دوره کیفیت نرم‌افزار

آزمون واحد در جاوا
Unit Testing in Java

صادق علی‌اکبری

سرفصل مطالب

- نیاز به تست
- رویکردهای آزمایش نرم افزار
- معرفی آزمون واحد
- آشنایی با JUnit
- مزایای استفاده از آزمون واحد
- ویژگی‌های تست خوب
- تمرین عملی

اهمیت و لزوم آزمایش محصول

- تولیدکننده، قبل از تحویل محصول باید از کیفیت آن مطمئن شود
  - خودروساز، خودرو را قبل از تحویل به مشتری ارزیابی می‌کند
  - آشپز، قبل از مهمان غذا را می‌چشد
- کنترل کیفیت و تضمین کیفیت در صنایع مختلف مورد توجه است
- سهل‌انگاری در تست محصول، گاهی ویرانگر است

سهل‌انگاری در ارزیابی کیفیت محصول

- سال ۱۹۹۸: برج «کاخ۲» در سائوپائولو فرو می‌ریزد
  - سه سال پس از ساخت
  - شش نفر کشته شدند
  - علت: کیفیت پایین بتن مورد استفاده

- آتش‌سوزی خودروهای پژو ۴۰۵
  - ده‌ها نفر کشته شدند
  - علت: استفاده از قطعه بی‌کیفیت در سیستم سوخت‌رسانی

سهل‌انگاری در ارزیابی نرم‌افزار

- موشک آریان۵ در سال ۱۹۹۶ توسط آژانس فضایی اروپا آزمایش شد
  - این موشک، ۴۰ ثانیه پس از پرتاب منفجر شد
  - هزینه: ۳۷۰ میلیون دلار
  - علت: وجود یک اشکال در نرم‌افزار
- سال ۱۹۸۰: پنج بیمار بر اثر دریافت مقدار زیاد پرتوی X جان باختند
  - علت: اشکال در نرم‌افزار ماشین پرتودرمانی

نیاز به آزمایش نرم‌افزار

- نرم‌افزار، مثل هر محصول دیگری، باید آزمایش شود
  - تا از کیفیت آن مطمئن شویم
- نرم افزاری که آزمایش نشده، هنوز کامل نیست
- انواع آزمایش‌ها، کیفیت نرم‌افزار را از دیدگاههای مختلف می‌آزمایند

آزمایش نرم‌افزار

- ویژگی‌های یک نرم‌افزار خوب چیست؟
  - عملکرد صحیح
  - ویژگی‌های کیفی (غیرعملکردی)
    - کارایی، سرعت، سهولت استفاده، امنیت و غیره
- تست نرم‌افزار : فرایندی برای آزمایش ویژگی‌های موردنظر نرم‌افزار
- انواع مختلفی از تست نرم‌افزار وجود دارد
- انواع تست، ویژگی‌های مختلف نرم‌افزار را ارزیابی می‌کنند

ابعاد آزمون نرم‌افزار

- سطح آزمون (آزمون واحد، آزمون یکپارچگی، ...، آزمون سیستم)
- نوع آزمون (آزمون عملکرد، آزمون ویژگی‌های کیفی)
- روش آزمون (white box یا black box)
- شکل آزمون (آزمون خودکار، آزمون دستی)
- نقش آزمون‌گر (برنامه‌نویس، تیم تست، کاربر یا ...)

آزمون واحد نرم‌افزار (Unit Testing)

- انواع مختلفی از تست در طول عمر یک پروژه انجام می‌شوند
  - برخی از این تست‌ها به دخالت مستمر کاربر، طراح یا مشتری نیاز دارد
  - برخی از تست‌ها نیز در تیم کنترل کیفیت اجرا می‌شوند
- اما آزمون واحد **توسط برنامه‌نویس** و برای برنامه‌نویس انجام می‌شود
  - جزو وظایف برنامه‌نویس است
- البته آزمون واحد کافی نیست
  - انواع دیگر آزمون برای تضمین کیفیت نرم‌افزار لازم است.

معنای آزمون واحد

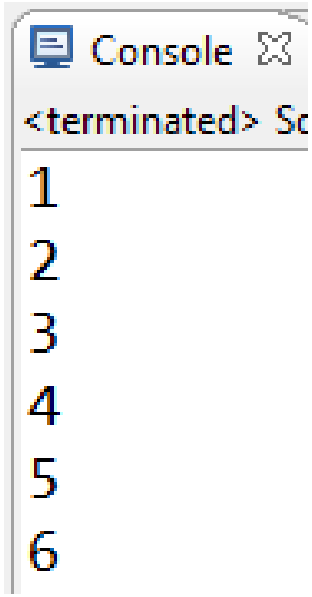
- آزمون یک واحد (بخش) کوچک از برنامه
  - مثلاً یک متد یا یک کلاس
- برای اطمینان از صحت عملکرد این واحد
  - آیا این متد درست عمل می‌کند؟
  - به ازای ورودی‌های مختلف، خروجی/رفتار مناسب تولید می‌کند؟
- هر بخش و جزء برنامه را جداگانه آزمایش می‌کنیم
  - قبل از آزمایش کل سیستم
  - مثال خودروسازی: آزمایش قطعات، قبل از آزمایش کل خودرو

روش سنتی آزمون واحد

- فرض کنید: یک متد نوشتیم که یک آرایه را مرتب می‌کند (sort)
- یک متد main برای کلاس آن می‌نویسیم
- چند حالت از ورودی‌های مختلف را امتحان می‌کنیم
- خروجی‌ها را چاپ می‌کنیم (system.out.println)
- صحت خروجی‌ها را به صورت دستی (چشمی) بررسی می‌کنیم
- کدهای تست نوشته شده را حذف می‌کنیم و کدهای دیگری می‌نویسیم
- نکته:
  - Business Code: متن اصلی برنامه‌ها
  - Test Code: کدهایی که برای آزمایش برنامه‌ها نوشته شوند

روش سنتی آزمون واحد

```
public static void sort(int[] values){
    Arrays.sort(values);
}
public static void main(String[] args) {
    int[] array = {3,2,1,5,6,4};
    sort(array);
    for (int i = 0; i < array.length; i++) {
        System.out.println(array[i]);
    }
}
```



- اشکال این روش چیست؟

معایب روش سنتی

- تست‌های نوشته شده دور ریخته می‌شود
  - نقض استفاده مجدد (software reuse) برای test code
- در هر لحظه یک تست انجام می‌شود
- برنامه نویس باید به صورت دستی تست‌ها را اجرا کند
  - اجرای تست‌ها خودکار نیست
- برنامه نویس باید شخصاً از صحت آنها مطمئن شود
  - تشخیص موفقیت آمیز بودن تستها خودکار نیست

ویژگی‌های آزمون واحد (Unit Test)

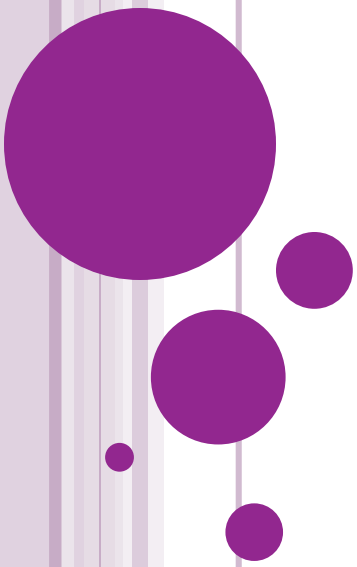
- اجرای خودکار
- تشخیص خودکار موفقیت تست
- قابل تکرار و استفاده مجدد

چارچوب‌های xUnit

- چارچوب‌هایی برای آزمون واحد در زبان‌های مختلف ایجاد شده است
- این ابزارها، **اجرا** و **تشخیص** نتیجه آزمون را خودکار می‌کنند
- مجموعه این ابزارها xUnit نامیده می‌شوند. مثال:
  - CppUnit
  - **JUnit**

خلاصه: آزمون واحد نرم‌افزار

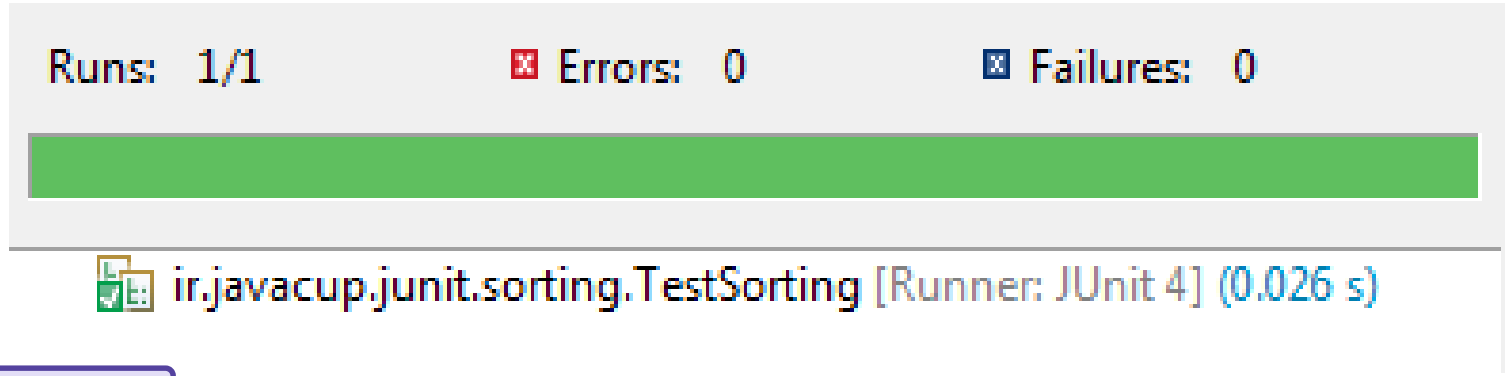
- آزمون white box (نه black box)
- آزمون واحد (نه آزمون سیستم، و نه حتی آزمون چند بخش یکپارچه شده)
- [معمولاً] آزمون عملکردی (نه آزمون ویژگی‌های کیفی)
- آزمون خودکار (نه دستی)
- تولید توسط خود برنامه‌نویس (نه توسط تیم تست یا کاربر)



آشنایی با JUnit

یک نمونه آزمون واحد در JUnit

```
public static void sort(int[] values){..
```




```
@Test
public void testSort(){
    int[] array = {3,2,1,5,6,4};
    sort(array);
    int[] sortedArray = {1,2,3,4,5,6};
    assertArrayEquals(array, sortedArray);
}
```

اجرای آزمون واحد

- هر متد تست با حاشیه @Test مشخص می‌شود
- هر متد تست یک نمونه‌آزمون (test-case) خوانده می‌شود
- نمونه‌آزمون‌ها متد main ندارند و با کمک TestRunner اجرا می‌شوند
  - غالباً از طریق محیطهای برنامه‌نویسی (مثل Eclipse) یا Maven اجرا می‌شوند
- یادآوری: موفقیت‌آمیز بودن آزمون باید به صورت خودکار بررسی شود
  - نه به صورت دستی: سندروم println
- نتیجه آزمون در JUnit با کمک assertion بررسی می‌شود
  - مثال: assertEquals

نمونه‌آزمون

- هر نمونه‌آزمون شامل این بخش‌هاست:
  - یک ورودی برای متد موردآزمون تعیین می‌شود (test data)
  - خروجی و رفتار موردنظر برای این ورودی مشخص می‌شود (expected result)
  - متد موردآزمون با این ورودی فراخوانی می‌شود (invocation)
  - خروجی و رفتار متد با خروجی موردنظر تطبیق داده می‌شود (assertion)
- اگر اجرای تست موفقیت‌آمیز باشد: تست pass شده است
- اگر اجرای تست موفقیت‌آمیز نباشد: تست fail شده است

JUnit Assertions مجموعه

- assertNull(x)
- assertNotNull(x)
- assertTrue(boolean x)
- assertFalse(boolean x)
- assertEquals(x, y)
  - x.equals(y)
- assertSame(x, y)
  - x ==y
- assertNotSame
- fail()
- ,…

مثال: Business Code

```
public class MyMath {
    private static final MyMath instance = new MyMath();

    public static MyMath getInstance() {
        return instance;
    }

    public int division(int a, int b){
        if(b==0)
            throw new ArithmeticException("Division by Zero");
        return a/b;
    }

    public int multiply(int a, int b) {
        return a*b;
    }
}
```

مثال: Test Code

```java
@Test
public void testDivision(){
    MyMath math = MyMath.getInstance();
    assertNotNull(math);
    assertEquals(math.division(4, 2), 2);
    assertEquals(math.division(5, 3), 1);
    try{
        math.division(5, 0);
        //The test should never reach this line:
        fail();
    }catch(Exception e){
        // The Exception is OK
        assertTrue(e instanceof ArithmeticException);
    }
}
```

سایر امکانات JUnit

- متد setup (با **@Before** مشخص می‌شود)
  - قبل از هر متد تست اجرا می‌شود
  - مسؤول کارهایی که قبل از اجرای هر تست لازم است
  - مثل اتصال به پایگاه داده، مقداردهی به فیلدها، ...
- **@BeforeClass** : یک بار قبل از همه تست‌ها
- متد teardown (**@After**)
  - اجرا بعد از هر متد تست
  - مثل بستن اتصال به پایگاه داده، بستن فایل، ...
- **@AfterClass** : یک بار بعد از همه تست‌ها

فرایند اجرای یک نمونه آزمون

```
public class TestDB {
    private static Database db ;
    private static List<String> storage ;
    @BeforeClass
    public static void startup(
        db = new Database();
        storage = new ArrayList
        db.setStorage(storage);
    }
    @AfterClass
    public static void shutdown(){
        db = null;
        storage = null;
    }
    @Before
    public void setup(){
        db.open();
    }
    @After
    public void teardown(){
        db.close();
    }
```

```
@Test
public void testNullValue(){
    db.insert(null);
    assertTrue(storage.contains(null));
    assertEquals(1,storage.size());
}
```

```
@Test
public void testNormal(){
    String name = "Ali Alavi";
    db.insert(name);
    assertTrue(storage.contains(name));
    db.delete(name);
    assertFalse(storage.contains(name));
}
```

کوییز

```
Startup
Setup
TestDiv
TearDown
Setup
TestMult
TearDown
Shutdown
```

کوییز

- اجرای تست زیر چه نتیجه‌ای را در خروجی چاپ می‌کند؟

```
public class MyMathTest {
    @BeforeClass
    public static void startup(){System.out.println("Startup");}
    @AfterClass
    public static void shutdown(){System.out.println("Shutdown");}
    @Before
    public void setup(){System.out.println("Setup");}
    @After
    public void tearDown(){System.out.println("TearDown");}
    @Test
    public void testDiv(){System.out.println("TestDiv");}
    @Test
    public void testMult(){System.out.println("TestMult");}
}
```

تمرین عملی

- نوشتن یک unit test

مزایای آزمون واحد

مزایای آزمون واحد

- بهبود کیفیت برنامه‌ها
- کاهش اشکالات (bugs)
  - وقت کمتری به خاطر debugging هدر میدهد
- به تأخیر نیافتادن زمان کشف اشکالات برنامه
- بهبود ساختار و طراحی برنامه
- افزایش اطمینان به اجزای برنامه و سطوح پایین کد
  - که سنگ بنای سطوح بالاتر هستند
  - بعد از تولید هر واحد
  - بعد از تغییر هر واحد
- مستند گویا، زنده و قابل اجرا برای برنامه

آزمون واحد و مستندسازی

- یک آزمون واحد یک مستند قابل اجراست
- نحوه استفاده از کد اصلی را مشخص می‌کند
- و نشان می‌دهد در شرایط مختلف رفتار متد/کلاس باید چگونه باشد
- برتری یک مستند قابل اجرا، نسبت به یک مستند معمولی (متنی):
  - مستند قابل اجرا صحیح است!
  - یک مستند متنی ممکن است با اصل برنامه تطابق نداشته باشد

زمان تولید آزمون واحد

- قبل یا بلافاصله بعد از نوشتن «واحد» (برنامه)
  - نه در انتهای تولید پروژه
- به تأخیر انداختن زمان تولید آزمون واحد باعث:
  - کاهش دقت آزمون
  - کاهش تمرکز روی برنامه موردتست
  - از دست رفتن مزایای جانبی مثل بهبود کیفیت کد
  - خسارت ناخواسته (Collateral Damage)
    - اختلال یا اشکالی که در اثر رفع یک اشکال دیگر در سیستم ظاهر شود
    - اگر تست، به انتهای پیاده‌سازی پروژه موکول شود، این پدیده رایج می‌شود

دردسر آزمون

- برخی از برنامه‌نویسان آزمون واحد را یک دردسر می دانند
  - و حتی مدیران
- دلایلی که بر ضد آزمون می‌آورند:
  - وقت نداریم، پروژه از زمانبندی عقب است!
  - زمان زیادی برای نوشتن نمونه‌تست‌ها هدر می رود
  - تست، کار و وظیفه من نیست

زمانی که صرف آزمون می‌شود، هدر نمی‌رود

- اگر در هنگام پیاده سازی، تست‌های لازم نوشته شود، وقت زیادی گرفته نخواهد شد
- اگر هنوز این زمان را زیاد می‌دانید به این سوالها فکر کنید:
  - چقدر زمان به خاطر debugging صرف می‌کنید؟!
  - چقدر وقت صرف تعمیر بخشهایی می‌کنید که ظاهراً صحیح بودند؟!
    - ولی اشکالات بزرگی داشته‌اند
  - چقدر زمان صرف رفع هر اشکال (bug)گزارش شده می‌کنید؟!
  - هزینه بروز اشکال در هنگام استفاده کاربر نهایی چقدر است؟!

کیفیت برنامه‌های آزمون

خودکار بودن آزمون واحد

- در تمام طول عمر پروژه، تمام نمونه‌آزمون‌ها (test-case) باید پاس شوند
- هر گاه یک نمونه‌آزمون fail شود:
  - فرآیند اضافه کردن یا گسترش دادن کد باید متوقف شود
  - تا زمانی که اشکال مورد نظر رفع شود
- هر نمونه‌آزمون باید خودش مشخص کند که pass یا fail شده
  - نه این که مثلاً در خروجی چیزی بنویسد و فرد دیگری مسئول پردازش این خروجی باشد
- بنابراین عملیات اجرای آزمون، خودکار می‌شود

قواعد تولید آزمون واحد

- آزمون‌های بدیهی برای بخش‌های قطعاً صحیح ایجاد نکنید
  - مثلاً getter و setter ها
- ورودی‌های مختلف را آزمایش کنید
  - ورودی‌های معمولی
  - ورودی‌های مرزی (مثلاً مقدار صفر)
  - ورودی‌های خاص (null ، مقدار منفی، آرایه خالی و ...)
- برای بخش‌های حساستر و مهمتر، پوشش و کیفیت آزمون‌ها را بالا ببرید

وقتی یک اشکال در برنامه کشف/گزارش می‌شود

- پس نمونه‌آزمون‌ها کامل نیستند
- اگر کامل بودند، قبل از انتشار اشکال، نمونه‌آزمون متناظر fail می‌شد
- فرایند رفع اشکال

1. نمونه‌آزمونی نوشته شود که در شرایط گزارش شده fail می‌شود
2. برنامه اصلی اصلاح شود
3. آزمون‌های واحد مجدداً اجرا شوند

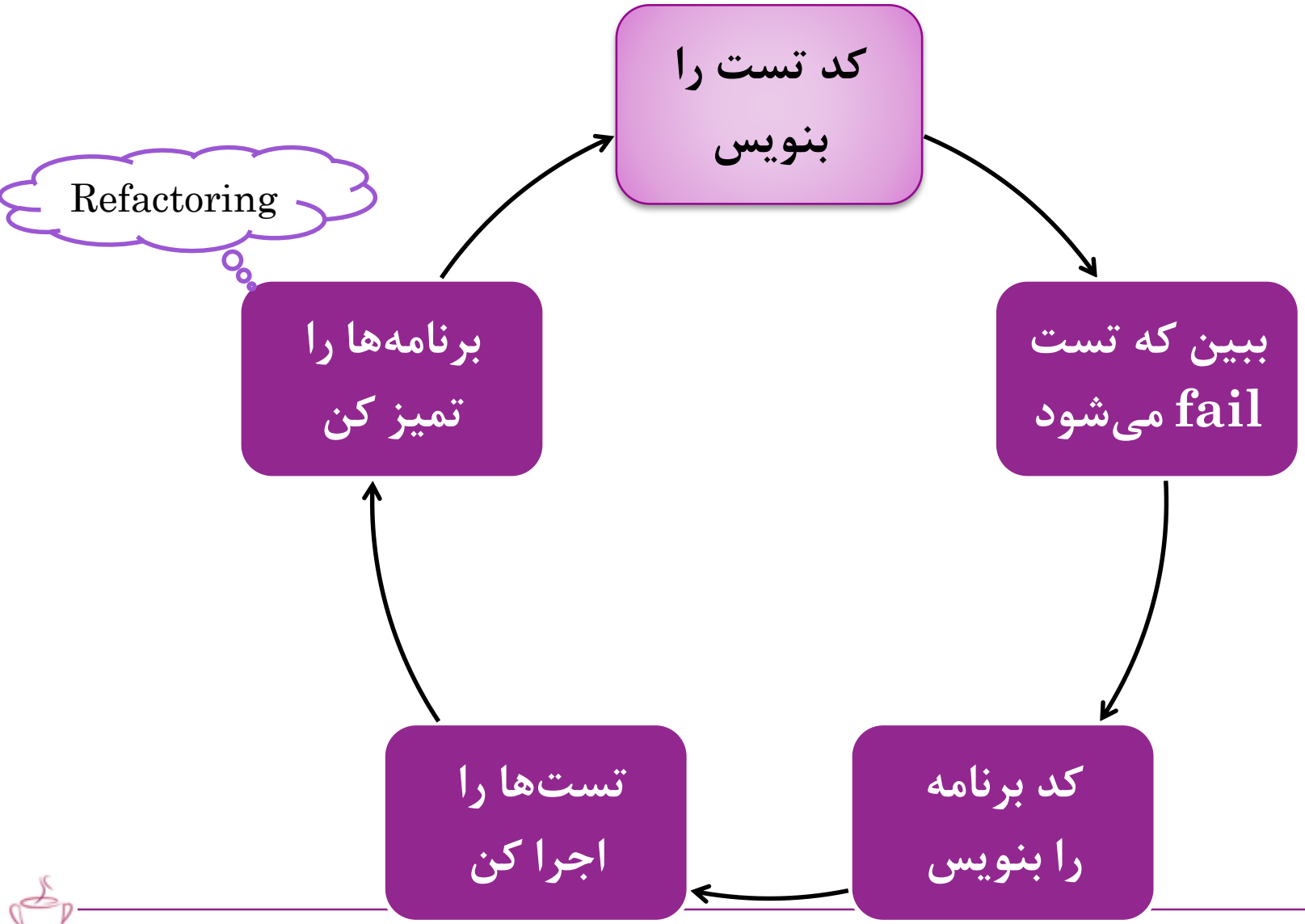
ویژگیهای آزمون‌های خوب

- برنامه‌های تست، مثل برنامه‌های اصلی مهم هستند
  - Test code is real code
  - باید با کیفیت تولید و نگهداری شوند
  - وگرنه زمان زیادی برای نگهداری خود تستها هدر می‌رود
- ویژگی‌های تست خوب:
  - خودکار (Automated)
  - کامل (Thorough)
  - قابل تکرار (Repeatable)
  - مستقل (Independent)
  - حرفه‌ای (Professional)

برنامه‌نویسی مبتنی بر تست

- Test-Driven Development (TDD)
  - در این رویکرد، تست‌ها **قبل از نوشتن کد** نوشته می‌شوند
- Test-First Development
  - در این روش، قبل از نوشتن متد به نحوه استفاده از آن فکر می‌کنیم
  - البته بحثهای فراوانی درباره مزایا و معایب این رویکرد وجود دارد

برنامه‌نویسی مبتنی بر تست (TDD)



مطالب فنی و جزئیات راه‌اندازی

اجرای تست‌ها در Eclipse

- نکته: JUnit جزئی از JavaSE (JDK یا JRE) **نیست**
  - یک کتابخانه مجزاست (**junit.jar**)
- انجام عملی فرایند ایجاد و اجرای تست در Eclipse
- فرض می‌کنیم business code موجود است
- این فرایند در سایر محیطهای برنامه‌سازی نیز به سادگی قابل اجراست
  - IntelliJ IDEA، NetBeans و ...
- دقت به import ها و import static ها
- تغییر در business code منتج به test case failure
  - مزیت وجود test-case ها

جمع‌بندی

مطالعه تکمیلی

- JUnit و Maven
- @Parameters, @Rule, @Category, ...
- الگوها و بایدهای تولید تست‌های باکیفیت
- سایر ابزارهای آزمون واحد در جاوا (مثل DBUnit)
- چارچوب‌های Mock (مانند JMockit)
- ابزارهای محاسبه Test Coverage
- ابزارهای گزارش‌دهی درباره وضعیت تست‌ها
  - مانند Jenkins و SonarQube
- TestNG

جمع‌بندی

- انواع آزمون نرم‌افزار
- آزمون واحد (توسط برنامه‌نویس، به صورت خودکار)
- امکانات JUnit : assertions، @Test، @Before، ...
- مزایای تست
- عدم انجام آزمون واحد، یک «وام فنی» (technical debt) است
  - این کار، صرفه‌جویی در زمان یا هزینه نیست
  - به تأخیر انداختن هزینه (مثل بازپرداخت وام) و افزایش هزینه (مثل پرداخت سود وام) است
- طبق یک آمار: مهمترین مهارت برنامه‌نویسان جاوا در میان ابزارها، ابزار JUnit است
- آزمون واحد موضوعی تفننی و تزیینی نیست. بلکه یک **باید** است
- آزمون واحد، مثل سایر بایدها (طراحی مناسب، مستندسازی و ...) لازم است
  - اما میزان، پوشش، زمان و هزینه آن، با توجه به حساسیت، بودجه و ... قابل تنظیم است

خاتمه: نقل قولی از Michael C. Feathers

- کد بدون تست، کد بدی است
- مهم نیست که چقدر خوب نوشته شده
- مهم نیست که چقدر ساختارمند، شیءگرا یا تمیز نوشته شده
- با وجود کدهای تست، می‌توانیم رفتار برنامه را به نحو **قابل تأیید** و با سرعت تغییر دهیم
- بدون کدهای تست، واقعاً نمی‌دانیم که برنامه‌های ما در حال بهتر شدن هستند، یا بدتر شدن...

Legacy code is code without tests...

پایان

سایر مطالب

آزمون نرم‌افزار در مقایسه با سایر صنایع

- نرم‌افزار، یک محصول نرم است
- امکان شبیه‌سازی شرایط سخت واقعی بدون صدمه دیدن اصل نرم‌افزار
- برج میلاد را در مقابل شبیه‌سازی زلزله ۸ ریشتری آزمایش نمی‌کنند
  - اگر دوام بیاورد، آسیب می‌بیند
- یک نرم‌افزار مبتنی بر وب را می‌توان در مقابل **شبیه‌سازی** حجم شدید درخواست کاربر آزمایش کرد
  - نرم‌افزار، آسیب نمی‌بیند

کیفیت نرم‌افزار

- تضمین کیفیت
  - فرایندی که در چرخه تولید نرم‌افزار نقش ایفا می‌کند
- کنترل کیفیت
  - یکی از مراحل پایانی که برخی جنبه‌های کیفیت را می‌سنجد

ویژگی‌های تست خوب – خودکار

- خودکار بودن حداقل از دو جنبه:
  - اجرای تست‌ها
  - بررسی نتایج (pass/fail)
- معمولاً در پروژه‌ها مجموعه تست‌ها در یک سرور اجرا می‌شوند
  - به صورت زمانبندی شده و خودکار
  - مثلاً هر نیمه‌شب، یا با هر تغییر در برنامه (code commit)

ویژگی‌های تست خوب – کامل

- تست‌های خوب کامل هستند
- هر آن چه ممکن است اشتباه باشد، را آزمایش می‌کنند
- سیاستهای مختلف:
  - همه جریان‌های ممکن (if ها ، exception ها و ...) بررسی شوند
  - فقط موارد اصلی و خروجی‌ها را تست کنیم و از جزئیات صرف نظر کنیم
- انتخاب سیاست: وابسته به میزان حساسیت برنامه موردآزمون
- هسته مرکزی برنامه که همه بخشها به آن وابسته هستند: سیاست اول
- یک پروژه کوچک با زمان محدود: سیاست دوم

پوشش تست

- بهره‌گیری از ابزارهای Test Coverage
- این ابزارها نشان می‌دهند چه نسبتی از متن برنامه تحت تست قرار گرفته است
- میزان branchها و exceptionهایی که در تستها مد نظر قرار گرفته‌اند را نیز محاسبه می‌کنند
- هر چه پوشش تست بیشتر باشد، گزارش اشکالات برنامه کمتر خواهد بود
  - Reported Bugs
- و برنامه‌هایی که اشکالات گزارش‌شده زیادی دارند، محکوم به بازنویسی هستند

ویژگی‌های تست خوب – قابل تکرار

- تست‌ها قرار است به دفعات اجرا شوند
- بنابراین نباید وابسته به هیچ چیز در خارج از خودشان باشند
- هر آن‌چه تکرار اجرای تست را محدود می‌کند، باید حذف شود
- تستها به هر ترتیبی ممکن است اجرا شوند
- در هر ترتیب باید همان خروجی را تولید کنند

ویژگی‌های تست خوب – استقلال

- هر نمونه‌آزمون، چیزی مستقل از نمونه دیگر را آزمایش کند
  - تا با fail شدن تست، دقیقاً بفهمیم کدام قسمت برنامه مشکل دارد
- این خاصیت به تکرارپذیری تست‌ها کمک می‌کند
- باید بتوانیم آنها را در هر زمان و با هر ترتیبی اجرا کنیم

ویژگی‌های تست خوب – حرفه‌ای

- کدهای تست باید همان‌قدر که کد اصلی حرفه‌ای نوشته می‌شوند، حرفه‌ای و با دقت نوشته شوند
  - تولید کد تست جدی گرفته شود
  - یک کار سرسری انگاشته نشوند
- آزمون‌های واحد تمام ویژگیهای یک طراحی خوب را باید داشته باشند
  - Encapsulation
  - Low coupling
  - High cohesion
- Test code is real code!

اشیاء بدلی (Mock Objects)

- هدف از آزمون واحد، فقط آزمون یک «واحد» است
  - نه واحدهای وابسته به آن
  - نه آزمون یکپارچگی
- بخشهایی که واحد موردآزمون به آن وابسته هستند، نباید در آزمون مؤثر باشند
- راهکار حذف این وابستگی‌ها: استفاده از بدل (Mock) به جای اصل مؤلفه
- چارچوب‌هایی برای این کار به وجود آمده‌اند
  - کار بدلی کردن مؤلفه‌ها را ساده می‌کنند
  - در برنامه اصلی تغییر نمی‌دهند
  - مانند Mockito ، Jmockit ، PowerMock و ...

سؤالات رایج

- *زمان کافی برای نوشتن آزمون واحد نداریم*

پاسخ: شما با ننوشتن تست‌ها، در زمان صرفه‌جویی نمی‌کنید.

ماجرا را میان‌مدت و درازمدت ببینید

- *نوع پروژه، طوری است که نمی‌توان برای آن آزمون واحد نوشت*
  - *مثلاً یک web application است*

پاسخ: هر پروژه، از اجزاء و واحدهایی تشکیل می‌شود که قابل تست است

نقل قول

Michael Feathers

- Legacy code is code without tests
- Code without tests is bad code. It doesn't matter how well written it is; it doesn't matter how pretty or object-oriented or well-encapsulated it is. With tests, we can change the behavior of our code quickly and verifiably. Without them, we really don't know if our code is getting better or worse.
- So you might be thinking "ok, all it takes to refactor bad code is to add tests". The problem is that writing tests on top of bad code is also terribly hard. In technical terms, bad code is tightly coupled and it has low cohesion. So this is the catch 22 of fixing legacy code: to refactor, you need tests, to test, you need to refactor.

لطیفه

- مدیر پروژه: کلاسی که باید می‌نوشتی، کامل شد؟

- برنامه‌نویس: بله من کارم تمام شده.

- مدیر: آفرین! تست‌ها هم پاس می‌شوند؟

- برنامه‌نویس: هنوز تست ننوشته‌ام!

تمرین

- در محیط برنامه‌سازی مورد علاقه خودتان، اولین تست را بنویسید و اجرا کنید
  - Eclipse, IDEA, NetBeans , ?
- به برنامه‌هایی که نوشته‌اید مراجعه کنید و برای آن‌ها تست بنویسید
- نمونه‌تست‌هایی برای یکی از کلاس‌های مهم جاوا بنویسید. مثلاً:
  - HashMap, String, ArrayList, ...

افراد مهم در حوزه آزمون واحد و JUnit

- Kent Beck
  - از پیشروان Extreme Programming
  - از صاحب‌نظران موضوع Refactoring

- Erich Gamma
  - همچنین: یکی از چهار نویسنده کتاب مشهور الگوهای طراحی

Design Patterns: Elements of Reusable Object-Oriented Software (Gang of Four, GoF)

- اریک گاما و کنت بک همراه هم JUnit را ایجاد کردند

مطالعه بیشتر

- http://www.tutorialspoint.com/junit/junit_tutorial.pdf