

Test와 Test Double의 종류

Test의 종류

연동의 범위에 따른

Clean Code Talks - Unit Testing

Unit Test의 종료(from 'Working Effectively with Unit Tests', Jay Fields)

Two schools of TDD

Test Double의 종류(xUnit Test Patterns: Refactoring Test Code)

The Little Mocker(<http://blog.cleancoder.com/uncle-bob/2014/05/14/TheLittleMocker.html>)

Mock Roles, not Objects(<http://jmock.org/oopsla2004.pdf>)

OOP

MOCK OBJECTS AND NEED-DRIVEN DEVELOPMENT

Test의 종류

연동의 범위에 따른

- The Test Pyramid

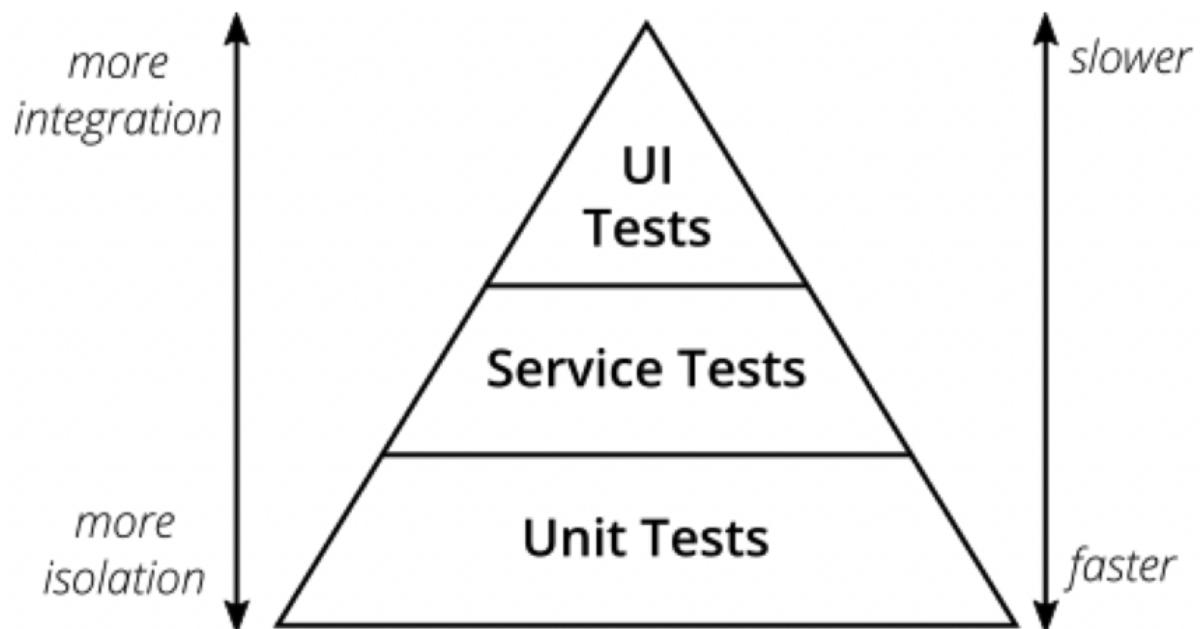


Figure 2: The Test Pyramid


from Succeeding with Agile, Mike Cohn

현재 적용하기에는 다소 모호할 정도 추상적임

Clean Code Talks - Unit Testing

msbaek/memo

"There is no secret to writing tests..... there are only secrets to writing testable code!" 테스트 작성에 대해서는 비법이 없다. 테스트 가능한 코드를 작성하는 비법만이 존재한다. Good OO

 <https://github.com/msbaek/memo/blob/master/CleanCodeTalks-UnitTesting.md>



Unit Test의 종료(from 'Working Effectively with Unit Tests', Jay Fields)


- Solitary Unit Test
 - constraints
 - 절대 테스트하고 있는 클래스 외에 다른 객체는 테스트에 포함되지 않음
 - Class Under Test가 테스트에서 발견되는 유일한 구현 클래스(concrete class)여야 함
- Sociable Unit test
 - Solitary Unit Test로 분류될 수 없는 모든 Unit Test는 Sociable Unit Test
- 이러한 정의에 따라 이 책에서는 Solitary Unit Test가 하나도 없음
- Solitary와 Sociable Unit Test의 적절한 혼합이 최적의 해결책이라고 생각함

Two schools of TDD

- 상태를 조사(query, algorithm) / 상호작용을 조사(command, business requirements)
- Classic TDD or "London School" ?
 - Triangulation
- London vs Chicago
 - MOCK OBJECTS AND NEED-DRIVEN DEVELOPMENT
- Inside-out vs Outside-in TDD

msbaek/atdd-example

ATDD를 활용한 REST API Server 개발. Contribute to msbaek/atdd-example development by creating an account on GitHub.

 <https://github.com/msbaek/atdd-example/blob/master/README.md>



Test Double의 종류(xUnit Test Patterns: Refactoring Test Code)

The Little Mocker(<http://blog.cleancoder.com/uncle-bob/2014/05/14/TheLittleMocker.html>)

Test Double의 의미

- You mean like “Stunt Doubles” in the movies?
 - Exactly
- 그럼 왜 test double이라고 하지 않고 사람들이 mock이라는 단어를 많이 사용하나?
 - mock은 구어적 슬랭으로 test double을 의미하기도 하고
 - 공식적으로는 test double의 약어로 사용됨
- 그럼 왜 Test Double이라는 용어 대신 mock이라는 단어를 사용했나?
 - 역사가 있음

<https://www2.ccs.neu.edu/research/demeter/related-work/extreme-programming/MockObjectsFinal.PDF>

- 이 논문에서 나옴. 이 논문을 본 사람들이 Mock Object라는 단어를 사용하기 시작
 - 이 논문을 읽지 않은 사람들을 Mock이라는 단어를 듣고 더 광범위한 의미로 사용하기 시작
 - "Let's mock that"이 "Let's make a test double for that"보다 쉬운 구어적 표현임.

Test Double의 종류

- Dummy

```

interface Authorizer {
    public Boolean authorize(String username, String password);
}

public class DummyAuthorizer implements Authorizer {
    public Boolean authorize(String username, String password) {
        return null;
    }
}

////
public class System {
    public System(Authorizer authorizer) {
        this.authorizer = authorizer;
    }

    public int loginCount() {
        //returns number of logged in users.
    }
}

////
@Test
public void newlyCreatedSystem_hasNoLoggedInUsers() {
    System system = new System(new DummyAuthorizer());
    assertThat(system.loginCount(), is(0));
}

```

- 어떻게 사용될지에 관심 없는 어떤 객체를 어떤 메소드나 객체에 전달하는 경우
 - 예를 들면 테스트의 일부로 어떤 인자를 반드시 전달해야 하지만 인자가 절대로 사용되지 않는다는 것을 알고 있는 경우
- interface의 모든 메소드들이 `return null;` 로 구현된 테스트 더블
 - null을 반환하는 것이 에러가 아닌 경우
 - 사실 null은 dummy가 반환할 수 있는 최선의 값임
 - 왜냐하면 dummy를 누군가 사용하려 들면 NPE를 받게 됨
 - dummy가 사용되는 것을 원치 않으니 이 상황은 원하는 바임
- 이 메소드들이 호출되지는 않지만 구현체가 필요한 경우(호출된다면 NPE 발생)
- Stub

```

public class AcceptingAuthorizerStub implements Authorizer {
    public Boolean authorize(String username, String password) {
        return true;
    }
}

```

- dummy의 일종. 0이나 null 대신 테스트가 필요로 하는 특정 값(true)을 반환
 - 로그인에 필요한 시스템의 일부를 테스트하는 것을 가정해 보자
 - 이미 로그인이 잘 동작하는 것을 알고 있다. 다른 방법으로 로그인은 테스트했다. 다시 테스트할 필요가 있나?
 - 로그인 테스트는 쉽지 않나?
 - 그러나 시간이 걸리고 setup도 필요로 함. 만일 로그인에 버그가 있다면 당신의 테스트도 깨질 것임. 결국 불필요한 결합(coupling)임
 - 간단히 AcceptingAuthorizerStub을 테스트를 위해 시스템에 주입할 수 있음
 - 로그인 세션에 대한 stub을 구현하여 항상 true 반환
 - 만일 인증 안된 사용자에게 대한 테스트를 작성하고 싶다면 false를 반환하는 stub을 사용할 수 있음
- Spy

```
public class AcceptingAuthorizerSpy implements Authorizer {
    public boolean authorizeWasCalled = false;

    public Boolean authorize(String username, String password) {
        authorizeWasCalled = true;
        return true;
    }
}
```

- stub의 일종. 자신이 호출된 fact를 기억하고 후에 테스트에 이러한 fact를 보고
 - 어떤 함수가, 언제, 몇번, 어떤 인자로 호출되었는지 등
- 이게 test code와 production code 사이의 결합도를 증가시킴
 - spy를 많이 사용할 수록 테스트 코드는 프로덕션 코드의 구현에 더 많은 결합도를 갖게됨
 - 이로 인해 테스트가 잘 깨지게(fragile)됨
- fragile test란
 - 깨지지 않아야 하는 다른 이유로 깨지는 테스트
 - 프로덕션 코드를 변경하면 종종 테스트가 깨지지 않나?
 - 맞다. 하지만 잘 설계된 테스트는 깨지는 것을 최소화함. spy는 이에 반함 (프로덕션의 구체적인 사항에 의존하므로)

- Mock

```
public class AcceptingAuthorizerVerificationMock implements Authorizer {
    public boolean authorizeWasCalled = false;

    public Boolean authorize(String username, String password) {
        authorizeWasCalled = true;
        return true;
    }

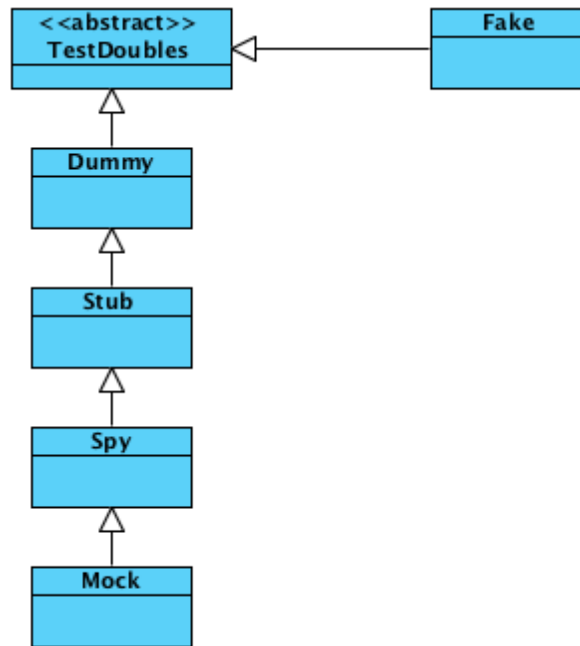
    public boolean verify() {
        return authorizeWasCalled;
    }
}
```

- spy의 일종. 어떤 일이 일어나야 하는지를 아는 spy
 - 호출이 일어났는지를 알려주는 `public boolean verify()` 메소드를 제공함
- test의 assert를 Mock의 verify 메소드로 옮김 느낌
 - 맞다. Mock은 자기가 무엇을 테스트하는 지를 안다
- Mock이 테스트하는 것은 행위임
 - Mock은 함수가 반환하는 결과에 관심이 없음
 - 이 보다는 어떤 함수가 호출되었고, 어떤 인자가 전달되었고, 언제/몇번이나 호출되었는지에 관심이 많음
- Mock은 항상 spy임
- 테스트 결과에 대한 검증(assertion)을 Mock으로 이동하는 것은 결합도를 증가시킴
- 그럼 왜 테스트 결과에 대한 검증을 Mock으로 이동하나?
 - mocking tool을 작성하기 쉽기 때문임
 - JMock, EasyMock, Mockito 등은 Mock 객체를 즉시 만들 수 있게 함
 - 이에 대한 마틴 파울러의 글:
<https://martinfowler.com/articles/mocksArentStubs.html>
 - 책: Growing Object Oriented Software, Guided by Tests

- Fake

```
public class AcceptingAuthorizerFake implements Authorizer {
    public Boolean authorize(String username, String password) {
        return username.equals("Bob");
    }
}
```

- 이 코드는 이상하다 Bob이라는 이름을 갖는 모든 사용자는 인증됨
 - 맞다. Fake는 비즈니스 행위를 갖음
 - 다른 데이터를 부여하여 다르게 동작하도록 새로운 Fake를 만들 수 있음
- simulator와 같다
 - simulator는 fake임
 - simulator란
 - 복잡한 작동 상황 등을 컴퓨터를 사용하여 실제 장면과 같도록 재현하는 장치. 항공기의 조종, 원자로 운전 등의 훈련이나 시험 연구 등에 사용됨.
- fake는 Stub이 아님
 - Fake는 stub과 달리 실제 비즈니스 행위를 가짐
 - fake 외의 다른 test double들은 비즈니스 행위를 갖지 않음
- fake는 복잡해 질 수 있음
 - 극도로 복잡해 질 수 있음
 - 너무 복잡해져서 fake는 자신 만의 unit test를 필요로 함
 - 궁극적으로는 fake는 real system이 됨
 - 앙클밥은 2014년 30년 동안 fake를 만들지 않았다고 함
- 복잡도와 유지보수를 위해서 가급적 피해야 함



- 어떤 test double을 사용하나
 - 주로 stub과 spy를 사용함. 직접 작성함. mocking tool은 거의 사용하지 않음
 - dummy를 사용하나 ?
 - 드물게 사용함
 - mock도 사용하나 ?
 - mocking tool을 사용할 때만 사용함. 하지만 mocking tool을 사용하지 않으므로 거의 사용하지 않음
- 왜 mocking tool을 사용하지 않나 ?
 - stub, spy는 만들기 쉬움.
 - interface에 대한 구현체를 IDE가 쉽게 만들어 줌 → Dummy를 얻게됨
 - 필요하면 단순한 변경을 해서 stub이나 spy를 만듦
 - 그래서 mocking tool을 사용할 필요가 거의 없음
 - 코드의 가독성도 mockito 보다 내가 만든 test double이 유리함

Mock Roles, not Objects(<http://jmock.org/oopsla2004.pdf>)

OOP

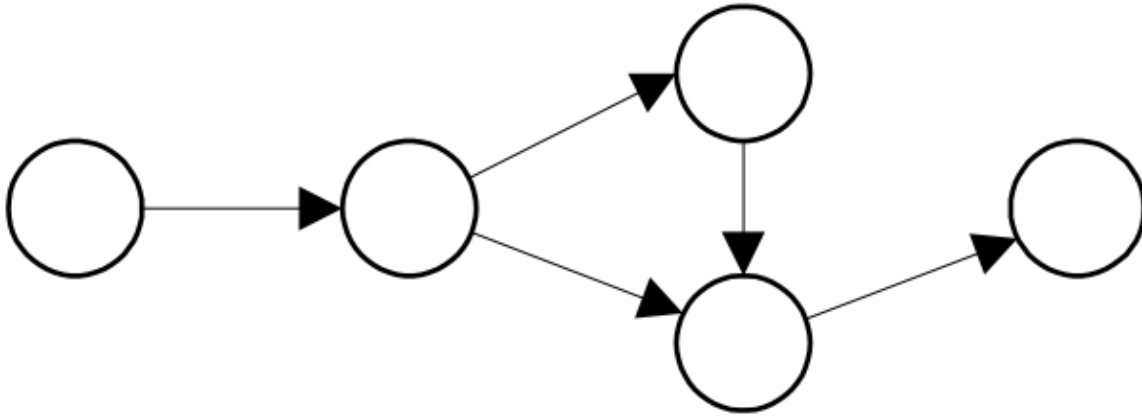


Figure 1. A Web of Collaborating Objects

MOCK OBJECTS AND NEED-DRIVEN DEVELOPMENT

- Mock 객체를 활용한 TDD는 객체가 필요로 하는 서비스를 위한 인터페이스 설계를 가이드함
 - 소비자가 필요한 것을 발견. 필요할 것이라고 예상되는 것을 생산자 관점에서 제공하는 것이 아니라
- Need-Driven Development
 - 이러한 접근법은
 - 클래스가 제공할 수 있는 모든 기능을 기술하는 인터페이스가 아니라
 - 필요한 것만 제공해서 시스템이 적은 인터페이스를 제공하게 함
 - 각 인터페이스는 객체 간의 상호작용에서 역할을 정의
- Figure2는 A 객체의 테스트를 표현함

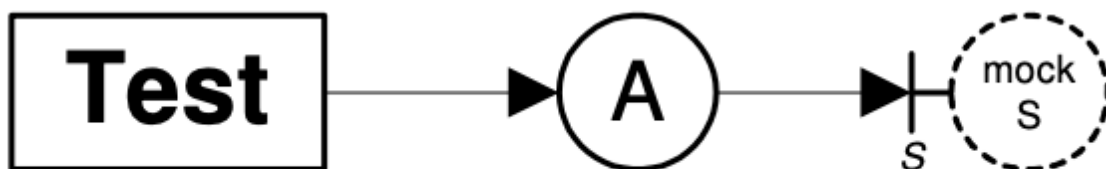


Figure 2. Interface Discovery

- A의 요구사항을 만족시키기 위해 S라는 서비스가 필요함을 발견함
- A를 테스트하는 동안 S의 책임을 mock함(S 인터페이스를 구현하는 대신)
 - 서비스를 구현할 때 다른 서비스나 리파지토리가 발견되면 Interface만 정의하고 Mocking하고 진행. context-switching 제거. narrow interface, 좋은 설계 가능
- 요구사항을 만족시키기 위해 A를 구현하고 나면 S의 역할을 수행할 객체를 구현으로 넘어갈 수 있음
- Figure 3의 B객체가 S의 구현 객체임

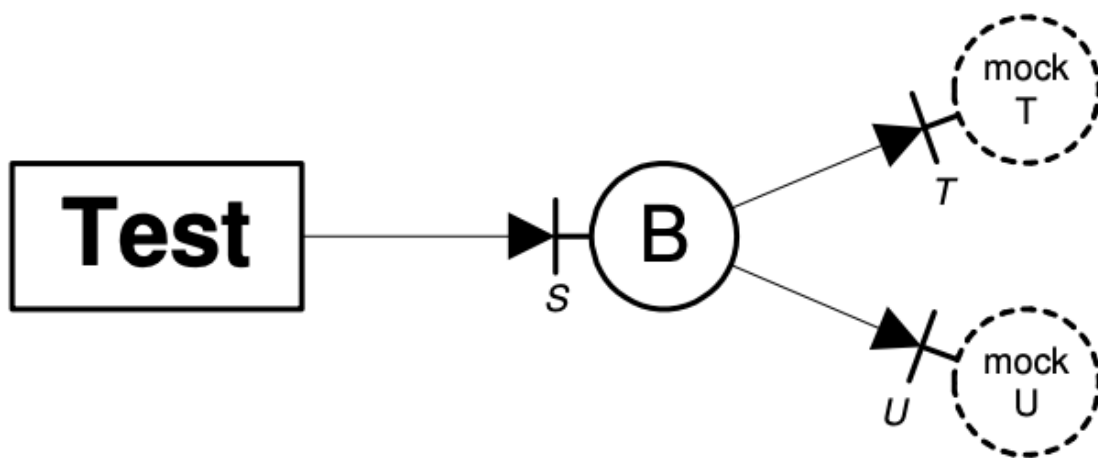


Figure 3. Iterative Interface Discovery

- 이 과정에서 다시 B가 필요로하는 T, U 인터페이스를 발견하게 됨. 이때 B의 구현을 마칠 때까지 T, U는 mocking됨
- 시스템 런타임이나 외부 라이브러리를 사용하여 실제 기능을 구현해야 하는 계층에 도달할 때까지 이 과정을 계속함
- 최종 결과는 좁은 인터페이스를 통해 서로 통신하는 객체들의 조합으로 구조화된 애플리케이션임(Figure 4)

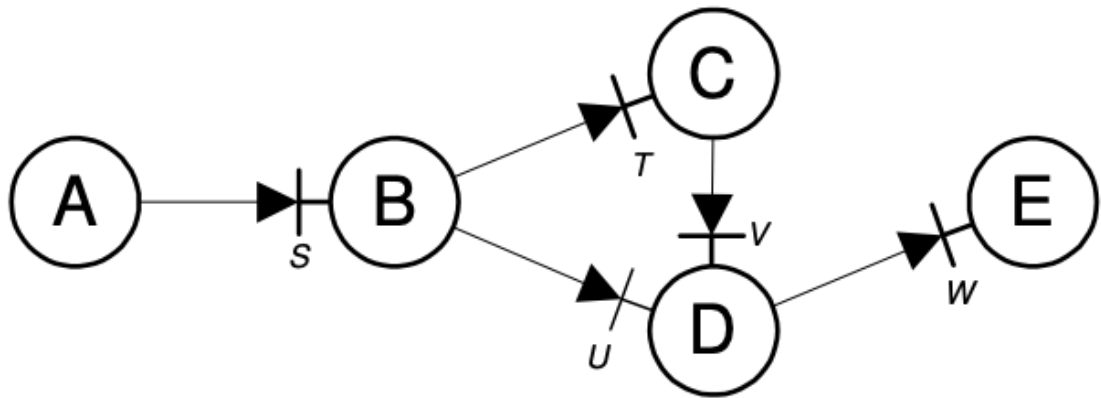


Figure 4. A Web of Objects Collaborating Through Roles