浅谈软件设计原则之 SOLID

reallyli

2018.10.13

引言

Taylor Otwell - Creator Of Laravel (Image Copyright @Laravel.com)

Q: To someone who wants to become a better PHP developer, what is your advice?

A: **Learn good design patterns**. This does not necessarily depend on PHP. You can learn and use these patterns in any language. In particular, learn all five of the **S.O.L.I.D**. patterns thoroughly. These five patterns will take you far as a developer, and I think about each of them almost every time I code.

How is Laravel different from other framework?

Laravel has a very simple and expressive syntax. It also has some of the most thorough documentation available for any PHP framework, rivaling that of CodeIgniter's.

How is Laravel better than other framework?

Laravel provides you with great tools to start writing your application as fast as possible. This includes the beautiful Eloquent ORM, dead simple authentication, great pagination, and more.

OOP 编程常见问题?

百花齐放类

全部的功能写在一个类里面，职责太大，合并、修改
或维护变得相当困难

BUG堆积

需求一直变，为了写新功能，需要改大量代码，可是修改的地方越多，BUG 也会更多。墨菲定律告诉我们如果你担心某种情况发生，那么它就更有可能发生。

继承用太多

PHP 不允许使用多重继承，常常为了方便共用同样的功能而直接继承，导致之后要改其中一部分代码，而影响到子类

协作开发要知道的太多

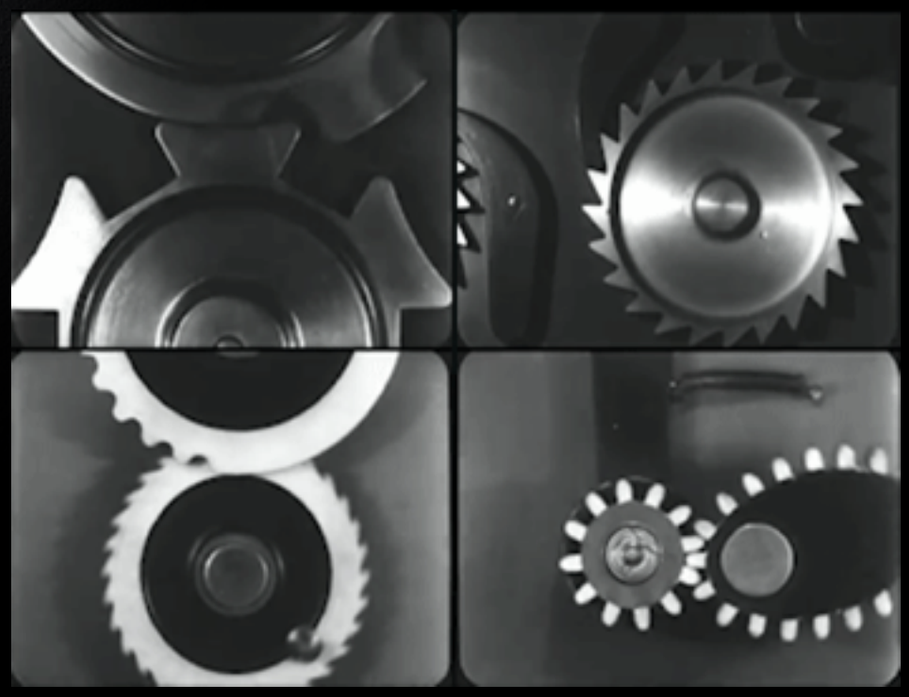
往往涉及到多人协同开发一个项目的时候，一个人要
用一个类的某个方法，由于使用方式写的太过于复杂，
花费太多时间结合上下文看如何实现的

给多用小

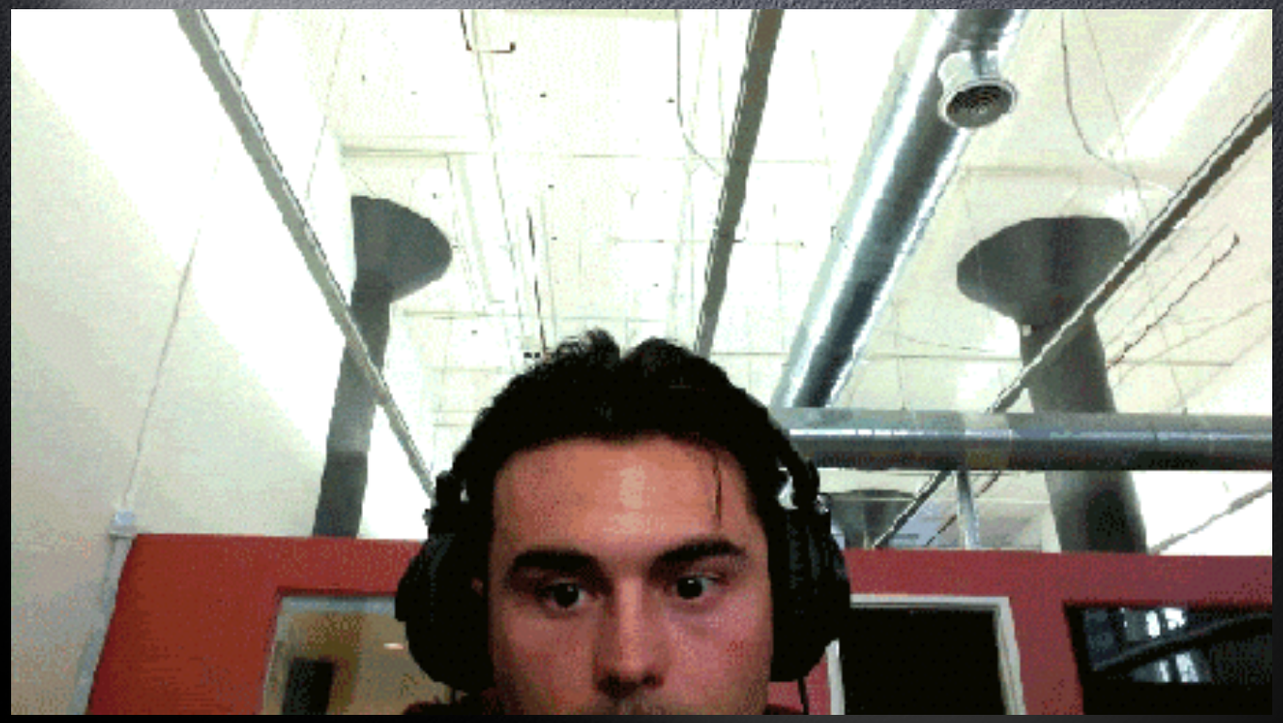
一个功能类有很多方法可以使用，可是只是需要其中一个方法

耦合度太高

类与类之间相互依赖，关系太复杂，导致牵一发动全身，修改起来很艰难，谨小慎微



该如何应对改变?



工作场景

- 小米推送要换成极光推送
- Mysql 换成 Sqlite
- 异常报告现在要增加微信企业号通知

面对代码改变的五种不同策略

SOLID

- Single Responsibility Principle 单一职责
- Opened Closed Principle 开放封闭
- Liskov Substitution Principle 里氏替换
- Interface Segregation Principle 接口隔离
- Dependency Inversion Principle 依赖反转

S	O	L	I	D
SRP	OCP	LSP	ISP	DIP

软件开发不是积木游戏。

SOLID

Software Development is not a Jenga game

单一职责原则

你可以这样干，并不是说你应该这样干

- 一个类只有一个职责
- 类具有多职责时,应该把多余职责分离出去
- 需求变化会反映为类的职责的变化
- 一个类中可以有多个方法,但是一个类只干一件事

```php
namespace Acme\Reproting;

use DB;

class SalesReporter
{
    public function between($startDate, $endDate)
    {
        //权限验证
        throw_unless(auth()->check(), '\Exception', 'Authentication required for reporting');

        //从数据库获取销售额
        $sales = $this->queryDBForSalesBetween($startDate,$endDate);

        //返回结果
        return $this->format($sales);
    }

    protected function queryDBForSalesBetween($startDate, $endDate)
    {
        return DB::table('sales')
                ->whereBetween('creates_at', [$startDate, $endDate])
                ->sum('charge') / 100;
    }

    protected function format($sales)
    {
        return "<h1>Sales:$sales</h1>";
    }
}
```

```php
namespace Acme\Reporting;

interface SalesOutputInterface
{
    public function output($sales);
}
```

```php
namespace Acme\Reporting;

use \Acme\Reporting\SalesOutputInterface;

class HtmlOutput implements SalesOutputInterface
{

    public function output($sales)
    {
        return "<h1>Sales:$sales</h1>";
    }
}
```

```php
namespace Acme\Reproting;

use Acme\Repositories\SalesRepository;

class SalesReporter
{
    private $repo;

    public function __construct(SalesRepository $repo)
    {
        $this->repo = $repo;
    }

    public function between($startDate, $endDate, SalesOutputInterface $formatter)
    {
        //从数据库获取销售额
        $sales = $this->repo->Between($startDate,$endDate);

        $formatter->output($sales);
    }
}
```

开放封闭原则 💙💚💛

OPEN CLOSED PRINCIPLE

Open Chest Surgery Is Not Needed When Putting On A Coat

开胸手术时不需要穿上一件外套

- 螺丝头可换：Open for extension 未来可应对不同的需求
- 马达不可换：Close for modification 通用部分事先做好 将来
- 也不用改 也不该改

- Open for extension
- 对功能扩展是开放的
- Closed for modification
- 扩展时不需要动原本的代码，对修改原功能代码是关闭的
- 多人开发以及维护的大原则

```php
namespace Acme;

class Square
{
    public $width;

    public $height;

    public function __construct($height, $width)
    {
        $this->height = $height;
        $this->width = $width;
    }
}

class AreaCalculator
{

    public function calcuate($squares)
    {
        $area = 0;
        foreach($squares as $square) {
            $area[] = $square->width * $square->height;
        }

        return array_sum($area);
    }
}
```

```php
namespace Acme;

class Circle
{
    public $radius;

    function __construct($radius)
    {
        $this->radius = $radius;
    }
}

class AreaCalculator
{
    public function calcuate($shapes)
    {
        foreach ($shapes as $shape) {
            if (is_a($shape, 'Square')) {
                $area[] = $shape->width * $shape->height;
            } else {
                $area[] = $shape->radius * $shape->radius * pi();
            }

            return array_sum($area);
        }
    }
}
```

```php
interface Shape {
    public function area();
}

class Square implements Shape
{
    public $width;

    public $height;

    public function __construct($height, $width)
    {
        $this->height = $height;
        $this->width = $width;
    }

    public function area()
    {
        return $this->width * $this->height;
    }
}

class Circle implements Shape
{
    public $radius;

    function __construct($radius)
    {
        $this->radius = $radius;
    }

    public function area()
    {
        return $this->radius * $this->radius * pi();
    }
}

class AreaCalculator
{
    public function calcuate($shapes)
    {
        foreach($shapes as $shape) {
            $area[] = $shape->area();
        }

        return array_sum($area);
    }
}
```

总结

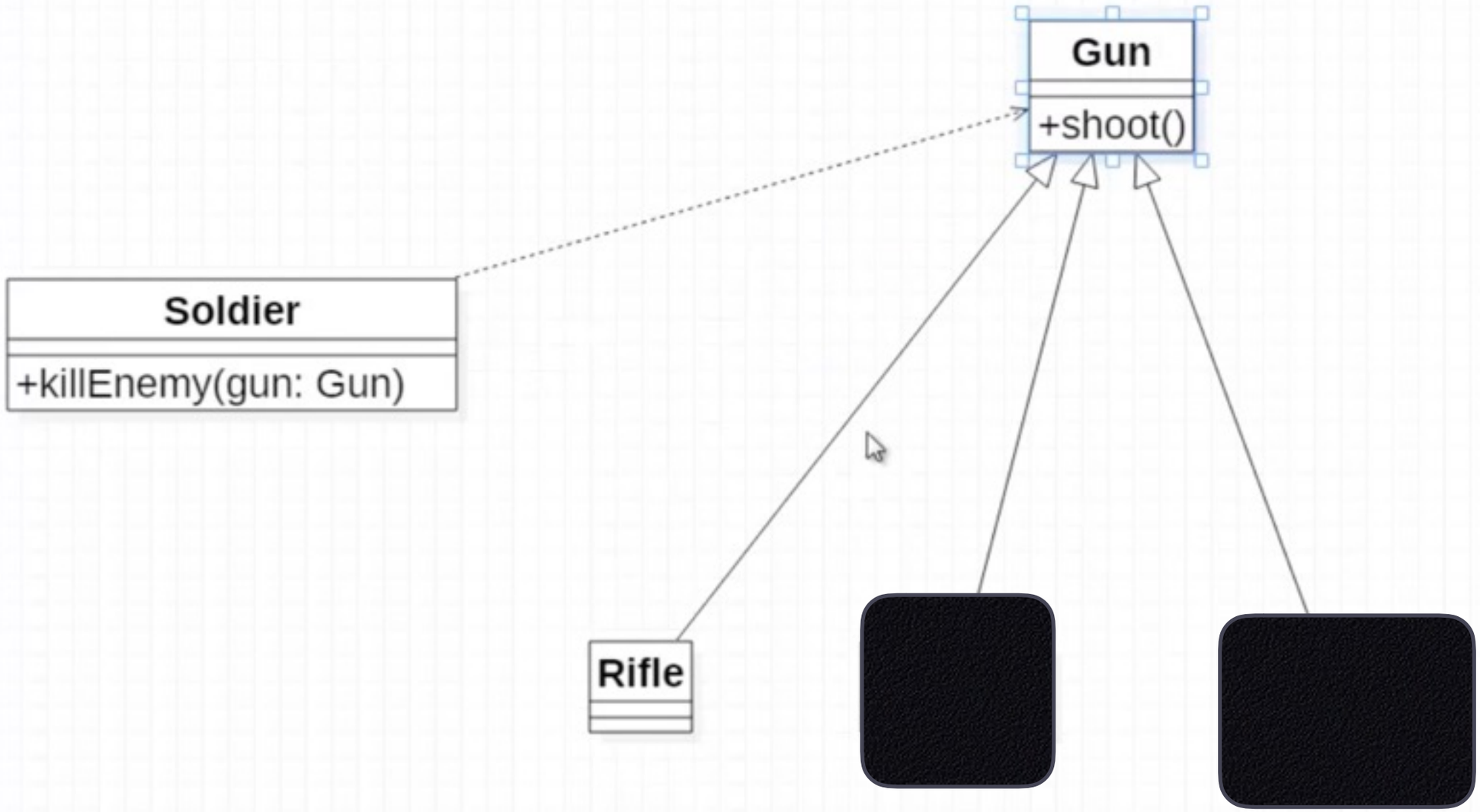
- Open-closed 开放封闭是软件开发的最高理想 在不影响既有代码的情况而增加新功能
- 猜测可能扩展的地方，区别出通用部分和扩展部分
- 将扩展部分抽象化
- 将抽象类编写出对未来扩展通用的操作

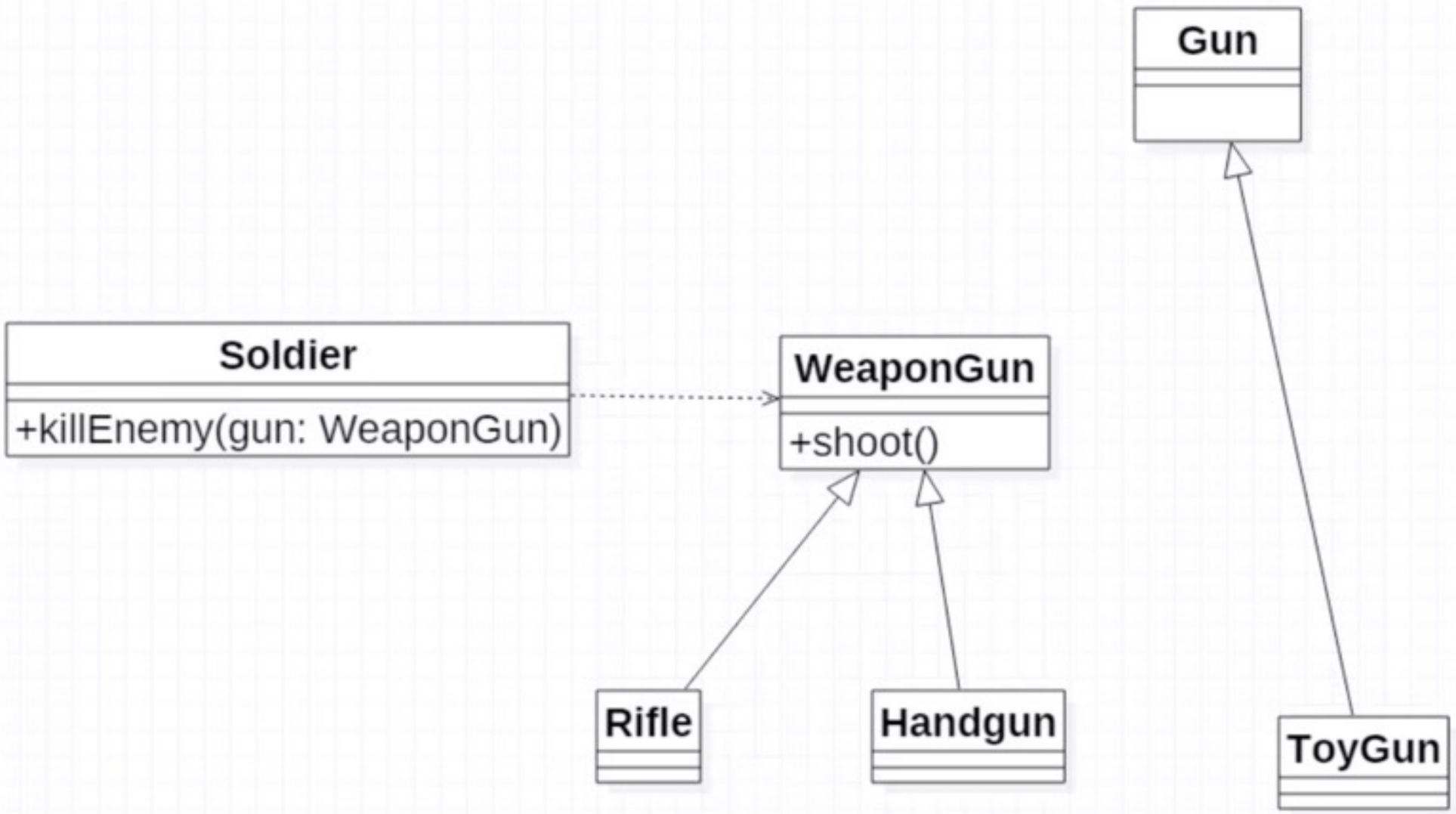
当变化到来时，我们首先需要做的不是修改代码，而是尽可能的将变化抽象出来进行隔离然后进行扩展。面对需求的变化，对程序的修改应该是尽可能通过添加代码来实现，而不是通过修改代码来实现。

里式替换原则

如果它看上去像一只鸭子，并且像鸭子一样嘎嘎叫，
但是需要电池-你可能错误的抽象了

- 类应该可以替换任何基类能够出现的地方，并且经过替换以后，代码还能正常工作
- 同一个继承体系中的对象应该有共同的行为特征。否则就不应当认可其继承关系，相当于重写出了新的父类
- 论证正方形不是长方形,鸵鸟不是鸟
- 低耦合


Gun
+shoot()
Soldier
+killEnemy(gun: Gun)
Rifle


Gun
Soldier
+killEnemy(gun: WeaponGun)
WeaponGun
+shoot()
Rifle
Handgun
ToyGun

```
class Rectangle
{
    double length;

    double width;

    public double getLength()
    {
        return length;
    }

    public void setLength(double height)
    {
        this.length = length;
    }

    public double getWidth()
    {
        return width;
    }

    public void setWidth(double width)
    {
        this.width = width;
    }
}
```

```
class Square extends Rectangle
{
    public void setWidth(double width)
    {
        super.setLength(width);
        super.setWidth(width);
    }

    public void setLength(double length)
    {
        super.setLength(length);
        super.setWidth(length);
    }
}
```

```
class TestRectangle
{
    public void resize(Rectangle objRect)
    {
        while(objRect.getWidth() <= objRect.getLength()  ) {
            objRect.setWidth(  objRect.getWidth () + 1 );
        }
    }
}
```

如果一个继承类的对象可能会在基类出现的地方出现运行错误，则该子类不应该从该基类继承，或者说，应该重新设计它们之间的关系。

接口隔离原则

我需要食物，我想吃（食物，食物），
不要去点亮枝状大烛台或者布置餐桌。

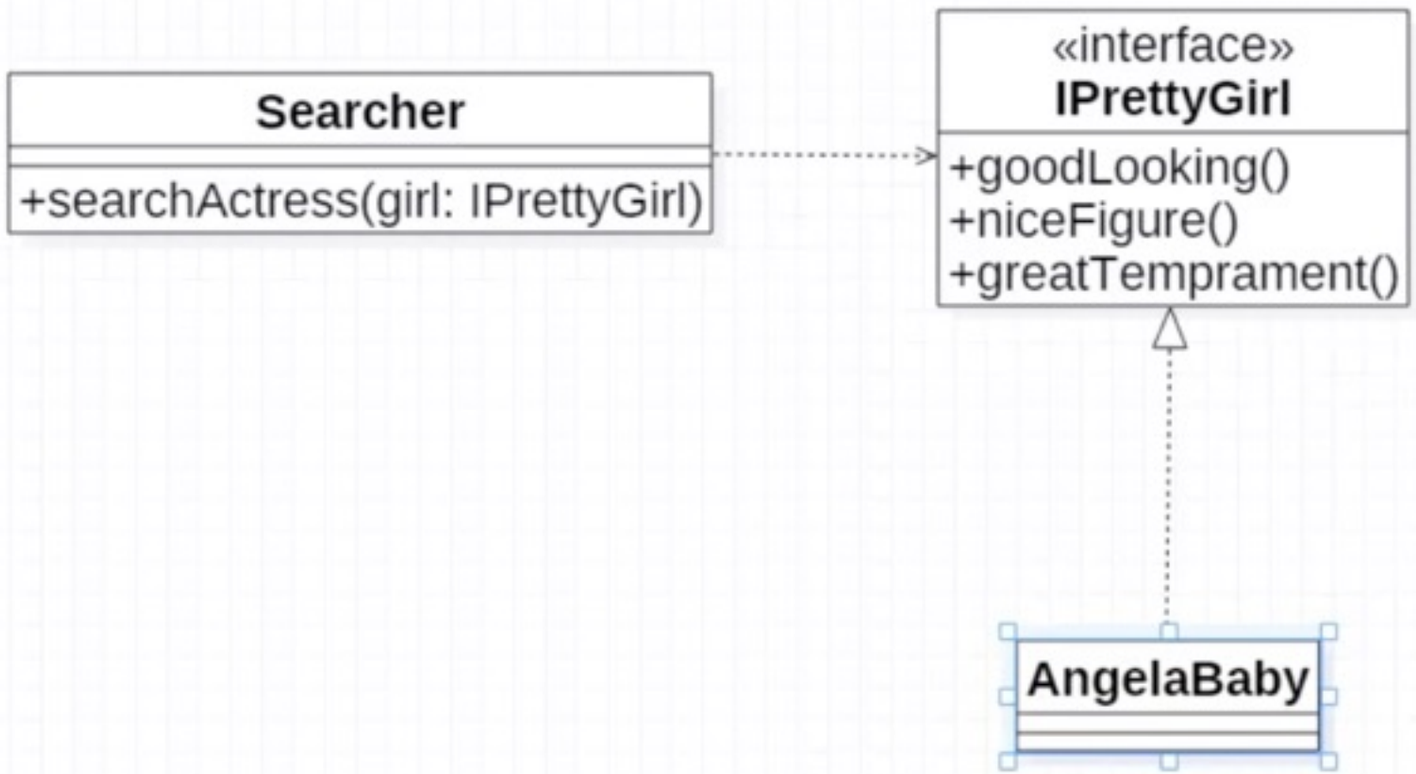
客户端不应该被迫依赖于它们不用的接口

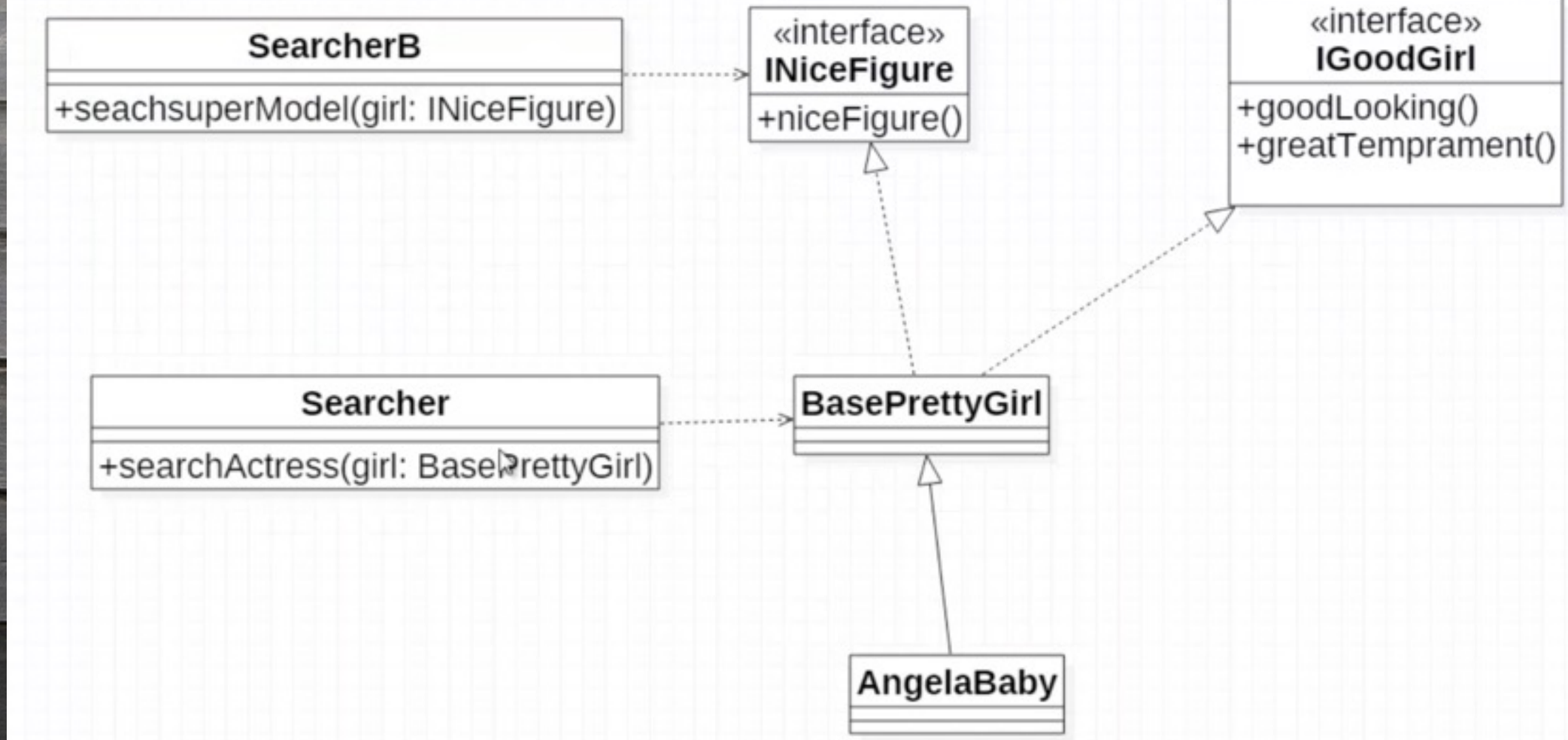
接口包含太多的方法会降低其可用性

无用方法的“胖接口”会增加类之间的耦合

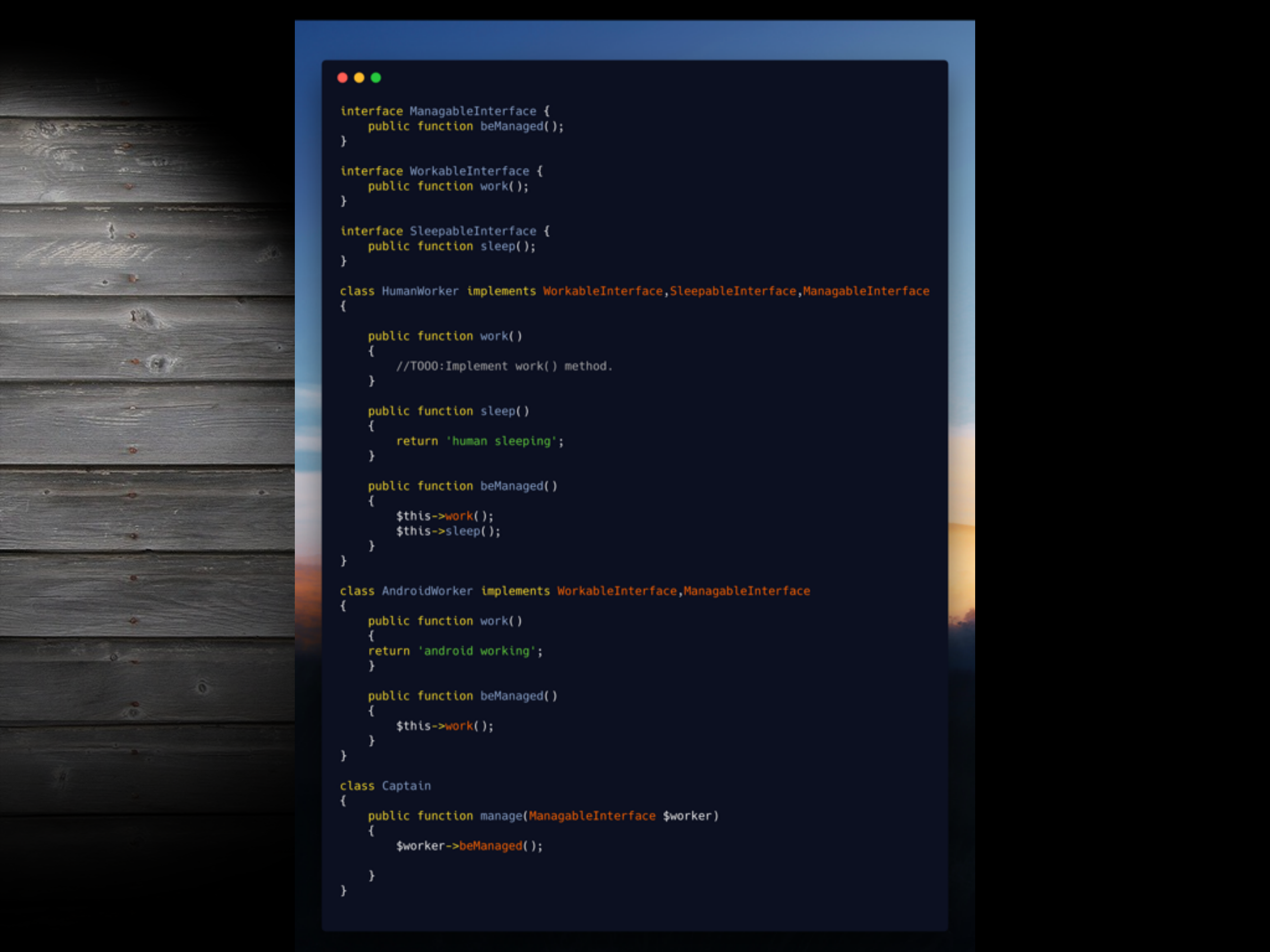
高内聚

优点: 会使一个软件系统功能扩展时，修改的压力不会传到别的对象力


Searcher
+searchActress(girl: IPrettyGirl)
«interface»
IPrettyGirl
+goodLooking()
+niceFigure()
+greatTemprament()
AngelaBaby


SearcherB
+seachsuperModel(girl: INiceFigure)
«interface»
INiceFigure
+niceFigure()
«interface»
IGoodGirl
+goodLooking()
+greatTemprament()
Searcher
+searchActress(girl: BasePrettyGirl)
BasePrettyGirl
AngelaBaby

```php
interface WorkerInterface
{
    public function work();
    public function sleep();
}

class HumanWorker implements WorkerInterface
{

    public function work()
    {
        //TODO:Implement work() method.
    }

    public function sleep()
    {
        return 'human sleeping';
    }
}

class AndroidWorker implements WorkerInterface
{
    public function work()
    {
        return 'android working';
    }

    public function sleep()
    {
        return null;
    }
}

class Captain
{
    public function manage(Worker $worker)
    {
        $worker->work();
        $worker->sleep();
    }
}
```



```php
interface ManagableInterface {
    public function beManaged();
}

interface WorkableInterface {
    public function work();
}

interface SleepableInterface {
    public function sleep();
}

class HumanWorker implements WorkableInterface,SleepableInterface,ManagableInterface
{

    public function work()
    {
        //TOOO:Implement work() method.
    }

    public function sleep()
    {
        return 'human sleeping';
    }

    public function beManaged()
    {
        $this->work();
        $this->sleep();
    }
}

class AndroidWorker implements WorkableInterface,ManagableInterface
{
    public function work()
    {
    return 'android working';
    }

    public function beManaged()
    {
        $this->work();
    }
}

class Captain
{
    public function manage(ManagableInterface $worker)
    {
        $worker->beManaged();

    }
}
```

总结

注意事项

- 接口尽量小，如果过小也会使设计复杂化
- 提高内聚，减少对外交互，使用用最少的方法去完成最多的事情
- 接口设计要适度，过大过小都不好，设计之前多思考

用户不应当被迫实现一个用不上的接口，使用多个专门的接口比使用单一的总接口要好。

依赖反转原则

高层模块不应该依赖底层模块，两者都应该依赖其抽象。

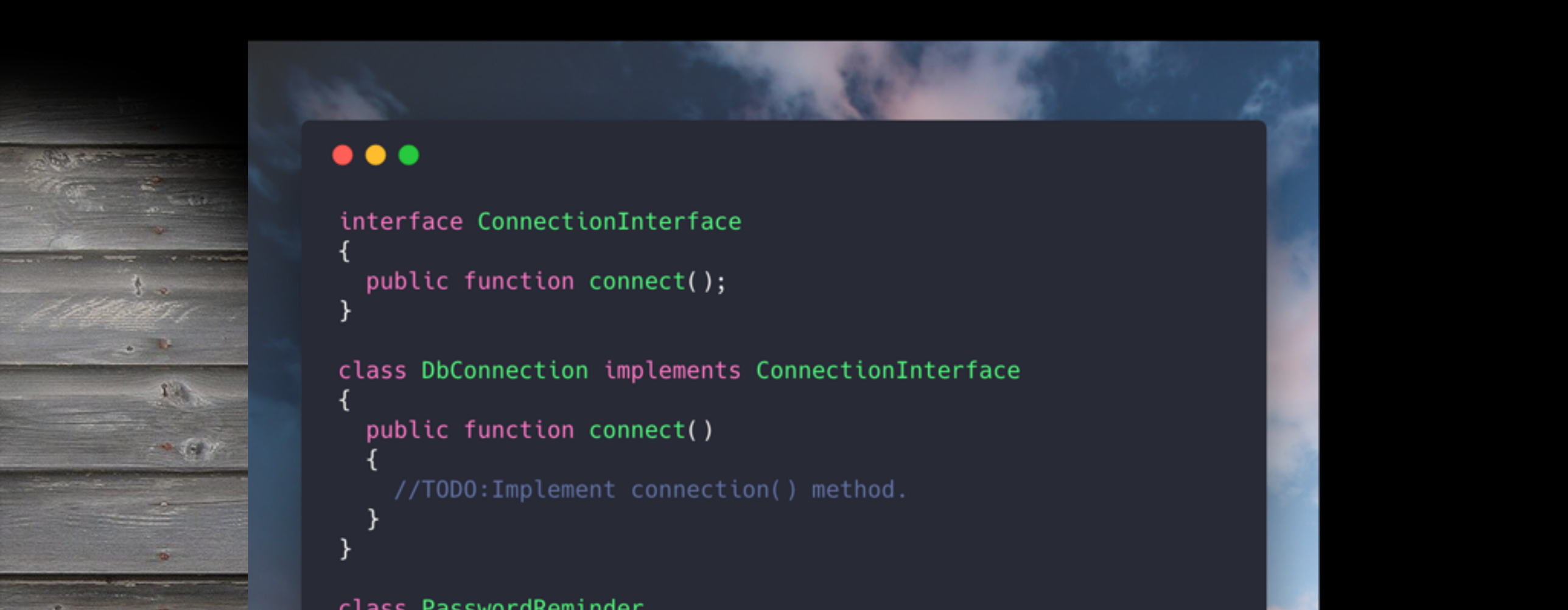
低耦合

面向接口编程，不要面向实现编程

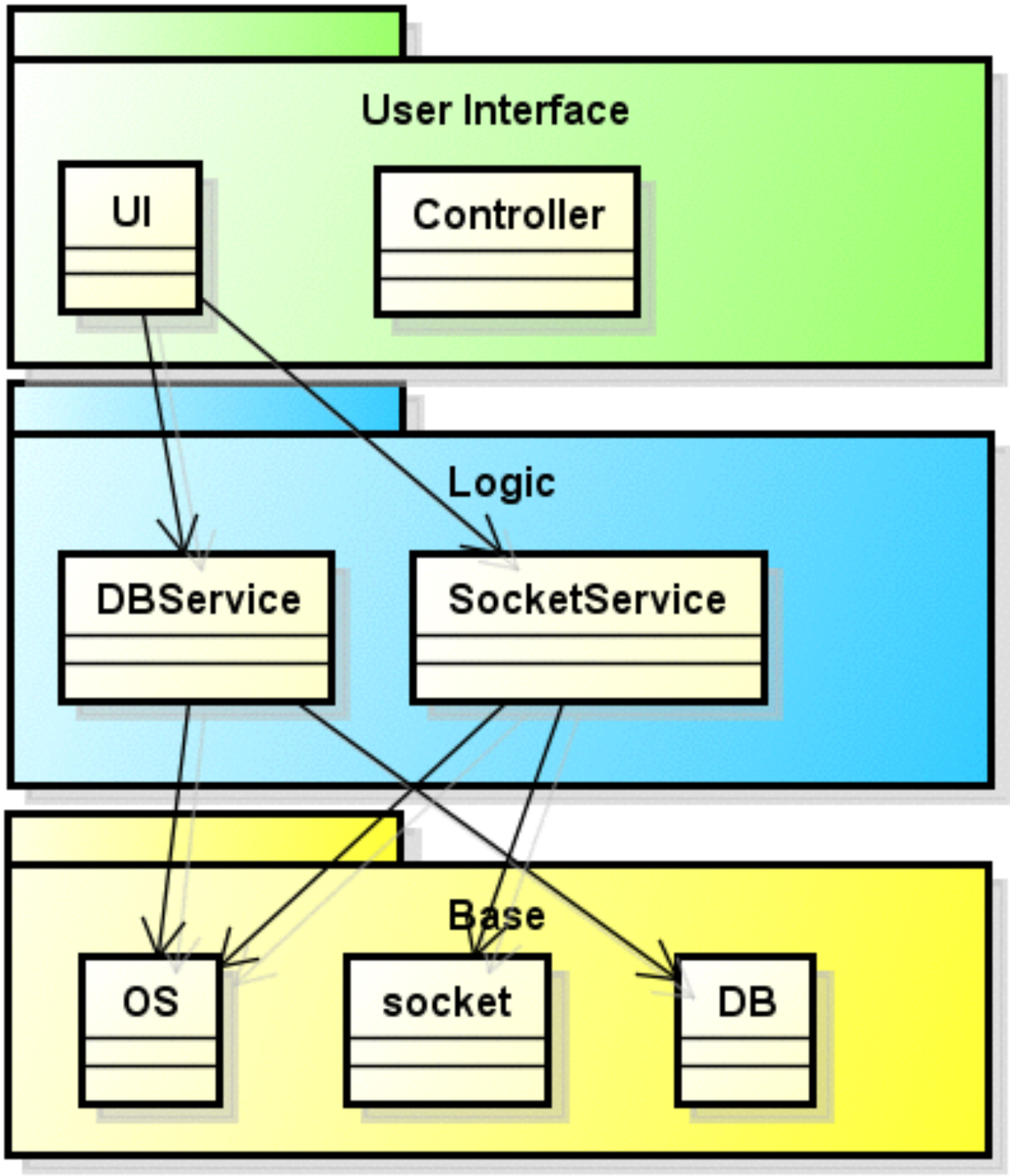
一切的关键在于解除耦合

Dependency Inversion Principle

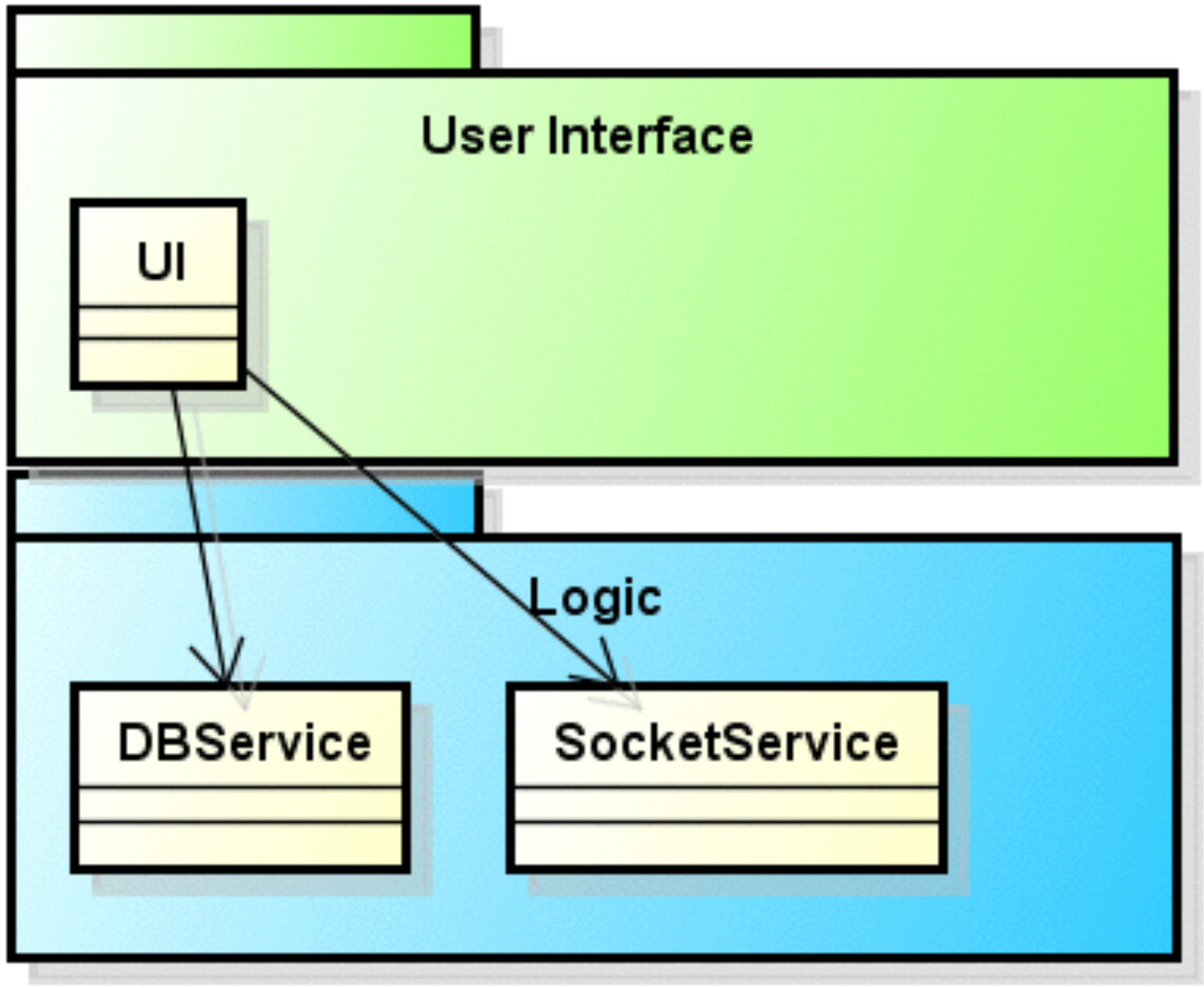
Would you solder a lamp directly
to the electrical wiring in a wall?



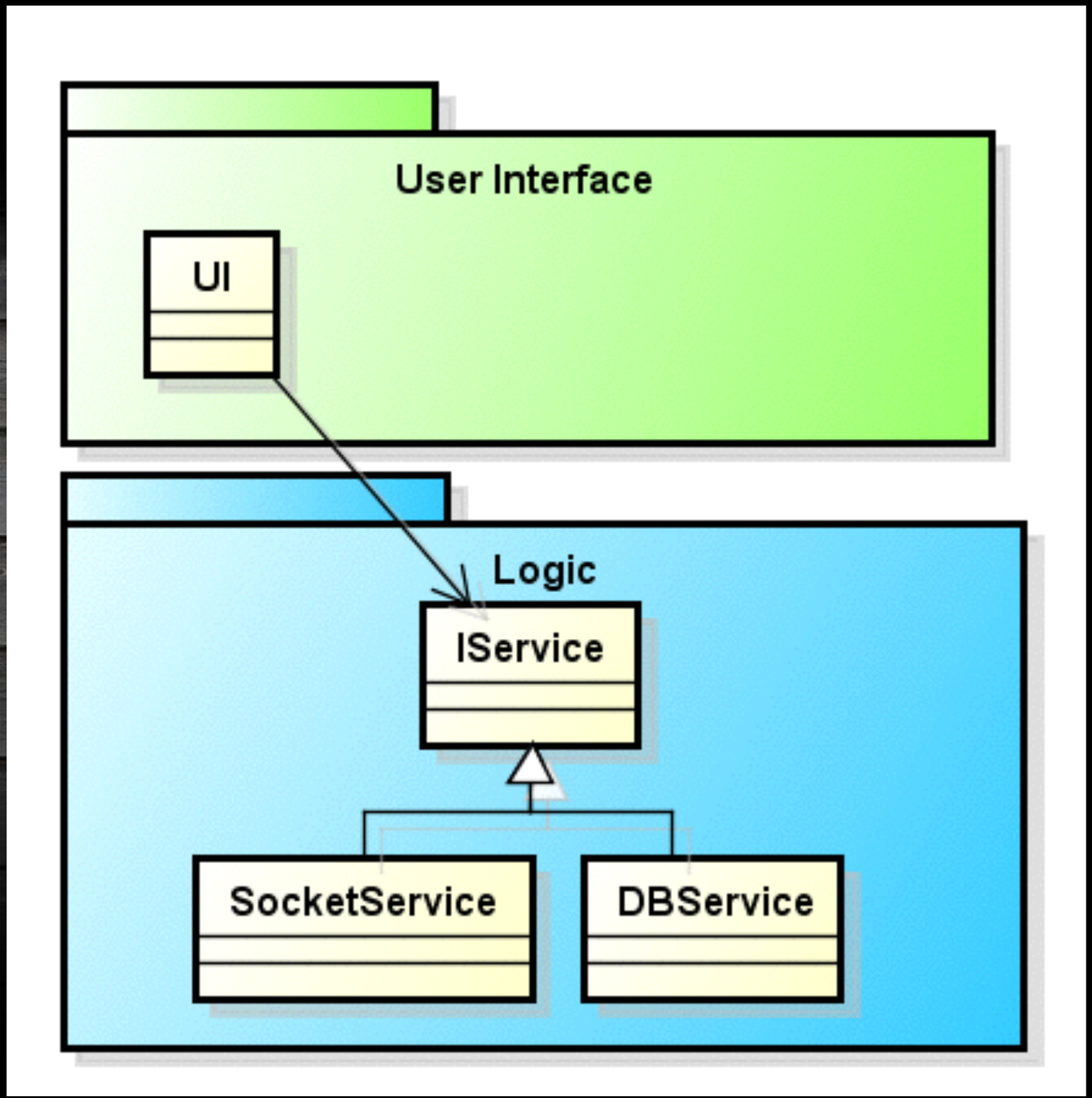
```php
interface ConnectionInterface
{
  public function connect();
}

class DbConnection implements ConnectionInterface
{
  public function connect()
  {
    //TODO:Implement connection() method.
  }
}

class PasswordReminder
{
  /**
  * @var MySQLConnection
  */
  private $dbConnection;

  public function __construct(ConnectionInterface $dbConnection)
  {
    $this->dbConnection = $dbConnection;
  }
}

$PasswordReminder = new PasswordReminder(new Dbconnection);
$PasswordReminder->dbConnection->connect();

// Photo by Jeremy Bishop on Unsplash
```



三层架构。依赖关系自上而下，也就是上层模块依赖于下层模块，而下层模块不依赖于上层



需要追加提供一种新的Service时，需要对UI层进行改动，增加了额外的工作。改动后，UI层和Logic层都必须重新再做Unit testing。


User Interface
UI
Logic
IService
SocketService
DBService

总结

- Laravel Cache Driver,Session Driver,Database Driver...
- 依赖反转是为了使高层与低层解耦。从而，高层可以高度重用。一次编写，多次使用，降低软件成本
- 抽象接口与高层模块在同一个包。接口的实现与低层模块在同一个包。可以使用多种模式来实现运行时高层接口对低层实现的选择，如Plugin, Service Locator, Dependency Injection。

上层代码依赖抽象而不是底层代码和实体。底层代码更多涉及具体的实现。

总结与回顾

面对代码改变策略

- SRP: 降低单一类别被改变所影响的机会
- OCP: 让主类别不会因为新增需求而改变
- LSP: 避免继承时子类别所造成的行为改变
- ISP: 降低用户端因为不相关的接口而改变
- DIP: 避免上层代码因为底层代码改变而改变

五个基本原则都在谈改变这一件事

概括来说它们都是在谈代码改变的五种不同的策略，或者说从五种不同的角度出发，来应对、管理代码的改变

面向接口编程，化繁为简

接受改变，预见改变，随机应变

高内聚 低耦合

The End!