alok.giri@ncit.edu.np

What is TDD?

Definition:

Test-Driven Development (TDD) is a software development approach where developers **write tests before writing the actual code** that satisfies those tests.

Core Idea:

- Tests drive the design and structure of the software.
- You only write enough code to pass the test.

Key Characteristics:

- Focus on small, iterative development
- High level of test coverage
- Refactoring happens with confidence

TDD vs Traditional Testing

The TDD Cycle – Red → Green → Refactor

Overview:

TDD follows a **three-step cycle** for every feature or function:

1. **Red** – Write a failing test

2. **Green** – Write just enough code to make the test pass 3.
Refactor – Improve the code without changing functionality

Step 1 – RED: Write a Failing Test

Goal:

Start by writing a **test for a specific behavior or requirement**. The test should **fail initially** because the corresponding code doesn't exist yet.

Why?

● Confirms that the test detects missing or incorrect functionality ●
Prevents false positives

Example (Python):

python

```
    def test_addition():
 assert add(2, 3) == 5 # The function 'add' is not
```

```
implemented yet
```

Step 2 – GREEN: Make the Test Pass

Goal:

Write **just enough code** to pass the test. Avoid over engineering or adding unnecessary features.

Focus:

- Minimal implementation
- Avoid premature optimization

Example (Python):

python

```
def add(a, b):
 return a + b
```

Important: Don't worry about elegance or edge cases at this point. Just make the test pass.

Step 3 – REFACTOR: Improve the Code

Goal:

Clean and improve the implementation without breaking any tests.

Common Refactorings:

- Rename variables
- Simplify logic
- Remove duplication
- Apply design patterns

Safety Net:

All tests remain **green**, ensuring that behavior hasn't changed.

Benefits of TDD

1. **Higher Code Quality:**
    - Forces developers to consider use cases and edge cases early.
2. **Less Debugging:**
    - Issues are caught immediately during development.
3. **Better Design:**
    - Encourages modular, loosely-coupled code.
4. **Improved Developer Confidence:**
    - Easy to refactor and extend code safely.
5. **Living Documentation:**
    - Tests describe what the code is supposed to do.
6. **Faster Feedback Loop:**
    - Errors are detected quickly and fixed early.

Challenges of TDD

1. **Initial Learning Curve:**
   ○ Requires change in mindset and discipline.

2. **Slower Start:**
   ○ Initial development may feel slower due to writing tests first.

3. **Over-Testing:**
   ○ Risk of writing tests for trivial code.

4. **Refactoring Tests:**
   ○ Test maintenance can be time-consuming if code changes frequently.

5. **Difficult for UI/UX Logic:**
   ○ Not all layers (like UI/UX) are easily testable in TDD fashion.

When to Use and Avoid TDD

Use TDD:

- Backend services
- Algorithms and core business logic
- APIs and SDKs
- Applications requiring long-term maintenance

Avoid TDD:

- Rapid prototyping
- UI/UX-heavy design where visual feedback is key
- One-off scripts or disposable code

What is BDD?

Definition:

Behavior-Driven Development (BDD) is a software development practice that encourages **collaboration between developers, QA, and non-technical stakeholders** to define the behavior of a system in **plain language**.

Core Idea:

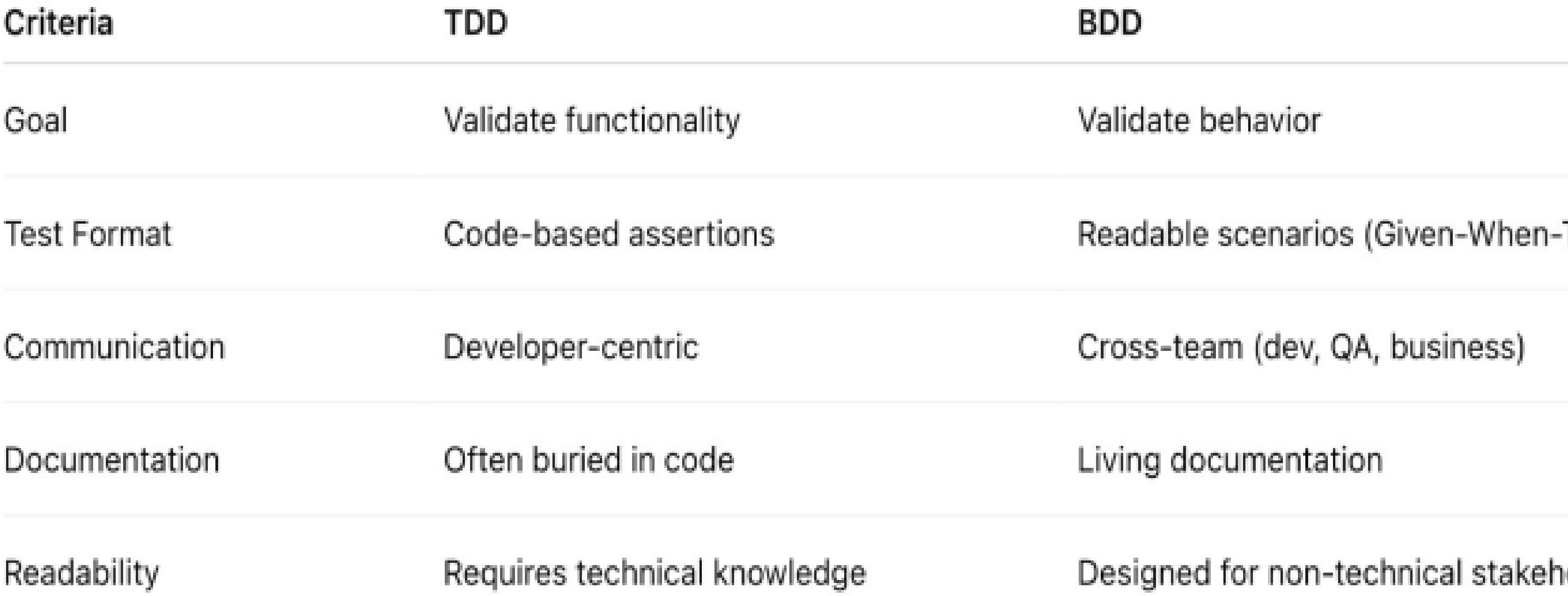
- Uses **natural language constructs** to describe software behavior
- Focuses on the **expected outcome** rather than implementation

Key Principle:

> If we can describe it clearly, we can build it correctly.

Evolution from TDD to BDD

BDD vs TDD



Criteria	TDD	BDD
Goal	Validate functionality	Validate behavior
Test Format	Code-based assertions	Readable scenarios (Given-When-T
Communication	Developer-centric	Cross-team (dev, QA, business)
Documentation	Often buried in code	Living documentation
Readability	Requires technical knowledge	Designed for non-technical stakeh

Structure of BDD – Given, When, Then

The Gherkin Syntax:

- **Given:** Initial context or precondition
- **When:** Event or action taken
- **Then:** Expected outcome or result

Advantages:

- Makes requirements **testable**
- Easy for all stakeholders to **understand and verify**

Writing Effective BDD Scenarios Tips

for Clear, Concise, and Testable Scenarios:

1. **Focus on business value:**
    - Scenarios should reflect real user behavior.

2. **Use simple and consistent language:** ○ Avoid technical jargon.
3. **Keep scenarios short and atomic:**
   ○ One behavior per scenario.
4. **Avoid overlapping conditions:**
   ○ Separate out unrelated behaviors.
5. **Use roles and actions clearly:**
   ○ Who is doing what and why?

Benefits of BDD

1. **Enhanced collaboration:**
   ○ Aligns technical and non-technical stakeholders.
2. **Improved clarity of requirements:**
   ○ Business rules are explicitly defined in scenarios.
3. **Living documentation:**

- Scenarios serve as executable specs and up-to-date documentation.

4. **Reduced miscommunication:**
   - Shared understanding through common language.
5. **Faster test creation:**
   - Reusable steps and readable formats make test writing easier.

Challenges of BDD

1. **Initial setup and training:**
   - Requires learning tools and process changes.
2. **Overhead for simple projects:**
   - May be too heavy for small scripts or prototypes.
3. **Scenarios can become too detailed:**
   - Risk of becoming verbose and brittle if over-specified.

4. **Maintenance of steps:**
    - Common step definitions must be well-organized.
5. **Misuse as a testing tool only:**
    - BDD is about collaboration and behavior, not just testing.

When to Use BDD

Best suited for:

- Agile teams with frequent collaboration
- Feature-rich applications with evolving requirements
- Projects where behavior is key to success (e.g., user workflows)

Avoid or limit BDD when:

- Team lacks communication with business users
- Small, one-off tools or utilities

- When requirements are too vague or volatile to formalize

Overview of Collaborative Programming

Definition:

Collaborative programming involves **two or more developers working together** in real-time on the same code base.

Why it matters:

- Encourages **continuous feedback**
- Promotes **collective ownership**
- Enhances **code quality and team learning**

Forms:

- **Pair Programming:** 2 developers
- **Mob Programming:** 3 or more developers (often the whole team)

What is Pair Programming?

Definition:

Pair programming is a development technique where **two developers work together at one workstation** on the same task.

- One developer types the code (**Driver**)
- The other reviews and guides (**Navigator**)
- They frequently **switch roles**

Quote:

> “Pair programming is a dialog between two people trying to simultaneously program and understand the problem and its solution.” – Laurie Williams

Roles in Pair Programming

1. Driver

- Writes the code
- Focuses on **syntax and implementation**
- Pays attention to immediate tasks

2. Navigator

- Reviews each line as it’s written
- Thinks about **design, strategy, and direction** ●
Spots potential bugs, edge cases, and improvements

Modes of Pair Programming

1. Expert–Novice

- Experienced developer guides a beginner
- Great for **mentorship and onboarding**
- Balance required to keep the novice engaged

2. Ping-Pong Pairing

- One writes a **failing test**, the other writes **code to pass it**
- Then switch roles
- Follows **TDD** style

3. Remote Pairing

- Pairs work remotely using tools like **VSCode Live Share**, **Tuple**, **CodeTogether**
- Requires clear audio, fast internet, and screen-sharing

Benefits of Pair Programming

1. **Improved code quality:**
    - Two sets of eyes reduce bugs and increase clarity

2. **Faster knowledge sharing:**

   - Developers learn from each other

3. **Better design decisions:**
   - Real-time discussion encourages thoughtful solutions

4. **Increased team cohesion:**
   - Builds communication and mutual trust

5. **Reduced bottlenecks:**
   - No single point of failure or dependency on one person

Challenges of Pair Programming

1. **Initial drop in productivity:**
   - Can feel slower until the team adjusts

2. **Personality mismatches:**
   - Requires interpersonal skills and mutual respect

3. **Fatigue:**
    - Pairing can be mentally intense without breaks

4. **Not all tasks are suitable:**
    - Trivial or repetitive tasks may not benefit

5. **Scheduling difficulties:**
    - Matching availability can be tough in distributed teams

What is Mob Programming?

Definition:

Mob programming is a style of programming where **the whole team works together on the same task, at the same time, on the same computer**.

- One person is the **Driver**
- Everyone else acts as **Navigators**

- All decisions are made **collaboratively**

Quote:

> “All the brilliant minds working on the same thing, at the same time, in the same space, and at the same computer.” – Woody Zuill

Roles in Mob Programming

1. Driver

- The only person typing
- Implements what is discussed
- **Does not make decisions alone**

2. Navigators

- Everyone else

- Discuss and guide the Driver
- Suggest improvements, spot issues, and strategize

Rotation Tip: Rotate the Driver every 10–15 minutes

Remote and In-Person Mob Programming

In-Person:

- One workstation with shared screen
- Use timer for rotation

Remote:

- Use tools like:
  - **Zoom/Teams/Google Meet** for voice/video
  - **VSCode Live Share**
  - **Miro or digital whiteboards** for brainstorming

Best Practices:

- Strong facilitation
- Clear goals and structure
- Frequent short breaks

Benefits of Mob Programming

1. **Rapid knowledge sharing:**
    - Everyone learns together
2. **Higher code quality:**
    - Multiple reviewers catch issues early
3. **Shared ownership:**
    - Everyone understands the code base
4. **Fewer interruptions:**
    - Team is aligned and focused

5. **Real-time mentoring:**
    - Juniors learn from seniors instantly

Challenges of Mob Programming

1. **Logistics and time zones:**
    - Harder to schedule full-team sessions
2. **Overcommunication:**
    - Requires structure to avoid chaos
3. **Burnout risk:**
    - Intense focus for extended periods
4. **Cost concerns:**
    - Appears expensive (entire team working on one thing)
5. **Requires strong facilitation:**
    - Otherwise can become inefficient

When to Use Pair vs Mob Programming

Use Pair Programming When:

- Two developers can work efficiently on a story
- Mentoring or onboarding a teammate
- Refactoring or debugging a specific feature

Use Mob Programming When:

● Complex or high-risk feature needs many inputs ●
Onboarding a new team or aligning understanding ●
Architectural decisions or major codebase changes

Why Focus on Maintainability?

Definition of Maintainability:

- The ease with which a software system can be understood, changed, and extended.

Why It Matters:

- Code is read far more often than it is written.
- Long-term cost of software is mostly **maintenance**, not initial development.

What is Refactoring?

Definition:

Refactoring is the process of **improving the internal structure of code** without changing its external behavior.

Key Goals:

- Enhance readability
- Improve design
- Reduce complexity
- Remove code smells

> *"Refactoring is like cleaning up your workspace—nothing changes in function, but everything becomes easier to work with."* – Martin Fowler

Technique	Purpose
Rename Variable/Method	Improve clarity and naming
Extract Method	Break down long functions
Inline Method/Variable	Remove unnecessary abstraction
Remove Dead Code	Eliminate unused or unreachable co
Replace Magic Numbers	Use named constants for readabilit
Encapsulate Fields	Use getters/setters to protect state
Simplify Conditionals	Use guard clauses, remove nesting
Split Large Classes	Follow SRP and reduce coupling

Refactoring Techniques

When Should You Refactor?

Best Times to Refactor:

- **Before adding a new feature**: Clean up the area you'll work on.
- **While fixing bugs**: Understand and improve the faulty area.
- **After code reviews**: Address quality feedback.
- **During TDD cycles**: The "Refactor" stage in Red-Green-Refactor.

Avoid:

- Refactoring without tests or clear understanding
- Big bang refactors with no incremental checkpoints

Benefits of Refactoring

1. **Improved readability and understanding**
2. **Lower technical debt**
3. **Easier debugging and modification**
4. **Improved performance (when optimizing)** 5. **Better team collaboration through clean, consistent code**

What is a Code Review?

Definition:

A code review is a **systematic examination of source code** by peers to identify bugs, improve code quality, and share knowledge.

Purpose:

- Find issues **before they reach production**
- Encourage **best practices**
- Promote **team-wide standards**

Types of Code Reviews

1. Synchronous Reviews:

- Real-time discussion via **pairing or walkthrough meetings**
- Tools: Screen sharing, IDE collaboration (e.g., Live Share)

2. Asynchronous Reviews:

- Developer submits a pull request (PR); reviewers comment later
- Common in distributed teams

- Tools: GitHub, GitLab, Bitbucket, Phabricator

Reviewer Mindset

Constructive:

- Provide **specific and actionable** feedback.
- Avoid sarcasm or nitpicking.

Respectful:

- Use inclusive and polite language.
- Focus on the code, **not the coder**.

Collaborative:

- Ask questions instead of making assumptions.
- Offer suggestions, not demands.

Examples:

- 🚫 “This code is awful.”
  ✅ “Could we simplify this method for readability?”

Author Mindset

Open to Feedback:

- Assume good intent from reviewers.
- Be willing to learn and adapt.

Prepare for Review:

- Run all tests
- Write meaningful commit messages
- Add comments for complex logic

Communicate Clearly:

- Tag reviewers appropriately ● Explain

reasoning behind tricky code

Benefits of Code Reviews

1. **Early bug detection**
2. **Improved code quality**
3. **Knowledge sharing among team members**
4. **Mentorship and skill development** 5. **Consistent coding standards**
6. **Team accountability and cohesion**

Refactoring + Code Review = Healthy Codebase

Why They Work Together:

- Refactoring improves code quality **internally**
- Code reviews enforce quality **externally**
- Together, they ensure the system remains clean, scalable, and

robust

Best Practice:

- Refactor **before submitting** code for review
- Use review comments as **refactoring triggers**