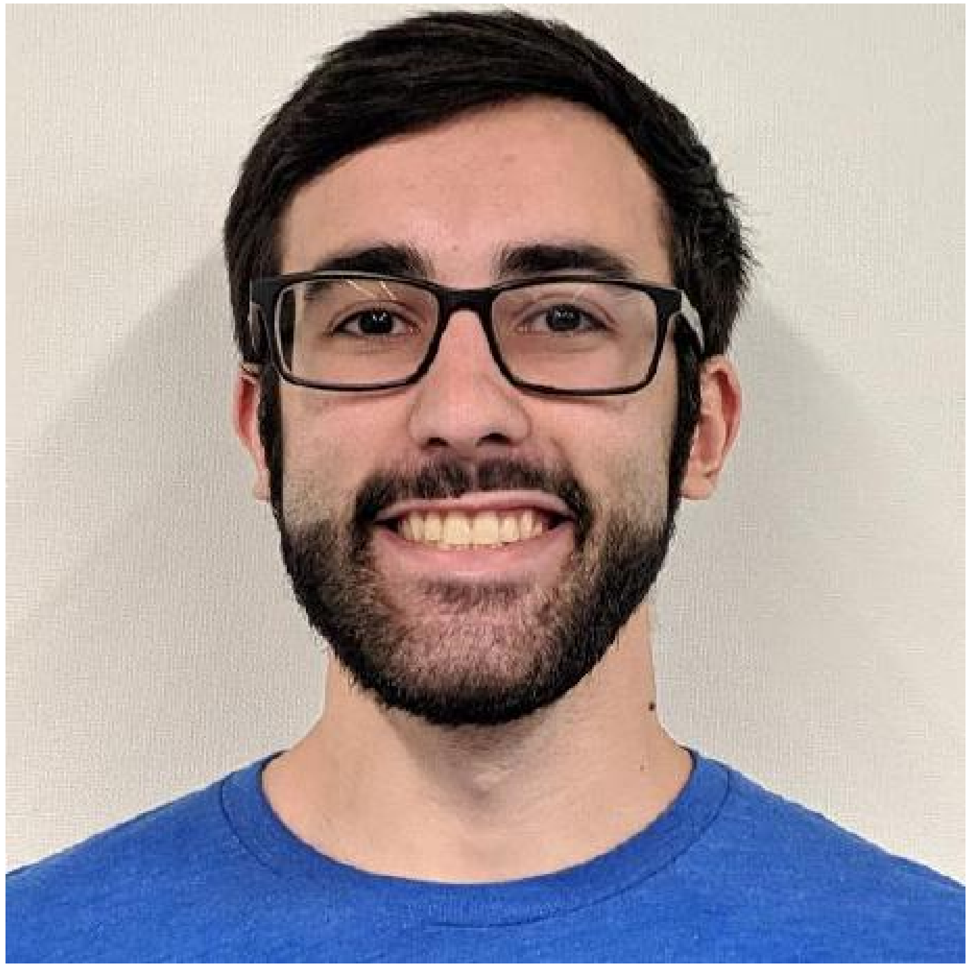
About Testing

Tomer Aberbach

Software Engineer @ Google

But First, About Me

- Tomer Aberbach
- Software Engineer on Google Docs, Sheets, and Slides
- Office Location: NYC (but working from home)
- TCNJ Alum (CS Major & Math Minor)
- Hobbies and Interests
  - Coding side projects
  - Playing piano
  - Composing music
  - Crocheting!



What is Software Testing?

Software testing is the process of evaluating and verifying that a software product or application does what it is supposed to do.

- *IBM*

Story Time

A software engineer has been tasked with writing some code that converts a `Color` enum to its RGB hex string.

```
enum Color {
  RED, GREEN, BLUE;

  public String getHex() {
    // TODO: do some coding mumbo jumbo
  }
}
```

They Code and They "Test"

It works! ... But does it?

```java
// Color.java
enum Color {
  RED, GREEN, BLUE;

  public String getHex() {
    String rgb = "";

    switch (this) {
      case RED:
        rgb = "#ff0000";
      case GREEN:
        rgb = "#00ff00";
      case BLUE:
        rgb = "#0000ff";
    }

    return rgb;
  }
}
```

```java
// ColorPrinter.java
public class ColorPrinter {
  public static void main(String[] args) {
    System.out.println(Color.BLUE.getHex());
    //=> #0000ff
  }
}
```

Oops...

Feeling a little blue?

```java
public class ColorPrinter {
  public static void main(String[] args) {
    System.out.println(Color.BLUE.getHex());
    //=> #0000ff

    System.out.println(Color.RED.getHex());
    //=> #0000ff

    System.out.println(Color.GREEN.getHex());
    //=> #0000ff
  }
}
```

The Fix

```
enum Color {
  RED, GREEN, BLUE;

  public String getHex() {
    String rgb = "";

    switch (this) {
      case RED:
        rgb = "#ff0000";
+       break;
      case GREEN:
        rgb = "#00ff00";
+       break;
      case BLUE:
        rgb = "#0000ff";
+       break;
    }

    return rgb;
  }
}
```

Testing to the Rescue!

Why do we test?

- To catch bugs *before* delivering code to the user
- Bugs can:
  - Be mildly inconvenient - *This link is broken!*
  - Cost money - *Ugh, I'll just download a different app!!*
  - Cause data loss or corruption - *My file didn't save!!!*
  - Result in privacy violations - *My private messages were leaked online!!!!*
  - Paint everything blue - *My eyes!!!!!*
  - etc.

But How?

Use a testing framework! They vary, but they all have:

- *Test suites*, which consist of one or more…
- *Tests*, which consist of one or more…
- *Assertions*: code that *asserts* some boolean expression is true

JUnit

```
// Just importing our testing framework
import static org.junit.Assert.assertEquals;
import org.junit.Test;

// Your first test suite: it's just a class!
public class ColorTest {
  // Your first test: it's just a method!
  @Test
  public void testGetHex_red() {
    Color color = Color.RED;

    String hex = color.getHex();

    // Your first assertion: it's just a method call!
    assertEquals("#ff0000", hex);
    // What we expect ^      ^ What we actually computed
  }
}
```

With Our Buggy Code

```java
// Just importing our testing framework
import static org.junit.Assert.assertEquals;
import org.junit.Test;

// Your first test suite: it's just a class!
public class ColorTest {
  // Your first test: it's just a method!
  @Test
  public void testGetHex_red() {
    Color color = Color.RED;

    String hex = color.getHex();

    // Your first assertion: it's just a method call!
    assertEquals("#ff0000", hex);
    // What we expect ^      ^ What we actually computed
  }
}
```

```
1) testGetHex_red(ColorTest)
java.lang.AssertionError: expected:<"#ff0000"> but was:<"#0000ff">
  at org.junit.Assert.fail(Assert.java:88)
  ...

FAILURES!!!
Tests run: 1,  Failures: 1
```

With Our Fixed Code

```java
// Just importing our testing framework
import static org.junit.Assert.assertEquals;
import org.junit.Test;

// Your first test suite: it's just a class!
public class ColorTest {
  // Your first test: it's just a method!
  @Test
  public void testGetHex_red() {
    Color color = Color.RED;

    String hex = color.getHex();

    // Your first assertion: it's just a method call!
    assertEquals("#ff0000", hex);
    // What we expect ^      ^ What we actually computed
  }
}
```

```
OK (1 test)
```

More Tests

```java
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class ColorTest {
  @Test
  public void testGetHex_red() {
    Color color = Color.RED;

    String hex = color.getHex();

    assertEquals("#ff0000", hex);
  }

  @Test
  public void testGetHex_green() {
    Color color = Color.GREEN;

    String hex = color.getHex();

    assertEquals("#00ff00", hex);
  }

  @Test
  public void testGetHex_blue() {
    Color color = Color.BLUE;

    String hex = color.getHex();

    assertEquals("#0000ff", hex);
  }
}
```

```
OK (3 tests)
```

Somewhat Frequently Asked Questions

- *When should we write tests?*
- *How many tests should we write?*
- *When should we run tests?*
- *What makes a good test?*

Somewhat Frequently Answered Questions

- *When should we write tests?*
  - Whenever we add new behavior to or change the existing behavior of the code!
- *How many tests should we write?*
  - As many as it takes to give us confidence that the code works!
- *When should we run tests?*
  - On every code addition or change! (e.g. on every Git commit or pull request) Especially if it's a fix for a bug that the tests didn't catch!
- *What makes a good test?*
  - A good test is small, simple, and deterministic
  - Common pitfall: tests so complex that they practically need their own tests!

Test Structure

- *Arrange* all necessary preconditions and inputs
- *Act* on the object of method under test
- *Assert* that the expected results have occured

```java
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class ColorTest {
  @Test
  public void testGetHex_red() {
    // Arrange
    Color color = Color.RED;

    // Act
    String hex = color.getHex();

    // Assert
    assertEquals("#ff0000", hex);
  }

  // ...
}
```

Testing Levels

- Unit testing:
  - Tests that verify a small unit of code (e.g. a single method call)
- Integration testing
  - Tests that verify interaction between multiple components (e.g. multiple classes that call each other's methods)
- System testing:
  - Tests the entire system as a whole (e.g. open the software)

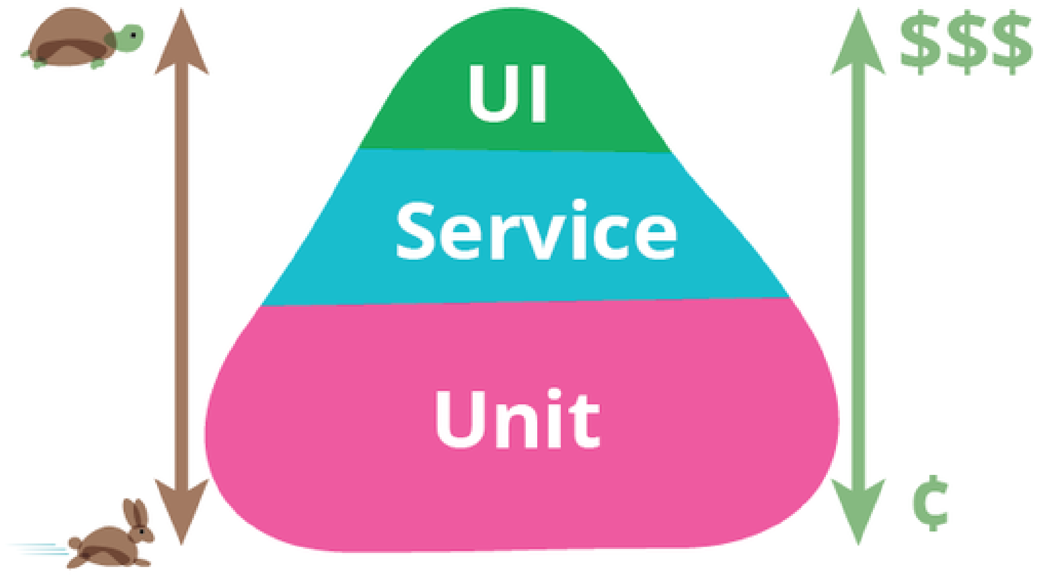


Image by Martin Fowler

Can't Only Have Unit Tests!

Test Doubles

How do we test code that uses production systems?

```java
public class MemeFetcher {
  private final MyDatabase database = new MyDatabase();

  public List<String> getMemeIds(String searchQuery) {
    return database.query("memes", searchQuery);
  }
}
```

Are we going to set up a whole database for our tests?

Dependency Injection to the Rescue!

A fancy name for taking parameters! Usually interfaces or abstract classes.

```
public class MemeFetcher {
- private final MyDatabase database = new MyDatabase();
+ private final Database database;
+
+ public MemeFetcher(Database database) {
+   this.database = database;
+ }

  public List<String> getMemeIds(String searchQuery) {
    return database.query("memes", searchQuery);
  }
}
```

Pass in `MyDatabase` in production and `FakeDatabase` in tests!

Test Suite Quality

How do we know if we have enough tests? And if our tests are good?

- **Coverage:** the percentage of code exercised by the test suite
- **Flakiness:** how often does the test randomly fail?
  - A test is *flaky* if it randomly fails sometimes (without changing the code)
- **Mutation Tests:** automatic random modifying of your software code
  - What does it mean if the test suite still passes?
- **Regression Tests:** tests that catch regressions in behavior, frequently rgressions that have happened before
  - Are the same bugs going undetected over and over?

Types of Tests

- Regression Testing
- Parameterized Testing
- Snapshot Testing
- Fuzz Testing
- Property-Based Testing
- Mutation Testing
- Compatibility Testing
- Smoke Testing
- Latency Testing
- Stress Testing

And there are many more!

Thanks for Listening!

Resources

- Articles
  - Google Testing Blog
  - Testing on the Toilet
  - Martin Fowler's Blog
  - Mutation Testing

- Libraries and Frameworks
  - JUnit
  - Mockito (for mocking)
  - TestParameterInjector (for parameterized testing)
  - JUnit QuickCheck (for property-based testing)
  - PIT (for mutation testing)
  - Selenium (for browser automation)