Day 3 Assignments:

Assignment 1:

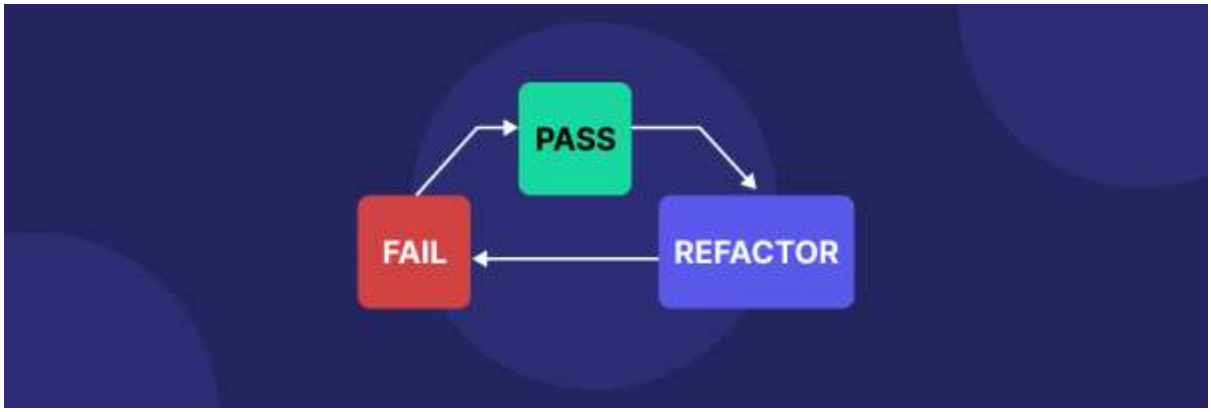
Create an infographic illustrating the Test-Driven Development (TDD) process. Highlight steps like writing tests before code, benefits such as bug reduction, and how it fosters software reliability.

Test-Driven Development (TDD) Infographic

TDD, or Test-Driven Development, is the process of writing and executing automated tests before writing code. Insights from the tests help the developer improve the code. TDD reflects the spirit of continuous feedback, resulting in faster bug identification and debugging

Test-Driven Development Cycle

TDD is a continuous, iterative process of improving code through tests. To do TDD, simply follow the mantra of **Red - Green - Refactor cycle**. Some may call it **Fail - Pass - Refactor**, but it's all the same thing.



The idea is to fail a test - make it pass - then rewrite the code. Let's see how it's done in detail here:

1. Write One Specific Test

You start by writing a test case for one specific feature. There is absolutely no code written for this feature yet. Essentially, we are writing the test for a feature that has not even been created.

2. Execute the Test

It should be quite easy to see that this test will definitely fail. Counterintuitive as it may seem, this is where the ingenuity of TDD lies. There are four major rationales as to why we must fail the test:

1. Running tests on early-stage code ensures the proper setup and function of your testing infrastructure. Any issues with the test environment can be addressed right from the start.

2. A failed test is not inherently a terrible thing. Your code is not written yet, and the test fails, which is an encouraging sign that the test is dependable.

3. This first test, along with future ones, is a checklist guiding your development activities. They become objectives to pursue (think: I code until the test is passed). It breaks down the seemingly overwhelming project into smaller chunks that can be tackled one by one. At the end of the day, TDD gives developers milestones to cross and project managers a more granular view over which features are being developed.

4. TDD offers immediate feedback on code quality.

Once you fail your test, you have reached the **Red** stage of the **Red - Green - Refactor** cycle.

In other words, to do TDD is to have a different mindset. A failed test is a good test. There is no failure, only lessons to learn from.

3. Write the Minimum Amount of Code to Make It Pass

Now is the time to code. But the good thing is you don't have to immediately flesh out a fully functioning and optimized application. Write **just enough** code to make it pass. After all, the goal of TDD is to use test results to guide development, not the other way around.

In a way, this approach requires developers to shift their focus away from the bigger picture and how the application works as a whole to the little details satisfying the requirements of that particular test case they are working on. This promotes **code minimalism**.

When we think of code minimalism, we think of:

- Simplicity
- Readability
- Focused functionality

- Modular design
- Testability
- Maintainability

And that is exactly what TDD is capable of. It turns writing code into this kind of goal-based activity with short feedback loops while also guiding the coder to write code that is concise and straight to the point.

TDD also automatically promotes **loose coupling**. When you write code for one test at a time, you don't have to constantly worry as much about the impact a certain module has on another. They are designed to operate independently. As long as they work by themselves, it's fine.

Now that the code is ready, you can run the test until it passes. A passed test now indicates that the code meets the specified requirements as outlined in the test case.

Once done, you have reached the **Green** stage of the **Red - Green - Refactor cycle**.

4. Continuously Run Tests and Refactor Code

We have arrived at the **Refactor stage**. To refactor is to restructure it to improve the code (of course without changing any external behaviour). You have found the solution to the problem, but is it the best solution yet? Here you understand the underlying mechanism of your code so ideas for optimization should come more easily.

The cycle repeats. Write another test. Run it and observe it fails. Write code for that test until it passes. Over time, you get more and more feedback from tests, continuously refactor the code, and improve its design, readability, and maintainability.

Benefits of Test-Driven Development (TDD)

TDD is incredibly powerful once you get into the rhythm of it.

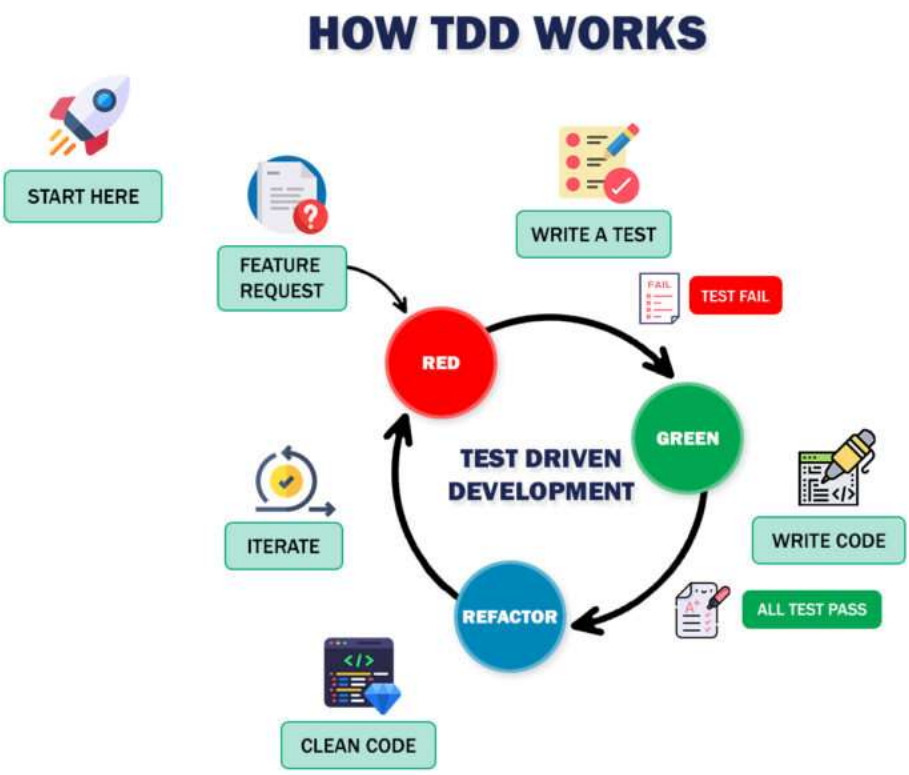
For starters, **it helps you stay focused on what truly matters**. It is basically just a three-stage process that tells you to write code for one particular feature at a time. Absolutely no tool or special technique is needed, just plain old coding and testing as you have always done. However, you will soon find that it can automatically improve the design of your application. Each component is written for itself. Such independence makes it so much easier to maintain code during updates.

Assignment 2:

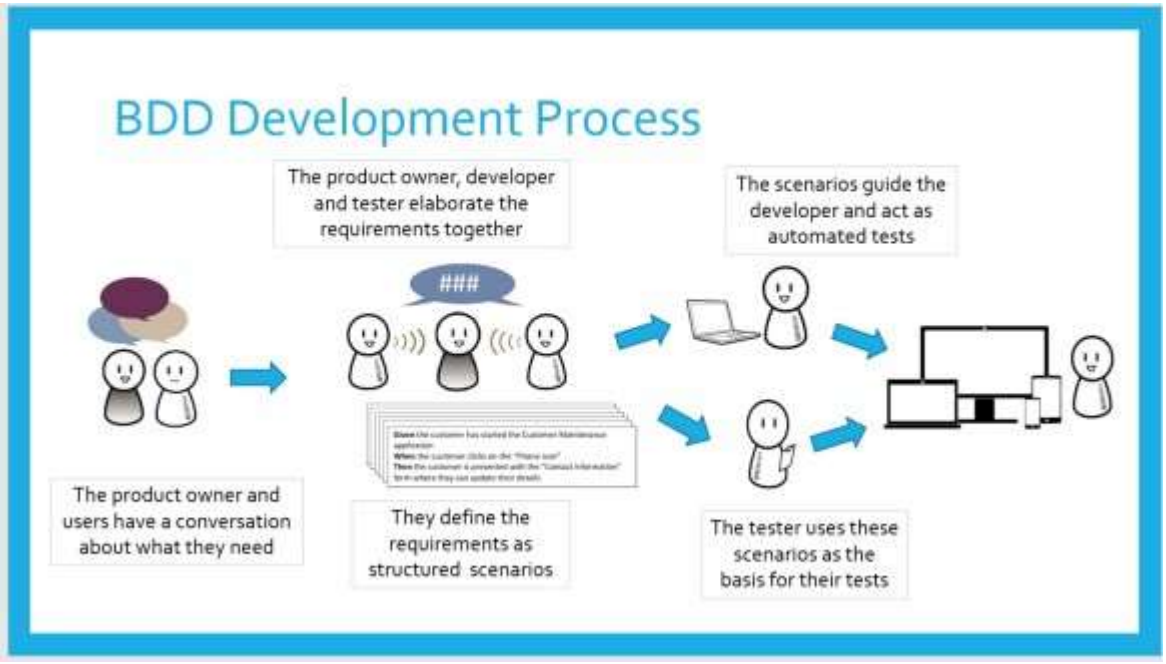
Produce a comparative infographic of TDD, BDD, and FDD methodologies. Illustrate their unique approaches, benefits, and suitability for different software development contexts. Use visuals to enhance understanding.

	TDD	BDD	FDD
Definition	**(Test-Driven Development):** Focuses on writing unit tests first to define expected behaviour before implementing code.	**(Behaviour-Driven Development):** Focuses on creating user stories and collaborating with stakeholders to define desired application behaviour.	**(Feature-Driven Development):** Focuses on breaking down projects into features, prioritizing them, and creating a development roadmap.
Unique Approaches	**TDD: "Test First, Code Later"**. Developers write unit tests that specify the desired behaviour, then write the minimal code needed to make those tests pass.	**BDD: "Focus on What, Not How"**. Stakeholders and developers collaborate to create user stories that describe how users will interact with the application. These stories are then translated into executable specifications or acceptance tests.	**FDD: "Plan the Work, Work the Plan"**. Features are identified, decomposed into smaller tasks, and assigned to development teams. Iterations are planned with clear milestones and deliverables.
Benefits	• Reduced Bugs: Early identification of issues through tests. • Improved Design: Encourages well-defined, modular code. • Faster Development: Clear test cases guide implementation.	• Clear Communication: Shared understanding between stakeholders and developers. • Early Feedback: User stories facilitate early feedback on requirements. • Improved User Experience: Focus on user behaviour leads to a more user-friendly application.	• Clear Project Scope: Defined features and milestones keep projects focused. • Reduced Risk: Prioritization helps mitigate risks associated with large projects. • Efficient Development: Iterative approach with clear deliverables.
Suitability	Well-suited for unit testing, core functionalities, and projects with a strong technical focus.	Ideal for user-centric features, collaborative development environments, and projects where user stories are well-defined.	Best suited for large projects with complex features, where risk management and clear roadmaps are crucial.

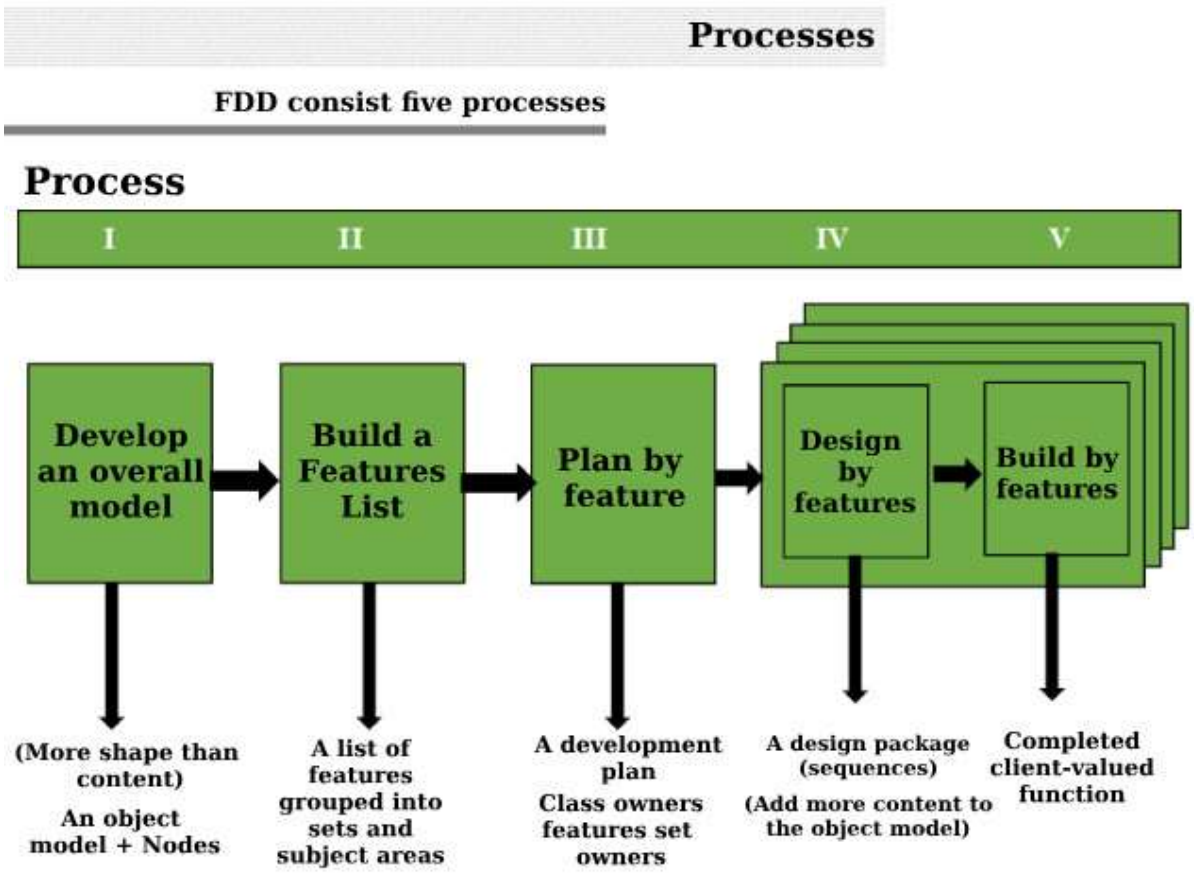
TDD works:



BDD works:



FDD works:


Processes
FDD consist five processes
Process
I
II
III
IV
V
Develop an overall model
Build a Features List
Plan by feature
Design by features
Build by features
(More shape than content)
An object model + Nodes
A list of features grouped into sets and subject areas
A development plan
Class owners features set owners
A design package (sequences)
(Add more content to the object model)
Completed client-valued function