Module 3 (Part II)

The xUnit Architecture

❖ xUnit Family Members (Most popular xUnit test frameworks)

- Junit (used with Java)
- CppUnit (C++)
- Nunit (xUnit for .NET)
- PyUnit (Python version of xUnit)
- Sunit (used with the Smalltalk language)
- vbUnit (xUnit for Visual Basic (VB))
- utPLSQL (xUnit for Oracle’s PL/SQL language)
- MinUnit (used to test C code)

❖ xUnit Extensions

- Many add-on tools are available that extend the functionality of existing unit test frameworks into specialized domains
  - XMLUnit (xUnit extension to support XML testing)
  - JUnitPerf (JUnit extension)
  - Cactus (JUnit extension for unit testing server-side code such as servlets, JSPs)
  - JFCUnit (extension that supports writing GUI tests)
  - NUnitForms (extension that supports GUI tests of Windows Forms applications)
  - HTMLUnit (extension to JUnit that tests web-based applications)
  - HTTPUnit (Another JUnit extension that tests web-based applications)
  - Jester (A helpful extension to JUnit that automatically finds and reports code that is not covered by unit tests)

❖ The xUnit Architecture

- The xUnits all have the same basic architecture
- The other xUnits vary in their implementation details, but follow the same pattern and generally contain the same key classes and concepts
- The key classes are

1) TestCase
2) TestRunner
3) TestFixture
4) TestSuite
5) TestResult

1) TestCase

- xUnit’s most elemental class is **TestCase**, the base class for a unit test
- It is shown in Figure 3-1

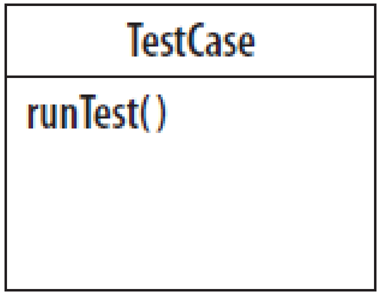
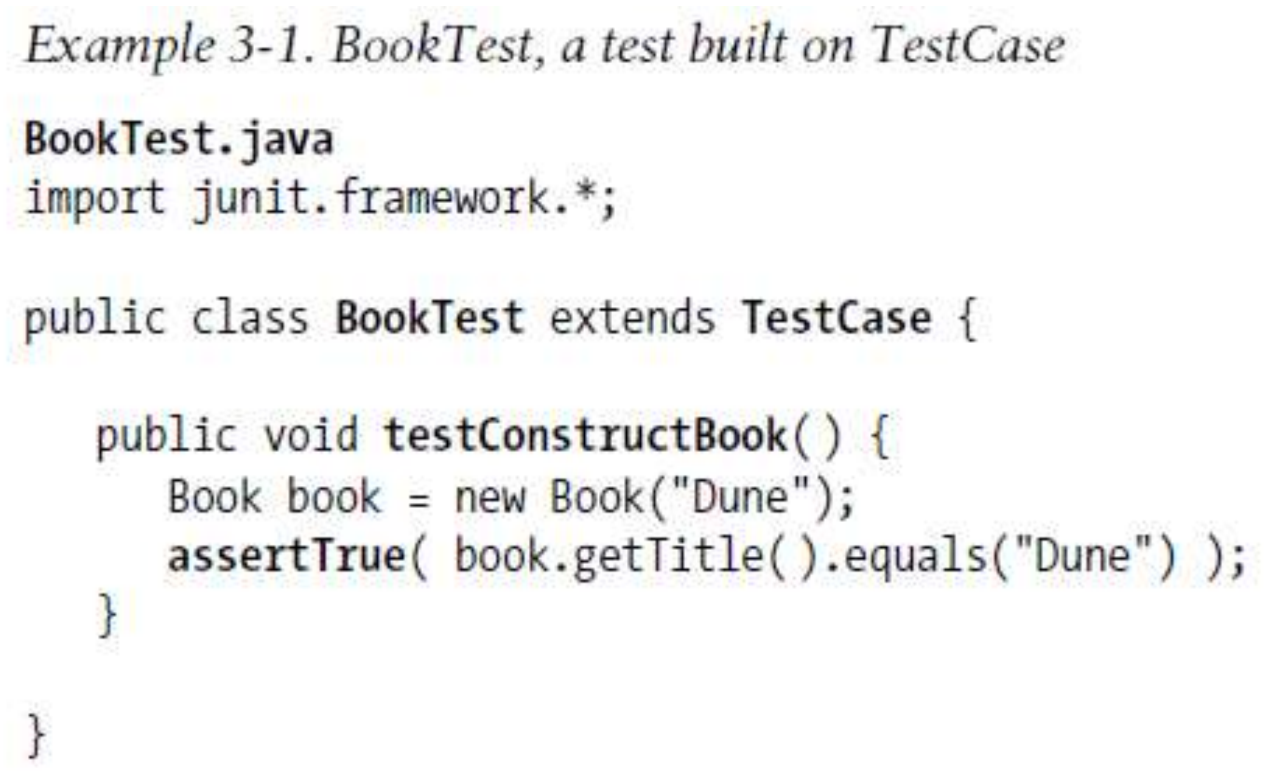


Figure 3-1. The abstract class TestCase, the parent of all xUnit unit tests

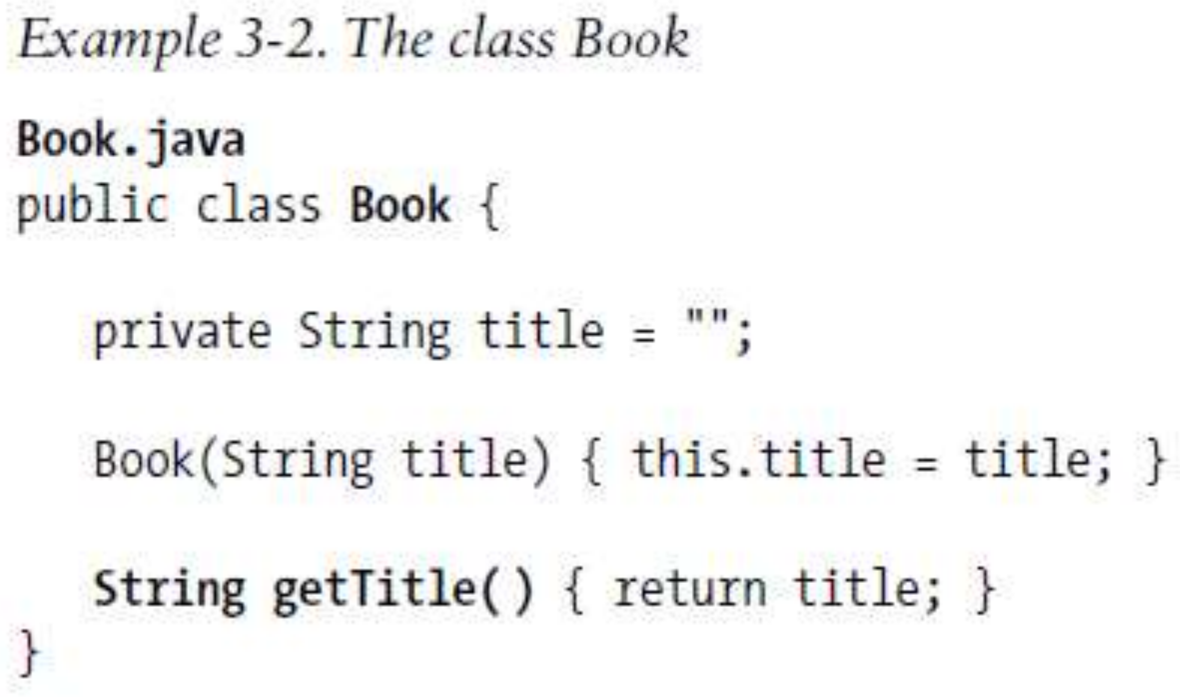
- All unit tests are inherited from **TestCase**
- To create a unit test, define a test class that is descended from TestCase and add a test method to it
- Example 3-1 shows the unit test BookTest



Example 3-1. BookTest, a test built on TestCase

```
BookTest.java
import junit.framework.*;

public class BookTest extends TestCase {

    public void testConstructBook( ) {
        Book book = new Book("Dune");
        assertTrue( book.getTitle( ).equals("Dune") );
    }

}
```

- The test method testConstructBook() uses assertTrue() to check the value of the Book’s title
- Test conditions always are evaluated by the framework’s assert methods
- If a condition evaluates to TRUE, the framework increments the successful test counter
- If it is FALSE, a test failure has occurred and the framework records the details, including the failure’s location in the code
- After a failure, the framework skips the rest of the code in the test method, since the test result is already known
- BookTest tests the class Book, shown in Example 3-2



Example 3-2. The class Book

```java
Book.java
public class Book {

   private String title = "";

   Book(String title) { this.title = title; }

   String getTitle() { return title; }
}
```

2) TestRunner

- A TestRunner reports details about the test results and simplifies the test
- It is a fairly complex object that, in JUnit, comes in three flavors: the AWT TestRunner, the Swing TestRunner, and the textual TestRunner (cleverly named TextTestRunner : Their purpose is to run one or more TestCases and report the results)
- Figure 3-2 shows TextTestRunner

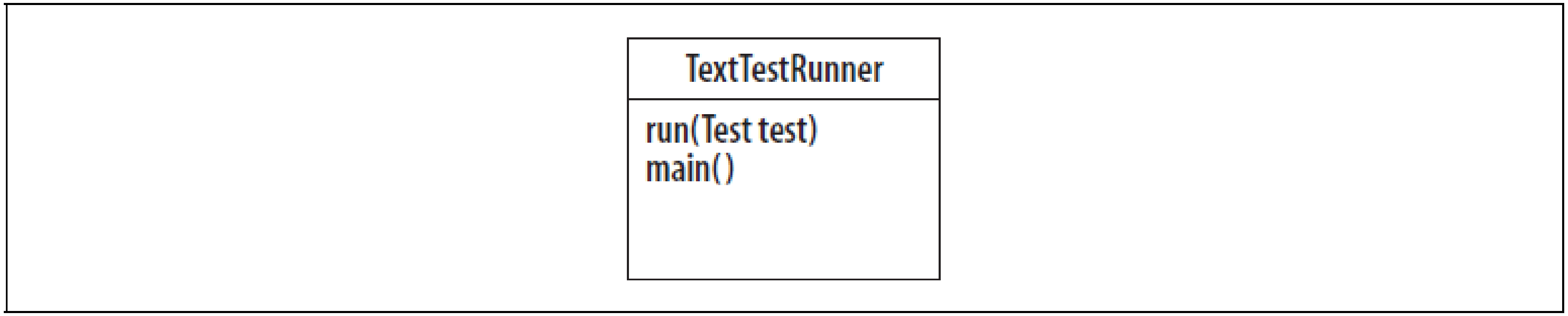
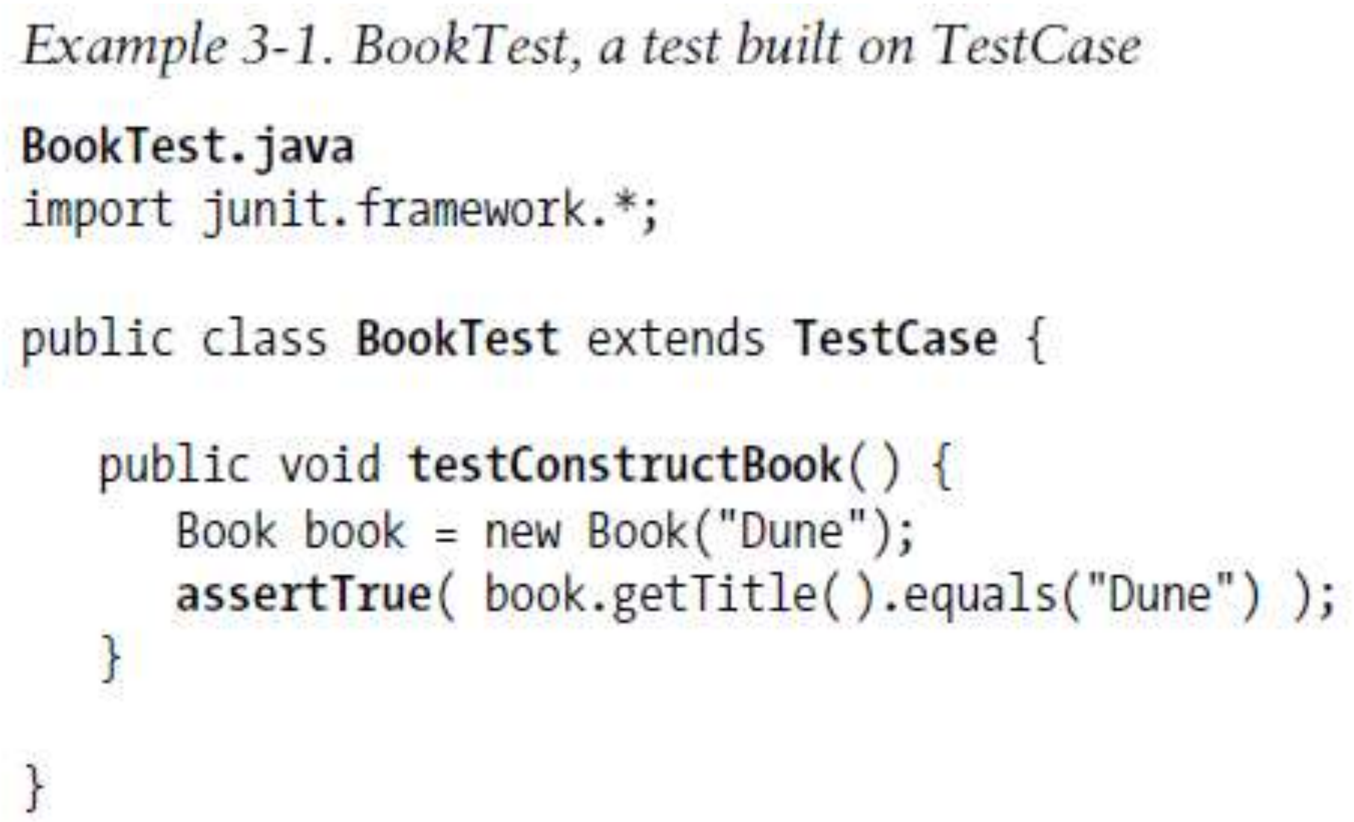


Figure 3-2. The class TextTestRunner

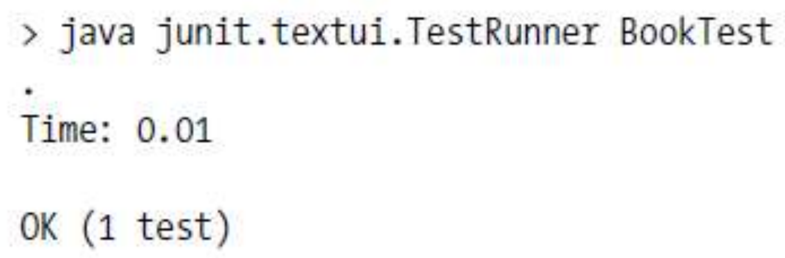
- The important methods of TextTestRunner are run(), which gives it a test to run, and main(), which makes TextTestRunner a runnable class
- TextTestRunner will be run with the test class BookTest as its argument
- It will find the test method testConstructBook and run it.



Example 3-1. BookTest, a test built on TestCase

```
BookTest.java
import junit.framework.*;

public class BookTest extends TestCase {

    public void testConstructBook( ) {
        Book book = new Book("Dune");
        assertTrue( book.getTitle( ).equals("Dune") );
    }

}
```

- Use TextTestRunner to run BookTest



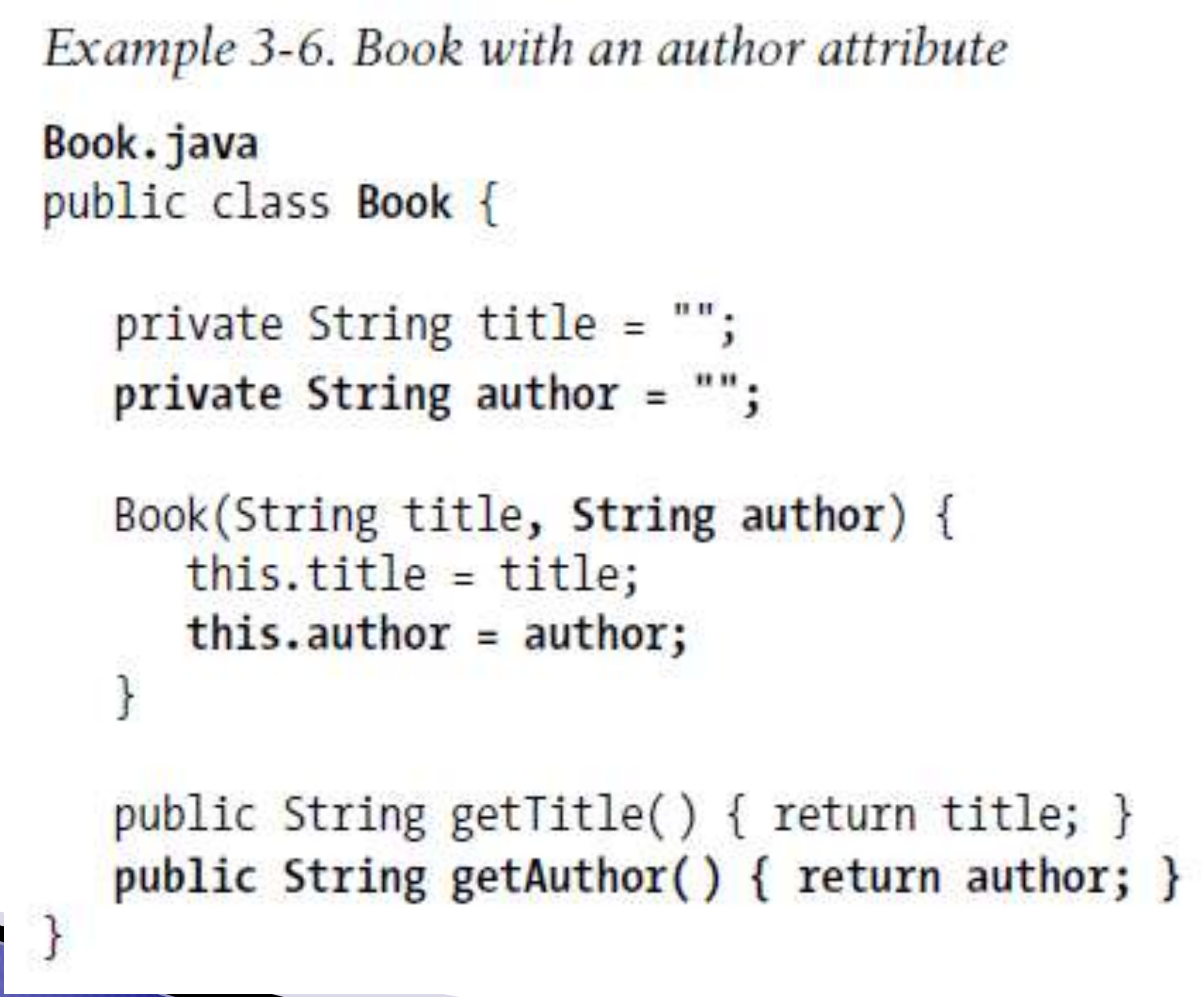
```
> java junit.textui.TestRunner BookTest
.
Time: 0.01

OK (1 test)
```

- Using the TestRunner not only takes unnecessary code out of BookTest, but also provides a nice report of how many tests were run and how long they took
- Test classes often have multiple test methods
- TestRunner will find all of the test methods that have names starting with test and run them
- Example 3-5 shows BookTest with a second test method added
- The new test validates a Book’s author

Example 3-5. BookTest with a second test method

```
BookTest.java
import junit.framework.*;

public class BookTest extends TestCase {

   public void testConstructBook( ) {
      Book book = new Book("Dune", "");
      assertTrue( book.getTitle( ).equals("Dune") );
   }

   public void testAuthor( ) {
      Book book = new Book("Dune", "Frank Herbert");
      assertTrue( book.getAuthor( ).equals("Frank Herbert") );
   }

}
```

- The author attribute and its accessor function getAuthor() are added to Book, as shown in Example 3-6



Example 3-6. Book with an author attribute

```
Book.java
public class Book {

   private String title = "";
   private String author = "";

   Book(String title, String author) {
      this.title = title;
      this.author = author;
   }

   public String getTitle( ) { return title; }
   public String getAuthor( ) { return author; }
}
```

Running BookTest shows that the framework now is running two tests:

```
> java junit.textui.TestRunner BookTest
..
Time: 0.01

OK (2 tests)
```

3) TestFixture

- The xUnit architecture helps to ensure test isolation with test fixtures
- A test fixture is a test environment used by multiple tests
- It is implemented as a TestCase with multiple test methods that share objects
- The shared objects represent the common test environment
- Figure 3-3 shows the relationship between a TestFixture and a TestCase

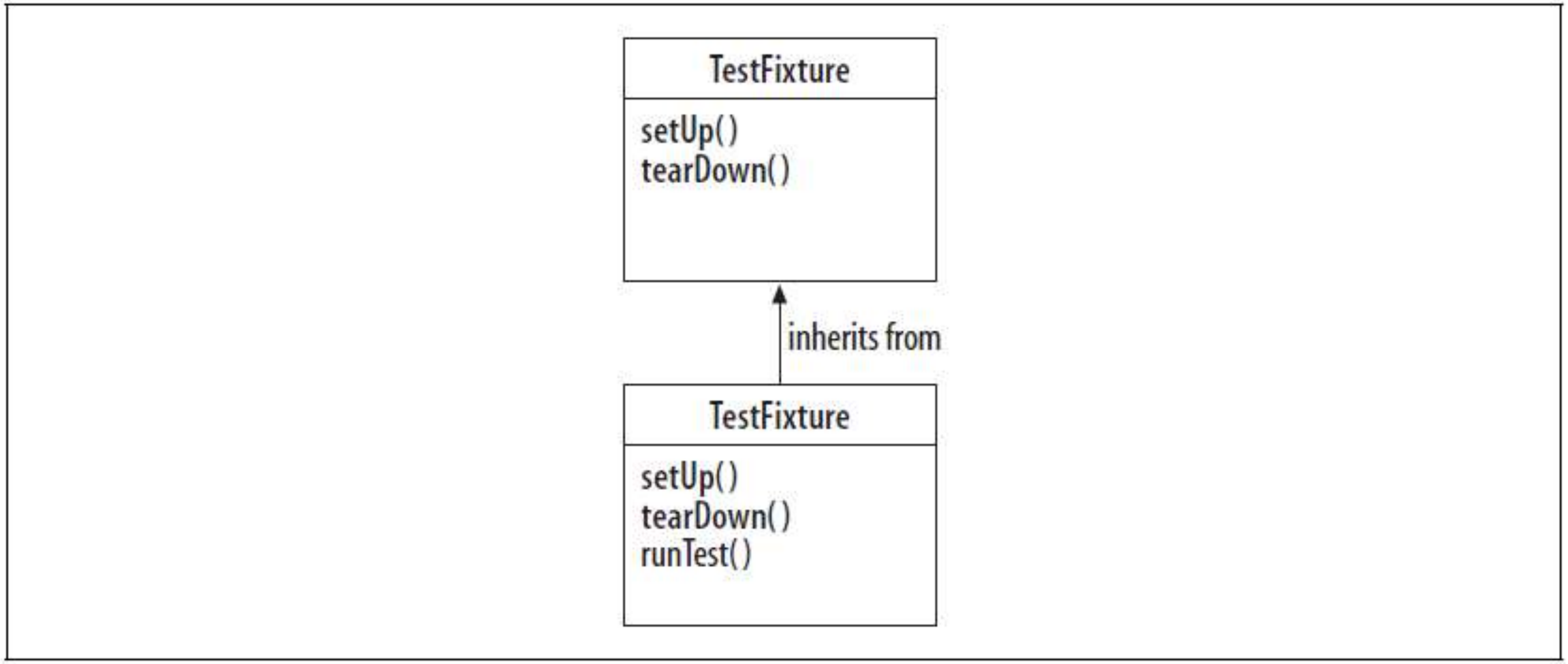


Figure 3-3. TestFixture and its child TestCase

- The setUp() method is called prior to each test method, establishing the initial environment for the test
- The tearDown() method is always called after each test method to clean up the test environment, even if there is a failure
- The TestFixture behavior effectively creates and destroys the test class each time one of its test methods is called

- Writing tests as TestFixtures has a number of advantages
  1) Test methods can share objects but still run in isolation
  2) Test coupling is minimized
  3) Test methods that share code can be grouped together in the same TestFixture
  4) Code duplication between tests is reduced
  5) The cleanup code is guaranteed to run whether a test succeeds or fails
  6) Finally, the test methods can be run in any order, since they are isolated
- Example 3-9 shows LibraryTest implemented as a TestFixture
- In this example, the test fixture's shared environment contains an instance of Library with two Books

Example 3-9. LibraryTest implemented as a TestFixture

```
LibraryTest.java
import junit.framework.*;
import java.util.*;

public class LibraryTest extends TestCase {

   private Library library;

   public void setUp( ) {
   library = new Library( );
   library.addBook(new Book("Dune", "Frank Herbert"));
   library.addBook(new Book("Solaris", "Stanislaw Lem"));
}

public void tearDown( ) {
}

public void testGetBooks( ) {
   Book book = library.getBook( "Dune" );
   assertTrue( book.getTitle( ).equals( "Dune" ) );
   book = library.getBook( "Solaris" );
   assertTrue( book.getTitle( ).equals( "Solaris" ) );
}

public void testLibrarySize( ) {
   assertTrue( library.getNumBooks( ) == 2 );
}

}
```

When LibraryTest is run, the sequence of function calls is:

```
setUp( )
testGetBooks( )
tearDown( )
setUp( )
testLibrarySize( )
tearDown( )
```

The calls to setUp( ) and tearDown( ) initialize and deinitialize the test fixture each time a test method is called, thus isolating the tests.

4) TestSuite

- xUnit contains a class for aggregating unit tests called TestSuite
- TestSuite is closely related to TestCase, since both are descendants of the same abstract class, Test
- Figure 3-4 shows the Test interface and how TestSuite and TestCase implement it

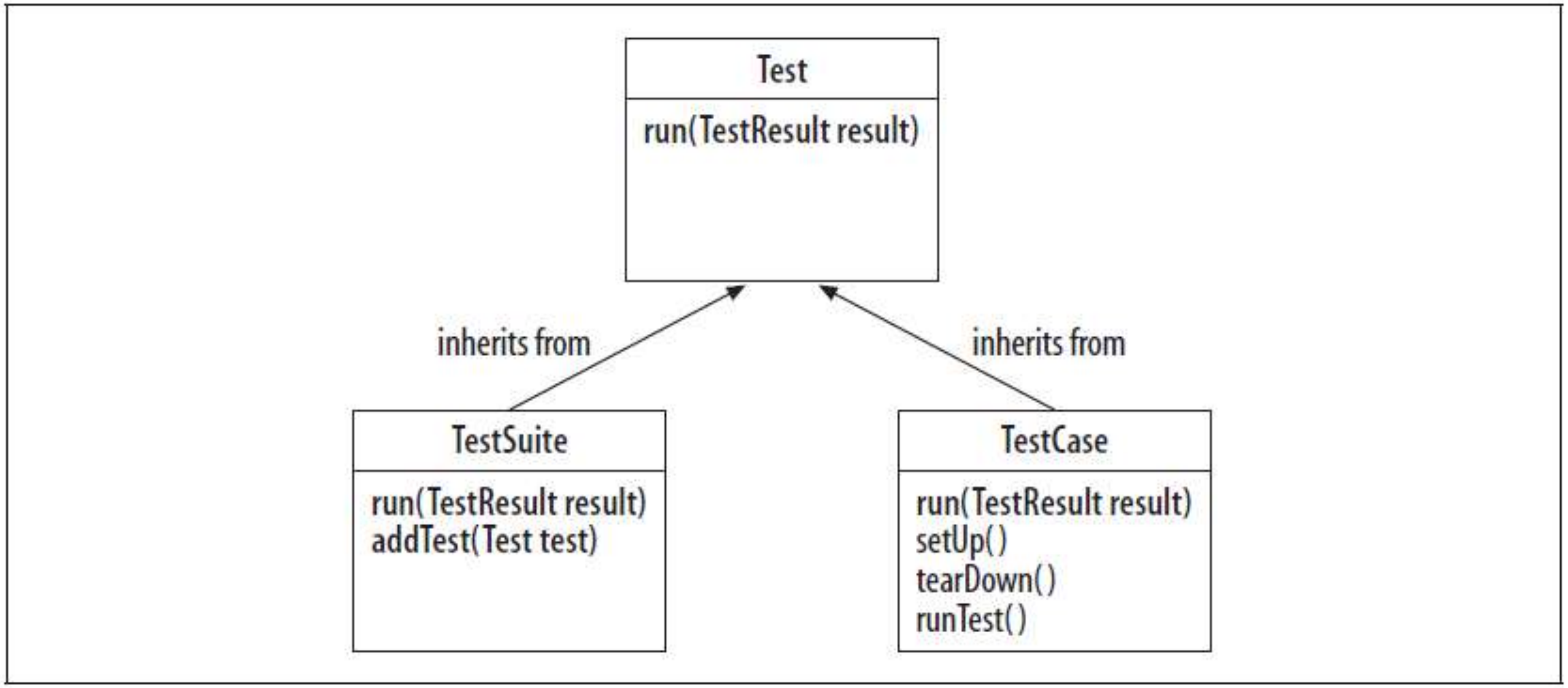


Figure 3-4. TestSuite, TestCase, and their parent interface Test

- The interface Test contains the run() method that the framework uses to run tests and collect their results
- Since TestSuite implements run(), it can be run just like a TestCase
- When a TestCase is run, its test methods are run
- When a TestSuite is run, its TestCases are run
- TestCases are added to a TestSuite using the addTest() method
- Since a TestSuite is itself a Test, a TestSuite can contain other TestSuites, allowing the intrepid developer to build hierarchies of TestSuites and TestCases

Example 3-10 shows a TestSuite-derived class named LibraryTests that contains both BookTest and LibraryTest.

Example 3-10. An instance of TestSuite named LibraryTests

LibraryTests.java

```
import junit.framework.*;

public class LibraryTests extends TestSuite {

   public static Test suite() {
      TestSuite suite = new TestSuite( );
      suite.addTest(new TestSuite(BookTest.class));
      suite.addTest(new TestSuite(LibraryTest.class));
      return suite;
   }

}
```

5) TestResult

- Each time a test is run, the TestResult object is passed in to collect the results
- Figure 3-5 shows TestResult

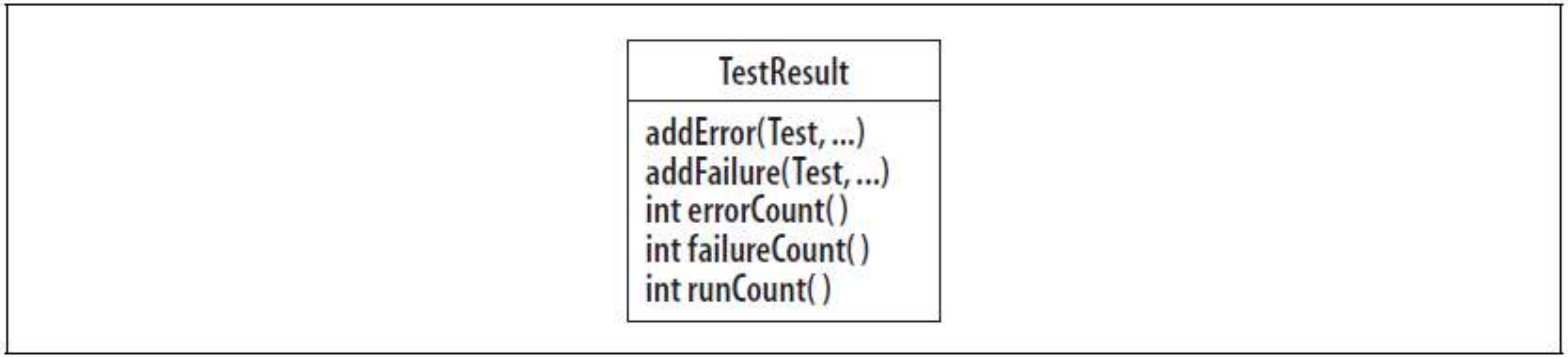


Figure 3-5. The class TestResult, used to collect test outcomes

- TestResult is a simple object
- It counts the tests run and collects test failures and errors so the framework can report them
- The failures and errors include details about the location in the code where they occurred and any associated test descriptions

❖ xUnit Architecture Summary

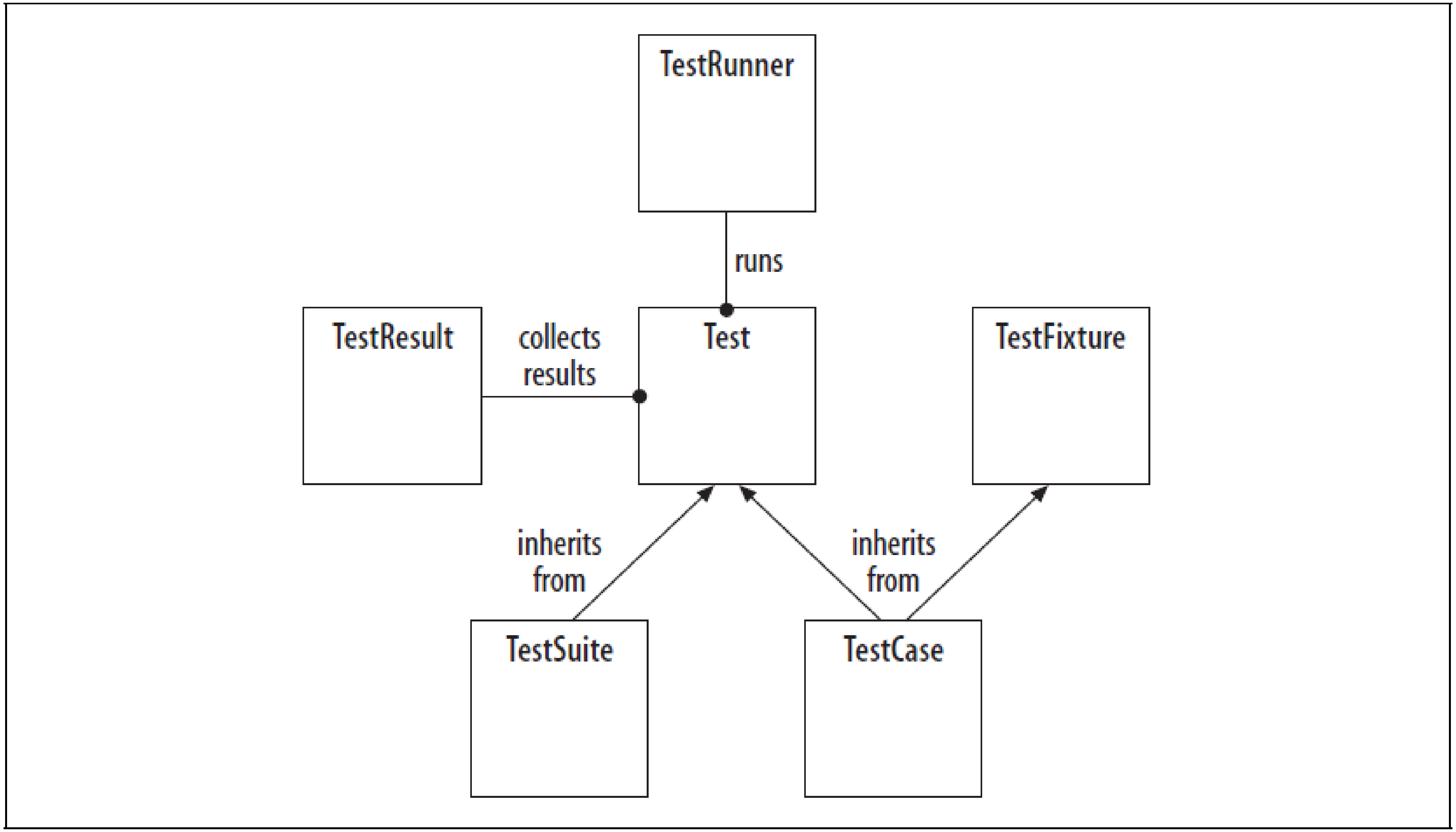


Figure 3-6. Core classes of the xUnit test framework architecture