Test Driven Development

Idaho State University | Computer Science

Isaac Griffith

CS 2263
Department of Informatics and Computer Science
Idaho State University

ROAR

Outcomes

After today's lecture you will be able to:

- Describe the basics of TDD
- Use and execute TDD with JUnit
- Know when to use and not to use TDD practices

Inspiration

"Testing by itself does not improve software quality. Test results are an indicator of quality, but in and of themselves, they don't improve it. Trying to improve software quality by increasing the amount of testing is like trying to lose weight by weighing yourself more often. What you eat before you step onto the scale determines how much you will weigh, and the software development techniques you use determine how many errors testing will find. If you want to lose weight, don't buy a new scale; change your diet. If you want to improve your software, don't test more; develop better." – Steve McConnell

Introduction

- Back in the old days, we would write the code then we would test it.
- Then to verify our code as soon as possible, we started writing the tests first.
- This led to code that worked better and that had better architecture, maintainability and API ease of use.
- This is now called Test-Driven Development (TDD)

When to Test

- After implementation
  - Test-Last, Code-First
- Before implementation
  - Test-First, Code-Last
- Everytime a bug is found

Code-First Development

- Tests written after production code is finished
- Traditional approach, only one available for legacy code
- **Main Advantage**: Main functionality of tested object is well understood
- **Main Disadvantage**: Developer focuses on testing implementation instead of testing the behavior of the SUT
  - Tests are tightly coupled to implementation
  - Developers subconsciously select test cases that will pass
- Temptation is always to not write tests

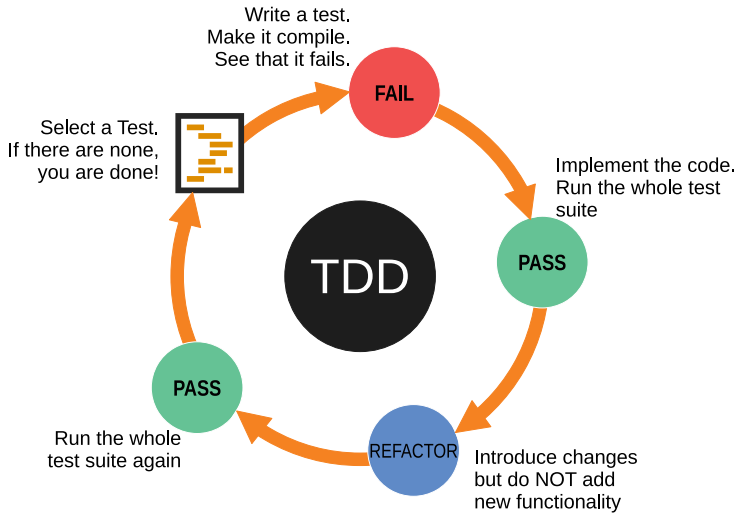
Test-First Development

- Makes developers think about behavior to be tested
- Cuts down on functionality implemented to the bare minimum
- Results in a very high-level of code coverage

After a Bug is Found

- When a bug is reported:
  - Resist the urge to immediately fix it
  - Write a test first, to detect if the bug returns
  - Fix the code so the test(s) pass

TDD Rhythm



RED– Write a Test that Fails

- Select functionality to be implemented
- Write it as a test
- The test won't pass as the functionality is not implemented
- This is important, as
  - Shows functionality really doesn't work
  - Once implemented, correctly, you should see the test turn from **red** to **green**
- The failure message should precisely indicate what went wrong

TDD is about API Design

- Your tests are the first client of your new API
- This way of thinking, from the client-perspective, allows you to concentrate on what is really required.
- **Concentrate on what the client (the test code) really needs. And write tests which test exactly this and no more.**
- Thus, we focus on behavior rather than implementation and produce more maintainable tests loosely coupled to the code.

GREEN – Fix the Code

- **Write the smallest amount of code that will satisfy the test.**
- This will make the test transition from **red** to **green**
- The key here is to **focus on the task at hand** and not on further enhancements or other requirements

REFACTOR – Improve the code

- Now that the code works, we turn to the following questions:
  - Will it be easy to enhance?
  - Will it be flexible?
  - Will it adhere to O-O Principles?
- To answer yes, we may need to refactor.
  - Remove duplicate code
  - Change method names
  - Update variable scope
  - Move code from one place to another
- Rerun tests after each change to ensure they all still pass.
- Goal is to constantly improve the quality of your code!

Refactoring Tests

- But what about our tests, should we refactor those? **Of course!**
- But we don't test our test code, what if we break them
  - Certainly, this is a risk
  - But, test code tends to be simple
  - Does not include complex logic that tends to break
- Prior to refactoring tests, always make sure all tests are passing
  - Thus, if you break them, you will see them turn **red**
- When finished refactoring, **run all the tests again** and verify everything passes

Test First Example

- We will work together to use TDD to develop a simple class
- So, open you Java IDE and follow along
- We will follow each phase one-by-one
  1. Start with a failing test
  2. Make error message information enough
  3. Fix the code
  4. Refactor

The Problem

- We will implement a `FootballTeam` class
  - To compare different teams to find league first place
  - Each team tracks games won
- For example, to compare two teams, we can
  - Use a `Comparable` interface
  - To compare, each team, will need to track its games won in a field that is accessible
  - We need to see that teams with more wins are ranked first
  - We need to check what happens when two teams have the same number of wins

RED – Write a Failing Test

- Let's Keep it simple our `FootballTeam` class will take # games won as a constructor param.
- Let's write the test
  - Create a new gradle project
  - Now add a new test class `FootballTeamTest`
  - Add test method `constructorShouldSetGamesWon` to test constructor sets games won

```
public class FootballTeamTest {

  @Test
  void constructorShouldSetGamesWon()
  {
    FootballTeam team = new
        FootballTeam(3);

    assertEquals(3, team.getGamesWon())
  }
}
```

RED – Write a Failing Test

- Next, we need to create enough code to make the test compile
  1. Create `FootballTeam` class with a constructor that takes an int parameter, that does nothing
  2. Create method `getGamesWon()` which simply returns 0.
- Improve the Assertion message

```
public class FootballTeam {

  public FootballTeam(int gamesWon) {
  }

  public int getGamesWon() {
    return 0;
  }
}
assertEquals("number of games won", 3,
```

GREEN – Fix the Code

- Simple, no fancy tricks
  - Just make `getGamesWon()`, return 3
- This **satisfies** the test, but only shows that **test is not good enough to cover certain functionality**
- Then rerun the tests to show it is now **green**

```
public class FootballTeam {

  public FootballTeam(int gamesWon) {
  }

  public int getGamesWon() {
    return 3;
  }
}
```

REFACTOR – only a bit

- The `FootballTeam` class is too simple to require refactoring at this time
- But, the `FootballTeamTest` class can be slightly refactored
  - We need to deal with the **magic number** 3 by replacing it with a constant like `THREE_GAMES_WON`
- **Run the whole test suite again!** (always the whole suite)
  - Should still be **green**

```
public class FootballTeamTest {

  private static final
  int THREE_GAMES_WON = 3;

  @Test
  void constructorShouldSetGamesWon() {
    FootballTeam team = new
        FootballTeam(THREE_GAMES_WON);

    assertEquals("number of games won",
        THREE_GAMES_WON,
        team.getGamesWon());
  }
}
```

Fail, Pass, Refactor

- The constructor, works fine for one value, but our one test only shows that our test suite is inadequate
- We should add tests for more values
- We should add tests that make sure the constructor throws an exception if an inappropriate value is passed to it
- For both cases, we use parameterized tests

Testing Valid Values

- The tests so far work, but only for one value
- We need to test more values
- This will put more strain on the `FootballTeam` class
- To do this we will use
  - Parameterized tests
  - `@ValueSource` annotation
- When we run this test, we see it fails for 3 of the 4 test cases

```
import org.junit.runner.Parametrized;
import org.junit.runners.Parameterized.Parameters;
import org.junit.runner.RunWith;
...
@RunWith(Parameterized.class)
public class FootballTest {
  private int gamesWon;
  private int expected;

  public FootballTest(int gamesWon, int expected) {
    this.gamesWon = gamesWon;
    this.expected = expected;
  }

  @Parameterized.Parameters
  public static Collection values() {
    return Arrays.asList(new Object[][] {
      {0, 0},
      {1, 1},
      {3, 3},
      {10, 10}
    });
  }
```

Testing Valid Values

- The fix is straight forward
- We then re-run the tests to see that they are **green**.

```
public class FootballTeam {
  private int gamesWon;

  public FootballTeam(int gamesWon) {
    this.gamesWon = gamesWon;
  }

  public int getGamesWon() {
    return gamesWon;
  }
}
```

Testing Invalid Values

- Here we deal with invalid values
- This test fails, as the code currently accepts any value.

```
@Parameterized.Parameters
public static Collection values() {
  return Arrays.asList(new Object[][] {
    {Type.LEGAL, 0, 0},
    {Type.LEGAL, 1, 1},
    {Type.LEGAL, 3, 3},
    {Type.LEGAL, 10, 10},
    {Type.ILLEGAL, -10, 0}
    {Type.ILLEGAL, -1, 0}
  });
}

enum Type {LEGAL, ILLEGAL};

public FootballTest(Type type, int gamesWon, int expected) {
    this.type = type;
    ...
}
```

Testing invalid values

```
@Test
void constructorShouldSetGamesWon()
{
    Assume.assumeTrue(type == Type.LEGAL);
    ...
}

@Test(expected = IllegalArgumentException.class)
void constructorShouldThrowExceptionForIllegalGamesNb() {
    Assume.assumeTrue(type == Type.ILLEGAL);
    new FootballTeam(gamesWon);
}
```

Testing Invalid Values

- The Fix is straight-forward, and the test should be **green** now.

```
public FootballTeam(int gamesWon) {
  if (gamesWon < 0) {
    throw new IllegalArgumentException(
      "Not possible to have less than 0 games won!" +
      "was (" + gamesWon + ")");
  }
  this.gamesWon = gamesWon;
}
```

But is it Comparable?

- The constructor is good now.
- So we turn to the requirement of being able to compare teams
- This test verifies the class is comparable.

```
private static final int ANY_NUMBER = 123;

@Test
void shouldBePossibleToCompareTeams() {
  FootballTeam team =
      new FootballTeam(ANY_NUMBER);

  assertTrue(team instanceof Comparable.class);
}
```

But is it Comparable?

- Add the `Comparable` interface and `compareTo()` method
- The test should now be **green**
- Be sure to rerun the entire test suite

```
public class FootballTeam
    implements Comparable<FootballTeam> {
  private int gamesWon;

  public FootballTeam(int gamesWon) {
    // ...
  }

  public int getGamesWon() {
    // ...
  }

  @Override
  public int compareTo(FootballTeam otherTeam)
  {
    return 0;
  }
}
```

Idaho State University
Computer Science

First Comparison Test

Test

- Idea is to compare two teams with different wins
- Should indicate team with larger number is greater than other

```
@Test
void teamsWithMoreWinsSholudBeGreater() {
    FootballTeam team2 = new FootballTeam(2);
    FootballTeam team3 = new FootballTeam(3);

    assertTrue(team3.compareTo(team2) > 0);
}
```

Code

- Fix the code
- Rerun the tests, should all be **green**

```
@Override
public int compareTo(FootballTeam o) {
    return 1;
}
```

ROAR

CompareTo recognizes better teams

- Update `compareTo` to be a descent implementation
- Rerun tests

```
@Override
public int compareTo(FootballTeam o) {
    if (gamesWon) > o.getGamesWon()) {
      return 1;
    }

    return 0;
}
```

- Refactor
  - `o` -> `otherTeam` in `compareTo()`
- Rerun tests

```
@Override
public int compareTo(FootballTeam otherTeam) {
    if (gamesWon) > otherTeam.getGamesWon()) {
      return 1;
    }

    return 0;
}
```

Idaho State University
Computer Science

Another Comparison Test

Test

- Test that those teams with lower results are found to be less than teams with higher number of games won.

Code

- Then fix the code
- Rerun tests, all should be **green**

```
@Test
void teamsWithLessGamesWonIsLesser() {
    FootballTeam team2 = new FootballTeam(2);
    FootballTeam team3 = new FootballTeam(3);

    assertTrue(team2.compareTo(team3) < 0);
}

@Override
public int compareTo(FootballTeam otherTeam) {
    if (gamesWon > otherTeam.getGamesWon()) {
      return 1;
    } else if (gamesWon < otherTeam
        .getGamesWone()) {
      return -1;
    }
    return 0;
}
```

ROAR

Testing for Equality

Test

- This test will pass instantly

```
@Test
void teamsWithSameGamesWonShouldBeEqual() {
  FootballTeam teamA = new FootballTeam(2);
  FootballTeam teamB = new FootballTeam(2);

  assertEquals(0, teamA.compareTo(teamB));
}
```

Code

- To follow TDD rhythm and see that the test fails we:
  - Introduce a change in production code that breaks the test
  - Run all tests to see failure
  - Revert the change
  - Run all tests to see all pass

```
@Override
public int compareTo(FootballTeam otherTeam) {
    if (gamesWon > otherTeam.getGamesWon()) {
      return 1;
    } else if (gamesWon < otherTeam.getGamesWon
      return -1;
    }

    return 23948732;
}
```

Testing for Equality

Refactor

- Finally, we can refactor `compareTo()` method to simplify its implementation.

```java
@Override
public int compareTo(FootballTeam otherTeam) {
    return gamesWon - otherTeam.getGamesWon();
}
```

Getting Started with TDD

- Take small steps
- Start with simple easy to implement tasks
- Build-up experience using a disciplined and focused approach

When NOT to Use TDD

- When you lack a good understanding of the technologies used
- When you lack knowledge in the problem domain
- When working with legacy code

Blindly Following TDD

- You can write the assertion messages up front
- You do not need to go through the process of breaking tests that pass by default
- You can also have the IDE generate good code and by-pass some steps
- All in all, you should run and ensure your test works for each of these.

TDD and JavaDoc

- During the refactoring phase you should add JavaDoc for both your test and production code, or at least update the JavaDocs
- Keep it short but descriptive and follow good JavaDoc style
- If you don't want to break your flow, then add a FIXME or TODO comment to remind you to come back to it.

In-Class Exercise

Write a simplified version of a booking system

- Returns a list of booked hours
- not allow a particular hour to be doulbe-booked
- deal in a sensible manner with illegal values (provided as input parameters)

Constraints

- has only one resource that can be booked (e.g., classroom, table, etc.)
- has no notion of days, or to put it differently, it assumes all reservations are for today
- Should only permit booking of whole clock-hours
- Does not need to remember any additional info considering the reservation (who booked it, when, etc.)

Are there any questions?