



# Software Testing

Claire Mitchell, William Reed, Krutivas Pradhan



# AD-HOC Test Definition

- Carried out informally and spontaneously
- No formal test plans or test cases
- There is a time constraint
- Formal testing documentation is not available
- Relies on the tester's knowledge to find issues
- You are just experimenting and trying different things to see what happens



# Who Performs AD-HOC Test

Can be performed by anyone involved in the software development process

- Developers
- Testers

However, it is usually performed by developers who really understand what is going on.



# Example of AD-HOC Test

- You have created an app for renting cars
  1. Open the app and begin the process
  2. Type in a date that does not exist (such as February 30)
  3. What does the app do?
    - Does it give an error message?
    - Does it respond like normal?
    - Does it even respond at all?
  4. Type in a past date
  5. What does the app do?
    - Does it give an error message?
    - Does it respond like normal?
    - Does it even respond at all?



# Unit Testing Definition

- Small parts (units) of the software are tested one at a time
- They are tested alone without the rest of the software
- Basically you are just trying to make sure that each unit works as intended
- It helps because you can find issues early on before putting everything all together
- You want to make sure that each individual part is good before you try to combine them so that you can catch mistakes early



# Who Performs Unit Testing

- Software developers
- Can be automated so that it tests the code every time you change it



# Example of Unit Testing

- You write a program that is a calculator
- It has specific functions for each basic mathematical operation (addition, subtraction, multiplication, and division)
- For the unit test, you will write separate functions for each of the functions that will end up actually going in the calculator. The test functions will plug in different combinations of numbers:
  - Positive and positive
  - Positive and negative
  - Negative and negative



# Smoke Test Definition

- Quickly checks if the most important functions are working like they are supposed to
- Performed early on or after a big change has been made to the software
- Looks for any huge issues that would mess up future development or testing
- It quickly goes through everything to make sure that it functions well enough to move on





# Who Performs Smoke Test

- Software testers
- Software developers
- Can be automated



# Example of Smoke Test

- You make a journal app
  1. Open the app to make sure that the homepage of the app works
  2. Create a new note and save it
  3. Close the app and open it again to make sure that the note you just created is still there
  4. Delete the note
  5. Close the app then open it again to make sure that the note is still deleted



# Regression Test Definition

- Checks if recent changes in the code have affected things that the code could previously do
- Rerun old tests to make sure that previous features still work correctly
- Try to find new issues that were caused by adding new code and make sure that all of the previous functions still work



# Who Performs Regression Test

- Software testers



# Example of Regression Test

- You are working on an app
- The first feature that you finish is the log in
- You have tested it several times and are certain that it is working fine
- You begin working on another feature then decide to run a regression test
  1. Make sure that you can still log in with a valid username and password
  2. Make sure you cannot log in with an invalid username and password
  3. Make sure you can still log out



# Functional Testing Definition

- Tests the functionality of software by comparing how it runs to the functional requirements
- Make sure that the software behaves as expected
- Make sure that the software meets the requirements that the stakeholders have given
- Test all functions of the software to make sure that they work correctly and give the anticipated results



# Who Performs Functional Testing

- Software testers
- Automated tests



# Example of Functional Testing

- You are testing an online shopping platform
  1. Make sure you can add items to the cart
  2. Make sure you can view your cart
  3. Make sure you can edit items in your cart
  4. Make sure you can go to checkout
  5. Make sure you can enter shipping information
  6. Make sure you can pay
  7. Make sure confirmation emails are sent after a purchase is made





# Acceptance Test Definition

- Checks if software meets the requirements
- Checks if software is ready to release to users
- Make sure that the software meets the acceptance criteria
  - Make sure it works
  - Make sure it is user friendly
  - Make sure it looks right
- Final phase of testing before it is released



# Who Performs Acceptance Test

- End users
- Stakeholders



# Example of Acceptance Test

- You have just finished an app for a new social media platform
  1. Log in with a valid username and password of a previously created account
  2. Log out
  3. Create a new account
  4. Close the app then open it again to make sure than you can log in with the new account
  5. Log in with a different account and search the account that you just created to make sure that it exists
  6. Test any other features that the stakeholders said were essential



# Integration Test

What is it?

The process of testing how two software units or models are interfaced. Integration testing should expose any faults in how integrated units interact. These tests should be done after unit testing.

When testing, the goal should be to find any problems that occur when combining different components. This should be done module by module to provide some structure, ensuring that none are skipped.



# Types of Integration Tests

## Big-Bang Test

- Simple and straightforward approach
- Combines all modules for testing, which can lead to some delay when waiting for modules to be integrated

## Bottom-Up Test

- Modules at lower levels are tested with modules at higher levels
- Ensures that each subsystem tests the interface between each module

## Top-Down Test

- Simulates the behavior of low level modules that aren't yet integrated in order to test high level modules

## Mixed/Sandwiched Test

- A combination of top-down and bottom-up testing



# Who Performs Integration Tests?

Testing can be done by QA testers, test engineers, or developers



# Example Integration Test

Mail application with 3 modules: “Login Page,” “Mailbox,” and “Delete Emails”

Case 1:

- Objective: Check the interface link between the Login and Mailbox module
- Description: Enter login credentials and click on the Login button
- Expected Result: To be directed to the Mailbox

Case 2:

- Objective: Check the interface link between the Mailbox and Delete Mails module
- Description: From Mailbox select the email and click a delete button
- Expected Result: Selected email should appear in the Deleted/Trash folder



# System Test

System testing is used to test a completed integrated system against the corresponding requirements. The goal of system testing is to test the software beyond the scope of the Software Requirements Specification (SRS). These tests typically follow integration tests, and test the whole system rather than individual units, but testing of individual units can still be done at this point.

## Types of System Tests

Performance Testing - tests the speed, reliability, and stability of a software product

Load Testing - used to determine the behavior of the product under a heavy load

Stress Testing - used to test the software under varying loads

Scalability Testing - used to test the software's performance under either large or small amounts of user requests at once.





# Who Performs System Tests?

System tests are generally performed by QA teams or a professional testing agent within a company once the software product has been completed



# Example System Test

You are testing an airline's online booking site:

- Test that login works
- Customers can browse flights and prices
- Flight dates and times can be selected
- Booking works properly
- The website works on multiple different browsers



# Load Test

Load testing is used to see the response of an application or system when multiple users are using it at the same time. Various conditions are used to test the system's ability to handle different loads. Load testing simulates real world loads on the system to accurately test the performance response of a system.

## Metrics of Load Testing

- Average Response Time
- Error Rate
- Throughput
- Requests Per Second
- Concurrent Users
- Peak Response Time



# Who Performs Load Tests?

Tests can be performed by developers and often utilize load testing tools that simulate user activity on a system. Load tests should be done regularly throughout the development life cycle to catch any problems as soon as they arise, rather than finding a lot of errors at once.



# Example of Load Testing

Suppose you have a website which users can log onto to purchase goods. A load test should:

- simulate many users logging onto the website
- browsing the website
- adding products to a cart
- initiating returns
- logging off



# Soak Test

Soak testing is a performance test which is designed to put the system under a fairly heavy load for a long period of time to see how it responds. The load it's put under should be about as heavy as the system is expected to be able to maintain under normal working conditions. Soak testing is a type of load testing.



# Who Performs Soak Tests?

Soak tests can be performed by

- Software testers
- Performance testing experts
- Developers



# Example Soak Test

One way a soak test can be implemented is in a 60-70 hour test. The test could be started on a Friday evening and will finish the following Monday morning. Suppose that 33,000 logins are put through the system in 7 days usually. During the soak test, the tester could simulate 33,000 logins to test how the system reacts over roughly 2 days handling a 7 day load.





# What is a testing framework?

A testing framework (test automation framework) is a set guidelines for creating test cases. Test frameworks increase efficiency when testing. They also cost less to maintain, require little manual intervention, get maximum test coverage, and make test code more reusable. While it is not a requirement that these testing guidelines be followed, the results that we gather can be different based on how closely we follow the guidelines.



# Types of Test Automation Frameworks: Linear Scripting

Linear Scripting (Record and Playback) is the simplest testing framework. A tester manually records each step and inserts checkpoints. After the first runthrough, the script is recorded and can be run automatically.

## Advantages

- Fastest way to generate a script
- No automation expertise required

## Disadvantages

- Little reuse of scripts
- Test data is hard coded in the script



# Types of Test Automation Frameworks: Test Library Architecture Framework


In this test, Linear Scripting is used initially. Afterwards, similar tasks in the script are grouped into smaller functions that are called by a main test script to create different tests cases.

## Advantages

- Higher level of code reusability than Linear Scripting
- Automation scripts are less costly due to code reusability

## Disadvantages

- More time is needed to plan and prepare test scripts
- Test data is hard coded within the scripts



# Types of Test Automation Frameworks: Data-driven Testing Framework

In this framework, the test case logic is kept in a test script, and test data is stored in a separate file, such as an Excel file, CSV file, text file, etc. and is loaded into the test script. Test scripts in this framework are prepared using Linear Scripting or Test Library Framework.

## Advantages

- Changes to the test script don't affect the test data
- Test cases can be executed with different sets of test data

## Disadvantages

- More time is needed to prepare both test scripts and test data




# Types of Test Automation Frameworks: Keyword-driven Testing Framework

This framework requires data tables and keywords to execute the tests. Additionally, tests can be designed with or without the Application. There are 3 parts of a Keyword-driven Framework: Keyword, Application Map, and Component Function.

Keyword – an action that can be performed on a GUI Component

Application Map – provides named references for GUI Components

Component Function – functions that actively manipulate or interrogate the GUI Component




# Types of Test Automation Frameworks: Keyword-driven Testing Framework

## Advantages

- Provides high code reusability
- Test tool independent

## Disadvantages

- High automation expertise is required
- The benefits of this tests are only really seen if the application is large and the tests are to be maintained over many years



## Types of Test Automation Frameworks: Hybrid Test Automation Framework

This framework is simply the combination of the previously listed testing frameworks. The reason one might choose to make a hybrid test framework is to pull from each of their strengths and weaknesses depending on the test that needs to be made.



# Introduction To Test-Driven Development

## What is Test-Driven Development

Test-Driven Development (TDD) is a software approach where tests are written before the code, ensuring testing at every development stage, unlike traditional methods where tests follow coding. This cycle enhances code quality and ensures functionality correctness from the start.

## Key Components of TDD

- **Test First**
- **Minimum Code Requirement**
- **Refactor**





# The Benefits of TDD

## Improves Code Quality

- TDD leads to a codebase with fewer bugs and errors because it encourages developers to think through the requirements or design before writing the code.

## Facilitates Code Understandability

- With TDD, developers can make changes to the codebase more confidently. Tests act as a safety net that ensures new changes don't break existing functionality.

## Enhances Code Understandability

- The tests written during TDD serve as documentation for the codebase. They clearly describe what the code is supposed to do, which helps new developers understand the existing codebase faster.

## Encourages Simplicity

- TDD encourages writing only the code necessary to pass tests, which can lead to simpler, more focused code development.



# The TDD Cycle

Test-Driven Development revolves around a simple but powerful cycle that ensures quality and functionality from the ground up. This cycle is often summarized as "Red-Green-Refactor."

## 1. Red: Write a Failing Test

The cycle begins with writing a test for the next bit of functionality you want to add. The test should fail because the functionality doesn't exist yet. This is the Red phase, indicating that the test suite is not passing.

## 2. Green: Make the Test Pass

The next step is to write the minimum amount of code necessary to make the test pass. This often means implementing just enough functionality for the test to succeed, leading to the Green phase. This phase is about proving that the new functionality works as intended.

## 3. Refactor: Improve the code

With a passing test, you then refactor the code. This involves cleaning up, optimizing, and improving the design of the new and existing code, but without changing its behavior. The tests are run again to ensure that the refactoring has not broken anything, maintaining the Green status.



# Repeating the TDD Cycle

The Test-Driven Development (TDD) process is inherently iterative. Each cycle of Red-Green-Refactor not only adds a new piece of functionality but also enhances the code's structure and maintainability. By repeating this cycle, developers ensure that the software evolves in a controlled, testable manner.

Repeating the TDD cycle is not just about adding new features; it's a philosophy of software development that emphasizes quality, adaptability, and continuous improvement. Through disciplined repetition of the Red-Green-Refactor steps, teams build robust, flexible software that can grow and evolve to meet the changing needs of users and stakeholders.



# Variants of TDD

1. **Behavior-Driven Development (BDD)**
2. **Acceptance Test-Driven Development (ATDD)**
3. **Exploratory Test-Driven Development (ETDD)**
4. **Developer Test-Driven Development (DevTDD)**



# Behavior-Driven Development

- **Focus:**

BDD extends TDD by specifying software behaviors using simple, domain-specific language. This approach encourages collaboration between developers, QA, and non-technical stakeholders to define software behavior before development begins.

- **Process:**

BDD uses scenarios (typically written in Gherkin syntax) to describe the desired behavior of the system, which then guide the writing of tests and code.

- **Benefits:**

Improves communication and understanding across the team, ensuring that the software meets business requirements.



# Acceptance Test-Driven Development(ATDD)

- **Focus:**

ATDD emphasizes collaboration among team members (developers, testers, and customers) to define acceptance criteria before development starts. It focuses on satisfying the user's requirements and ensuring the system behaves as expected from the user's perspective.

- **Process:**

Starts with creating acceptance tests that represent the user's requirements. These tests guide development and ensure the software delivers the intended value to the user.

- **Benefits:**

Enhances clarity of requirements, improves customer satisfaction, and ensures that the product meets its defined acceptance criteria.



# Exploratory Test Driven Development (ETDD)

- **Focus:**

ETDD combines TDD with exploratory testing techniques. It encourages simultaneous learning, test design, and test execution. This approach is particularly useful in complex domains where requirements are unclear or rapidly changing.

- **Process:**

Developers write tests to explore potential behaviors and outcomes rather than to confirm specific, predefined behaviors.

- **Benefits:**

Promotes a deeper understanding of the software and its potential issues, encouraging innovation and adaptive design.



# Developer Test Driven Development (DevTDD)

- **Focus:**

This is the classic form of TDD, focusing on writing unit tests for small pieces of functionality before writing the corresponding code.

- **Process:**

Follows the Red-Green-Refactor cycle closely, with a strong emphasis on writing concise, clear unit tests that drive the design of the software.

- **Benefits:**

Leads to well-tested, modular, and easily maintainable code.





# Testing and Agile

Agile development methodologies emphasize flexibility, customer satisfaction, and rapid delivery of functional software. Testing in Agile is not a standalone phase but a core activity that complements every stage of the development process. Here's how Agile principles transform testing into a dynamic and integral component of software development.

Agile testing is a holistic and integrated approach that ensures continuous improvement and delivery of high-quality software. By embedding testing into the Agile process, teams can adapt to changes swiftly, meet customer needs more effectively, and foster a culture of collaboration and quality.



# The Role of TDD in Agile

## 1. Ensures Continuous Quality Improvement

TDD, by its nature, ensures that testing is an integral part of the development process. This leads to early bug detection and resolution, which is vital for maintaining high code quality throughout the Agile development cycle.

## 2. Facilitates Agile Iterations

The iterative nature of TDD—with its short Red-Green-Refactor cycles—complements Agile sprints. It allows teams to build, test, and refine software incrementally, ensuring that each iteration is robust and meets user needs.

## 3. Enhances Code Maintainability

Refactoring is a core component of TDD, which ensures that code remains clean and maintainable. This is crucial in Agile projects, where changes are frequent and the codebase must be adaptable to new requirements without increasing technical debt.

## 4. Improves Team Collaboration

TDD encourages collaboration between developers, testers, and even business stakeholders by making the requirements clear and testable from the start. This shared understanding of objectives and quality standards fosters a unified approach to product development.

## 5. Supports Agile Principles of Customer Satisfaction and Responding to Change

By integrating TDD, teams can more confidently introduce changes and new features, knowing that the test suite will catch regressions. This agility supports the ultimate goal of delivering maximum value to customers, in line with Agile principles.



# Benefits of TDD in Agile Environments

- **Enhanced Product Quality:** Continuous testing and early defect detection contribute to higher quality products.
- **Faster Time to Market:** Agile testing, especially when automated, speeds up the development process, allowing for quicker releases.
- **Increased Collaboration:** Involving testers, developers, and stakeholders throughout the project strengthens the team and enhances product relevance and quality.
- **Customer-Centric Approach:** Regular feedback loops with stakeholders ensure the product evolves in line with customer needs and expectations.



# Challenges of Implementing TDD in Agile

- **Learning Curve:** Transitioning to a test-first mindset requires time and patience, slowing initial progress as teams adapt.
- **Perceived Slowness:** The upfront investment in writing tests can seem to delay feature development, challenging Agile's fast-paced nature.
- **Test Maintenance:** As the project evolves, maintaining and updating a growing suite of tests can become burdensome.
- **Test Coverage Balance:** Finding the right level of test coverage—avoiding both excessive and insufficient testing—is a continuous challenge.
- **Process Integration:** Incorporating TDD into existing Agile workflows may disrupt established practices, requiring careful management of change.
- **Team Buy-in:** Achieving unanimous support for TDD across the development team and stakeholders is crucial but often difficult.



# Case Study: TDD in an agile Project (Part 1)

This case study explores the integration of Test-Driven Development (TDD) within an Agile project at TechSolutions Inc., a software development company facing challenges with code quality and project timelines.

## Background:

TechSolutions Inc. was struggling with frequent delays in product releases and a high incidence of bugs in production. The development team was using Agile methodologies but had not fully embraced TDD.

## Implementation:

The company decided to integrate TDD into their existing Agile framework with the goal of improving code quality and adherence to project schedules.

## Steps Taken:

**Training:** The team underwent comprehensive training on TDD principles and practices.

**Pilot Project:** A small, low-risk project was chosen to pilot the TDD approach.

**Tool Integration:** Tools and frameworks supportive of TDD, such as JUnit for Java and Jest for JavaScript, were integrated into the development process.

**Continuous Integration (CI):** The CI pipeline was configured to run the test suite on every commit, ensuring immediate feedback.



# Case Study: TDD in an agile Project (Part 2)

## Outcomes:

- **Improved Code Quality:** The number of bugs reported in production decreased by 40% within six months.
- **Enhanced Productivity:** The clarity provided by TDD led to a more focused development effort, reducing the time spent on debugging by 30%.
- **Better Project Predictability:** With fewer bugs and less time spent on unplanned work, project timelines became more predictable, improving client satisfaction.
- **Team Morale:** The development team reported higher satisfaction levels due to reduced stress and clearer development objectives.

## Lessons Learned:

- **Early Buy-in is Crucial:** Securing team buy-in from the start was key to the successful adoption of TDD.
- **Continuous Improvement:** The team learned to continuously refine their testing practices and not to be deterred by initial setbacks.
- **Integration with Agile:** TDD complemented the Agile framework, enhancing its focus on quality and iterative improvement.

## Conclusion:

TechSolutions Inc.'s experience demonstrates that integrating TDD into Agile projects can significantly improve both product quality and project management outcomes. The key to success lies in comprehensive training, choosing the right tools, and fostering a culture of continuous improvement and quality focus.

This case study serves as a powerful example for other organizations considering TDD as a strategy to enhance their Agile practices.



# Tools and Frameworks for TDD

- 1. JUnit (Java):** A cornerstone testing framework for Java developers, facilitating easy creation and management of tests.
- 2. NUnit (C#/.NET):** A widely used testing framework for .NET languages, offering a rich set of assertions to test code behavior.
- 3. PyTest (Python):** Offers a no-boilerplate way to write simple and scalable test cases for Python applications.
- 4. RSpec (Ruby):** A behavior-driven development (BDD) framework for Ruby, making tests readable and maintainable.
- 5. Jest (JavaScript/Node.js):** Popular in web development for its simplicity and support for client-side and server-side JavaScript testing.
- 6. TestNG (Java):** Similar to JUnit but with more advanced features, including annotations, parameterized tests, and dependency testing.
- 7. Mocha (JavaScript):** A flexible testing framework for Node.js and browsers, often paired with assertion libraries like Chai.
- 8. XCTest (Swift/Objective-C):** Integrated into Xcode, it's the primary framework for testing iOS and macOS applications.



# Best Practices for TDD

- **Start Small:** Begin with simple tests to build confidence and understanding before tackling more complex scenarios.
- **Test One Thing at a Time:** Each test should focus on a single functionality or aspect of the code to ensure clarity and simplicity.
- **Keep Tests Independent:** Ensure that tests do not rely on each other to pass, maintaining the ability to run each test in isolation.
- **Write Clear and Descriptive Tests:** Naming tests clearly and descriptively makes it easier to understand what is being tested and why.
- **Refactor Regularly:** Use the refactor phase not just to improve code, but also to refine tests for better readability and efficiency.
- **Prioritize Test Maintenance:** Regularly review and maintain test suites to keep them relevant and effective as the codebase evolves.
- **Embrace Simple Designs:** Let TDD guide you towards simpler, more modular designs that are easier to test and maintain.
- **Integrate with Continuous Integration (CI):** Automate test execution within your CI pipeline to ensure immediate feedback on the impact of code changes.





# Conclusion

- **TDD Fundamentals:** At its core, TDD is a development technique where tests are written before the code, ensuring every piece of code is tested from the outset.
- **The TDD Cycle:** The Red-Green-Refactor cycle encourages a systematic approach to coding, focusing on minimal code to pass tests and constant refinement.
- **Variants of TDD:** Understanding Behavior-Driven Development (BDD), Acceptance Test-Driven Development (ATDD), and other variants allows teams to tailor TDD principles to their specific project needs and stakeholder expectations.
- **TDD and Agile Synergy:** TDD aligns with Agile principles by supporting iterative development, continuous feedback, and high product quality.
- **Challenges and Solutions:** While implementing TDD within Agile environments presents challenges, such as the initial learning curve and maintaining test suites, these can be navigated through best practices like keeping tests simple and clear, ensuring test independence, and integrating testing into the CI/CD pipeline.
- **Tools and Frameworks:** The adoption of TDD is supported by a wide array of tools and frameworks designed to facilitate testing across different programming languages and platforms.
- **Best Practices for Success:** Adopting best practices, including starting small, testing one thing at a time, and embracing simple designs, ensures the successful integration of TDD into Agile projects.



Thank You



# Sources

- <https://www.guru99.com/adhoc-testing.html>
- <https://www.geeksforgeeks.org/unit-testing-software-testing/>
- <https://aws.amazon.com/what-is/unit-testing/>
- <https://www.techtarget.com/searchsoftwarequality/definition/smoke-testing#:~:text=Smoke%20testing%2C%20also%20called%20build.not%20delve%20into%20finer%20details.>
- <https://www.geeksforgeeks.org/smoke-testing-software-testing/>
- <https://smartbear.com/learn/automated-testing/what-is-regression-testing/>
- <https://www.opentext.com/what-is/functional-testing#:~:text=is%20Functional%20Testing%3F-.Overview.with%20the%20end%20user%27s%20expectations.>
- <https://www.geeksforgeeks.org/software-testing-functional-testing/>
- <https://www.techtarget.com/searchsoftwarequality/definition/acceptance-test#:~:text=Acceptance%20testing%20occurs%20after%20system.stability%20and%20checks%20for%20flaws.>



# Sources

- <https://www.geeksforgeeks.org/software-engineering-integration-testing/>
- <https://www.practitest.com/resource-center/article/system-testing-vs-integration-testing/>
- <https://www.guru99.com/integration-testing.html>
- <https://www.geeksforgeeks.org/system-testing/>
- <https://www.guru99.com/system-testing.html>
- <https://www.browserstack.com/guide/what-is-system-testing>
- <https://www.geeksforgeeks.org/software-testing-load-testing/>
- <https://www.blazemeter.com/blog/performance-testing-vs-load-testing-vs-stress-testing#what-is-load-testing>
- [https://tryqa.com/what-is-load-testing-in-software/#Examples of load testing include](https://tryqa.com/what-is-load-testing-in-software/#Examples%20of%20load%20testing%20include)
- <https://katalon.com/resources-center/blog/soak-testing>
- [https://testsigma.com/blog/soak-testing/#Who is involved in soak testing](https://testsigma.com/blog/soak-testing/#Who%20is%20involved%20in%20soak%20testing)
- <https://smartbear.com/learn/automated-testing/test-automation-frameworks/>
- <https://www.guru99.com/test-automation-framework.html>



# Sources

- <https://www.guru99.com/test-driven-development.html>
- [https://www.oreilly.com/library/view/modern-c-programming/9781941222423/f\\_0054.html](https://www.oreilly.com/library/view/modern-c-programming/9781941222423/f_0054.html)
- <https://www.techtarget.com/searchsoftwarequality/tip/TDD-vs-BDD-vs-ATDD-and-other-Agile-development-techniques>
- <https://www.atlassian.com/agile/software-development/testing>
- <https://www.xcubelabs.com/blog/an-overview-of-test-driven-development-tdd-tools-and-techniques/>
- <https://www.linkedin.com/pulse/test-driven-development-tdd-agile-comprehensive-guide-mehbub-rabba-ni/>
- <https://www.testingxperts.com/blog/test-driven-development-agile>