Testing

Gal Lalouche
Oath Inc.

Computer Science
Technion

Spring 2018
Why do we write tests?
Why do we write tests?

- Common answer: to make sure our code works
- But writing automated tests is a lot more work than just writing a main method and checking some inputs and outputs
Remember our primary assumption: software changes

- But when software changes, tests must change with them...
- If we change the structure of our classes, we would have to adapt the tests, wasting time
- If new features are added, they obviously won’t have tests
- If bugs are fixed, they obviously didn’t have a test to begin with
But not all changes are functional

Refactoring

The process of restructuring existing source-code without changing its external behavior

Refactorings catalog

Reasons for refactoring:

- Fixing code smells
- Paying for technical debt
- Improving performance parameters
- Updating to new versions of an API
Refactoring (cont.)

Non-trivial functional changes will probably also include some refactoring around the actual change

- If you can, split your change into two:
  1. Refactor, i.e., non-functional change, preparing the code base for the actual change
  2. Add the required functionality
Refactoring (cont.)

- Should be a day-to-day activity
  - No need for a special occasion
  - Although some teams dedicate a few weeks a year to doing only that
    - Not recommended
    - Remember, we prefer our Δs to be small
    - Think of all the time you’re wasting working with a less-than-optimal design
  - The boy scout rule

- Most simple refactoring operations can be safely automated by the IDE
  - Formatting (spaces, line length, javadocs, etc.)
  - Renaming
  - Extracting stuff
  - Many, many others
Non-safe refactorings

But not all refactorings are safe
- Inlining (side-effects)
- Changing dependencies
- Changing algorithms and data structures
- Pretty much any non-trivial change

To refactor safely, we have to write tests
- We write tests not to make sure our code works
- But to make sure that our code *still* works
Use of tests

If you don’t write tests, you’re creating **legacy code**

But the relationship isn’t just linear, it’s **cyclic**!
Refactoring cycle: bad

- Bad code
- Hard to test
- Can’t refactor
- Low test coverage
Refactoring cycle: good

- Clean code
- Easy to test
- Lots of test
- Can refactor with confidence
Other reasons to test

- Proof (e.g., to a reviewer) that our code works
- Tests are the usage example of our code and are the canonical source for documentation
  - Comments and javadoc can become outdated, but passed tests are always up to date
  - Want to know how the code behaves on some edge case? Check the test
- Testable good is often better code
  - More on that when we talk about testability and Test-driven design
- For outward facing API, tests may be the only code that activates your implementation
  - Checks your usage examples compile
Test levels (cont.)

- **Unit tests.** Test a single element of the system (Our focus in this course)

- **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 (more on this next lecture)

- **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 levels (cont.)
What’s a unit?

So what size should the “unit” be in unit tests?

- A single class (most often)
- Several related classes
- An entire package
  - Usually only the (single) public API
  - Internal APIs are tested implicitly
Our tests should be:
- Fast
- Coherent
- Modular
- Implementation independent
Testing requirements: Fast

- It’s not uncommon to have hundreds of tests for even small projects, tens of thousands for medium-sized
- Module-tests are run very often
  - Often multiple times a minute!
  - The more often we run the test the earlier we can detect bugs
  - The earlier we detect bugs, the easier they are to fix
- The faster they are, the easier they are to incorporate into your development cycle
  - Generally, 10 milliseconds per test is fine, above 100 is too much
  - You don’t want to be in a position where you’re avoiding writing new tests because of a slow cycle
- Ways to avoid slow tests:
  - Small input
    - How small? As small as possible to get good coverage
  - Avoid IO (disk access; if your tests are doing network access you’re doing something very wrong)
Test readability is just as important as production code readability
- The tests are probably the first thing you’ll look at when a test fails
- Tests usually outlive production code
- As mentioned, tests can be used as documentation

Luckily, tests are much easier to write since they’re very structured
Testing requirements: Coherent (cont.)

Three A’s of testing:

1. Arrange
2. Act
3. Assert

```java
@Test
public void test() {
    String input = "abc";

    String result = Util.reverse(input);

    assertEquals("cba", result);
}
```
Testing requirements: Modular

- Ideally, the “unit” being tested should be as small as possible
  - And tests even smaller
  - Prefer many short tests to a single large test
  - Rule of thumb: one assertion per test
    - could be multiple when asserting data properties (leads to better errors)
- You usually don’t want to fail due to a bug in your dependencies
  - Too noisy, hard to ascertain root cause of bug
- Too many (real) dependencies can cause test slowness
  - Especially if they do IO
- We’ll talk about this next lecture when we discuss test-doubles
Testing requirements: Implementation independent

- Tests check the output of your code against some input
- You don’t want your tests to change if your implementation changes, only if the external functionality does
- Avoid tests that simply repeat your production code
  - Remember, we’re writing tests (mainly) to facilitate easy refactoring
  - Tests impeding that are counter-productive by definition
- Prefer black-box to white-box testing whenever possible
  - Unfortunately, it’s not always possible
**Black and white box testing**

**White-box testing**
Testing aware of internal implementation details of the class

**Black-box**
Testing the interface of the class only

- Technically, we’re almost always writing white-box testing, because we wrote the production code
- But that doesn’t mean we should *use* our knowledge
It’s not always possible to do black-box testing, as white-box testing is sometimes needed to:

- Replacing dependencies
  - For example, ones that do IO or network access
  - Have nasty side-effects
- Forcing failures
  - To see how your system reacts
- Checking crucial side-effects
The term white-box testing is overloaded

1. Exposing internal details for the sake of testing (Bad)
   - For example, private fields, private methods, knowledge of IO operations

2. Using your knowledge of the class to write the tests (Good)
   - For example, edge cases, branches, regression testing
   - Although random tests can take care of many of these without explicit developer input

The first binds our tests to our implementation, making changes harder.
The latter guides us in what tests we should write, but still test the interface and not the implementation.
The most important question you should ask is: “Will this test make sense if the implementation change”?
   - If not, will this test still pass?
   - Will this test even run?
What to test?

What should we test? A long spectrum between two extremes

1. NOTHING!
   - Tests take too long to write
   - They slow us down when we have to change our code
   - You shouldn’t write bugs to begin with
   - My code doesn’t have bugs anyway

2. EVERYTHING!
   - Without tests, every single change can wreak havoc on your program
   - 100% test coverage is required, but not sufficient
   - If a single element of functionality is changed, a test should fail

The truth probably lies somewhere in the middle
One common pitfall to watch out for is test code that just repeats the production code

```java
public int add(int x, int y) {
    return x + y;
}
```

```java
@Test
public void testAdd() {
    int x = 3;
    int y = 5;

    int result = add(x, y);
    assertEquals(x + y, result);
}
```

What does this achieve?
So should we just avoid testing trivial methods? Well, not necessarily, but we should test them differently.

```java
// Super simple code, no need for test!!!
ComparisonResult compareWithEnum<T extends Comparable<T>>(
    T t1, T t2) {
    int result = t1.compareTo(t2);
    if (result < -1)
        return ComparisonResult.LEFT_IS_BIGGER;
    if (result == 0)
        return ComparisonResult.EQUAL;
    return ComparisonResult.RIGHT_IS_BIGGER;
}
```
So should we just avoid testing trivial methods?
Well, not necessarily, But we should test them differently

```java
// Super simple code, no need for test!!!
ComparisonResult compareWithEnum<T extends Comparable<T>>(
    T t1, T t2) {
    int result = t1.compareTo(t2);
    if (result < -1)
        return ComparisonResult.LEFT_IS_BIGGER;
    if (result == 0)
        return ComparisonResult.EQUAL;
    return ComparisonResult.RIGHT_IS_BIGGER;
}
```

But, this code has a bug!
@Test
public void testCompareWithEnum() {
    assertEquals(ComparisonResult.LEFT_IS_BIGGER, compareWithEnum(2, 1));
    assertEquals(ComparisonResult.RIGHT_IS_BIGGER, compareWithEnum(1, 2));
    assertEquals(ComparisonResult.EQUAL, compareWithEnum(2, 2));
}
Tests allow us to use a different language

- Production code describes a **general** solution; test code is **concrete**
- By specifying exact input and output, our test code can avoid repeating production
  - And therefore avoid writing the same bugs twice
- In general, you should avoid logic in tests
  - Otherwise you need to write tests for you tests, and nobody wants that
  - *Quis custodiet ipsos custodes*
Avoiding logic in tests

Duplication in tests isn’t as big of a deal as in production code (example taken from Google’s testing blog)

```java
@Test public void shouldNavigateToPhotosPage() {
    String baseUrl = "http://plus.google.com/";
    Navigator nav = new Navigator(baseUrl);
    nav.goToPhotosPage();
    // Hidden bug
    assertEquals(baseUrl + "/u/0/photos", nav.getCurrentUrl());
}
```

To see the bug, we need to inline the variables

```java
@Test public void shouldNavigateToPhotosPage() {
    Navigator nav = new Navigator("http://plus.google.com/");
    nav.goToPhotosPage();
    // Oh -oh , there are two slashes
    assertEquals("http://plus.google.com //u/0/photos", nav.getCurrentUrl());
}
```
Avoiding logic in tests

Duplication in tests isn’t as big of a deal as in production code (example taken from Google’s testing blog)

```java
@Test public void shouldNavigateToPhotosPage() {
    String baseUrl = "http://plus.google.com/";
    Navigator nav = new Navigator(baseUrl);
    nav.goToPhotosPage();
    // Hidden bug
    assertEquals(baseUrl + "/u/0/photos", nav.getCurrentUrl());
}
```

To see the bug, we need to inline the variables

```java
@Test public void shouldNavigateToPhotosPage() {
    Navigator nav = new Navigator("http://plus.google.com/");
    nav.goToPhotosPage();
    // Oh-oh, there are two slashes
    assertEquals("http://plus.google.com//u/0/photos", nav.getCurrentUrl());
}
```
What not to test?

Problem: how can we test the below code?

```java
class ColorToRGB {
    private Map<Color, RGB> map = Map.of(
        COLOR.RED, new RGB(255, 0, 0),
        COLOR.GREEN, new RGB(0, 255, 0),
        COLOR.BLUE, new RGB(0, 0, 255));

    public RGB colorToRGB(Color c) {
        return map.get(c);
    }
}
```

Answer: there's no real logic here, so there's nothing to test!
Problem: how can we test the below code?

class ColorToRGB {
    private Map<Color, RGB> map = Map.of(
        COLOR.RED, new RGB(255, 0, 0),
        COLOR.GREEN, new RGB(0, 255, 0),
        COLOR.BLUE, new RGB(0, 0, 255));

    public RGB colorToRGB(Color c) {
        return map.get(c);
    }
}

Answer: there’s no real logic here, so there’s nothing to test!
What not to test (cont.)

But, *software changes*

```java
class ColorToRGB {
    private final RGB[] array =
        {RGB(255, 0, 0), RGB(0, 255, 0), RGB(0, 0, 255)};
    public RGB colorToRGB(Color c) {
        return array[c.ordinal()];
    }
}
```

- On the one hand, we want our tests to be implementation independent.
- On the other, even the *existence* of tests can depend on the implementation.
  - “Let’s only add tests after we modify the code!”
But, *software changes*

```java
class ColorToRGB {
    private final RGB[] array = {
        RGB(255, 0, 0), RGB(0, 255, 0), RGB(0, 0, 255)
    };
    public RGB colorToRGB(Color c) {
        return array[c.ordinal()];
    }
}
```

- On the one hand, we want our tests to be implementation independent
- On the other, even the *existence* of tests can depend on the implementation
  - “Let’s only add tests after we modify the code”!
    - Hard to prove you didn’t change anything
- Work in iterations:
  1. Test the trivial code
  2. Change the code
  3. Verify the tests still pass
Avoiding regression

Whenever you fix a bug, you should write *at least one* test

- The test(s) should pass iff the bug was fixed
- Prevents the bug from recurring again (regression)
- If the test(s) had been there to begin with, the bug wouldn’t have happened
Private methods

How do we test private methods?

1. Using reflection?
   - Absolutely not!
   - All validation (name, input parameters, etc.) are done at run-time
   - Test can fail for the wrong reasons, adding noise

2. Change visibility to package-private?
   - Doable, but still not ideal
   - Only as last resort
Private methods (cont.)

- Your private methods are private for a reason!
  - Remember: test the **interface**, not **implementation**
  - Your private methods are tested implicitly through the public interface of the class
- “But I have a really crucial method that’s hard to test using only the public methods”!
Private methods (cont.)

- Your private methods are private for a reason!
  - Remember: test the **interface**, not **implementation**
  - Your private methods are tested implicitly through the public interface of the class
- “But I have a really crucial method that’s hard to test using only the public methods”!
  - Then extract it to a class
  - Improves the **cohesion** (more on this later in the course)
  - If it’s important enough to test, it’s important enough to have its own class
Making sure your tests can fail

Always make sure your tests can fail before they pass

- A good test can fail but passes; a bad test fails but should pass; a terrible test passes but **should** fail
- Could be done manually, e.g., by temporarily breaking the code
- We get this for free when doing TDD
Do tests slow us down?

- Often, test code is longer than production code
  - Although most of it is **boilerplate**
  - But it still takes at-least as long to write, if not more
- “So obviously, writing tests slows us down, right?”
- Not necessarily (although actual research varies)
Do tests slow us down? (cont.)

- Speed up bug finding when a test fails
  - Tests act as an exact pointer to your defective code
  - The smaller the test, the more accurate the pointer
- Less bugs are introduced
- Refactoring code improves its quality, often leading to more productiveness
  - I.e., the software design “hypothesis”
- Psychological safety
  - “Move fast and break things”...
  - …But easy to detect and fix what we broke as soon as we did
“Why do we need to write unit tests? Just write integration/end-to-end tests!”

- We can avoid all the pit-falls of testing
  - Impeding change
  - Integration tests are much more black-box, since we don’t care about the implementation
  - We can get good coverage with less tests, saving time (i.e., money)
“Why do we need to write unit tests? Just write integration/end-to-end tests!”

- We can avoid all the pit-falls of testing
  - Impeding change
  - Integration tests are much more black-box, since we don’t care about the implementation
  - We can get good coverage with less tests, saving time (i.e., money)
- Unit tests are cleaner and easier to set-up
- Unit tests are several orders of magnitude faster
  - And therefore easier to integrate into your development cycle
- Unit tests are much more precise
  - A failing integration test can only tell us that *something* went wrong
  - A unit test can often pin-point us to the exact bug
Why not just unit tests

But that doesn’t mean you shouldn’t write integration tests!

- Integration tests offer an extra level of protection
- It’s possible to write code that passes unit tests but still fails in production
- Like unit tests, end-to-end tests let specify concrete input and output
  - Easier to find errors
  - But not very useful at finding causes
IDEs can give you numerical and visual representation of your test 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 of 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?
Too many tests

Is there such a thing as too many tests?

- Depends on the tests
- Fast, short, black-boxy tests have almost zero cost beyond their initial writing time
  - Not all refactorings are pretty or even predictable
  - The more tests we have, the easier it is to catch bugs
- But some tests are redundant
  - They don’t check anything new
  - They fail with other tests, adding noise
  - But they might still be useful, e.g., for documentation
  - Or for increased confidence
How do you know when to stop testing? No clear answer

1. When you reach good coverage?
2. When you have high confidence in your code?

Probably a combination of the two
There are other kinds of tests other than unit tests:

- **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, but in a good way
  - Much stronger than coverage

- **Design by Contract** Asserting pre and post-conditions on class methods
  - fails in runtime
  - possible to check some stuff even at compile time using annotation processors