ĐẠI HỌC QUỐC GIA TP.HỒ CHÍ MINH
ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

Lập trình Java

Lab 08 – JUnit

Version 1.0

Bộ môn Công Nghệ Phần Mềm
Khoa Công Nghệ Thông Tin
Đại học Khoa Học Tự Nhiên TP.HCM

Tp. Hồ Chí Minh, tháng 10 năm 2019

Mục lục

Danh mục hình

1

Qui định nộp bài

1.1 Qui định đặt tên file nộp bài

- Tạo **MSSV_LabXY** với **XY** là mã tuần *(mã Lab)*.
- Mỗi bài tập sẽ tạo file có định dạng **ExerciseNM.java** với **MN** là *mã bài tập* ở phần 2.
- Nén thư mục trên thành file zip/rar có định dạng **MSSV_LabXY.zip/rar**.
- Nộp file nén zip/rar ở link nộp bài trên Moodle.
- Trong trường hợp file lớn 10M quy định thì các em upload lên Drive hoặc Dropbox sau đó tạo file **MSSV_AssXY.txt**, dáng link của Drive hoặc Dropbox vào file này. Sau đó nộp file **MSSV_AssXY.txt** lên Moodle.

1.2 Lưu ý

- Các bài làm không đúng quy định không chấm.
- Các hình thức làm bài thực hành (lab) đều là bài làm cá nhân.
- Các bài giống nhau sẽ bị điểm 0đ toàn môn.

2

Giới thiệu JUnit[1]

2.1 Unit-test

2.1.1 Unit-test là gì?

- Trong kiểm thử phần mềm có 04 mức độ kiểm thử:
  - Unit test – kiểm thử đơn vị.
  - Intergration test – kiểm thử tích hợp.
  - System test – kiểm thử hệ thống.
  - Acceptance test – kiểm thử chấp nhận.
- Unit-test là mức độ kiểm thử nhỏ nhất trong quy trình kiểm thử.
- Unit-test sẽ kiểm thử các đơn vị nhỏ nhất trong mã nguồn như method, class, module… Do đó, Unit-test nhằm kiểm tra mã nguồn (White-box) của các chương trình, các chức năng riêng lẻ hoạt động có đúng với đặc tả/yêu cầu hay không.
- Unit-test được thực hiện bởi các ***Lập trình viên***.

2.1.2 Lợi ít của Unit-test

- Unit-test tốt sẽ tăng sự tin tưởng vào mã nguồn được thay đổi khi bảo trì. Do nếu viết Unit-test tốt, thì mỗi lần có sự thay đổi trong mã nguồn thì chúng ta chỉ cần run lại Unit-test thì chúng ta có thể bắt được những lỗi sảy ra do thay đổi mã nguồn.
- Chúng ta có thể kiểm thử các thành phần riêng lẻ của project mà không cần đợi các thành phần khác hoàn thành.

[1] Reference: http://www.softwaretestingstuff.com/2010/09/unit-testing-best-practices-techniques.html

- Do thực hiện test (kiểm thử) trên từng đơn vị nhỏ của các module riêng lẻ nên khi phát hiện lỗi thì cũng dễ khoanh vùng và sử chữa.
- Có thể tái sử dụng mã nguồn.
- Chi phí cho việc sửa chữa lỗi trong giai đoạn Unit-test sẽ ít hơn so với các giai đoạn sau.

2.1.3 Một số lưu ý khi viết Unit-test:

- Phải chắc chắn mỗi Test-case là kiểm thử mức đơn vị độc lập với những Test-case khác. Không nên gọi (call) một Test-case khác trong một Test-case. Test-case không nên phụ thuộc vào nhau cả về Data lẫn thứ tự thực hiện.
- Đặt tên các Unit-test phải rõ ràng và nhất quán. Đảm bảo các test-case phải dễ đọc.
- Khi triển khai việc thay đổi giao diện hoặc chức năng, cần chạy lại các test-case trước đó nhằm đảm bảo việc thay đổi này không ảnh hưởng đến những test-case cũ đã pass.
- Luôn đảm bảo lỗi được xác định trong quá trình Unit-test được sửa trước khi chuyển sang giai đoạn test tiếp theo.
- Không cố gắn viết test-case để kiểm thử tất cả mọi thứ, thay vào đó nên tập trung việc kiểm thử việc ảnh hưởng của hành vi hệ thống.
- Bên cạnh viết test-case để kiểm thử hành vi hệ thống, cần viết them test-case để kiểm thử hiệu năng của mã nguồn.
- Các Unit-test nên đặt riêng ra, đọc lập code với module.
- Không nên có nhiều Assert (tính đúng đắng) trong một test-case vfi khi một điều kiện không thỏa mãn thì các Assert khác sẽ bị bỏ qua.
- Sau một thời gian dài, số lượng test-case nhiều, thời gian chạy lớn. Nên chia ra nhóm test-case cũ và test-case mới, test-case cũ sẽ chạy với tầng xuất ít hơn.

2.1.4 Tiêu chí đánh giá một Unit-test tốt

- Chạy nhanh.
- Chạy độc lập giữa các Test-case, không phụ thuộc vào thứ tự kiểm thử.
- Sử dụng data dễ đọc, dễ hiểu.

- Sử dụng dữ liệu thực tế có thể.
- Test-case đơn giản, dễ đọc, dễ bảo trì.
- Phản ảnh đúng hoạt động của module.

2.2 JUnit

2.2.1 Lịch sử phát triển JUnit

- JUnit là một framework đơn giản được sử dụng cho việc tạo và kiểm thực các Unit-Test một cách tự động và chạy test, có thể lặp đi lặp lại. Nó chỉ là một phần của họ kiến trúc xUnit cho việc tao ra các Unit-Testing. JUnit là một chuẩn trên thực tế cho unit-testing trong Java. JUnit được phát triển bởi 2 tác giả Erich Gamma và Kent Beck.
- Vào giữa những năm 90 của thế kỷ 20, Kent Beck đã phát triển một bộ test xUnit đầu tiên cho SmallTalk.
- Beck và Gamma phát triển JUnit trên một chuyến bay từ Zurich đến Washington DC. Từ đó, JUnit trở thành công cụ chuẩn cho Test-Driven Development trong Java. Ngày nay, JUnit được tích hợp sẵn trong các Java IDEs như Eclipse, JBuilder …
- JUnit được phát hành trực tiếp tại trang chủ https://junit.org/junit5/

2.2.2 Những đặc điểm JUnit

- Assert – xác nhận việc kiểm tra kết quả được mong đợi.
- Các Test-Suite cho phép chúng ta dễ dàng tổ chức và chạy các test.
- Hỗ trợ giao diện đồ họa và giao diện dòng lệnh:
    - Các test-case của JUnit là các lớp của Java, các lớp bao gồm một hay nhiều các phương thức unit-testing và những test này lại được nhóm thành các Test-Suite.
    - Mỗi phương thức test trong JUnit phải được thực thi nhanh chóng. Tốc độ là điều tối quan trọng vì càng nhiều Test-case được viết và tích hợp vào bên trong quá trình xây dựng phần mềm thì cần phải tốn nhiều thời gian hơn cho việc chạy toàn bộ Test-Suite.
    - Các test-case trong JUnit có thể là các test-case quan trọng được chấp nhận hay thất bại, các test-case này được thiết kế để khi chạy mà không có sự can thiệp của con người.

- Từ những thiết kế như thế, chúng ta có thể them các bộ test-case vào quá trình tích hợp và xây dựng phần mềm một cách liên tục để các test-case chạy một cách tự động.

2.2.3 Kiến trúc tổng quan của JUnit

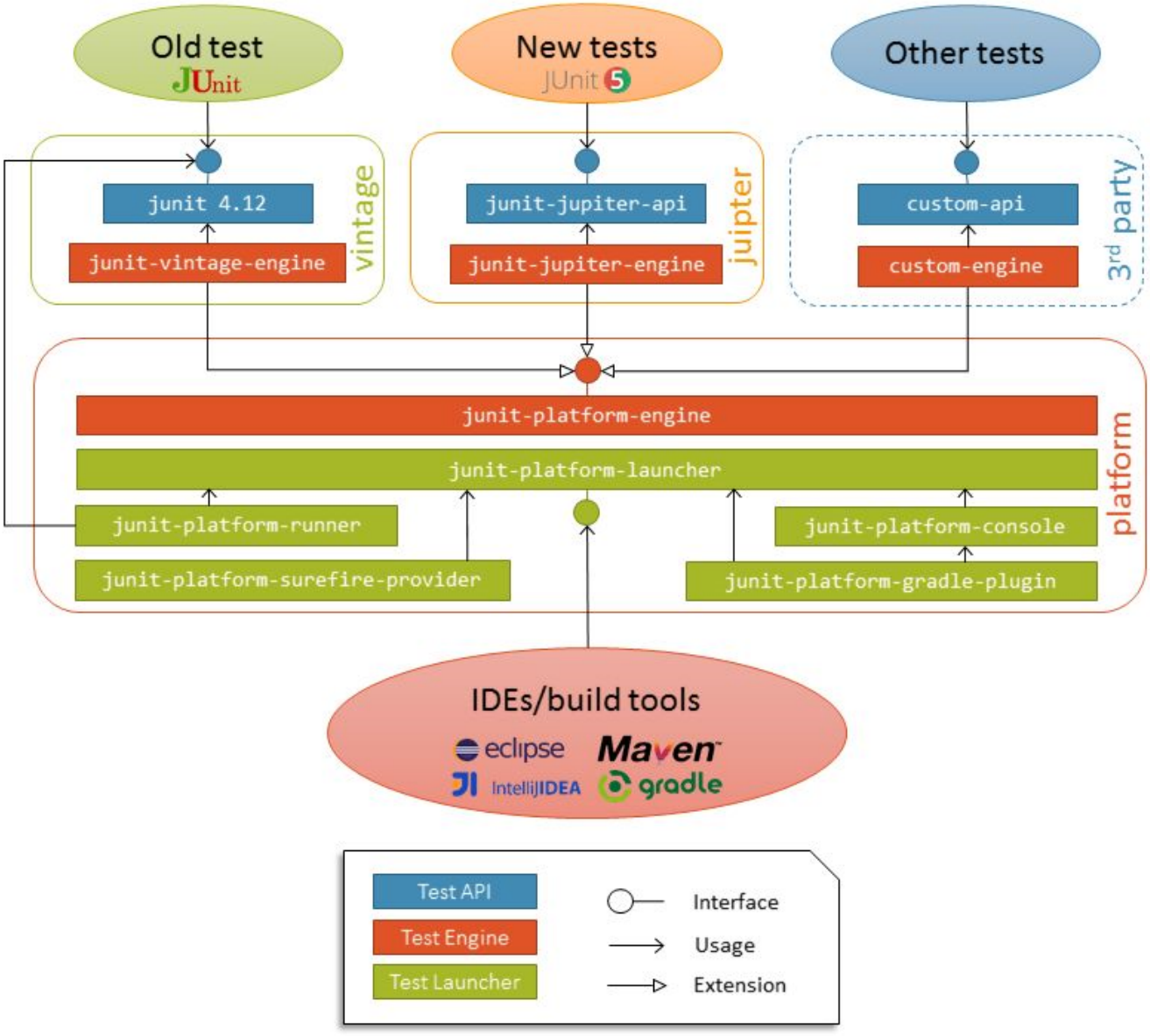


Figure 2-1 Kiến trúc chi tiết của JUnit 5

- Figure 2-1 là kiến trúc trúc chi tiết của framework JUnit 5. Qua mô hình này, cho thấy JUnit hỗ trợ việc tích hợp các Test-case được phát triển ở các JUnit version trước đó cũng như hỗ trợ việc tích các test-case của Third-party như custom-api …

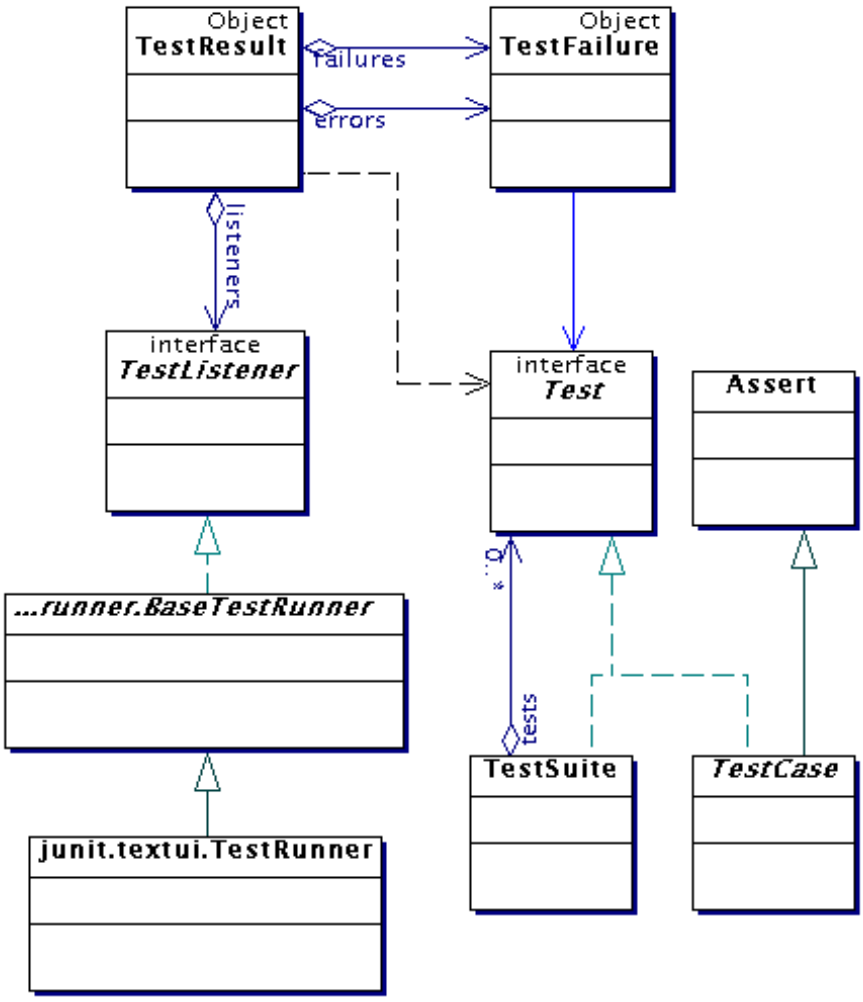


Figure 2-2 Kiến trúc tổng quan của JUnit

- JUnit test framework cung cấp cho chúng ta các gói lớp có sẵn cho phép chúng ta viết các phương thức test một cách dễ dàng.
- TestRunner sẽ chạy các test và trả về kết quả là các Test Results.
- Các lớp của chương trình test chúng ta sẽ được kế thừa các lớp trừu tượng TestCase.
- Khi viết các Test Case chúng ta cần biết và hiểu lớp Assert class.
- Một số định nghĩa trong mô hình tổng quát:

- Test case : test-case định nghĩa môi trường mà nó có thể sử dụng để chạy nhiều test khác nhau
- TestSuite : Test-Suite là chạy một tập các test-case và nó cũng có thể bao gồm nhiều test suite khác, test suite chính là tổ hợp các test.

2.2.4 Unit Test-case

- Unit Test Case là 1 chuỗi code để đảm bảo rằng đoạn code được kiểm thử làm việc như mong đợi. Để đạt được những kết quả mong muốn một cách nhanh chóng, test framework là cần thiết. JUnit là 1 framework hoàn hảo cho việc kiểm thử đối với ngôn ngữ Java.
- Format viết Unit test : đặc trưng bởi 1 chuỗi đầu vào cho trước và kết quả mong muốn để so sánh với kết quả thực tế. Ví dụ: Kiểm tra cho từng cầu: Kiểm trả số dương, số âm. Có thể chia nhỏ các yêu cầu để test như : positive and negative.

2.2.5 Các viết một unit test-case với JUnit

Việc đầu tiên chúng ta phải tạo một lớp con thừa kế lớp ***junit.framework.TestCase***. Mỗi unit-test được đại diện bởi một phương thức ***testXXX()*** bên trong lớp caon của lớp ***TestCase***. Ta có một lớp Person như sau:

```
public class Person {
    private String firstName;
    private String lastName;
    public Person(String firstName, String lastName) {
        if (firstName == null && lastName == null) {
            throw new IllegalArgumentException("Both names cannot be null");
        }
        this.firstName = firstName;
        this.lastName = lastName;
    }
    public String getFullName() {
        String first = (this.firstName != null) ? this.firstName : "?";
        String last = (this.lastName != null) ? this.lastName : "?";
        return first + last;
    }
    public String getFirstName() {
        return this.firstName;
    }
    public String getLastName() {
        return this.lastName;
    }
}
```

- Sau đó chúng ta sẽ viết một test-case đơn giản để test một số phương thức của lớp trên.

```
import junit.framework.TestCase;
public class TestPerson extends TestCase {
    public TestPerson(String name) {
        super(name);
    }
    /**
        * Xac nhan rang name duoc the hien dung dinh dang
    */
    public void testGetFullName() {
        Person p = new Person("Aidan", "Burke");
        assertEquals("Aidan Burke", p.getFullName());
    }
    /**
    * Xac nhan rang nulls da duoc xu ly chinh xac
    */
    public void testNullsInName() {
        Person p = new Person(null, "Burke");
        assertEquals("? Burke", p.getFullName());
        p = new Person("Tanner", null);
        assertEquals("Tanner ?", p.getFullName());
    }
}
```

- ***Lưu ý:*** Mỗi unit-test là một phương thức public và không có tham số, được bắt đầu bằng tiếp đầu ngữ ***test***. Nếu chúng ta không tuân thủ theo quy tắc ***đặt tên*** này thì JUit sẽ không xác định được các phương thức test một các tự động.

- Để biên dịch class TestPerson, chúng ta phải khai báo gói thư viện JUnit trong biến môi trường classpath:

```
set classpath=%classpath%;.;junit.jar
javac TestPerson
```

- Để chạy một unit test-case, ta có 2 cách:
    - Chạy với môi trường TEXT, chúng ta thực hiện lệnh sau:

```
java junit.textui.TestRunner TestPerson
```

- Chạy với môi trường đồ họa, chúng ta thực hiện lệnh sau:

```
java junit.swingui.TestRunner TestPerson
```

- Chúng ta có thể chạy trực tiếp các test-case mà không muốn kích hoạt một trong các test runner của JUnit. Chúng ta sẽ thêm phương thức ***main()*** vào test-case. Ví dụ:

```
public class TestGame extends TestCase {
    …
    public static void main(String [args) {
        junit.textui.TestRunner.run(new TestSuite(TestGame.class))
    }
}
```

2.2.6 Các phương thức Assert()

- Các phương thức ***assertXXX()*** được dùng để kiểm tra các điều kiện khác nhau.junit.framework.TestCase, lớp cha cho tất cả các test case, thừa kế từ lớp junit.framework.Assert. Lớp này định nghĩa khá nhiều các phương thức ***assertXXX()***. Các phương thức test hoạt động bằng cách gọi những phương thức này.

- Sau đây là mô tả các phương thức ***assertXXX()*** khác nhau có trong lớp ***junit.framework.Assert***.

  - ***assertEquals()***: So sánh 2 giá trị để kiểm tra bằng nhau. Test sẽ được chấp nhận nếu các giá trị bằng nhau.

  - ***assertFalse()***: Đánh giá biểu thức luận lý. Test sẽ được chấp nhận nếu biểu thức sai.

  - ***assertNotNull()***: So sánh tham chiếu của một đối tượng với null. Test sẽ được chấp nhận nếu tham chiếu đối tượng khác null.

  - ***assertNotSame()***: So sánh địa chỉ vùng nhớ của 2 tham chiếu đối tượng bằng cách sử dụng toán tử ==. Test sẽ được chấp nhận nếu cả 2 đều tham chiếu đến các đối tượng khác nhau

- ***assertNull()***: So sánh tham chiếu của một đối tượng với giá trị null. Test sẽ được chấp nhận nếu tham chiếu là null.
- ***assertSame()***: So sánh địa chỉ vùng nhớ của 2 tham chiếu đối tượng bằng cách sử dụng toán tử ==. Test sẽ được chấp nhận nếu cả 2 đều tham chiếu đến cùng một đối tượng.
- ***assertTrue()***: Đánh giá một biểu thức luận lý. Test sẽ được chấp nhận nếu biểu thức đúng ***fail()***: Phương thức này làm cho test hiện hành thất bại, phương thức này thường được sử dụng khi xử lý các biệt lệ.

- Mặc dù bạn có thể chỉ cần sử dụng phương thức ***assertTrue()*** cho gần như hầu hết các test, tuy nhiên thì việc sử dụng một trong các phương thức ***assertXXX()*** cụ thể sẽ làm cho các test của bạn dễ hiểu hơn và cung cấp các thông điệp thất bại rõ ràng hơn.

- Tất cả các phương thức của bảng trên đều nhận vào một String không bắt buộc làm tham số đầu tiên. Khi được xác định, tham số này cung cấp một thông điệp mô tả test thất bại. Ví dụ:

```
assertEquals(employeeA, employeeB);
assertEquals("Employees should be equal after the clone() operation.", employeeA, employeeB);
```

- Phiên bản thứ 2 được ưa thích hơn vì nó mô tả tại sao test thất bại, điều này sẽ giúp cho việc sửa lỗi được dễ dàng hơn.

2.2.7 Cách tạo một unit-test tốt

- Mỗi unit test chỉ nên kiểm tra phần cụ thể của một chức năng nào đó. Chúng ta không nên kết hợp nhiều test không liên quan với nhau lại vào trong một phương thức ***testXXX()***.

- Ta có một lớp ***Game*** như sau:

```
public class Game {
    private Map ships = new HashMap();
    public Game() throws BadGameException {
    }
    public void shutdown() {
    // dummy method
    }
    public synchronized Ship createFighter(String fighterId) {
        Ship s = (Ship) this.ships.get(fighterId);
        if (s == null) {
            s = new Ship(fighterId);
            this.ships.put(fighterId, s);
        }
        return s;
    }
    public boolean isPlaying() {
        return false;
    }
}

public class BadGameException extends Exception {
    public BadGameException(String s) {
        super(s);
    }
}
```

- Sau đó ta viết một đoạn test sau đây:

```
public void testGame() throws BadGameException{
    Game game = new Game();
    Ship fighter = game.createFighter(“001”);
    assertEquals("Fighter did not have the correct identifier", "001", this.fighter.getId());
    Ship fighter2 = this.game.createFighter("001");
    assertSame("createFighter with same id should return same object", fighter, fighter2);
    assertTrue("A new game should not be started yet", !this.game.isPlaying());
}
```

- Đây là một thiết kế không tốt vì mỗi phương thức ***assertXXX()*** đang kiểm tra phần không liên quan của chức năng. Nếu phương thức ***assertEquals()*** thất bại, phần còn lại của test sẽ không được thi hành. Khi xảy ra điều này thì chúng ta sẽ không biết các test khác có đúng chức năng hay không?

- Tiếp theo chúng ta sẽ sửa test trên lại để kiểm tra các khía cạnh khác nhau của trò chơi một cách độc lập.

```
public void testCreateFighter() {
    System.out.println("Begin testCreateFigher()");
    assertEquals("Fighter did not have the correct identifier", "001", this.fighter.getId());
    System.out.println("End testCreateFighter()");
}

public void testSameFighters() {
    System.out.println("Begin testSameFighters()");
    Ship fighter2 = this.game.createFighter("001");
    assertSame("createFighter with same id should return same object", this.fighter
, fighter2);
    System.out.println("End testSameFighters()");
}

public void testGameInitialState() {
    System.out.println("Begin testGameInitialState()");
    assertTrue("A new game should not be started yet", !this.game.isPlaying());
    System.out.println("End testGameInitialState()");
}
```

- Với cách tiếp cận này, khi một test thất bại sẽ không làm cho các mệnh đề ***assertXXX()*** còn lại bị bỏ qua.

- Có thể bạn sẽ đặt ra câu hỏi có khi nào một phương thức test chứa nhiều hơn một các phương thức ***assertXXX()*** hay không? Câu trả lời là có. Nếu bạn cần kiểm tra một dãy các điều kiện và các test theo sau sẽ luôn thất bại nếu có một test đầu tiên thất bại, khi đó bạn có thể kết hợp nhiều phương thức assert vào trong một test.

2.2.8 Phương thức Set Up và Tear Down

- Hai phương thức ***setUp()*** và ***tearDown()*** là một phần của lớp ***junit.framework.TestCase*** Bằng cách sử dụng các phương thức ***setUp*** và ***tearDown***. Khi sử dụng 2 phương thức ***setUp()*** và ***tearDown()*** sẽ giúp chúng ta tránh được việc trùng mã khi nhiều test cùng chia sẻ nhau ở phần khởi tạo và dọn dẹp các biến.

- JUnit tuân thủ theo một dãy có thứ tự các sự kiện khi chạy các test. Đầu tiên, nó tạo ra một thể hiện mới của test case ứng với mỗi phương thức test. Từ đó, nếu bạn có 5 phương thức test thì JUnit sẽ tạo ra 5 thể hiện của test case. Vì lý do đó, các biến thể hiện không thể được

sử dụng để chia sẻ trạng thái giữa các phương thức test. Sau khi tạo xong tất cả các đối tượng test case, JUnit tuân theo các bước sau cho mỗi phương thức test:

  - Gọi phương thức ***setUp()*** của test case.
  - Gọi phương thức test..
  - Gọi phương thức ***tearDown()*** của test case

- Quá trình này được lặp lại đối với mỗi phương thức test trong test case.
- Sau đây chúng ta sẽ xem xét ví dụ:

```
public class Ship {
    private String id;
    public Ship(String id) {
        this.id = id;
    }
    public String getId() {
        return this.id;
    }
}

public class TestGame extends TestCase {
    private Game game;
    private Ship fighter;
    public void setUp() throws BadGameException {
        this.game = new Game();
        this.fighter = this.game.createFighter("001");
    }

    public void tearDown() {
        this.game.shutdown();
    }

    public void testCreateFighter() {
        System.out.println("Begin testCreateFigher()");
        assertEquals("Fighter did not have the correct identifier""001", this.fighter.getId
());
        System.out.println("End testCreateFighter()");
    }

    public void testSameFighters() {
        System.out.println("Begin testSameFighters()");
        Ship fighter2 = this.game.createFighter("001");
        assertSame("createFighter with same id should return same object",this.fighter,
fighter2);
        System.out.println("End testSameFighters()");
    }

    public void testGameInitialState() {
        System.out.println("Begin testGameInitialState()");
        assertTrue("A new game should not be started yet",!this.game.isPlaying());
        System.out.println("End testGameInitialState()");
    }
}
```

- Thông thường bạn có thể bỏ qua phương thức ***tearDown()*** vì mỗi unit test riêng không phải là những tiến trình chạy tốn nhiều thời gian, và các đối tượng được thu dọn khi JVM thoát. ***tearDown()*** có thể được sử dụng khi test của bạn thực hiện những thao tác như mở

kết nối đến cơ sở dữ liệu hay sử dụng các loại tài nguyên khác của hệ thống và bạn cần phải dọn dẹp ngay lập tức. Nếu bạn chạy một bộ bao gồm một số lượng lớn các unit test, thì khi bạn trỏ tham chiếu của các đối tượng đến null bên trong thân phương thức ***tearDown()*** sẽ giúp cho bộ dọn rác lấy lại bộ nhớ khi các test khác chạy.

- Đôi khi bạn muốn chạy vài đoạn mã khởi tạo chỉ một lần, sau đó chạy các phương thức test, và bạn chỉ muốn chạy các đoạn mã dọn dẹp chỉ sau khi tất cả test kết thúc. Ở phần trên, JUnit gọi phương thức ***setUp()*** trước mỗi test và gọi ***tearDown()*** sau khi mỗi test kết thúc, vì thế để làm được điều như trên, chúng ta sẽ sử dụng lớp ***junit.extension.TestSetup*** để đạt được yêu cầu trên.

- Ví dụ sau sẽ minh họa việc sử dụng lớp trên:

```
import junit.extensions.TestSetup;
import junit.framework.*;
public class TestPerson extends TestCase {
    public TestPerson(String name) {
        super(name);
    }

    public void testGetFullName() {
        Person p = new Person("Aidan", "Burke");
        assertEquals("Aidan Burke", p.getFullName());
    }

    public void testNullsInName() {
        Person p = new Person(null, "Burke");
        assertEquals("? Burke", p.getFullName());
        p = new Person("Tanner", null);
        assertEquals("Tanner ?", p.getFullName());
    }

    public static Test suite() {
        TestSetup setup = new TestSetup(new TestSuite(TestPerson.class)) {
            protected void setUp() throws Exception {
            //Thực hiện các đoạn mã khởi tạo một lần ở đây
            }

            protected void tearDown() throws Exception {
                //Thực hiện các đoạn mã dọn dẹp ở đây
            }
        }
        return setup;
    }
}
```

- TestSetup là một lớp thừa kế từ lớp ***junit.extension.TestDecorator***, lớp ***TestDecorator*** là lớp cơ sở cho việc định nghĩa các test biến thể. Lý do chính để mở rộng ***TestDecorator*** là để có được khả năng thực thi đoạn mã trước và sau khi một test chạy. Các phương thức ***setUp()*** và ***tearDown()*** của lớp ***TestSetup*** được gọi trước và sau khi bất kỳ Test nào được truyền vào constructor.

- Trong ví dụ trên chúng ta đã truyền một tham số có kiểu ***TestSuite*** vào constructor của lớp ***TestSetup***.

```
TestSetup setup = new TestSetup(new TestSuite(TestPerson.class)) { }
```

- Điều này có nghĩa là 2 phương thức ***setUp()*** được gọi chỉ một lần trước toàn bộ bộ test và ***tearDown()*** được gọi chỉ một lần sau khi các test trong bộ test kết thúc.

- ***Chú ý***: các phương thức ***setUp()*** và ***tearDown()*** bên trong lớp ***TestPerson*** vẫn được thực thi trước và sau mỗi phương thức test bên trong lớp ***TestPerson***.

3

Hướng dẫn cài đặt và triển khai JUnit

3.1 Cài đặt môi trường

- JUnit là một framework của Java, nên việc đầu tiên là phải cài đặt JDK.
- Yêu cầu hệ thống: version JDK > 1.5

3.2 Bước 1 – cài đặt java

- Mở terminal/console và thực hiện các lệnh java và kết quả như sau:

OS	Output
Windows	java version "1.6.0_21" Java(TM) SE Runtime Environment (build 1.6.0_21-b07) Java HotSpot(TM) Client VM (build 17.0-b17, mixed mode, sharing)
Linux	java version "1.6.0_21" Java(TM) SE Runtime Environment (build 1.6.0_21-b07) Java HotSpot(TM) Client VM (build 17.0-b17, mixed mode, sharing)
Mac	java version "1.6.0_21" Java(TM) SE Runtime Environment (build 1.6.0_21-b07) Java HotSpot(TM)64-Bit Server VM (build 17.0-b17, mixed mode, sharing)

Nếu chưa cài đặt JDK được thì có thể tham khảo link sau:
http://www.oracle.com/technetwork/java/javase/downloads/index.html

3.3 Bước 2 – thiết lập môi trường Java

- Thiết lập biến JAVA_HOME trỏ đến thư mục cơ sở nơi Java được cài đặt trên máy tính.

OS	Output
Windows	Set the environment variable JAVA_HOME to C:\Program Files\Java\jdk1.6.0_21
Linux	export JAVA_HOME=/usr/local/java-current
Mac	export JAVA_HOME=/Library/Java/Home

OS	Output
Windows	Append the string ;C:\Program Files\Java\jdk1.6.0_21\bin to the end of the system variable, Path.
Linux	export PATH=$PATH:$JAVA_HOME/bin/
Mac	not required

3.4 Bước 3 - Download và cài đặt JUnit

- Download: http://www.junit.org

3.5 Bước 4 - Thiết lập môi trường JUnit

- Thiết lập biến môi trường JUNIT_HOME để trỏ đến thư mục cơ sở JUnit Jar được lưu trữ trên máy tính.

OS	Output
Windows	Set the environment variable JUNIT_HOME to C:\JUNIT
Linux	export JUNIT_HOME=/usr/local/JUNIT
Mac	export JUNIT_HOME=/Library/JUNIT

3.6 Bước 5 - Thiết lập biến CLASSPATH

- Thiết lập biến môi trường CLASSPATH để trỏ vào vị trí JUnit Jar.

OS	Output
Windows	Set the environment variable CLASSPATH to %CLASSPATH%;%JUNIT_HOME%\junit4.10.jar;.;
Linux	export CLASSPATH=$CLASSPATH:$JUNIT_HOME/junit4.10.jar:.
Mac	export CLASSPATH=$CLASSPATH:$JUNIT_HOME/junit4.10.jar:.

3.7 Bước 6 – Cài đặt JUnit Setup

- Cài đặt JUnit Setup, tạo class java file name TestJunit:

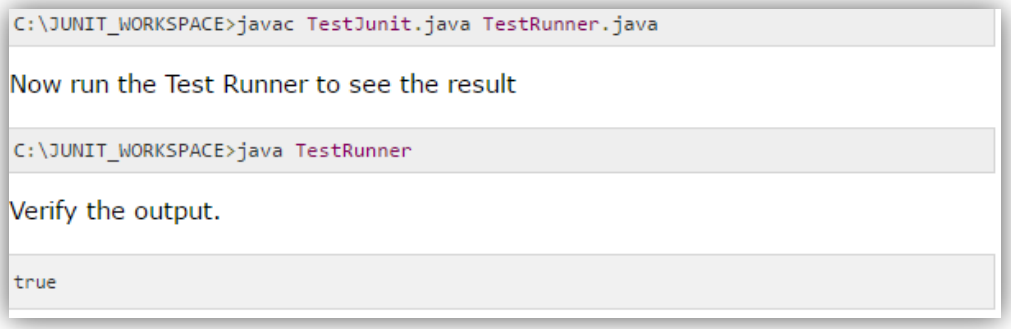
```
c:\> JUNIT_WORKSPACE
```

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;
public class TestJunit {
   @Test
   public void testAdd() {
      String str= "Junit is working fine";
      assertEquals("Junit is working fine",str);
   }
}
```

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(TestJunit.class);
      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }
      System.out.println(result.wasSuccessful());
   }
}
```

3.8 Bước 7 – Kết quả



---------- Hết ----------