



编写可测试的PHP代码

聚合数据 罗承成

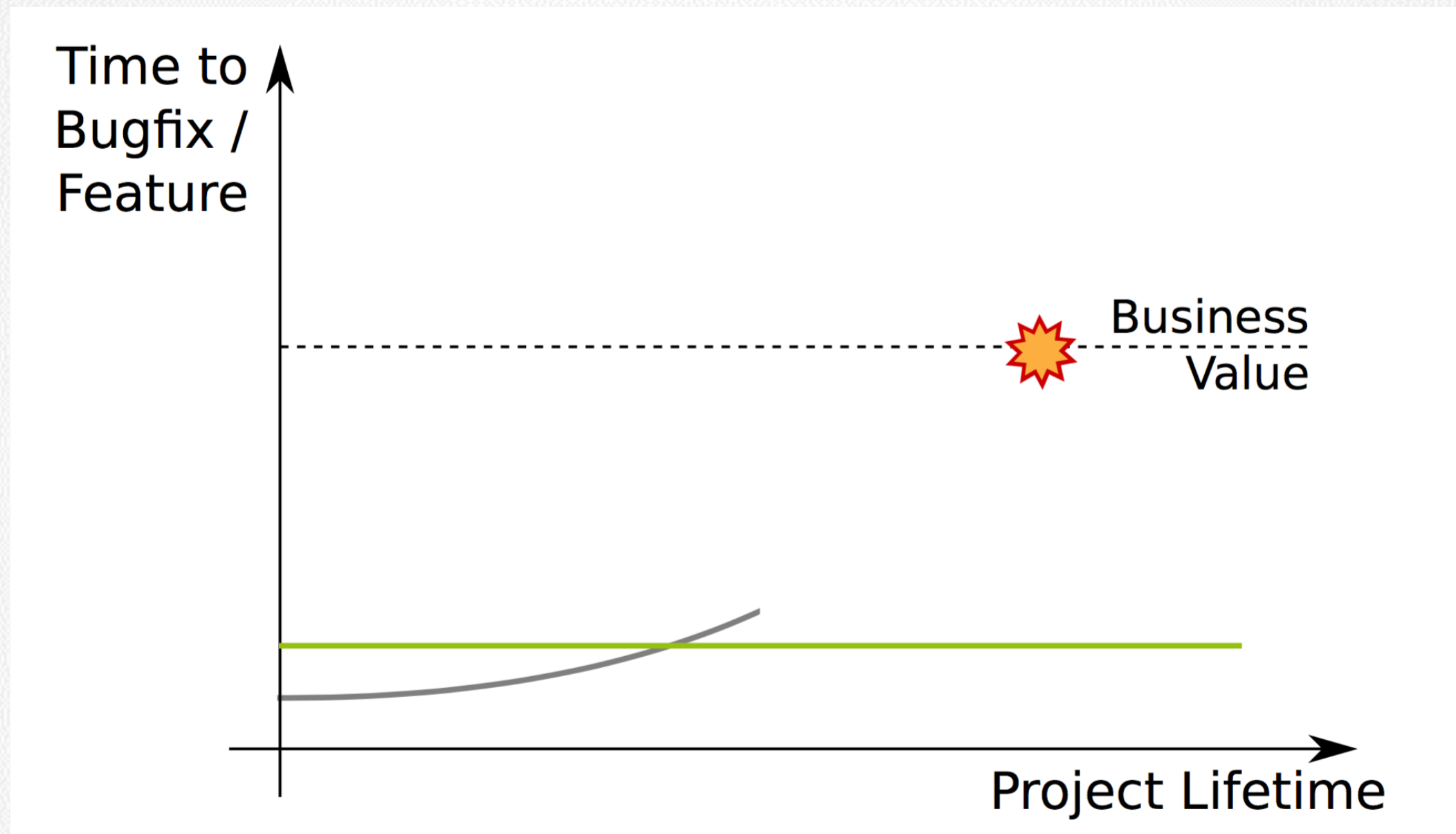
SYLLABUS

- **Part1: About Test**
 - 自动化测试
 - 单元测试
 - PHPUnit简介
- **Part2: Testable Code**
 - 什么是不可测试的代码
 - 可测试的代码长什么样
- **Part3: Test in action**
 - 持续集成
 - TDD简介

PART1: ABOUT TEST

- 自动化测试
- 单元测试
- PHPUnit简介

自动化测试



From: <https://qafoo.com>

自动化测试是项目唯一有效的防腐剂

单元测试

- 在计算机编程中，单元测试（英语：**Unit Testing**）又称为模块测试，是针对程序模块（软件设计的最小单位）来进行正确性检验的测试工作。程序单元是应用的最小可测试部件。在过程化编程中，一个单元就是单个程序、函数、过程等；对于面向对象编程，最小单元就是方法，包括基类（超类）、抽象类、或者派生类（子类）中的方法。
——Wikipedia
- “可测试”的代码，主要指的是容易实现单元测试的代码。
- 单元测试包含一个隐含的要求，需要在隔离的环境中运行被测试代码，以确保测试的是这个代码单元的行为，而不会被系统的其他部分影响。

认识一下PHPUnit

下载 <https://phar.phpunit.de/phpunit.phar>

src/Money.php

```
<?php
class Money
{
    private $amount;

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

    public function getAmount()
    {
        return $this->amount;
    }

    public function negate()
    {
        return new Money(-1 * $this->amount);
    }
}
```

tests/MoneyTest.php

```
<?php
class MoneyTest extends PHPUnit_Framework_TestCase
{
    //...
    public function testCanBeNegated()
    {
        // Arrange
        $a = new Money(1);

        // Act
        $b = $a->negate();

        // Assert
        $this->assertEquals(-1, $b->getAmount());
    }
}
```

php phpunit.phar --bootstrap vendor/autoload.php tests/

认识一下PHPUnit

```
→ pc3.0 ./phpunit.phar MoneyTest.php
PHPUnit 4.8.21 by Sebastian Bergmann and contributors.

8
.
Time: 114 ms, Memory: 13.50Mb
OK (1 test, 1 assertion)
```

认识一下PHPUnit

下载 <https://phar.phpunit.de/phpunit.phar>

src/Money.php

tests/MoneyTest.php

```
<?php
class Money
{
    private $amount;

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

    public function getAmount()
    {
        return $this->amount;
    }

    public function negate()
    {
        return new Money(-1 * $this->amount);
    }
}
```

准备对象

执行要测试的代码单元

检查结果

```
<?php
class MoneyTest extends PHPUnit_Framework_TestCase
{
    //...
    public function testCanBeNegated()
    {
        // Arrange
        $a = new Money(1);

        // Act
        $b = $a->negate();

        // Assert
        $this->assertEquals(-1, $b->getAmount());
    }
}
```

php phpunit.phar --bootstrap vendor/autoload.php tests/

PART2: TESTABLE CODE

- 可测试的设计原则
- 一些不可测试的代码
- 可测试的代码长什么样

可测试的设计原则

- 单元测试中，代码在一个隔离的环境中运行（周围没有它熟悉的框架和配置），所以尽量低的耦合和明确的依赖关系是可测试代码的最主要的要求。
- 实现低耦合的代码，重点是按照单一职责原则，对系统进行功能划分，达到高内聚（一个单元能完成一个具体的功能），低耦合（模块之间的接口最简单）。
- 明确的依赖关系需要通过依赖注入的方式来实现，不能对代码的运行环境有任何隐含的假设，这样仅通过接口的声明，而不需要了解实现，就可以知道让这个模块跑起来所需要的资源。

不可测试的代码

- 超全局变量（**Superglobals**） `$_`
- 写在构造函数中的业务逻辑
- 静态方法（**static class method**） `Class::`
- 单例（**Singleton**） `&get_instance()`
- **PHP**的动态魔术
- 超长的函数和类

下方的链接介绍了如何写出完美的不可测试代码

<http://misko.hevery.com/2008/07/24/how-to-write-3v1l-untestable-code/>

\$_GLOBAL

和运行环境的耦合

```
class Captcha extends BaseService
{
    public function checkCode($code)
    {
        //...
        if ($code == $_SESSION['auth_code']) {
            //...
        }
        //...
    }
}
```

这里假设了程序是在HTTP的环境中运行的，隐含的依赖了代码运行的方式

```
class CaptchaTest extends PHPUnit_Framework_TestCase
{
    public function testCheckCode()
    {
        $captcha = new Captcha();
        $_SESSION['auth_code'] = '123456';
        $this->assertTrue($captcha->checkCode('123456'));
    }
}
```

不看看被测试方法的具体实现，怎么能知道需要在这里设置\$_SESSION呢？

FIX：把运行环境包装进对象

```
<?php
use Symfony\Component\HttpFoundation\Session\SessionInterface;

class Captcha extends BaseService
{
    public function checkCode($code, SessionInterface $session)
    {
        //...
        if ($code == $session->get('auth_code', null)) {
            //...
        }
        //...
    }
}
```

明确对SESSION的依赖，不再访问\$_全局变量

```
class CaptchaTest extends PHPUnit_Framework_TestCase
{
    public function testCheckCode()
    {
        $captcha = new Captcha();
        $session = new Session(new MockArraySessionStorage());
        $session->set('auth_code', '123456');
        $this->assertTrue($captcha->checkCode('123456', $session));
    }
}
```

多管闲事的构造函数

隐含的依赖关系

```
class Cache extends BaseService {  
  
    private $redis;  
  
    public function __construct($config) {  
        $this->redis = new Redis();  
        $this->redis->connect($config['servers']);  
        $this->redis->select($config['database']);  
    }  
    ...  
}
```

用数组作为参数，不看看具体实现怎么知道这个参数什么格式呢？注释靠得住么？

这个类隐含的依赖一个REDIS，要测试这个类还需要连接REDIS服务器吗？如果我的REDIS服务器有密码怎么办？

FIX：依赖注入

遵循依赖注入的代码可以在任何地方重用，只要满足了它在构造函数声明的依赖条件。

```
<?php
class CacheService extends BaseService
{
    private $driver;

    public function __construct(DriverInterface $driver)
    {
        $this->driver = $driver;
    }
    //...
}

interface DriverInterface
{
    public function getData($key);
    public function storeData($key, $data, $expiration);
    public function clear($key = null);
    public function purge();
    public static function isAvailable();
    public function isPersistent();
}
```

SINGLETON和静态方法

和框架的耦合

- 大部分**PHP**框架为了方便，都会把框架提供的功能通过全局函数，静态方法，或者**Singleton**的形式提供给使用者。



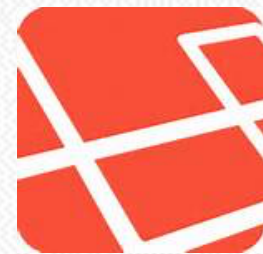
`&get_instance()`



`M() V() C()`



`App::`



`Facades`

- 这些对框架的依赖往往都是隐含的，分布在整个代码库中，不看具体的实现没法知道某个类具体用到了哪些框架提供的功能。
- 使用了这些框架服务的代码，离开了框架就跑不起来了，要在**PHPUnit**的环境下运行代码，必须先把整个框架**bootstrap**起来，也就无法做到单元测试需要的隔离。

FIX：依赖注入容器

- 近几年出现的**PHP**框架都包含了依赖注入容器，把框架提供的功能以服务对象的形式提供给应用。
 - 使用**PHP-DI**的例子

```
class CaptchaService extends BaseService {  
    private $cache;  
  
    public function __construct(CacheInterface $cache) {  
        $this->cache = $cache;  
    }  
    ...  
}  
  
$container->set(CacheInterface::class, \DI\object('RedisCache'));  
$captcha = $container->make(CaptchaService::class)
```

FIX：依赖注入容器

- 依赖注入的设计模式让代码简单了（需要什么东西直接伸手要），但是对象的创建变得复杂了（递归的依赖关系，可能会需要创建很多对象），所以需要有一个工具来完成对象的创建。
- 这个容器不也是上帝对象吗，和之前流行的全局静态方法，**Singleton**相比，不过是把依赖的对象换成了依赖注入容器吗？
- **Singleton**或者静态方法是被代码直接使用的，对他们的调用散布在应用的各个角落
- 遵循依赖注入的代码从不直接使用依赖注入容器，只声明自己使用的对象，由依赖注入容器来实例化自己

PHP的魔法

和运行时数据的耦合

- `$$var_name`
- `__call()` `__get()` `__set()`
- `extract()`
- `compact()`

FIX： 不是必须，尽量别用

XXXXXL函数

一定是承担了太多的职责

- 这样的类或者函数设计必然是违背了单一职责的要求。
- 例如一个加入游戏的函数，要做的事会分为几个部分：
 - 确定用户和状态
 - 确定游戏和状态
 - 加入游戏
 - 通知游戏中的其它用户
- 如果混在一起实现，新增了用户类型，新增了游戏状态，游戏限制了加入时间，通知用户的方式，这些变化都需要修改这个函数。
- 如果这些职责分散在不同的代码中，每个变化只需要修改和测试相关的代码



FIX：重构

- **Step1:** 准备一些可以成功运行的单元测试用例
- **Step2:** 识别出混在一起的多个职责
 - 庞大的**IF**分支
 - 重复的代码块
 - ...
- **Step3:** 将这些职责在别的类或者方法中单独实现
- **Step4:** 将原来的特大函数修改成对这些拆分出来的方法的调用和集成
- **Step5:** 使用单元测试检查函数行为是否改变

总结

- 可测试的代码，一定是遵照依赖倒置原则**DIP**写出的代码
- 代码模块之前的依赖关系，是按照单一职责**SRP**的划分产生的
- 单一职责决定了接口设计需要小而专一，即接口隔离**ISP**
- 单一职责原则要求把系统中变化的和不变的部分分离，以达到对扩展开放，对修改封闭**OCP**
- 继承和多态是实现可扩展架构的主要工具，里氏替换原则**LSP**明确了对继承的要求：子类需要能胜任父类工作的所有岗位，才算个合格的子类。

可测试的代码==SOLID

- | | | |
|---|--------|-----|
| • Single Responsibility Principle | 单一职责原则 | SRP |
| • Open-Close Principle | 开闭原则 | OCP |
| • Liskov Substitution Principle | 里氏替换原则 | LSP |
| • Interface Segregation Principle | 接口隔离原则 | ISP |
| • Dependency Inversion Principle | 依赖倒置原则 | DIP |

PART3: TEST IN ACTION

- 持续集成
- TDD简介

PART3: TEST IN ACTION

“

“Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead.”

— Martin Fowler

”

持续集成

- 持续集成并不属于自动化测试，但它是保证自动化测试能顺利实施的必要条件。
- 持续集成会在代码每次提交时执行代码构建，自动化测试，代码检查，自动部署等工作，给开发人员及时的反馈，来保证小问题不会积累。
- 单元测试时需要同代码一起变化的，只有一直在运行的测试代码才不会被遗忘，才能被不断维护。
- 解决开发与测试之间“在我机器上没问题”的问题

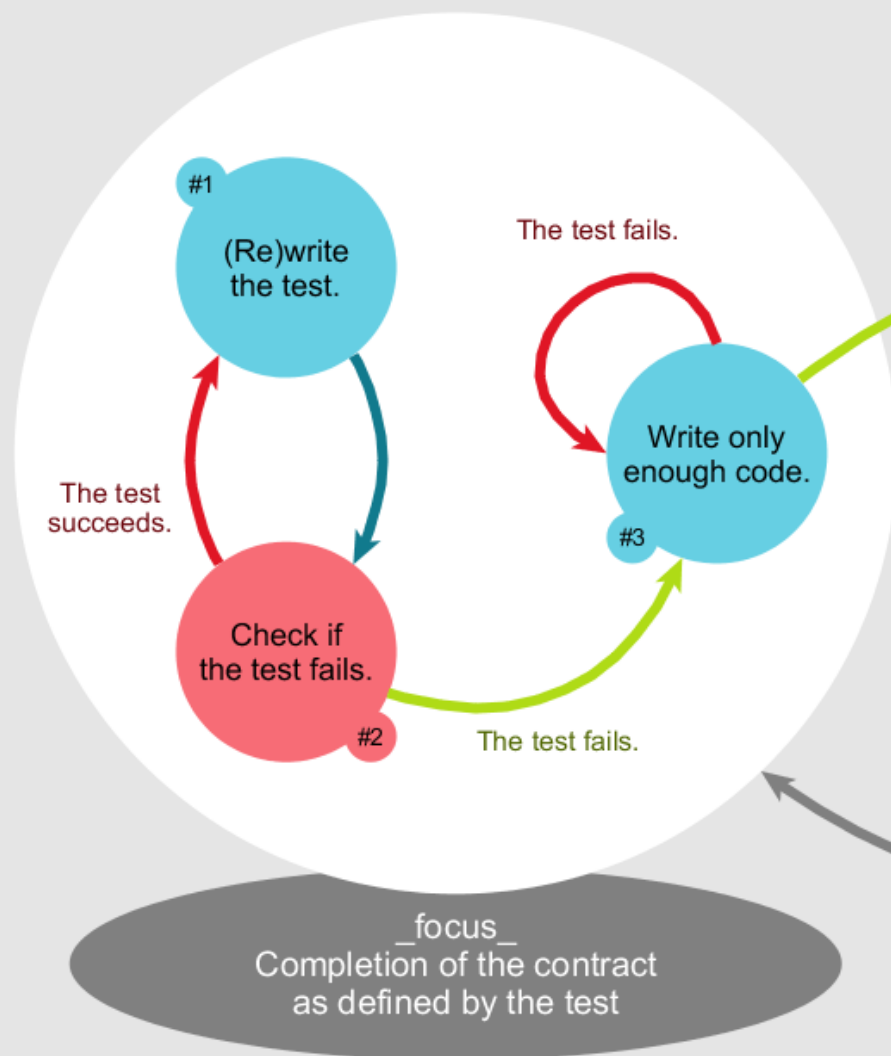
CI工具箱

- Jenkins
- PHPUnit
- Codeception
- Behat
- PHPSpec
- PHP Mess Detector
- PHP Code Sniffer

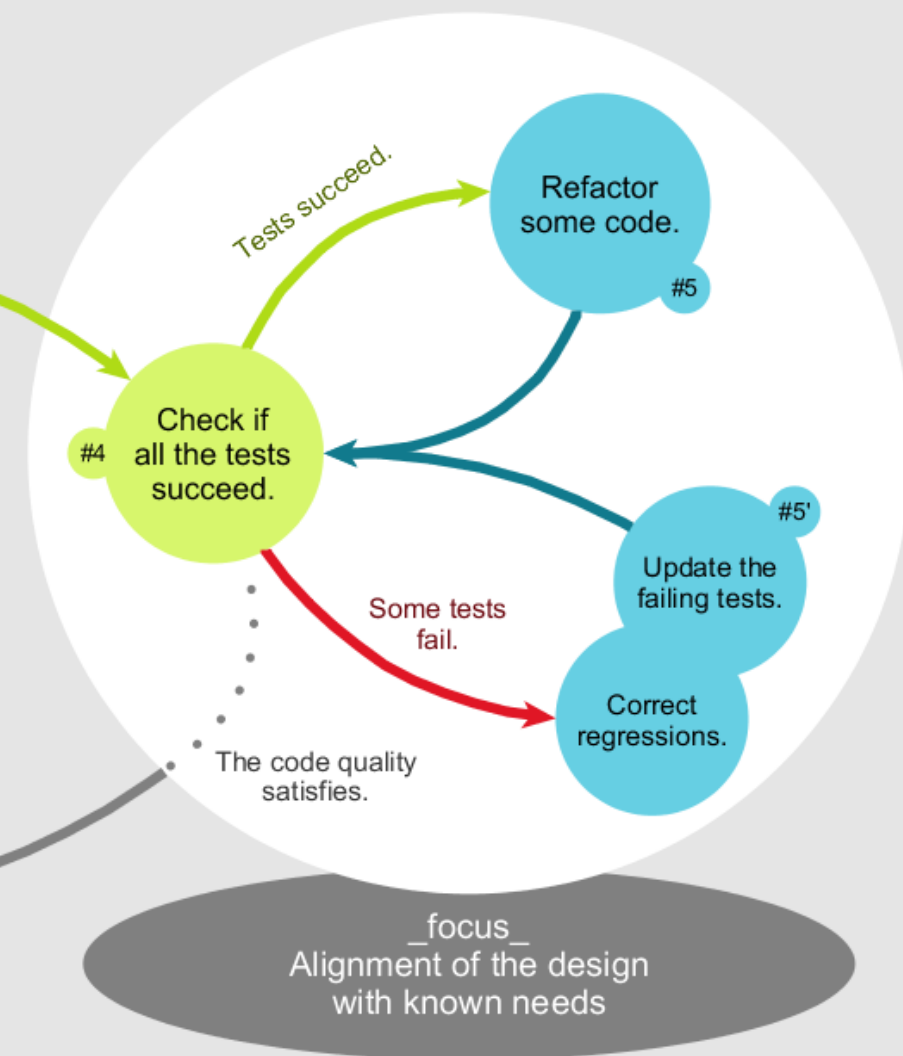
TDD

测试驱动开发

TEST-FIRST DEVELOPMENT



REFACTORING



Iterate

测试驱动开发会带来的改变

- 一开始的进度变慢了
 - 写代码前需要更多的思考，分解问题，设计接口
 - 学习测试工具，准备测试代码
- 可测试的代码
 - 测试驱动会促使你写出可测试的代码，不然写测试用例没法下手
 - 有了自动测试的保护，可以随时重构看着不顺眼的代码
- 信心
 - 对于已经完成的代码正确性有十足的信心
 - 面对新功能和需求变更，可以对开发时间有准确的估计
 - 遇到bug可以快速定位和修复问题

Thank you



微信公众号