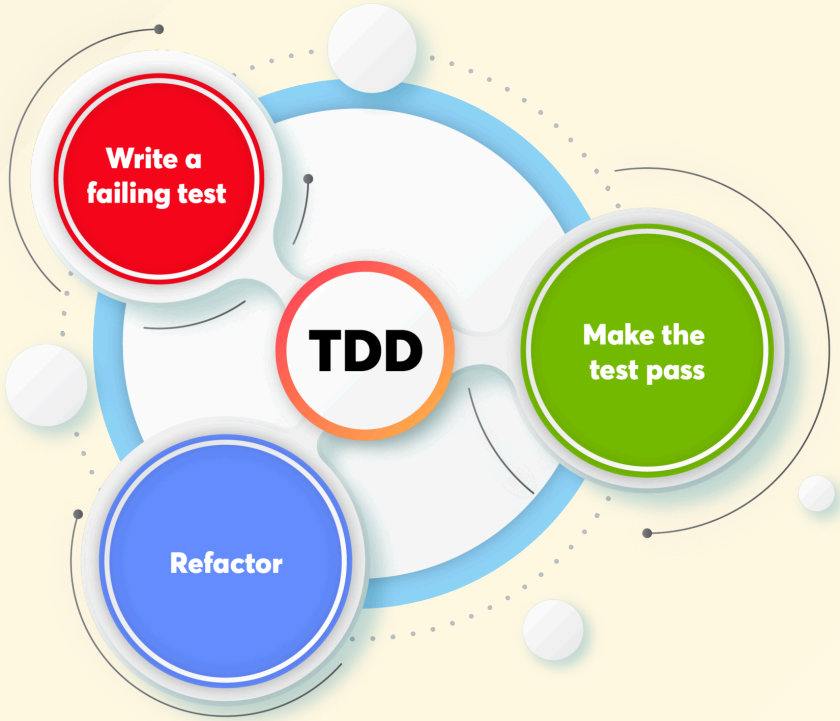
# Test Driven Development (TDD)



# What is Test-Driven Development (TDD)?

- Test-Driven Development (TDD) is a technique for building software that guides software development by writing tests.
- It was developed by Kent Beck in the late 1990's as part of `Extreme Programming`.
- Core principles: Red, Green, Refactor

# Why TDD is important in modern software development?

# 1. Ensures Code Quality

- **Prevents bugs early:** Writing tests before code helps catch bugs early in the development process, reducing the likelihood of defects in production.
- **Promotes clean code:** TDD encourages writing small, testable, and maintainable pieces of code, leading to a cleaner and more modular codebase.

## 2. Facilitates Better Design

- **Testable architecture:** TDD pushes developers to design software with testability in mind, leading to better overall software architecture.
- **Focus on requirements:** Tests are written based on user requirements, ensuring that the code aligns with what the user actually needs.

## 3. Increases Developer Confidence

- **Safe refactoring:** With a comprehensive test suite, developers can refactor code confidently, knowing that any breaking changes will be caught by the tests.
- **Documentation:** Tests serve as living documentation, making it easier for new developers to understand the codebase.

## 4. Improves Productivity

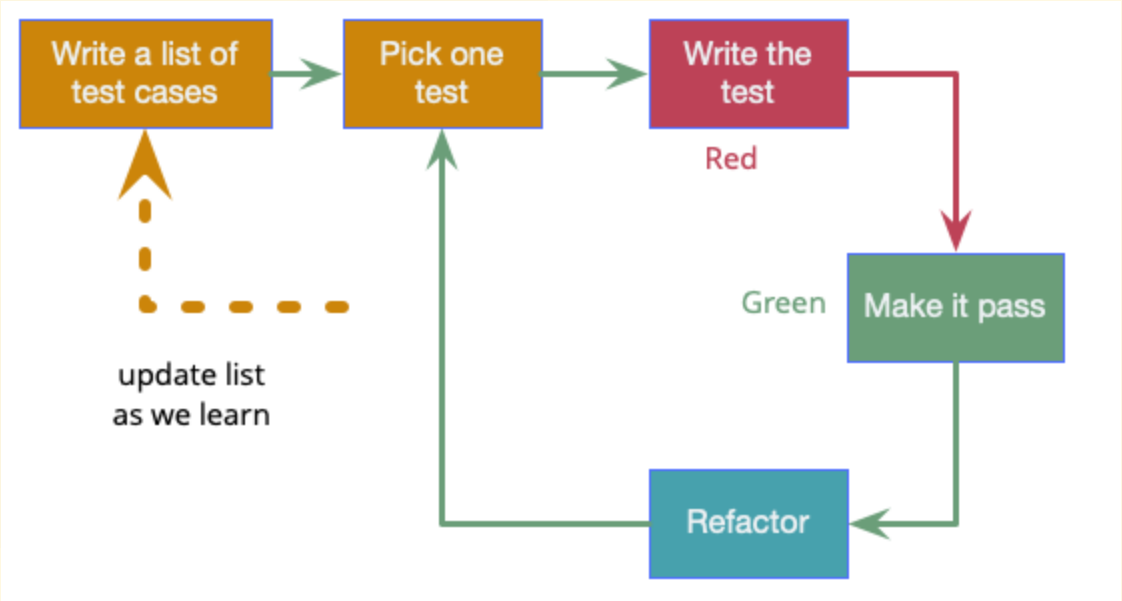
- **Reduced debugging time:** With fewer bugs reaching production, developers spend less time debugging and more time building features.
- **Continuous feedback:** Immediate feedback from tests allows developers to identify issues and make adjustments quickly.

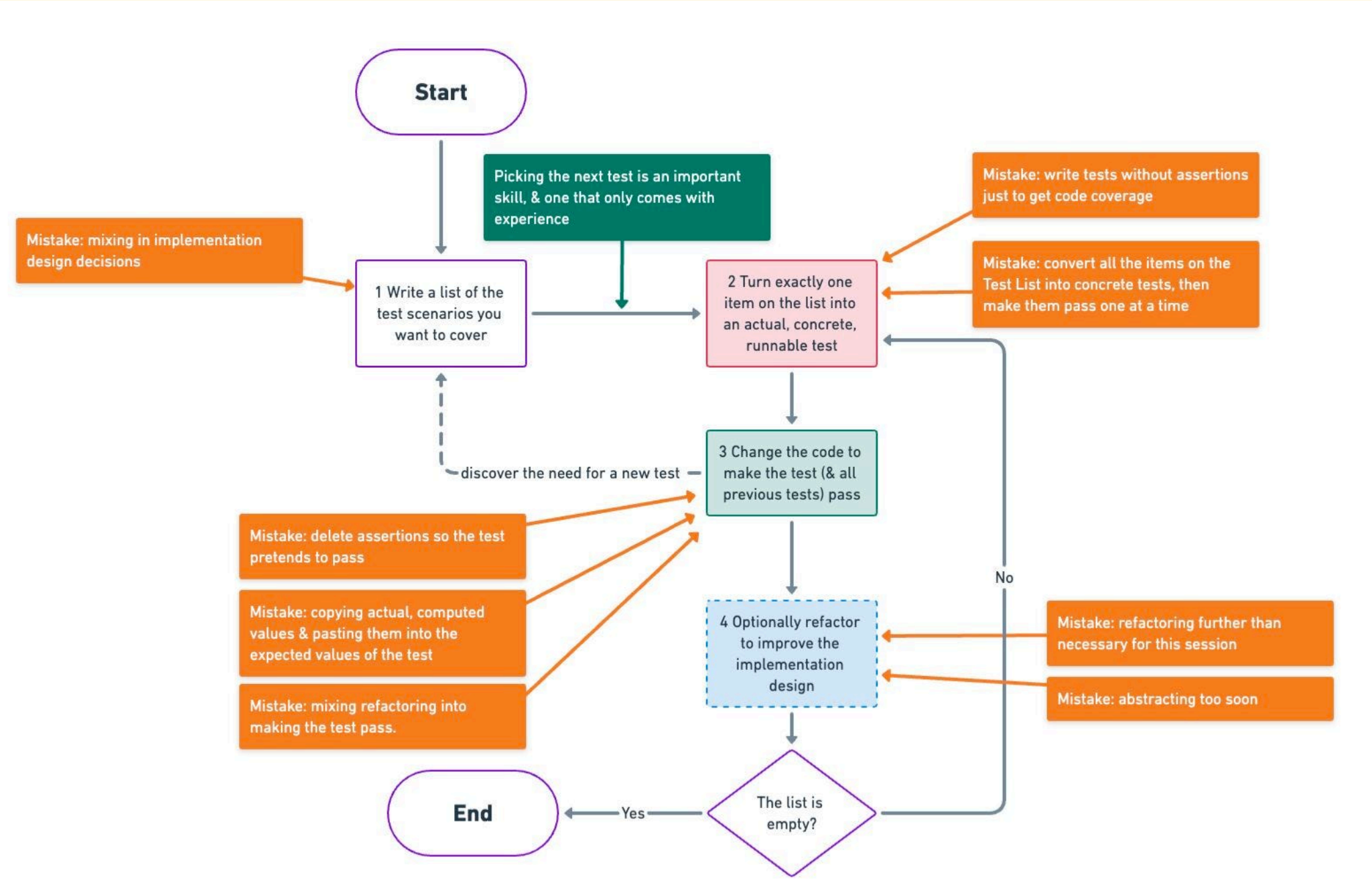
## 5. Supports Continuous Integration (CI)

- **Automated testing:** TDD fits seamlessly with CI/CD pipelines, where tests are automatically run with each code change, ensuring continuous code quality.
- **Prevents regression:** Tests help prevent regression by ensuring that new code changes don’t break existing functionality.

# Essence of TDD

- Write a test for the next bit of functionality you want to add.
- Write the functional code until the test passes.
- Refactor both new and old code to make it well structured.




Start
Picking the next test is an important skill, & one that only comes with experience
Mistake: mixing in implementation design decisions
Mistake: write tests without assertions just to get code coverage
Mistake: convert all the items on the Test List into concrete tests, then make them pass one at a time
1 Write a list of the test scenarios you want to cover
2 Turn exactly one item on the list into an actual, concrete, runnable test
discover the need for a new test
3 Change the code to make the test (& all previous tests) pass
Mistake: delete assertions so the test pretends to pass
Mistake: copying actual, computed values & pasting them into the expected values of the test
Mistake: mixing refactoring into making the test pass.
No
4 Optionally refactor to improve the implementation design
Mistake: refactoring further than necessary for this session
Mistake: abstracting too soon
End
Yes
The list is empty?

# TDD Best Practices

1. **Start with Simple Tests**
    - Begin with the simplest test case that can possibly fail. This keeps the focus on incremental development and prevents overengineering.
2. **Follow the Red-Green-Refactor Cycle**
    - Always stick to the TDD cycle: write a failing test (Red), write code to pass the test (Green), and then refactor while ensuring the test still passes. This keeps development disciplined and code clean.

3. **Write Small, Focused Tests**
    - Ensure each test covers a single unit of functionality. This makes tests easier to understand, maintain, and debug when they fail.
4. **Use Descriptive Test Names**
    - Name your tests based on what they verify, e.g., `shouldReturnTrueWhenUserIsAuthenticated`. This makes it easier to understand test failures without diving into the implementation.

5. **Keep Tests Independent**
    - Tests should be independent of each other; one test's failure should not cause others to fail. This ensures that tests are reliable and consistent.
6. **Refactor Code and Tests Regularly**
    - Refactor both your production code and tests to improve clarity, performance, and maintainability. Don’t shy away from improving test code structure.

7. **Test Edge Cases and Exceptions**
    - Include tests for edge cases, unexpected inputs, and exceptions to ensure robustness of the code under different scenarios.
8. **Leverage Mocks and Stubs Wisely**
    - Use mocks and stubs to isolate the unit under test, but avoid overusing them as they can make tests harder to understand and maintain.

9. **Maintain a High Test Coverage**
    - Aim for high test coverage but prioritize meaningful tests over hitting arbitrary coverage percentages. Quality over quantity is key.
10. **Integrate with Continuous Integration (CI)**
    - Integrate your tests into a CI pipeline to ensure tests are run automatically on each commit, catching issues early.

# TDD Anti-Patterns

1. **The Liar Test**
    - Writing tests that always pass, even when the code is incorrect. This defeats the purpose of TDD and gives a false sense of security.
2. **Over-Mocking**
    - Using too many mocks or stubs can make tests brittle and overly coupled to implementation details, making refactoring difficult.

3. **Testing Everything**
    - Testing trivial getters, setters, or trivial one-liner methods adds little value but increases maintenance overhead. Focus on testing behavior, not implementation.
4. **Skipping the Refactor Step**
    - Skipping the refactor phase leads to accumulating technical debt, resulting in complex and hard-to-maintain code.

5. **Ignoring Test Failures**
    - Ignoring or commenting out failing tests instead of fixing the underlying issue leads to code rot and diminishes the reliability of the test suite.
6. **Test Duplication**
    - Writing multiple tests that verify the same behavior adds unnecessary maintenance burden without increasing test coverage.

7. **Big Bang TDD**
    - Writing large chunks of code between tests, which goes against the incremental nature of TDD. This can lead to more complex debugging and less focused tests.
8. **Testing Non-Deterministic Behavior**
    - Writing tests for non-deterministic or flaky behavior (like timing-dependent features) can lead to unreliable tests that frequently fail without cause.

9. **Not Refactoring Tests**
    - Just like production code, tests need refactoring to stay maintainable. Ignoring test refactoring can lead to a cluttered and hard-to-understand test suite.
10. **Test Last Development**
    - Writing tests after the code is implemented and calling it TDD. This negates the benefits of TDD and often leads to poor test coverage and quality.

# Hands On: Let's Code!

- Code Conway's **Game of Life** using TDD
  - Java 17
  - JUnit 5
  - Gradle 8.5

# Conway's Game of Life

- A cellular automaton devised by the British mathematician John Horton Conway in 1970.
- It is a zero-player game, meaning that its evolution is determined by its initial state, requiring no further input.

- The game takes place on a 2D grid of cells
- Each cell can be in one of two possible states:
  - `alive` (1) or
  - `dead` (0).
- The grid evolves through a series of discrete time steps according to a set of simple rules.
- Online Demo: Game of Life

# Rules:

1. **Birth**: A dead cell with exactly three live neighbors becomes a live cell (birth).
2. **Survival**: A live cell with two or three live neighbors stays alive (survival).
3. **Death by Isolation**: A live cell with fewer than two live neighbors dies (underpopulation).
4. **Death by Overcrowding**: A live cell with more than three live neighbors dies (overcrowding).

# Grid and Neighbors:

- The grid is infinite, but in practical implementations, it is often bounded with wraparound (toroidal) or fixed boundaries.
- Each cell has eight neighbors: horizontally, vertically, and diagonally adjacent cells.

# Input

- **Initial Grid Configuration**: A 2D array (matrix) where each element is either `1` (alive) or `0` (dead).
- **Number of Generations**: An integer specifying how many generations the game should simulate.

# Output

- The grid after the specified number of generations.

# Test Scenarios

# Cell Behavior Tests

- A live cell with fewer than two live neighbors dies (**underpopulation**)
- A live cell with two or three live neighbors survives (**survival**) [P]
- A live cell with more than three live neighbors dies (**overcrowding**) [P]
- A dead cell with exactly three live neighbors becomes a live cell (**reproduction**)

# Grid Initialization and State Tests

- Initialize a grid with all dead cells and confirm its dimensions and state
- Set and retrieve the state of a specific cell in the grid [P]

# Grid Boundaries

- Should determine if a given cell is within the grid [P]
- Should determine if a given cell is outside the grid [P]

# Neighbors Calculation Tests

- Correctly calculate the number of live neighbors for a cell (middle of grid)
- Correctly calculate the number of live neighbors for an edge cell
- Correctly calculate the number of live neighbors for a corner cell
- Handle neighbor calculation with grid boundaries (dead cells outside the grid)

# Generation Transition Tests

- Apply rules to a single cell and update its state
- Apply rules to the entire grid and produce the next generation
- Verify that an isolated live cell dies in the next generation [P]
- Verify that a stable structure (e.g., block) remains unchanged over generations [P]
- Verify that an oscillating structure (e.g., blinker) oscillates correctly [P]

> A blinker (a line of three live cells) should oscillate between horizontal and vertical.

# Edge Cases and Larger Structures

- An empty grid should remain empty over generations.
- Ensure that all cells die if the grid is full of live cells due to overcrowding
- Test a glider pattern and verify its movement over multiple generations

# Special Conditions and Final Tests

- Ensure the game rules apply correctly on non-square grids (e.g., 3x5 grid)
- Verify that spaceship patterns move correctly across the grid
- Test with a known pattern that stabilizes after a few generations.

# End 🚀🚀🚀