آزمون نرم افزار

تمرین ۱

محمد جواد رنجبر

۸۱۰۱۰۱۱۷۳

کدها در این لینک قابل مشاهده می‌باشند.

سوال ۱

الف)

برای کلاس‌های طراحی شده تست‌های ساده‌ای می‌نویسم که عملکرد ساخته شدن کلاس و دریافت ورودی‌های مختلف را چک کند:

Invoice:

```
public void testInvoiceCreation()
```

- این تست بررسی می‌کند که یک فاکتور با یک مشتری و دو اجرای نمایشی به درستی ایجاد می‌شود. ابتدا، نام مشتری و دو اجرای نمایشی به صورت نمونه ایجاد می‌شوند. سپس، یک لیست از این اجراها ساخته شده و فاکتور با استفاده از این اطلاعات ایجاد می‌شود.
- با استفاده از **assertNotNull**، بررسی می‌شود که فاکتور ایجاد شده نباید null باشد.
- با استفاده از **assertEquals**، تعداد اجراهای نمایشی و اطلاعات مشتری در فاکتور بررسی می‌شوند.

```
public void testEmptyInvoice()
```

- این تست بررسی می‌کند که یک فاکتور با یک مشتری و بدون اجرای نمایشی به درستی ایجاد می‌شود.
- ابتدا، نام مشتری و یک لیست خالی از اجراها به عنوان ورودی‌ها در نظر گرفته می‌شوند.
- سپس، فاکتور با استفاده از این اطلاعات ایجاد می‌شود.
- با استفاده از **assertEquals**، بررسی می‌شود که نام مشتری و لیست خالی اجراها در فاکتور درست باشد.

Performance:

```
public void testPerformanceConstructor()
```

- این تست بررسی می‌کند که یک شیء **Performance**با شناسه نمایش و تعداد مخاطبین مشخص به درستی ایجاد می‌شود.
- مقدارهای ورودی به سازنده **playID)** و **audience)**با مقدارهای موجود در شیء ایجاد شده مقایسه می‌شوند.

```
public void testAudienceIncrease() {
```

- این تست بررسی می‌کند که مقدار مخاطبین پس از افزایش به درستی قابلیت به‌روز رسانی دارند.

- ابتدا یک شیء **Performance**با تعداد اولیه مخاطبین ایجاد می‌شود.
- سپس مقدار مخاطبین افزایش یافته و مقدار نهایی بررسی می‌شود.

```
public void testNoNegativeAudience()
public void testZeroAudionce() {
```

- این دو تست بررسی می‌کند که تعداد مخاطبین باید یک از عدد بزگتر مساوی صفر باشد. که البته این مورد در ساخت کلاس‌ها رعایت نشده است و یکی از تست‌های شکست می‌خورد.
- یک شیء **Performance**با تعداد مخاطبین صفر و منفی ایجاد می‌شود و بررسی می‌شود که تعداد مخاطبین برابر یا بیشتر از یا برابر با صفر باشد.

Play:

```
public void testPlayCreation() {
```

- این تست بررسی می‌کند که یک شیء **Play**با نام نمایشنامه و نوع نمایشنامه مشخص به درستی ایجاد می‌شود.
- مقدارهای ورودی به سازنده **playName**) و **playType**)با مقدارهای موجود در شیء ایجاد شده مقایسه می‌شوند.

```
public void testDifferentPlayType() {
```

- این تست بررسی می‌کند که نوع نمایشنامه می‌تواند متفاوت باشد و مقدار آن به درستی ذخیره شود.
- یک شیء **Play**با نام و نوع نمایشنامه متفاوت ایجاد می‌شود و نوع نمایشنامه بررسی می‌شود.

ب)

متد print در کلاس FactorPrinter وظیفه تولید یک فاکتور متنی از اطلاعات موجود در شیء Invoice و نمایشنامه‌ها (Play) را بر عهده دارد. این متد با توجه به نوع نمایشنامه‌ها و تعداد مخاطبین، مبلغ کل و مجموع امتیازات را محاسبه کرده و به شکل یک رشته متنی نمایش می‌دهد. در ادامه به توضیح کامل این متد می‌پردازیم:

متغیرهای استفاده شده در متد print

- totalAmount جمع کل مبلغی که باید پرداخت شود.

- volumeCredits مجموع امتیازات که مشتری کسب می‌کند.
- Result رشته‌ای که نتیجه نهایی فاکتور را نگهداری می‌کند.
- Frmt برای فرمت کردن اعداد به صورت ارزی (Currency) به فرمت دلار آمریکا.

مراحل اجرای متد print:

ایجاد متغیرهای اولیه:

- totalAmount و volumeCredits به ترتیب برای نگهداری مجموع مبلغ و مجموع امتیازات استفاده می‌شوند.
- resultبرای نگهداری نتیجه نهایی فاکتور استفاده می‌شود و با نام مشتری شروع می‌شود.
- Frmt برای فرمت کردن اعداد به صورت ارزی به فرمت دلار آمریکا استفاده می‌شود.

۲. پردازش هر اجرای نمایشی:

یک حلقه برای پردازش هر اجرای نمایشی (perf) در فاکتور (invoice.performances) ایجاد می‌شود.

Play نمایشنامه مربوط به این اجرا را از نقشه plays استخراج می‌کند.

thisAmount برای محاسبه مبلغ مربوط به این اجرای نمایشی استفاده می‌شود.

محاسبه مبلغ هر اجرا بر اساس نوع نمایشنامه:

- مبلغ thisAmountبر اساس نوع نمایشنامه محاسبه می‌شود.
- برای نمایشنامه‌های تراژدی(tragedy) ، مبلغ پایه ۴۰۰۰۰ است و برای هر مخاطب بیشتر از ۳۰ نفر، مبلغ اضافی اضافه می‌شود.
- برای نمایشنامه‌های کمدی(comedy) ، مبلغ پایه ۳۰۰۰۰ است و برای هر مخاطب بیشتر از ۲۰ نفر، مبلغ اضافی اضافه می‌شود و برای هر مخاطب، ۳۰۰ واحد اضافی نیز اضافه می‌شود.
- در صورت نامشخص بودن نوع نمایشنامه، خطای Error بازگردانده می‌شود.

۲. محاسبه مجموع امتیازات:

مجموع امتیازات برای هر مخاطب بیشتر از ۳۰ نفر محاسبه می‌شود.

برای نمایشنامه‌های کمدی، برای هر پنج مخاطب، یک امتیاز حجمی اضافی در نظر گرفته می‌شود.

اضافه کردن اطلاعات هر اجرا به نتیجه نهایی:

اطلاعات مربوط به نام نمایشنامه، مبلغ آن (با فرمت ارزی) و تعداد مخاطبین به نتیجه نهایی اضافه می‌شود.

thisAmount به totalAmount اضافه می‌شود.

اضافه کردن مبلغ کل و مجموع امتیازات به نتیجه نهایی:

مبلغ کل و مجموع امتیازات به نتیجه نهایی اضافه می‌شود. رشته نهایی result که شامل تمامی اطلاعات فاکتور است، بازگردانده می‌شود.

نوشتن تست برای این کلاس:

در اینجا قصد بررسی بررسی همه‌ی حالت‌ها را داریم و می‌خواهیم وارد تمام ifها شویم و edge caseها را نیز بررسی کنیم برای همین تعداد برای موارد زیر تست می‌نویسیم:

۱. تست برای سالن‌هایی که هیچ نمایشنامه‌ای وجود ندارد.

۲. تست برای سالن‌هایی که نوع نمایشنامه‌های جدیدی وجود دارد که توسط برنامه پشتیبانی نمی‌شود.

۳. تست‌های مختلف برای نمایشنامه‌های کمدی با تعداد مخاطبین متفاوت (که شامل صفر تماشاگر، کمتر از ۲۰ مخاطب، بین ۲۰ تا ۳۰ مخاطب بیشتر از ۳۰ مخاطب و دقیقا به اندازه‌ی کافی مخاطب)

۴. تست‌های مختلف برای نمایشنامه‌های تراژدی با تعداد مخاطبین متفاوت. (که شامل صفر تماشاگر، کمتر از ۲۰ مخاطب، بین ۲۰ تا ۳۰ مخاطب، ۳۰ مخاطب، بیشتر از ۳۰ مخاطب و دقیقا به اندازه‌ی کافی مخاطب)

۵. تست‌های برای ترکیب نمایشنامه‌های تراژدی و کمدی با تعداد مخاطبین متفاوت (که شامل صفر تماشاگر، کمتر از ۲۰ مخاطب، بین ۲۰ تا ۳۰ مخاطب بیشتر از ۳۰ مخاطب)

تمام تست‌های بالا برای بررسی عملکرد این کلاس نوشته شده است.

سوال ۲

در این سوال کلاس‌هایی برای کارنامه و بررسی پیش‌نیاز دانشجویی نوشته شده است به این صورت که تابع hasPassedPre بررسی می‌کند که آیا یک دانشجویی شرایط پیش‌نیاز برای گذراندن یک درس را دارد یا خیر. این تابع دو ورودی دریافت می‌کند: لیستی از رکوردهای درسی (rec) و یک شیء از نوع Course (course). در این تابع، برای هر درس پیش‌نیازی (pre)، ابتدا چک می‌شود که آیا دانشجو آن درس را گذرانده است یا نه. اگر رکوردی با courseId مشخص شده وجود داشت و شرایط گذراندن درس را داشت (نمره بزرگتر یا مساوی با ۱۰ و اگر مهمان نبود، نمره بزرگتر یا مساوی با ۱۲)، متغیر prePassed به true تغییر می‌یابد. در صورتی که برای یکی از دروس پیش‌نیاز، شرط گذراندن برقرار نبود، تابع false را باز می‌گرداند. در غیر این صورت، به این معناست که تمامی شرایط پیش‌نیاز برای درس مورد نظر برآورده شده‌اند و تابع true را باز می‌گرداند.

برای تست این کلاس به صورت زیر عمل می‌کنیم:

تست‌های نوشته شده برای کلاس Passcheck بررسی می‌کنند که تابع hasPassedPre با دریافت لیستی از رکوردهای درسی و یک شیء از نوع Course ، به درستی عملکرد مورد انتظار را ارائه می‌دهد یا خیر. تست‌ها انواع مختلفی از ورودی‌ها را بررسی می‌کنند، از جمله:

- مواردی که دانشجویی شرایط پیش‌نیاز را بدون اینکه به عنوان مهمان ثبت شده باشد، گذرانده است.
- مواردی که هیچ پیش‌نیازی برای درس مورد نظر وجود ندارد.
- مواردی که دانشجویی شرایط پیش‌نیاز را بدون اینکه به عنوان مهمان ثبت شده باشد، نگذرانده است. (فقط تک درس پیش نیاز، چند درس پیش نیاز و درس پیش نیاز را افتاده باشد)

- مواردی که دانشجویی شرایط پیش‌نیاز را به عنوان مهمان نگذرانده است. . (فقط تک درس پیش نیاز، چند درس پیش نیاز و درس پیش نیاز را افتاده باشد)
- مواردی که دانشجو هیچ کارنامه‌ای برای درس‌ها ندارد.
- مواردی که درس پیشنیاز در کارنامه نباشد.

تمام موارد با در تست‌های اعمال شده موجود می‌باشد.

سوال ۳

الف)

```
@Test
public void testA() {
    Integer result = new SomeClass().aMethod();
    System.out.println("Expected result is 10. Actual result is " + result);
}
```

مشکل این تست این است که فاقد assertion است. در unit test، assertion برای تأیید رفتار مورد انتظار کد مورد آزمایش بسیار مهم هستند. بدون assertion، حتی اگر روش آزمایش شده به درستی عمل نکند، آزمون ممکن است با موفقیت انجام شود. برای حل این مشکل به صورت زیر عمل می‌کنیم:

```
@Test
public void testA() {
    Integer expectedResult = 10;
    Integer actualResult = new SomeClass().aMethod();
    assertEquals(expectedResult, actualResult);
}
```

ب)

```
@Test
public void testC() expects Exception {
    int badInput = 0;
    new AnotherClass().process(badInput);
}
```

کد بالا در جاوا ارور خواهد خورد و مشکل سینتکسی دارد، نحوه‌ی صحیح نوشتن به شرح زیر می‌باشد:

```
@Test(expected = SomeException.class)
public void testC() {
```

```
    int badInput = 0;
    new AnotherClass().process(badInput);}
```

همچنین در نسخه‌هایی از جاوا نوع بالا کار نمی‌کند و باید به صورت زیر پیاده‌سازی شود:

```
@Test
public void testC() {
    int badInput = 0;
    AnotherClass anotherClass = new AnotherClass();
    assertThrows(SomeException.class, () -> anotherClass.process(badInput));
```

ج)

```
@Test
public void testInitialization() {
    // Initialize the configuration and resources
    Configuration.initialize();
    ResourceManager.initialize();
    // Perform assertions to validate the initialization
    // ...
}
```

مشکل این مجموعه تست‌ها این است که ممکن است به یکدیگر وابسته باشند، اما برای اجرا در یک ترتیب خاص طراحی نشده‌اند. آزمون‌های واحد عموماً باید مستقل از یکدیگر باشند تا اطمینان حاصل شود که می‌توانند به صورت مجزا و به هر ترتیبی اجرا شوند، بدون اینکه بر نتایج سایر آزمون‌ها تأثیر بگذارند.

برای رفع این مشکل، باید مقداردهی اولیه لازم را در روش @Before تنظیم کنید، که اطمینان حاصل می‌کند که مراحل اولیه‌سازی مورد نیاز قبل از هر آزمایش اجرا می‌شود. که در کد زیر اجرا شده است.

```
import org.junit.Before;
import org.junit.Test;

import static org.junit.Assert.assertTrue;

public class ResourceManagerTest {

    @Beforeall
    public void setUp() {
        Configuration.initialize();
        ResourceManager.initialize();
    }

    @Test
    public void testResourceAvailability() {
```

```
        boolean isResourceAvailable
=ResourceManager.isResourceAvailable("exampleResource");
        assertTrue(isResourceAvailable);
    }
}
```

سوال ۴

از تست واحد (unit testing) برای بررسی برنامه‌های چندنخی (multi-threaded) می‌توان استفاده کرد، ولی با مشکلاتی شامل شرایط رقابتی مواجه خواهیم بود که لزوما بهترین نتجه را نمی‌دهد. تست‌های واحد باید قطعی باشند و این لزوما در برنامه‌های چند نخی رخ نمی‌دهد. یکی از این مشکلات ناشی از اشتراک حافظه و ارسال پیام و ارتباط نخ‌ها هست. به عبارت دیگر هماهنگی و ارتباط بین نخ‌ها است که می‌تواند باعث بروز رفتارهای غیرقابل پیش‌بینی شود. نخ‌های که از حافظه مشترک استفاده می‌کنند اغلب نیاز به محرومیت متقابل با استفاده از قفل‌ها دارند که منجر به مشکلاتی مانند شرایط رقابتی، بن‌بست‌ها و گرسنگی نخ‌ها می‌شود. برای مثال در صورتی که دو نخ از یک منبع مشترک استفاده کنند ممکن است به یکی از نخ‌ها منبع نرسد و باعث fail شدن تست واحد شود.

بنابراین استفاده از تست واحد برای برنامه‌های چندنخی دارای چالش‌هایی هست و ممکن است بعد از هربار اجرا کردن این تست‌ها نتیجه یکسانی نگیریم زیرا این تست‌ها توسط نحوه‌ی زمان‌بندی، دسترسی به منابع و حافظه، ارتباط بین نخ‌ها تحت تاثییر قرار می‌گیرند و نتایج مختلفی را خواهند داد. برای این برنامه‌ها می‌توان از تست‌های دیگری همچون Concurrency Testing یا integration Testing استفاده کرد تا از همگامی و کارکرد درست نخ‌ها اطمینان حاصل کنیم.