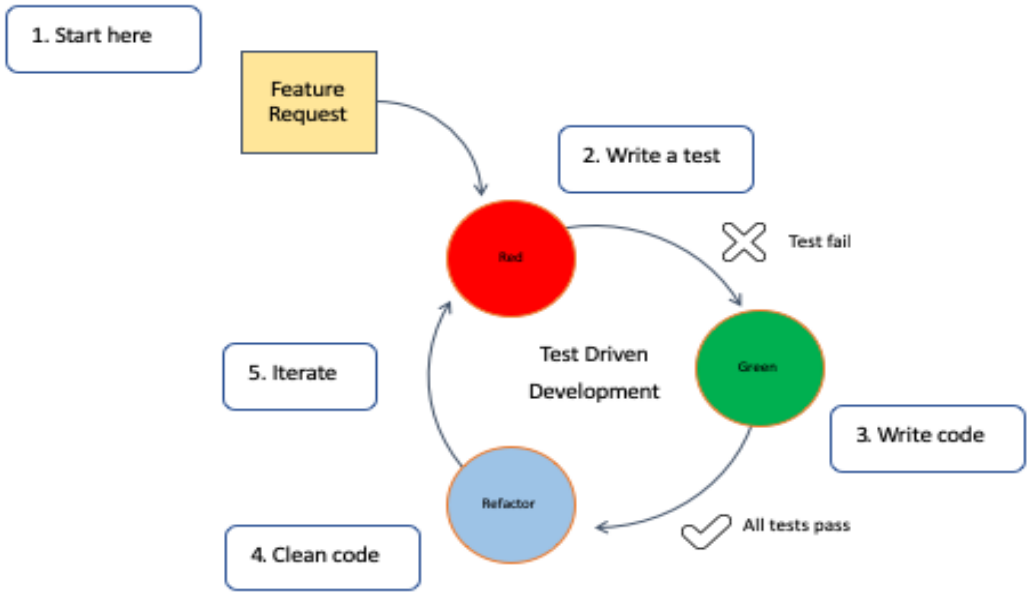
Assignment 1

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) is a software development methodology that emphasizes writing tests before writing the actual code. The main idea behind TDD is to create a feedback loop that guides the development process and helps ensure the reliability and correctness of the software being developed. TDD follows a cyclical pattern, often referred to as the "Red-Green-Refactor" cycle, which consists of the following steps:



TDD Cycle

1. **Red**: In this phase, you start by writing a test that defines the desired behaviour or functionality of a specific piece of code. Initially, this test will fail because the corresponding code hasn't been written yet. This failing test is often referred to as a "red" test.

2. **Green**: Once you have a failing test, your next step is to write the minimum amount of code necessary to make the test pass. This code may not be perfect or efficient; the goal is to satisfy the test's conditions and make it pass. When the test passes, it becomes a "green" test, indicating that the desired functionality has been implemented.

3. **Refactor**: After making the test pass, you can improve the code's design, structure, and efficiency while keeping the test green. Refactoring involves making changes to the code without changing its external behaviour. The tests act as a safety net, helping you catch unintended side effects of your changes.

The TDD cycle is then repeated, starting with the creation of a new test for the next piece of functionality. This process helps ensure that your code is always backed by tests, making it easier to catch bugs and regressions as the codebase evolves.

Let's explore TDD through a simple example: Imagine we need to create a simple function that adds two numbers.

Here's how we can apply TDD to develop this function:

1. **Write the Test:** We start by writing a test that defines the behaviour of the `add` function. In a file named `AdderTest.kt`, we could have:

```
import org.junit.Test
import kotlin.test.assertEquals

class AdderTest {
    @Test
    fun `test addition`() {
        val result = 2.add(3)
        assertEquals(5, result)
```

```
    }
}
```

Notice that the `add` function is referenced even though it's not defined yet.

2. Implement the Minimum Code: Next, we implement the `add` function in a separate file named `Adder.kt`:

```
fun Int.add(a: Int): Int = this + a
```

The goal here is to make the test pass as quickly as possible.

3. Run the Test:

Run the tests, and the `test addition` test should pass, confirming that the `add` function works as expected.

4. Adding More Tests: To further validate sum functionality with more test cases:

```
@Test
    fun `test addition with overflow`() {
        val result = 2.add(Int.MAX_VALUE)
        assertTrue(result < 0)
    }
```

5.Refactor: We can refactor code based on requirement, whether we want overflow value or can filter/validate input values.

Benefits of TDD: Here are the benefits of Test-Driven Development (TDD):

1. Improved Code Quality: TDD enforces a focus on writing clean, maintainable, and modular code from the outset. By writing tests first, developers must think critically about the design and architecture of their code, leading to higher code quality and fewer design flaws.

2. Reduced Bugs and Defects: With TDD, bugs and defects are identified early in the development process as tests are written before code implementation. This proactive approach helps catch issues before they propagate and become more challenging and costly to fix.

3. Faster Debugging and Development: TDD accelerates the debugging process by pinpointing issues in smaller, isolated sections of code. This leads to quicker identification and resolution of problems, ultimately speeding up the overall development cycle.

4. Confident Refactoring: TDD provides the confidence to refactor code without fear of breaking existing functionality. If tests pass after refactoring, developers can be assured that their changes haven't introduced new defects, resulting in a more maintainable and adaptable codebase.

These benefits make Test-Driven Development a powerful practice for creating high-quality software with fewer defects, faster development cycles, and increased developer confidence.

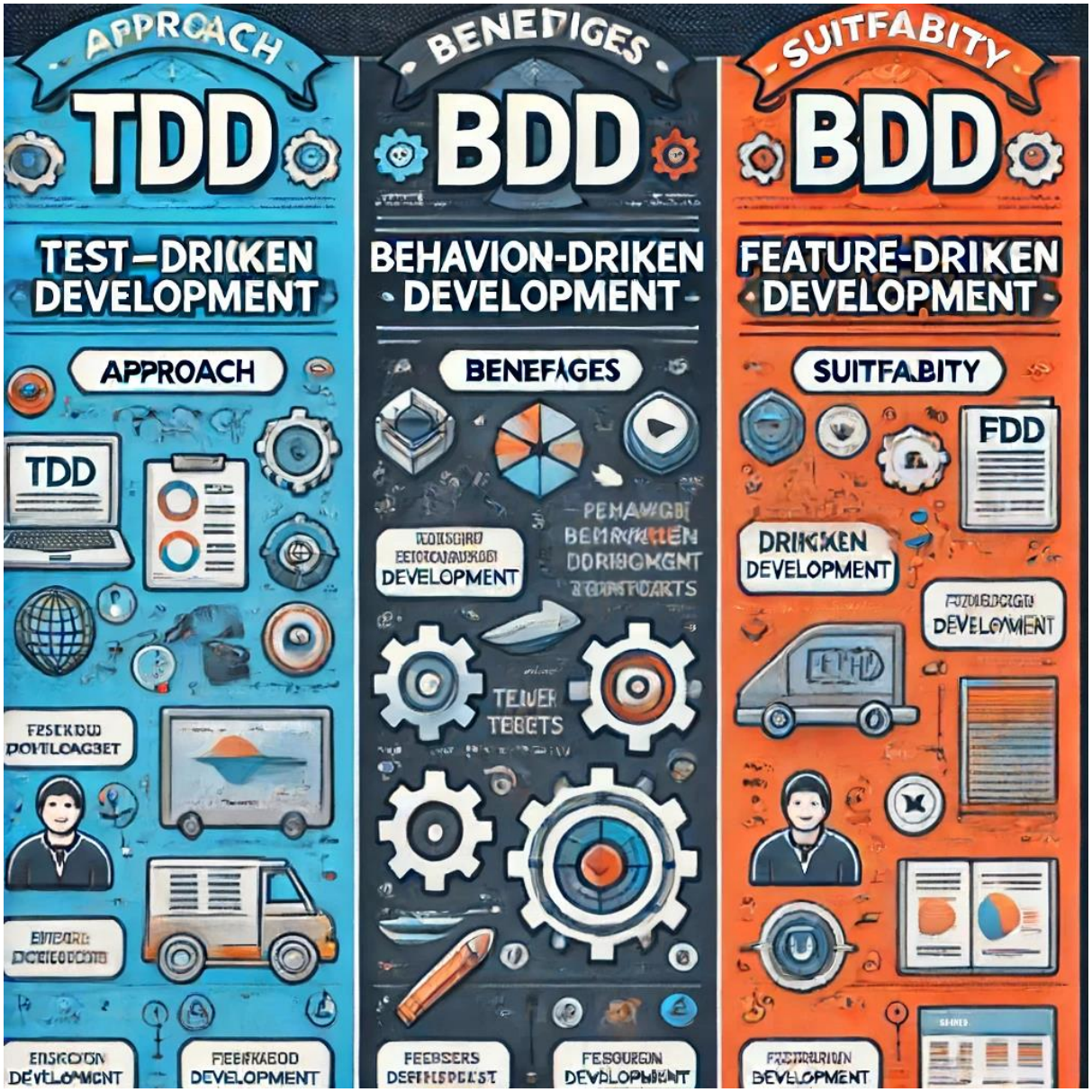
How TDD Fosters Software Reliability:

Continuous Testing: Regular testing ensures each code change is validated.
Early Detection of Issues: Bugs are caught earlier in the development cycle.
Consistent Quality: Maintains a high standard of code quality and reliability.

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.

Here is the comparative infographic of TDD, BDD, and FDD methodologies. It illustrates their unique approaches, benefits, and suitability for different software development contexts, using visuals and colour coding for clarity. The infographic is designed to enhance understanding with clear headings and bullet points for easy reading.



Creating a comparative infographic for TDD (Test-Driven Development), BDD (Behavior-Driven Development), and FDD (Feature-Driven Development) is a great way to visually illustrate their unique approaches, benefits, and suitability for different software development contexts. Here's how you can structure such an infographic:

Infographic Title: Comparative Analysis of TDD, BDD, and FDD

1. Introduction: Overview of Methodologies

- Brief introduction to TDD, BDD, and FDD.
- Importance of each methodology in software development.

2. Methodology Comparison: Visual Representation

Test-Driven Development (TDD):

- Definition: Write tests before writing code.
- Workflow: Red-Green-Refactor cycle.
- Benefits:
    - Ensures code coverage and reliability.
    - Encourages modular and loosely coupled design.
- Suitability: Best for iterative development and maintaining code quality.

Behavior-Driven Development (BDD):

- Definition: Focuses on behaviors and outcomes.
- Workflow: Define scenarios using Given-When-Then format.
- Benefits:
    - Improves collaboration between developers, QA, and stakeholders.
    - Aligns development with business goals and user expectations.
- Suitability: Ideal for complex projects with diverse stakeholders and for defining system behaviors.

Feature-Driven Development (FDD):

- Definition: Iterative and incremental development based on features.
- Workflow: Develop features in short iterations.

- Benefits:
    - Emphasizes on tangible deliverables.
    - Provides clear accountability and progress tracking.
- Suitability: Suitable for large teams and projects requiring clear feature prioritization and management.

3. Visual Representation: Comparative Features

- Use icons or symbols to represent key features of each methodology:
    - TDD: Testing-first approach, red-green-refactor cycle.
    - BDD: Behavior-driven scenarios, Given-When-Then format.
    - FDD: Feature-centric development, iterative delivery.

4. Comparison Chart: Benefits and Suitability

- Create a matrix or table to compare:
- Benefits: Reliability, collaboration, accountability.
- Suitability: Project size, team size, complexity.

5. Conclusion: Choosing the Right Methodology

- Summarize when to choose each methodology based on project requirements.
- Consider hybrid approaches based on specific project needs.

Design Tips:

- Use contrasting colours for each methodology to visually differentiate them.
- Utilize icons, charts, and graphs to enhance clarity and engagement.
- Keep text concise and focused on key points to maintain readability.

By following this structure, you can create a comprehensive infographic that effectively communicates the differences, benefits, and contexts of TDD, BDD, and FDD in software development.