43-Test Driven Design (TDD)

Author: Vincent Lau

Learning Objectives

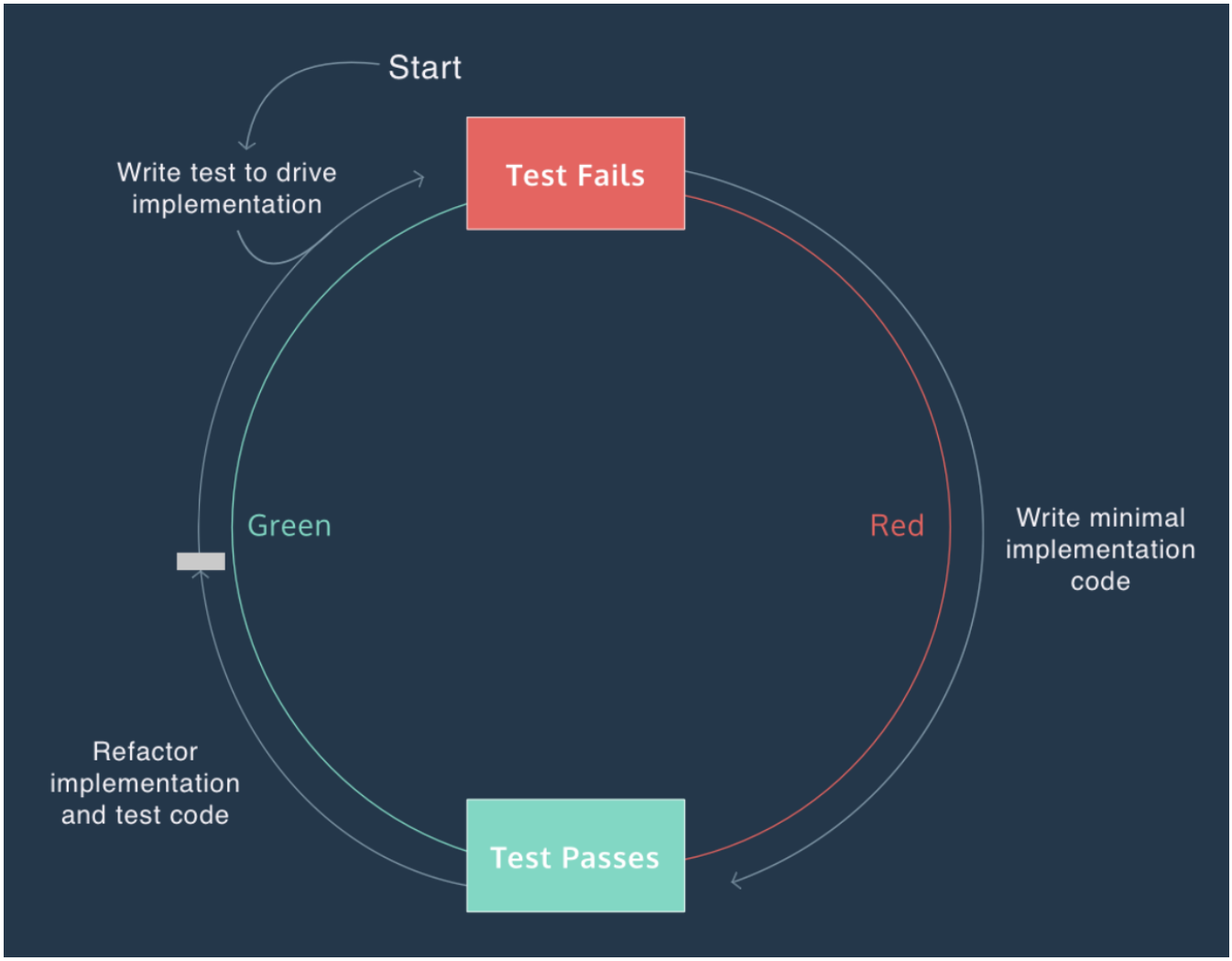
- Understand the purpose of creating Index for tables
- Understand the ways of implementing indexes
- Understand the advantages and disadvantages of creating indexes

Introduction

Test-Driven Development (TDD) is a software development methodology in which the creation of automated tests precedes the actual coding of the software's features or functionality. The primary goal of TDD is to ensure that the software functions correctly and reliably by constantly running tests throughout the development process.

Red-Green-Refactor Cycle

TDD follows a simple and iterative process known as the "Red-Green-Refactor" cycle



Red

Think about what you want to develop

Initially, a developer writes a failing test case that defines the expected behavior of a small piece of code. This test is called the "red" phase because it fails when run against the code that hasn't been implemented yet.

Green

Think about how to make your tests pass

The developer then writes the minimum amount of code required to make the failing test pass successfully. This is often referred to as "making the test green." The focus here is solely on making the test pass, not on writing extensive code.

Refactor

Think about how to improve your existing implementation

Once the test is green, the developer can refactor the code as needed to improve its structure, readability, and efficiency. The goal is to maintain or enhance the code's quality without altering its external behavior.

Benefits

Higher Code Quality

TDD encourages developers to write clean, modular, and maintainable code from the outset, as they need to make sure their code passes tests.

Improved Software Reliability

Constant testing helps catch and fix bugs early in the development process, leading to more reliable software.

Regression Testing

By having a suite of automated tests, developers can easily run these tests whenever changes are made to ensure that new code doesn't break existing functionality (regression testing).

Clearer Requirements

Writing test cases before coding forces developers to think about the expected behavior of their code, which can lead to clearer requirements and specifications.

Faster Debugging

When a test fails, it's usually easier to pinpoint and fix the issue because you have a failing test case that demonstrates the problem.

Challenges

Learning Curve

TDD can be a paradigm shift for some developers, requiring them to learn new practices and habits.

Initial Time Investment

Writing tests before coding may seem time-consuming initially, but it often pays off in reduced debugging and maintenance time.

Not Suitable for All Projects

TDD may not be the best fit for all types of projects, particularly those with rapidly changing requirements.

Experiment

Here's a simple Java code example that demonstrates the Red-Green-Refactor cycle using Test-Driven Development (TDD). In this example, we'll create a basic `Calculator` class and write tests for it step by step.

Design Phase

Before writing the failing test, let's design the `ShoppingCart` class and decide on some basic functionality.

The Java class should be compiled successfully, and the class and methods design should be clear enough for developers to read and basically understand the business meaning by checking the test cases, which will be developed in Red Phase.

In this example, we've designed the class with basic methods: `addItem` to add items to the cart, `getItems` to retrieve the items, and `clear` to empty the cart.

Developers & seniors should consider the clarity of the method presentation, and how the methods are decoupled for unit tests by JUnit in Design Phase.

In a more comprehensive TDD process, it's a good practice to design the class and method and provide placeholder return values (i.e. -1, null, etc) before writing the failing test case. This step helps establish a clear direction for your code implementation.

```java
import java.util.ArrayList;
import java.util.List;

public class ShoppingCart {
    private List<String> items;

    public ShoppingCart() {
        // TBC.
    }

    public void addItem(String item) {
        // Placeholder implementation
    }

    public List<String> getItems() {
        // Placeholder implementation
        return null;
    }
```

```
19
20      public void clear() {
21          // Placeholder implementation
22      }
23  }
```

Red Phase

Now, let's write a failing test for the `addItem` method. The test expects the cart to contain one item after adding one, but our placeholder implementation doesn't handle this yet.

But, at this stage, all main code and test code should be compiled successfully. All or some test cases fail.

Test Case Developers should cover the business & functionality test cases as much as possible, which should be able to cover most of the scenario of each unit in target programs.

```
1  import org.junit.jupiter.api.Test;
2  import static org.junit.jupiter.api.Assertions.*;
3
4  public class ShoppingCartTest {
5
6      @Test
7      public void testAddItem() {
8          ShoppingCart cart = new ShoppingCart();
9          cart.addItem("Item1");
10         assertEquals(1, cart.getItems().size()); // This test will fail.
11     }
12 }
```

Green Phase

Implement the `addItem` method to make the test pass. Now, if you run the test again, it should pass successfully.

In this phase, main code developers should focus on the requirements and the written test cases to make all test cases pass, so that the process should be focusing on the accuracy of functionality results. The algorithm tuning should not be the first priority of concern in this phase.

```
1  public void addItem(String item) {
2      items.add(item);
```

```
3 }
```

Refactor Phase

In this simple example, there's not much to refactor. However, let's make a small improvement by initializing the `items` list in the constructor instead of the declaration.

This refactoring improves code readability by clearly initializing the `items` list in the constructor, and changed the list implementation to LinkedList.

In this phase, there may be some non-functional requirements, such as time complexity. Developers improve the algorithm complexity (time & space), for example, by changing the data structure implementation and revisiting the algorithm (i.e. Clean Code Practice).

```
1  public class ShoppingCart {
2      private List<String> items;
3
4      public ShoppingCart() {
5          this.items = new LinkedList<>();
6      }
7
8      public void addItem(String item) {
9          items.add(item);
10     }
11
12     public List<String> getItems() {
13         return items;
14     }
15
16     public void clear() {
17         items.clear();
18     }
19 }
```

That's the complete TDD cycle with Design, Red, Green, and a simple Refactor phase for a shopping cart class. You can continue to add more features, write more tests, and refactor as needed to build a more comprehensive shopping cart system.

Agile & TDD

`Agile` and `Test-Driven Development` (TDD) are two complementary approaches used in software development to improve the quality, flexibility, and responsiveness of the development process. Here's an introduction to their relationship:

Agile Methodology

Agile is a broad set of software development methodologies and practices that prioritize flexibility, collaboration, and customer feedback. Agile methodologies, such as Scrum, Kanban, and Extreme Programming (XP), emphasize iterative development, incremental delivery of features, and close collaboration between development teams and stakeholders.

TDD (Test-Driven Development)

TDD is a specific development practice that falls under the Agile umbrella. It focuses on ensuring software quality and maintainability by writing automated tests before writing the actual code. TDD follows the Red-Green-Refactor cycle, where tests are written before the code is implemented. This process helps catch bugs early, maintain code integrity, and improve overall software design.

Relationship between Agile and TDD

Alignment with Agile Principles

a) Iterative and Incremental Development

Both Agile and TDD promote an iterative and incremental approach to software development. In Agile, you work in short iterations, delivering potentially shippable increments of the product. TDD follows a similar philosophy by continuously iterating through the Red-Green-Refactor cycle.

b) Customer-Centric Focus

Agile methodologies prioritize customer feedback and collaboration. TDD ensures that the software meets customer requirements by validating functionality through automated tests. This helps ensure that code meets the specified criteria and maintains the desired functionality as it evolves.

c) Responding to Change

Agile principles embrace changes in requirements, and TDD supports this by making it easier to refactor code and accommodate changing needs while maintaining confidence in the software's correctness.

Quality Assurance and Continuous Integration

In Agile development, there's a strong emphasis on continuous integration and automated testing. TDD aligns well with this aspect by providing a suite of automated tests that can be run continuously to ensure that new code changes don't introduce regressions.

Collaboration and Communication

Both Agile and TDD encourage close collaboration among team members. TDD fosters communication between developers, testers, and stakeholders because it requires clear specification of test cases and **expected behavior**.

Risk Mitigation

Agile methodologies aim to identify and address risks early in the development process. TDD contributes to risk mitigation by identifying and addressing defects as soon as they are introduced.

TDD is a development practice that aligns well with Agile principles and methodologies. It helps Agile teams ensure software quality, responsiveness to change, and customer satisfaction by continuously validating and improving code through automated testing. TDD is often considered a best practice within Agile development, as it supports the Agile goal of delivering high-quality software efficiently and with flexibility.

Summary

TDD is closely associated with agile and iterative development methodologies, and it has become a best practice in the software industry for improving code quality and reliability. It's commonly used in conjunction with testing frameworks and tools like JUnit (for Java), pytest (for Python), and many others, depending on the programming language and environment.