

Tài liệu hướng dẫn unit test

Bảng ghi nhận thay đổi

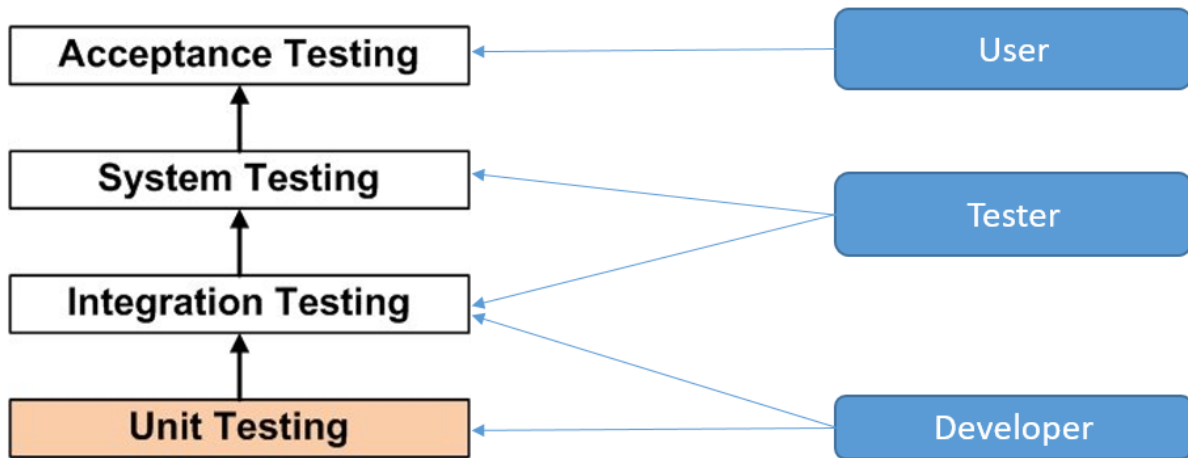
STT	Version	Nội dung	Người tạo	Ngày tạo
1	1.0	Thêm mới toàn bộ	Quangnx10	10/10/2020
2	2.0	Thêm mới hướng dẫn sử dụng tool unit test phần 3	Quangnx10	14/1/2021
3	3.0	Cập nhật thư viện chạy junit (2.1.1), Thay đổi api tool gen test (Phần 3)	Quangnx10	08/03/2021

Contents

Phần 1: Khái niệm Unit Test.....	4
1. Khái niệm	4
2. Tại sao nên thực hiện unit test	4
3. Một số công cụ kiểm thử	5
4. Solitary Unit test và Sociable Unit test	5
Phần 2: Triển khai Unit Test.....	7
1. Cài đặt môi trường (sử dụng maven).....	7
1.1. Các thư viện chạy junit cơ bản.....	7
1.2. Thư viện jacoco để chạy test coverage.....	9
2. Tạo testcase đơn giản	10
3. Triển khai Unit Test với Mockito.....	13
3.1. Mock.....	14
3.2. Spy.....	15
3.3. InjectMock	15
3.4. Các hàm hay được sử dụng.....	16
4. Triển khai Unit Test với các lớp.....	17
4.1. Lớp Controller	17
4.2. Lớp DTO, Entity	18
4.3. Lớp Service	19
5. Demo triển khai test chức năng create patient.....	20
6. Một số testcase khác	25
6.1. Test Exception	25
6.2. Test private method.....	25
6.3. Ghi đè static method.....	26
6.4. Ghi đè phương thức của class cần unit test.....	27
Phần 3: Hướng dẫn viết test sử dụng tool gen test.....	30
1. Import thư viện vào file pom.xml	30
2. Tạo file MainGenTest như sau:	30
3. Gen DTO test.....	30
4. Gen Controller test.....	30
5. Gen Service test	31

Phần 4: Đánh giá Unit Test.....	32
1. Good Unit Tests?	32
2. Sử dụng jacoco đo code coverage	32

Phần 1: Khái niệm Unit Test



1. Khái niệm

- Unit test nghĩa là kiểm thử đơn vị. Đơn vị có thể là một class, một method.
- Unit test là bước kiểm thử đầu tiên trong quá trình phát triển phần mềm. Unit test được thực hiện bởi Developer.
- Unit test là một đoạn mã tự động gọi một đơn vị công việc trong hệ thống và sau đó kiểm tra một giả định duy nhất về hoạt động của đơn vị công việc đó.

2. Tại sao nên thực hiện unit test

- Cải thiện chất lượng code. Viết test trước khi triển khai code khiến Developer suy nghĩ nhiều hơn về vấn đề cần giải quyết, các trường hợp biên có thể có.
- Các vấn đề được phát hiện sớm trước giai đoạn tích hợp code nên có thể giải quyết dễ dàng mà không làm ảnh hưởng đến các thành phần khác. Vấn đề bao gồm lỗi của lập trình viên và đặc tả hành vi của đơn vị cần kiểm thử.
- Unit test cho phép Developer thay đổi và cập nhật code mà vẫn đảm bảo hệ thống chạy chính xác. Vì Unit test xác định được phần thay đổi nào có thể phá vỡ cấu trúc nghiệp vụ, phần nào vẫn chạy chính xác.
- Kiểm tra sự chính xác của từng đơn vị, giúp quá trình tích hợp nhanh chóng.
- Unit test giúp đơn giản hóa quá trình debug. Nếu một test fail, ta biết đầu vào là gì.
- Giảm chi phí fix lỗi. Lỗi được tìm ra trong quá trình System Testing hoặc Acceptance Testing cần rất nhiều thời gian để fix vì không biết cụ thể lỗi xảy ra ở đâu. Trong khi đó các lỗi này có thể được xác định và xử lý từ sớm.
- Giúp xác định rõ chức năng của một đơn vị, đơn vị đó chịu trách nhiệm gì, đầu vào và đầu ra như thế nào, thực hiện chức năng như thế nào.

3. Một số công cụ kiểm thử

Java: Junit4, Junit5, TestNG

.Net: Xunit, NUnit, MSTest

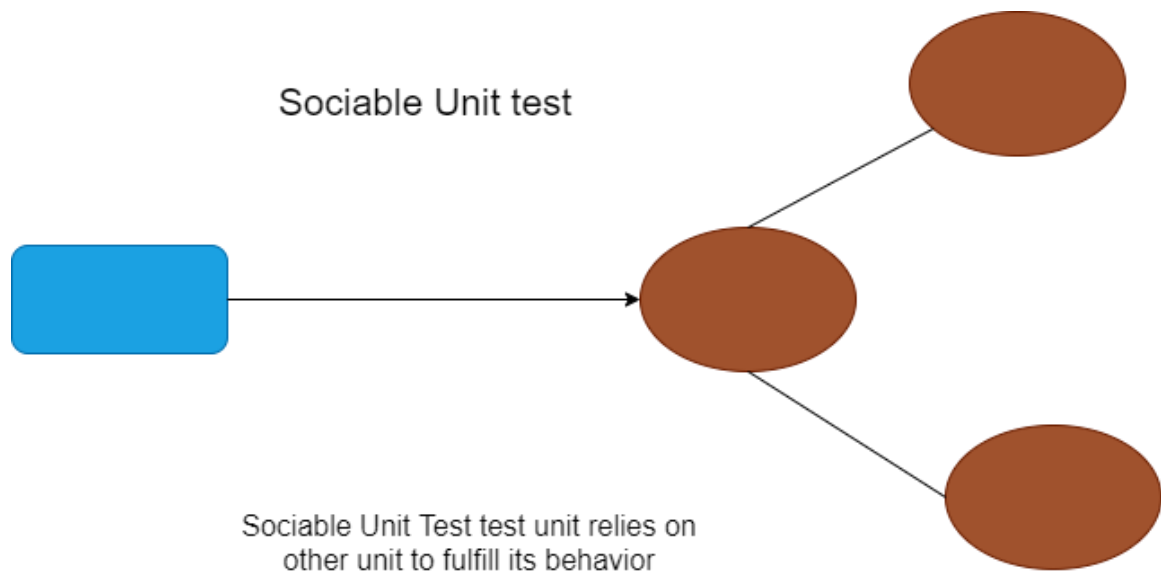
Swift: Xcode

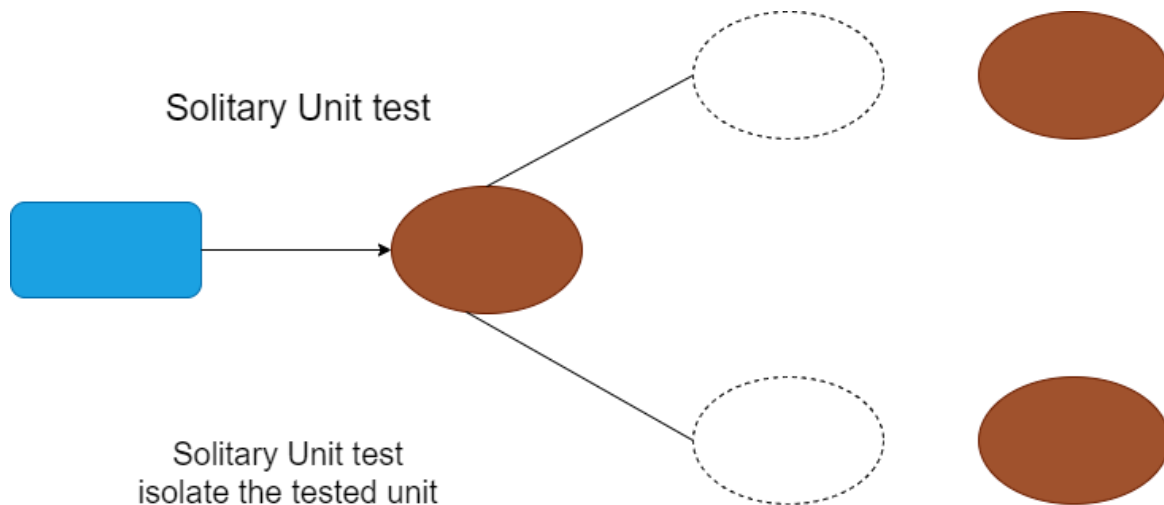
Python: PyUnit

PHP: PHPUnit

Javascript: Mocha, Jest

4. Solitary Unit test và Sociable Unit test





Unit Test được phân thành hai loại: Solitary Unit Test và Sociable Unit Test

- Sociable Unit Test: Kiểm thử hành vi của một đơn vị bằng cách giữ nguyên sự phụ thuộc của đơn vị cần kiểm thử và các đơn vị khác.
- Solitary Unit Test: Kiểm thử đơn vị một cách riêng biệt, độc lập. Giả lập tất cả các đơn vị khác.

Trong quá trình kiểm thử, ta thường áp dụng Solitary Unit Test vì nó tập trung kiểm thử một đơn vị cụ thể, dễ viết testcase, dễ tìm lỗi. Không gọi đến các đơn vị khác nên tốc độ chạy nhanh và không gây ra các ảnh hưởng phụ.

Sociable Unit Test được sử dụng khi ta muốn kiểm thử một luồng nghiệp vụ trong code, và luồng nghiệp vụ này phải không gây ra ảnh hưởng phụ (cập nhật database, gọi đến dịch vụ bên ngoài).

Phần 2: Triển khai Unit Test

1. Cài đặt môi trường (sử dụng maven)

1.1. Các thư viện chạy junit cơ bản

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>jmockit</artifactId>
  <version>1.49</version>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>com.google.code.bean-matchers</groupId>
  <artifactId>bean-matchers</artifactId>
  <version>0.12</version>
  <scope>test</scope>
</dependency>
```

Trong spring-boot-starter-test đã bao gồm junit5-jupiter-api, mockito-all, hamcrest

Nếu không sử dụng spring boot ta cần thêm các dòng lệnh sau vào file pom.xml:

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-api</artifactId>
  <version>5.6.2</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-all</artifactId>
  <version>1.9.5</version>
</dependency>
<dependency>
```

```
<groupId>org.hamcrest</groupId>
<artifactId>hamcrest-library</artifactId>
<version>1.3</version>
<scope>test</scope>
</dependency>
```

Nếu sử dụng spring boot phiên bản thấp không có junit 5, ta cần ghi đè thư viện junit sẵn có bằng cách thêm các dòng lệnh sau vào file pom.xml:

```
<!-- Unit test-->
<dependency>
    <groupId>com.google.code.bean-matchers</groupId>
    <artifactId>bean-matchers</artifactId>
    <version>0.12</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.7.0</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test-mvc</artifactId>
    <version>1.0.0.M2</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.platform</groupId>
    <artifactId>junit-platform-launcher</artifactId>
    <version>1.7.0</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.platform</groupId>
    <artifactId>junit-platform-engine</artifactId>
    <version>1.7.0</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>3.1.0</version>
    <scope>test</scope>
</dependency>
```



```

<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-junit-jupiter</artifactId>
  <version>3.1.0</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.junit</groupId>
  <artifactId>junit-bom</artifactId>
  <version>5.7.0-M1</version>
  <type>pom</type>
</dependency>
<dependency>
  <groupId>org.jmockit</groupId>
  <artifactId>jmockit</artifactId>
  <version>1.49</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.junit.platform</groupId>
  <artifactId>junit-platform-commons</artifactId>
  <version>1.7.0</version>
</dependency>
</dependencies>

```

1.2. Thư viện jacoco để chạy test coverage

```

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>3.0.0-M5</version>
  <configuration>
    <forkCount>5</forkCount>
    <reuseForks>true</reuseForks>
    <parallel>methods</parallel>
    <threadCount>4</threadCount>
    <perCoreThreadCount>true</perCoreThreadCount>
    <reportFormat>plain</reportFormat>
    <trimStackTrace>false</trimStackTrace>
    <redirectTestOutputToFile>true</redirectTestOutputToFile>
    <argLine>@{argLine} -
    javaagent:"${settings.localRepository}/org/jmockit/jmockit/1.49/jmockit-1.49.jar</argLine>
  </configuration>
</plugin>
<plugin>
  <groupId>org.jacoco</groupId>

```

```

<artifactId>jacoco-maven-plugin</artifactId>
<version>0.8.5</version>
<executions>
  <execution>
    <id>prepare-agent</id>
    <goals>
      <goal>prepare-agent</goal>
    </goals>
  </execution>
  <execution>
    <id>report</id>
    <phase>prepare-package</phase>
    <goals>
      <goal>report</goal>
    </goals>
  </execution>
  <execution>
    <id>post-unit-test</id>
    <phase>test</phase>
    <goals>
      <goal>report</goal>
    </goals>
    <configuration>
      <dataFile>target/jacoco.exec</dataFile>
      <!-- Sets the output directory for the code coverage report. -->
      <outputDirectory>target/jacoco-aggregate-report</outputDirectory>
    </configuration>
  </execution>
</executions>
<configuration>
  <excludes>
    <exclude>jdk.internal.*</exclude>
  </excludes>
</configuration>
</plugin>

```

Nếu không sử dụng maven:

B1. Tải file jar của thư viện tương ứng trên trang maven

B2. Chọn project -> Build Path -> chọn file Jar vừa tải về

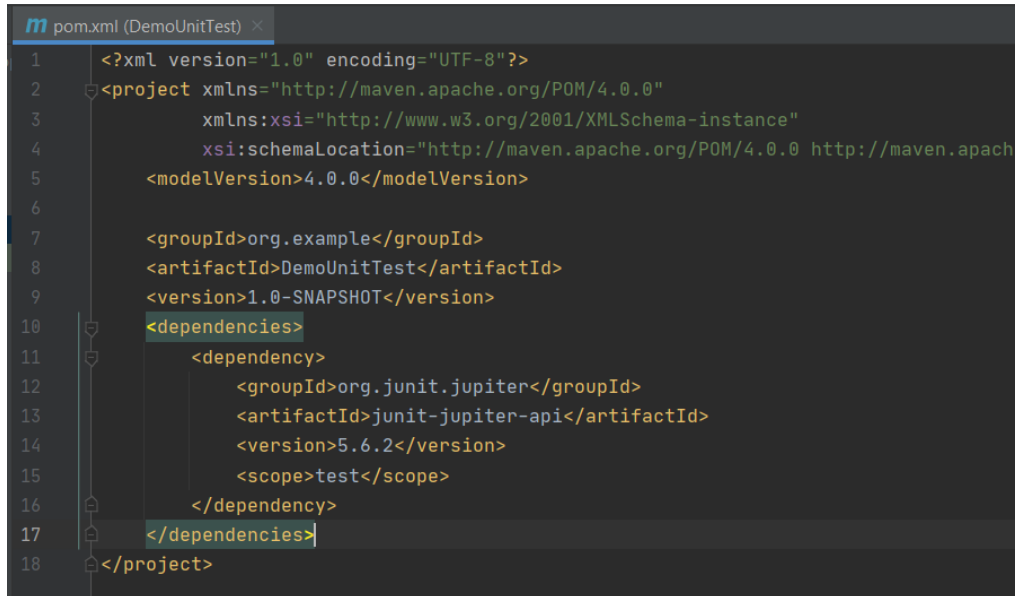
2. Tạo testcase đơn giản

Công cụ sử dụng: IntelliJ Community 2019.3, Junit 5, java 8

B1. Tạo project maven đơn giản:

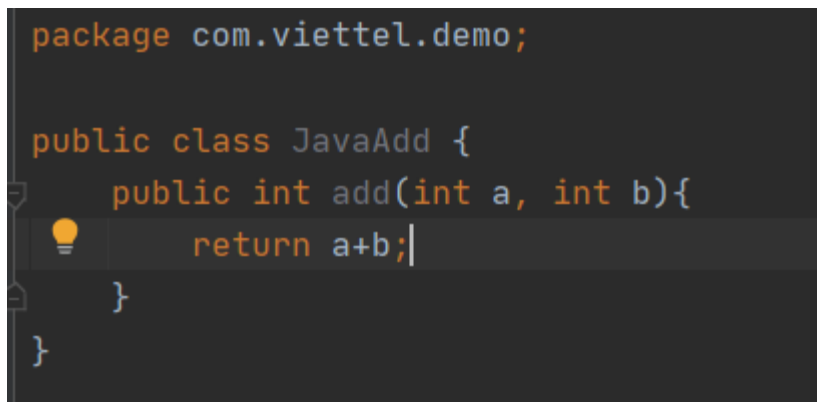
File -> New -> Project -> Maven Project -> đặt tên project là DemoUnitTest -> Finish

B2. Sửa file pom.xml như sau:



```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apach
5     <modelVersion>4.0.0</modelVersion>
6
7     <groupId>org.example</groupId>
8     <artifactId>DemoUnitTest</artifactId>
9     <version>1.0-SNAPSHOT</version>
10    <dependencies>
11        <dependency>
12            <groupId>org.junit.jupiter</groupId>
13            <artifactId>junit-jupiter-api</artifactId>
14            <version>5.6.2</version>
15            <scope>test</scope>
16        </dependency>
17    </dependencies>
18 </project>
```

B3. Tạo class JavaAdd



```
package com.viettel.demo;

public class JavaAdd {
    public int add(int a, int b){
        return a+b;
    }
}
```

B4. Tạo lớp JavaAddTest

```

package com.viettel.demo;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class JavaAddTest {

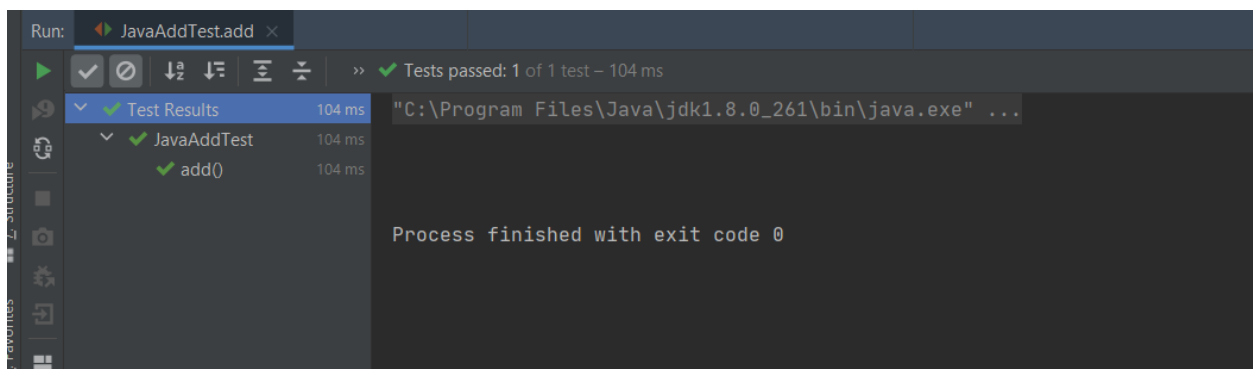
    JavaAdd javaAdd;

    @BeforeEach
    void setUp() {
        javaAdd = new JavaAdd();
    }

    @Test
    void add() {
        int a = 3;
        int b = 4;
        assertTrue( condition: javaAdd.add(a,b) == 7);
    }
}

```

B5. Chạy Test bằng chuột phải -> Run File



Ta có thể chạy test bằng maven như sau:

Thêm maven plugin vào file pom.xml:

```

<build>
  <pluginManagement>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>

```

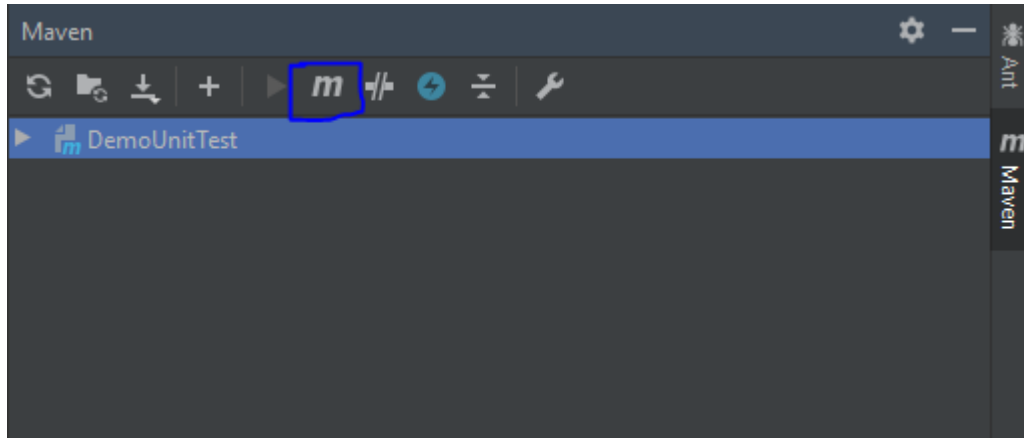
```

        <artifactId>maven-surefire-plugin</artifactId>
        <version>3.0.0-M5</version>
    </plugin>
</plugins>
</pluginManagement>
</build>

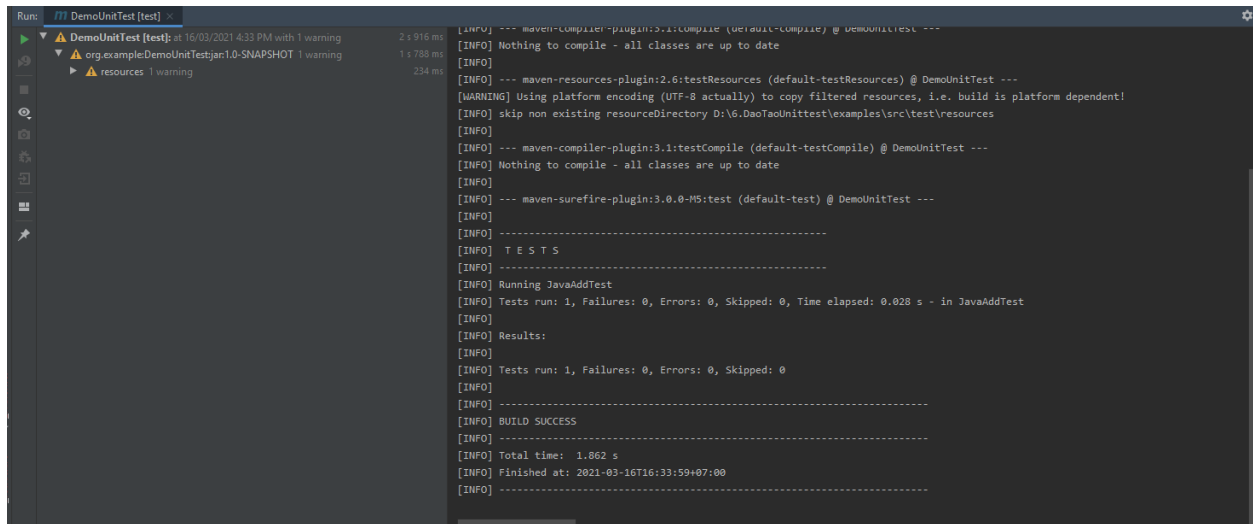
```

Chạy lệnh mvn test bằng intellij hoặc cài maven và chạy trong folder project:

Chạy lệnh mvn test bằng intellij: Chọn tab maven ở thanh menu bên phải -> Chọn chữ M -> gõ mvn test



Kết quả:



3. Triển khai Unit Test với Mockito

Unit test dùng để kiểm thử đơn vị. Tuy nhiên các hàm thường gọi lẫn nhau, hoặc gọi tới database, service bên ngoài.

Do đó ta phải giả lập các đơn vị khác. Để khi đơn vị cần kiểm thử gọi đến các đơn vị khác. Nó sẽ gọi tới các đối tượng giả lập.

Một số framework mock: Mockito, Jmockit, EasyMock, PowerMock.

Trong hướng dẫn sau ta sẽ sử dụng Mockito và Jmockit

Để sử dụng Mockito cần thêm thư viện Mockito vào file pom.xml:

- Các phiên bản Mockito từ 2014 đến nay tên là mockito-core
- Các phiên bản Mockito trước đó tên là mockito-all

```
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-core</artifactId>
  <version>3.1.0</version>
  <scope>test</scope>
</dependency>
```

3.1. Mock

Cách mock một đối tượng:

- Cách 1: Mockito.mock()

```
@Test
public void testUserMockFunction() {
    List mockList = Mockito.mock(List.class);

    Mockito.when(mockList.size()).thenReturn(2);

    assertTrue( condition: mockList.size() == 2);
}
```

- Cách 2: @Mock

```

@Mock
List<String> mockList;

@BeforeEach
void setUp() {
    MockitoAnnotations.initMocks( testClass: this);
}

@Test
public void testUserMockFunction() {
    Mockito.when(mockList.size()).thenReturn(2);

    assertTrue( condition: mockList.size() == 2);
}

```

3.2. Spy

Spy nghĩa đen là gián điệp. Trong Mockito, nó để chỉ một đối tượng được giả lập một số thành phần, các thành phần còn lại giữ nguyên.

Spy cũng có hai cách khởi tạo như Mock.

```

@Test
public void testSpy() {
    List<String> list = Mockito.spy(ArrayList.class);
    list.add("one");
    list.add("two");
    // show the list items
    System.out.println(list.size());

    // @Spy thực sự gọi hàm .add của List nên nó có size là 2 mà không cần giả lập
    assertTrue( condition: list.size() == 2);

    // Vẫn có thể làm giả thông tin gọi hàm với @Spy
    Mockito.when(list.size()).thenReturn(100);

    System.out.println(list.size());
    assertTrue( condition: list.size() == 2);
}

```

3.3. InjectMock

InjectMock: đối tượng được inject bởi các đối tượng mock

Tạo class JavaInjectMock gọi đến checker.Positive

```
public class JavaInjectMock {
    Checker checker = new Checker();
    public int addPositiveNumber(int a, int b){
        if (checker.checkPositive(a) && checker.checkPositive(b))
            return a+b;
        return 0;
    }
}
```

Tạo class test JavaInjectMockTest

```
class JavaInjectMockTest {

    @InjectMocks
    JavaInjectMock javaInjectMock = new JavaInjectMock();

    @Mock
    Checker checker;

    @BeforeEach
    void setUp() {
        MockitoAnnotations.initMocks( testClass: this);
    }

    @Test
    void addPositiveNumber() {
        Mockito.when(checker.checkPositive( a: 4)).thenReturn(true);
        Mockito.when(checker.checkPositive( a: 6)).thenReturn(true);
        assertEquals( expected: 10, javaInjectMock.addPositiveNumber( a: 4, b: 6));
    }
}
```

Checker sẽ được inject vào đối tượng javaInjectMock

3.4. Các hàm hay được sử dụng

Các hàm hay được sử dụng:

- Giả lập throw exception

Mockito.when(<call method>).thenThrow(<Exception>);

- Giả lập hàm void

Mockito.doNothing().when(<đối tượng>).<hàm>()

- Giả lập throw exception từ hàm void

Mockito.doThrow(<Exception>).when(<đối tượng>).<hàm>();

- Khớp tất cả đầu vào thuộc một class

Mockito.anyString(), Mockito.anyInt(), Mockito.any(Class),...

4. Triển khai Unit Test với các lớp

4.1. Lớp Controller

Giả lập restful request bằng MockitoExtension và MockMvcBuilders.

```
@ExtendWith(MockitoExtension.class)
class AccountControllerTest {

    private MockMvc mvc;
    @Mock
    AccountService accountService;
    @Mock
    MessageService messageService;
    @Mock
    SysUsersServiceJPA sysUserServiceJPA;
    @Mock
    KeycloakService keycloakService;
    @InjectMocks
    AccountController accountController;

    @BeforeEach
    void setUp() {
        mvc = MockMvcBuilders.standaloneSetup(accountController).build();
        JacksonTester.initFields( testInstance: this, new ObjectMapper());
    }
}
```

```

@Test
void getAccountInfo3() throws Exception {
    Authentication authentication = null;

    //mock method
    ResultSelectEntity resultData1 = new ResultSelectEntity();
    when(accountService.getAccount(Mockito.any())).thenReturn(resultData1);
    MockHttpServletResponse responseActual = mvc.perform(
        get( urlTemplate: "/info")
        .accept(MediaType.APPLICATION_JSON)
        .contentType(MediaType.APPLICATION_JSON)
        .andReturn().getResponse());

    // assert result
    assertThat(responseActual.getStatus(), Matchers.equalTo(HttpStatus.OK.value()));
}

```

MockMvcBuilders có thể giả lập giống hệ restful api bình thường, bao gồm get, post, put, delete.

Với Lớp controller ta thường đánh giá response status, response content body.

4.2. Lớp DTO, Entity

```

@Data
@JsonInclude(JsonInclude.Include.NON_NULL)
public class AccountActiveDTO {

    Integer userId;

    @NotNull
    boolean active;

    public Integer getIsActive() { return active ? 1 : 0; }
}

```

Sử dụng bean matcher để kiểm thử lớp DTO, Entity. Kiểm thử lớp DTO để kiểm tra lớp DTO có đúng định dạng của Bean trong java không và tránh việc bỏ qua logic trong DTO.

```

class AccountActiveDTOTest {

    @BeforeEach
    void setUp() {
    }

    @Test
    void AccountActiveDTO () {
        assertThat(AccountActiveDTO.class, allOf(hasValidBeanConstructor(), hasValidGettersAndSettersExcluding(new String[]{"isActive"})));
    }

}

```

Testcase bên trên kiểm thử AccountActiveDTO có noArgumentConstructor hợp lệ và getter, setter hợp lệ không. Bỏ qua trường isActive vì isActive có getter chứa logic.

4.3. Lớp Service

Kiểm thử giống như bình thường

```

@Test
void getAccounts() throws TeleCareException {
    AccountDTO dto = new AccountDTO();
    Authentication authentication = new AuthenticationTest();
    TelecareUserEntity telecareUserEntity = new TelecareUserEntity();
    TelecareUserEntity.JwtAccountEntity jwtAccountEntity = new TelecareUserEntity.JwtAccountEntity();
    JwtRoleEntity jwtRoleEntity = new JwtRoleEntity();
    jwtRoleEntity.setRoles(Arrays.asList("Telecare_Admin"));
    jwtAccountEntity.setTelecare(jwtRoleEntity);
    telecareUserEntity.setResource_access(jwtAccountEntity);

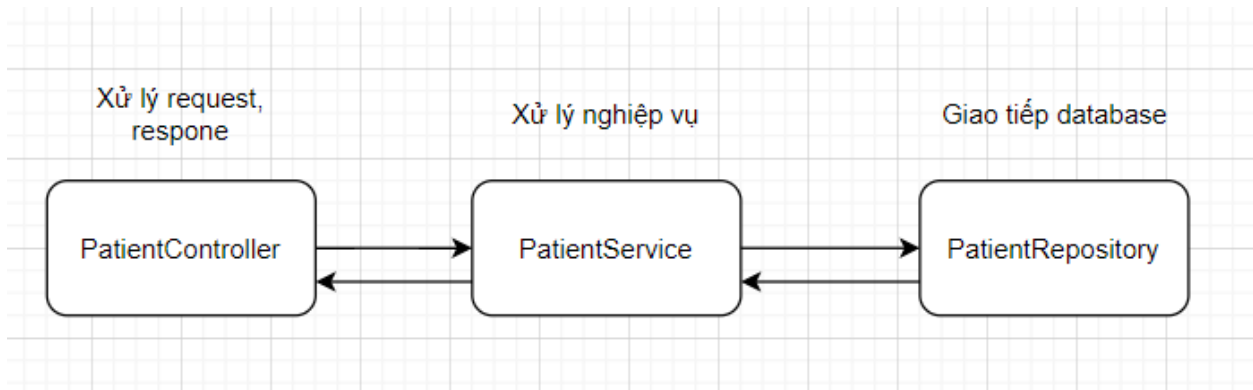
    telecareUserEntity.getResource_access().getTelecare().setRoles(Arrays.asList("Telecare_Admin"));

    (MockUp) getTelecareUserInfo(authentication) -> { return telecareUserEntity; };
    MatcherAssert.assertThat(accountService.getAccounts(dto, authentication), Matchers.nullValue());
}

```

5. Demo triển khai test chức năng create patient

Chức năng create patient hoạt động đơn giản như sau:



Công cụ sử dụng: IntelliJ 2019.3, Spring Boot Framework

Lớp PatientController:

```
@RestController
public class PatientController {
    @Autowired
    private PatientService patientService;
    @PostMapping(value = "/createpatient", produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<Object> createPatient(@RequestBody RequestPatientDTO requestPatientDTO) {
        Object resultObj = patientService.createPatient(requestPatientDTO);
        return new ResponseEntity<>(resultObj, HttpStatus.OK);
    }
}
```

Lớp PatientService:

```
@Service
public class PatientServiceImpl implements PatientService {

    @Autowired |
    private PatientRepository patientRepository;

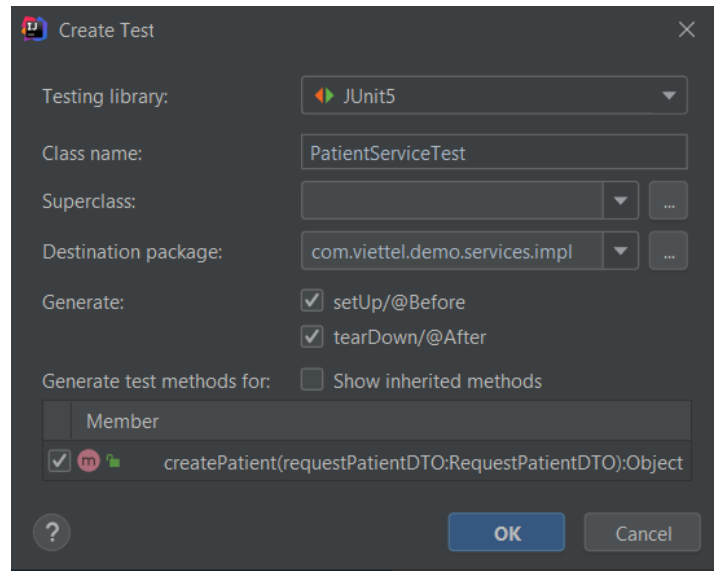
    @Override
    public Object createPatient(RequestPatientDTO requestPatientDTO) {
        /*
        =====
        TODO: (Code at here) Thực hiện luồng nghiệp vụ chi tiết
        =====
        */
        PatientDTO result = patientRepository.createPatient(requestPatientDTO);
        return result;
    }
}
```

Lớp PatientRepository:

```
@Repository
public class PatientRepository {
    public PatientDTO createPatient(RequestPatientDTO itemParamsEntity){
        return null;
    }
}
```

Vì PatientRepository giao tiếp với bên ngoài nên không thực hiện unit test ở các class này mà chỉ thực hiện unit test ở class PatientService.

B1: Alt + Enter vào tên class: PatientService. Chọn Create Test:



- Generate:
 - o SetUp: hàm được gọi trước khi các Test chạy, thường dùng để khởi tạo giá trị.
 - o TearDown: hàm được gọi sau khi các Test chạy, thường dùng để clear dữ liệu sau khi test.
- Member: Tích chọn các phương thức muốn test.

B2: Nhấn OK. IntelliJ tạo ra class PatientServiceTest như sau:

```
package com.viettel.demo.services.impl;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class PatientServiceTest {

    @BeforeEach
    void setUp() {
    }

    @AfterEach
    void tearDown() {
    }

    @Test
    void createPatient() {
    }
}
```

B3: Khởi tạo lớp cần Test và các đối tượng phụ thuộc sử dụng thư viện Mockito.

```

class PatientServiceTest {

    @Mock
    PatientRepository patientRepository;

    @InjectMocks
    PatientService patientService;

    @BeforeEach
    void setUp() {
        patientService = new PatientService();
        MockitoAnnotations.initMocks( testClass: this);
    }

    @AfterEach
    void tearDown() {
    }

    @Test
    void createPatient() {
    }
}

```

@Mock: đối tượng được Mockito khởi tạo, chỉ là một đối tượng ảo, không ánh xạ đến thuộc tính, phương thức của lớp thật.

@InjectMocks: đối tượng được inject các Mock.

MockitoAnnotations.initMocks(this) dùng để khởi tạo các đối tượng Mock và inject Mock vào đối tượng có annotation là @InjectMocks.

B4: Viết test case

Trong class PatientServiceTest ta viết thêm đoạn code sau:

```

@Test
void createPatientStandardData() {
//    Khởi tạo data cho test case
    RequestPatientDTO requestPatientDTO = new RequestPatientDTO( fullName: "Nguyen Quang",
        phoneNumber: "123456789", email: "quang123@gmail.com");
    PatientDTO patientDTO = new PatientDTO((long) 1, fullName: "Nguyen Quang",
        phoneNumber: "123456789", email: "quang123@gmail.com" );

//    override phương thức createPatient của đối tượng patientRepository
    Mockito.when(patientRepository.createPatient(requestPatientDTO)).thenReturn(patientDTO);

//    check với đầu vào chuẩn hàm createPatient của patientService có trả ra kết quả mong
//    muốn không
    MatcherAssert.assertThat(patientService.createPatient(requestPatientDTO),
        Matchers.notNullValue());
}

```

Trong đó:

requestPatientDTO là đầu vào của hàm patientService.createPatient() và hàm patientRepository.createPatient().

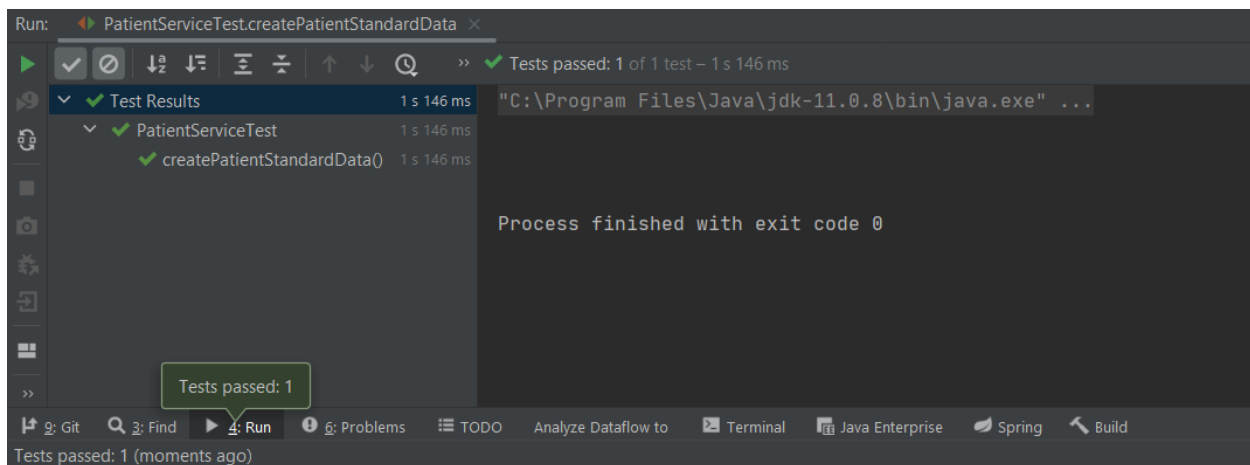
Mokito.when(...).thenReturn(...) dùng để giả lập đầu vào và đầu ra của một phương thức.

Ví dụ: Trong hình bên trên, ta giả lập với tham số truyền vào là requestPatientDTO thì hàm patientRepository.createPatient() trả ra patientDTO.

Lưu ý: Với phương thức void ghi đè bằng: Mockito.doNothing().when(patientRepository).createPatient()

B5: Chạy Testcase

Chọn maven góc bên phải và chọn test, hoặc chuột phải chọn run Class PatientServiceTest hoặc run phương thức createPatientStandardData().



6. Một số testcase khác

6.1. Test Exception

Vấn đề: Trong rất nhiều trường hợp, ta muốn kiểm thử xem hàm có bắn ra lỗi khi truyền đầu vào sai hoặc lỗi bắn ra có hợp lý không.

Giải pháp: Ta sử dụng `Assertions.assertThrows` để test exception.

```
public Object getBookingInformationResult(BookingInformationResultDTO itemParamsEntity, Authentication authentication) throws TeleCareException {
    Integer currentPatientId = patientsServiceJPA.getUserIdFromToken(authentication);
    Boolean hasRelationship = patientsServiceJPA.checkRelationship(itemParamsEntity.getPatientId(), currentPatientId);
    if (!hasRelationship) {
        FnCommon.throwErrorApp(ErrorApp.ERR_PATIENT_RELATIONSHIP_NOT_EXIST);
    }
    ResultSelectEntity dataResult = bookingInformationResultRepository.getBookingInformationResult(itemParamsEntity);
    return dataResult;
}
```

```
@Test
void getBookingInformationResultFalseRelationship() throws TeleCareException {
    Integer currentPatientId = 1;
    Integer patientId = 2;
    Boolean hasRelationship = false;
    BookingInformationResultDTO itemParamsEntity = new BookingInformationResultDTO();
    itemParamsEntity.setPatientId(patientId);
    AuthenticationTest authentication = new AuthenticationTest();

    Mockito.when(patientsServiceJPA.getUserIdFromToken(authentication)).thenReturn(currentPatientId);
    Mockito.when(patientsServiceJPA.checkRelationship(itemParamsEntity.getPatientId(), currentPatientId))
        .thenReturn(hasRelationship);

    //result
    ResultSelectEntity dataResult = new ResultSelectEntity();
    dataResult.setListData(Arrays.asList(new BookingInformationResultDTO()));
    dataResult.setCount(1);

    Mockito.when(bookingInformationResultRepository.getBookingInformationResult(itemParamsEntity))
        .thenReturn(dataResult);
    TeleCareException thrown3 = Assertions.assertThrows(TeleCareException.class, () -> {
        service.getBookingInformationResult(itemParamsEntity, authentication);
    });
}
```

6.2. Test private method

Vấn đề: Ta không thể gọi phương thức private từ bên ngoài class để test.

Giải pháp: Sử dụng `java.lang.reflect.Method` để gọi đến hàm private.

```

@Service
public class PatientService {

    @Autowired
    private PatientRepository patientRepository;

    public Object createPatient(RequestPatientDTO requestPatientDTO) throws Exception {
        if(requestPatientDTO == null) throw new Exception("input error");
        if(patientRepository.checkPhoneExist(requestPatientDTO.getPhoneNumber()))
            return new Exception("phone exist");
        PatientDTO result = patientRepository.createPatient(requestPatientDTO);
        return result;
    }

    private Long convertPhoneToLong(String phoneNumber) throws NumberFormatException {
        return Long.parseLong(phoneNumber.replace( target: "+", replacement: ""));
    }
}

```

Thêm hàm private convertPhoneToLong vào class PatientService. Ta viết hàm test như sau:

```

@Test
void testPrivateMethod() throws NoSuchMethodException, InvocationTargetException, IllegalAccessException {
    Method method = PatientService.class.getDeclaredMethod( name: "convertPhoneToLong", String.class);
    method.setAccessible(true);
    MatcherAssert.assertThat(method.invoke(patientService, ...args: "+84039655555"),
        Matchers.equalTo(Long.valueOf("84039655555")));
}

```

6.3. Ghi đè static method

Vấn đề: Mockito không ghi đè được static method.

Giải pháp: Sử dụng Jmockit để ghi đè static method

Ta tạo thêm lớp KeyCloakService:

```

@Service
public class KeyCloakService {
    public static UUID createUser(RequestPatientDTO requestPatientDTO){
        return null;
    }
}

```

Sửa class PatientService gọi đến hàm KeyCloakService.createUser

```
public Object createPatient(RequestPatientDTO requestPatientDTO) throws Exception {
    if(requestPatientDTO == null) throw new Exception("input error");
    if(patientRepository.checkPhoneExist(requestPatientDTO.getPhoneNumber()))
        return new Exception("phone exist");

    PatientDTO result = patientRepository.createPatient(requestPatientDTO);
    result.setKeycloakId(KeyCloakService.createUser(requestPatientDTO));
    return result;
}
```

Lớp Test ta phải sử dụng thêm Mockup của jmockit để ghi đè phương thức static như sau:

```
@Test
void createPatientWithKeyCloak() throws Exception {
    RequestPatientDTO requestPatientDTO = new RequestPatientDTO( fullName: "Nguyen Quang",
        phoneNumber: "123456789", email: "quang123@gmail.com");
    PatientDTO patientDTO = new PatientDTO((long) 1, fullName: "Nguyen Quang",
        phoneNumber: "123456789", email: "quang123@gmail.com" );

    Mockito.when(patientRepository.createPatient(requestPatientDTO)).thenReturn(patientDTO);
    Mockito.when(patientRepository.checkPhoneExist(requestPatientDTO.getPhoneNumber())).thenReturn(false);
    new MockUp<KeyCloakService>() {
        @mockit.Mock
        public UUID createUser(RequestPatientDTO requestPatientDTO) {
            return UUID.randomUUID();
        }
    };
    MatcherAssert.assertThat(patientDTO.getKeycloakId(), Matchers.notNullValue());
}
```

6.4. Ghi đè phương thức của class cần unit test

Vấn đề: Trong phương thức cần test gọi đến phương thức insertToTableDetail. Ta cần ghi đè phương thức insertToTableDetail vì phương thức tác động đến database, mỗi lần test sẽ ảnh hưởng đến database hoặc báo lỗi vì không thể kết nối với database.

```
if (!updates.isEmpty()) {
    medicalHealthcarePatientSummaryRepositoryJPA.saveAll(updates);
    insertToTableDetail(dto);
}
```

Giải pháp: Dùng Mockito Spy để khởi tạo lại đối tượng, ghi đè phương thức insertToTableDetail. Sau đó khởi tạo lại các Mock bằng MockitoAnnotations.initMocks(this).

```
service = Mockito.spy(new MedicalHealthcarePatientSummaryServiceImpl(){
    @Override
    void insertToTableDetail(MedicalHealthcarePatientSummaryDTO dto){

    }
});
MockitoAnnotations.initMocks( testClass: this);
```

Lưu ý: Không thể ghi đè phương thức private và static. Ghi đè static dùng jmockit như đã đề cập ở phần 2.3.3.

Vấn đề: Phương thức gọi đến hàm private, mà không thể ghi đè hàm private bằng jmockit hoặc mock.

Giải pháp: Ta coi hàm private là một phần của hàm cần test, viết mock như bình thường.

Ta viết test cho createPatientSummary và coi như đoạn mã của savePatientDetail là một phần của createPatientSummary.

```
public Object createPatientSummary(MedicalHealthcarePatientSummaryDTO dto) throws TeleCareException {
    // do something
    savePatientDetail(dto);
    //do something
    return dto;
}

private void savePatientDetail(MedicalHealthcarePatientSummaryDTO dto) throws TeleCareException {
    MedicalHealthcarePatientDetailEntity entity = new MedicalHealthcarePatientDetailEntity();
    dto.setSummaryId(null);
    FnCommon.copyProperties(dto, entity);
    if (entity.getClass().getDeclaredFields().length == FnCommon.getNullPropertyName(entity).length) {
        FnCommon.throwErrorApp(ErrorApp.ERROR_INPUTPARAMS);
    }

    medicalHealthcarePatientDetailRepositoryJPA.save(entity);
}
```

```
@Test
void createPatientSummaryStandardData() throws TeleCareException {
    MedicalHealthcarePatientSummaryDTO dto = new MedicalHealthcarePatientSummaryDTO();
    // write mock for do something

    // write mock for private method
    int detailId = 1;
    MedicalHealthcarePatientDetailEntity entity = new MedicalHealthcarePatientDetailEntity();
    dto.setSummaryId(null);
    FnCommon.copyProperties(dto, entity);
    MedicalHealthcarePatientDetailEntity result = new MedicalHealthcarePatientDetailEntity();
    FnCommon.copyProperties(dto, result);
    result.setDetailId(detailId);
    Mockito.when(medicalHealthcarePatientDetailRepositoryJPA.save(entity)).thenReturn(result);

    //write mock for do something

    //write assert
    MatcherAssert.assertThat(service.createPatientSummary(dto), Matchers.notNullValue());
}
```

Phần 3: Hướng dẫn viết test sử dụng tool gen test

1. Import thư viện vào file pom.xml

```
<dependency>
  <groupId>com.viettel</groupId>
  <artifactId>gentest</artifactId>
  <version>1.0.21</version>
</dependency>
```

check version mới nhất để thay đổi

2. Tạo file MainGenTest như sau:

```
public class MainGenTest {
    public static void main(String[] args) throws NoSuchMethodException,
IOException, ClassNotFoundException {
        // gọi các hàm gen test ở đây

GenServiceTest.genAllInModule("D:\\4.Source\\Iva\\iva_web_product_svc\\product-
data\\src\\main\\java",
        "D:\\4.Source\\Iva\\iva_web_product_svc\\product-
data\\src\\test\\java");

    }
}
```

Tham số đầu vào thứ nhất là địa chỉ chứa source code. Tham số đầu vào thứ hai là địa chỉ chứa source test gen ra.

3. Gen DTO test

- Để gen tất cả các lớp có tên kết thúc bởi DTO|Entity|Form|Response|Request gọi hàm:

```
GenDTOTest.genAllInModule();
```

- Để gen cho một lớp gọi hàm:

```
GenDTOTest.genOneInModule ();
```

```
VD: GenDTOTest.genOneInModule ("D:\\4.Source\\Iva\\iva_web_product_svc\\product-
data\\src\\main\\java",
        "D:\\4.Source\\Iva\\iva_web_product_svc\\product-
data\\src\\test\\java",TopicModel.class.getSimpleName());
```

4. Gen Controller test

- Chỉ áp dụng với các project có base giống telecare, etc

- Để gen tất cả các lớp có tên kết thúc là Controller gọi hàm:

```
GenControllerTest.genAllInModule();
```

- Để gen cho một lớp gọi hàm:

```
GenControllerTest.genOneInModule ();
```

5. Gen Service test

- Để gen tất cả các lớp có tên kết thúc là ServiceImpl|ServiceJPA gọi hàm:

```
GenServiceTest.genAllInModule ();
```

- Để gen cho một lớp gọi hàm:

```
GenServiceTest.genOneInModule ();
```

Phần 4: Đánh giá Unit Test

1. Good Unit Tests?

- Automatic: tự động, tích hợp vào luồng CI/CD
- Speed: Thường không đến 1s/testcase.
- Independence: không phụ thuộc vào môi trường, vào các test case khác. **Không khởi tạo ngữ cảnh của ứng dụng**
- Readable: viết test dễ hiểu.
- Repeatable: đầu vào cố định. Hạn chế sử dụng Random()

2. Sử dụng jacoco đo code coverage

Sử dụng thư viện jacoco để đánh giá độ phủ của Unit Test, Ta chạy câu lệnh sau trong folder chính của project:

```
mvn test org.jacoco:jacoco-maven-plugin:0.8.5:report-aggregate
```

Truy cập file target/jacoco-aggregate-report/index.html để xem báo cáo của jacoco

serviceDemo

serviceDemo

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.viettel.demo.dto	<div><div></div></div>	5%	<div><div></div></div>	0%	91	96	14	28	33	38	0	2
com.viettel.demo.utils	<div><div></div></div>	0%	n/a	n/a	15	15	26	26	15	15	5	5
com.viettel.demo.controllers	<div><div></div></div>	0%	n/a	n/a	2	2	3	3	2	2	1	1
com.viettel.demo	<div><div></div></div>	0%	n/a	n/a	3	3	4	4	3	3	1	1
com.viettel.demo.repositories.impl	<div><div></div></div>	0%	n/a	n/a	3	3	3	3	3	3	1	1
com.viettel.demo.services.impl	<div><div></div></div>	93%	<div><div></div></div>	100%	1	7	1	10	1	5	0	2
Total	773 of 854	9%	116 of 120	3%	115	126	51	74	57	66	8	12

Theo các nguồn trên mạng thì unit test nên đặt mục tiêu là 70-80%:

<https://www.bullseye.com/minimum.html#:~:text=Code%20coverage%20of%2070%2D80,higher%20than%20for%20system%20testing.>

<https://medium.com/@DomBurf/how-much-code-coverage-is-enough-30e162839112>

Số liệu coverage không thật sự quan trọng, điều quan trọng là test đã bao phủ được phần nghiệp vụ của chương trình chưa, test đang tập trung vào đâu, phần nào đang thiếu test.