COMP0104 Software Development Practice: Unit Test with JUNIT

Jens Krinke

Centre for Research on Evolution, Search & Testing
Software Systems Engineering Group
Department of Computer Science
University College London

Overview

- Faults, Failures and Errors
- Software Testing
- Unit Testing
- JUnit

Failure

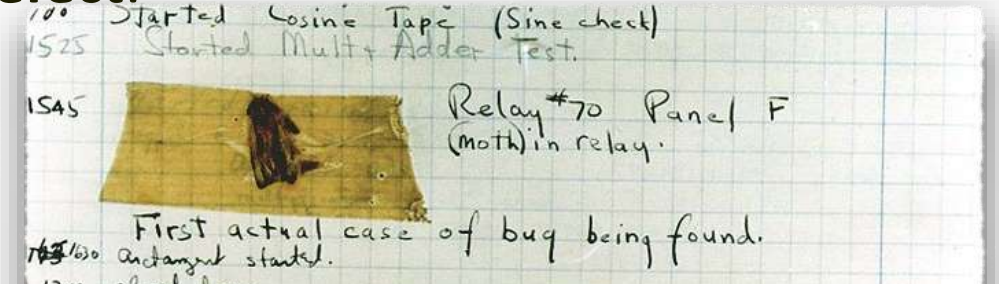
A **failure** is the inability of a system or component to perform its required functions within *specified* performance *requirements*.

- Observable incorrect behaviour.
- Deviation of the software from its expected delivery or service.

Fault

A **fault** is an incorrect step, process, or data definition
in a computer program
which causes the program to perform
in an unintended or unanticipated manner.

- A fault, if encountered, may cause a failure.
- Synonymous with **bug** or **defect**.



Error (1)

An **error** is a human action that produces an incorrect result.

- An error is the cause of a fault.
- Synonymous with **mistake**.

Error (2)

An **error** is a discrepancy between a computed, observed, or measured value or condition and
the true, specified, or correct value or condition.

- An error can lead to a system's failure
  (unless the system can deal with it).
- A fault is the cause of an error.
- A single error can have many possible root causes (faults).

Software Testing

- One aim of software testing is to turn faults into failures.

Software Testing

- An investigation conducted to provide stakeholders with information about the quality of the product or service under test.
- Observing the execution of a software system to validate whether it behaves as intended.
- The process of comparing the expected behaviour and the observed behaviour of the implemented system.

Unit Testing

- In unit testing, individual "**units**" of source code are tested.
- Unit tests are typically **automated** tests written and run by **software developers** to ensure that a "unit" behaves as intended.
- The goal of unit testing is to **isolate** each part of the program and show that the individual parts are correct.

Unit Test

- Unit tests should run automatically, such that undesired effects of changes can quickly be detected.
- Unit tests are realised in a testing framework for units.
- Unit tests are typically specified in the language that the unit is written in.

Unit test framework

- provides the required environment for the component,
- executes the individual services of the unit,
- compares the observed program state with the expected program state,
- reports any deviations from the expectations,
- and does all of this automatically.

Component Test vs Unit Test

Component Testing and Unit Testing
are usually used interchangeably.

The main difference is the scope:

- Unit Testing is more low-level,
  units are not subdivided into other components.
- Units can be any fragment of code,
  usually methods and classes
- Components are classes, packages,
  up to complete sub-systems.

Assertions

- The main tool is the comparison of the observed state with the expected state.
- An assertion expresses some expected property of the program state.
  - Upon execution, check whether the property holds.
  - If not, a failure is generated.
- C, C++, Java have an `assert(`*p*`)` statement

Example

Assume a class `Rational` for rational numbers:
`Rational(1,3)` creates the rational number 1/3.

Suppose you want to test the comparison of numbers under the following properties:

- Identity: 1/3 = 1/3?
- Representation: 2/6 = 1/3?
- Integers: 3/3 = 1?
- Non-equality: 1/3 ≠ 2/3?

A separate class **RationalAssert**

```
class RationalAssert {
  public static void main(String args[]) {
    assert new Rational(1, 3).equals(new Rational(1, 3));
    assert new Rational(2, 6).equals(new Rational(1, 3));
    assert new Rational(3, 3).equals(new Rational(1, 3));
    assert !new Rational(2, 3).equals(new Rational(1, 3));
  }
}
```

```
$ java -ea RationalAssert
Exception in thread "main" java.lang.AssertionError
     at RationalAssert.main(RationalAssert.java:5)
```

Executing unit tests

- Testing with standard assertions
  is not well suited for larger test tasks.
- A testing framework is needed
  that uses assertions to organise tests.
- The standard framework for Java is JUNIT
  by Kent Beck and Erich Gamma.

JUNIT test cases

- In JUNIT, tests are organised in test cases.
- Each test case is realised by its own class.
- JUNIT makes use of annotations.

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class RationalTest {
```

JUNIT tests

- Each test of the test case is realised by its own method which is annotated with `@Test`.
- The `assertTrue()` method is similar to `assert`
- instead of aborting execution, `assertTrue()` reports the failed assertion to JUNIT.

JUNIT test example

```
@Test
public void testEquality()
throws Rational.Illegal {
  assertEquals(new Rational(1, 3),
               new Rational(1, 3));
  assertEquals(new Rational(1, 3),
               new Rational(2, 6));
  assertEquals(new Rational(1, 1),
               new Rational(3, 3));
}
```

Assertions in JUNIT

- `fail(msg)`
- `assertTrue(msg, condition)`
- `assertFalse(msg, condition)`
- `assertNull(msg, object)`
- `assertNonNull(msg, object)`
- `assertSame(msg, object, object)`
- `assertNotSame(msg, object, object)`
- `assertEquals(msg, value, value)`
- `assertNotEquals(msg, value, value)`
- `assertEquals(msg, value, value, delta)`
- `assertNotEquals(msg, value, value, delta)`

Test cases with multiple tests

A test case (class) can hold an arbitrary number of tests (methods).

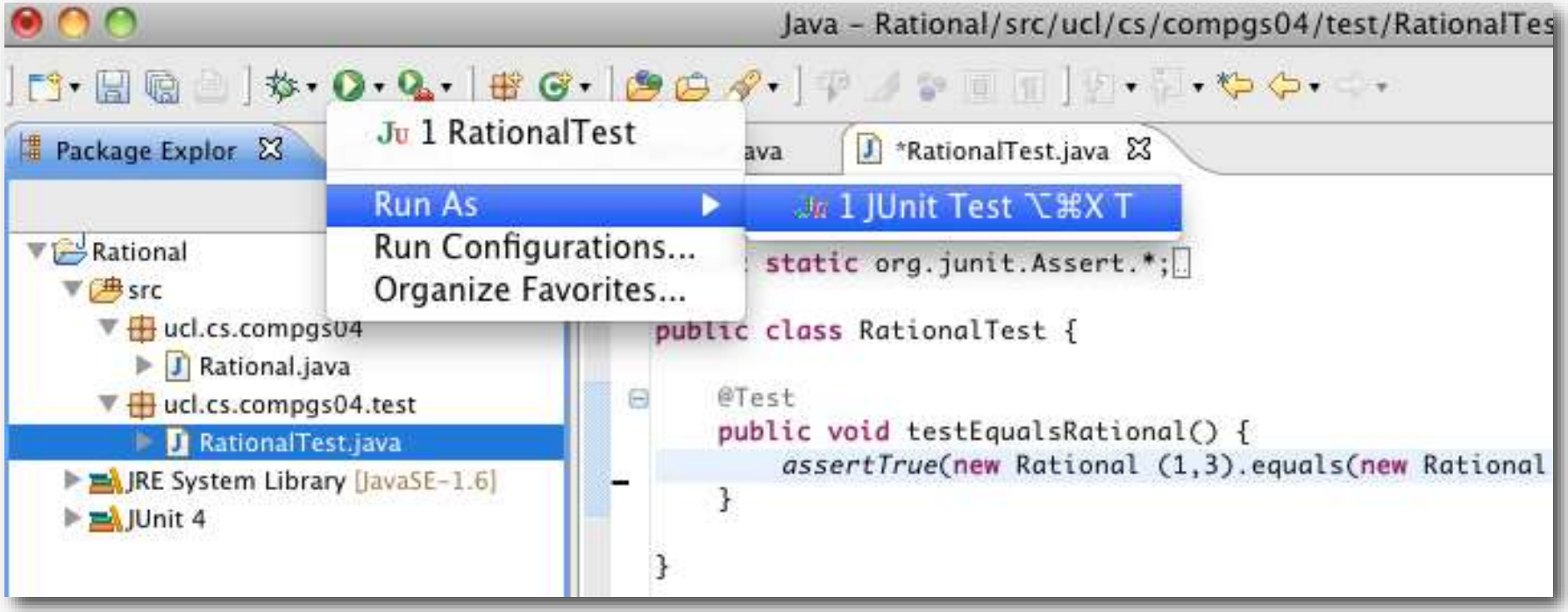
```
// Test for non-equality
@Test
public void testNonEquality() {
    assertFalse(new Rational(2, 3)
      .equals(new Rational(1, 3)));
}
```

Compiling and executing the test cases

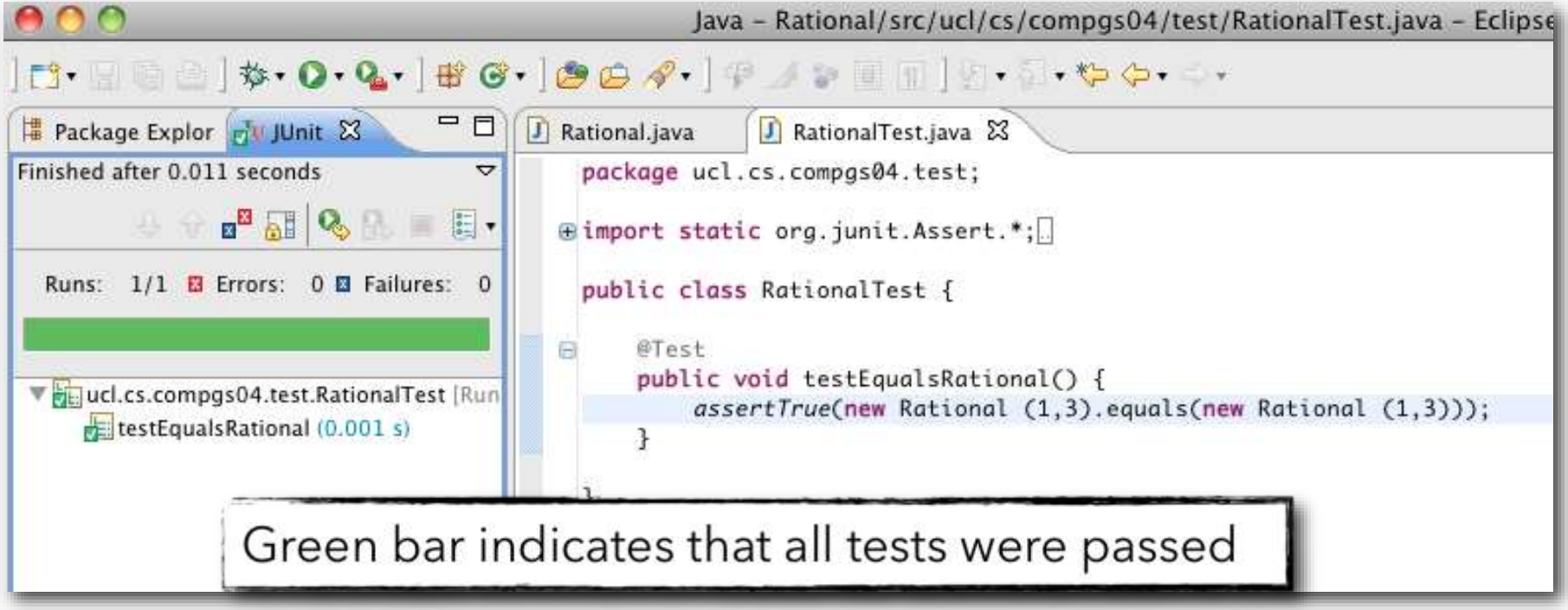
To compile the test cases,
`junit.jar` must be in the classpath.

```
$ J=/usr/share/java
$ javac -cp $J/junit.jar:. \
  RationalTest.java
$ java -cp $J/junit.jar:$J/hamcrest-core.jar:. \
      org.junit.runner.JUnitCore RationalTest
…

Time: 0.005

OK (2 tests)
```

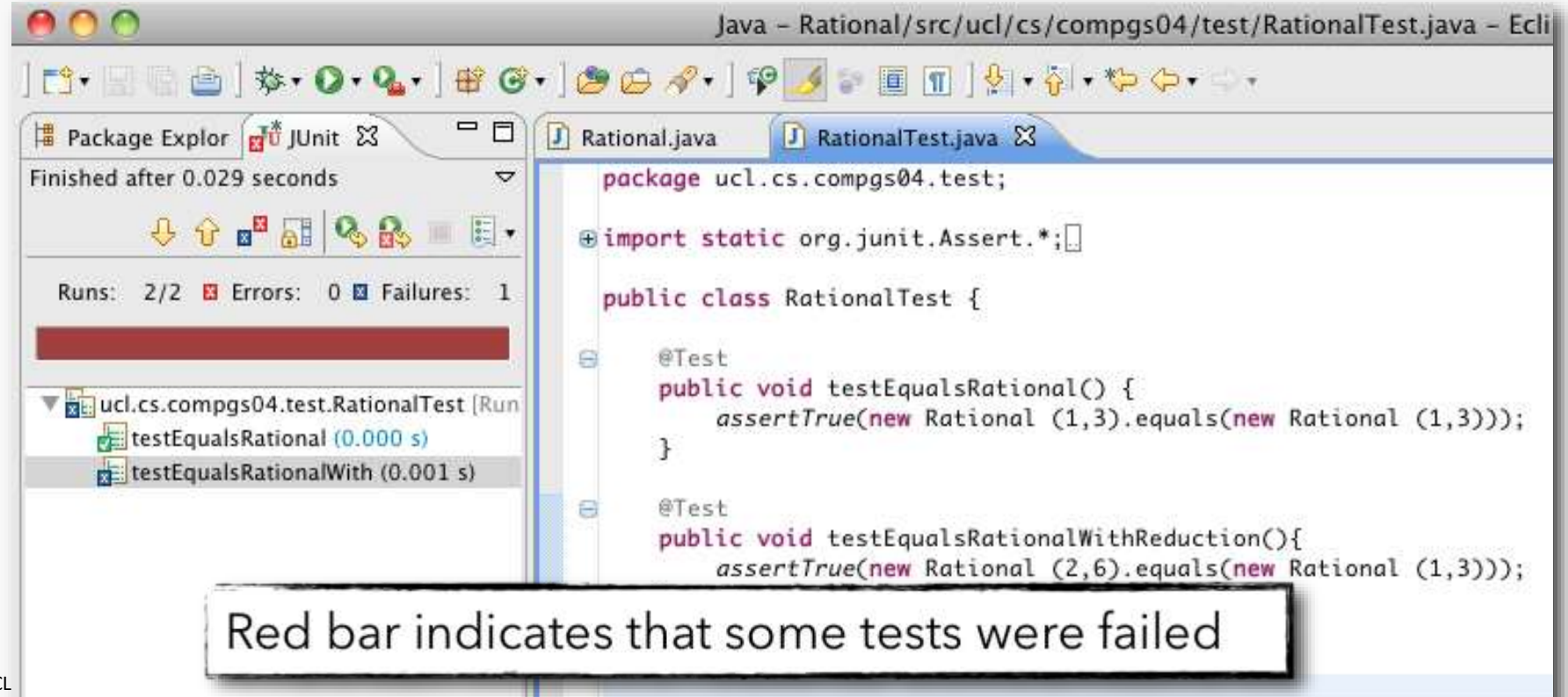
Run JUnit Test



JUnit View (Passing)



JUnit View (Failing)



Setting up a fixture

- Tests frequently need some fixture to execute.
- Typical examples include:
  - configuration files that must be read and processed,
  - external resources that must be requested and set up,
  - services of other units that must be initialised.
- JUNIT provides special annotations for methods that set up the fixture for the tests
  - `@Before` is called before each test of the class
  - `@After` is called after each test

Example fixture

```
public class RationalTest {

    private Rational a_third;

    @Before
    public void setUp() throws Rational.Illegal {
        a_third = new Rational(1, 3);
    }

    @After
    public void tearDown() {
        a_third = null;
    }
```

JUNIT test method

```
@Test
public void testEquality() throws Rational.Illegal
{
  assertEquals(a_third, new Rational(1, 3));
  assertEquals(a_third, new Rational(2, 6));
  assertEquals(new Rational(1, 1),
               new Rational(3, 3));
}
```

A test-driven development (TDD) process

- Component tests do not only facilitate regression testing of **existing** code.
- The test cases can also be used for testing code that has **not been written** yet.
- Test cases serve as **specification** of the component to be tested.
- Test cases can serve as **documentation** on typical uses of the component.

A test-first approach

- Before writing one line of production code, a test motivating that code is created.
- One writes just as much production code as required by the test.
- Development takes place in small steps, in which testing and coding alternate.
- Upon integration of the components into the complete system, all component tests must pass.

Advantages of a test-first approach

- The code can be tested automatically.
  This increases confidence into one's own work.
- Debugging is significantly easier.
  - Every new code is tested after 10 minutes at most.
  - If failures occur,
    first write a test case that reproduces the failure.
- The code is as simple as possible.
  - The code is no more complex than needed
    to satisfy the tests.
  - Of course, complex test cases result in complex code.

Test cases as specification are fun!

The multiple value of automated test cases

- first as specification,
- then as validation device,
- then as documentation

motivates developers to make use of them.

Concepts

- The programming language of unit tests
  usually is the language of the component.
- The most important language tools are assertions that
  express an expected state.
- In unit tests, methods represent tests
  and classes represent test cases;
  test suites group multiple tests.
- Component tests can serve as specifications that are written
  before the actual component (test first).