School of Computing
FACULTY OF ENGINEERING & PHYSICAL SCIENCES

UNIVERSITY OF LEEDS

COMP5911M
Advanced Software Engineering

4: Unit Testing to Support Refactoring

Nick Efford

https://comp5911m.info

Objectives

UNIVERSITY OF LEEDS

- To explore the idea that unit testing is the foundation of good software designs
- To consider how unit tests support the idea of design improvement through **refactoring**
- To examine two modern unit testing frameworks that can be used to test Java code

Unit Testing

UNIVERSITY OF LEEDS

- Testing of individual units of code, in isolation from the rest of the system
- Done by developer of the code being tested
- Just one element of a whole suite of testing techniques that are needed when developing software
  - Integration testing
  - User acceptance testing
  - Stress testing
  - Penetration testing
  - etc…

Why Write Unit Tests?

UNIVERSITY OF LEEDS

- To check that specifications have been followed
  - Tests should somehow encode those specifications in a form that is executable
- To check that the code ‘works as expected’ (which doesn’t necessarily mean it does what is required…)
- To increase confidence in the correctness of the code (NOT prove its correctness!)
- **To allow us to make future changes to the code and have more confidence that we won’t break things**

Refactoring

UNIVERSITY OF LEEDS

Refactoring (noun): A change made to the internal structure of software to make it easier to understand and cheaper to modify, without changing its observable behaviour.

Refactor (verb): To restructure software, via a series of refactorings, without changing its observable behaviour.

Refactoring

UNIVERSITY OF LEEDS

- Efficient, controlled method for ‘cleaning up code’
- Fundamentally different activity from adding features (Kent Beck’s ‘Two Hats’ analogy)
- Addresses **technical debt** and improves design
- Makes software easier to understand (for you and others)
- Makes finding bugs easier
- Helps you program faster, in the long term
- **Absolutely requires a good suite of unit tests**

JUnit 5

UNIVERSITY OF LEEDS

- Latest version of the most popular Java unit testing framework, based on the xUnit pattern
- Testing class is written for each class to be tested
- Testing class can contain
  - Methods tagged with `@Test` that perform the tests, making **assumptions** of preconditions and **assertions** about what the code does
  - Methods to create **test fixtures**, either once per set of tests or before every test
  - Methods to clean up after tests (if needed)

Writing Tests: The 'Four As'

UNIVERSITY OF LEEDS

- **Assume**: make **assumptions** about any preconditions needed for the test to be valid
- **Arrange**: create an instance of the class under test, configure it if necessary
- **Act**: execute the method(s) whose behaviour we are testing
- **Assert**: make **assertions** to check whether the desired postconditions have been achieved

Test logic should be as simple as possible…

Assumptions Examples

UNIVERSITY OF LEEDS

```
@Test
public void onlyOnCIServer() {
  assumeTrue("CI".equals(System.getenv("SYSTEM")),
    "Aborting: not on CI server");

  // remainder of test runs only if environment
  // variable SYSTEM has the value "CI"
  ...
}

@Test
public void onlyOnDevWorkstation() {
  assumeTrue("DEV".equals(System.getenv("SYSTEM")),
    "Aborting: not on dev workstation");
  ...
}
```

Standard JUnit Assertions

UNIVERSITY OF LEEDS

- `assertTrue`, `assertFalse`
- `assertEquals`, `assertNotEquals`
- `assertArrayEquals`
- `assertLinesMatch` (compares lists of strings)
- `assertSame`, `assertNotSame`
- `assertNull`, `assertNotNull`

```
Money money = new Money(1, 50);
assertEquals(1, money.getEuros());
```

expected value

value being tested

Hamcrest

UNIVERSITY OF LEEDS

- Hamcrest library provides `assertThat`, which replaces many of the standard JUnit assertions
- … along with various **matchers**:
  - `is`, `equalTo`, `sameInstance`, `not`
  - `closeTo`, `greaterThan`, `lessThan`
  - `equalToIgnoringCase`, `equalToIgnoringWhiteSpace`, `containsString`, `startsWith`, `endsWith`
  - `hasItem`, `hasItemInArray`, `hasKey`
- Many useful extensions exist, e.g. to handle dates, filesystem paths, JSON data, SQL query results, etc

Hamcrest Examples

UNIVERSITY OF LEEDS

```
assertThat(noon, equalTo(noon));
assertThat(noon, equalTo(new Time(12, 0, 0)));
assertThat(noon, not(equalTo(new Time(13, 0, 0))));

assertThat(noon.compareTo(noon), is(0));
assertThat(noon.compareTo(midnight), greaterThan(0));
assertThat(midnight.compareTo(noon), lessThan(0));

assertThat(Math.sqrt(2), closeTo(1.41421, 0.00001));

assertThat(greeting, startsWith("Hello"));

assertThat(value, isIn(list));
assertThat(list, hasItem(value));
assertThat(map, hasKey("name"));
```

more fluent than standard assertions!

Testing Exceptions

UNIVERSITY OF LEEDS

If you don't care about the message:

```
assertThrows(IllegalArgumentException.class,
  () -> new Money(-1, 50));
```

lambda expression

If you want to also check the message text:

```
Throwable exception = assertThrows(
  IllegalArgumentException.class,
  () -> new Money(-1, 50)
);

assertThat(exception.getMessage(), is("invalid euros"));
```

Grouping Assertions

UNIVERSITY OF LEEDS

```
@Test
public void stringConversion() {
  Money oneFifty = new Money(1, 50);
  Money oneFive = new Money(1, 5);
  assertAll(
    () -> assertThat(oneFifty.toString(), is("€1.50")),
    () -> assertThat(oneFive.toString(), is("€1.05"))
  );
}
```

- `assertAll` always runs each of the contained assertions (i.e., doesn't stop at the first failure)
- Test passes if all assertions are true, fails if any fail

Test Fixtures

UNIVERSITY OF LEEDS

```
private Money oneFifty;

@BeforeEach
public void setUp() {
  oneFifty = new Money(1, 50);
}

@Test
public void stringConversion() {
  Money oneFive = new Money(1, 5);
  assertAll(
    () -> assertThat(oneFifty.toString(), is("€1.50")),
    () -> assertThat(oneFive.toString(), is("€1.05"))
  );
}

...
```

can also use `@BeforeAll` for 'once per class' set-up

Running Tests: Command Line

UNIVERSITY OF LEEDS

You can use JUnit's `ConsoleLauncher` application, which displays test results in the terminal window:

```
│
├─ JUnit Jupiter ✔
│  └─ MoneyTest ✔
│     ├─ stringConversion() ✘ Multiple Failures (2 failures)
│     │
│     │      Expected: is "€1.50"
│     │           but: was "€ 1.50"
│     │
│     │      Expected: is "€1.05"
│     │           but: was "€ 1.05"
│     ├─ centsTooHigh() ✔
│     ├─ eurosTooLow() ✔
│     ├─ addWithoutCarry() ✔
│     ├─ centsTooLow() ✔
│     ├─ addOneCent() ✔
│     ├─ addOneEuro() ✔
│     ├─ addWithCarry() ✔
│     └─ creation() ✔
└─ JUnit Vintage ✔
```

Running Tests: Build Tool

UNIVERSITY OF LEEDS

Better approach is to use unit testing support provided by your build automation tool – e.g., in Gradle, add this to your build file:

```
dependencies {
  testImplementation(
    'org.junit.jupiter:junit-jupiter-api:5.8.2',
    'org.hamcrest:hamcrest-all:2.2'
  )
  testRuntimeOnly(
    'org.junit.jupiter:junit-jupiter-engine:5.8.2'
  )
}

test {
  useJUnitPlatform()
}
```

Gradle will download dependencies if needed and generate an HTML report containing test results

Class ase.money.MoneyTest

all > ase.money > MoneyTest

9	0	0	0.053s
tests	failures	ignored	duration

100%
successful

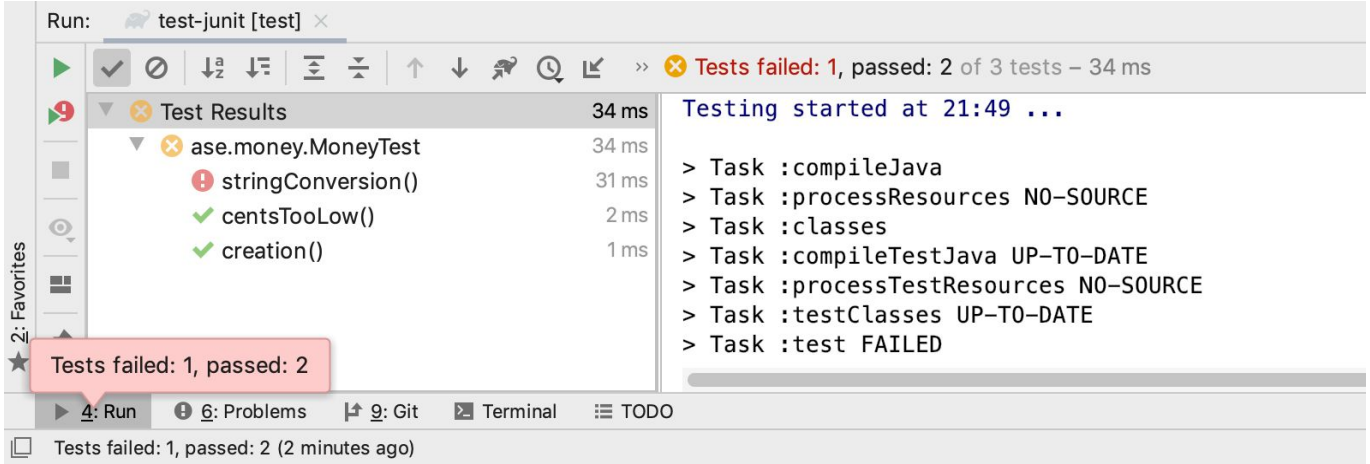
Tests

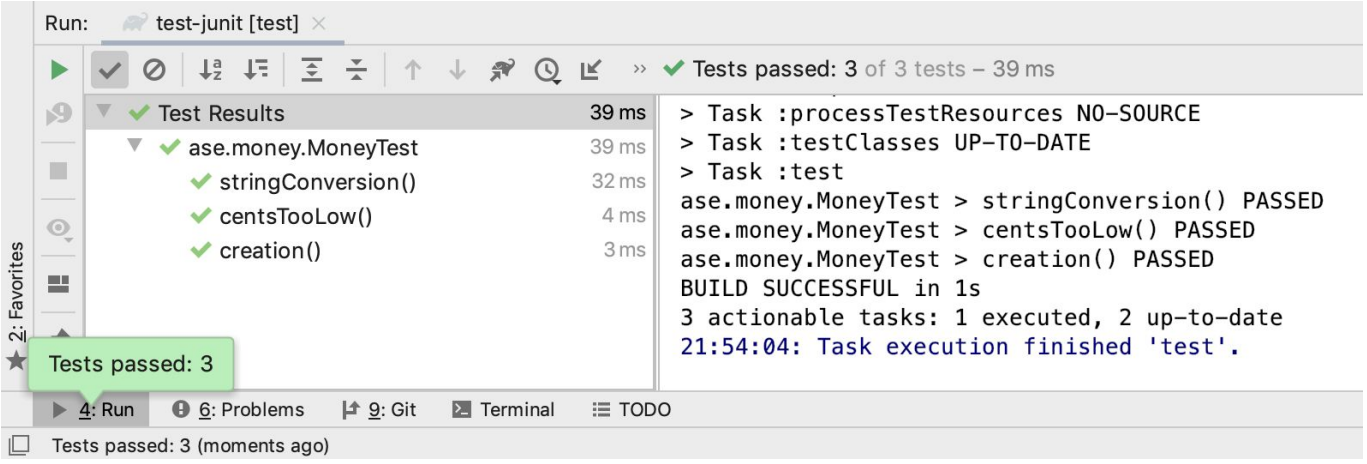
Test	Duration	Result
addOneCent()	0.002s	passed
addOneEuro()	0.003s	passed
addWithCarry()	0.002s	passed
addWithoutCarry()	0.002s	passed
centsTooHigh()	0.004s	passed
centsTooLow()	0.002s	passed
creation()	0.002s	passed
eurosTooLow()	0.002s	passed
stringConversion()	0.034s	passed

Example of a Gradle-generated HTML report

IDE Support

UNIVERSITY OF LEEDS





Spock

UNIVERSITY OF LEEDS

- Testing framework for Java & Groovy applications
  - Tests themselves are written in Groovy
- Provides a **domain-specific language** for unit testing
  - Tests can be simpler, easier to write & read
- Supports the idea of ‘data-driven testing’
  - Simplifies cases where same test code needs to be used for different sets of inputs and outputs

Example 1

UNIVERSITY OF LEEDS

```
class MoneyTest extends Specification {

    def oneFifty = new Money(1, 50)

    def "creating a Money"() {
        expect:
        oneFifty.getEuros() == 1
        oneFifty.getCents() == 50
    }
    ...
}
```

implicit test fixture; this object is created for every test

`expect:` block lists all of the conditions we expect to be true (assertions are implicit)

Example 2

UNIVERSITY OF LEEDS

```
class MoneyTest extends Specification {
    ...
    def "creating a Money with cents too low"() {
        when:
        new Money(1, -1)

        then:
        thrown(IllegalArgumentException)
    }
}
```

Paired `when:` and `then:` blocks provide a **stimulus-response model**:

WHEN the code below is executed *THEN* the following things are expected to happen…

Example 3

UNIVERSITY OF LEEDS

```
def "adding Money to Money"() {
    when:
    def result = oneFifty.plus(amount)

    then:
    result.getEuros() == euros
    result.getCents() == cents

    where:
    amount            || euros | cents
    new Money(1, 0)   || 2     | 50
    new Money(0, 1)   || 1     | 51
    new Money(1, 49)  || 2     | 99
    new Money(1, 50)  || 3     | 0
}
```

test code is run for all the cases in the data table

Summary

UNIVERSITY OF LEEDS

We have

- Discussed why good unit tests are an essential prerequisite for design improvement by refactoring
- Seen how to write JUnit 5 tests using the ‘Assume, Arrange, Act, Assert’ pattern
- Looked at how `assertThat` and Hamcrest matchers improve the readability of JUnit 5 tests
- Introduced Spock’s Groovy-based DSL as an alternative for writing unit tests

Follow-Up / Further Reading

UNIVERSITY OF LEEDS

- JUnit 5 User Guide
- Spock Primer
- Exercise 2