遗留代码重构&测试

番外篇：测试替身

CAC@OPPO by 黄俊彬 & 覃宇

讨论：要让被测的代码工作，有哪些条件很难满足？

以一段 Retrofit 请求为例...

```
interface MyService {
  @GET("/user")
  Observable<User> getUser();
}

Retrofit retrofit = new Retrofit.Builder()
    // 服务可能挂掉，或者还没实现，或者网络延时、中断
    .baseUrl("https://example.com/")
    .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
    .build();

MyService myService = retrofit.create(MyService.class)

myService.getUser()
         // 异步代码不稳定
         .subscribeOn(Schedulers.io())
         .observeOn(AndroidSchedulers.mainThread())
         // 可能与界面代码耦合
         .subscribe(user -> view.load(user));
```

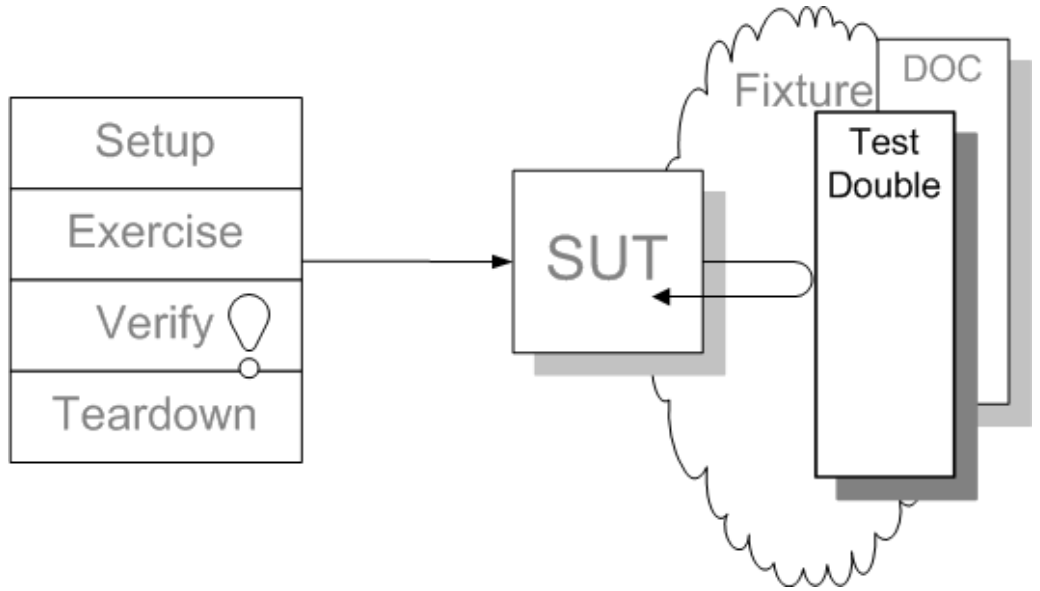
难以满足的依赖条件

- 还没有实现的代码，比如先定义了接口，但实现还没有写
- 可用性不受控制，比如不稳定的第三方服务、网络
- 数据一直在变化，比如生产环境的数据数据库
- 使用起来成本太高，比如I/O延时、定时任务、物理环境
- 执行时间不稳定，比如异步代码
- 传递依赖太多，创建没完没了

> 可以使用**安全重构**先**解除依赖**，但有没有其他同样成本较低的方法

什么是测试替身

替换**被测系统**的**依赖**的**等价**实现



还是以这个 Retroft 请求为例

```
interface MyService {
  @GET("/user")
  Observable<User> getUser();
}

Retrofit retrofit = new Retrofit.Builder()
    // 服务可能挂掉，或者还没实现，或者网络延时、中断
    .baseUrl("https://example.com/")
    .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
    .build();

MyService myService = retrofit.create(MyService.class)

myService.getUser()
        // 异步代码不稳定
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        // 可能与界面代码耦合
        .subscribe(user -> view.load(user));
```

> 依赖：Retrofit、网络连接、https://example.com/、View、异步

1. 使用自定义的 MyService 实现

```
// 以下是测试代码
// 自定义“假”实现，按照预期返回 User 实例
MyService stubService = new MyService() {
    @Override Observable<User> getUser() {
        return Observable.from(new User());
    }
}

// 使用“假”实现
stubService.getUser()
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(user -> view.load(user));

assertThat(view)...
```

> 解除的依赖：Retrofit、网络连接、https://example.com/

2. 直接测试回调

```
// 以下是测试代码
User userResponse = null;

// 使用“假”实现
stubService.getUser()
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(user -> userResponse = user));

assertThat(user)...
```

> 解除的依赖：View

3. 替换 Retrofit 的网络请求行为

```
interface MyService {
  @GET("/user")
  Observable<User> getUser();
}

Retrofit retrofit = new Retrofit.Builder()
    .baseUrl("https://example.com/")
    .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
    .build();

// 以下是测试代码
// 使用 Retrofit 提供的 BehaviorDelegate 自定义“假”实现
public class MyServiceMock implements MyService {
    private final BehaviorDelegate<MyService> delegate;
    public MyServiceMock(BehaviorDelegate<MyService> delegate) {
        this.delegate = delegate;
    }

    public Observable<String> user() {
        return delegate.returningResponse(Observable.from(new User())).user();
    }
}
```

3. 替换 Retrofit 的网络请求行为(续)

```
// 使用 Retrofit 提供的 MockRetrofit 组装“假”实现和“假”网络行为
NetworkBehavior behavior = NetworkBehavior.create();
MockRetrofit mockRetrofit = new MockRetrofit.Builder(retrofit)
                .networkBehavior(behavior)
                .build();

BehaviorDelegate<MyService> delegate = mockRetrofit.create(MyService.class);
MyService mockService = new MyServiceMock(delegate);

// 使用 “假”实现
mockService.getUser()
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(user -> view.load(user));
assertThat(view)...
```

3.替换 Retrofit 的网络请求行为(续)

```
// Retrofit 提供的 BehaviorDelegate 还可以模拟网络失败
behavior.setDelay(0, MILLISECONDS);
        behavior.setVariancePercent(0);
        behavior.setFailurePercent(failurePercent);

// 使用 “假”实现
mockService.getUser()
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(user -> view.load(user));

assertThat(view)...
```

> 解除的依赖：https://example.com/、网络连接

4. 解决异步的稳定性问题

```
// 以下是测试代码
// RxJava 提供给的“假”异步实现 TestSubscriber
TestSubscriber<User> testSubscriber = TestSubscriber.create();

mockService.name().subscribe(testSubscriber);
testSubscriber.assertValue(expectedUser);
testSubscriber.assertCompleted();
```

> 解除的依赖：异步

5. 使用本地的“假”服务代替真实服务

```
// 以下是测试代码
// 让 Retrofit 用本地服务代替 https://example.com/
MockWebServer mockWebServer = new MockWebServer();
Retrofit retrofit = new Retrofit.Builder()
    .baseUrl(mockWebServer.url("/"))
    .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
    .build();
MyService myService = retrofit.create(MyService.class)
```

5. 使用本地的“假”服务代替真实服务(续)

```
// 存放在本地测试得 Response 文件
{"name":"qinyu", "phnoe":"11123456789"}
```

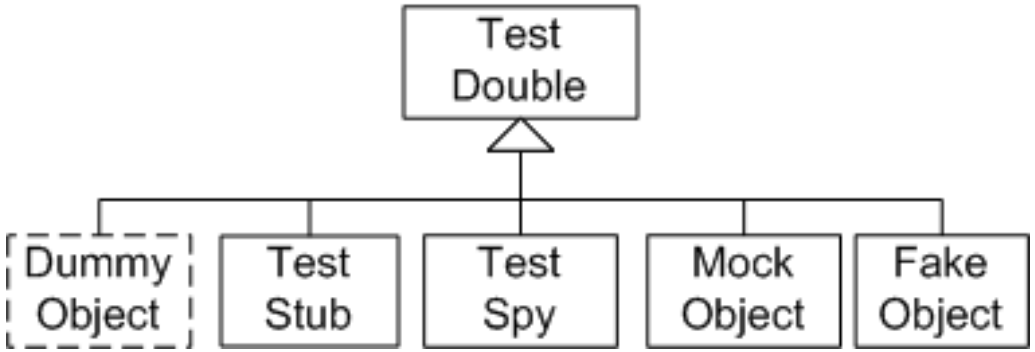
```
// 读取本地文件模拟“假”的 Response
MockResponse response = new MockResponse()
        .setResponseCode(HttpURLConnection.HTTP_OK)
        .setBody(readContentFromFilePath());
mockWebServer.enqueue(response);

myService.getUser()
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(user -> view.load(user));

assertThat(view)...
```

> 解除的依赖：网络连接、https://example.com/

小结：测试替身的种类


Test
Double
Dummy
Object
Test
Stub
Test
Spy
Mock
Object
Fake
Object

Dummy，为了让测试可以进行（编译通过），不会在测试中使用，一般用来填充参数
Stub，事先准备好返回数据，在测试中的调用时使用这些数据代替真实调用的返回值
Spy，记录如何测试中调用的方法的执行情况，包括调用次数、调用顺序、调用时传入参数
Fake，是一种完整的实现，但和生产环境实现不同，但更轻量、更简单，如内存数据库
Mock，使用库自动生成的替身，可以完成 Stub 和 Spy 的功能。

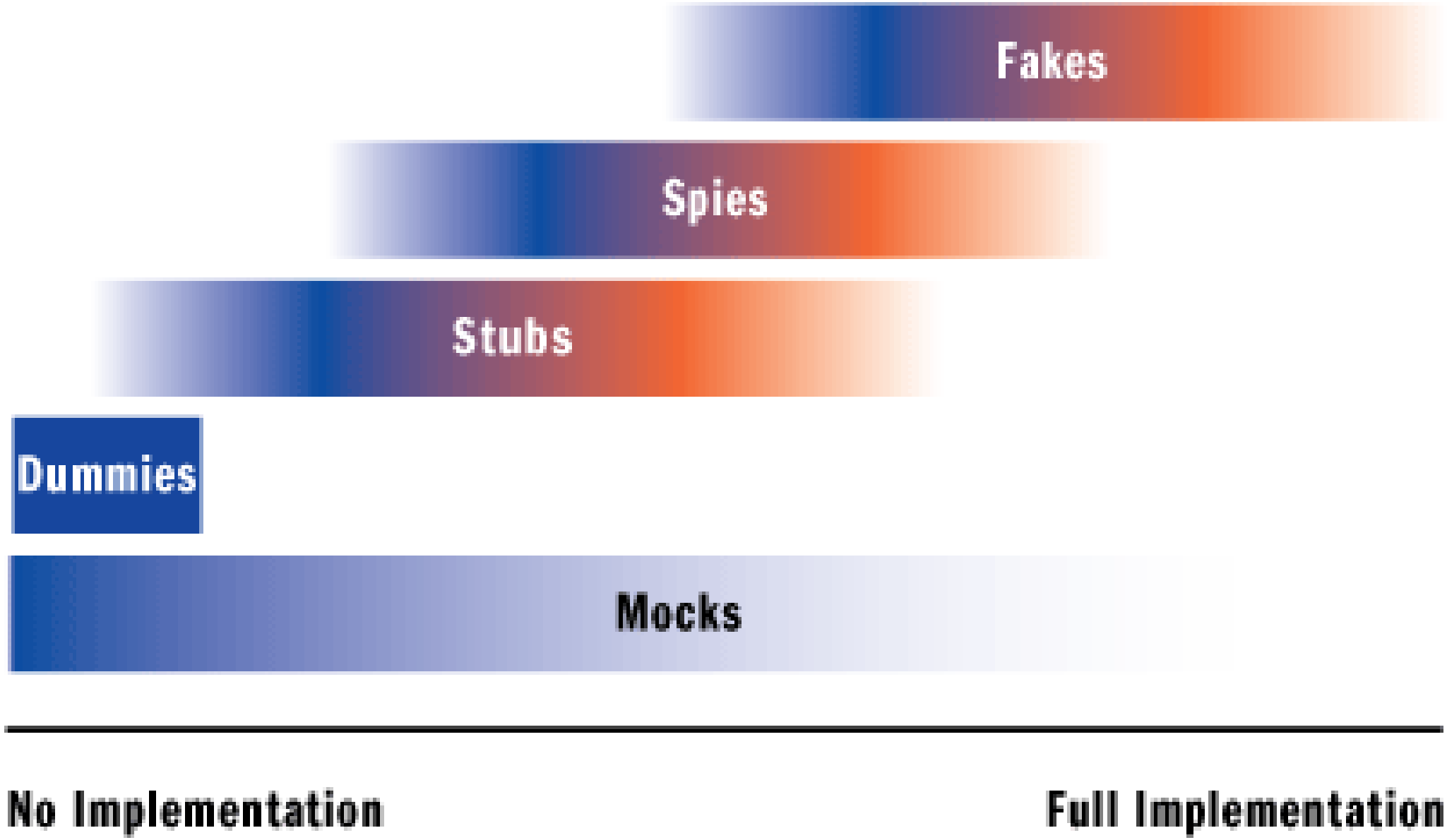
讨论：上面使用的这些手段分别是哪种替身？

1. 使用自定义的 MyService 实现
2. 直接测试回调
3. 替换 Retrofit 的网络请求行为
4. 解决异步的稳定性问题
5. 使用本地的“假”服务代替真实服务

我们的看法

1. 使用自定义的 MyService 实现（Stub）
2. 直接测试回调（Spy）
3. 替换 Retrofit 的网络请求行为（Stub）
4. 解决异步的稳定性问题（Stub+Spy/Mock）
5. 使用本地的“假”服务代替真实服务（Fake）

测试替身的局限性


Fakes
Spies
Stubs
Dummies
Mocks
No Implementation
Full Implementation

讨论：如何保证替身和依赖的行为是等价的？

请记住：

- 替身始终是**假**的
- 做得越真**投入越大**
- 用得越多**问题越多**

我们的建议：

1. 结合重构使用 Mock，并且优先考虑重构
2. 不要模拟不确定的行为
3. 不要模拟传递依赖，只模拟直接依赖
4. 尽量把要模拟的行为隔离起来

选择值得信任的外部依赖

- 接口行为有自动化测试保障
- 接口文描述清晰

> 作为技术选择的必要条件

选择有保障的 Mock 库

- Mockito（使用得最多，可读性最好）
- Robolectric（使用最新 Android 源码在 JVM 上编译，基本上等同原生行为）
- 各种来源库自带的 Mock/Test Utils
- 不建议使用：Powermock

依赖一定要双方共同维护接口“契约”

- 接口 Producer 提供自动化测试和执行结果
- 接口 Consumer 及时根据变化更新 Mock 的行为