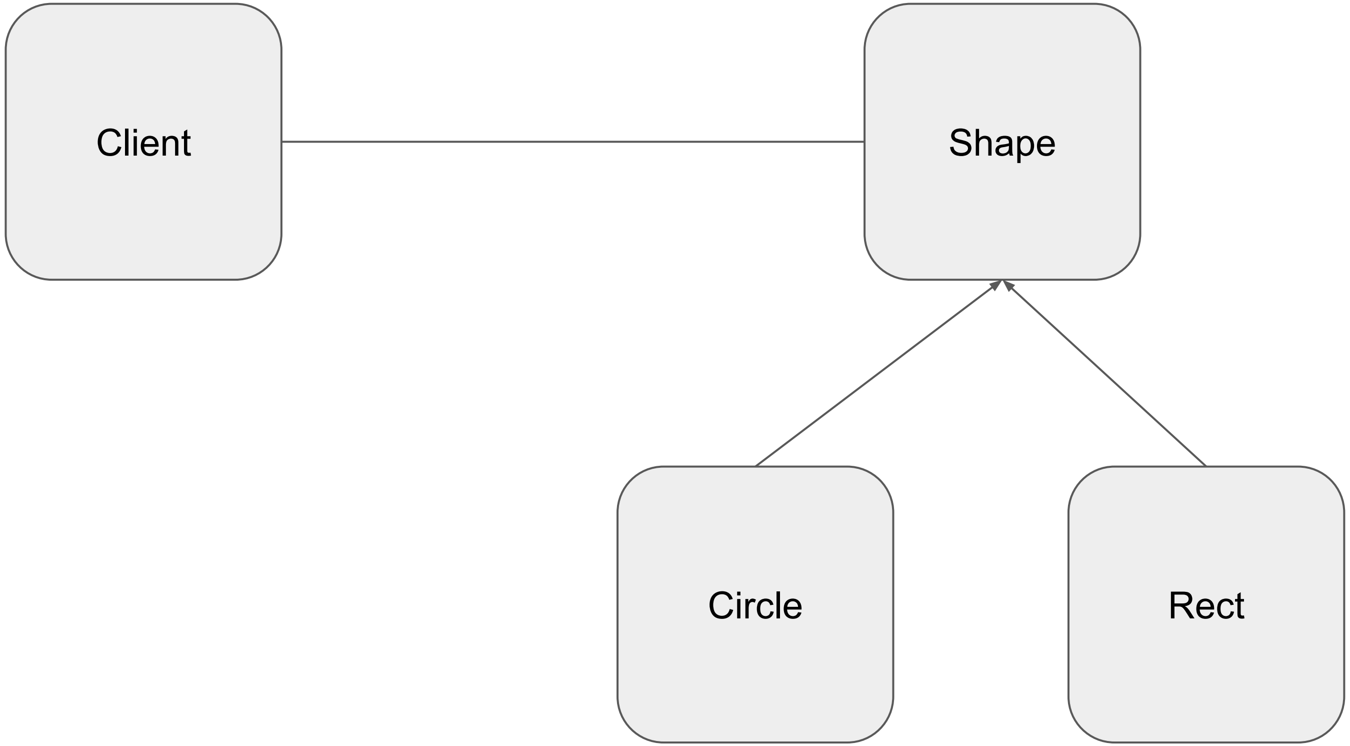
```
class Client1 {
    Parent1* cir = new Circle(...);
    void foo() {
        cir->scale(25);
        cout << cir->area();    // error
    }
}

class Client2 {
    Parent2* cir = new Circle(...);
    void bar() {
        cout << cir->area();
    }
}
```

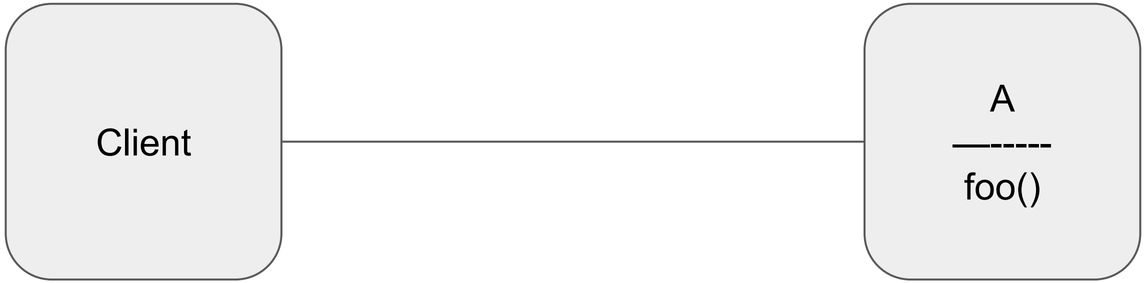
Dependency Inversion Principle

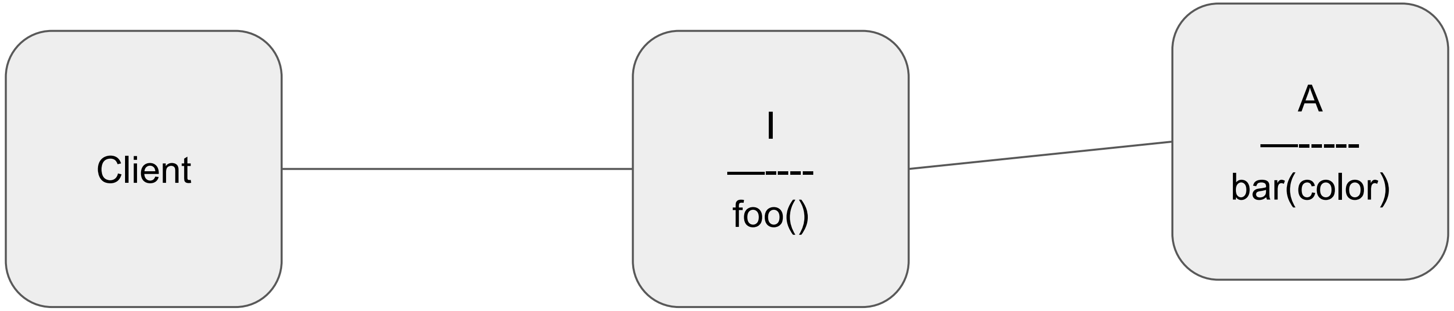
If a module is expected to change then do not interact with it directly

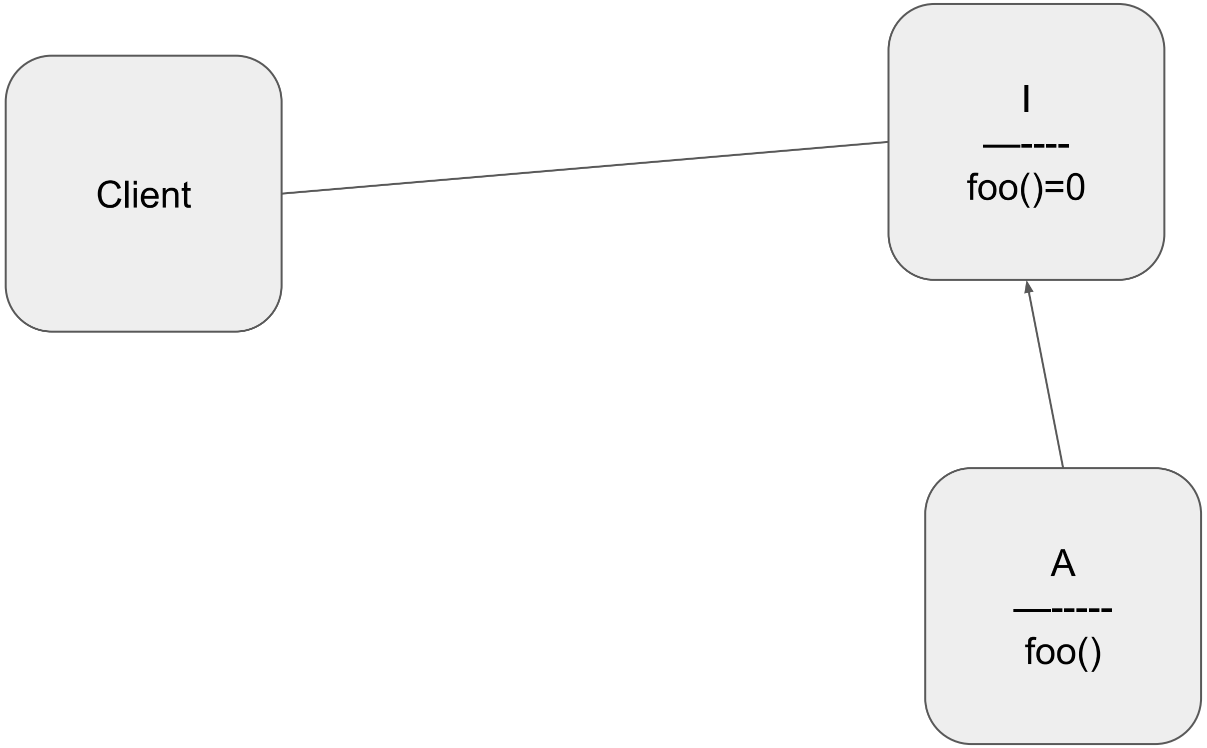

Client
I
A

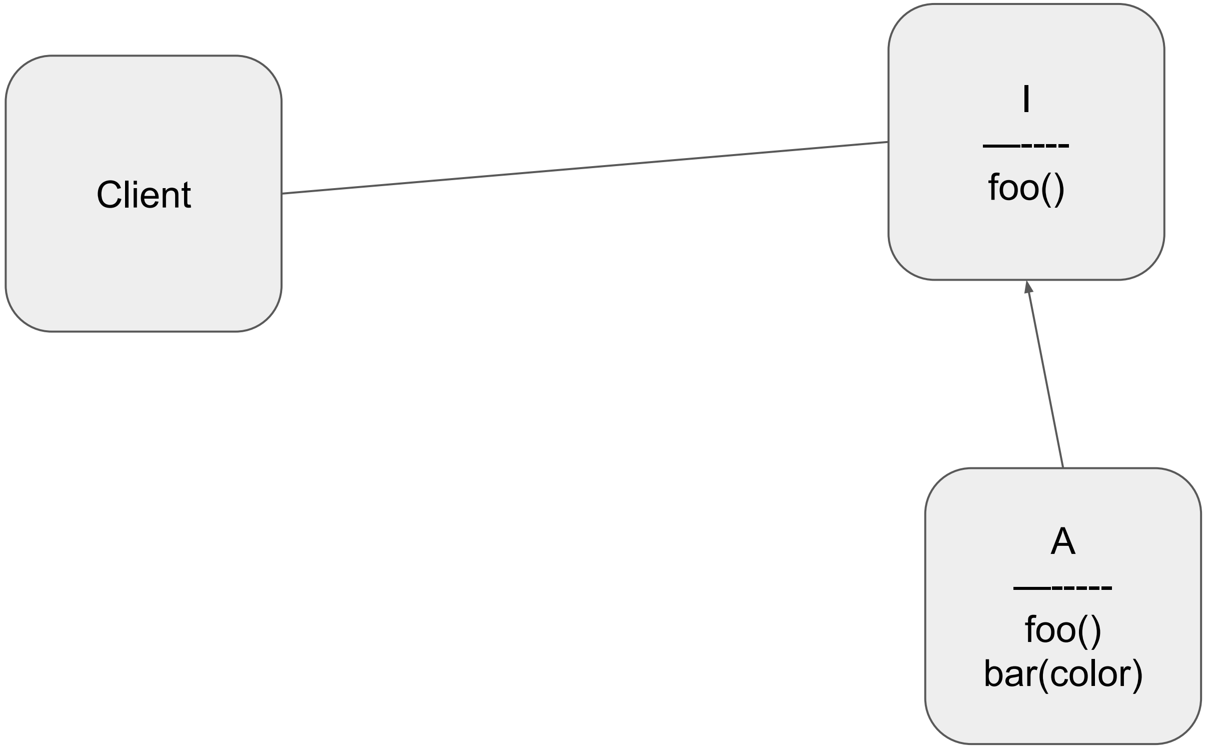

Client
Circle

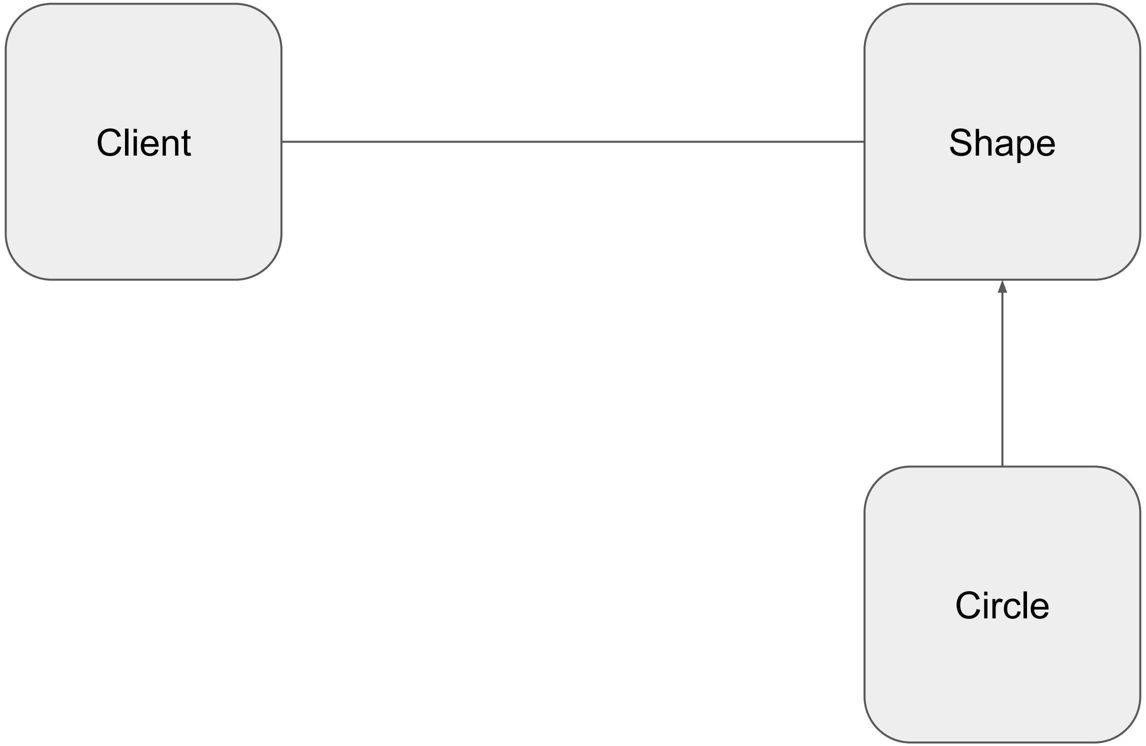

Client
Shape
Circle
Rect


Client
A
—------
foo()


Client
I
—----
foo()
A
—------
bar(color)


Client
I
—-----
foo()=0
A
—------
foo()


Client
I
—-----
foo()
A
—------
foo()
bar(color)


Client
Shape
Circle