Unit Testing

PU Tools, Ressourcen, Infrastruktur

Nils Reiter,
nils.reiter@uni-koeln.de

January 28, 2021
(Winter term 2020/21)

Section 1

Recap

Exercise 10

`https://github.com/idh-cologne-tools-ressourcen-infra/exercise-10`

Section 2

Unit Testing

Introduction

- Automatic verification, that the code works properly
- Why?
  - Large, complex projects: Changes at one place might break something else
  - Humans make mistakes

Introduction

- Automatic verification, that the code works properly
- Why?
  - Large, complex projects: Changes at one place might break something else
  - Humans make mistakes
- Unit tests
  - Collection of expected output for a given input
    - E.g., for `add(2,3)`, we expect 5
  - Include edge cases
    - E.g., illegal input

JUnit

- Java library to support unit testing
- Uses code annotations and static methods
- Version 4 vs. 5
  - User guide: https://junit.org/junit5/docs/current/user-guide/

```
1  import static org.junit.jupiter.api.Assertions.assertEquals;
2
3  import org.junit.jupiter.api.Test;
4
5  class TestCalculator {
6    Calculator calculator = new Calculator();
7
8    @Test
9    void addition() {
10     assertEquals(2, calculator.add(1, 1));
11   }
12 }
```

Maven Integration: Adding the Dependency

```
<properties>
  <junit.version>5.7.0</junit.version>
</properties>
<!-- ... -->
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-api</artifactId>
  <version>${junit.version}</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <version>${junit.version}</version>
  <scope>test</scope>
</dependency>
```

Maven Integration: Code Location

- Test code in a maven project goes in `src/test/java`, resources used for testing in `src/test/resources`
- It's good practice to use the same package names within testing
- Prefix test classes with 'Test'

Maven Integration: Code Location

- Test code in a maven project goes in `src/test/java`, resources used for testing in `src/test/resources`
- It's good practice to use the same package names within testing
- Prefix test classes with 'Test'

Example (Classes and Packages)

`src/main/java`

- `com.example.package.MyClass`
  - `public int calculcateStuff(double x, double y)`
  - `public String generateString(Object o)`

`src/test/java`

- `com.example.package.TestMyClass`
  - `public void testCalculateStuff()`
  - `public void testGenerateString()`

Maven Integration: Code Location

- Test code in a maven project goes in `src/test/java`, resources used for testing in `src/test/resources`
- It's good practice to use the same package names within testing
- Prefix test classes with 'Test'

Example (Classes and Packages)

`src/main/java`

- `com.example.package.MyClass`
  - `public int calculcateStuff(double x, double y)`
  - `public String generateString(Object o)`

`src/test/java`

- `com.example.package.TestMyClass`
  - `public void testCalculateStuff()`
  - `public void testGenerateString()`

Example

Assertions

- Core of unit-testing
- Static methods of `org.junit.jupiter.api.Assertions` class

Assertions

- Core of unit-testing
- Static methods of `org.junit.jupiter.api.Assertions` class
  - `assertTrue(boolean b)` / `assertFalse(boolean b)`
  - `assertNull(Object o)` / `assertNotNull(Object o)`
  - `assertEquals(int expected, int actual)` / `assertEquals(Object expected, Object actual)` / `assertEquals(char expected, char actual)` / …

Assertions

- Core of unit-testing
- Static methods of `org.junit.jupiter.api.Assertions` class
  - `assertTrue(boolean b)` / `assertFalse(boolean b)`
  - `assertNull(Object o)` / `assertNotNull(Object o)`
  - `assertEquals(int expected, int actual)` / `assertEquals(Object expected, Object actual)` / `assertEquals(char expected, char actual)` / …
- Optional third argument
  - `assertEquals(int expected, int actual, String message)` – `message` is given if test fails

Setup

- One class can contain multiple test methods
  - Each method will be run individually
- If test setup is more than a single line, it's useful to encapsulate it in separate function
- Annotations
  - `@BeforeEach` – run before each test function
  - `@AfterEach` – run after each test function
  - `@BeforeAll` – run once before the first test function
  - `@AfterAll` – run once after the last test function

Best Practices

- Unit testing works best with well structured code
- Small functions for specific purposes
- Each function/method should be tested with multiple unit tests
  - Different inputs
  - Unexpected, but technically possible inputs (e.g., non-sensical strings from user input, null, negative values, closed file handles, ...)
    - It's also possible to test for thrown exceptions
- Don't test trivial functions (e.g. getters/setters)

demo

Section 3

Next Exercise

https://github.com/idh-cologne-tools-ressourcen-infra/exercise-11

Section 4

Appendix

References I