Contents

[1. Giới thiệu về Test Driven Development 2](#_Toc456951156)

[1.1. Tổng quan về TDD 2](#_Toc456951157)

[1.1.1. So sánh TDD với phương pháp phát triển phần mềm truyền thống 2](#_Toc456951158)

[1.2. Quá trình phát triển theo TDD 3](#_Toc456951159)

[1.3. Ưu điểm của việc áp dụng TDD và các vấn đề thường gặp 3](#_Toc456951160)

[1.3.1. Ưu điểm 3](#_Toc456951161)

[1.3.2. Các câu hỏi thường gặp khi phát triển TDD 4](#_Toc456951162)

[1.4. Các thuật ngữ sử dụng 4](#_Toc456951163)

[1.5. Các framework hỗ trợ 4](#_Toc456951164)

[1.5.1. Java testing framework 4](#_Toc456951165)

[1.6. Kết luận 4](#_Toc456951166)

[2. Tổng quan về JUnit và cài đặt thử nghiệm theo Red/Green/Refactor 6](#_Toc456951167)

[2.1. Tổng quan về JUnit 6](#_Toc456951168)

[2.1.1. Các annotation và class sử dụng trong JUnit 6](#_Toc456951169)

[3. Giới thiệu về Mock và Mockito 8](#_Toc456951170)

[3.1. Mockito 8](#_Toc456951171)

[3.2. Các chức năng chính của Mokito 8](#_Toc456951172)

[3.2.1. Argument Matching 8](#_Toc456951173)

[3.2.2. Stubbing consecutive calls 9](#_Toc456951174)

[3.2.3. Verify Order 9](#_Toc456951175)

[3.2.4. Capturing Arguments 9](#_Toc456951176)

[3.3. PowerMock 10](#_Toc456951177)

**TEST-DRIVEN DEVELOPMENT (TDD)**

# Giới thiệu về Test Driven Development

Trong chương này ta cần nắm được các phần như sau:

* Hiểu được tổng quan về TDD và các lợi ích thu được khi áp dụng TDD vào quá trình phát triển phần mềm
* Biết cách xây dựng 1 module theo TDD bằng phương pháp: Red/Green/Refactor
* Một số testing framework khi phát triển theo TDD

## Tổng quan về TDD

Test-Driven Development là quy trình phát triển phần mềm hướng kiểm thử, được hiểu là developer sẽ viết test case trước cho một module, sau đó sẽ lập trình để đảm bảo chương trình chạy qua được test case đó.

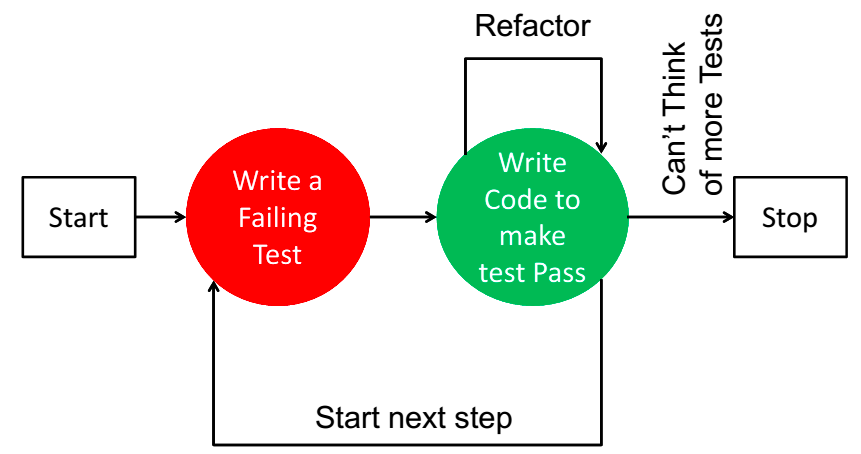
Như vậy phương pháp phát triển này sẽ đảm bảo được:

* Việc kiểm thử liên tục trên code
* Sau khi lập trình sẽ đưa ra kết quả mong muốn trên từng đoạn code.

### So sánh TDD với phương pháp phát triển phần mềm truyền thống

|  |  |  |
| --- | --- | --- |
|  | **Traditional Approach** | **Test-Driven Development** |
| *Developer* | Lập trình theo bản thiết kế, đôi khi không hiểu nghiệp vụ, mà chỉ cần đảm bảo input và output theo yêu cầu đưa ra | Cần tìm hiểu trước nghiệp vụ, có cái nhìn bao quát hơn về chức năng cần xây dựng để viết test case cho chức năng đó. |
| *Unit test* | Thủ công và thường thiếu xót hoăc bỏ qua việc unit test khi phát triển | Được thực hiện tự động và liên tục |
| *Automation test* | Không áp dụng | Áp dụng |
| *Khi thay đổi hoặc thêm mới chức năng* | Phải kiểm thử lại toàn bộ hệ thống, không kiểm soát hết được ảnh hưởng của việc thay đổi | Chỉ cần viết test case cho phần thay đổi, kiểm soát được ảnh hưởng khi thay đổi |
| Lãng phí nguồn lực khi kiểm thử lặp đi lặp lại một chức năng khi có bất kỳ thay đổi nào | Chỉ cần kiểm thử riêng phần thay đổi |
| *Bảo trì hệ thống* | Lãng phí nguồn lực và thời gian khi hệ thống liên tục thay đổi và scale | Bảo trì đơn giản |

## Quá trình phát triển theo TDD



*Hình 1.1****:*** *Quá trình áp dụng TDD trên 1 module*

Việc xây dựng một chức năng khi áp dụng TDD như sau:

* ***Trước khi code (RED):***
  + Xây dựng test case cho chức năng mà mình cần làm, xây dựng input và output mong muốn
  + Chạy thử test case và đảm bảo là test case **FAIL** (vì chức năng chưa được lập trình, nên test case fail)
* ***Thực hiện code (GREEN):***
  + Lập trình theo input và output như trên, đảm bảo pass được qua test case. (nếu quá trình viết test case sai, thì có thể quay lại viết lại test case)
* ***Sau khi code (REFACTOR):***
  + Tối ưu đoạn code trên tới khi thấy hợp lý, sau đó kết thúc

## Ưu điểm của việc áp dụng TDD và các vấn đề thường gặp

### Ưu điểm

Lợi ích khi áp dụng TDD:

* Áp dụng được việc kiểm thử tự động vào việc lập trình, việc kiểm thử được diễn ra liên tục ngay trong quá trình phát triển để đảm bảo đầu ra chính xác.
* Kiểm soát được ảnh hưởng khi có thay đổi trên hệ thống, sẽ ngăn được việc kiểm thử lặp đi lặp lại với các chức năng, ngay cả khi chức năng đó không bị ảnh hưởng.
* Lập trình viên sẽ có cái nhìn bao quát hơn về hệ thống, hiểu được chính xác nghiệp vụ cần làm gì, để xây dựng Unit test
* Phần code luôn chạy đúng và đủ để đảm bảo chạy qua được test case trong quá trình phát triển từng chức năng.

### Các câu hỏi thường gặp khi phát triển TDD

Các câu hỏi thường gặp đối với việc áp dụng TDD:

* ***Phát triển theo TDD gây lãng phí vì công việc lập trình phải thực hiện cả code và test***: Thực tế việc phát triển theo TDD có thể tốn chi phí trong thời gian đầu mới phát triển vì việc lập trình cần phải thực hiện viết test case và function, nhưng TDD lại thể hiện lợi ích lớn trong quá trình bảo trì và phát triển mở rộng sau này.
* ***Viết test case sau khi lập trình***: Việc viết test phải đảm bảo viết trước khi lập trình, và test case có input và output rõ ràng, chính xác theo chức năng cần phát triển. Thường các lập trình viên hay viết test case sau khi phát triển, để đảm bảo test case chính xác với kết quả đã code như vậy là không chính xác.

## Các thuật ngữ sử dụng

* **xUnit**: tập hợp tên của các chuẩn unit test tùy với từng ngôn ngữ, “x” thường được thay bằng chữ cái đầu của ngôn ngữ lập trình như: JUnit cho Java, CUnit cho C
* **Class-Under-Test (or CUT)**:
* **Method-Under-Test (or MUT)**:
* **Test Fixture**:

## Các framework hỗ trợ

### Java testing framework

* Mockito
* DBUnit
* PowerMockito

## Kết luận

Khi hệ thống mới được phát triển, có thể TDD không thể hiện được rõ ràng lợi ích vì số lượng chức năng còn ít và đơn giản. Nhưng khi các module được liên kết với nhau, việc thay đổi 1 module có thể ảnh hưởng tới cả hệ thống, thì TDD lại đem lại giá trị rất lớn trong việc phát hiện được ảnh hưởng tới các chức năng khác

TDD không sử dụng để thay thế cho quá trình kiểm thử truyền thống, mà chỉ có tác dụng trong quá trình unit test khi lập trình và đảm bảo đem lại kết quả chính xác sau khi lập trình.

Trong các hệ thống lớn hay các hệ thống có yêu cầu thay đổi liên tục đều gặp các vấn đề như sau:

* Với các hệ thống lớn, việc thay đổi một chức năng nhỏ dẫn đến kiểm thử cả hệ thống lớn => Tốn nguồn lực và thời gian
* Với các hệ thống có yêu cầu thay đổi liên tục, ví dụ như Google, hàng ngày các chức năng mới liên tục được phát triển, nếu như quy trình phát triển thông thường, việc kiểm thử một phiên bản có thể chưa xong thì phiên bản mới hơn đã được đưa ra => Không kiểm soát được hệ thống khi thay đổi.

Như vậy việc áp dụng TDD trong quá trình phát triển phần mềm, sẽ giảm thiểu được nguồn lực và thời gian cho quá trình phát triển và bảo trì về sau này. Đặc biệt với các hệ thống lớn và có yêu cầu thay đổi thường xuyên thì TDD thể hiện rất rõ ưu điểm của mình.

# Tổng quan về JUnit và cài đặt thử nghiệm theo Red/Green/Refactor

## Tổng quan về JUnit



JUnit là một test framework sử dụng cho ngôn ngữ Java, xây dựng theo xUnit architecture, và sử dụng annotation để định nghĩa một đoạn test sử dụng cho function cụ thể nào đó.

Một quá trình JUnit Testing diễn ra trong vòng 4 bước như sau:



* **Setup**: Xác định chức năng cần test và thiết lập các điều kiện cần để chạy test UUT (Unit Under Test) ví dụ như: setup dữ liệu, mở connect database.
* **Execution:** Cài đặt các UUT để chạy đoạn test.
* **Verification**: Đảm bảo rằng input và output là chính xác.
* **Teardown**: Sau khi chạy xong test, bước này thực hiện xóa dữ liệu và các tài nguyên sử dụng cho đoạn test như database connection.

### Các annotation và class sử dụng trong JUnit

**Class:**

|  |  |
| --- | --- |
| **Assert** | Tập các phương thức xác nhận được sử dụng khi viết test |
| **Assume** | Tập các phương thức giả định yêu cầu về các điều kiện để đoạn test đó có nghĩa |
| **Test.None** | Mặc định exception rỗng |

**Annotation**:

|  |  |  |  |
| --- | --- | --- | --- |
| **Annotation** | **Step** | **Ý nghĩa** | **Ví dụ** |
| **After** | **Teardown** | Định nghĩa phương thức chạy sau khi đoạn test thực hiện | Xóa data, restore trạng thái |
| **AfterClass** | **Teardown** | Định nghĩa phương thức chạy duy nhất 1 lần, sau tất cả các bài test | Disconnect database |
| **Before** | **Setup** | Định nghĩa phương thức chạy trước mỗi đoạn test, chuẩn bị môi trường để test | Read data, khởi tạo class |
| **BeforeClass** | **Setup** | Định nghĩa phương thức được chạy duy nhất 1 lần, trước tất cả các bài test. Thường để khai báo các hoạt động không có thay đổi theo thời gian. | Tạo connection tới database |
| **Ignore** | **Executing** | Bỏ qua phương thức test. |  |
| **Test** | **Executing** | Định nghĩa phương thức test |  |

# Giới thiệu về Mock và Mockito

Khi cài đặt Unit test ta thường gặp các vấn đề như cần sử dụng dữ liệu thực mà dữ liệu chưa được khởi tạo, hay gọi đến các method nhưng các method này chưa được implement. Vì vậy, khi thiết lập các method ta cần cài đặt giả lập dữ liệu cũng như các method này, được gọi là mock, để cài đặt unit test.

Việc sử dụng mock để giả lập giúp:

* Cài đặt các function trong thời gian ngắn để trả lại các kết quả mong muốn mà không cần phải cài đặt một function thực có thể không chính xác và tiêu tốn nguồn lực
* Tránh việc tạo ra các class không mong muốn khi chưa sử dụng đến

Một trong những thư viện hỗ trợ giả lập là Mockito, JMock… Dùng để giả lập class, method để đưa ra kết quả mong muốn. Trong nội dung tài liệu sẽ tập trung vào Mockito.

## Mockito

Mockito hỗ trợ chuẩn cho 1 chu trình Unit test 4 bước: Setup, Executing, Verification và Teardown.

## Các chức năng chính của Mokito

### Argument Matching

Thông thường khi so sánh các đối tượng hay giá trị, ta thường sử dụng các hàm: equal(), “==”, “<=”,… Đối với Mokito thì việc so sánh mang tính trừu tượng hơn, sử dụng Matchers, để xem chi tiết các hàm của Matchers, tham khảo link sau: <http://hamcrest.org/JavaHamcrest/javadoc/1.3/org/hamcrest/Matchers.html>

Dưới đây là một số hàm thường được sử dụng:

|  |  |
| --- | --- |
| **Core** | |
| **any()** | assertThat(new Canoe(), instanceOf(Canoe.class)); |
| **is()** | assertThat("String", is("String")) |
| **describedAs()** | describedAs("a big decimal equal to %0", equalTo(myBigDecimal), myBigDecimal.toPlainString()) |
| **Logical** | |
| **allOf()** | assertThat("myValue", allOf(startsWith("my"), containsString("Val"))) |
| **anyOf()** | assertThat("myValue", anyOf(startsWith("foo"), containsString("Val"))) |
| **not()** | assertThat(cheese, is(not(smelly))) |
| **Object** | |
| **equalTo()** | assertThat("foo", equalTo("foo")); |
| **instanceOf()** | assertThat(new Canoe(), instanceOf(Paddlable.class)); |
| **notNullValue()** | assertThat(cheese, is(notNullValue())) |
| **nullValue()** | assertThat(cheese, is(nullValue())) |

### Stubbing consecutive calls

Khi xây dựng một chức năng, ta thường phải sử dụng để gọi tới hàm của đối tượng khác. Tuy nhiên, các hàm này không có sẵn, hoặc hiện đang được phát triển nhưng chưa hoàn thiện. Vậy nên sử dụng mockito cho phép tạo ra các hàm nhánh giả mà ta cần sử dụng và các hàm này sẽ đưa ra kết quả mong muốn khi gọi tới chúng.

### Verify Order

Việc kiểm thử theo kết quả trả về chỉ kiểm tra được hàm đó có thực hiện đúng hay không, chứ chưa xác định được những công việc hàm đó thực hiện, và thứ tự công việc thực hiện chúng ra sao.

Liên tưởng tới một ví dụ thực tế, điều này giống ta nhận được một kết quả như: “Cô ấy đã đi ra ngoài rồi”, nhưng ta không rõ về việc cô ấy đã dọn dẹp trước khi đi chưa, hay cô ấy đã tắt điện trước khi đi chưa.

Mockito có thể giúp ta xác định được chính xác những công việc đó. Mockito có thể kiểm tra kết quả của phương thức đưa ra, có sử dụng tương tác với các đối tượng khác hay không, tương tác như thế nào.

### Capturing Arguments

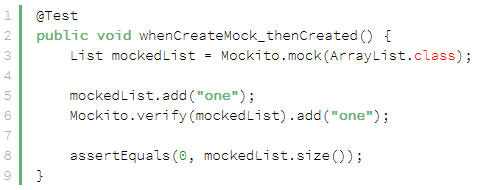
Khi thực hiện kiểm thử, một câu hỏi đặt ra tiếp theo đó là, làm sao để ta biết kết quả của phương thức đó chính xác khi mà phương thức không trả về kết quả. Trường hợp này rất hay xảy ra với các hàm void.

Tuy nhiên Mockito cũng hỗ trợ ta validate các hàm không trả về kết quả trực tiếp bằng cách sử dụng ArgumentCaptor

### Spy

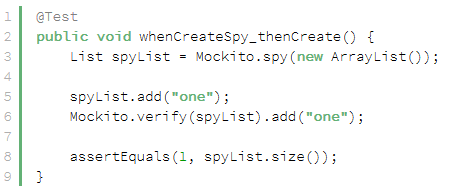
Sử dụng Mock phần lớn có thể bao phủ được tất cả các test case, tuy nhiên trong một số trường hợp vẫn sử dụng Spy. Spy có tác dụng tương đương với mock dùng để giả lập một đối tượng và phương thức, nhưng Mock và Spy khác nhau là Mock chỉ giả lập đối tượng dựa trên class hoặc type của đối tượng giả lập. Còn Spy được giả lập dựa trên instance của đối tượng. Vậy nên Spy có tất cả các đặc điểm như một đối tượng thật. Để rõ hơn về việc này có 1 ví dụ như sau:

Đây là cách để mock một đối tượng. Khi thực hiện thêm 1 phần tử vào List, thì list đó không thực sự thêm phần tử vào List. Khi call một method của mock, thì mock sẽ gọi tới method giả lập, chứ không gọi tới method thực, mock chỉ trả về giá trị khi đã được giả lập method và giá trị return mong muốn.



*mockedList.size() = 0, khi mock đã được add “one”*

Còn đây là spy, spy được tạo nên từ hàm khởi tạo của đối tượng, có đủ các đặc tính như đối tượng đó và method tương. Khi call tới method, thì spy sẽ thực hiện trên method thực.



*spyList.size() = 1, khi mock đã được add “one”*

### PowerMock

Vì vậy dẫn đến việc sử dụng PowerMock kết hợp với Mockito (hoặc EasyMock) để có thể bao phủ được

* Không mock được các phương thức static, final và private
* Bỏ qua các phương thức khởi tạo được override

Vì vậy dẫn đến việc sử dụng PowerMock kết hợp với Mockito (hoặc EasyMock) để có thể bao phủ được tất cả các test case

# Fixture management và kiểm thử với dữ liệu

## Fixture management

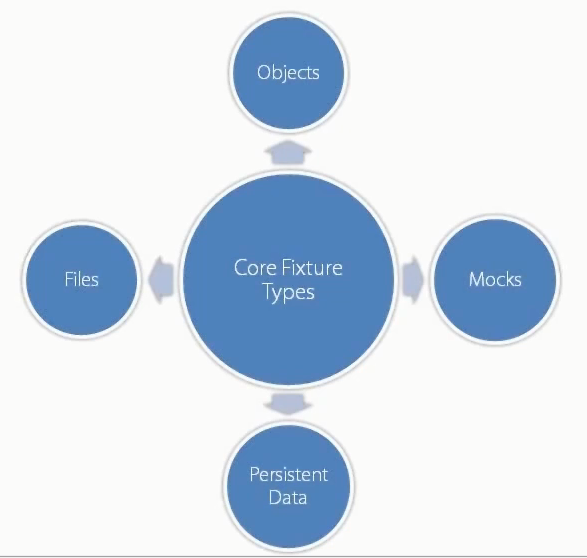
Khi thực hiện 1 unit test, ta cần theo những công việc sau đây để đảm bảo bài test được độc lập và chính xác, không ảnh hưởng tới dữ liệu đang có.

### Thực hiện clean và run code độc lập

* **Setup:** Khởi tạo những điều kiện cần thiết để thực hiện bài test
  + Đối tượng object truyền vào các method
  + Định nghĩa các đối tượng mock hay các đối tượng được mock trả về
  + Thêm dữ liệu vào trong database trong trường hợp test truy cập data
  + Khởi tạo file trong trường hợp sử dụng tới file
* **Teardown:** Clean dữ liệu và các đối tượng không được JVM Garbage Collector thu dọn
  + Xóa data insert trong database khi thực hiện test
  + Xóa file được khởi tạo ra khi chạy test, hoặc file tạo ra để phục vụ việc test

### Quản lý các fixture

Khi thực hiện unit test ta cần chú ý trong vấn đề quản lý fixture.



Với 1 đối tượng, khi cần unit test tới đối tượng nào thì ta mới setup dữ liệu cho đối tượng đó và setup những dữ liệu cần thiết sử dụng cho test. Việc setup tất cả dữ liệu dẫn tới mất thời gian và dư thừa không cần thiết. Khởi tạo các biến độc lập với mỗi test case, không tái sử dụng qua mỗi lần test. Việc này có thể dẫn tới kết quả test không chính xác.

Không sử dụng các data có sẵn, hay sử dụng lại data đã test, việc test với data cần tạo các bộ dữ liệu mới, và đảm bảo dữ liệu được clean sau mỗi lần test.

Không nên sử dụng chung database, vì có thể sẽ dẫn tới viêc thay đổi dữ liệu, mỗi developer nên có 1 database riêng để lập trình và test. Cần thực hiện setup và teardown với mỗi unit test.

Việc test với file cũng cần đảm bảo như data và đảm bảo các file được clean sau mỗi lần test.

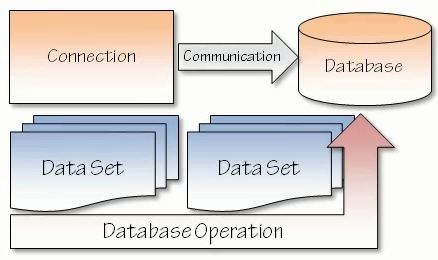
## Tổng quan về DBUnit

Thông thường với chức năng thêm một bản ghi vào database thì lập trình viên, để unit test được chức năng chạy đúng hay không thì cần kiểm tra bản ghi có được insert vào database chính xác hay không.

Cũng như việc lấy một bản ghi từ database ra, người dùng cũng phải kiểm tra bản ghi đó có tồn tại hay không, hoặc so sánh bản ghi lấy ra có chính xác hay không.

Để đảm bảo chính xác và không ảnh hưởng tới dữ liệu, developer cần phải tạo ra các bản ghi riêng để thực hiện việc kiểm thử và công việc này thường khá mất thời gian khi chức năng chưa chính xác, lập trình viên phải thao tác nhiều lần, và phải tạo các bản ghi mới liên tục để thực hiện test chính xác. Khi phát triển chức năng xong, developer cần phải xóa toàn bộ dữ liệu vừa tạo ra khi test. Việc kiểm thử với database trở nên khá mất công.

Tuy nhiên, DBUnit có thể hỗ trợ được phần lớn các công việc này. DBUnit gồm 3 phần:



* Database Connection: - IDatabaseConnection: Quản lý connection tới Database
* Dataset management – IDataset: quản lý dataset, hỗ trợ đọc data từ các file Excel, XML, CSV.
* Database Operation – DatabaseOperation: Quản lý các câu điều khiển tới Database như INSERT, CLEAN\_INSERT, DELETE…