Testing

Technion – Institute of Technology

236700
Why (unit) test?

• We want to prove (e.g., to a reviewer) that our code succeeds in all scenarios
  • Rule of thumb: No test? Doesn’t work!
• Tests are the usage example of our code and are an excellent source for documentation
• Programming is incremental by nature
  • We want to verify we haven’t broken anything
  • We can’t safely refactor without tests
• Testable code is (arguably) better designed code
  • We will talk about this later in the course
Test types

- **Unit Tests** – Test a single element of the system
- **Integration Tests** – Verify that a few units can work together and are wired up correctly
- **End-to-End Tests** – Test the complete flow of interactions throughout the system
  - No “mocks”/”stubs”/”fakes”
- **Acceptance / Business logic Tests** – Answers the question: “Did we build what we wanted?”
  - Usually centered around a user story
  - Described using the domain language
Test types (Cont.)

JUnit

- A unit testing framework for Java
- A **Test** is a Method
- A **Test Case** is a class which includes tests methods
  - Aggregates tests for a single tested unit
  - Generally speaking, 1 Test Case per class
  - Uses the same name as the class being tested + "Test"
- A **Test Suite** is a collection of Test Cases / Test Suites (or both)
  - Used to create a hierarchy of tests
- The de-facto testing framework for Java
- Part the **JUnit** family
  - PyUnit, HUnit, CUnit, etc.
JUnit 4.0 annotations

- Test methods are annotated with `@Test`
- `@BeforeClass` static method – runs once before the first test
  - e.g., opening DB Connections
- `@AfterClass` static method – runs once after the last test
  - e.g., closing DB connections
- `@Before` method – runs before every test
  - Preparing data
  - Could be used for advanced construction
- `@After` method – runs after every test
  - Resetting stubs / cleaning data
JUnit test case life-cycle

- Test Case Class load
  - Static C’tor
    - @BeforeClass*
      - C’tor
        - @Before*
          - @Test
            - @After*
  - @AfterClass*
JUnit life-cycle - Example

- Local **resources** should be initialized and cleaned before/after every test

```java
public class FileTests {
    private File output = File.createTempFile();
    @Before public void setUp() {
        output.write("Hello world");
    }
    @Test public void testSomething() { ... }
    @After public void tearDown() {
        output.delete();
    }
}
```

- Fields initialized this way cannot be **final** 😞
- Always prefer to use inline initialization if possible
JUnit life-cycle - Example

- Expensive and **global** resources should be initialized and destroyed before/after all tests

```java
public class DbTests {
    private static DatabaseConnection database;
    @BeforeClass public static void setupClass() {
        database = new DatabaseConnection();
        database.establishConnection();
    }
    @Test public void testSomething() { ... }
    @AfterClass public static void tearDownClass() {
        database.logout();
    }
}
```
Test naming

It should be easy pin-pointing the problem when a test fails without looking at its code

- Consider the following test:

```java
@Test public void addItemTest() {
    Product p = new Product();
    p.setAvailable(false);
    ShoppingCart cart = new ShoppingCart();
    cart.addItem(p);
    assertEquals(0, p.numItems());
}
```

- **How about addItemWithOutOfStockProduct?**
  - Describes the method and the parameter
  - But does not describes the desired behavior

- **A good name:** doNotCountOutOfStockItem
  - It should be trivial understanding what is the expected behavior from the name alone
  - the test name should include an explicit expectation
  - Use **verbs** such as should, does, returns, throws, etc.
A recipe for a good test:

```java
@Test
public void testShouldNameExplicitlyStatesExpectation() {
    (1) - Initialize collaborators (optional)
    (2) - Initialize object under test (optional)
    // one line of spacing
    (3) - Exercise behavior / get return value
    // one line of spacing
    (4) - Assert outputs
}
```

- If (1) and (2) are needed at all (e.g., can't be initialized in constructor/@Before), it will be as short as possible (ideally just a few calls to `new`)
- (3) will be a single method invocation
- (4) will be a single assert
- Of course this ideal cannot always be achieved
- But if you can’t, you should always ask yourselves why?
Exceptions

- The hard way – Using the `try/catch` pattern

```java
@Test
public void badEncodeArgument() {  
    List<Integer> list = new ArrayList<>();  
    try {  
        list.get(0);   
        fail("Exception should have been thrown");  
    } catch (IndexOutOfBoundsException e) {  
        assertEquals("Index 0 was out of bounds", e.getMessage());  
    }
}
```

- ☺ Checks for everything
- 😞 Easy to forget the `fail` resulting in a test that **never fails**
- 😞 Quite long and verbose
  - This is exemplary **imperative** code
  - For such a common case, there should always be a declarative alternative...
  - ...even if we have to write one ourselves (but we usually don't)
Exceptions – expected Exception

• The easy way (JUnit 4.0) – expected value in @Test annotation

```java
@Test(expected = IndexOutOfBoundsException.class)
public void badEncodeArgument() {
    List<Integer> list = new ArrayList<>();
    list.get(0);
}
```

• 😞 Can’t check the exception message
• 😞 Can’t verify which line threw the exception
• 😊 Very short
• Is “good enough” in most cases
Exceptions – Using @Rule

• The smart way – Using @Rule annotation and ExpectedException

```java
@Rule public ExpectedException thrown = ExceptionException.none();

@Test
google void badEncodeArgument() {
    List<Integer> list = new ArrayList<>();
    thrown.expect(IndexOutOfBoundsException.class);
    thrown.expectMessage("Index 0 was out of bounds");
    list.get(0);
}
```

• ☺ Can check for everything
• ☹ Still quite short
• Always make sure you initialize the thrown object before the last line
• JUnit Rules offers a few additional candies
JUnit – Timeout

• In JUnit – Use the timeout annotation value on a method

```java
@Test(timeout = 10)
public void testTimeout() {
    // Test will fail after 10 milliseconds.
}
```

• Global timeouts using @Rule (JUnit 4.7)

```java
@Rule public MethodRule globalTimeout = new Timeout(10);
```

• Both tests use InterruptedException
  • If your code is stuck in an infinite loop, it will not fail!
  • See https://github.com/junit-team/junit4/wiki/timeout-for-tests
Leveraging OOP

- Tests are classes too!
- Can use **inheritance** to our advantage
- Use abstract test classes against abstract classes / interfaces
- Hierarchy parallels the implementation's hierarchy
public abstract class Stack<T> {
    public abstract void push(T element);
    public abstract int size();
}

public class ArrayStack<T> extends Stack<T> {
    // … implementation code
}

public class LinkedListStack<T> extends Stack<T> {
    // … implementation code
}
OOP – Base class

```java
public abstract class StackTest {
    // this is called a factory method
    protected abstract <T> Stack<T> createStack();

    @Test
    public void sizeIsOneAfterOnePush() {
        Stack<Object> stack = createStack();
        stack.push(new Object());
        assertEquals(1, stack.size());
    }
}
```

- All test methods of the parent class are run for each concrete test class
- If A extends B, then A “is-a” B, and should therefore pass all of B’s tests!
Testing – Derived classes

• Extend abstract test class to and override the factory method to initialize the concrete test subjects

• ArrayStack:

```java
public class ArrayStackTest extends StackTest {
    @Override protected <T> ArrayStack<T> createStack() {
        return new ArrayStack<>();
    }
    // specific ArrayStack tests
}
```

• LinkedListStack:

```java
public class LinkedListStackTest extends StackTest {
    @Override protected <T> LinkedListStack<T> createStack() {
        return new LinkedListStack<>();
    }
    // specific LinkedListStack tests
}
```
Testing & Debugging

• When you work on a bug, write a failing test **first**
  • Your work is **done** when the test **succeeds**
  • If you write the test after the bug is fixed, you can’t know if the test succeeded in reproducing the bug
  • Helps with **regression** testing, i.e., ensuring the bug won’t reappear later in the development lifetime

• You can actually **debug** using unit tests!
  1. Write a test that fails
  2. If you still aren’t sure what the bug is, write a **smaller** unit test that fails
  3. Repeat step 2 as necessary
  • This could be preferable to running a debugger, as the tests remain when done
Testing private methods

• Don't!
  • Private methods are private for a reason
  • They're not part of the class's contract, so why should you test them?
• But I have a really important method that I need to know works!
  • Then you should probably extract its behavior to an external class, improve cohesion in the process
  • If it's important enough to test, it's important enough to be extracted and made public somewhere
• I can't, now what?
  • Fine, make the method package-private (no visibility modifier)
  • If at all possible, make the method static
  • But you should be asking yourself 'why?'
• Never expose private fields for sake of testing!
  • Completely breaks encapsulation
  • Yes, even if they're final
Declarative versus Imperative tests

- Our most basic building block is `assert/assertTrue`.
- `assertFalse(b)` is more declarative than `assertTrue(!b)`.
- What about `assertTrue(a.equals(b))`?
  - If it fails, the error message will be "false was not equal to true"
  - We can add a String description to each assertion, but that doesn't scale.
- Use `assertEquals(a, b)`.
  - More declarative, better error message.
- Other useful methods: `assertNull`, `assertNotNull`, `assertArrayEquals`.
- What about `assertEquals(list.size(), 2)`?
  - "3 was not equal to 2" isn't very informative.
  - What about empty lists? Containing a specific element? Sublists?
D v. I: Hamcrest

- Hamcrest is a (DSL) testing library

```java
import static org.hamcrest.Matchers.*;
// Make sure you import this version and not junit's
import static org.hamcrest.MatcherAssert.asserThat;
public class MyTest {
    @Test public TestExamples() {
        assertThat(1, is(lessThan(2)));
        assertThat("foobar", startsWith("foo"));
        assertThat(new ArrayList<String>(), is(empty()));
        assertThat(Arrays.asList(1, 2, 3), contains(1));
    }
}
```

- And many more!
- More type safe than assertEquals, and much more expressive
  - The is method is an example of a purely syntactic construct
  - It's never needed, but it adds expressiveness
D v. I: Custom Hamcrest matchers

- We can also add custom matchers to tests our classes, or add missing tests

```java
public class Even extends TypeSafeDiagnosingMatcher<Integer> {
    @Override protected boolean matchesSafely(
        Integer integer, Description description) {
        description.appendText("was ").appendValue(integer)
            .appendText(" , which is an Odd number");
        return integer % 2 == 0;
    }

    @Override public void describeTo(Description description) {
        description.appendText("An Even number");
    }

    public static Even even() { return new Even(); }
}
```

- Now we can write `assertThat(2, is(even()))`
- A more complete example
Cons to testing?

- **Over-testing** can be a "waste of time"
  - Depending on your new feature / refactoring ratio
- **Bad** tests can **impede** code writing
  - Now you have to maintain **two** modules: the actual code, and its tests
  - Smells to watch out for: repeating the implementing algorithm, or tests that break if the **implementation** (but not the **contract**) changes.
  - Prefer **black box** testing to **white box** testing whenever possible
    - Input in, output out; don't check the internal condition or side effects
- Test writing can be **hard**
  - That's usually a sign you have problems in design
  - **Hard to test? Re-think the design!**
  - Consider investing some time in building a good testing infrastructure (e.g., using Hamcrest)
Appendix

Other nice to know features
Testing – Order (or lack thereof)

• Do not assume order between tests
  • There’s no order guaranties in TestCases
  • There is one in TestSuites

• Avoid side effects in tests!

```java
public class StackTest {
    private static final Stack<Object> s = new Stack<>();

    @Test
    public void pushFirstAndCheckSize() {
        s.push(new Object());
        assertEquals(1, s.size());
    }

    @Test
    public void pushSecondAndCheckSize() {
        s.push(new Object());
        assertEquals(2, s.size());
    }
}
```
Ignoring tests

- You can ignore tests using the `@Ignore` annotation
  - Test will not run, but will still be visible in the test reports
- Commented out tests will usually be forgotten about
  - Won’t be deleted and possibly will not be fixed
- Ignored tests are easily visible in the output window

- Make sure you disable tests temporarily
  - Bad tests should either be deleted or fixed
  - Ask yourself – why are you ignoring the test in the first place?
Testing – Resource Management

- Don't load data from hard-coded locations in file systems

```java
@Before
public void setUp () {
    InputStream inp = new FileInputStream("C:\TestData\dataSet1.dat");
    ...
}
```

- **Use** `Class.getResource()` or `Class.getResourceAsStream()`

```java
@Before
public void setUp () {
    InputStream inp = getClass().getResourceAsStream("dataSet1.dat");
    ...
}
```

- Will search the file relatively to the Test Case .class file
- Maven will automatically create a src & test resource folders
Appendix: Source code organization

- **Option 1**: Test class in same folder as subject
  - A single hierarchy of folders
  - Locality

- **Option 2** (much more common): Keep tests in the mirrored directory structure of source location
  - A duplicated hierarchy of folders
  - Easy to find all tests
  - Easy to separate source code from its tests
  - Easier bytecode separation

Author: Gal Lalouche - Technion 2017 ©
Code Coverage - EclEmma
Code Coverage

• Code coverage is correlated with good testing, but does not equate it
  • You can achieve very high coverage with bad tests (e.g., testing without asserts)
  • You can write great tests with low coverage
• 100% coverage may look “pretty”, but should not be a goal in and off itself
  • You should always suspect tests that were written solely for achieving high coverage
  • Do you really need to test for NullPointerException for each of your parameters?
Other test types

• Behavior-driven development
  • Using **Domain Specific Language** (DSL) to structure tests around stories and features

• Property tests
  • Large **randomized** input to check **invariants** and properties of output, rather than **hard coded** inputs and outputs

• Mutation testing
  • Changing (**mutating**) the source code or tests slightly, to see if our tests fail
  • This ensures **tight coupling** between tests and source code

• Design by Contract
  • Asserting **pre and post-conditions** on class’s methods
  • Fails in **runtime**, possible to check even at compile time