**Assignment 2**

**Produce a comparative infographic of TDD, BDD and FDD methodologies. illustrate their unique approaches, benefits and suitability for different software development contexts. use visuals to enhance understandings.**

**Test Driven Development (TDD)**

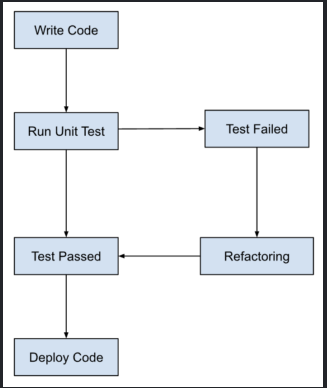
In layman’s terms, Test Driven Development (TDD) is a software development practice that focuses on creating unit test cases before developing the actual code. It is an iterative approach combining programming, unit test creation, and refactoring.

* The TDD approach originates from the Agile manifesto principles and Extreme programming.
* As the name suggests, the test process drives software development.
* Moreover, it’s a structuring practice that enables developers and testers to obtain optimized code that proves resilient in the long term.
* In TDD, developers create small test cases for every feature based on their initial understanding. The primary intention of this technique is to modify or write new code only if the tests fail. This prevents duplication of test scripts.

**Three Phases of Test-Driven Development**

* **Create precise tests:** Developers need to create exact unit tests to verify the functionality of specific features. They must ensure that the test compiles so that it can execute. In most cases, the test is bound to fail. This is a meaningful failure as developers create compact tests based on their assumptions of how the feature will behave.
* **Correcting the Code:** Once a test fails, developers must make the minimal changes required to update the code to run successfully when re-executed.
* **Refactor the Code:** Once the test runs successfully, check for redundancy or any possible code optimizations to enhance overall performance. Ensure that refactoring does not affect the external behavior of the program.

The image below represents a high-level TDD approach toward development:



**Approach:**

* Cycle: Red-Green-Refactor
* Write a Test: Define a test for the next bit of functionality.
* Run the Test: Run all tests and watch the new test fail.
* Write Code: Write the minimum amount of code necessary to pass the test.
* Refactor: Refactor the new code to acceptable standards.
* Repeat: Continue the cycle with new tests.

**Benefits:**

* Early Bug Detection: Catch bugs early before they escalate.
* Better Code Quality: Continuous refactoring leads to cleaner, more maintainable code.
* Increased Confidence: Developers can make changes with confidence knowing that tests will catch regressions.

**Suitability:**

* Complex Domains: Best suited for projects with complex logic where bugs can have significant impacts.
* High Reliability Needs: Ideal for environments where reliability is critical.

**Behavior Driven Development (BDD)**

BDD is a way for software teams to work that closes the gap between business people and technical people by:

* Encouraging collaboration across roles to build shared understanding of the problem to be solved
* Working in rapid, small iterations to increase feedback and the flow of value
* Producing system documentation that is automatically checked against the system’s behavior

We do this by focusing collaborative work around concrete, real-world examples that illustrate how we want the system to behave. We use those examples to guide us from concept through to implementation, in a process of continuous collaboration.

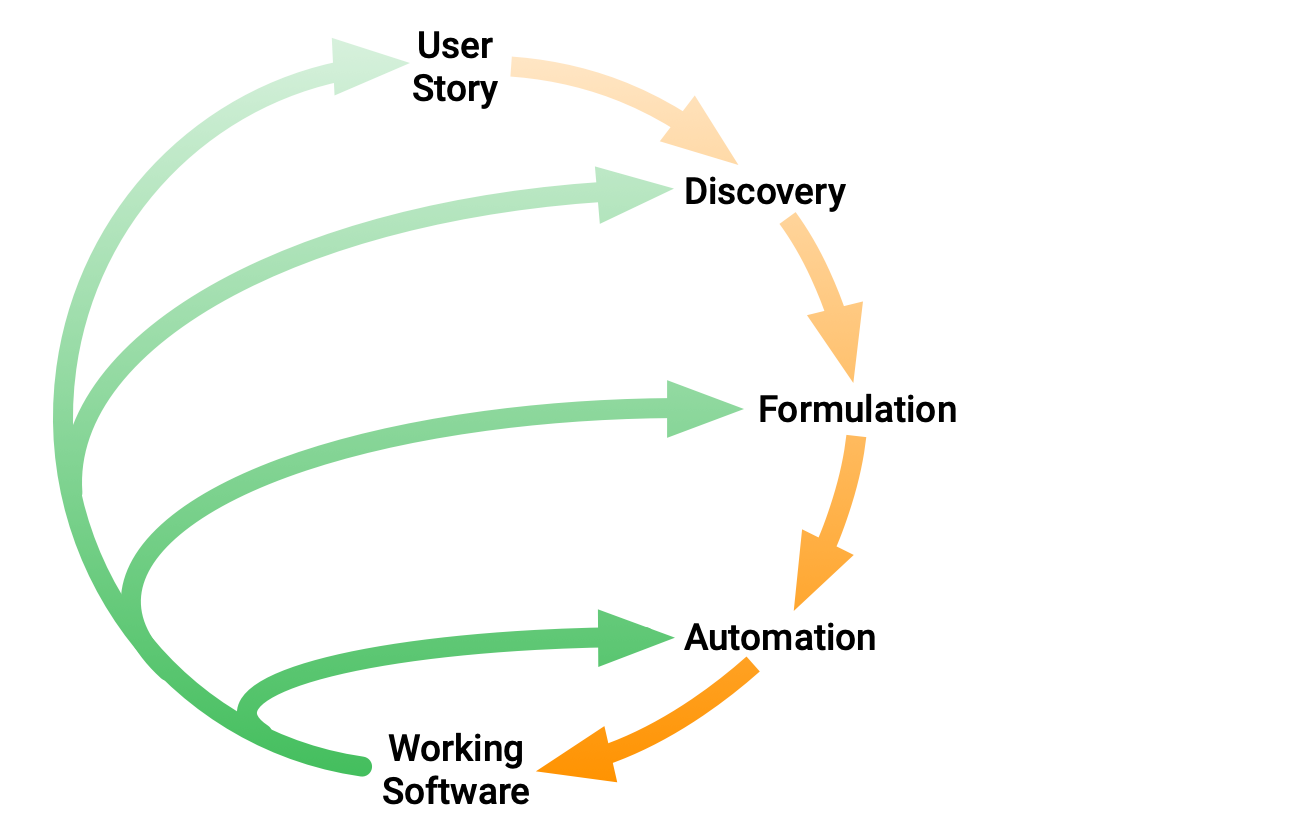
**Three Practices**

Essentially, day-to-day BDD activity is a three-step, iterative process:

* First, take a small upcoming change to the system – a [User Story](https://cucumber.io/docs/terms/user-story/) – and talk about concrete examples of the new functionality to explore, discover and agree on the details of what’s expected to be done.
* Next, document those examples in a way that can be automated, and check for agreement.
* Finally, implement the behavior described by each documented example, starting with an automated test to guide the development of the code.

The idea is to make each change small and iterate rapidly, moving back up a level each time you need more information. Each time you automate and implement a new example, you’ve added something valuable to your system, and you’re ready to respond to feedback.

We call these practices *Discovery*, *Formulation*, and *Automation*.



Over time, the documented examples become an asset that enables your team to continue confidently and rapidly making changes to the system. The code reflects the documentation, and the documentation reflects the team’s shared understanding of the problem domain. This shared understanding is constantly evolving.

**Approach:**

* Collaboration: Involves stakeholders (business analysts, developers, testers) in defining the behavior of the system using a common language.
* Gherkin Syntax: Uses Given-When-Then format to describe the behavior.
* Feature Definition: Define features in plain language.
* Scenario Writing: Write scenarios to describe different aspects of the feature.
* Automated Tests: Convert scenarios into automated tests using tools like Cucumber.

**Benefits:**

* Improved Communication: Bridges the gap between technical and non-technical team members.
* Shared Understanding: Ensures all stakeholders have a clear understanding of the system behavior.
* Focused Development: Helps in building only what is needed by the business.

**Suitability:**

* Customer-Facing Applications: Best for applications where user interactions are critical.
* Collaborative Environments: Ideal for teams that require high collaboration and clear communication.

**Feature Driven Development (FDD)**

Feature-Driven Development (FDD) is an agile methodology specifically designed to provide a structured and incremental approach to software development. It emphasizes building software by focusing on individual features. Each feature is a small, client-valued function that can be developed and delivered within a few days to a couple of weeks. Here’s a deeper look into FDD:

**Key Concepts of Feature-Driven Development**

* **Domain Object Modeling:** This involves creating a comprehensive model of the domain, often through workshops and discussions with domain experts. It helps in understanding the problem space and defining the scope of the project.
* **Feature List:** Features are client-valued functions that can be completed within two weeks. These are cataloged into a comprehensive list of features, which serves as a high-level view of the entire project.
* **Planning by Feature:** Based on the feature list, the team plans the project in terms of which features will be developed in which order. Prioritization is based on client needs and project constraints.
* **Design by Feature:** Each feature is designed in detail before development. This phase includes creating sequence diagrams and refining the domain model to support the feature.
* **Build by Feature:** This phase involves the actual implementation of the feature, including coding, unit testing, and integration. It’s done by small, cross-functional teams that follow the design prepared in the previous phase.

**Approach:**

* **Feature-Centric:** Development is driven by features that are valuable to the client. It involves five main processes:

1. **Develop Overall Model:** Build an object model of the domain problem through collaboration.
2. **Build Feature List:** Identify and prioritize features valuable to the client.
3. **Plan by Feature:** Plan the implementation of features, considering aspects like risk and complexity.
4. **Design by Feature:** Design the feature in detail, including necessary domain classes and interactions.
5. **Build by Feature:** Implement and integrate the feature, ensuring it meets quality standards through unit tests and inspections​​.

**Benefits:**

* **Clear Focus:** Emphasizes delivering tangible, client-valued features, ensuring alignment with client needs​.
* **Efficient Planning:** Breaks down development into manageable feature sets, reducing the complexity of planning and tracking progress​.
* **Scalability:** Suitable for large teams and complex projects, as it facilitates structured and scalable development processes​​.

**Drawbacks:**

* **Upfront Planning Required:** Requires significant upfront planning and a thorough understanding of project requirements​​.
* **Complex Coordination:** Managing large teams and coordinating feature development can be challenging​​.

**Suitability:**

**Large Projects:** Best for large-scale projects where feature prioritization and clear client focus are essential.

**Structured Environments:** Ideal for environments that benefit from detailed upfront planning and structured processes.