ADVANCED JAVA

Unit Testing Using JUnit

Vivek Shah bonii@di.ku.dk

August 20, 2018

DIKU, University of Copenhagen

SOFTWARE CORRECTNESS

- Correctness w.r.t. specification.

SOFTWARE CORRECTNESS

- Correctness w.r.t. specification.
- Formal proof.
    - Mechanized program verification.

SOFTWARE CORRECTNESS

- Correctness w.r.t. specification.
- Formal proof.
  - Mechanized program verification.
- Hoare logic.
  - Pre- and post-conditions.

SOFTWARE CORRECTNESS

- Correctness w.r.t. specification.
- Formal proof.
  - Mechanized program verification.
- Hoare logic.
  - Pre- and post-conditions.
- Software testing:
  - Functional testing / integration testing.
  - *Unit testing*.
  - White-box and black-box.
  - Smoke test.

UNIT TESTING

- Break down your program into *units*.
  - Isolate test cases $\sim$ isolate bugs.
  - Granularity:
    - Code blocks, methods, classes, packages, projects.

      White-box $\longrightarrow$ Black-box

UNIT TESTING

- Break down your program into *units*.
  - Isolate test cases $\sim$ isolate bugs.
  - Granularity:
    - Code blocks, methods, classes, packages, projects.
      White-box $\longrightarrow$ Black-box
- Partial testing:
  - Almost never feasible to test entire input domain.
    1. Cover only “interesting” cases.
    2. Randomly generate cases and check invariants.

UNIT TESTING

- Break down your program into *units*.
  - Isolate test cases $\sim$ isolate bugs.
  - Granularity:
    - Code blocks, methods, classes, packages, projects.
      White-box $\longrightarrow$ Black-box
- Partial testing:
  - Almost never feasible to test entire input domain.
    1. Cover only “interesting” cases.
    2. Randomly generate cases and check invariants.
- Manual (by hand) or automatic (by code).

UNIT TESTING

- Break down your program into *units*.
  - Isolate test cases $\sim$ isolate bugs.
  - Granularity:
    - Code blocks, methods, classes, packages, projects.
      White-box $\longrightarrow$ Black-box
- Partial testing:
  - Almost never feasible to test entire input domain.
    1. Cover only “interesting” cases.
    2. Randomly generate cases and check invariants.
- Manual (by hand) or automatic (by code).
- Integrated part of build system:
  - Quickly verify implementation changes (optimizations). ☺
  - Large-scale refactoring requires change in tests. ☹

DISCUSSION ON UNIT GRANULARITY

The choice of unit granularity is important. Briefly discuss the following:

1. Pros and cons of very fine-grained units.
2. Pros and cons of very coarse-grained units.
3. What are the properties of a good choice of granularity?

Unit granularity discussion points

Reflections on unit granularity follows:

1. Fine grained
   - ☺ Bug isolation.
   - ☺ Simple tests (easy to write).
   - ☹ Many units, many tests.
   - ☹ Simple tests (too trivial).
   - ☹ No integration testing.
2. Coarse grained
   - ☹ Little bug isolation.
   - ☹ Complicated tests (hard to cover all interesting cases).
   - ☺ Few units, few tests.
   - ☺ Complicated tests (catches many bugs).
   - ☺ Integration testing within units.

Unit granularity discussion points (contd.)

What is a good choice for granularity?

- When a test fails, it should be relatively easy to locate bug.
- A good tradeoff between the amount of interesting test cases in each test and the number of tests you have to write.
- Tests should be complicated enough that they can actually find bugs, but not so complicated that bugs might slip through unnoticed.

Test-driven development

- Write unit tests *before* implementation.

Test-driven development

- Write unit tests *before* implementation.
- Unit tests $\sim$ (partial) formalization of specification.

Test-driven development

- Write unit tests *before* implementation.
- Unit tests $\sim$ (partial) formalization of specification.
- Works sometimes:
  1. Think about interface.
  2. Write tests.
  3. Implement $\rightarrow$ gain experience (interface sucks).
  4. Goto 1.

Test-driven development

- Write unit tests *before* implementation.
- Unit tests $\sim$ (partial) formalization of specification.
- Works sometimes:
  1. Think about interface.
  2. Write tests.
  3. Implement $\rightarrow$ gain experience (interface sucks).
  4. Goto 1.

  - Pre-condition: Good, a priori interface design.

Brief detour: Reflection and annotations

- *Reflection*
  - Observe and manipulate runtime behavior at runtime.
  - `java.lang.reflect`.
    - *Exercise on Wednesday*.

Brief detour: reflection and annotations

- *Reflection*
  - Observe and manipulate runtime behavior at runtime.
  - `java.lang.reflect`.
    - *Exercise on Wednesday*.
  - Example: Run all methods with name prefix `test`.
    - Automatic test harness.

Brief detour: reflection and annotations

- *Reflection*
  - Observe and manipulate runtime behavior at runtime.
  - `java.lang.reflect`.
    - *Exercise on Wednesday*.
  - Example: Run all methods with name prefix `test`.
    - Automatic test harness.
    - Prone to error:
      `tetsFoo()` never runs (fails silently). ☹
      `testosterone()` runs. ☹

Brief detour: Reflection and annotations

- *Reflection*
  - Observe and manipulate runtime behavior at runtime.
  - `java.lang.reflect`.
    - *Exercise on Wednesday*.
  - Example: Run all methods with name prefix `test`.
    - Automatic test harness.
    - Prone to error:
      `tetsFoo()` never runs (fails silently). ☹
      `testosterone()` runs. ☹
- *Annotations*
  - Structured meta information, verified by compiler.
  - Builtins: `@Override`, `@SupressWarnings`, `@Deprecated`
  - Make your own, define where it can be used, define fields.
  - Annotations visible through reflection.
    - Example: Run all methods with `@Test` annotation.

JUNIT 4

- De facto unit testing for Java. Shiniest version is JUnit 5, not being taught this year (needs some stability).
- Unit tests of Java programs written in Java.
- Unit granularity: Somewhere between methods or classes.
- Based on annotations and reflection:

```java
@Test
public void testFoo() {
    Bar bar = new Bar();
    assertEquals(5, bar.foo(10,2));
}
```

JUnit 4

- De facto unit testing for Java. Shiniest version is JUnit 5, not being taught this year (needs some stability).
- Unit tests of Java programs written in Java.
- Unit granularity: Somewhere between methods or classes.
- Based on annotations and reflection:

```
@Test
public void testFoo() {
    Bar bar = new Bar();
    assertEquals(5, bar.foo(10,2));
}
```

- Eclipse integration:
  - Automatic test-code generation.
  - Automatic build integration.
  - Reports.

USEFUL JUNIT4 ANNOTATIONS

@Test	Test method
@Test (expected=Exception.class)	Method should throw Exception, if exception is not thrown test fails
@Before	Method runs before *each* test
@After	Method runs after *each* test
@BeforeClass	Method runs once before any test
@AfterClass	Method runs once after all test have finished

JUnit4 Assertions

```
import static org.junit.Assert.*;
```

JUnit4 Assertions

import static org.junit.Assert.*;

assertTrue(boolean cond)	Asserts that condition is true
assertFalse(boolean cond)	Asserts that condition is false
assertEquals	
(T expected, T actual)	Different variants of method for different Ts. Asserts that two values are equal.

JUnit4 Assertions

import static org.junit.Assert.*;

assertTrue(boolean cond)	Asserts that condition is true
assertFalse(boolean cond)	Asserts that condition is false
assertEquals (T expected, T actual)	Different variants of method for different Ts. Asserts that two values are equal.

Find out more in JUnit 4 documentation:
http://junit.org/junit4/

- Testing private methods - 3 choices:
  1. Put test method inside tested class.
  2. Change private to protected and inherit tested class.
  3. Use the reflection API.

ADVANCED STUFF

- Testing private methods - 3 choices:
  1. Put test method inside tested class.
  2. Change private to protected and inherit tested class.
  3. Use the reflection API.
- Use Hamcrest matchers
  1. assertThat(0, is(equalTo(someObject.getZero())))
  2. assertThat("Blah", is(allOf(instanceOf(String.class), notNullValue())))
  3. http://hamcrest.org/JavaHamcrest/

- Testing private methods - 3 choices:
  1. Put test method inside tested class.
  2. Change private to protected and inherit tested class.
  3. Use the reflection API.
- Use Hamcrest matchers
  1. assertThat(0, is(equalTo(someObject.getZero())))
  2. assertThat("Blah", is(allOf(instanceOf(String.class), notNullValue())))
  3. `http://hamcrest.org/JavaHamcrest/`
- *Parametrized* tests and *theories*.
  - More fun with annotations and reflection.
  - Assume we have:
    1. Data set *D* of data points.
    2. Collection of tests *T* taking a data point as input.

- Testing private methods - 3 choices:
  1. Put test method inside tested class.
  2. Change private to protected and inherit tested class.
  3. Use the reflection API.
- Use Hamcrest matchers
  1. assertThat(0, is(equalTo(someObject.getZero())))
  2. assertThat("Blah", is(allOf(instanceOf(String.class), notNullValue())))
  3. http://hamcrest.org/JavaHamcrest/
- *Parametrized* tests and *theories*.
  - More fun with annotations and reflection.
  - Assume we have:
    1. Data set D of data points.
    2. Collection of tests T taking a data point as input.
  - Automatically generate all combination of tests $T \times D$.

- Testing private methods - 3 choices:
  1. Put test method inside tested class.
  2. Change private to protected and inherit tested class.
  3. Use the reflection API.
- Use Hamcrest matchers
  1. assertThat(0, is(equalTo(someObject.getZero())))
  2. assertThat("Blah", is(allOf(instanceOf(String.class), notNullValue())))
  3. `http://hamcrest.org/JavaHamcrest/`
- *Parametrized* tests and *theories*.
  - More fun with annotations and reflection.
  - Assume we have:
    1. Data set D of data points.
    2. Collection of tests T taking a data point as input.
  - Automatically generate all combination of tests $T \times D$.
  - Separation of test code and data points.
    - Easy to add new data points.