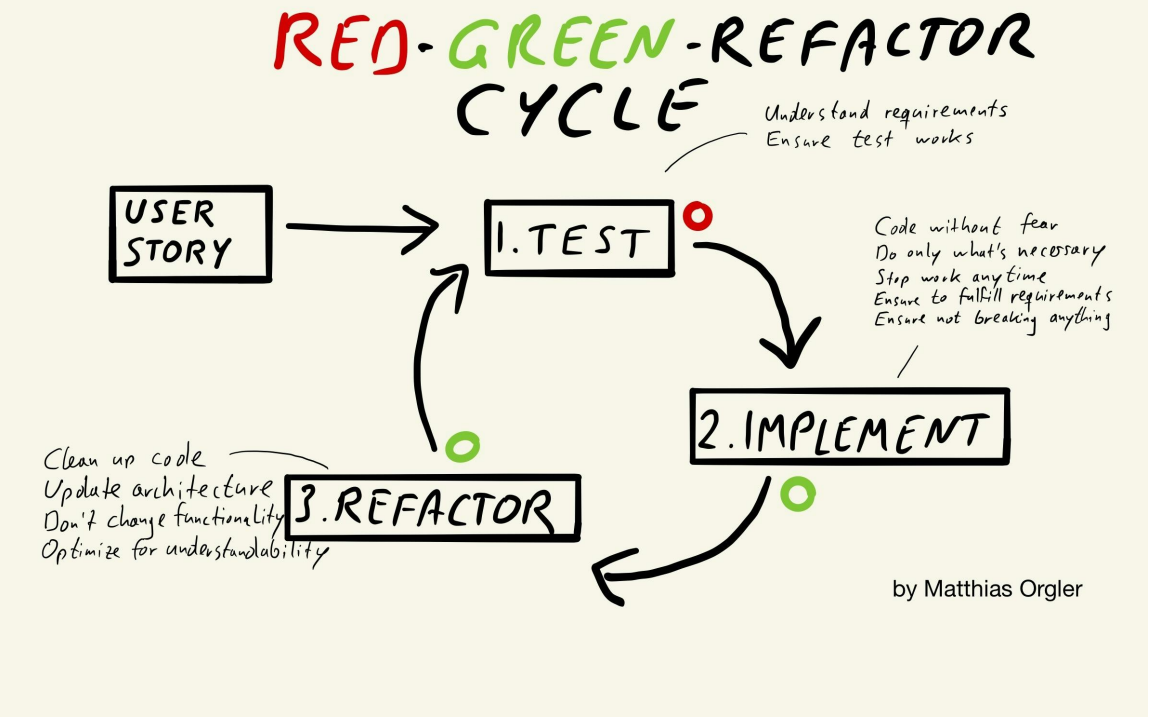
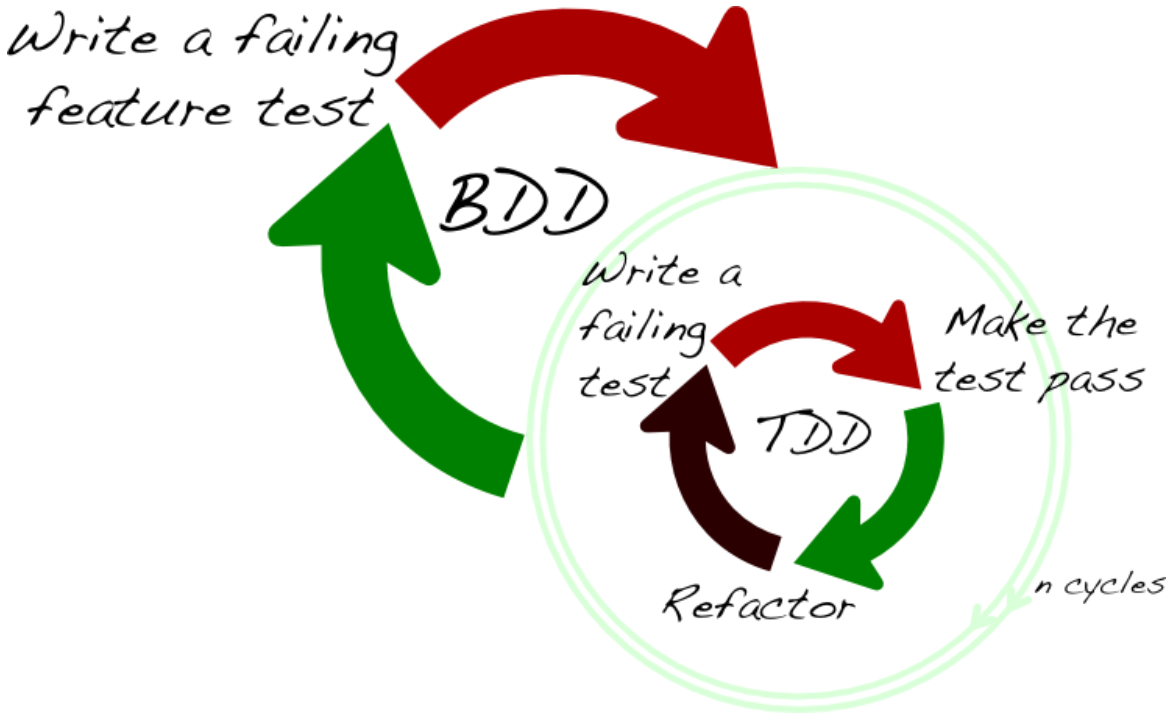
Test-Driven Development (TDD)

- A software development process where tests are written before the code that fulfills those tests.
- TDD follows a simple cycle of writing a test, writing code to make the test pass, and refactoring the code for optimization.
- **TDD Cycle**: Red → Green → Refactor.
  - **Red**: Write a failing test (test case does not pass initially).
  - **Green**: Write just enough code to make the test pass.
  - **Refactor**: Improve the code without changing its behavior.



▼ The TDD Cycle Explained

1. **Write a Test**: Focus on small, specific units of behavior (e.g., a method).
    - The test should be **failing** at this point (e.g., a test for a function that doesn't exist yet).
2. **Run the Test**: The test fails since the code doesn’t exist or is incomplete.
3. **Write Code**: Implement just enough code to make the test pass.
4. **Run the Test Again**: If it passes, you're ready to move on.
5. **Refactor**: Clean up the code for readability, performance, and maintainability.
    - Ensure that tests still pass after refactoring.



- **Behaviour-Driven Development (BDD)** is a software development methodology that extends Test-Driven Development (TDD) by encouraging collaboration between developers, testers, and non-technical stakeholders.
- BDD focuses on defining software behavior through examples written in natural language, often using a Given-When-Then format to specify how the system should behave.

▼ Why Use TDD?

- Catch bugs early by testing small pieces of code individually.
- Developers gain more confidence when refactoring since existing functionality is protected by tests.
- Writing tests first encourages modular and cleaner code because the code is written to satisfy the test.
- The tests themselves serve as documentation, showing how the code is expected to behave.
- When tests fail, it's easier to pinpoint the exact piece of functionality that broke.

Slide 4: The Red, Green, Refactor Process in Action

▼ **Phase 1: Red (Write a Failing Test First)**

- In the red phase, you write a unit test that defines a function's desired behavior before implementing it. Since the functionality does not exist yet, the test will fail.

```
// Red Phase: Failing Test
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class GradeBookTest {

    @Test
    void testAverageGrade() {
        GradeBook gradeBook = new GradeBook();
        gradeBook.addGrade(90);
        gradeBook.addGrade(80);
        gradeBook.addGrade(70);

        // Expecting the average of 90, 80, and 70 to be 8
        assertEquals(80, gradeBook.calculateAverage());
```

```
    }
}
```

▼ Phase 2: Green (Write Just Enough Code to Make the Test Pass)

- In the green phase, you implement the minimal amount of code to make the test pass.
- Now, the test will pass because we've implemented the `addGrade` and `calculateAverage` methods. The code works but can be refactored for readability and performance.

```
// Green Phase: Passing Test
import java.util.ArrayList;
import java.util.List;

class GradeBook {

    private List<Integer> grades = new ArrayList<>();

    public void addGrade(int grade) {
        grades.add(grade);
    }

    public double calculateAverage() {
        int sum = 0;
        for (int grade : grades) {
            sum += grade;
        }
        return (double) sum / grades.size();
    }
}
```

▼ Phase 3: Refactor (Improve Code Without Changing Functionality)

- One potential improvement is **using Java Streams** to make the `calculateAverage` method cleaner and more concise. Additionally, we can check for edge cases, such as when there are no grades in the `GradeBook`.

- **Edge Case Handling:** We added logic to return `0` if no grades have been added, preventing division by zero.
- **Stream API:** The `for-loop` was replaced with a more concise and modern approach using Java Streams (`mapToInt` and `average()`), improving code readability and performance.

```
// Refactor Phase: Improved Code
import java.util.ArrayList;
import java.util.List;

class GradeBook {

    private List<Integer> grades = new ArrayList<>();

    public void addGrade(int grade) {
        grades.add(grade);
    }

    public double calculateAverage() {
        if (grades.isEmpty()) {
            return 0; // Handle case when no grades have
been added
        }

        // Use Java Streams for a cleaner calculation
        return grades.stream()
                     .mapToInt(Integer::intValue)
                     .average()
                     .orElse(0);
    }
}
```

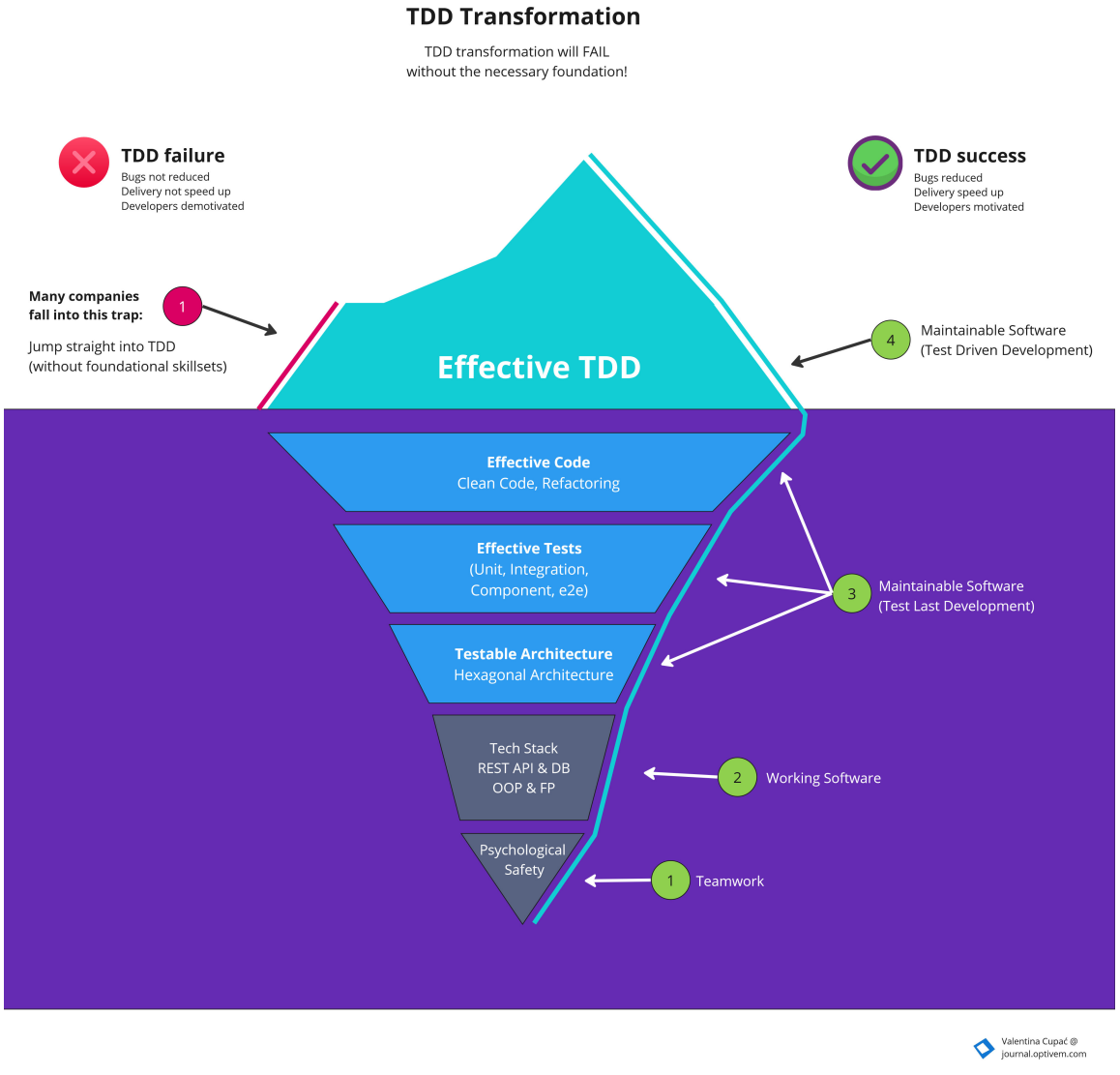
▼ Best Practices for TDD

- **Write Tests First**: Always write your tests before writing the code.
- **Keep Tests Small**: Focus on testing one behavior at a time.

- **Frequent Testing**: Run tests frequently to get immediate feedback on changes.
- **Refactor Often**: Don’t skip the refactor step; clean code is easier to maintain.
- **Avoid Over-Mocking**: While mocks can help isolate units, overuse can lead to brittle tests that fail too often.

Common Challenges with TDD

- **Time Investment**: TDD initially requires more time, especially for writing tests, but pays off later.
- **Legacy Code**: Introducing TDD into a legacy codebase is challenging because existing code wasn’t designed with testability in mind.



You're Not Qualified to Have an Opinion on TDD

One of the marks of a good senior developer is that they have lots of interesting opinions.

https://blog.boot.dev/clean-code/youre-not-qualified-for-tech-opinions/

Course overview - Test-Driven Development

Free online course for learning Test-Driven Development

https://tdd.mooc.fi/

You won’t believe how old TDD is

Kent Beck is credited as the TDD inventor. Yet, he claims he just re-discovered it.

https://arialdomartini.wordpress.com/2012/07/20/you-wont-believe-how-old-tdd-is/