Test-driven design

Gal Lalouche
Oath Inc.

Computer Science
Technion

Spring 2018
The measure of how easily testable your code is

- Of course, we want our code to be testable to make writing tests easier
- But often we have to work hard to make our code testable
- Is this a waste of time?
  - After all, time spent refactoring our code for tests is time spent not adding features and fixing bugs
- No!
- Making our code testable improves its design
Improving testability

How can we improve the testability of our code?

1. **Controllability.** Modifying the state and behavior of the Component Under Test (CUT)
2. **Isolateability.** Testing the CUT without in isolation, i.e., without its dependencies
3. **Observability.** Asserting and verifying the result of the test
4. **Separation of concerns.** The CUT should have a single, well defined responsibility
5. **Understandability.** How well do we understand the CUT?
How can we improve each of those requirements?

1. **Controllability and Isolateability.** Using dependency injection to replace real objects with test test doubles

2. **Observability.** Avoiding side-effects

3. **Separation of concerns.** Extracting behavior to other classes

4. **Understandability.** Improve code structure, refactoring, external documentation (javadocs)
Testability improves design

But, we already saw that applying those techniques and guidelines improves our design

1. **Controllability and Isolateability.** Using dependency injection makes our class more flexible, cohesive, and robust.

2. **Observability.** Preferring pure functions and immutability increases our ability to reason about our program.
   - Will be revisited when we talk about functional programming.

3. **Separation of concerns.** Increasing cohesion, single responsibility principle.

4. **Understandability.** Reveals intent.
Hard to test? Rethink the design!

- If it is hard to test, it will be hard to use
  - Creating new objects is too cumbersome?
  - Class is tightly coupled with dependencies?
  - Code is too complicated to test?
  - Are you not even sure what you should be testing?

- All of these issues will plague your production code as well
Hard to test code

But some code is simply harder or impossible to test

- Graphical user interfaces
- Distributed systems
- Database schema
- Auto generated code

So what do we do then?
As we saw in other parts of the course, the solution is decomposition

- Separate the hard to test parts from the testable ones
- Test what you can in isolation
- Keep what can’t be tested to as thin a layer as possible
- Once again, these steps improve our design!
Refactoring code for testability

Mixing IO, side effects, and logic

```java
public void registerUser() {
    String userId = readLine();
    String passwd = readLine();
    if (!isValidUserName(user_name))
        return;
    if (!isStrongEnoughPassword(passwd))
        return;
    Url url = getDatabaseUrl();
    Connection connection = getUserDatabaseConnection();
    if (!checkAvailableUsername(url, connection, user_name))
        return;
    putUser(connection, url, user_name, passwd);
}
```
Extracting the easy to test parts out

```java
public boolean isValid(String userName, String passwd, Predicate<String> isUserNameAvailable) {
    return isValidUserName(userName) &&
        isStrongEnoughPassword(passwd) &&
        isUserNameAvailable.test(userName);
}

public void registerUser() {
    String userId = readLine();
    String passwd = readLine();
    Url url = getDatabaseUrl();
    Connection connection = getUserDatabaseConnection();
    Predicate<String> isUserNameAvailable = userName ->
        checkAvailableUsername(url, connection, userName);
    if (!isValid(userName, passwd, isUserNameAvailable))
        return;
    putUser(connection, url, userName, passwd);
}
```
TDD in a nutshell

Test driven development
The process of writing tests before production code

The development cycle in TDD follows a simple Red-Green-Refactor cycle

1. Write a test that fails
2. Write the simplest code that passes the tests
3. Refactor (tests and code!), and make sure tests still pass

Repeat until task is complete
TDD Benefits

- **Testability**
  - You can’t add features that aren’t tested
- **Fast and clear development cycle**
  - The TDD heartbeat really does *drive* development
- **Confidence in code and in test**
  - We know our tests can fail
  - We know we have high coverage so we can refactor at ease
TDD in depth: Tests

- Write simple and short tests
  - Usually we start the simplest tests and work our way up
  - Use the IDE to automatically generate class and method signatures
- Tests are the specification of your code
  - Once you write down a test, you know the feature is done when it’s passing
- You can write multiple tests to cover a single feature
  - But always make sure your tests can fail
- This part is done when the test runs and fails
TDD in depth: Code

- Write the simplest code you can to make all the tests pass
  - Including code duplication
  - Including hard coding the results
  - Including writing code that works only for the specific test input
  - See Do The Simplest Thing That Could Possibly Work
- This part is done as soon as the tests pass
Fake it ’till you make it

An extreme example of doing the simplest thing possible is the “Fake it ’till you make it” or “Triangulation” pattern

- Delay writing “real” production code for as long as you can
- Write “fake”, hard-coded logic only for the sake of writing the tests
- Keep writing tests until you are forced to replace your fake code with real one
- As a result you end up with very high code coverage
Let’s look at an example method for reversing a string:

```java
@Test public void testEmpty() {
    assertEquals("", reverse(""));
}
```

We can pass the test by returning a hard-coded value:

```java
public static String reverse(String s) {
    return "";
}
```
Let’s look at an example method for reversing a string:

```java
@Test public void testEmpty() {
    assertEquals("", reverse(""));
}
```

We can pass the test by returning a hard-coded value

```java
public static String reverse(String s) {
    return "";
}
```
Fake it ’till you make it: example (cont.)

```java
@Test public void testSingleLetter() {
    assertEquals("a", reverse("a"));
}
```

We can pass the test by returning the input:
Fake it ’till you make it: example (cont.)

```java
@Test public void testSingleLetter() {
    assertEquals("a", reverse("a"));
}
```

We can pass the test by returning the input

```java
public static String reverse(String s) {
    return s;
}
```
Forcing an actual implementation

```java
@Test public void testString() {
    assertEquals("foobar", reverse("raboof"));
}

public static String reverse(String s) {
    // Okay, now we are forced to write real logic
}
```
TDD in depth: Refactor

This is the most important part!
- Remove any code duplication from the previous part
  - Both in production code and in the tests
- Redesign your class and tests
  - Refactoring is easy since our tests cover all of our functionality by definition
- This part is complete only when you are satisfied with the quality of your design
Productivity boosts

By focusing on writing tests first, we get a very efficient production cycle

- Tests are (usually) easier to write than production code
  - If it’s harder, then there might be something wrong with your design
  - We pick the lowest hanging fruit first, but once we got a test we know what our next goal is

- The coding part is usually pretty fast too
  - We always write the simplest code to make the tests pass
  - We’re working incrementally, adding a small feature every time

- Refactoring is easier and safer (and therefore faster), since by the definition of TDD all our features are covered by tests
  - Impossible to break existing features without tests failing
  - But refactoring tests should be done more carefully
Despite its name, the point of TDD isn’t writing tests

- TDD is a methodology for writing production code
  - As its name suggests, tests drive development, not the other way around
  - You should write tests whether you use TDD or not
- By writing the tests, we can imagine our ideal API without uglying it (yet) with the actual implementation worries
  - Focus on the abstraction, not the details
And, as a side-effect, it improves our unit tests

- Our tests tend to be more declarative and readable, since they are the specification of our code,
  - And they are written without considering implementation logic (which is a good thing!)
- Units tests written after we wrote (a lot of) production code tend to be poorer
- It’s common to "be-little" the importance of tests if we already got our code working
- Easy to forget what the code was supposed to be doing in the first place
- Hard to remember all the edge cases
TDD and Testability

TDD puts testability to the front of center

- You can’t even write code if it’s not testable
- Code tends to be more testable than usual: If our code is hard to test, it will be difficult to add new features
- By writing the tests before the implementation, we focus on our external API instead of internal details
TDD and YAGNI

TDD takes YAGNI to an extreme level

- Not only are we not writing extra features, we’re not even writing any more code than the bare necessities to make the tests pass
- Even after refactoring, we don’t usually add complex logics to our code
- TDD takes the mentality of "Not tested? Not needed!"
TDD Pros

Use TDD when you:
- Know what your inputs and output are, but you don’t know what the ideal implementation will be
- Have a lot of edge cases that you want to cover
- Want to ensure high test coverage of your code
TDD Cons

But TDD can be problematic when

- Dealing with legacy code
- Not all code needs or should be tested
  - Writing throwaway code is harder when there is more overhead involved
- Some code is just plain harder to test
  - And trying to apply TDD to it will only slow you down without any real benefit
- Focusing on testability can lead to over-abstraction and code complications
  - See Test Induced Design Damage