محمد مهدی نظری ۹۹۳۱۰۶۱ – ابراهیم صدیقی ۹۹۳۱۰۹۸

۱)

توسعه تست محور (TDD) یک رویکرد توسعه نرم افزار است که در آن تست ها قبل از کد واقعی نوشته می شوند. این فرآیند معمولاً شامل سه مرحله است: نوشتن یک آزمون خطا دار، نوشتن حداقل کد برای قبولی در آزمون، و سپس اصلاح مجدد کد برای بهبود ساختار آن بدون تغییر رفتار. TDD کمک می کند تا اطمینان حاصل شود که کد با الزامات مشخص شده توسط آزمایش ها مطابقت دارد و توسعه دهندگان را تشویق می کند تا روی نوشتن کد تمیز و قابل نگهداری تمرکز کنند. با پیروی از روش‌های TDD، توسعه‌دهندگان می‌توانند باگ‌ها را در اوایل چرخه توسعه پیدا کنند، کیفیت کد را بهبود بخشند و محصول نرم‌افزاری قوی‌تر و قابل اعتمادتری ایجاد کنند.

۲)

در روش چابک، تست پذیرش یک مرحله مهم است که تضمین می کند نرم افزار نیازهای مشتری را برآورده می کند. این شامل تایید اعتبار می شود که آیا نرم افزار مطابق انتظار عمل می کند و userstory تعریف شده در اسپرینت را برآورده می کند. آزمون های پذیرش معمولاً با همکاری ذینفعان نوشته می شود تا انتظارات آنها را برآورده کند. این آزمایش‌ها تا حد امکان خودکار می‌شوند تا فرآیند اعتبارسنجی را ساده‌تر کنند. در طول هر اسپرینت، آزمون‌های پذیرش انجام می‌شود تا تایید شود که افزایش نرم‌افزار با معیارهای پذیرش مطابقت دارد. این رویکرد تکراری امکان بازخورد و تنظیمات مستمر را فراهم می‌کند و تضمین می‌کند که محصول نهایی با نیازها و انتظارات مشتری مطابقت دارد.

۳)

آزمایش پذیری نرم افزار اساساً معیاری است که نشان می دهد چگونه به راحتی می توان خطاهای یک سیستم نرم افزاری را از طریق روش های آزمایش شناسایی کرد. این کارآیی تست را در آشکار کردن هرگونه مشکل یا نقص بالقوه، بدون توجه به کامل بودن فرآیند آزمایش، می سنجد. یک سیستم بسیار قابل آزمایش، سیستمی است که ایجاد معیارهای آزمون واضح و اجرای آزمایش‌ها را برای ارزیابی اینکه آیا آن معیارها برآورده شده‌اند یا خیر، تسهیل می‌کند. به عبارت ساده‌تر، مشخص کردن مشکلات درون نرم‌افزار از طریق آزمایش، حتی زمانی که از کیس‌های آزمایشی به خوبی طراحی شده استفاده می‌شود، بسیار ساده است. هرچه تست پذیری یک سیستم یا جزء بالاتر باشد، احتمال بیشتری وجود دارد که هر گونه خطای پنهان در مرحله آزمایش به طور موثر شناسایی شود، بنابراین به قابلیت اطمینان و کیفیت کلی نرم افزار کمک می کند.

٤)

JUnit یک چارچوب تست پرکاربرد برای زبان برنامه نویسی جاوا است. این در درجه اول برای نوشتن و اجرای تست های خودکار استفاده می شود تا اطمینان حاصل شود که کد جاوا مطابق انتظار عمل می کند. JUnit راه ساده و استاندارد شده ای را برای ایجاد و اجرای موارد آزمایشی ارائه می دهد که آزمایش کدهای خود را برای توسعه دهندگان آسان تر می کند و هر گونه مشکل یا باگ را در مراحل اولیه توسعه شناسایی می کند. با JUnit، توسعه‌دهندگان می‌توانند روش‌های تست را تعریف کنند، آن‌ها را در مجموعه‌های آزمایشی سازماندهی کنند، و آنها را به صورت خودکار اجرا کنند، که امکان آزمایش کارآمد و قابل اعتماد برنامه‌های جاوا را فراهم می‌کند. به طور کلی، JUnit به تضمین کیفیت و قابلیت اطمینان نرم‌افزار جاوا کمک می‌کند تا توسعه‌دهندگان بتوانند تست‌های خودکار را سریع و آسان بنویسند و اجرا کنند.

٥)

ادعای مورد بحث صرفاً بر جنبه ای محدود از وضعیت نهایی، به ویژه اولین عنصر در فهرست، متمرکز است. در نتیجه، اگر یک آزمایش باعث ایجاد یک خطا شود که به سایر بخش‌های حالت نهایی سرایت کند، این شکست ممکن است شناسایی نشده باقی بماند. برای اطمینان از پوشش جامع، اوراکل آزمایشی باید کل لیست را برای مشکلات احتمالی بررسی کند. این نقص را می توان دقیقاً با استفاده از مدل RIPR که مخفف Precision، Intuition، Range و Recall است، توصیف کرد.

- Range: نقص در روش آزمون مربوط به محدوده مقادیر مورد آزمایش است. این آزمون فقط شامل مجموعه محدودی از رشته ها ("Ashley"، "Alex"، "Han"، "Laura") است که به شی "names" اضافه شده است. این محدوده محدود ممکن است برای پوشش همه سناریوهای ممکن، مانند موارد لبه یا کاراکترهای خاص که می‌تواند بر الگوریتم مرتب‌سازی تأثیر بگذارد، کافی نباشد.
- Intuition: نقص بر شهود پشت آزمون نیز تأثیر می گذارد. با افزودن عناصر به ترتیبی خاص ("لورا"، "هان"، "الکس"، "اشلی") و سپس بیان اینکه "الکس" باید اولین عنصر پس از مرتب‌سازی باشد، آزمون یک نتیجه از پیش تعیین شده را در نظر می‌گیرد. این رویکرد فاقد انعطاف‌پذیری است و ممکن است تغییرات در الگوریتم‌های مرتب‌سازی یا رفتار غیرمنتظره را در نظر نگیرد.
- Precision: دقت تست به دلیل نقص در روش تست به خطر افتاده است. این آزمون تنها بر روی موقعیت مورد انتظار "الکس" پس از مرتب‌سازی تمرکز می‌کند و از سایر جنبه‌های مهم مانند ثبات، منحصربه‌فرد بودن یا عملکرد الگوریتم مرتب‌سازی غفلت می‌کند. یک آزمون دقیق تر، طیف وسیع تری از ورودی ها و نتایج مورد انتظار را در نظر می گیرد.
- Recall: این نقص بر توانایی آزمون برای یادآوری مسائل احتمالی در متد sort() تأثیر می گذارد. از آنجایی که تست به طور محدود برای یک سناریوی خاص تنظیم شده است، ممکن است در تشخیص رگرسیون ها یا موارد گوشه ای که ممکن است در اجرای مرتب سازی ایجاد شود، شکست بخورد. یک آزمون جامع با یادآوری بهتر ترکیب های ورودی مختلف و نتایج مورد انتظار را در بر می گیرد.

٦)

a) هنگامی که متد quals() در یک کلاس نادیده گرفته می شود، بسیار مهم است که متد hashCode() را نیز برای حفظ قرارداد بین این دو متد نادیده بگیریم. عدم انجام این کار می تواند منجر به رفتار غیرمنتظره در هنگام ذخیره نمونه های کلاس در ساختارهای مجموعه مانند HashSet شود. بیایید این موضوع را با کلاس Point توضیح دهیم:

```java
import java.util.HashSet;

class Point {
    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null || getClass() != obj.getClass()) {
            return false;
        }
        Point point = (Point) obj;
        return x == point.x && y == point.y;
    }
}

public class Main {
    public static void main(String[] args) {
        HashSet<Point> pointSet = new HashSet<>();
        Point p1 = new Point(1, 2);
        Point p2 = new Point(1, 2);

        pointSet.add(p1);
        pointSet.add(p2);

        System.out.println(pointSet.size()); // Output: 2 (U
    }
}
```

در مثال بالا، حتی اگر p۱ و p۲ بر اساس متد quals() برابر هستند، اما به عنوان عناصر متمایز در HashSet در نظر گرفته می‌شوند، زیرا متد hashCode برای تولید کدهای هش برابر برای اشیاء مساوی لغو نشده است.

b) رابطه ریاضی بین متدهای quals() و hashCode() به این صورت است که اگر دو شیء طبق متد quals() برابر باشند، کدهای هش آنها نیز باید برابر باشند. این تضمین می‌کند که اشیاء مساوی کد هش یکسانی را تولید می‌کنند و در مجموعه‌های مبتنی بر هش یکنواختی حفظ می‌شود.

c) در اینجا یک تست ساده JUnit برای نشان دادن مشکل عدم اجرای صحیح hashCode() اشیاء Point وجود دارد:

```java
import org.junit.Test;
import static org.junit.Assert.*;

public class PointTest {

    @Test
    public void testPointEquality() {
        Point p1 = new Point(1, 2);
        Point p2 = new Point(1, 2);

        assertEquals(p1, p2); // Fails due to missing hashCode()
    }
}
```

d) برای رفع خطا در کلاس Point، باید متد hashCode() را نادیده بگیریم تا کدهای هش را بر اساس فیلدهای شی تولید کنیم. در اینجا کلاس Point تصحیح شده است:

```java
import java.util.Objects;

class Point {
    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null || getClass() != obj.getClass()) {
            return false;
        }
        Point point = (Point) obj;
        return x == point.x && y == point.y;
    }

    @Override
    public int hashCode() {
        return Objects.hash(x, y);
    }
}
```

e) برای ارزیابی کلاس Point تصحیح شده، می توانیم آزمون JUnit را به عنوان یک نظریه JUnit با استفاده از DataPoint های مناسب بازنویسی کنیم:

```java
import org.junit.Test;
import org.junit.experimental.theories.DataPoint;
import org.junit.experimental.theories.Theories;
import org.junit.runner.RunWith;
import static org.junit.Assert.*;

@RunWith(Theories.class)
public class PointTest {

    @DataPoint
    public static Point p1 = new Point(1, 2);

    @DataPoint
    public static Point p2 = new Point(1, 2);

    @Test
    public void testPointEquality() {
        assertEquals(p1, p2); // Succeeds with correct hashCode()
    }
}
```

با اجرای صحیح متد hashCode()، کلاس Point اکنون تضمین می‌کند که اشیاء مساوی کد هش یکسانی را تولید می‌کنند، مشکل مشاهده شده در HashSet را حل می‌کند و عملکرد مناسب در ساختارهای مجموعه را فعال می‌کند.

۷)

الف) خیر، ایجاد یک تست واحد برای خود رابط «PaymentService» منطقی نیست. تست‌های واحد برای تأیید رفتار پیاده‌سازی یک کلاس یا اینترفیس است، نه تعریف خود رابط.

یک رابط در جاوا به عنوان قراردادی عمل می کند که متدهایی را که کلاس های پیاده سازی باید ارائه کنند، تعریف می کند. این شامل هیچ منطق پیاده سازی نیست، بنابراین هیچ رفتاری برای آزمایش در خود رابط وجود ندارد.

همچنین اینترفیس ها ماهیت ثابتی دارند و هیچگونه رفتار زمان اجرا ندارند. آنها فقط امضاها و ثابت های متد را تعریف می کنند. آزمون‌های واحد معمولاً برای تأیید رفتار مؤلفه‌های پویا با رفتار زمان اجرا، مانند کلاس‌هایی که رابط‌ها را پیاده‌سازی می‌کنند، استفاده می‌شوند.

بنابراین، نوشتن تست‌های واحد برای کلاس‌هایی که واسط «PaymentService» را پیاده‌سازی می‌کنند، مانند کلاس «PaymentServiceImpl» مناسب‌تر است. این آزمایش‌ها تأیید می‌کنند که پیاده‌سازی رابط طبق قرارداد خود به درستی عمل می‌کند.

ب) سناریو های تست باید حالت موفقیت و حالات شکست متدهارا در بر بگیرند. سناریوها شامل موارد زیر است:

۱. برداشت موفقیت آمیز (Successful Withdrawal): بررسی اینکه کمیسیون بتواند با موفقیت برداشت شود. بررسی کنید که روش "true" را برمی گرداند و عملیات برداشت به درستی انجام شده است.

۲. برداشت ناموفق با اکشپسن (IOException): تست رفتار زمانی که "IOException" در طول عملیات برداشت رخ می دهد.

۳. برداشت ناموفق با اکسپشن (DuplicateTransactionId): آزمایش رفتار زمانی که یک استثنا 'DuplicateTransactionId' در طول عملیات برداشت رخ می دهد یعنی زمانی که آیدی تراکنش تکراری است.

٤. برداشت ناموفق با اکسپشن (InvalidAmount): آزمایش رفتار زمانی که "InvalidAmountException" در طول عملیات برداشت رخ می دهد یعنی زمانی که مبلغ تراکنش از نوع معتبر یعنی متغیر دابل نیست.

٥. برداشت ناموفق با اکسپشن (InsufficientFund): آزمایش رفتار زمانی که "InsufficientFundException" در طول عملیات برداشت رخ می دهد یعنی زمانی که موجودی حساب برای انجام تراکنش کافی نباشد.

٦. بازگشت موفقیت آمیز: تست بازگرداندن کمیسیون که بتواند با موفقیت انجام شود. بررسی اینکه "true" را برمی گرداند و عملیات بازگشت به درستی انجام شده است.

۷. بازگرداندن ناموفق با اکسپشن (IOException): آزمایش رفتار زمانی که یک «IOException» در طول عملیات بازگشت رخ می دهد.

۸. بازگرداندن ناموفق با اکسپشن (InvalidTransactionId): آزمایش رفتار برنامه زمانی که یک استثنا 'InvalidTransactionId' در طول عملیات بازگشت رخ می دهد یعنی آیدی تراکنش از نوع تعریف شده نیست.

۹. بازگرداندن ناموفق با اکسپشن (AlreadyReversed): آزمایش رفتار زمانی که یک «AlreadyReversedException» در طول عملیات بازگشت رخ می دهد یعنی عمل بازگشت قبلا رخ داده است.

۱۰. تولید شناسه تراکنش منحصربفرد: تست که شناسه تراکنش ایجاد شده برای هر فراخوانی روش "withdrawCommission" منحصر به فرد باشد.

ج) هنگام نوشتن تست‌های واحد برای کلاس «PaymentServiceImpl»، توصیه نمی‌شود که مستقیماً یک سرویس خارجی برای «Invoker» را کال کنیم.

هدف تست های واحد جداسازی واحد کد تحت آزمایش است که در این مورد کلاس «PaymentServiceImpl» است. تماس با یک سرویس خارجی وابستگی های خارجی را معرفی می کند که می تواند آزمایش را پیچیده کند و آن را کمتر قابل اعتماد کند.

همچنین تست های واحد باید سریع و قابل اعتماد باشد و به توسعه دهندگان این امکان را می دهد که به سرعت کد را تکرار کرده و با اطمینان مجدداً اصلاح کنند. تماس با یک سرویس خارجی می‌تواند اجرای آزمایش را کندتر کند و به دلیل تأخیر شبکه، در دسترس بودن سرویس و مشکلات سازگاری داده‌ها، آزمایش‌ها را شکننده‌تر کند.

به جای فراخوانی سرویس خارجی واقعی، باید از تکنیک هایی مانند Mock یا Stub برای شبیه سازی رفتار «فرستنده» در تست های واحد خود استفاده کنید. این به شما این امکان را می‌دهد که رفتار «Invoker» را کنترل کنید و فقط بر روی آزمایش منطق در کلاس «PaymentServiceImpl» تمرکز کنید. این رویکرد کمک می کند تا تست های واحد متمرکز، قابل اعتماد و سریع باشد.

د) کد این قسمت ضمیمه شده است.