



CHAPTER 12

12

Unit Test

Unit test

Suppose that you are writing a `CourseCatalog` class to record the information of some courses:

```
class CourseCatalog {
    CourseCatalog() {
        ...
    }
    void add(Course course) {
        ...
    }
    void remove(Course course) {
        ...
    }
    Course findCourseWithId(String id) {
        ...
    }
    Course[] findCoursesWithTitle(String title) {
        ...
    }
}
class Course {
    Course(String id, String title, ...) {
    }
    String getId() {
        ...
    }
    String getTitle() {
        ...
    }
}
```

In order to ensure that `CourseCatalog` is free of bug, we should test it. For example, in order to see if its `add` method is really adding a `Course` into it, we can do it this way:

```
class TestCourseCatalog {
    static void testAdd() {
        CourseCatalog cat = new CourseCatalog();
        Course course = new Course("c001", "Java programming", ...);
        cat.add(course);
        if (!cat.findCourseWithId(course.getId()).equals(course)) {
            throw new TestFailException();
        }
    }
}
public static void main(String args[]) {
    testAdd();
}
```

```
    }
}
```

Because `testAdd` needs to check if two `Course` objects are equal, the `Course` class needs to provide an `equals` method:

```
class Course {
    ...
    boolean equals(Object obj) {
        ...
    }
}
```

Maybe we are not very confident about the `add` method yet. Maybe we are worried that it may not be able to store two or more courses. Therefore we write another test:

```
class TestCourseCatalog {
    static void testAdd() {
        ...
    }
    static void testAddTwo() {
        CourseCatalog cat = new CourseCatalog();
        Course course1 = new Course("c001", "Java prgoramming", ...);
        Course course2 = new Course("c002", "OO design", ...);
        cat.add(course1);
        cat.add(course2);
        if (!cat.findCourseWithId(course1.getId()).equals(course1)) {
            throw new TestFailException();
        }
        if (!cat.findCourseWithId(course2.getId()).equals(course2)) {
            throw new TestFailException();
        }
    }
    public static void main(String args[]) {
        testAdd();
        testAddTwo();
    }
}
```

Similarly, we can test the `remove` method in `CourseCatalog`:

```
class TestCourseCatalog {
    static void testAdd() {
        ...
    }
    static void testAddTwo() {
        ...
    }
    static void testRemove() {
        CourseCatalog cat = new CourseCatalog();
        Course course = new Course("c001", "Java prgoramming", ...);
        cat.add(course);
        cat.remove(course);
        if (cat.findCourseWithId(course.getId()) != null) {
            throw new TestFailException();
        }
    }
}
```

```

    }
}
public static void main(String args[]) {
    testAdd();
    testAddTwo();
    testRemove();
}
}

```

The purpose of `testAdd`, `testAddTwo` and `testRemove` above is to test the `CourseCatalog` class. Each of them is called a test case. However, they are not acceptance tests, but are the so-called "unit tests", because they only test a unit (`CourseCatalog` class).

How is a unit test differ from an acceptance test? An acceptance test tests the external behavior of a system, while a unit test tests a unit (class). Acceptance tests belong to the client. We have no right to determine their contents. We can at most help the customer write down the acceptance tests according to the user stories. Units tests belong to us, because what classes are there in the system and what each class should do are decided by us. The client has no right to get involved, nor do we need his participation. We simply write the unit tests according to our expectation on a unit (class). Because of this, this kind of test is also called "programmer test".

Use JUnit

When writing unit tests, we can make use of a software package called "JUnit". With JUnit, we should change the above unit testing code to:

```

import junit.framework.*;
import junit.swingui.*;
public class TestCourseCatalog extends TestCase {
    CourseCatalog cat;
    protected void setUp() {
        cat = new CourseCatalog();
    }
    public void testAdd() {
        Course course = new Course("c001", "Java prgoramming", ...);
        cat.add(course);
        assertEquals(cat.findCourseWithId(course.getId()), course);
    }
    public void testAddTwo() {
        Course course1 = new Course("c001", "Java prgoramming", ...);
        Course course2 = new Course("c002", "OO design", ...);
        cat.add(course1);
        cat.add(course2);
        assertEquals(cat.findCourseWithId(course1.getId()), course1);
        assertEquals(cat.findCourseWithId(course2.getId()), course2);
    }
    public void testRemove() {
        Course course = new Course("c001", "Java prgoramming", ...);
        cat.add(course);
    }
}

```

```
        cat.remove(course);
        assertTrue(cat.findCourseWithId(course.getId()) == null);
    }
    public static void main(String args[]) {
        TestRunner.run(TestCourseCatalog.class);
    }
}
```

Below we will explain the changes.

The `assertEquals(X, Y)` method checks if `X` and `Y` are equal (it calls the `equals` method of `X`). If not, it throws an exception. `assertEquals` is provided by the `TestCase` class. Because now `TestCourseCatalog` extends `TestCase`, we can directly call `assertEquals`.

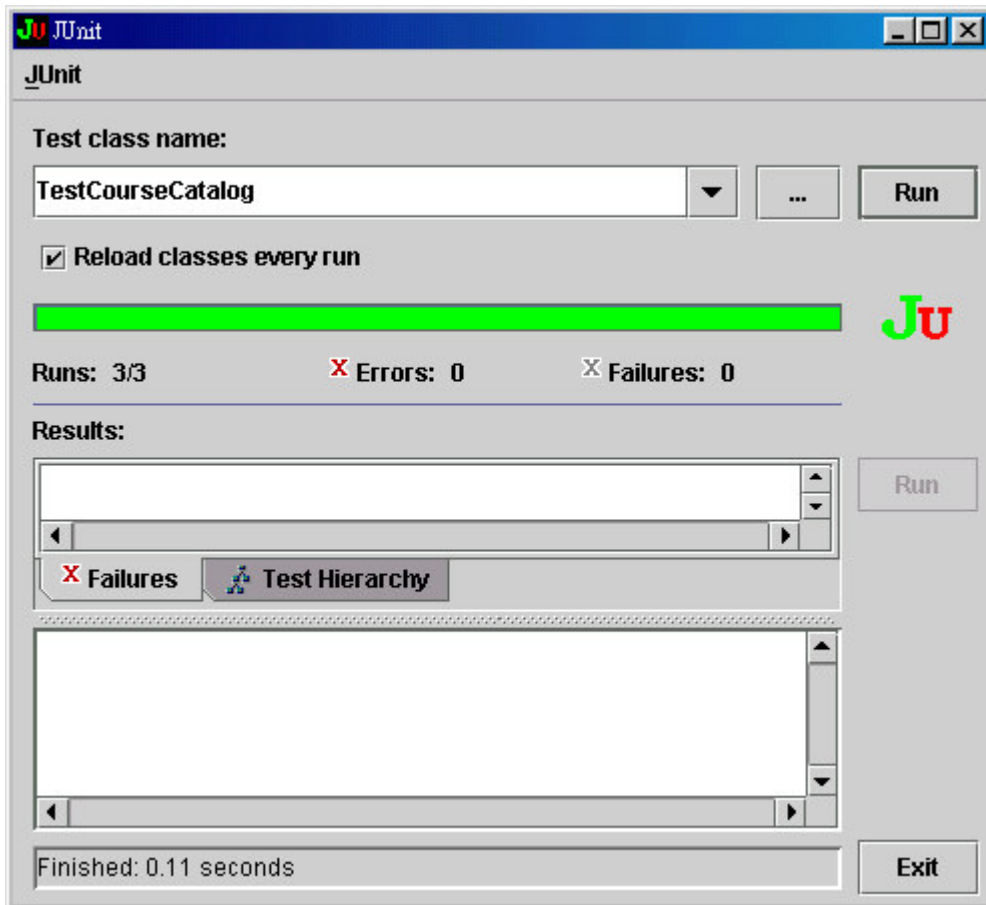
The `TestCase` class is provided by JUnit. It is in the `junit.framework` package. Therefore, we need to import `junit.framework.*` in order to use it.

`testAdd`, `testAddTwo` and `testRemove` all need to create a `CourseCatalog` object. We have put this statement into the `setUp` method. As we will see, before each test case is run, this `setUp` method will be called.

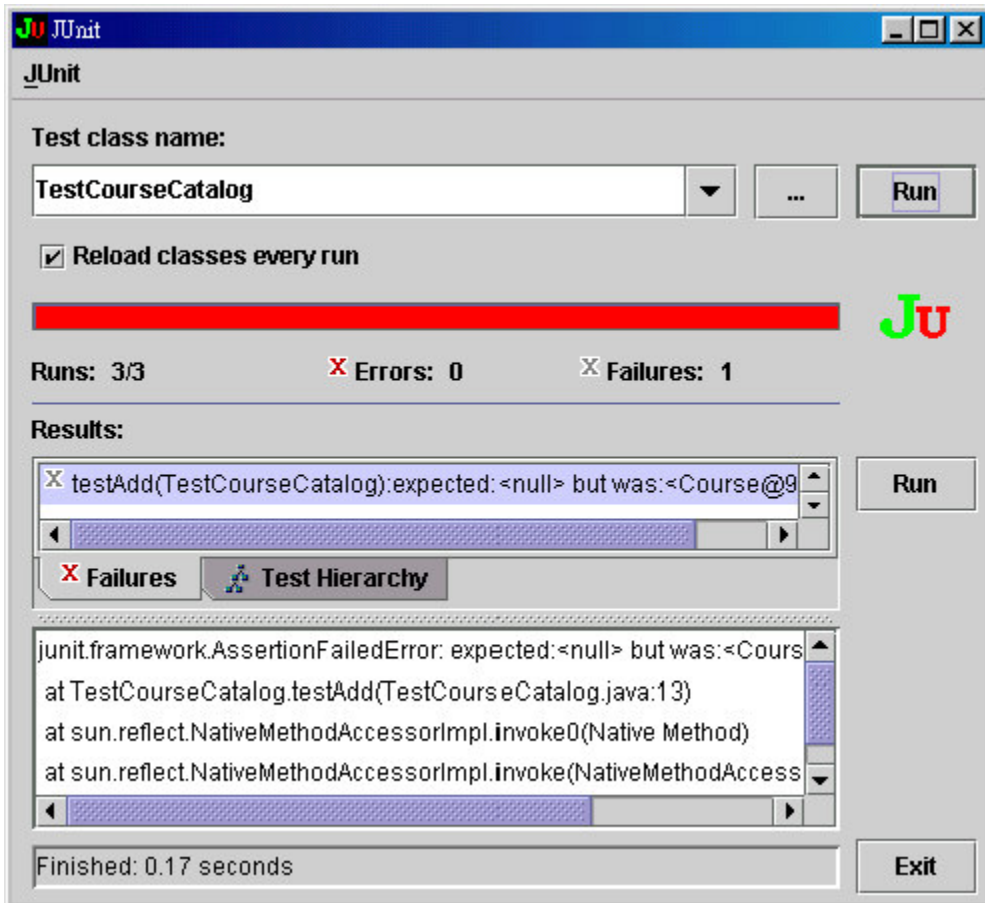
Because the `cat` variable is now initialized in `setUp`, but it is used in `testAdd` and etc., it has to be an instance variable (originally it was a local variable in each method). In addition, `testAdd` and etc. must now be instance methods (originally they were static methods).

`testRemove` uses `assertTrue(X)`, which checks if `X` is true. If not, it throws an exception. Just like `assertEquals`, it is provided by the `TestCase` class.

The main method calls a static method named `run` in the `TestRunner` class. It will display a window like the one shown below and run the three test cases: `testAdd`, `testAddTwo` and `testRemove`.



If all three test cases pass, it will display a green progress bar (as shown above). If some test cases fail, it will display a red progress bar and indicate which line in which source file caused the error:



The `TestRunner` class is provided by JUnit. It is in the `junit.swingui` package. Therefore, we need to import `junit.swingui.*` in order to use it.

How does `TestRunner` know that we have defined these three test cases (`testAdd` and etc.)? We call it like this: `TestRunner.run(TestCourseCatalog.class)`. This `run` method will use the reflection ability of Java to find those methods in `TestCourseCatalog` that are public, whose names start with "test" and that take no parameters. It takes each such method as a test case. Therefore, now we must declare `testAdd` and etc. as public.

In order to run `testAdd`, `TestRunner` will create a new `TestCourseCatalog` object, call its `setUp` method (if any), then call `testAdd` and finally call its `tearDown` method (if any). In order to run `testAddTwo`, `TestRunner` will create another new `TestCourseCatalog` object, call its `setUp` method (if any), then call `testAdd` and finally call its `tearDown` method (if any). Therefore, `testAdd` and `testAddTwo` are not running on the same `TestCourseCatalog` object.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

TestRunner are not in the same package as TestCourseCatalog. In order to let TestRunner create a TestCourseCatalog object, the TestCourseCatalog class must be made public.

Usually what are setUp and tearDown used for? If we establish a database connection in setUp, usually we will close the connection in tearDown. Similarly, if we create a temp file in setUp, usually we will delete it in tearDown.

Do we need to unit test all classes

Basically, we should unit test every method in every class in the system, unless that method is too simple to break. For example, for the Course class:

```
class Course {
    String id;
    String title;
    Course(String id, String title, ...) {
        this.id = id;
        this.title = title;
    }
    String getId() {
        return id;
    }
    String getTitle() {
        return title;
    }
    boolean equals(Object obj) {
        if (obj instanceof Course) {
            Course c = (Course)obj;
            return this.id.equals(c.id) && this.title.equals(c.title);
        }
        return false;
    }
}
```

Because its constructor, getId and getTitle methods are very simple, we don't need to test them. What is left is just the equals method:

```
import junit.framework.*;
import junit.swingui.*;
public class TestCourse extends TestCase {
    public void testEquals() {
        Course course1 = new Course("c001", "Java prgoramming");
        Course course2 = new Course("c001", "Java prgoramming");
        Course course3 = new Course("c001", "OO design");
        Course course4 = new Course("c002", "Java prgoramming");
        assertEquals(course1, course2);
        assertTrue(!course1.equals(course3));
        assertTrue(!course1.equals(course4));
    }
    public static void main(String args[]) {
        TestRunner.run(TestCourse.class);
    }
}
```

```
}
```

How to run all the unit tests

Now there are four unit test cases in the system. Three of them test CourseCatalog; One tests Course. We hope to be able to frequently run all the unit test cases in the system in one go. To do that, we may do it this way:

```
public class TestAll {
    public static void main(String args[]) {
        TestRunner.run(TestAll.class);
    }
    public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTest(new TestSuite(TestCourseCatalog.class));
        suite.addTest(new TestSuite(TestCourse.class));
        return suite;
    }
}
```

The suite method in the above code creates and returns a TestSuite object, which contains all the test cases in the system. We can add several test cases to a TestSuite. We can also add several TestSuites into another TestSuite. When TestRunner runs a TestSuite, it will actually run all the test cases in the TestSuite.

The main method in the code above calls TestRunner.run(TestAll.class). As mentioned above, the run method will find those methods in the TestAll class that are public, whose names start with "test" and that take no parameters and then take them as test cases. However, before performing this searching, it will first check if TestAll has a static method named "suite". If yes, it will call this suite method and use the TestSuite it returns instead of performing the searching.

If we create a new class such as TimeTable, we need to create the corresponding TestTimeTable to unit test TimeTable. At that time, we need to add a line to TestAll (very easy to forget. Be careful):

```
public class TestAll {
    public static void main(String args[]) {
        TestRunner.run(TestAll.class);
    }
    public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTest(new TestSuite(TestCourseCatalog.class));
        suite.addTest(new TestSuite(TestCourse.class));
        suite.addTest(new TestSuite(TestTimeTable.class));
        return suite;
    }
}
```

When to run the unit tests

When to run the unit tests? For example:

- After we have just finished writing or changing CourseCatalog, we should run the main in TestCourseCatalog.
- After we have just finished writing or changing Course, we should run the main method in TestCourse.
- When we are about to run all the acceptance tests that have passed, we should run the main method in TestAll first.
- When we have free time, we should run the main method in TestAll.

Keep the unit tests updated

Suppose that we have added a new method like getCount in CourseCatalog. We should add one or more test cases in TestCourseCatalog to test it, e.g.,:

```
public class TestCourseCatalog extends TestCase {
    CourseCatalog cat;
    ...
    public void testGetCountOnEmptyCatalog() {
        assertEquals(cat.getCount(), 0);
    }
    public void testGetCount() {
        Course course1 = new Course("c001", "Java prgoramming", ...);
        Course course2 = new Course("c002", "Java prgoramming", ...);
        cat.add(course1);
        cat.add(course2);
        assertEquals(cat.getCount(), 2);
    }
}
```

Use unit tests to prevent the same bug from striking again

When we find a bug in our code, e.g., the remove(Course course) method in CourseCatalog should only delete this particular course object, but it incorrectly deletes all the other courses with the same titles as this one. After finding this bug, we shouldn't rush to fix it. Instead, we should add the following test case in TestCourseCatalog:

```
public class TestCourseCatalog extends TestCase {
```

```

CourseCatalog cat;
...
public void testRemoveKeepOthersWithSameTitle() {
    Course course1 = new Course("c001", "Java prgoramming", ...);
    Course course2 = new Course("c002", "Java prgoramming", ...);
    cat.add(course1);
    cat.add(course2);
    cat.remove(course1);
    assertEquals(cat.findCourseWithId(course2.getId()), course2);
}
}

```

Of course, currently this test case will fail. It serves as a bug report, telling us that there is a bug in our code. An ordinary bug report can be put aside and ignored. In contrast, we have to run all the unit tests. When faced with the red progress bar in TestRunner, we will never forget this bug.

Another more important effect of this unit test is, if in the future we introduce the same bug into the remove method, this unit test will immediately tell us something is wrong.

How to test if an exception is thrown

We can call remove to delete a Course from CourseCatalog. If there is no such a Course in CourseCatalog, what should remove do? It can do nothing, return an error code or throw an exception. Now, let's assume that it should throw a CourseNotFoundException:

```

class CourseNotFoundException extends Exception {
}
class CourseCatalog {
    void remove(Course course) throws CourseNotFoundException {
        ...
    }
}

```

Accordingly, we need to test if it is really doing that. Therefore, in TestCourseCatalog we add a test case:

```

public class TestCourseCatalog extends TestCase {
    CourseCatalog cat;
    ...
    public void testRemove() {
        ...
    }
    public void testRemoveNotFound() {
        Course course1 = new Course("c001", "Java prgoramming", ...);
        Course course2 = new Course("c002", "OO design", ...);
        cat.add(course1);
        cat.remove(course2); //how to test if it throws a CourseNotFoundException?
    }
}

```

The question is, how to test if `cat.remove(course2)` really throws a `CourseNotFoundException`? We may do it this way:

```
public class TestCourseCatalog extends TestCase {
    ...
    public void testRemoveNotFound() {
        Course course1 = new Course("c001", "Java prgoramming", ...);
        Course course2 = new Course("c002", "OO design", ...);
        cat.add(course1);
        try {
            cat.remove(course2);
            assertTrue(false);
        } catch (CourseNotFoundException e) {
        }
    }
}
```

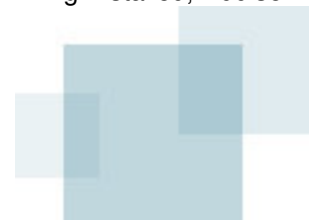
If it doesn't throw any exception, `assertTrue(false)` will be executed and will make the test fail. If the code does throw a `CourseNotFoundException`, it will be caught and then nothing more is done. So the test will be considered to have succeeded. If the code throws an exception other than a `CourseNotFoundException`, the exception will propagate to JUnit. When JUnit receives an exception, it considers that the test has failed.

Instead of calling `assertTrue(false)`, we can call the fail function provided by JUnit:

```
public void testRemoveNotFound() {
    ...
    try {
        cat.remove(course2);
        fail(); //It is the same as assertTrue(false).
    } catch (CourseNotFoundException e) {
    }
}
```

References

- <http://www.junit.org>.
- <http://c2.com/cgi/wiki?UnitTest>.
- Ron Jeffries, Ann Anderson, Chet Hendrickson, Extreme Programming Installed, Addison-Wesley, 2000.



Chapter exercises

Problems

1. Unit test the `findCoursesWithTitle` method in `CourseCatalog`.
2. If we use the `add` method in `CourseCatalog` to add a `Course` object but there is another `Course` object in `CourseCatalog` with the same ID, the `add` method will throw a `CourseDuplicateException`. Unit test this behavior.
3. Add a `clear` method to `CourseCatalog`. It will delete all the `Course` objects in `CourseCatalog`. Unit test this `clear` method.

Hints

1. No hint for this one.
2. No hint for this one.
3. In order to test this `clear` method, you may need to add a new method in `CourseCatalog`.

Sample solutions

1. Unit test the `findCoursesWithTitle` method in `CourseCatalog`.

```
public class TestCourseCatalog extends TestCase {
    ...
    public void testFindCoursesWithTitle() {
        Course course1 = new Course("c001", "Java prgramming", ...);
        Course course2 = new Course("c002", "OO design", ...);
        Course course3 = new Course("c003", "Java Programming", ...);
        cat.add(course1);
        cat.add(course2);
        cat.add(course3);
        Course courses[] = cat.findCoursesWithTitle(course1.getTitle());
        assertEquals(courses.length, 2);
        assertEquals(courses[0], course1);
        assertEquals(courses[1], course3);
    }
}
```

2. If we use the `add` method in `CourseCatalog` to add a `Course` object but there is another `Course` object in `CourseCatalog` with the same ID, the `add` method will throw a `CourseDuplicateException`. Unit test this behavior.

```
public class TestCourseCatalog extends TestCase {
    ...
    public void testAddDup() {
        Course course1 = new Course("c001", "Java prgramming", ...);
        Course course2 = new Course("c002", "OO design", ...);
        cat.add(course1);
        try {
            cat.add(course2);
            fail();
        } catch (CourseDuplicateException e) {
        }
    }
}
```

3. Add a `clear` method to `CourseCatalog`. It will delete all the `Course` objects in `CourseCatalog`. Unit test this `clear` method.

```
public class TestCourseCatalog extends TestCase {
    public void testClear() {
        Course course1 = new Course("c001", "Java prgramming", ...);
        Course course2 = new Course("c002", "OO design", ...);
        cat.add(course1);
        cat.add(course2);
        cat.clear();
        assertEquals(cat.countCourses(), 0);
    }
}
class CourseCatalog {
    ...
    int countCourses() {
        ...
    }
}
```

}