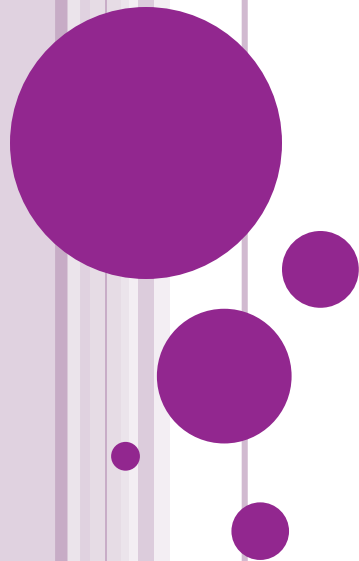


دوره کیفیت نرم افزار

آزمون واحد در جاوا UNIT TESTING IN JAVA

صادق علی اکبری



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 می کند
- آشپز، قبل از مهمان غذا را می چشد
- کنترل کیفیت و تضمین کیفیت در صنایع مختلف مورد توجه است
- سهل انگاری در تست محصول، گاهی ویرانگر است



سهل انگاری در ارزیابی کیفیت محصول

● سال ۱۹۹۸: برج «کاخ ۲» 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



سهل انگاری در ارزیابی نرم افزار

- موشک آریان ۵ 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 نرم افزار
- سال ۱۹۸۰: پنج بیمار بر اثر دریافت مقدار زیاد پرتوی X جان باختند
 - علت: اشکال در نرم افزار ماشین پرتودرمانی



نیاز به آزمایش نرم افزار

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



آزمایش نرم افزار

- ویژگی های یک نرم افزار خوب چیست؟

- عملکرد صحیح

- ویژگی های کیفی (غیر عملکردی)

- کارایی، سرعت، سهولت استفاده، امنیت و غیره

- تست نرم افزار : فرایندی برای آزمایش ویژگی های مورد نظر نرم افزار

- انواع مختلفی از تست نرم افزار وجود دارد

- انواع تست، ویژگی های مختلف نرم افزار را ارزیابی می کنند



ابعاد آزمون نرم افزار

- 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 نرم افزار (UNIT TESTING)

- انواع مختلفی از تست در طول عمر یک پروژه انجام می شوند
- برخی از این تست ها به دخالت مستمر کاربر، طراح یا مشتری نیاز دارد
- برخی از تست ها نیز در تیم کنترل کیفیت اجرا می شوند
- اما آزمون واحد توسط برنامه نویس و برای برنامه نویس انجام می شود
 - جزو وظایف برنامه نویس است
- البته آزمون واحد کافی نیست
- انواع دیگر آزمون برای تضمین کیفیت نرم افزار 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 می کند؟
 - به ازای ورودی های مختلف، خروجی /رفتار مناسب تولید می کند؟
 - هر بخش و جزء برنامه را جداگانه آزمایش می کنیم
 - 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 می کند (sort)
- یک متد main برای کلاس آن می نویسیم
- چند حالت از ورودی های مختلف را امتحان می کنیم
- خروجی ها را چاپ می کنیم (system.out.println)
- صحت خروجی ها را به صورت دستی (چشمی) بررسی می کنیم
- کدهای تست نوشته شده را حذف می کنیم و کدهای دیگری می نویسیم
- نکته:
- Business Code: متن اصلی برنامه ها
- Test Code: کدهایی که برای آزمایش برنامه ها 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]);
    }
}
```

```
Console ✖
<terminated> Sc
1
2
3
4
5
6
```

● 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 نرم افزار

- آزمون white box (نه black box)

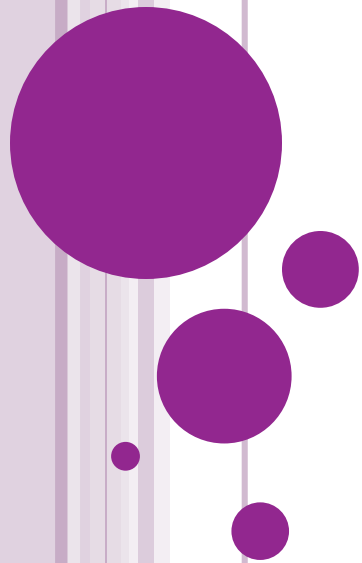
- آزمون واحد (نه آزمون سیستم، و نه حتی آزمون چند بخش یکپارچه شده)

- [معمولاً] آزمون عملکردی (نه آزمون ویژگی های کیفی)

- آزمون خودکار (نه دستی)

- تولید توسط خود برنامه نویس (نه توسط تیم تست یا کاربر)





آشنایی با JUNIT

یک نمونه آزمون واحد در JUNIT

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

Runs: 1/1

Errors: 0

Failures: 0

ir.javacup.junit.sorting.TestSorting [Runner: JUnit 4] (0.026 s)

```
@Test
```

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



اجرای آزمون واحد

- هر متد تست با حاشیه `@Test` مشخص می شود
- هر متد تست یک نمونه آزمون (`test-case`) خوانده می شود
- نمونه آزمون ها متد `main` ندارند و با کمک `TestRunner` اجرا می شوند
- غالباً از طریق محیط های برنامه نویسی (مثل `Eclipse`) یا `Maven` اجرا می شوند
- یادآوری: موفقیت آمیز بودن آزمون باید به صورت خودکار بررسی شود
- نه به صورت دستی: سندروم `println`
- نتیجه آزمون در `JUnit` با کمک `assertion` بررسی می شود
- مثال: `assertEquals`



نمونه آزمون

- هر نمونه آزمون شامل این بخش‌هاست:
 - یک ورودی برای متد مورد آزمون تعیین می‌شود (test data)
 - خروجی و رفتار مورد نظر برای این ورودی مشخص می‌شود (expected result)
 - متد مورد آزمون با این ورودی فراخوانی می‌شود (invocation)
 - خروجی و رفتار متد با خروجی مورد نظر 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

```
@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);
    }
}
```



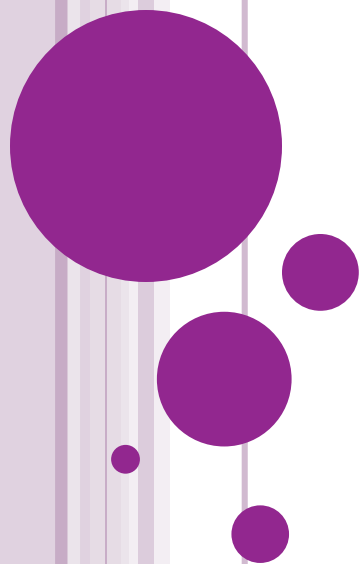
سایر امکانات JUnit

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



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

کوئیز

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

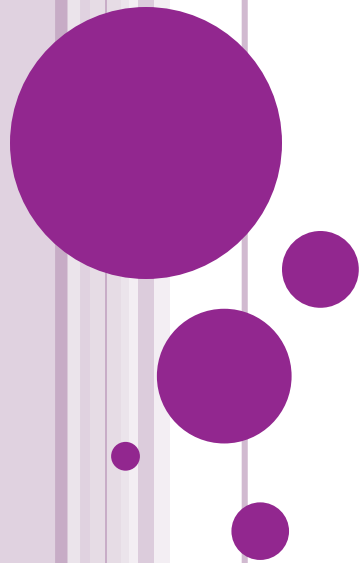
```
public class MyMathTest {
    @BeforeClass
    public static void startup(){System.out.println("Startup");}
    @AfterClass
    public static void shutdown(){System.out.println("Shutdown");}
    @Before
    public void setup(){System.out.println("Setup");}
    @After
    public void tearDown(){System.out.println("TearDown");}
    @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 می کند
- و نشان می دهد 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 می‌دانند
 - و حتی مدیران
- دلایلی که بر ضد آزمون می‌آورند:
 - وقت نداریم، پروژه از زمانبندی عقب است!
 - زمان زیادی برای نوشتن نمونه‌تست‌ها هدر می‌رود
 - تست، کار و وظیفه من نیست

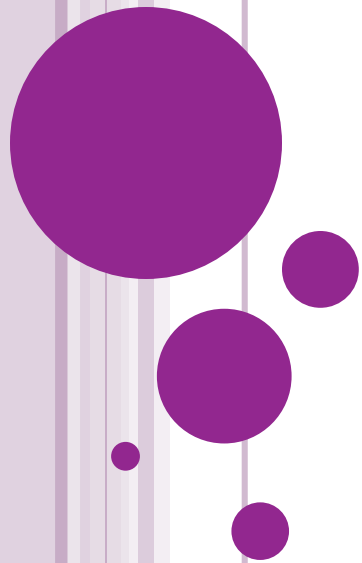


زمانی که صرف آزمون می‌شود، هدر نمی‌رود

- 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 نمونه آزمون‌ها (test-case) باید پاس شوند
- هر گاه یک نمونه آزمون fail شود:
 - فرآیند اضافه کردن یا گسترش دادن کد باید متوقف شود
 - تا زمانی که اشکال مورد نظر رفع شود
- هر نمونه آزمون 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 می شود

- پس نمونه آزمون ها کامل نیستند
- اگر کامل بودند، قبل از انتشار اشکال، نمونه آزمون متناظر fail می شد

● فرایند رفع اشکال

1. نمونه آزمونی نوشته شود که در شرایط گزارش شده fail می شود
2. برنامه اصلی اصلاح شود
3. آزمون های 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

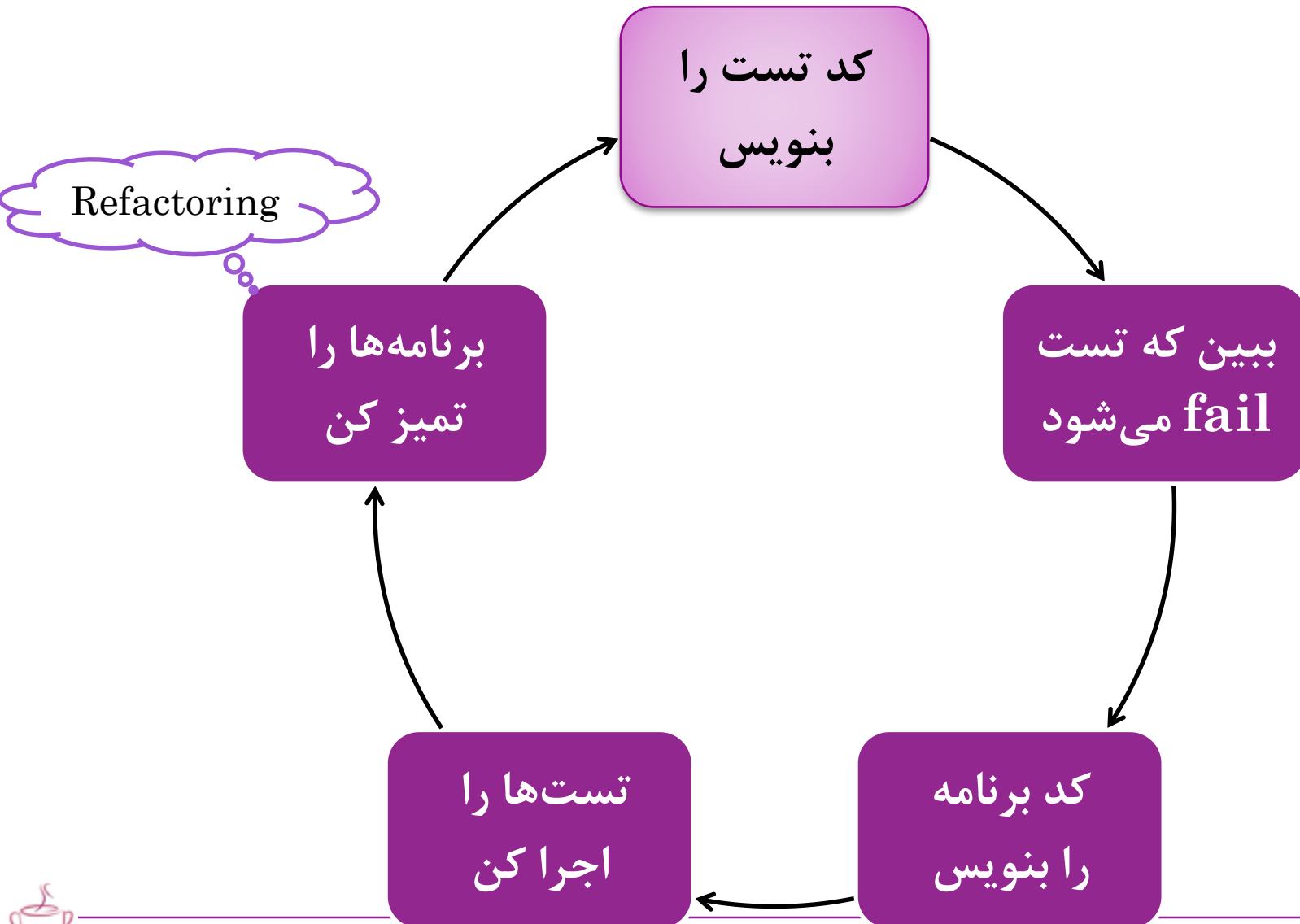
- Test-First Development

- در این روش، قبل از نوشتن متد به نحوه استفاده از آن فکر می‌کنیم

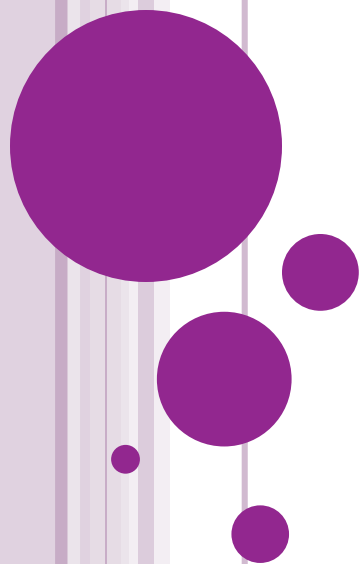
- البته بحث‌های فراوانی درباره مزایا و معایب این رویکرد وجود دارد



برنامه‌نویسی مبتنی بر تست (TDD)



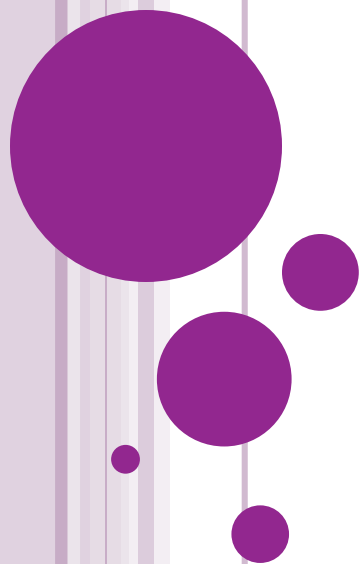
مطالب فنی و جزئیات راه اندازی



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
- NetBeans، IntelliJ IDEA و ...
- دقت به **import** ها و **import static** ها
- تغییر در **business code** منتج به **test case failure**
- مزیت وجود **test-case** ها





جمع بندی

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 SonarQube و Jenkins

TestNG



جمع بندی

- انواع آزمون نرم افزار
- آزمون واحد (توسط برنامه نویس، به صورت خودکار)
- امکانات JUnit : `@Test`، `@Before`، ...
- مزایای تست
- عدم انجام آزمون واحد، یک «وام فنی» (technical debt) است
 - این کار، صرفه جویی 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 برنامه نویسان جاوا در میان ابزارها، ابزار JUnit است
- آزمون واحد موضوعی تفنی و تزینی 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...



A decorative graphic on the left side of the slide, featuring a vertical stack of stripes in various shades of purple and white. To the right of these stripes are several circles of different sizes, also in shades of purple, arranged in a roughly vertical line.

پایان

The slide features a dark purple background. On the left side, there are several vertical stripes of varying shades of purple and white. Below these stripes, there are five circles of different sizes, also in shades of purple, arranged in a cluster. The text 'سایر مطالب' is written in white, bold, Persian script, positioned to the right of the circles.

سایر مطالب

آزمون نرم افزار در مقایسه با سایر صنایع

- نرم افزار، یک محصول نرم است
- امکان شبیه سازی شرایط سخت واقعی بدون صدمه دیدن اصل نرم افزار
- برج میلاد را در مقابل شبیه سازی زلزله ۸ ریشتری آزمایش نمی کنند
 - اگر دوام بیاورد، آسیب می بیند
- یک نرم افزار مبتنی بر وب را می توان در مقابل **شبیه سازی حجم شدید** درخواست کاربر آزمایش کرد
 - نرم افزار، آسیب نمی بیند



کیفیت نرم افزار

- تضمین کیفیت
- فرایندی که در چرخه تولید نرم افزار نقش ایفا می کند
- کنترل کیفیت
- یکی از مراحل پایانی که برخی جنبه های کیفیت را می سنجد



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



- 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 گزارش شده 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 همان قدر 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 ...



سوالات رایج

• زمان کافی برای نوشتن آزمون واحد نداریم

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

ماجرای میان مدت 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.



- مدیر پروژه: کلاسی که باید می‌نوشتی، کامل شد؟
- برنامه‌نویس: بله من کارم تمام شده.
- مدیر: آفرین! تست‌ها هم پاس می‌شوند؟
- برنامه‌نویس: هنوز تست ننوشته‌ام!



● 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 صاحب نظران 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



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

