Programming Assignment #2 (Java)

CS–671

due 23 Feb 2014 11:59pm

1. Implement the class `cs671.Tester`. 100 pts

The objective of this assignment is to implement a unit testing system similar in design to JUnit and TestNG. The assignment illustrates the use of Java reflection.

In JUnit and TestNG, test methods are annotated with a `@Test` annotation. In the system to implement in this assignment, we will use a `@TestMethod` annotation so both the `cs671` and the JUnit packages can be imported without conflicts. In such systems, the annotation identifies a method as a test method and specifies testing parameters (such as a textual description or a timeout value).

This assignment uses the annotation defined in Lis. 1, with an `info` and a `weight` parameters. This annotation is used to annotate methods of classes that implement the `Testable` interface, defined in Lis. 2. This interface specifies two methods, `beforeMethod` and `afterMethod`, which the system will run before and after each test method, respectively. These methods can be used to setup a test (e.g., instantiate objects, open files, connect to databases) and free resources (e.g., close file descriptors, delete temporary files) after the test has run.

The testing system, to be implemented, is defined in class `Tester`, which offers the following features:

- The class is abstract and has no public constructor. Instances are built using static methods: `makeTester` and `makeSuite`.
- `makeTester` builds a tester from a `Testable` class. When the tester is run, the class is instantiated and all the test methods are run on this instance. It is assumed that the class has a public no-argument constructor, which is used for instantiation.
- `makeSuite` builds a tester from a collection of testers. Each of these testers represent a test class or a suite (i.e., suites can be nested). By using `makeSuite`, one can build large, hierarchical collections of tests.
- Test methods are identified by the `@TestMethod` annotation. They must be zero-argument, non-static methods. However, they are not required to be public or `void`.[1]
- The `Tester` class is `Runnable`. Its `run` method runs all the tests in the test class or suite. For each class, a single instance is created using a no-argument constructor and all the tests are run on it. Tests are run in order of their names. All the test classes (or sub-suites) of a suite are run in the order in which they were given.
- For each test method, the system first runs `beforeMethod`. If this returns `false` or fails, the test is not run. Otherwise, the test method is run, followed by `afterMethod` (whether the test was successful or not).
- For each test that runs, a `TestResult` object is created. It contains information about the test (test name and weight, whether the test was successful, cause of failure (if any) and running time). These test results can be retrieved via method `getResults`.
- The class has a `setPrintWriter` method, used to specify how the tester should output warnings and errors. By default, warnings and errors are sent to `System.err`. Calling `setPrintWriter(w)` redirects all warning and errors to writer `w`. Test failures are *not* reported to the writer. A call `tester.setPrintWriter(null)` is valid and makes `Tester.run` completely silent, even in the presence of errors.
- Finally, the class implements a terminal-based application, which takes testable class names on the command-line and produces a report on `System.out`. A sample output of the application is shown on Fig. 1 for the testing class defined in Lis. 3.

[1] This is a difference with JUnit.

```java
package cs671;

import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;
import java.lang.annotation.Retention;

/** Annotation for test methods. This is the only annotation used in the testing system. */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
public @interface TestMethod {

  /** The weight of this test.  Tests with negative weight are silently ignored. */
  double weight () default 0;

  /** A description of the test. */
  String info () default "";
}
```

Listing 1: `TestMethod.java`

```java
package cs671;

import java.lang.reflect.Method;

/** Testable classes.  Test methods are annotated with {@code TestMethod}. @see TestMethod */
public interface Testable {

  /** Executed before each test.  If this method fails or return false, the corresponding test is not run.
   *
   * @param m the test method soon to be run
   */
  public boolean beforeMethod (Method m) throws Exception;

  /** Executed after each test.  This method runs after each test, whether the test was successful or not.
   * It does not run if the corresponding test was not run.
   *
   * @param m the test method that was just run
   */
  public void afterMethod (Method m) throws Exception;
}
```

Listing 2: `cs671.tester.Testable.java`

```
> java cs671.Tester SomeTests
WARNING: beforeMethod(failTest) failed: failed to init
WARNING: beforeMethod(initTest3) is false; test not run.
WARNING: test6 is static; ignored.
SUCCESSFUL TESTS:
  SomeTests.test2: a test that 2+2=4 (3.0) in 0.008 milliseconds
  SomeTests.test3: a long running test (0.0) in 5432.736 milliseconds
FAILED TESTS:
  SomeTests.test1: a test that 2+2=5 (2.0) from java.lang.AssertionError: expected [5] but found [4]
  SomeTests.test4 (0.0) from java.lang.OutOfMemoryError: Java heap space
SCORE = 60.0%
```

Figure 1: Sample output from `cs671.Tester.main`

```
import java.lang.reflect.Method;
import cs671.TestMethod;
import static org.testng.Assert.*;

class SomeTests implements cs671.Testable {
  public boolean beforeMethod (Method m) throws Exception {
    String name = m.getName();
    if (name.startsWith("init"))
      return false;
    if (name.startsWith("fail"))
      throw new Exception("failed to init");
    return true;
  }
  public void afterMethod (Method m) throws InterruptedException {
    Thread.sleep(1000);
  }

  @TestMethod(weight=Math.PI)
  void failTest () {
  }
  @TestMethod(weight=2, info="a test that 2+2=5")
  void test1 () {
    assertEquals(2 + 2, 5);
  }
  @TestMethod(weight=3, info="a test that 2+2=4")
  int test2 () {
    assertEquals(2 + 2, 4);
    return 42;
  }
  @TestMethod void initTest3 () {
  }
  @TestMethod(info="a long running test")
  void test3 () throws InterruptedException {
    Thread.sleep(5432);
  }
  @TestMethod void test4 () {
    class C {
      C next;
      C (C c) {
        next = c;
      }
    }
    C c = null;
    while (true)
      c = new C(c);
  }
  @TestMethod(weight=-1)
  void test5 () {
    System.out.println("not run for now");
  }
  @TestMethod static void test6 () {
    System.out.println("never run");
  }
}
```

Listing 3: `SomeTests.java`

Notes:

- APIs of classes, annotations and interfaces are described at: `http://www.cs.unh.edu/~cs671/Java`.
- The implementation of class `Tester` relies on *reflection*. Specifically, reflection is used to:
  - lookup and load classes given their names (e.g., `Class.forName`);
  - check if classes are testable (e.g., `Class.isAssignableFrom`);
  - create instances of testable classes (e.g., `Class.getDeclaredConstructor`, `Constructor.newInstance`);
  - list methods in classes (e.g., `Class.getDeclaredMethods`);
  - examine method annotations (e.g., `Method.getAnnotation`);
  - examine method modifiers (e.g., `Method.getModifiers`);
  - invoke test methods (e.g., `Method.invoke`);
  - ...
- One reason for using static methods instead of constructors is that static methods can return instances of different classes. This is often convenient, as in this assignment. There is no good reason for `makeTester` and `makeSuite` to build instances using the same class and a good design should use two different (and non-public) classes.
- For grading purposes, it is important that the output from the terminal-based application follows the pattern shown in Fig. 1. In particular, it should includes the strings `"SUCCESSFUL TESTS:"`, `"FAILED TESTS:"` and `"SCORE ="` *exactly*. Each test result line should also be formatted *exactly* as `"␣␣<info>␣(<weight>)␣in␣<millis>␣milliseconds"` or `"␣␣<info>␣(<weight>)␣from␣<cause>"`. In the same way, the string returned by `TestResult.getInfo` is precisely defined in the `TestResult` API and its definition should be followed rigorously.
- Running times can be measured using `System.currentTimeMillis` or `System.nanoTime`. Note that running times *do not* include `beforeMethod` and `afterMethod`, and that `TestResult.duration` returns a duration *in seconds*.
- *Errors* and *warnings* are reported to the specified `PrintWriter` (unless null). When a test class cannot be instantiated, it is an error (and none of its tests are run). Warnings include:
  - A method annotated with `@TestMethod` requires parameters.
  - A method annotated with `@TestMethod` is static.
  - A call to `beforeMethod` fails or return false.
  - A call to `afterMethod` fails.
- Test methods are not necessarily `void`; classes, test methods and constructors are not necessarily `public`.
- Class lookup, loading, initialization and instantiation can throw all sorts of exceptions and errors. These tend to be wrapped inside reflection exceptions.
- The "score" in the application is calculated from the weights of all tests methods that were run. Tests that were ignored do not participate in the calculation: $60 = 100 \cdot \frac{3+0}{3+0+2+0}$.
- If one of the classes specified on the command-line of the application cannot be loaded or is not testable, the application stops with an error message and does not run any test, even if other classes were acceptable. For instance, the invocation below *does not* run the test methods of class `SomeTests`:

```
> java cs671.Tester SomeTests java.lang.String
class 'java.lang.String' is not Testable
>
```