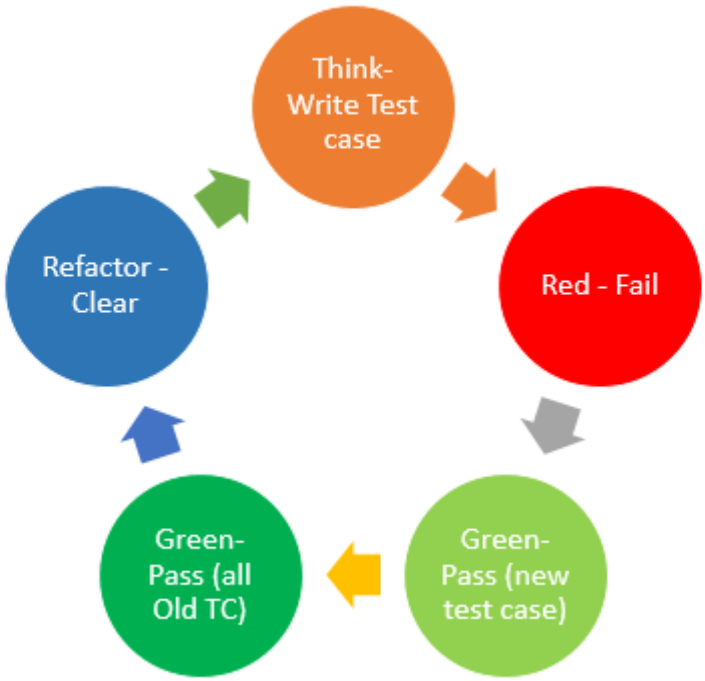
TDD(Test-Driven Development)

For the basic approach, TDD is writing software with taking small steps. TDD, block to approach that “it might be useful in development process”. In TDD, whenever we need new functionalities for our Project, add that functions that time, not in advance. While build our architecture we can make mistakes and they enable to big problems, in TDD, we can obviate from that problem because we can catch the details with writing test.

TDD comes from Waterfall Model Project Management and Waterfall Model transformed to Test-Driven-Development.



As seen above, according to TDD, we write test before code and it is enough to pass the test for next step.

Consequently, Test-Driven Development is a development method in which the test is written first and then the production code.

Spring Profile

Spring Profiles provide to map our beans to different profiles. Here is an example. In this example, there are two profiles, dev and prod, and depending on the profile activated there will be a different level of 'logging'. Each concrete

LoggingAspect class contains an @Profile annotation. This can specify one or more profiles.

```
public abstract class LoggingAspect {

    @Pointcut("execution(* com.city81.service.CustomerService.processCustomer(..))'
    public void processCustomer() {
    }

}


@Configuration
@Profile("prod")
@Aspect
public class ConciseLoggingAspect extends LoggingAspect {

    @Before("processCustomer()")
    public void logCustomerId(JoinPoint jp) {
        Customer customer = (Customer) jp.getArgs()[0];
        System.out.println("id = " + customer.getId());
    }

}


@Configuration
@Profile("dev")
@Aspect
public class VerboseLoggingAspect extends LoggingAspect {

    @Before("processCustomer()")
    public void logCustomerAll(JoinPoint jp) {
        Customer customer = (Customer) jp.getArgs()[0];
        System.out.println("id = " + customer.getId());
        System.out.println("name = " + customer.getName());
        System.out.println("age = " + customer.getAge());
        System.out.println("address = " + customer.getAddress());
    }

}
```

Mockito

Mockito is a library that enables to mock our objects. I want to explain mocking objects with an example. We have a class that its name is UserDao iand also have LoginService. When we want to write a test for LoginService, we need to depend UserDao. We can use mocking objects, without using UserDao but

mocking UserDao and also we can specify attributes of UserDao with this way. We need to use @Mock and @InjectMocks annotations.

We can have a findUserByName method in UserDao but we want to appropriate that method. We want to use that method with parameters that we want to send to that method. We can make it with @Mock and @InjectMocks annotations easily. We can appropriate methods, not override but mocking, that method seems like same as real method itself.

```
@Mock
Map<String, String> wordMap;

@InjectMocks
MyDictionary dic = new MyDictionary();

@Test
public void whenUseInjectMocksAnnotation_thenCorrect() {
    Mockito.when(wordMap.get("aWord")).thenReturn("aMeaning");

    assertEquals("aMeaning", dic.getMeaning("aWord"));
}
```

```
public class MyDictionary {
    Map<String, String> wordMap;

    public MyDictionary() {
        wordMap = new HashMap<String, String>();
    }
    public void add(final String word, final String meaning) {
        wordMap.put(word, meaning);
    }
    public String getMeaning(final String word) {
        return wordMap.get(word);
    }
}
```

JUnit

JUnit is an open source unit testing framework for Java. It is indispensable for Test-Driven-Development. JUnit provides annotations to identify test methods, write code faster, increases quality of code. Developers spend more time in reading than writing and JUnit bridges that duration and help to fix bugs early in cycle of development. Unit tests help with code re-use.

Unit Test has two types as Manual and Automated. Commonly automated but can still be performed manually. Unit Test has advantages and disadvantages. Allows the developer to refactor code at a later date and make sure the Project still works correctly. Fort he modular build of unit testing, we cantest parts of

the project without waiting for other modules to be completed. Unit testing can’ t be expected to catch every error in the program.

When you want to create a test class, you need to add @Test annotation. Use an assert method provided by JUnit or another framework, to check the results. You should provide meaningful messages in asserts. It makes easy to understand and fix problems. Here is a basic exapmle of JUnit Test.

```
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

public class Temp {

   @BeforeClass
   public static void staticInit(){
       System.out.println( "Static Init" );
   }

   @Before
   public void testInit(){
       System.out.println( "Test Init - " + this );
   }

   @Test
   public void testOne(){
       System.out.println( "Test One - " + this );
   }

   @Test
   public void testTwo(){
       System.out.println( "Test Two - " + this );
   }

   @After
   public void testCleanUp(){
       System.out.println( "Test Clean Up - " + this );
   }

   @AfterClass
   public static void staticCleanUp(){
       System.out.println( "Static Clean Up" );
   }

}
```

Şeyma Tezcan