JUnit Fundamentals

On Testing

Introducing JUnit

Why JUnit?

Test-Driven Development

On Testing

- Why focus on testing?
  - **Testing is a critical activity for ensuring that a software system meets its functional and nonfunctional requirements**
  - Testing activities are a part of a sound software development lifecycle where testing is done at different levels of granularity and complexity
  - We cannot make reasonable claims about quality of a particular software product or a product line without systematic testing results
- Testing methods may span different dimensions
  - **Granularity**, including system, subsystem, component, unit
  - **Coverage**, such as line coverage, module coverage, feature coverage
  - **Component Dependence**, including black box testing, white box testing, black/white box testing
  - **Automation Level**, such as manual, semi automated, **automated**
  - **Validation Method**, including no validation, using assertion, formal methods
  - **Metric Dependence**, including product metrics (e.g., robustness, security), business metrics (e.g., budget and time constraints)

Test, Test, and Test Some More

- How frequently should we test our code?
  - We should test our code whenever we have time available for it
  - Unfortunately, **testing activities require additional time and result in additional development costs**
  - They also detract from business metrics of productivity (e.g., number of features implemented, amount of chargeable hours invested)
  - We code, compile, and then run thereby testing the code written
  - We test individual lines of code or we test particular behavior (e.g., selecting a particular command or entering specific input)
- Most developers have their own “testing patterns” such as running a small **testing suite** of **test cases** after each significant change
  - The testing activities can be viewed as a break from analytical and problem solving programming tasks
  - As a result, testing is rarely given equal importance as programming, and exhaustive testing is seen as a secondary development activity
  - However, since **test cases reflect software requirements** they must be updated and expanded along with corresponding code changes

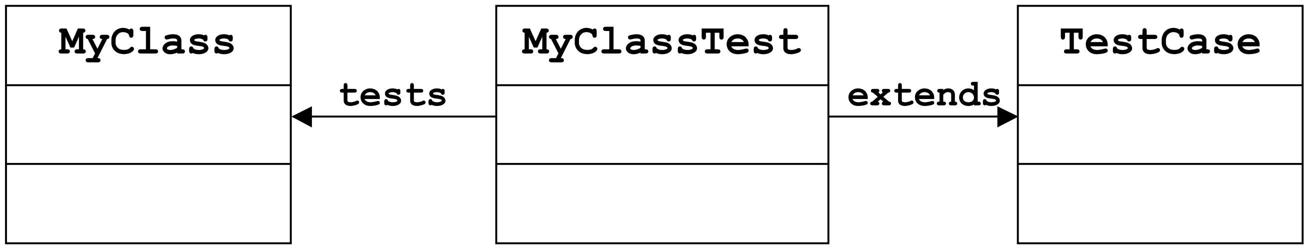
Introducing JUnit

- **JUnit is a testing framework** intended to promote structured approach to automated unit testing of software written in Java
  - It represents a reusable pattern (structure) for testing Java code
  - It was developed by Erich Gamma (co-author of “Design Patterns”) and Kent Back (known for work on Extreme Programming) in 1997
    - It is an instance of *xUnit testing architecture* that offers instances for testing other languages such as C/C++, Python, and Perl
  - JUnit is now an open-source project hosted by Source Forge
    - It is distributed under the Common Public License thereby making it easier to be incorporated into commercial tools
  - In the context of JUnit, a **“unit test”** refers to testing “unit of work” such as a method, a group of related methods, or even a class
    - The key distinction is **isolation** of the unit from other units
    - JUnit is not intended for testing unit interaction such as integration testing of components

JUnit 101

- **JUnit** is a combination of two concepts
  - **Design patterns** – the framework in its structure represents an instance of a Command design pattern
  - **Assertions** – the framework makes use of assertions to generate output of individual tests

- The JUnit is meant to follow the structure of the code
  - For each Java class, typically there is a JUnit Test class
  - Each JUnit Test class extends the TestCase base class, which provides the JUnit framework interfaces
  - Within the JUnit test class, individual **`test[UnitTestName]`** methods are defined to test specific units, such as class methods
  - With each **`test[UnitTestName]`** method, the **`assert[Type]`** assertions are used to check desired testing conditions



Testing with JUnit – Advantages /1

- JUnit tests are easy to write since they are written in Java, and they are more reflective of the software requirements
  - For each test, typically we need to decide how to define the test, what test variables to instantiate, and how to analyze the output
  - With JUnit, we need to focus on the test itself, as the framework resolves questions about the test case instantiation and output (e.g., `setUp` method for setting up shared test variables)
- Unit test cases costs less since much of the creation and maintenance overhead is eliminated through the framework
  - The need for defining a new test suite structure for every module is eliminated
  - JUnit structure does not change but additional suites can be added or existing ones extended
  - Overhead of deciding how to create individual test cases is reduced through a familiar test-suite assertions interface
  - Individual test cases run independently of each other so the need for testing coordination is decreased

Testing with JUnit – Advantages /2

- JUnit tests internally check results and instantly provide feedback
  - With the use of assertions, the need for an output language and its analysis is eliminated
  - JUnit provides a summary of passed and failed tests, with corresponding details for each one of the tests that failed
- The use of JUnit tests leads to increase in software quality
  - By promoting simplicity of writing test cases and decreasing cost of creating, running, and maintaining test suites, the appeal of testing to developers should increase
  - Given that the JUnit suites can be rerun inexpensively after each change or a set of changes, disruption to the expected software operation can be detected earlier
  - Test-driven development promotes focus on satisfying software requirements, and in turn leads to higher quality software
- JUnit provides a method for creation of hierarchical test suites
  - Individual test classes can be composed into test suits
  - Test suits can be composed into other test suits to create a hierarchy of testing modules that reflects the code structure

Testing with JUnit – Disadvantages

- JUnit **does not guarantee** better quality software
  - It is a framework that promotes the use of testing in software development
  - Software systems are inherently complex and their inherent complexity is present even at the lowest levels of abstraction
  - Creation of test cases and testing suites in highly complex or critical domains may require a more dedicated approach to testing such as the use of formal methods or simulations
  - Moreover, using JUnit by itself will not improve quality unless it is a part of a structured system testing approach
- JUnit does not invalidate previous testing methods
  - It provides a framework that simplifies test case definition and use, but the analysis algorithms such as boundary-case analysis still apply

Using JUnit within Eclipse

- Eclipse v3.1 should already come with the JUnit v3.8.1 as a plug-in
- Otherwise, download JUnit v4.1, junit4.1.zip, from
  - **`http://prdownloads.sourceforge.net/junit/junit4.1.zip`**
- Create a new Java project in Eclipse
  - As a project name, enter “JUnitModule”
  - Under Contents, select “Create project from existing source”, click Browse, and place the project on your Desktop under a new folder “JUnitModule”
  - Select “Next” and on the next screen select “Allow output folders for source folders”, and then click “Finish”
- Extract junit4.1.zip into the project folder, “JUnitModule”
- Right-click on the project title in Eclipse, and select Properties
  - Select “Java Build Path” and then select Source
  - Select “Add Folder”, enter “src”, and select “Yes” to adjust the build path
  - Select “Libraries” and then select “Add External JARs”
  - If preinstalled, browse to the JUnit folder within Eclipse (“C:\Program Files\Eclipse\plugins\org.junit._3.8.1”) and select “junit.jar”
  - Otherwise, browse to the project folder and select “junit-4.1.jar” file
  - Finally return to the Java perspective by clicking “OK”
- Once JUnit is setup, create a new class under the “(default package)” of your current project called “Account”

Account.java

```
public class Account {
    private int balance;

    public Account (int balance) {
        this.balance = balance;
    }

    public int increaseBalance (int increase) {
        balance += increase;
        return balance;
    }

    public int decreaseBalance (int decrease) {
        balance -= decrease;
        return balance;
    }

    public int getBalance () {
        return balance;
    }
}
```

AccountTest.java

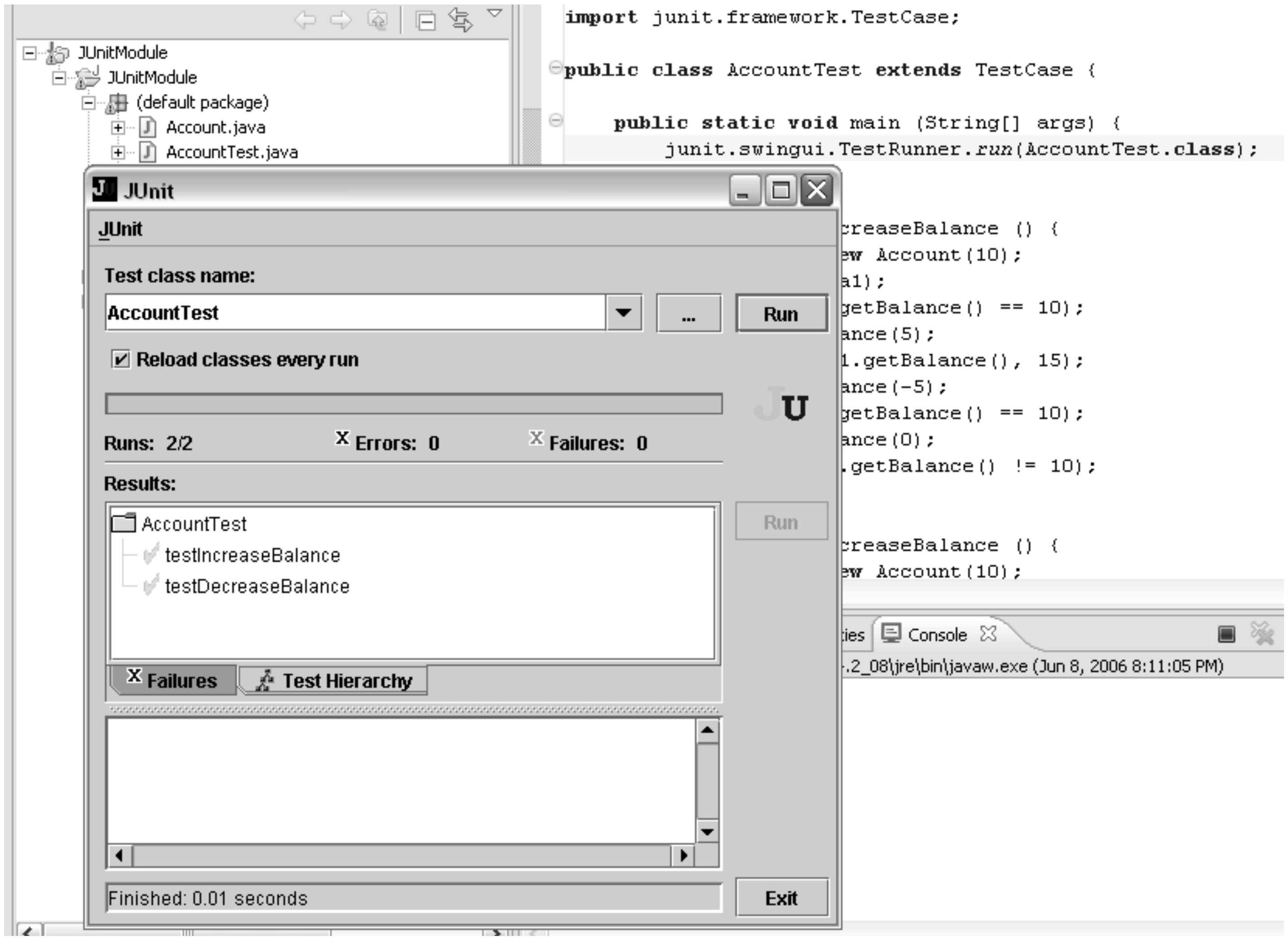
```
import junit.framework.TestCase;

public class AccountTest extends TestCase {

    public static void main (String[] args) {
        junit.swingui.TestRunner.run(AccountTest.class);
    }

    public void testIncreaseBalance () {
        Account a1 = new Account(10);
        assertNotNull(a1);
        assertTrue(a1.getBalance() == 10);
        a1.increaseBalance(5);
        assertEquals(a1.getBalance(), 15);
        a1.increaseBalance(-5);
        assertTrue(a1.getBalance() == 10);
        a1.increaseBalance(0);
        assertFalse(a1.getBalance() != 10);
    }
}
```

`AccountTest` Output with No Failures



```
import junit.framework.TestCase;

public class AccountTest extends TestCase {

    public static void main (String[] args) {
        junit.swingui.TestRunner.run(AccountTest.class);
```

```
creaseBalance () {
ew Account(10);
a1);
getBalance() == 10);
ance(5);
1.getBalance(), 15);
ance(-5);
getBalance() == 10);
ance(0);
.getBalance() != 10);

creaseBalance () {
ew Account(10);
```

Test class name: AccountTest

Reload classes every run [✓]

Runs: 2/2   Errors: 0   Failures: 0

Results:

- AccountTest
  - testIncreaseBalance
  - testDecreaseBalance

Finished: 0.01 seconds

.2_08\jre\bin\javaw.exe (Jun 8, 2006 8:11:05 PM)

AccountTest.java

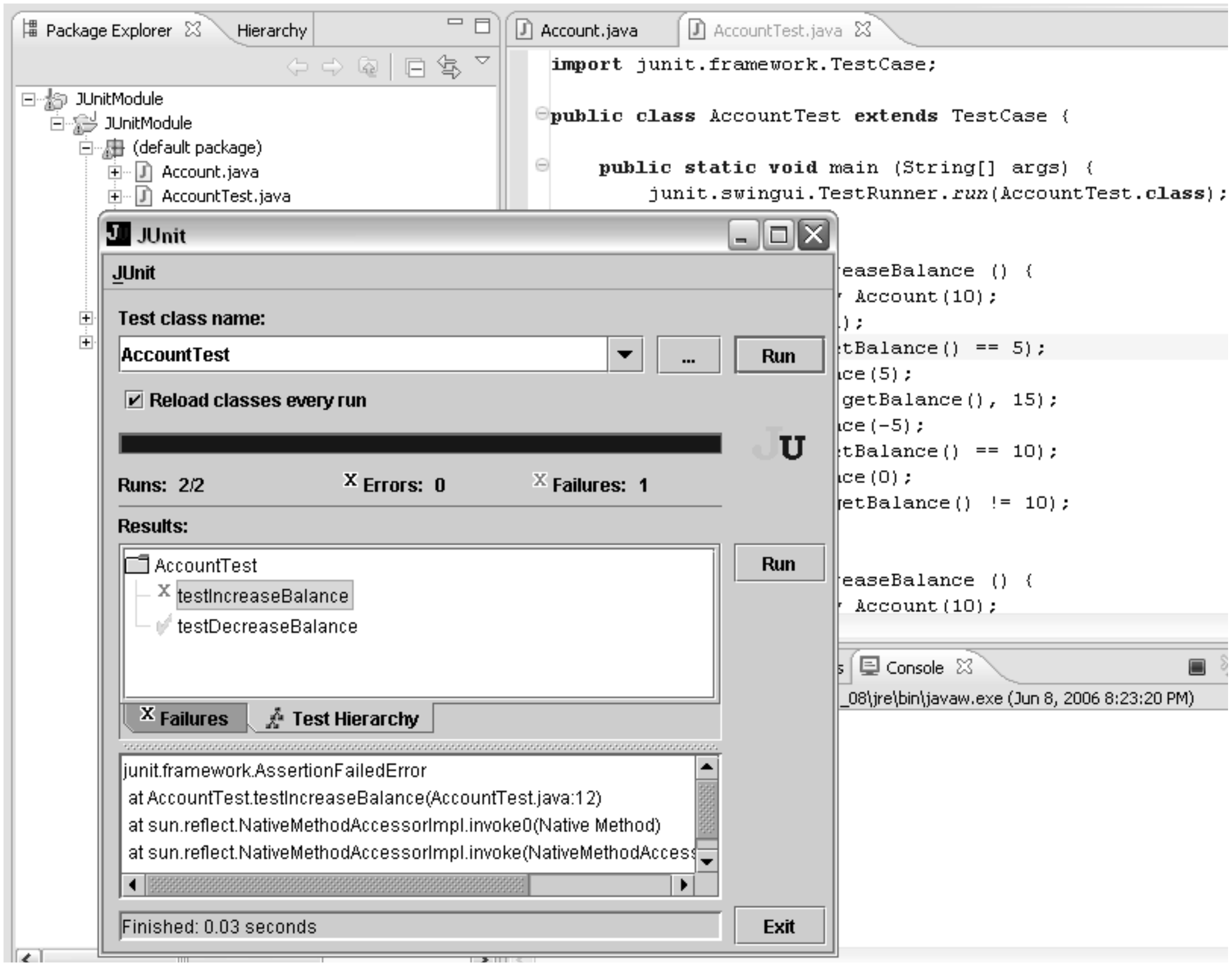
```
import junit.framework.TestCase;

public class AccountTest extends TestCase {

    public static void main (String[] args) {
        junit.swingui.TestRunner.run(AccountTest.class);
    }

    public void testIncreaseBalance () {
        Account a1 = new Account(10);
        assertNotNull(a1);
        assertTrue(a1.getBalance() == 0);
        a1.increaseBalance(5);
        assertEquals(a1.getBalance(), 15);
        a1.increaseBalance(-5);
        assertTrue(a1.getBalance() == 10);
        a1.increaseBalance(0);
        assertFalse(a1.getBalance() != 10);
    }
}
```

`AccountTest` Output with One Failure



```
import junit.framework.TestCase;

public class AccountTest extends TestCase {

    public static void main (String[] args) {
        junit.swingui.TestRunner.run(AccountTest.class);
```

```
easeBalance () {
 Account(10);
);
tBalance() == 5);
ce(5);
getBalance(), 15);
ce(-5);
tBalance() == 10);
ce(0);
etBalance() != 10);

easeBalance () {
 Account(10);
```

Test class name: AccountTest

✓ Reload classes every run

Runs: 2/2   Errors: 0   Failures: 1

Results:

- AccountTest
  - testIncreaseBalance
  - testDecreaseBalance

```
junit.framework.AssertionFailedError
 at AccountTest.testIncreaseBalance(AccountTest.java:12)
 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
 at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAcces
```

Finished: 0.03 seconds

_08\jre\bin\javaw.exe (Jun 8, 2006 8:23:20 PM)

AccountTest.java

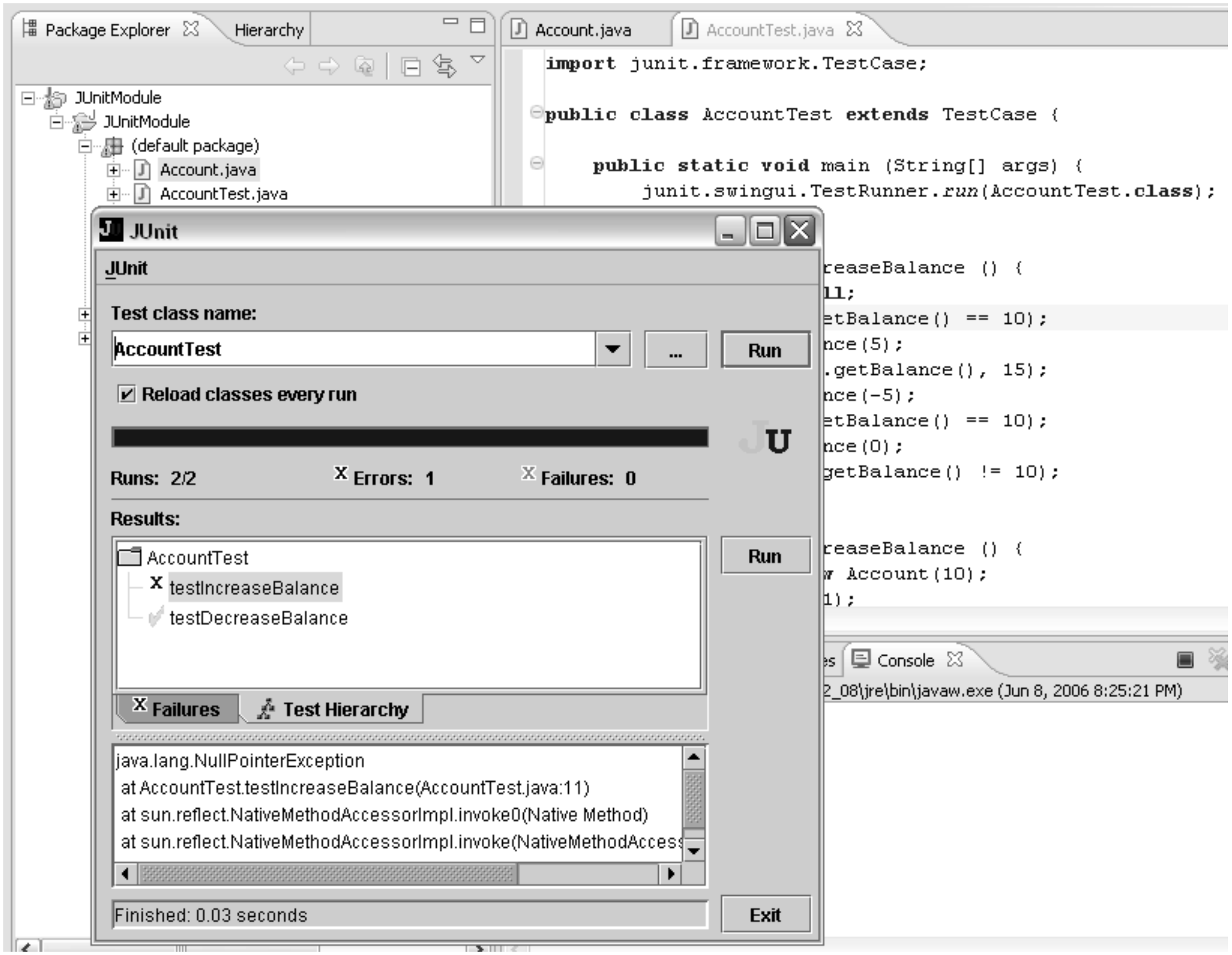
```
import junit.framework.TestCase;

public class AccountTest extends TestCase {

    public static void main (String[] args) {
        junit.swingui.TestRunner.run(AccountTest.class);
    }

    public void testIncreaseBalance () {
        Account a1 = null;
        assertTrue(a1.getBalance() == 0);
        a1.increaseBalance(5);
        assertEquals(a1.getBalance(), 15);
        a1.increaseBalance(-5);
        assertTrue(a1.getBalance() == 10);
        a1.increaseBalance(0);
        assertFalse(a1.getBalance() != 10);
    }
}
```

`AccountTest` Output with One Error



```
import junit.framework.TestCase;

public class AccountTest extends TestCase {

    public static void main (String[] args) {
        junit.swingui.TestRunner.run(AccountTest.class);
```

```
reaseBalance () {
ll;
etBalance() == 10);
nce(5);
.getBalance(), 15);
nce(-5);
etBalance() == 10);
nce(0);
getBalance() != 10);

reaseBalance () {
w Account(10);
1);
```

Test class name: AccountTest

☑ Reload classes every run

Runs: 2/2   Errors: 1   Failures: 0

Results:

- AccountTest
  - testIncreaseBalance
  - testDecreaseBalance

```
java.lang.NullPointerException
 at AccountTest.testIncreaseBalance(AccountTest.java:11)
 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
 at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAcces
```

Finished: 0.03 seconds

2_08\jre\bin\javaw.exe (Jun 8, 2006 8:25:21 PM)

Assertions in `TestCase`

- `assertTrue(boolean condition)`
- `assertFalse(boolean condition)`
- `assertNull(Object o)`
- `assertNotNull(Object o)`
- `assertEquals(`*`Type`*` expected, `*`Type`*` actual)`
- `assertSame(`*`Type`*` expected, `*`Type`*` actual)`
- `assertNotSame(`*`Type`*` expected, `*`Type`*` actual)`
- `fail()`
- JUnit also provides overloaded versions of these methods that take an addition `String message` as their first parameter
- Declared in `junit.framework.Assert`

Composing Test Cases

- Test cases generally use test variables which are initialized with specific values
  - To avoid redundancy of redefining these methods every time each test is run, a method **`setUp()`** can be used to initialize common variables
  - Additional method **`tearDown()`** can be used to de-initialize variables that need explicit cleanup (e.g. database connections, file handles, etc.)
- Test cases (individual **`testMethodName`** methods) of a class that extends **`TestCase`** JUnit class are all run by default **`TestSuite`** using the `suite()` method
- **`TestSuite`** provides two methods for instantiation
  - Using its constructor

    ```
    public static Test suite() {
      return new TestSuite(AccountTest.class);
    }
    ```
  - Using `addTestSuite` method

    ```
    public static Test suite() {
      TestSuite suite = new TestSuite();
      suite.addTestSuite(AccountTest.class);
      return suite;
    }
    ```

Combining Test Suites

```
import junit.framework.Test;
import junit.framework.TestSuite;

public class AccountTestSuite {

    public static Test suite () {
        TestSuite suite = new TestSuite();
        suite.addTestSuite(AccountTest.class);
        return suite;
    }

    public static void main (String[] args) {
        junit.textui.TestRunner.run(suite());
    }
}
```

- Additional test suites or a combination of test suites can be added using the **`suite.addTestSuite()`** method

Showing Test Results

- JUnit provides several different methods for running and output of test results through implementations of the `TestRunner` interface
  - **`junit.swingui.TestRunner.run(TestingClass.class);`** makes use of the Swing GUI framework to output results
  - Other options include **`junit.awtgui`** for the AWT GUI framework and **`junit.textui`** for standard output to console
- Note that each unit test with JUnit is separately loaded to avoid possible conflicts in different test scenarios
  - Each successful test is counted under Pass count, each failed test is counted under Failed count, white each test that caused an exception is counted under Error count
  - **`BaseTestRunner`** is a super-class for all **`TestRunner`** instances and hence can be extended (if needed) to provide further customization

Test-Driven Development

- JUnit promotes testing as a development activity but test cases can be developed before coding
  - In extreme programming approaches test cases are first-level development artifacts
  - The test suites are created based on the requirements and in most cases are created before any code is written
  - In an iterative and incremental fashion, for each test case a corresponding code module is implemented and refactored until the matching test cases are completely satisfied
  - With new requirements, test cases are changed to reflect the requirements and then the code is changed to reflect the test cases
- In the following exercises, we will make use of test-driven development to satisfy given requirements

Exercise – Testing Simple Data Structures

- Description:
  - Create an empty `DVD` class with the following attributes and methods (do not implement the body of these methods just yet)
    - **`private int discs, year;`**
    - **`private String title, genre;`**
    - **`public boolean equals(Object)`**
    - **`public int hashCode()`**
    - ***`getters and setters`***
  - Implement a `DVDTest` JUnit class to test the `DVD` class methods
    - Use the `setUp` method to make the testing *easier*
    - Also test that all of the attributes within the `DVD` class are encapsulated (visibility set to `private`)
  - Once you have finished your `DVDTest` class, go back and implement the body of the methods in the `DVD` class
    - Ensure that the `DVD` methods pass all the tests in `DVDTest`
  - Learning objectives
    - Practice test-driven development of simple data structures
    - Share your discoveries with your instructor and class colleagues

Exercise – Testing Complex Data Structures /1

- Create an empty `DVDCollection` class with the following attributes and methods (do not implement the body of these methods just yet)
  - **`private String name;`**
  - **`private Collection<DVD> dvds;`**
  - **`private Collection<DVDCollection> subcollections;`**
  - **`public String getName()`**
  - **`public void addDVD(DVD dvd)`**
  - **`public void removeDVD(DVD dvd)`**
  - **`public DVD getDVD(String title, int year, boolean searchNested)`**
  - **`public Collection<DVD> getDVDs()`**
  - **`public addDVDCollection(DVDCollection sub)`**
  - **`public DVDCollection getCollection(String name)`**
  - **`public Collection<DVDCollection> getCollections()`**
  - **`public boolean equals(Object o)`**
  - **`public int hashCode()`**

Exercise – Testing Complex Data Structures /2

- Description:
  - Implement a `DVDCollectionTest` JUnit class to test the `DVDCollection` class
    - Add several stress tests to handle large collections with several sub-collections and a large number of DVDs
  - Once you have finished your `DVDCollectionTest` class, go back and implement the body of the methods in the `DVDCollection` class
    - Ensure that the `DVDCollection` methods satisfy tests in `DVDCollectionTest`
  - Learning objectives
    - Practice test-driven development of complex data structures
    - Practice use of JUnit testing for improving test and code quality
    - Share your discoveries with your instructor and class colleagues

Exercise – Testing XML Parsing with Java

- Description:
  - Recall the XSLT Java Exercise (Exercise 6), where we have used Java to perform the transformation from one schema to another
  - In this exercise, we would like to create a corresponding JUnit test suite to validate the implemented transformation scenario
  - For instance, we would like to test the following
    - Only schema-compliant (valid) XML files are parsed
    - Source file is correctly parsed into a source DOM tree
    - Target DOM tree is created
    - For each node type, correct target node and value are generated
    - For each attribute, correct target attribute and value are generated
    - Target DOM tree is correctly output into an external file
  - Additional testing goals, as time permits
    - Ensure well-formedness of source and target XML files
    - Add several stress tests to handle large files with many nodes
  - Learning objectives
    - Practice usage of JUnit to create a practical test suite in Java
    - Recognize benefits of JUnit-based testing for improving software quality
    - Share your discoveries with your instructor and class colleagues

Testing with JUnit 4 and Java 5

- No need to `extend TestCase`, simply statically import assert statements: `import static org.junit.Assert.*;`
- No need to prefix method names with `test`, simply annotate them with JUnit's `@Test` annotation:

```
import org.junit.Test; //import annotation
…
@Test public void checkInvalidParams() { … }
```

  - Expect exceptions with `@Test(`*`expected=MyException`*`.class)`
  - Automatically time-out tests with `@Test(timeout=20)` (in ms)
- No need for just one `setUp` or `tearDown` methods, simply annotate any *setup* method with `@Before` and any teardown method with `@After` annotations from `org.junit.*`
  - Automatically inherited from parent classes and run in the right order
  - Also available `@BeforeClass` and `@AfterClass` for one-time setup and tear-down
- Temporarily ignore tests with `@Ignore("`*`Reason`*`")`
- Adapt to old test-runners with `junit.framework.JUnit4TestAdapter`

References

- K. Beck, "Test Driven Development: By Example", Addison-Wesley Professional, 2002.
- K. Beck, "JUnit Pocket Guide", O'Reilly Media, 2004.
- T. Husted and V. Massol, "JUnit in Action", Manning Publications, 2003.
- J. B. Rainsberger, "JUnit Recipes : Practical Methods for Programmer Testing", Manning Publications, 2004.