

Test Driven Development

Lesson1: Introduction to Test
Driven Development &
Behaviour Driven Development

Lesson Objectives

- What is TDD?
- TDD and traditional testing
- TDD and V-model
- TDD and documentation
- Test-driven database development
- Scaling TDD via Agile Model-Driven Development (AMDD)
- Why TDD?
- TDD team overview
- Myths and misconceptions
- Tools
- Example of TDD



Lesson Objectives

- Introduction to Behaviour Driven Development
- What is Behaviour Driven Development?
- Waterfall To Agile Behavior Driven Methodologies
- The BDD Approach
- The BDD Approach – Key Elements
- Tools support for BDD
- Key Benefits of BDD
- TDD Vs. BDD



1.1 What is Test Driven Development?

- 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?

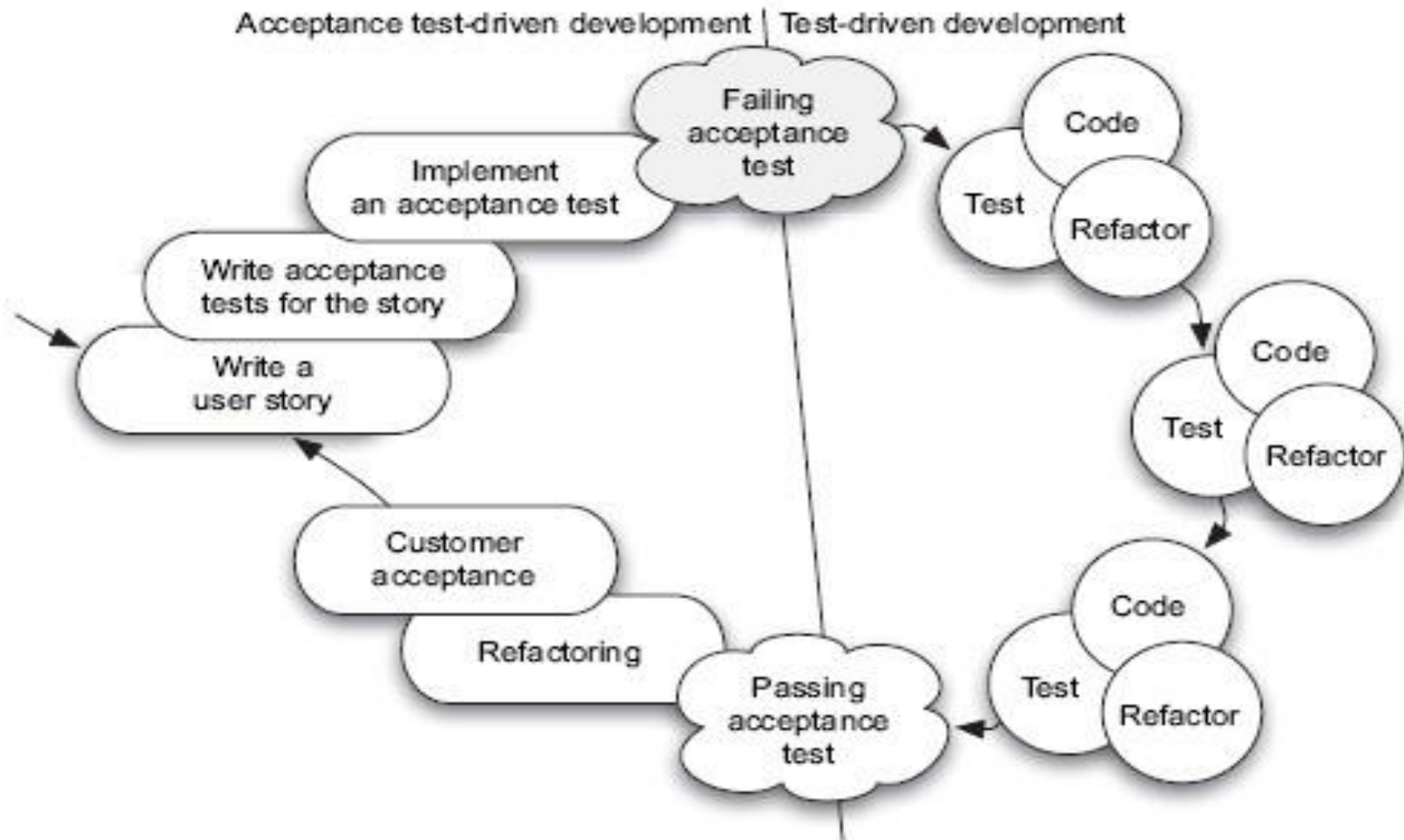
- TDD is done at Unit level - i.e. testing the internals of a class
- Tests are written for every function
- Mostly written by developers using one of the tool specific to the application

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, or Test-Driven Development, 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 and TDD

1.1 Levels of Test Driven Development?



Acceptance TDD

1.1 Levels of Test Driven Development?

- Acceptance Test 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
- 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.

Acceptance TDD

1.1 Levels of Test Driven Development?

- Acceptance Test 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

Acceptance TDD

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

Acceptance TDD

1.1 Levels 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 used to 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

Acceptance TDD

1.1 Levels of Test Driven Development?

- A sample User Story

1.1

Given a user is not logged in

When any page is hit

Then the user should be asked to authenticate

1.2

Given a user with admin role is logged in When an admin page is hit

Then the user should see the page

1.3

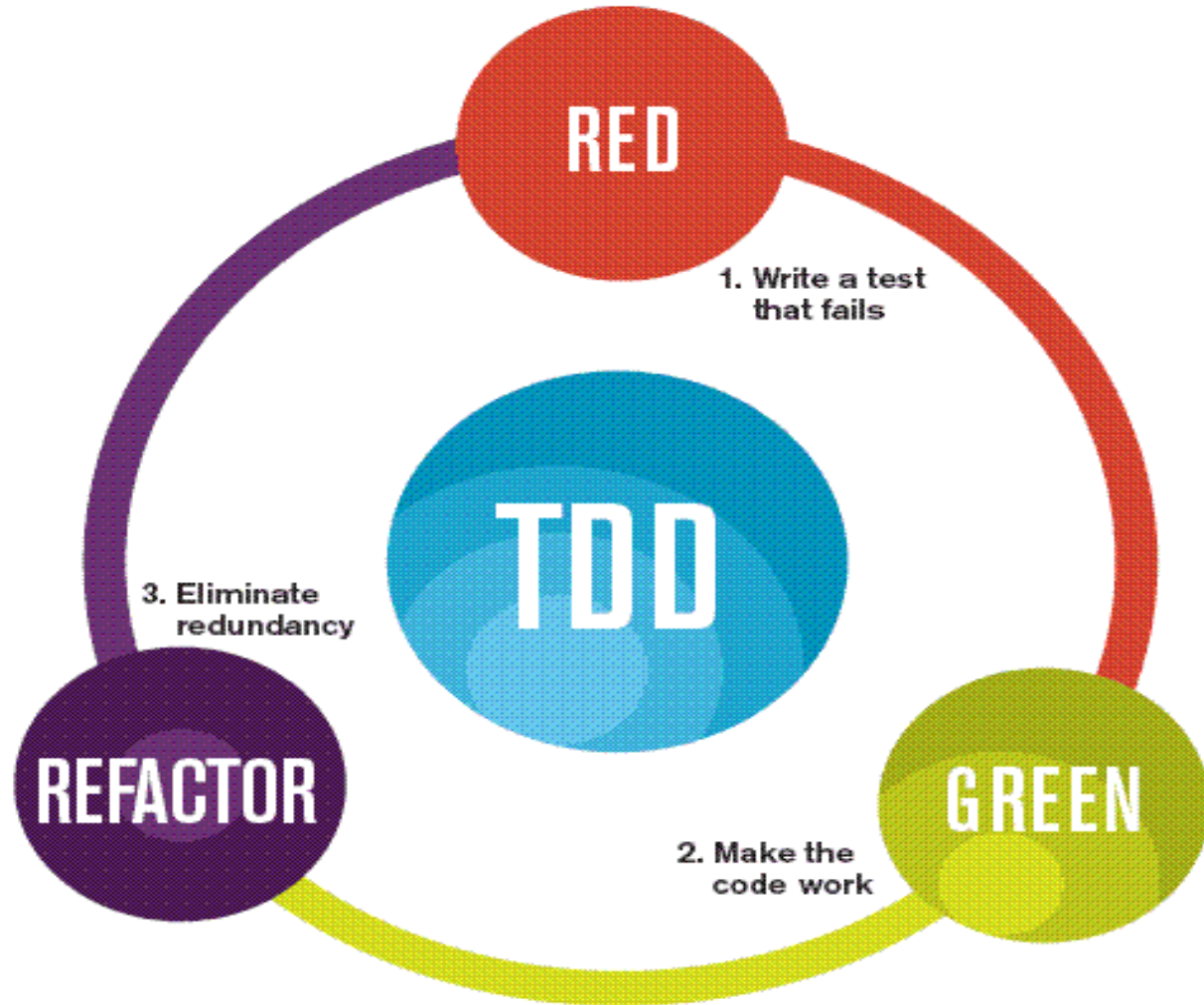
Given a user with just regular user role is logged in When an admin page is hit Then the user should see an error page

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 are
 - 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?


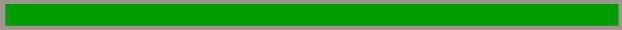

- Refactor: Change the code to remove duplication in your project and to improve the design while ensuring that all tests still pass
 - Remove duplication caused by the addition of the new functionality
 - Make design changes to improve the overall solution
 - After each refactoring, rerun all the tests 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 the withdraw function should return false and show a message "Withdrawal not permitted"
 - When the Account balance > Amount to withdraw, the 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

Unit Test vs TDD

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

Unit Test vs TDD

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

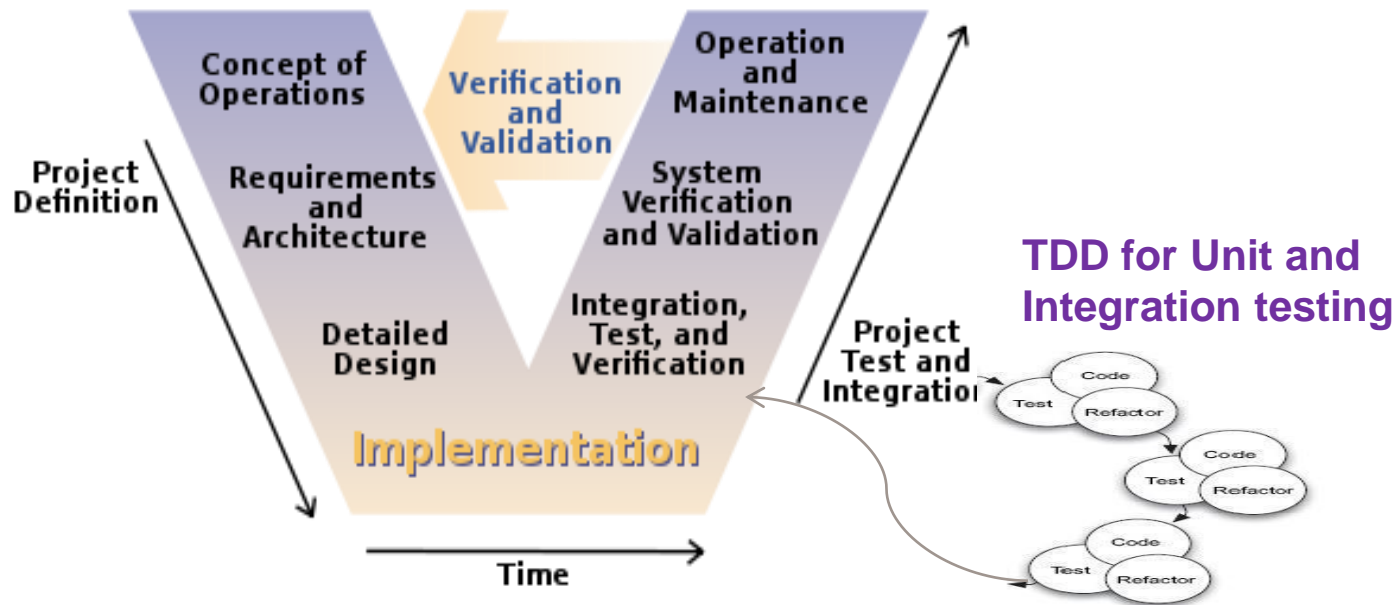
Unit Test vs TDD

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

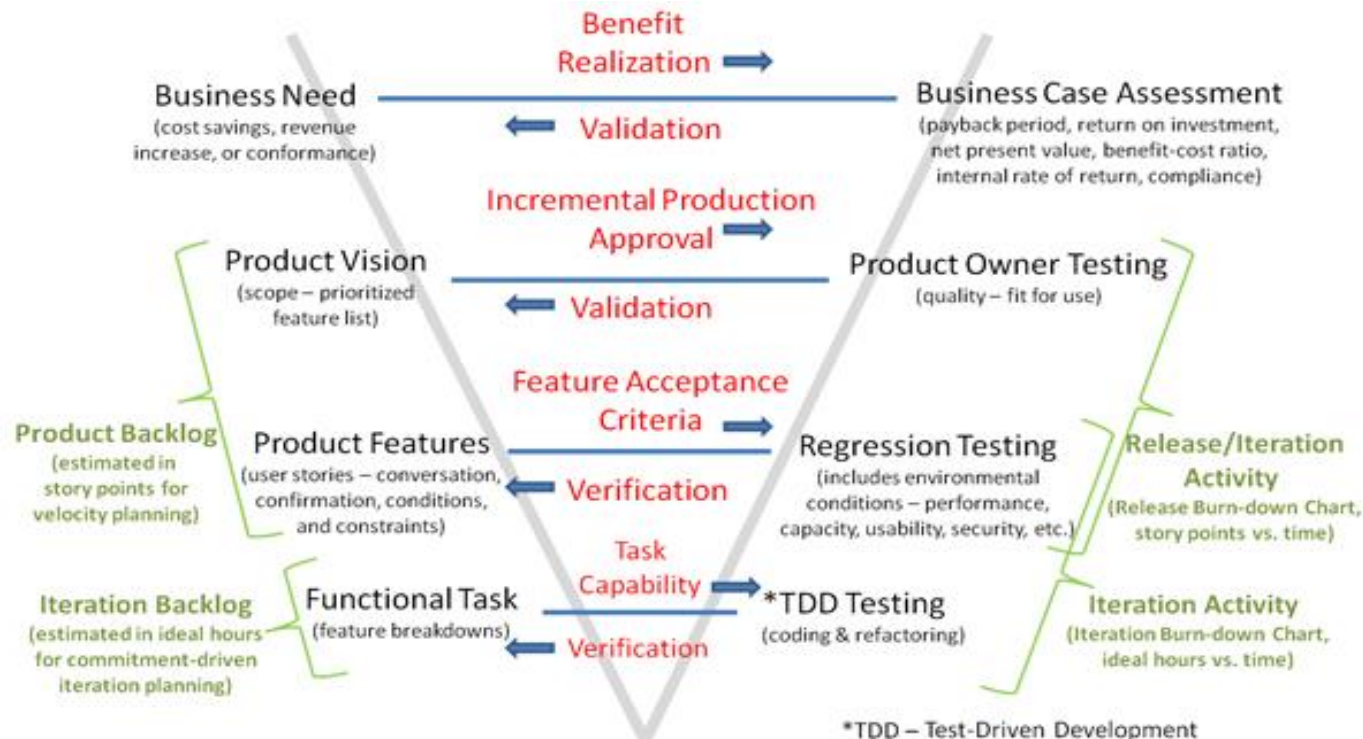
1.3 TDD and V-model

- 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



1.3 TDD and V-model

- V model for Agile development
 - TDD fits well into the agile development methodology.
 - The V model when applied to Agile development is as shown below



1.4 TDD and Documentation

- Test suites for a part of documentation:
- Unit tests created in TDD become a significant portion of technical documentation
 - Well written unit tests provide a working specification of the functional code
 - All the functionalities expected from a code are documented well before the development
- Acceptance tests form an important part of requirement documentation
 - All the user requirements are captured well in Acceptance tests and hence they can be viewed as a document that says about the critical requirements, the missing functionality etc.

1.5 Test -Driven Database Development

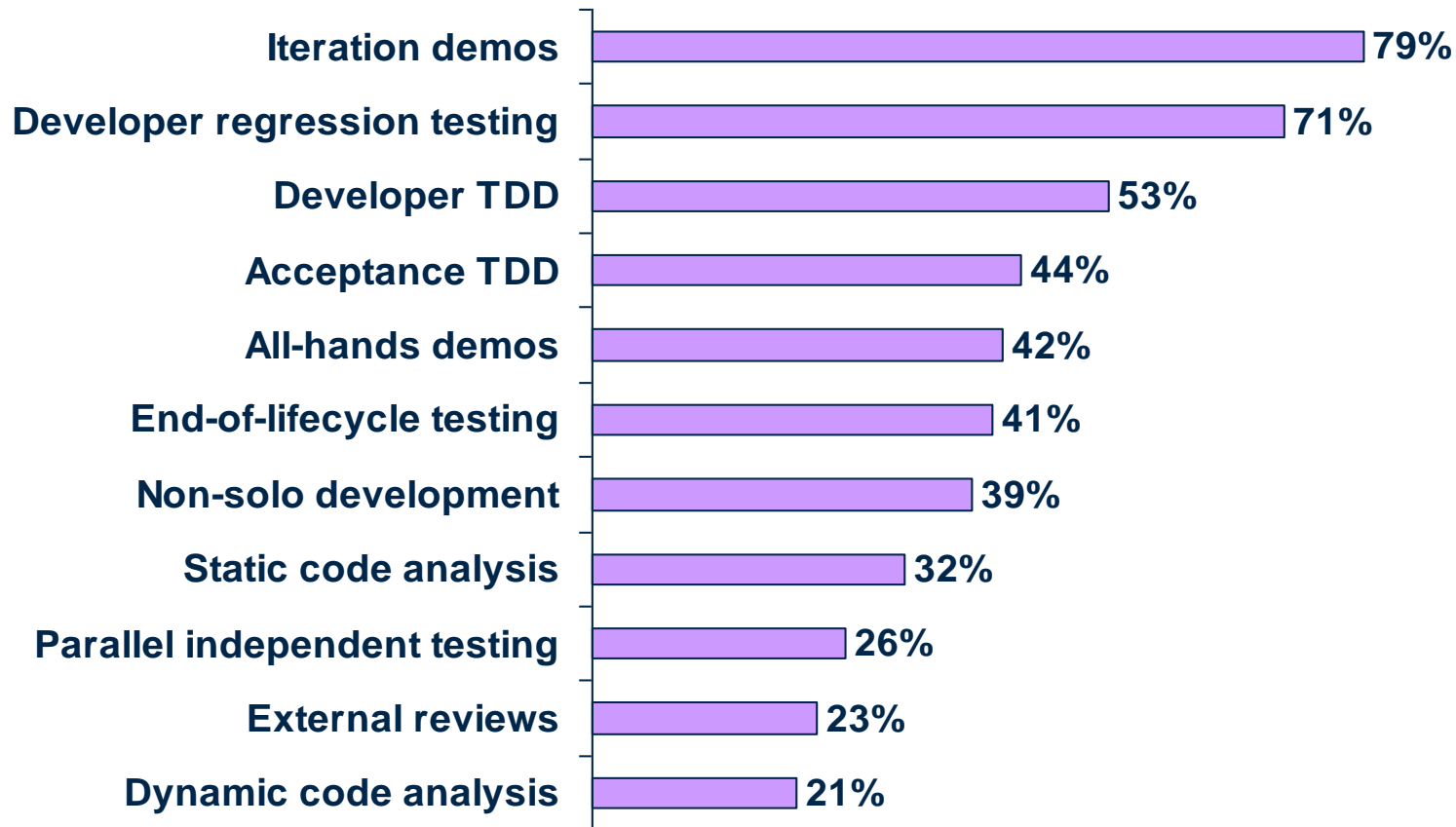
- Test Driven development for databases:
 - Database is not a special case for testing, as it's only a part of your application
 - It enables to ensure the quality of data and validate the functionality implemented within the database
 - To apply TDD to database applications we need database equivalents of regression tests, refactoring and continuous integration
 - There are two categories of database tests
 - Interface tests
 - Internal database tests

1.5 Test -Driven Database Development

- Test Driven development for databases:
 - Interface tests
 - They validate what data is flowing in, going out and getting mapped to the database
 - Ex. Admin can access all tables whereas user can access only few tables
 - The test cases are written to check what would be the different access rights on these tables
- Internal database tests
 - They validate the internal structure, behavior and content of database
 - The basic CRUD operations on a table are tested
 - There are various Mock objects available to simulate database connectivity
 - Use of embedded lightweight database like Hypersonic SQL can make Test driven Database development easier
 - Tool like DBUnit are emerging with features to test various database features

1.6 The significant role of TDD in Agile projects

- Agile teams validate their work with the below techniques



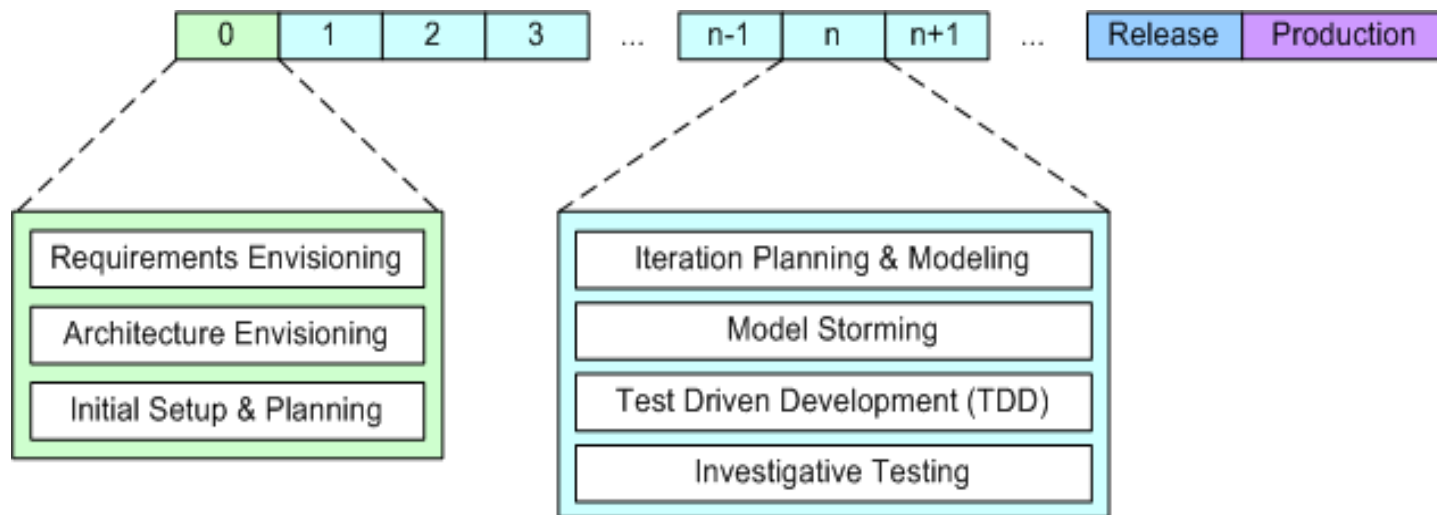
1.6 Scaling TDD via Agile Model-Driven Development (AMDD)

■ AMDD:

- AMDD is the agile version of Model Driven Development (MDD)
- MDD is an approach to software development where extensive models are created before source code is written
- The difference with AMDD is that instead of creating extensive models before writing source code , create agile models which are just barely good enough that drive the overall development efforts
- AMDD addresses the agile scaling issues that TDD does not
- TDD is very good at detailed specification and validation, but not so good at thinking through bigger issues such as the overall design, how people will use the system, or the UI design
- Hence AMDD can be used for this purpose

1.6 Scaling TDD via Agile Model-Driven Development (AMDD)

- The AMDD Lifecycle consists of
 - Envisioning (Iteration 0)
 - Includes two main sub-activities, initial requirements envisioning and initial architecture envisioning.
 - Iteration 1 to Iteration n
 - Includes iteration modeling, model storming, reviews, and implementation



Development (AMDD)

- Combining TDD with AMDD:
 - AMDD should be used to create models with project stakeholders to help explore their requirements and then to explore those requirements sufficiently in architectural and design models
 - TDD should be used as a critical part of build efforts to ensure that you develop clean, working code
 - The end result of combining both TDD and AMDD is a high-quality, working system that meets the actual needs of your project stakeholders

Development (AMDD)

- Which approach should you take?
 - Some people are primarily "visual thinkers", also called spatial thinkers, and they may prefer to think things through via drawing
 - Other people are primarily text oriented, non-visual or non-spatial thinkers, who don't work well with drawings and therefore they may prefer a TDD approach
 - Of course most people land somewhere in the middle of these two extremes and as a result they prefer to use each technique when it makes the most sense
 - In short, the answer is to use the two techniques together so as to gain the advantages of both

Development (AMDD)

- Comparing TDD with AMDD:
 - TDD shortens the programming feedback loop whereas AMDD shortens the modeling feedback loop
 - TDD provides detailed specification (tests) whereas AMDD is better for thinking through bigger issues
 - TDD promotes the development of high-quality code whereas AMDD promotes high-quality communication with your stakeholders and other developers
 - TDD provides concrete evidence that the software works whereas AMDD supports your team, including stakeholders, in working toward a common understanding
 - TDD “speaks” to programmers whereas AMDD speaks to business analysts, stakeholders, and data professionals

Development (AMDD)

- Comparing TDD with AMDD:
 - TDD provides very finely grained concrete feedback on the order of minutes whereas AMDD enables verbal feedback on the order minutes (concrete feedback requires developers to follow the practice Prove It With Code and thus becomes dependent on non-AM techniques)
 - TDD helps to ensure that your design is clean by focusing on creation of operations that are callable and testable whereas AMDD provides an opportunity to think through larger design/architectural issues before code
 - TDD is non-visually oriented whereas AMDD is visually oriented
 - Both techniques are new to traditional developers and therefore may be threatening to them
 - Both techniques support evolutionary development

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
 - 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
- VUnit
- XUnit
- 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);
    assertEquals("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";
      case 6: return "six";
      case 7: return "seven";
```

1.11 Example of TDD

```
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

1.12 Introduction to Behaviour Driven Development

- While launching a new project, there can be a disconnect between :
 - The business being accurately able to define the desired results
 - Sometimes the requirements are well expressed, but each team has developed their own internal language, making communication ineffective
 - At other hand non-technical roles such as product owners (POs) and business analysts, define the requirements, creating further challenges in understanding exactly what is required
 - These challenges are intensified when the teams involved are from different countries and cultures.
- Clear communication and understanding of requirements represent key elements in project success.
- Behaviour Driven Development (BDD) can help achieve all of the above and ultimately, helps a business and its technical team deliver software that fulfills business goals
- It has evolved out of established agile practices and is designed to make them more accessible and effective for teams new to agile software delivery
- Over time, BDD has grown to encompass the wider picture of agile analysis and automated acceptance testing

1.13 What is Behaviour Driven Development?

- In software engineering, behavior-driven development (BDD) is a software development process that emerged from test-driven development (TDD)
- The concept of BDD was introduced by Dan North, an experienced technology and organizational consultant based in London, as a means of clearing confusion between testers, developers and business people
- BDD is an approach to development that improves communication between business and technical teams to create software with business value
- During “Agile specifications, BDD and Testing eXchange” in November 2009 in London, Dan North gave the following definition of BDD:
 - BDD is a second-generation, outside-in, pull-based, multiple-stakeholder, multiple-scale, high-automation, agile methodology. It describes a cycle of
 - Interactions with well-defined outputs, resulting in the delivery of working, tested software that matters.

1.14 Waterfall To Agile Behavior Driven Methodologies

- One of the most common problems with traditional waterfall methodologies is its slow response to change
- It takes good amount of time to establish the well defined requirements, analysis, design, implementation and testing, to end up with a working product
- This ultimately results in missed deadlines, obsolete product deliveries, and increased costs
- Agile methodologies provides way to address or mitigate these project killers to ensure project success
- Agile methodologies deliver working software as quickly as possible by dividing the software building process into small and relatively short iterations or sprints (typically 2 or 3 weeks), and deliver a working product with a functional increments by the end of each iteration

1.14 Waterfall To Agile Behavior Driven Methodologies (C0nt.)

- Benefits of Agile Methodologies over Waterfall model :
 - Higher priority requirements delivered first
 - Stakeholders see faster ROI because higher priority requirements are delivered first
 - Frequent delivery ensures that stakeholders receive a working product with new features at the end of each sprint, compared with Waterfall projects where it happens at the end of the project cycle
 - The more frequent the product is delivered, the sooner stakeholders can provide feedback
 - Closely related to the higher frequency of delivery, Agile methodologies enable frequent change
 - The sooner feedback is given, the faster the team can react and adapt to change
- However despite these advantages of Agile over Waterfall, there can still be challenges
- What if the delivered product doesn't meet client needs?
- Spending one iteration delivering something that may not accomplish stakeholder's expectations is still expensive
- If the software does not do what it is supposed to do, it means there are communication issues to fix

1.15 The BDD Approach

- The BDD approach can largely be divided into two main parts
 - The practice of using examples written in ubiquitous language to illustrate behaviours as in how users will interact with the product
 - The practice of using those examples as the basis of automated tests as well as checking functionality for the user, this ensures the system works as defined by the business throughout the project lifetime

1.16 The BDD Approach – Key Elements

- Starting with a Goal
 - In BDD, projects are delivered in sets of capabilities
 - Capabilities are essentially features in software development that enable users to be more efficient
 - For example, automating the ordering process of an eCommerce store or automating warehouse management.
 - In order to meet business requirements, you need to understand exactly what you want the software to achieve from a business perspective
 - In BDD, this then becomes your business goal: the driving force behind the project
 - A good business goal is specific, measurable, attainable, relevant and time-bound
- Here are two examples:
- Bad example:
 - We are going to generate more revenue by making more sales
- Good example:
 - We are planning to achieve a 15% increase in revenue through our online channels in 6 months

1.16 The BDD Approach – Key Elements

■ Impact Mapping

- The increasingly popular Agile practice for managing the project roadmap and laying out all potential features is called Impact Mapping
- Impact Mapping is a technique that helps you to outline all of the alternative ways to reach your decided goal and works by analyzing users' behaviour
- We assume that software cannot achieve business goals on its own, the only way software can get us closer to the goal is by supporting particular human behaviours
- For example, an online shop doesn't bring you money on its own - people bring you money by purchasing products from it

1.16 The BDD Approach – Key Elements

- Value and Complexity Analysis
 - In collaboration-heavy Agile methodologies like BDD, it is very important to clearly distinguish and agree priorities
 - In Behaviour Driven Development this is done by using two techniques: value and complexity analysis
 - Value analysis helps us to quickly identify low-cost, high-value features in the backlog
 - Complexity analysis helps us to choose the right development and collaboration approach to the project as a whole, as well as each individual feature

1.16 The BDD Approach – Key Elements

■ Planning with Examples

- Developers often misunderstand what a customer actually wants or requires from the software product and customer often misunderstand what developers are actually capable of delivering the product features
- This all leads to a situation in the project where the technical team continuously delivers, but does not meet the needs of the customers and can miss a number of opportunities
- This was always a serious problem, but the growth of Agile has made it much more apparent
- Understanding through examples is one of the best learning practices we start with as to remove ambiguity and provide context to what we are trying to understand

1.16 The BDD Approach – Key Elements

- Planning with Examples
- Example
 - Consider that we have been given a task to add an "Include VAT and delivery cost to the total price of the customer basket" feature to an existing eCommerce project with following set of business rules:
 - VAT is 20%
 - Delivery cost for small basket i.e. less than 20000 is 3000
 - Delivery cost for small basket i.e. more than 20000 is 2000
- Ambiguities or questions unanswered in the rules :
 - These rules do not specify whether to add VAT before delivery cost to the basket total or after
 - How should you handle a situation where the basket delivery cost is exactly 20000?
 - What happens if there are multiple products in the basket?
- In addition to these three business rules, we can provide following set of very simple examples on how the application will behave with them
 - Given a product is priced at 20000, when I add it to the basket, then the total basket price should be 21000
 - Given a product is priced at 25000, when I add it to the basket, then the total basket price should be 32000
 - Given a product is priced at 20000, when I add it to the basket, then the total basket price should be 26000

1.16 The BDD Approach – Key Elements

- Usage-Centered Design
 - One of the biggest promises of BDD is to deliver software that matters
 - Usage-centered design is an approach to software development where every single functionality delivered by the software product primarily focuses on the intentions and patterns of users
 - Instead of jumping straight away into the implementation stage, we try to identify who the audience is for the feature

1.16 The BDD Approach – Key Elements

- Ubiquitous Language
 - Ubiquitous language is a language created by developers and the business in order to discuss features and talk effectively in examples
 - It is not a technical language, but it is not a business language either; it is a combination of both
 - The key in the two sides becoming effective for joint communication lies in their ability to accept, to a certain extent, the points of view and understanding of each other
 - Ubiquitous language is an integral part of BDD
 - It is almost impossible to be effective with examples without striving for a shared language and understanding
 - The most common and adopted format implemented by this language is Given-When-Then which is the format most used by the BDD community.
- The format is given below and this format is referred to as the Gherkin language:
 - "Given" describes the initial context for the example
 - "When" describes the action that the actor in the system or stakeholder performs
 - "Then" describes the expected outcome of that action

1.16 The BDD Approach – Key Elements

■ Story: Returns go to stock

In order to keep track of stock

As a store owner

I want to add items back to stock when they're returned.

Scenario 1: Refunded items should be returned to stock

Given that a customer previously bought a black sweater from me

And I have three black sweaters in stock.

When he returns the black sweater for a refund

Then I should have four black sweaters in stock.

Scenario 2: Replaced items should be returned to stock

Given that a customer previously bought a blue garment from me

And I have two blue garments in stock

And three black garments in stock.

When he returns the blue garment for a replacement in black

Then I should have three blue garments in stock

And two black garments in stock.

1.17 Tools support for BDD

- SpecFlow
- Nspec
- Jbehave
- Gradle
- Jenkins
- NBehave
- Cucumber
- Freshen
- Behat

1.18 Key Benefits of BDD

- All development work can be traced back directly to business objectives
- Software development meets user need, Satisfied users = good business
- Efficient prioritization - business-critical features are delivered first
- Specifications reside in plain text files, consisting a series of scenarios written in a ubiquitous language understood not only by stakeholders but by everyone in the development team and they are also the base of the automated tests
- Promotes better understanding of the software being built
- BDD offers more precise guidance on organizing the conversation between developers, testers and domain experts, thinking through solutions, before starting to code
- Improved quality code reducing costs of maintenance and minimizing project risk

1.19 TDD Vs. BDD

- Test-driven development focuses on the developer's opinion on how parts of the software should work
- Behavior-driven development focuses on the users' opinion on how they want your application to behave
- TDD directs focus on testing and BDD directs focus on behaviour and specification
- Test-driven development is rather a paradigm than a process
- It describes the cycle of writing a test first, and application code afterwards – followed by an optional refactoring
- But it doesn't make any statements about - Where do I begin to develop? What exactly should I test? How should tests be structured and named?
- Dan North suggested that instead of writing tests you should think of specifying behavior
- Behavior is how the user wants the application to behave
- Many people refer to Behavior-driven development as “test-driven development done right”

1.19 TDD Vs. BDD

- TDD captures low-level requirements (usually at the class/method level) and BDD captures high-level requirements
- TDD captures behavior or intentions just like BDD with the help of writing test cases upfront and BDD achieves the same with help of usage of Ubiquitous language

Summary

- So 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
 - Agile methodology uses TDD extensively to validate their works
 - TDD can be applied for testing database connecting code also



Summary

- What is Behaviour Driven Development?
- Behaviour Driven Development is an Agile methodology that aims to allow software development and management teams to use shared tools and processes to communicate and collaborate
- It describes the behaviour of the application in a language that is understood by business analysts, testers, developers in order to minimize misunderstandings and make the development process more visible



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.
- Question 5: In software engineering, behavior-driven development (BDD) is a software development process that emerged from _____.

