Test Driven Development By Example

Introduction: Embracing Test Driven Development Through Practical Examples

Software development is a discipline where precision and clarity can mean the difference between a smooth, maintainable project and a tangled web of bugs. Among the many methodologies that have emerged to tame complexity, Test Driven Development (TDD) stands out as a practice that promises higher quality code and fewer regressions. However, the concept of TDD can feel abstract or intimidating to newcomers. This is where learning by example becomes invaluable. **Test Driven Development By Example** is not just a catchphrase; it is a powerful approach that transforms theoretical rules into tangible, repeatable habits. By walking through concrete scenarios, developers can internalize the famous Red-Green-Refactor cycle and see firsthand how it shapes cleaner design and robust functionality.

In this article, we will explore the essence of TDD through multiple examples, starting with a simple arithmetic operation and moving to a more realistic business rule. We will break down each step, explain the reasoning behind the tests, and highlight the mindset shifts that occur when you let tests drive your code. Whether you are a seasoned developer looking to refine your practice or a novice eager to adopt a reliable workflow, understanding TDD by example will give you the confidence to apply it daily.

The Heart of TDD: Red, Green, Refactor

Before diving into examples, it is essential to grasp the three core states of TDD. The process begins with writing a test that defines a desired behavior – but the test should fail first (Red). Then you write the minimum amount of production code to make that test pass (Green). Finally, you clean up the code, removing duplication and improving design, while ensuring all tests remain green (Refactor). This cycle is repeated for every small increment of functionality. Learning TDD by example makes this abstract cycle concrete because you see each phase in action.

Why Examples Matter More Than Theory

Reading about TDD in books or articles gives you the vocabulary, but working through examples embeds the rhythm in your muscle memory. When you see a failing test and then a passing one, you understand the feedback loop. When you observe how a test can force you to write decoupled code, you grasp the design benefits. Therefore, the best way to learn is to start with a tiny, non-threatening example.

Example 1: Adding Two Numbers – The Simplest TDD Cycle

Let us begin with an absurdly simple requirement: a function that adds two integers. This example may feel trivial, but it perfectly illustrates the TDD flow without distractions.

Step 1 – Write a Failing Test (Red)

You start by imagining how the function should behave. You decide that `add(2, 3)` should return 5. You write a test that calls `add(2, 3)` and asserts the result equals 5. At this point, the function does not exist, so the test fails. The failure message is clear: `add` is undefined. This is your red state.

Step 2 – Make the Test Pass (Green)

Now you write the minimal code to eliminate the failure. You define a function `add(a, b) { return a + b; }`. You run the test again, and it passes. Congratulations – you are now in the green state. Note that you did not add error handling, type checks, or edge cases. The rule is: write only enough production code to satisfy the test.

Step 3 – Refactor

With the test passing, you look for opportunities to improve the code. In this trivial case, there might be nothing to refactor. However, you might notice the function name could be more descriptive, or you might want to rename parameters. The key is that any refactoring is done under the safety net of the passing test. If you break something, the test will turn red immediately.

This three-step cycle may seem overly simple, but it establishes the discipline. When you repeat it for many small behaviors, the result is a well-tested, cleanly designed system.

Example 2: A String Calculator – TDD in Action

A more illustrative example is the famous **String Calculator** kata. This kata introduces complexity step by step, making it perfect for learning TDD by example. The requirement: write a function that takes a string of comma-separated numbers and returns their sum.

Start with the simplest case: an empty string

- **Test (Red):** `Add("")` should return `0`. Write this test and watch it fail because the function does not exist.
- **Code (Green):** Create a function that returns `0` for any input. That is the minimal code to pass the test.
- **Refactor:** The implementation is trivial, but you can already think about the next test.

One number: "1" returns 1

- **Test:** `Add("1")` should equal 1. At this point the function returns `0` always, so the test fails.
- **Code:** Change the function to check if the input is empty; if not, parse the number and return it. The test passes.
- **Refactor:** You might extract a helper to parse numbers, but keep it simple.

Two numbers: "1,2" returns 3

- **Test:** `Add("1,2")` should return 3.
- **Code:** Modify the function to split the string by commas, convert each part to an integer, and sum them. The test passes.
- **Refactor:** Notice that splitting logic could be reused. You might also handle whitespace trimming.

Unknown amount of numbers: "1,2,3" returns 6

- The same splitting logic already works for any number of comma-separated values. The test simply confirms the existing code handles it.

Newline as delimiter: "1\n2,3" returns 6

- **Test:** Write a test using a newline character.
- **Code:** Update the split logic to accept both commas and newlines. The minimal change is to replace newlines with commas before splitting.
- **Refactor:** The delimiter handling is now more flexible; consider extracting a method that normalizes delimiters.

Support for custom delimiters: "//;\n1;2" returns 3

- **Test:** The input starts with "//" followed by a delimiter, then newline, then numbers. This is a new behavior.
- **Code:** Detect the custom delimiter prefix, extract it, and use that for splitting.
- **Refactor:** The code may now have a conditional for parsing the first line. Organize it into a clear function.

Notice how each test adds a small increment of functionality. The final implementation is robust and covers many edge cases, all driven by tests that were written before the code. This is the essence of **Test Driven Development By Example** – each example forces you to think about expected behavior, then implement only what is necessary.

Example 3: A Shopping Cart – Testing Business Rules

Let us move to a more realistic domain: an e-commerce shopping cart. The business rules can become complex, but TDD keeps them manageable.

Scenario: Add an item to an empty cart

- **Test:** Create a new cart, add an item with product ID and quantity. Assert that the cart's item count is 1 and the total price is correct.
- **Code:** Implement a Cart class with an `addItem` method that stores items in a list. Minimal code to pass.
- **Refactor:** Consider using a private list and exposing read-only properties.

Scenario: Add duplicate item increases quantity

- **Test:** Add the same product twice; the cart should have one line item with quantity 2, not two separate entries.
- **Code:** Modify `addItem` to check if the product already exists in the cart and update quantity instead of adding a new entry.
- **Refactor:** Extract a method to find an existing item by product ID.

Scenario: Calculate total with discounts

- **Test:** Cart applies a 10% discount when total exceeds $100. Add items worth $120, then assert total is $108.
- **Code:** After summing the line item prices, apply a conditional discount. The test drives you to implement the discount logic.
- **Refactor:** Move discount rules to a separate policy class to keep the cart clean.

These examples demonstrate how TDD naturally leads to modular, maintainable code. Each test acts as a specification and also as a safety net for future changes.

The Benefits of Learning TDD Through Examples

When you learn TDD by example, you receive immediate feedback. You see how a simple test can guide the design before it becomes tangled. The following benefits become evident:

- **Improved Code Quality:** You write only what is needed, avoiding speculative features.
- **Faster Debugging:** When a test fails, you know exactly which requirement was broken.
- **Better Design:** Writing testable code forces you to dependency injection and separation of concerns.
- **Documentation:** Tests serve as living documentation that always reflects the actual behavior.
- **Confidence to Refactor:** With a comprehensive test suite, you can improve code without fear.

Downloaded from ges.cursor.cl by Reynolds & Black • Explore the unseen. •

Common Pitfalls and How Examples Help Avoid Them

Newcomers to TDD often fall into traps like writing too many tests at once, testing internal implementation instead of behavior, or forgetting the refactor step. Working through examples explicitly demonstrates the correct rhythm:

- Start with one trivial test.
- Never write production code without a failing test.
- Keep tests focused on one aspect.
- Celebrate the refactor phase – it’s not optional.

Seeing these pitfalls in the context of an example makes the lessons stick. For instance, the String Calculator kata shows what happens if you try to test all delimiters at once – you get a large failing test that is hard to fix. Better to iterate.

Conclusion: Make TDD a Habit through Examples

Test Driven Development is a discipline that transforms the way you think about code. But like any skill, it must be practiced. The most effective way to internalize the Red-Green-Refactor cycle is to work through examples that incrementally introduce complexity. Whether you choose the String Calculator, a shopping cart, or