**Assignment 5**

**Comparative Info graphic of TDD, BDD, and FDD**

**Title: Comparative Analysis of Software Development Methodologies**

**Introduction**

"Software development methodologies like Test-Driven Development (TDD), Behavior-Driven Development (BDD), and Feature-Driven Development (FDD) offer unique approaches to building software. Understanding the differences between these methodologies is crucial for selecting the right approach for your project."

* **TDD Overview**

Simply put, the TDD process involves writing automated tests before writing the code. The results from these automated tests provide insights for the developer to improve their code. TDD is a more focused and disciplined approach to development, and is itself a way to provide continuous feedback for faster bug identification and debugging.

## ****How To Do TDD?****

TDD is a continuous and iterative process of improving your code through tests. The goal after each test is to make the code incrementally better.

### ****1. Create Tests****

Developers start the TDD process by **writing a test**. In this test, they specifically define the expected behavior or functionality of a small unit of code, which may include:

* Code input (what parameters the code will receive)
* Code output
* Preconditions/dependencies that must exist/be true for test execution
* Assertions

The developer can write the test themselves using a programming language or leverage some automatio testing with low-code features for faster test authoring and test execution. The final decision lies in whether the developer wants to **build**a test framework or **buy**a pre-built framework from a vendor, which we have explained quite thoroughly in this article that you may want to take a look at.

### ****2. Execute 1 Specific Test****

Next, developers run 1 specific test and observe it fail (since no code was developed at this stage, yet). It may sound counterintuitive, but this is actually the fundamental principle behind TDD. There are 4 major reasonings for why we need this step:

1. Running tests on an underdeveloped code ensures that your testing infrastructure, frameworks, and environment are properly set up and functioning correctly. If there are any issues with the test environment, you can address them from the very beginning.
2. A failed test confirms that it is indeed checking the behavior of the functionality you want to code. It is a positive signal that this test is reliable.
3. This test (and future tests that will be executed) act as a checklist that guides your development activities. They become objectives to pursue (think: I am coding this feature in order to pass this test), keeping developers focused on meeting the requirements and specifications laid out in the [test plan](https://katalon.com/resources-center/blog/test-plan).
4. As the cycle repeats, this step provides immediate feedback on the correctness of the code, allowing developers to catch misassumptions early on, minimizing the risk of introducing issues later down the road.

In other words, in TDD, a failed test is a good test. The real question is: Which specific test should be executed and failed first? Usually we should just start the initial test with a small, focused scope that captures the core requirements of the feature or function. 

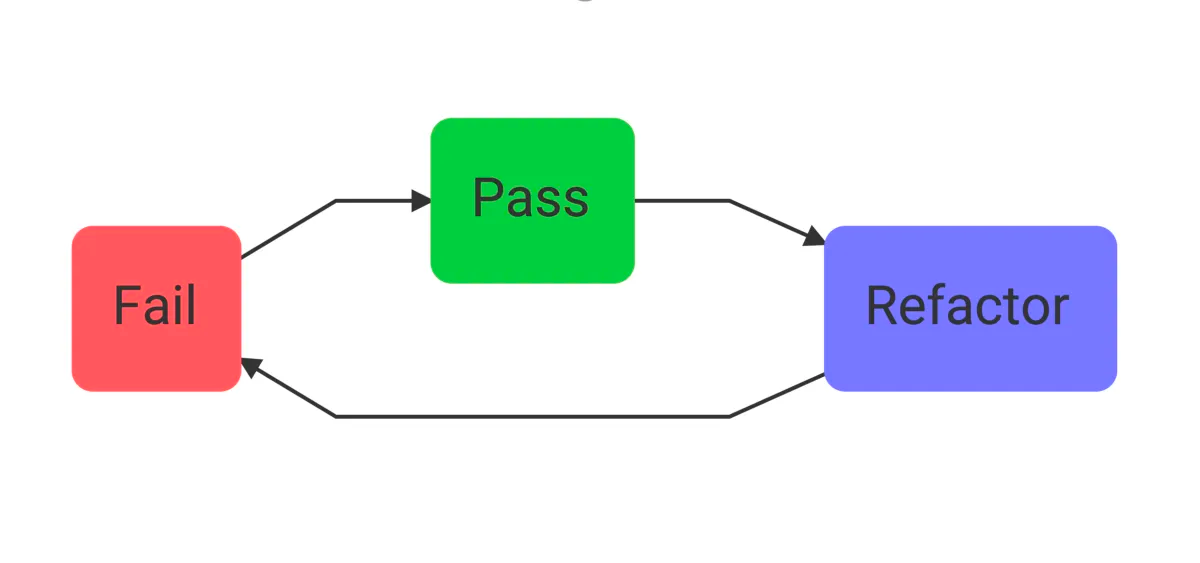
### ****3. Implement The Code****

With the insights from the tests, developers start to write **just enough** code to make the tests pass. At this stage, the goal is still not to make a complete and fully optimized solution, since, again, the reasoning behind TDD is to use test results to guide development, and not the other way around. Developers should only optimize their code after receiving input from tests.

### ****4. Run All Tests And Refactor The Code****

Now that we have some newly added code, developers can run all of the remaining tests, including the one they’ve just written to further confirm that the new code is working fine, and does not break any existing functionality. It is similar to [regression testing](https://katalon.com/resources-center/blog/regression-testing) where the tests are executed to ensure that the features are still working after a code change. In a way, TDD is about continuously executing regression tests to provide feedback on the quality of newly written code.

With this feedback, developers can refactor the code, improve its design, readability, and maintainability. After that, the cycle of Fail - Pass - Refactor begins again, and we call this the Red - Green - Refactor cycle of TDD.



In companies with a TDD policy, developers don’t have to be afraid of show-stopping bugs slipping into production, since quality checks happen way before code implementation. Over time, QA teams can even develop a comprehensive set of test suites that can be reused across scenarios and environments, allowing TDD to unfold at a much faster rate

* **BDD Overview**

BDD testing is an Agile approach to software testing where testers write test cases in simple language that even people without technical expertise can understand. The goal of BDD testing is to increase collaboration between the technical side and the business side of the organization. 

The Gherkin language is a unique business-readable language used to describe system behaviors and scenarios. It is structured by three primary statements: Given, When, and Then, with each statement describing a specific aspect of the system.

* **The Given statement** sets up the initial context for the behavior and defines the starting point of the system.
* **The When statement** describes the trigger that brings about a change or behavior in the system.
* **The Then statement** defines the expected outcome that should be observed after the event mentioned in the When statement.

After articulating the system behavior in Gherkin, BDD testers **turn it into a test script** that can be understood by the computer.

## ****Why Do We Need BDD Testing?****

BDD testing is based on the idea that the **tres amigos (i.e., three friends)**in software development don’t always share the same perspectives, and a common ground between them must be established to facilitate smooth communication. The tres amigosinclude:

* **Product Owner**: representing the business side, they simply want to **solve a problem**they have.
* **Developer**: representing the development side, they want to **build a solution to solve the problem**of the Product Owner.
* **Tester**: representing the QA side, they want to **know if the solution can actually solve the problem**, and **what can possibly go wrong with the solution**.

The traditional approach to testing does not connect the perspectives of the tres amigos. Stakeholders come to the product owner to transfer their requirements, then the product owner explains those requirements to developers and testers. After that, developers translate the requirements into code, while the testers translate them into test cases. It was a lengthy process, and loss in translation can occur, which inevitably leads to misunderstandings.

Instead, the amigos come together and have a discussion, then turn the insights into a documentation using a shared language to create a common understanding of the issue at hand. Testers can then use a BDD testing frameworkto create test cases.

## ****Advantages of BDD Testing****

With BDD testing, you can:

* Have a common language for engineers and stakeholders to communicate about user needs and software solutions.
* Craft user stories and scenarios to define acceptance criteria for specific software features.
* Promote collaboration among users, quality team, product team, and engineers to achieve consensus on the project's goals.
* Reduce uncertainties.
* Enable faster product delivery by the rapid creation of mutually agreed upon [automated tests](https://katalon.com/resources-center/blog/what-is-automation-testing).
* Reduce errors in software implementation by converting business requirements into a clear format for all executed tests.
* Minimize the learning curve and jargon.

## ****Limitations of BDD Testing****

However, BDD testing is not without limitations:

* Potentially leads to lengthy stakeholder meetings and the inclusion of all key stakeholders in the BDD process.
* Demands well-prepared software requirements for the Gherkin scripts to accurately express the business needs, which may conflict with the fast-paced nature of Agile teams working from concise specifications.
* Less efficient for small development teams.

## ****Steps To Do BDD Testing****



Day-to-day BDD activity involves a three-step process: discovery, formulation, and automation. These steps help your team confidently make changes to the system. The code reflects the documentation, which reflects everyone's understanding of the problem.

### ****1. Discovery Stage****

During the **Discovery stage**,stakeholders participate in structured conversations called discovery workshops (or more commonly known as brainstorming sessions) where they discuss and reach a mutual agreement on the desired goals. They start by stating the user story or requirement in a concise format, such as “As a [role], I want [goal] so that [benefit].”

After that, they brainstorm examples of how the user story should behave in different scenarios. These examples focus on concrete situations and help to define the expected behavior. The team then writes each example on sticky notes or a digital tool, arranging them in a structured format. A common format includes columns for the example, a brief description, rules or constraints, and any open questions or areas.

This approach is known as “Example Mapping.” Created by the founder of Cucumber, one of the most popular BDD testing frameworks, Example Mapping’s goal is to create a shared understanding of user stories or requirements by breaking them down into smaller, concrete examples for the three amigos to follow.

### ****2. Formulation Stage****

The issue with Example Mapping is that it does not present information in a structured format, which is something Gherkin was created to address. Once the desired behavior is fully understood, the team moves on to formulating executable specifications using Gherkin.

These specifications are written in a way that stakeholders can easily comprehend and validate, serving as a foundation for creating automated tests. This stage involves defining the feature files, writing scenarios, and identifying the steps required to implement each scenario.

### ****3. Automation Stage****

Finally, in the **Automation stage**, theyimplement the behavior described in each example, starting with the automated test. The executable specification guides the implementation process. 

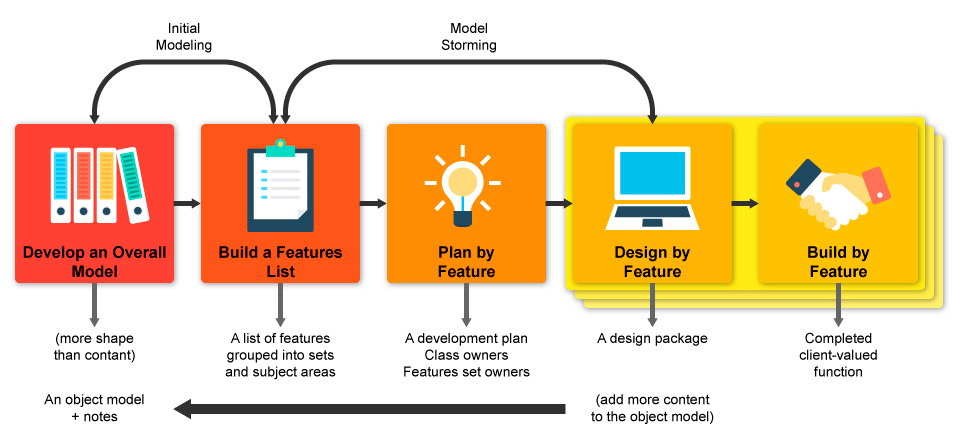
These steps can be implemented as unit tests or integrated into a larger test suite using a testing framework. The automation process involves mapping each step to the corresponding code and ensuring that the tests accurately reflect the desired behavior. Automation testing allows for efficient and repeatable execution of the tests, reducing the need for manual regression testing, enabling testers to focus on more engaging tasks like [exploratory testing](https://katalon.com/resources-center/blog/exploratory-testing).

## ****Examples of Using Gherkin for BDD Testing****

There are several rules in writing Gherkin statements:

* Besides Given, When, and Then statements, testers can also use other keywordssuch as Feature, Rule, Example, And, But, Background, Scenario Outline, Examples to describe the behavior.
* You should use these keywords at the start of a line.
* Comments are permitted at the start of a new line and begin with optional spaces, followed by a hash sign (#) and text.
* Indentation can be achieved using spaces or tabs, with a recommended indentation level of two spaces.
* **FDD Overview**

Feature-Driven Development (FDD) is an iterative and incremental software development methodology. It combines several industry-recognized best practices into a cohesive whole, focusing on delivering tangible, working software repeatedly in a timely manner. Here’s an overview of FDD, its key features, and its advantages and disadvantages:



**Key Features of FDD**

Domain Object Modeling: This involves creating a comprehensive model of the problem domain, which helps in understanding the system and identifying the key features.

Feature List: The entire system is broken down into a list of features, which are small, client-valued functions that can be developed in less than two weeks. This list forms the backbone of the development process.

Planning by Feature: Features are grouped into sets, and a high-level plan is created, which includes prioritizing the features and assigning them to iterations.

Design by Feature: Each feature is designed in detail before it is implemented. This includes creating sequence diagrams and refining the object model as necessary.

Build by Feature: Features are implemented one by one. The development team designs, codes, tests, and integrates each feature independently, ensuring that it works properly before moving on to the next.

Regular Builds: Frequent builds are conducted to integrate the features and ensure that the system is working correctly as a whole. This helps in early detection of integration issues.

**Advantages of FDD**

Scalability: FDD can scale well to large teams and large projects because it focuses on developing small, manageable features.

Predictability: By breaking down the project into small features and using short iterations, FDD provides more predictable and manageable development cycles.

Client-Focused: Features are defined in terms of client-valued functions, ensuring that the development work is aligned with the client’s needs and priorities.

Improved Quality: Regular builds and the emphasis on designing each feature before building it can lead to higher quality code and fewer defects.

Team Collaboration: FDD encourages collaboration among team members through practices such as domain object modeling and design inspections.

**Disadvantages of FDD**

Complexity in Initial Setup: The initial stages of FDD, such as domain object modeling and creating a comprehensive feature list, can be complex and time-consuming.

Less Flexibility: FDD’s structure and focus on upfront design can make it less flexible in adapting to changes late in the development process compared to methodologies like Extreme Programming (XP) or Scrum.

Requires Skilled Teams: The success of FDD heavily depends on the skill levels of the team members, particularly in modeling and design. It may not be suitable for inexperienced teams.

Documentation Overhead: The emphasis on creating detailed designs and models can lead to a significant amount of documentation, which may be seen as overhead in some environments.

* **Unique Approaches**

"While TDD focuses on writing tests first and ensuring code correctness, BDD emphasizes behavior-driven scenarios and user stories. FDD, on the other hand, prioritizes feature development and iterative delivery."

* **Benefits**

"TDD, BDD, and FDD offer various benefits, including improved communication, faster development cycles, and reduced defects. TDD reduces bugs by catching them early, BDD improves collaboration by focusing on behavior, and FDD enhances project visibility through feature-based development."

**Conclusion**

"In conclusion, understanding the differences between TDD, BDD, and FDD is crucial for selecting the right methodology for your project. Each approach offers unique benefits and is suitable for different contexts, ultimately leading to the successful delivery of high-quality software."