



تولید نرم‌افزار مبتنی بر آزمایش

TEST-DRIVEN DEVELOPMENT

ارائه دهنده

صادق علی‌اکبری

همایش رقابت و فراگیری جاوا(JCAL)– آبان ۸۸

سرفصل مطالب

- رویکردهای تست نرم‌افزار
- معرفی Unit Test
- آشنایی با jUnit
- مزایای Unit Testing
- ویژگی‌های تست خوب
- پوشش تست
- تولید نرم‌افزار مبتنی بر تست
- dbUnit

تست نرم‌افزار

اهمیت و لزوم آزمایش

- 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.

آزمون نرم‌افزار

- انواع مختلفی از تست در طول عمر یک پروژه انجام می‌شوند.
- برخی از این تستها به دخالت مستمر کاربر نیاز دارد.
- برخی دیگر به تیم‌هایی مانند تیم تضمین کیفیت احتیاج دارند.
- در این ارائه ما فقط با **Unit Testing** سر و کار داریم.
- فرایندی برای برنامه‌نویس که تولید کد را بهتر و سریعتر می‌کند.
- **Unit Testing** ، توسط برنامه نویس، و برای برنامه نویس انجام می‌شود. نه برای کاربر، یا مدیر یا ...

مفهوم Unit Testing

- در ساختن یک ساختمان، هر بخش و هر جزء سازنده را جداگانه آزمایش کنیم و سپس در بدنه ساختمان استفاده کنیم.
- این آزمایش‌ها، قبل از آزمایش کل ساختمان لازم است.

روش معمول آزمایش

- فرض کنید یک کلاس نوشتیم که یک آرایه را مرتب می‌کند.
- معمولاً یک متد **main** برای آن می‌نویسیم و چند حالت از ورودی‌های مختلف را امتحان می‌کنیم.
- سندروم **system.out.println()**

روش سنتی آزمایش

```
public static void main(String[] args) {
    //testing sort() method:
    int[] list = {3,2,5,7,6,1,3};
    sort(list);
    for (int i=0;i<list.length;i++) {
        System.out.println(list[i]);
    }
}
```

Problems | Javadoc | Declaration | Search | Console

<terminated> Business [Java Application] D:\dev\jdk1.6.0_12\bin\javaw.exe (Oc

```
1
2
3
3
5
6
7
```

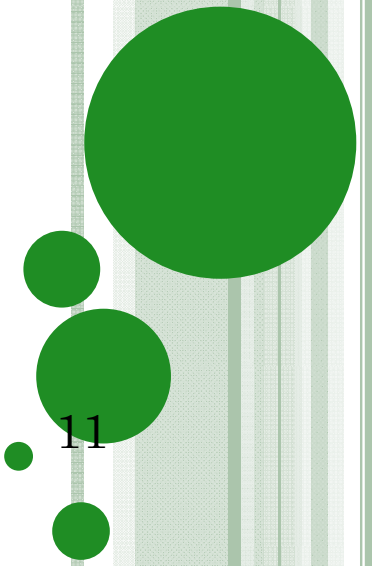
معایب روش سنتی

- تستهای نوشته شده دور ریخته می‌شود
- در هر لحظه یک تست انجام می‌شود
- برنامه‌نویس باید به صورت دستی تستها را اجرا کند
  - اجرای تستها اتوماتیک نیست
- برنامه‌نویس باید شخصاً از صحت آنها مطمئن شود
  - تشخیص موفقیت آمیز بودن تستها اتوماتیک نیست

ویژگی‌های Unit Test

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

JUNIT



تست در روش سنتی

```
public static void main(String[] args) {
    //testing sort() method:
    int[] list = {3,2,5,7,6,1,3};
    sort(list);
    for (int i=0;i<list.length;i++) {
        System.out.println(list[i]);
    }
}
```

Console

<terminated> Business [Java Application] D:\dev\jdk1.6.0_12\bin\javaw.exe (Oct 27

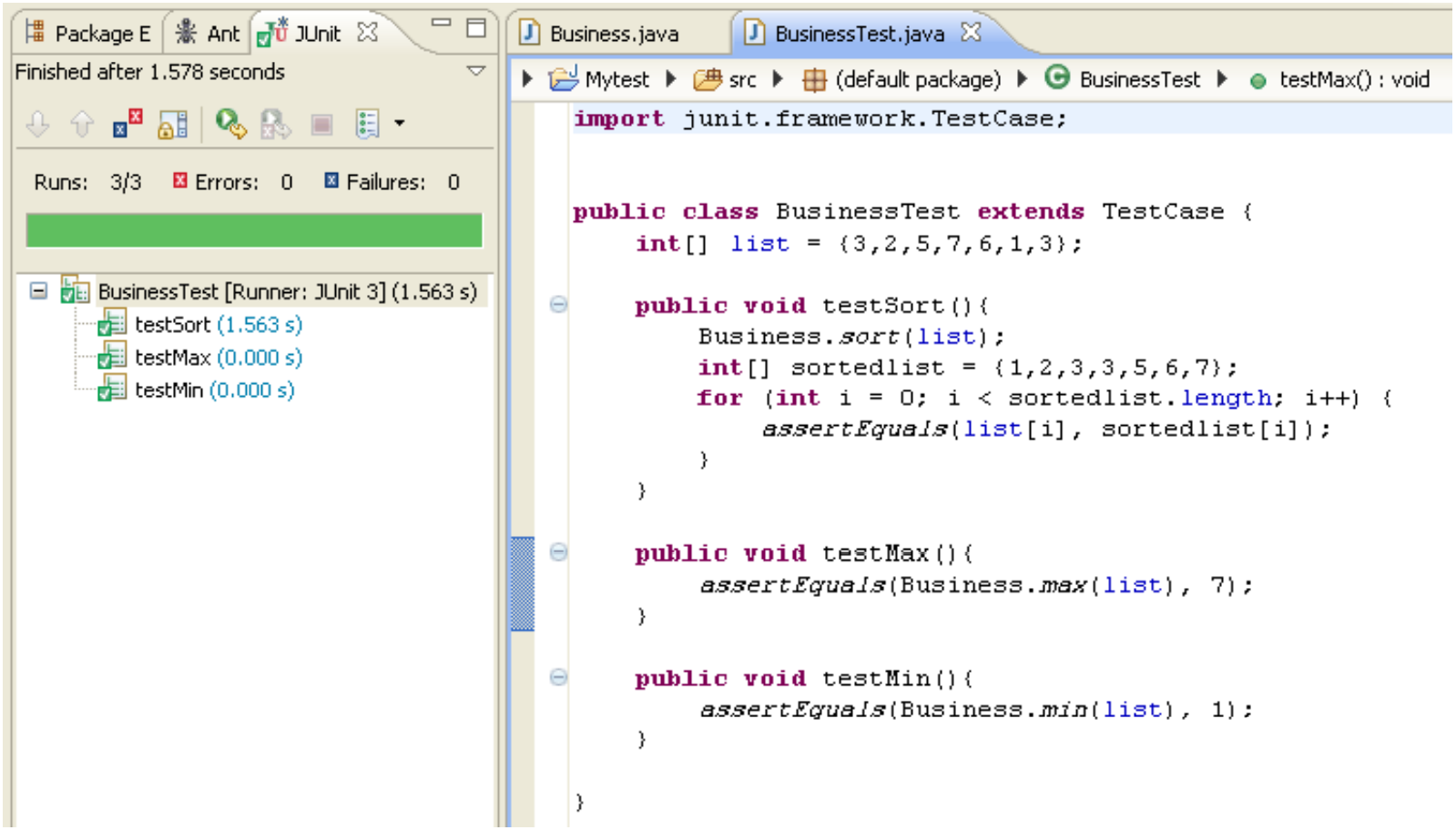
```
1
2
3
3
5
6
7
```

تست با کمک jUnit

```
public class BusinessTest extends TestCase {
    public void testSort(){
        int[] list = {3,2,5,7,6,1,3};
        Business.sort(list);

        int[] sortedlist = {1,2,3,3,5,6,7};
        for (int i = 0; i < sortedlist.length; i++) {
            assertEquals(list[i], sortedlist[i]);
        }
```

تست با کمک jUnit (ادامه)



اجرای تست

- گفتیم که Unit Test خود مسئول پی بردن به pass شدن یا fail شدن تست است.
- این کار در junit به کمک assertion ها انجام می‌شود.
- برای اجرای Test Unit ها، بسیاری از IDE ها امکاناتی دارند.
- اگر از یک IDE استفاده نمی‌کنید، باید از TestRunner استفاده کنید.
  - نام کلاس تست را به یک Test Runner پاس کنید
- اگر یکی از تست ها fail شود، پیغام مناسبی نشان داده می‌شود.

نوشتن تستها با jUnit

- برای هر متد، یک متد متناظر که test به ابتدای نام آن اضافه شده خواهیم داشت.
- در تستها از assertion استفاده می‌کنیم.
- در این جا به تعدادی از assertion ها که توسط Junit فراهم شده می‌پردازیم.

jUnit assertions

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

Unit Testing مزایای

چرا Unit Test ؟

- کسی که Unit Testing انجام می‌دهد، وقت کمتری را به خاطر debugging هدر می‌دهد.
- یک Unit Test ، یک قطعه برنامه است که یک قسمت از کد را برای داشتن یک کارایی خاص می‌آزماید.
  - مثال آرایه
- به کمک Unit Testing ، درجه اطمینان به توابع سطح پایین، که سنگ بنای سطوح بالاتر هستند، بالاتر می‌رود.

Unit Test و رفع اشکال

- Unit Testing نه تنها زمان debugging را کاهش می‌دهد، design را هم بهبود می‌بخشد.
- * آیا جملاتی مانند «این غیرممکنه!» یا «من نمی‌فهمم این چطور ممکنه اتفاق بیفته!» به خصوص موقع debugging برای شما تداعی می‌شود؟
- این به خاطر عدم اطمینان شما در مورد توابع سطح پایین است.

Unit Test و مستندسازی

- Test Unit ها کار documentation را نیز به نوعی انجام می‌دهند.
- در واقع یک Unit Test یک document قابل اجرا است.
  - نشان می‌دهد، در شرایط مختلف چه توقعی باید از تابع داشت.
- همچنین اعضای تیم با نگاه کردن به Unit Testها، نحوه استفاده از تابع را می‌فهمند.
- به خصوص یک مستند قابل اجرا، نسبت به یک مستند معمولی این برتری را دارد که «صحیح» است!
  - چون یک تابع ممکن است آنطور نباشد که document نوشته شده‌اش می‌گوید.
- نوشتن Unit Test ها قبل یا همزمان با نوشتن توابع انجام می‌شود، نه در اتمام کار پروژه.

Unit Test و اتوماتیک بودن تست

- در تمام طول عمر پروژه، تمام **Unit Test** ها باید پاس شوند.
- هر گاه یک **Unit Test**، **fail** شود، فرآیند اضافه کردن یا گسترش دادن کد متوقف می‌شود، تا زمانی که **bug** مورد نظر رفع شود.
- هر **Test Unit** خودش باید بفهمد **pass** شده یا **fail** شده.
  - نه این که مثلاً در خروجی چیزی بنویسد و کس دیگری مسئول پردازش این گزارش باشد.
- بدین ترتیب عملیات تست، اتوماتیک می‌شود.

Unit Test و اتوماتیک بودن تست

- خسارت غیرمستقیم یا Collateral Damage، یعنی bug یا اختلالی که در اثر رفع شدن یک bug در سیستم ظاهر شده است.
- اگر تست، به انتهای پیاده‌سازی پروژه موکول شود، این پدیده بسیار رخ می‌دهد.

دردسر تست

- بسیاری از برنامه‌نویسان، Unit Testing را یک دردسر می‌دانند.
- دلایلی که بر ضد Testing می‌آورند:
  - زمان زیادی برای نوشتن Test ها هدر می‌رود.
  - اجرای تست‌ها زمان زیادی می‌برد.
  - تست کردن کار من نیست.

زمان زیادی برای نوشتن Test ها هدر می‌رود

- این دلیل، اعتراض شماره یک در مقابل **Testing** است.
- اگر در هنگام پیاده‌سازی، تستهای لازم نوشته شود، وقت زیادی گرفته نخواهد شد.
- اگر هنوز فکر می‌کنی این زمان زیاد است به این سوالها فکر کن:
  - چقدر زمان به خاطر **debugging** صرف می‌کنی؟
  - چقدر وقت صرف تعمیر تابع‌هایی می‌کنی که فکر می‌کردی درست هستند، ولی **bug** های بزرگی دارند؟
  - چقدر زمان صرف رفع یک **bug** گزارش شده می‌کنی؟

دلایلی که بر ضد Testing می‌آورند (ادامه)

- اجرای تست‌ها زمان زیادی می‌برد.
  - معمولاً اینطور نیست.
  - اگر برخی Test ها زیاد طول می‌کشند، آنها را روزی یک بار، یا چند روزی یکبار اجرا کن.
- تست کردن کار من نیست.
  - با اینکه گروه‌هایی مانند QA برای تست وجود دارند
  - ولی ماهیت Unit Test با تستهای آنها متفاوت است
  - باید توسط خود برنامه‌نویس انجام شود.

باز هم JUnit

Test Suite

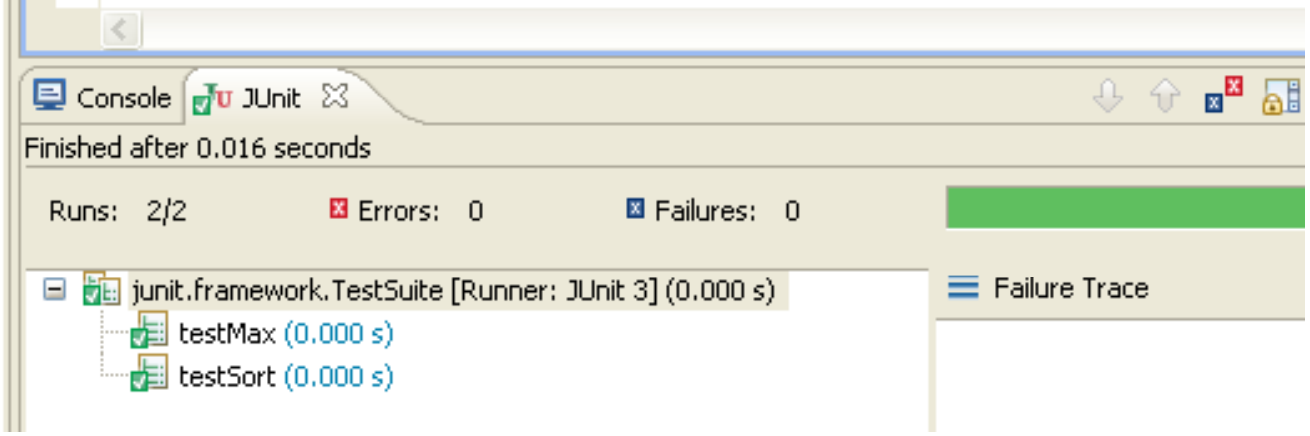
- هنگام اجرای یک Unit Test به کمک Test Runner، تمام متدهایی که با test شروع می‌شوند به کمک reflection، استخراج و اجرا می‌شوند.
- این فرایند پیش فرض را با کمک Suite ها می‌توان تغییر داد.
- اگر کلاس تست شامل متد Suite() باشد، فقط متدهای مشخص شده در این متد به عنوان تست اجرا می‌شوند. (نه همه متدهایی که با test شروع می‌شوند)
- نوشتن Constructor با پارامتر String برای نوشتن Suiteها کاربرد دارد.
- متد addTestSuite

TestSuite

```
    public BusinessTest(String methodName){
        super(methodName);
    }

    public static Test suite(){
        TestSuite s = new TestSuite();
        s.addTest(new BusinessTest("testMax"));
        s.addTest(new BusinessTest("testSort"));
        return s ;
    }

}
```



آماده سازی و پاک‌سازی

- متد setUp() قبل از همه متدهای test اجرا میشود
  - کارهایی که قبل از اجرای هر تست لازم است، در این متد قرار دهید
  - اتصال به پایگاه داده
  - مقداردهی به فیلدها
- متد tearDown() بعد از همه متدهای test اجرا میشود
  - کارهایی که قبل از اجرای هر تست لازم است، در این متد قرار دهید
  - قطع اتصال از پایگاه داده

آماده سازی و پاک‌سازی

```
public void testSort(){
    System.out.println("  testSort()");
    Business.sort(list);
    int[] sortedlist = {1,2,3,3,5,6,
    for (int i = 0; i < sortedlist.l
        assertEquals(list[i], sorted
    }
}

public void testMax(){
    System.out.println("  testMax()"
    assertEquals(Business.max(list),
}

public void testMin(){
    System.out.println("  testMin()"
    assertEquals(Business.min(list),
}

public void setUp(){
    System.out.println("setUp()");
}

public void tearDown(){
    System.out.println("tearDown()\n");
}
```

Console JUnit

<terminated> BusinessOldTest [JUnit] D:\dev\jdk

```
setUp()
  testSort()
tearDown()

setUp()
  testMax()
tearDown()

setUp()
  testMin()
tearDown()
```

jUnit 4

- از نسخه ۴ به بعد، با کمک امکانات Annotation، نوشتن unit test آسان‌تر شده است.
- متدهایی که دارای @Test هستند به عنوان تست اجرا می‌شوند.
- متدهایی که دارای @Before هستند، قبل از همه تست‌ها اجرا می‌شوند.
  - مانند متد SetUp()
- متدهایی که دارای @After هستند، بعد از همه تست‌ها اجرا می‌شوند.
  - مانند متد tearDown()
- @BeforeClass
- @AfterClass

نمونه تست با jUnit4

```
@Test
public void theTestMax(){
    System.out.println("  testMax()");
    assertEquals(Business.max(list), 7);
}

@Test
public void theTestMin(){
    System.out.println("  testMin()");
    assertEquals(Business.min(list), 1);
}

@Before
public void theSetUp(){
    System.out.println("setUp()");
}

@After
public void theTearDown(){
    System.out.println("tearDown()\n");
}
```

کیفیت تست

ویژگیهای تست های خوب

- اگر در نوشتن تستها دقت نشود، وقت زیادی برای نگهداری و debug کردن خود تستها هدر می‌رود
  - به این ترتیب به پروژه لطمه می‌خورد.
- تستهای خوب ویژگی‌های زیر را دارند:
- خودکار (Automation)
- کامل (Through)
- قابل تکرار (Repeatable)
- مستقل (Independence)
- حرفه‌ای (Professional)

خودکار (Automation)

- اتوماتیک بودن حداقل از دو جنبه باید موجود باشد:
  - اجرای تستها
  - چک کردن نتایج
- برخی IDE ها حتی این امکان را دارند که تستها را به صورت متناوب در background اجرا کنند.

کامل (Through)

- Unit Test های خوب باید کامل باشند.
- هر آن چه ممکن است اشتباه باشد، را تست کنند.
- سیاست‌های مختلف:
  - همه branch های ممکن (همه if ها) را تست کنیم و هر exception ای که باید throw شود را در تست ها هم بیاوریم.
  - و یا فقط موارد اصلی و خروجی‌ها را تست کنیم و از جزئیات صرف نظر کنیم.
- انتخاب بین یکی از این دو در واقع به پروژه بستگی دارد.
- در یک پروژه ارزان با زمان محدود، روش دوم معقول تر است.

پوشش تست

- در این زمینه می‌توان از ابزارهایی که در مورد **Test Coverage** هستند کمک گرفت.
- این ابزارها نشان می‌دهند چه نسبتی از کد تحت تست قرار گرفته است.
- این ابزارها می‌توانند میزان **branch** ها و **exception** هایی هم که در تستها مد نظر قرار گرفته‌اند را نیز محاسبه کنند.
- هر چه **Test Coverage** بیشتر باشد، میزان **Reported Bugs** کمتر خواهد بود.
- کدهایی که **report bug** های زیادی دارند محکوم به دوباره‌نویسی هستند.

قابل تکرار (Repeatable)

- تست‌ها قرار است به دفعات تکرار شوند
- بنابراین نباید وابسته به هیچ چیز در خارج از خودشان باشند
- تستها به هر ترتیبی ممکن است اجرا شوند
- در هر ترتیب باید همان خروجی را تولید کنند.

استقلال (Independent)

- هر Unit Test ، چیزی مستقل از Unit Test دیگر را تست کند.
  - با fail شدن تست، دقیقاً می‌فهمیم کدام قسمت کد مشکل دارد.
- این خاصیت به Repeatable بودن تستها کمک می‌کند.
- باید بتوانیم آنها را در هر زمان و با هر ترتیبی اجرا کنیم.
- مکانیزمهایی مانند setUp، و tearDown در این زمینه کمک می‌کنند.

حرفه‌ای (professional)

- Unit Test ها باید همانقدر که کد اصلی حرفه‌ای نوشته می‌شوند، حرفه‌ای و با دقت نوشته شوند
- یک کار سرسری انگاشته نشوند.
- Unit Test ها تمام ویژگیهای یک طراحی خوب را باید داشته باشند.
  - Encapsulation
  - Low coupling
  - High cohesion
- Test code is real code!

تست حرفه‌ای

- بنابراین هر جا لازم است کلاس جدید بسازید تا طراحی تست‌هایمان بهتر باشد.
- صرفاً برای بالا بردن پوشش تست، تست ننویسید.
- توقع داشته باشید که حداقل به اندازه کد برنامه اصلی(code production)، برنامه تست داشته باشید!
- مثلاً اگر 20000 خط برنامه نوشته‌اید، معقول است که حداقل 20000 خط هم Test code داشته باشید.

xUnit

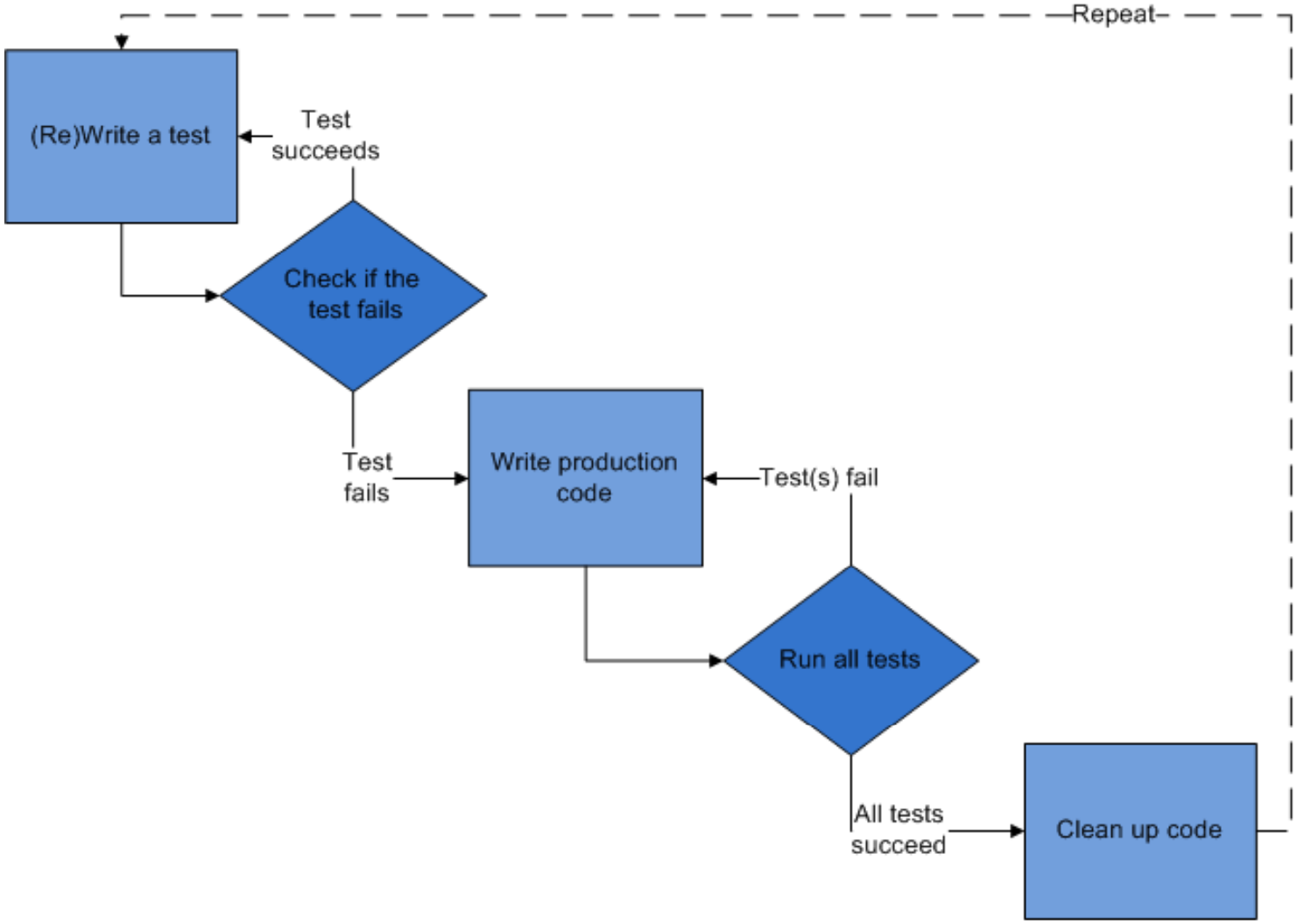
- بسترهای تست دیگری مانند **jUnit** ، برای بیش از ۶۰ زبان برنامه‌نویسی وجود دارد، که به صورت رایگان قابل استفاده هستند:

http://xprogramming.com/software.htm

Test-Driven Development

- در این رویکرد، تست‌ها، قبل از نوشتن کد نوشته می‌شوند.
- **Test-First Development**
- قبل از نوشتن متد، به نحوه استفاده از آن فکر می‌کنیم.
- دست‌پخت خودت را قبل از دیگران بخور!

چرخه تولید برنامه در TDD



DBUnit

- dbUnit یک الحاقیه به jUnit است
- گاهی یک تست، طوری محتوای دیتابیس را تغییر می‌دهد که تست‌های بعدی fail می‌شوند.
- گاهی برخی تستها پیش‌شرط‌های خاصی درباره محتوای دیتابیس دارند.
- dbUnit کمک می‌کند بین اجرای تست‌های مختلف، دیتابیس در وضعیت موردنظر بماند.
- امکانات import و export به فایل xml را دارد
- برای مقایسه محتوای دیتابیس با مقادیر موردنظر کاربرد دارد

اجزای اصلی dbUnit

- کلاس IDatabaseConnection
  - کلاس connection به دیتابیس
- IDataSet
  - شیءی شامل محتوای تعدادی جدول از دیتابیس
- DatabaseOperation
  - نشان‌دهنده یک عمل بر روی دیتابیس

عملهای ممکن در DatabaseOperation

- UPDATE
- INSERT
- REFRESH
- DELETE
- CLEAN_INSERT
- DELETE_ALL
- NONE

```
protected IDatabaseConnection getConnection() throws Exception
{
    Class driverClass = Class.forName("com.mysql.jdbc.Driver");
    Connection jdbcConnection = DriverManager.
        getConnection("jdbc:mysql://leahpar/test", "phil", "phil");
    return new DatabaseConnection(jdbcConnection);
}

protected IDataSet getDataSet() throws Exception
{
    loadedDataSet = new FlatXmlDataSet(this.getClass().getClassLoader().
                getResourceAsStream("input.xml"));
    return loadedDataSet;
}

public void testCheckDataLoaded() throws Exception
{
    assertNotNull(loadedDataSet);
    int rowCount = loadedDataSet.getTable(TABLE_NAME).getRowCount();
    assertEquals(2, rowCount);
}
```

```java
public void testCompareDataSet() throws Exception
{
    IDataSet createdDataSet = getConnection().createDataSet(new String[]
    {
        TABLE_NAME
    });
    Assertion.assertEquals(loadedDataSet, createdDataSet);
}

public void testCompareQuery() throws Exception
{
    QueryDataSet queryDataSet = new QueryDataSet(getConnection());
    queryDataSet.addTable(TABLE_NAME, "SELECT * FROM " + TABLE_NAME);
    Assertion.assertEquals(loadedDataSet, queryDataSet);
}
```

```
public void testExportData() throws Exception
{
    IDataSet dataSet = getConnection().createDataSet(new String[]
    {
        TABLE_NAME
    });

    URL url = DatabaseTestCase.class.getResource("/input.xml");
    assertNotNull(url);
    File inputFile = new File(url.getPath());
    File outputFile = new File(inputFile.getParent(), "output.xml");
    FlatXmlDataSet.write(dataSet, new FileOutputStream(outputFile));

    assertEquals(FileUtils.readFileToString(inputFile, "UTF8"),
            FileUtils.readFileToString(outputFile, "UTF8"));

}
```

```
public void setUp() throws Exception {
    super.setUp();

    IDatabaseConnection connection = null;
    try {
        connection = getConnection();
        DatabaseOperation.REFRESH.execute(connection, loadedDataSet);
    } finally {
        connection.close();
    }
}
```

مطالب باقیمانده

- پوشش تست (Test Coverage)
- اشیاء بدلی (Mock Objects)
- تأثیر تست بر فرایند تولید نرم‌افزار
  - فرکانس اجرای تست
  - فرایند رفع باگ
  - تست و مخزن کد
  - تست و مرور کد
- آزمایش اشیاء به جای متدها

مراجع

- Test-Driven Development By Example. (Kent Beck)
- Pragmatic Unit Testing (Andy Hunt, Dave Thomas)
- Extreme Programming Explained: Embrace Change (Kent Beck)
- http://www.extremeprogramming.org/

پایان