TDD & BDD

Lesson1: Introduction to Test Driven Development

Lesson Objectives

- What is TDD?
- TDD and traditional testing
- TDD and V-model
- Why TDD?
- TDD team overview
- Myths and misconceptions
- Tools
- Example of TDD

1.1 What is Test Driven Development (TDD) ?

- TDD is a technique whereby you write your test cases before you write any implementation code
- Tests drive or dictate the code that is developed
- An indication of “intent”
- Tests provide a specification of “what” a piece of code actually does
- Some might argue that “tests are part of the documentation

> “Before you write code, think about what it will do. Write a test that will use the methods you haven’t even written yet.”

1.1 What is Test Driven Development?

- TDD is done at Unit level - i.e. testing the internals of a class
- Tests are written for every function
- Mostly written by developers using one of the tool specific to the application

1.1 Levels of Test Driven Development

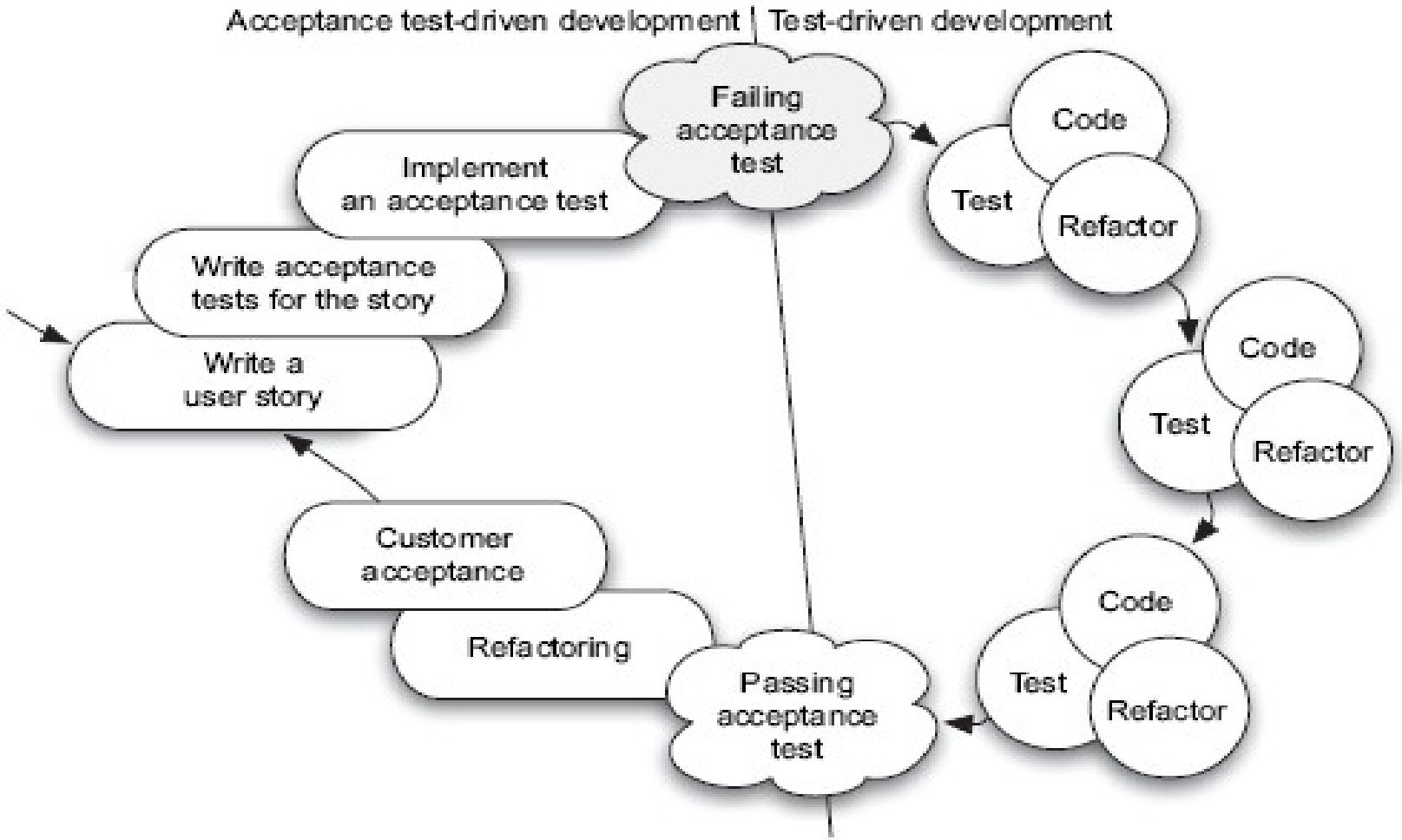
- There are two levels of TDD:
  - Acceptance TDD (ATDD)
    - ATDD is also called Behavior Driven Development (BDD)
    - tools such as FitNesse, RSpec ,JBehave, Easyb are used
  - Developer TDD
    - Developer TDD is often simply called TDD
    - xUnit family of open source tools are used

- TDD is a highly effective development strategy that helps developers write code more accurately and precisely.

- The low-level requirements used to drive TDD are directly derived from the high-level acceptance tests, so the two techniques complement each other : automated acceptance tests describe the high level business objectives, while TDD helps developers implement them as requirements.

Relationship between ATDD & TDD

1.1 Levels of Test Driven Development



1.1 Levels of Test Driven Development

- Acceptance Tests vs Acceptance TDD (ATDD)
  - **Acceptance tests** - a set of tests that must pass before an application can be considered finished
  - testers will prepare test plans and execute tests manually at the end of the software development phase
  - Acceptance testing is done relatively independent of development activities

- **Drawbacks of Acceptance tests** : Problems in testing an application after it has been developed :
  - Having feedback about problems raised at a late stage of development makes it very difficult to correct bugs of any size
  - Costly rework
  - Wasted developer time
  - Delayed deliveries.

1.1 Levels of Test Driven Development

- Acceptance Tests vs Acceptance TDD (ATDD)
  - **Acceptance TDD** - collaboratively define and automate the acceptance tests for upcoming work before it even begins
  - Rather than validating what has been developed at the end of the development process, ATDD actively pilots the project from the start
  - It is not an activity reserved for the QA team
  - rather than just testing the finished product, ATDD helps to ensure that all project members understand precisely what needs to be done, even before the programming starts

- Advantages of ATDD
  - Issues are raised faster
  - Fixed more quickly
  - Less expensive
  - Team can respond to a change faster and more effeciently

1.1 Levels of Test Driven Development

- Acceptance TDD (ATDD).
  - write a single acceptance test, or behavioral specification depending on preferred terminology,
    - Acceptance tests are specifications for the desired behavior and functionality of a system
  - write enough production functionality/code to fulfill that test
  - The goal of ATDD is to specify detailed, executable requirements for your solution on a just in time (JIT) basis

- Acceptance tests are:
  - Owned by the customer
  - Written together with the customer, developer, and tester
  - About the what and not the how
  - Expressed in the language of the problem domain
  - Concise, precise, and unambiguous

1.1Levels of Test Driven Development

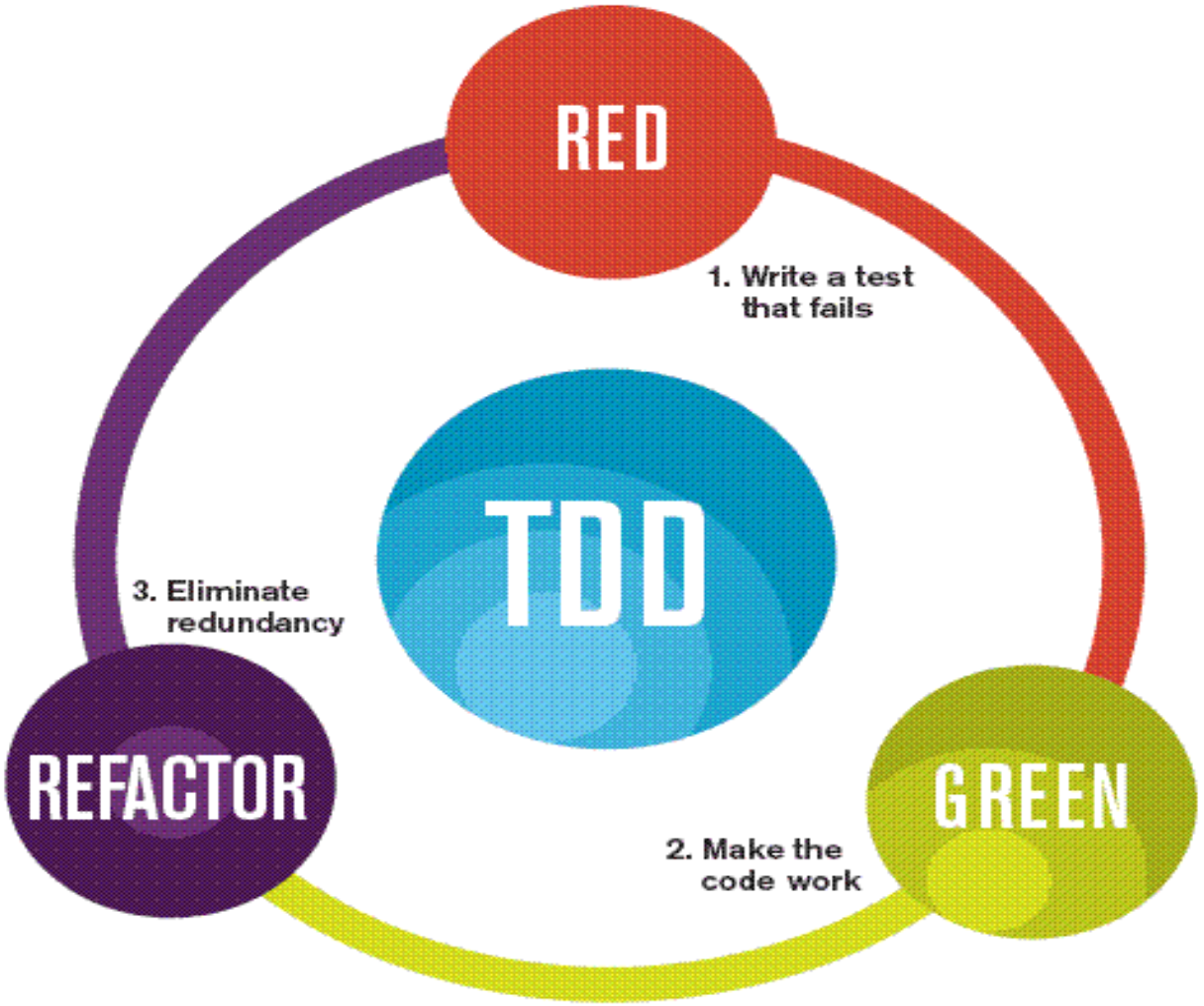
- Acceptance TDD (ATDD).
  - Describe a lightweight and extremely flexible requirements format called user stories
  - User Stories have proven to be an effective mechanism for ATDD
  - Plain English constructs "Given, When, Then" are usedto build automated acceptance tests
- User stories are created for each behavior that is expected from the system
- Create acceptance tests that assert the user stories in an acceptance tests project
- Tools like Selenium help us to build effective Acceptance Tests based on user stories

1.1 Levels of Test Driven Development?

- Developer TDD.
  - Write a single developer test, (referred to as a unit test),
  - Write enough production code to fulfill that test.
  - The goal of developer TDD is to specify a detailed, executable design for your solution on a JIT basis
  - Developer Tests
    - Run fast (they have short setups, run times, and break downs)
    - Run in isolation (you should be able to reorder them)
    - Use data that makes them easy to read and to understand
    - Use real data (e.g. copies of production data) when they need to
    - Represent one step towards your overall goal

1.1 What is Test Driven Development?

- Red
- Green
- Refactor



1.1 Steps in Test Driven Development

- **Red:** Create a test and make it fail
  - Imagine how the new code should be called and write the test as if the code already existed
  - Create the new production code stub.
  - Write just enough code so that it compiles
  - Run the test. It should fail. This is a calibration measure to ensure that your test is calling the correct code and that the code is not working by accident. This is a meaningful failure, and you expect it to fail

- **Green:** Make the test pass by any means necessary
  - Write the production code to make the test pass. Keep it simple
  - If you've written the code so that the test passes as intended, you are finished. You do not have to write more code speculatively. The test is the objective definition of "done. If new functionality is still needed, then another test is needed. Make this one test pass and continue.
  - When the test passes, you might want to run all tests up to this point to build confidence that everything else is still working

1.1 Steps in Test Driven Development

- **Refactor:** Change the code to remove duplication in your project and to improve the design while ensuring that all tests still pass
  - Remove duplication caused by the addition of the new functionality
  - Make design changes to improve the overall solution
  - After each refactoring, rerun all the tests to ensure that they all still pass

- Repeat the cycle. Each cycle should be very short, and a typical hour should contain many Red/Green/Refactor cycles

1.1 What is Test Driven Development?

- Scenario 1:
  - Consider a Banking application that has several modules for the various banking activities
  - The application has the following functions that returns true or false depending on the success/failure of the operation
    - Boolean Withdraw(float amt)
    - Boolean Deposit (float amt)

1.1 What is Test Driven Development?

In Test Driven Development, the following activities are done.

- The behavior expected from these functions are listed
- Tests are written to ensure that the function behaves as expected
- The tests are executed. (Tests fail as the function is not yet defined)

- The functions are written
- The tests are rerun, the test should pass.
- If it doesn't pass, there is something wrong, fix it now.

- Refactor the code
- Run the test again to ensure all test pass.
- Repeat the steps above until you can't find any more tests that drive writing new code.

1.1 What is Test Driven Development?

Applying TDD to Scenario 1

- The Withdraw function is expected to behave as follows
  - When the account is in blocked state the withdraw function should return false and show a message "Withdrawal not permitted"
  - When the Account balance < Amount to withdraw, the withdraw function should return false and show a message "Withdrawal not permitted"
  - When the Account balance > Amount to withdraw, the withdraw function should return true and Balance should be updated
- TDD enforces that 3 or more tests are written to ensure that the function withdraw behaves as mentioned in the above cases
- After the tests are ready the actual withdraw function is written
- Now on executing the tests all tests pass if the withdraw function behaves as expected

1.2 TDD vs Traditional Testing

- In traditional testing
  - A successful test finds one or more defect
  - Testing happens as an isolated activity either with or after development
  - the greater the risk profile of the system the more thorough your tests need to be
  - 100% coverage test is not achieved– every single line of code is not tested

- In TDD
  - with TDD; when a test fails you have made progress because you now know that you need to resolve the problem
  - Testing happens even before any coding begins
  - The more thorough the tests are the less is the risk profile of the application modules
  - 100% coverage test is achieved– every single line of code is tested

1.2 TDD vs Traditional Testing

- Unit tests in Traditional testing

  - TDD tests are very similar to unit tests, since you use a unit testing framework such as JUnit or NUnit to create both types of tests
  - The purpose of a unit test is to test a unit of code in isolation
  - For example, you might create a unit test that verifies whether a particular class method returns the value that you expect
  - Data access logic is not tested in Unit testing
  - If the code that is covered by unit tests is modified, the tests can be used to immediately determine whether the existing functionality is broken

1.2 TDD vs Traditional Testing

- Unit tests in Traditional testing
  - Example :the Add() method of a Math class

```
public class Math
{
        public int Add(int val1, int val2)
       {
        return val1 * val2;
        }
}
```

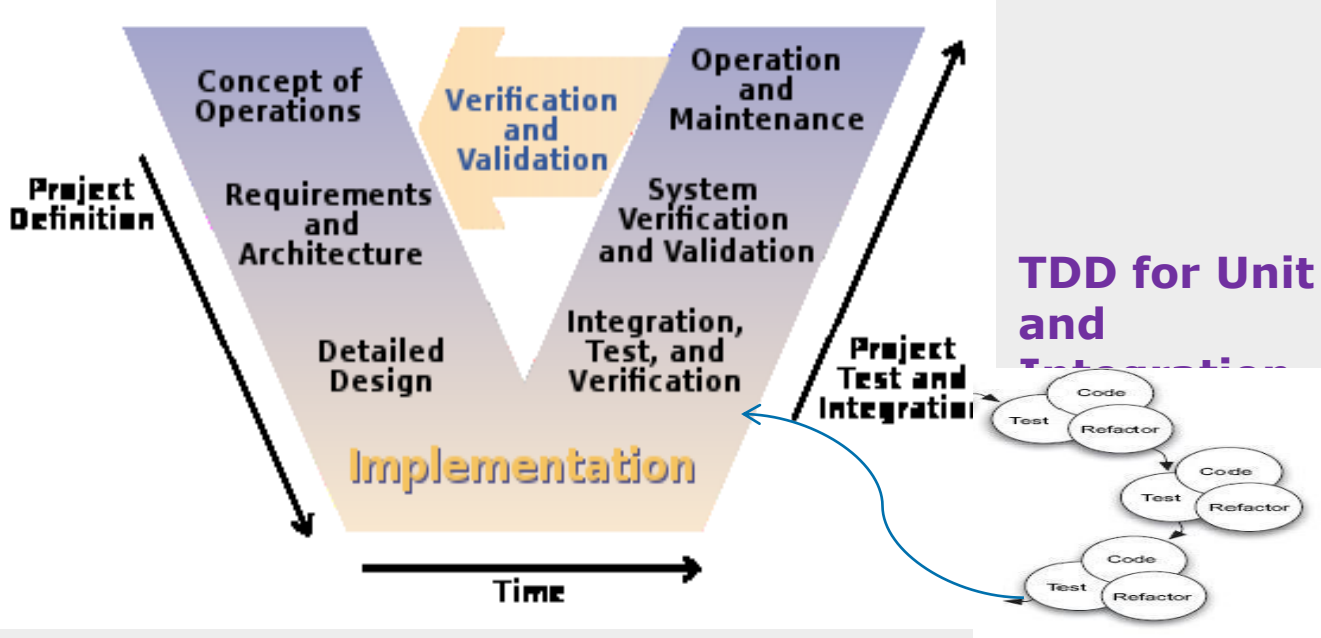
- A unit test is one that verifies if the function is called as Add(6,4) the output received is 10

1.2 TDD vs Traditional Testing

- TDD differs from Unit testing as
  - Unlike a unit test, a TDD test is used to drive the design of an application
  - A TDD test is used to express what application code should do before the application code is actually written
  - Test-Driven Development grew out of a reaction to waterfall development
  - One important goal of Test-Driven Development is incremental design and evolutionary design
  - Instead of designing an application all at once and up front, an application is designed incrementally test-by-test

1.3 TDD and V-model

- TDD can be applied to V-model
  - TDD can be applied to unit testing phase and even on integration testing phase
  - But system testing occurs after the development of the product
  - Hence System testing is not suitable for TDD



1.7 Why TDD?

The major advantages of Test Driven Development are:

- Significantly decreased regressions
  - Tells you whether your last change (or refactoring) has broken previously working code
  - Less time spent in the debugger and when it is required you usually get closer to problem quickly
- Refactoring becomes easier (and large, sweeping refactoring becomes possible)
  - The time in rework is greatly reduced

1.7 Why TDD?

The major advantages of Test Driven Development are (contd…) :

- Interfaces are better-designed
  - Allows the design to evolve and adapt to your changing understanding of the problem.
- The resulting implementation tends to be simpler
- Short feedback loop
- Creates a detailed specification

1.7 Why TDD?

The Projects that implement TDD are found to:

- More Stable
  - If your tests are good, then your code will be more stable.
- More Accountable
  - When you boss asks, "how are things coming?", you can reply, "72% of the tests pass."
  - No more replying with vague answers and trying not to commit to anything.
- Separated Concerns
  - Good layered, encapsulated, modular code is the goal of any architect, and TDD, forces to separate concerns to test them separately.
  - Mocking helps to understand the boundaries of the encapsulation of my classes
  - If you are finding it very hard to test something without mocking up 5 different objects, you may need to rethink your design.

1.7 Why TDD?

Benefits of TDD in various phases of the project:

- Requirements gathering
  - ATDD clearly captures the business requirements
  - Hence understanding of requirements increases by 90% and takes 50% lesser time
- Design
  - TDD when used with AMDD produces a perfect design
- Coding
  - Incremental development is enabled by TDD and hence helps create loosely coupled code
  - Code developed by TDD is observed that the minimum quality increased linearly with the number of programmer tests
- Functional verification and regression tests
  - Approximately 40% fewer defects than a baseline prior product developed in a more traditional fashion

1.7 Why TDD?

Success stories:

- A TDD team at a large insurance company delivered a ~100,000-line Java application against which only 15 defects were reported in its first 11 months
- One team was able to reduce their code base from 180,000 to 60,000 lines of code in 10 months by test-driving new features and incrementally adding tests to existing code
- Using TDD, the theoretical outcome should be 100% coverage. In practice the coverage levels are typically in the high-90% range
- A study by Maximilien and Williams in 2008 shows that TDD has reduced defect by 40-50% in 90% of the projects

1.8 TDD team overview

- The TDD involves a series of tasks that are done by various members of the team

- The key players involved in TDD are
  - The customer
  - The Manager
  - The Architect or Senior developer
  - Engineers

1.8 TDD team overview - Roles of TDD team members

- The customer
  - The customers define the roadmap, the key scenario and functionality
- The Manager
  - The manager defines the initial tasks and features, gathers the necessary resources(hardware, tools) and forms the cross-functional team
- The Architect or Senior developer
  - Creates Use cases and interfaces that fulfill the use cases
  - The models required in AMDD are created by the architect
- Engineers
  - Create Unit test drivers to test the feature
  - Implement the first version which is tested against the test driver
  - Refines the implementation and incrementally re-tests the same
  - Upgrades the original version by refactoring

1.9 Myths and Misconceptions

- You create a 100% regression test suite
  - Though black-box tests can be created which validate the interface of the component these tests won't completely validate the component
  - The user interface is really hard to test. Although user interface testing tools do in fact exist, not everyone owns them and sometimes they are difficult to use
  - Some developers on the team may not have adequate testing skills
- The unit tests form 100% of your design specification
  - The reality is that the unit test form a fair bit of the design specification, similarly acceptance tests form a fair bit of your requirements specification, but there's more to it than this
- You only need to unit test
  - Only for simplest systems this statement is true. Only 5% of the systems are simple
  - For all other systems a host of other testing techniques are required

1.9 Myths and Misconceptions

- TDD is sufficient for testing
  - TDD, at the unit/developer test as well as at the customer test level, is only part of your overall testing efforts
  - At best it comprises your confirmatory testing efforts, but you must also be concerned about independent testing efforts which go beyond this
- TDD doesn’t scale
  - There are multiple scalability issues with TDD, hence the above statement is partly true
  - The scalability issues with TDD are
  - Your test suite takes too long to run
  - Not all developers know how to test
  - Everyone might not be taking a TDD approach

1.10 Tools

- The following tools are available for Unit testing. Popularly they are referred as xUnit framwork

  - cpputest
  - csUnit (.Net)
  - CUnit
  - DUnit (Delphi)
  - DBFit
  - DBUnit
  - DocTest (Python)
  - Googletest
  - HTMLUnit
  - HTTPUnit
  - JMock
  - JUnit
  - NDbUnit
  - NUnit
  - OUnit
  - PHPUnit
  - PyUnit (Python)
  - SimpleTest
  - TestNG
  - VBUnit
  - XTUnit
  - xUnit.net

1.11 Example of TDD

- Example of how to create unit tests up front for a simple method :
  - Consider a method that takes an integer and turns it into words, in English
  - So if the input is the integer 1, the result will be "one". If the input is 23 the result will be "twenty three" and so on
  - The code is written in Java and the unit tests are written in JUnit
  - Similar test cases can be written in NUnit for testing the functionality of a dot net application

1.11 Example of TDD

- The stub looks like this

```
public static String NumberToEnglish(int p)
 { throw new Exception("The method
```

- The test for the above method is

```
@Test
public void NumberToEnglishShouldReturnOne()
 {
String actual = English.NumberToEnglish(1);
assetrtEquals("one", actual, "Expected the result to be
    \"one\"");
}
```

- The test should fail because the stub throws an exception, rather than do what the test expects.

1.11 Example of TDD

- The next thing to do is to ensure that the code satisfies the demands of the unit test.
- Agile methodologies, such as XP, suggests that only the simplest change should be made to satisfy the current requirements
- In that case the method being tested will be changed to look like this:

```
public static String NumberToEnglish(int p)
 {
    return “one”;
}
```

- At this point the unit tests are rerun and they pass and hence work

1.11 Example of TDD

- Test "two"
- Since the overall requirement is that any integer should be translatable into words, the next test should test that 2 can be translated
- The test looks like this:

```
@Test
 public void NumberToEnglishShouldReturnTwo()
{
string actual = English.NumberToEnglish(2);
assertEquals("two", actual, "Expected the result to be
\"two\"");
 }
```

- However, since the method being tested returns "one" regardless of input at this point the test fails

1.11 Example of TDD

- Again keeping the simplest change principle the code is updated to look like this

```
public static String NumberToEnglish(int p)
 {
          if (number == 1)
                    return "one";
          else
                     return "two";
}
```

- Now the tests pass

1.11 Example of TDD

- Test “three” to “twenty"
  - A third test can now be written. It tests for an input of 3 and an expected return of "three". Naturally, at this point, the test fails. The code is updated again to make this test pass
  - To cut things short, the new tests and counter-updates continue like this until the numbers 1 to 20 can be handled. The code will eventually look like the one below

```
public static string NumberToEnglish(int number)  {
        switch (number)  {
                        case 1: return "one";
                        case 2: return "two";
                        case 3: return "three";
                        case 4: return "four";
                        case 5: return "five";
```

1.11 Example of TDD

```
case 6: return "six";
case 7: return "seven";
case 8: return "eight";
case 9: return "nine";
case 10: return "ten";
case 11: return "eleven";
case 12: return "twelve";
case 13: return "thirteen";
case 14: return "fourteen";
case 15: return "fifteen";
case 16: return "sixteen";
case 17: return "seventeen";
case 18: return "eighteen";
case 19: return "nineteen";
default: return "twenty";
 } }
```

1.11 Example of TDD

- Test "twenty one" to "twenty nine"
- At this point it looks like it will be pretty easy to do 21, but a pattern is about to emerge. After the tests for 21 and 22 have been written, the code is refactored to look like this:

```
public static string NumberToEnglish(int number)
 {
 if (number < 20)
            return TranslateOneToNineteen(number);
 if (number == 20)
            return "twenty";
return string.Concat("twenty ", TranslateOneToNineteen(number - 20));
}

private static string TranslateOneToNineteen(int number)
 {
switch (number)
```

1.11 Example of TDD

```
 {
 case 1: return "one";
 case 2: return "two";
…
case 16: return "sixteen";
case 17: return "seventeen";
case 18: return "eighteen";
default: return "nineteen";
}
}
```

- Now all the tests from 1 to 22 pass. 23 to 29 can be assumed to work because it is using well tested logic

- Similarly the TDD continues and the code is developed to work for all integers

Summary

What is Test Driven Development?

- It is a methodology
- It replaces traditional testing
- It does testing for every unit but before units are ready
- Acceptance TDD tests the behavior
- ATDD is the driver for individual TDD

Review Question

- **Question 1 :**Test Driven Development has two levels _____ and _____.

- **Question 2 :** TDD can be used to test methods that connect to database.

  True/False

- **Question 3:** TDD requires that every test passes in the first execution itself.

  True/False

- **Question 4:** _____ are essentially features in software development that enable users to be more efficient.