What is a unit test?



www.yzokul.com.tr

What is a unit test?



A unit test is a test that checks a small part of a program with specific inputs. It confirms that the part behaves as expected. For example, a unit test can check that a function that calculates a discount gives the correct result for a given rate. This book explains what a unit test is, step by step, for complete beginners.



A program is made of many small parts, like functions and classes. A function is a piece of code that does one job; a class is a blueprint that groups data and functions. Each part does one job inside the whole program. For example, one function might calculate a total, another might format a date. Understanding these parts helps you see what a unit test checks.



To test a small part, you first pick out just one part of the program to look at. This narrow focus is the starting point of a unit test. For example, choose only the function that calculates a discount and ignore the rest of the program for now. By zooming in on one part, you can check its behavior without interference from other parts.



A function works like a small machine: it takes inputs and gives back a result. A unit test checks that this input-to-result behavior is correct. For example, a discount function takes a price and a discount rate as inputs and returns the discounted price. The test confirms that for given inputs, the function returns the expected result.



Every unit test needs specific inputs and the result you expect for them. The test passes when the real result matches the expected one. For example, if you input a price of 100 and a discount rate of 10%, you expect the result 90. You write the test to compare the function's actual output with 90.



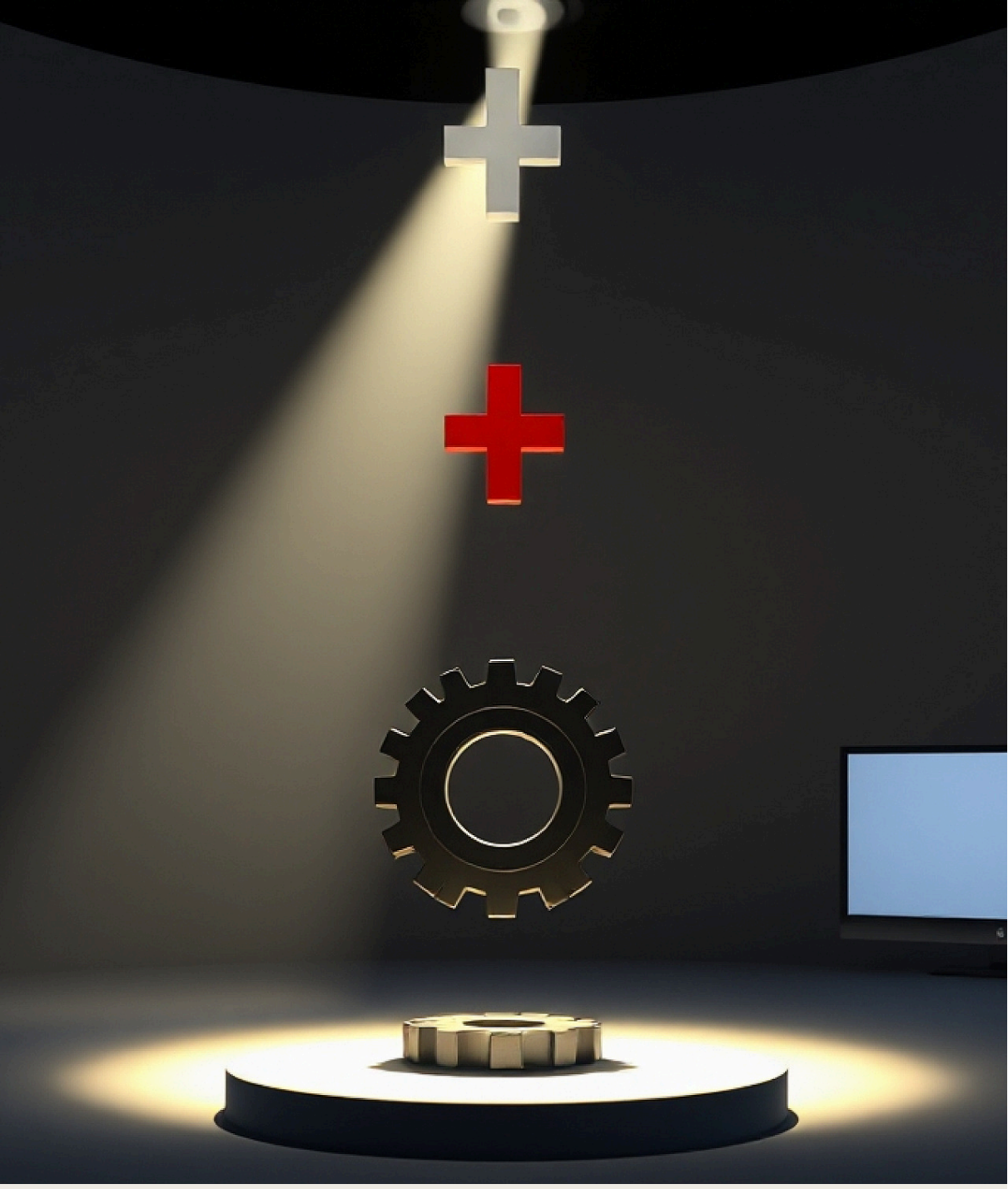
Consider a function that calculates a discount. It takes a price and a discount rate, then returns the price after the discount is applied. This small function becomes the unit under test for the rest of this book. All examples will use this function to show how unit tests work in practice.



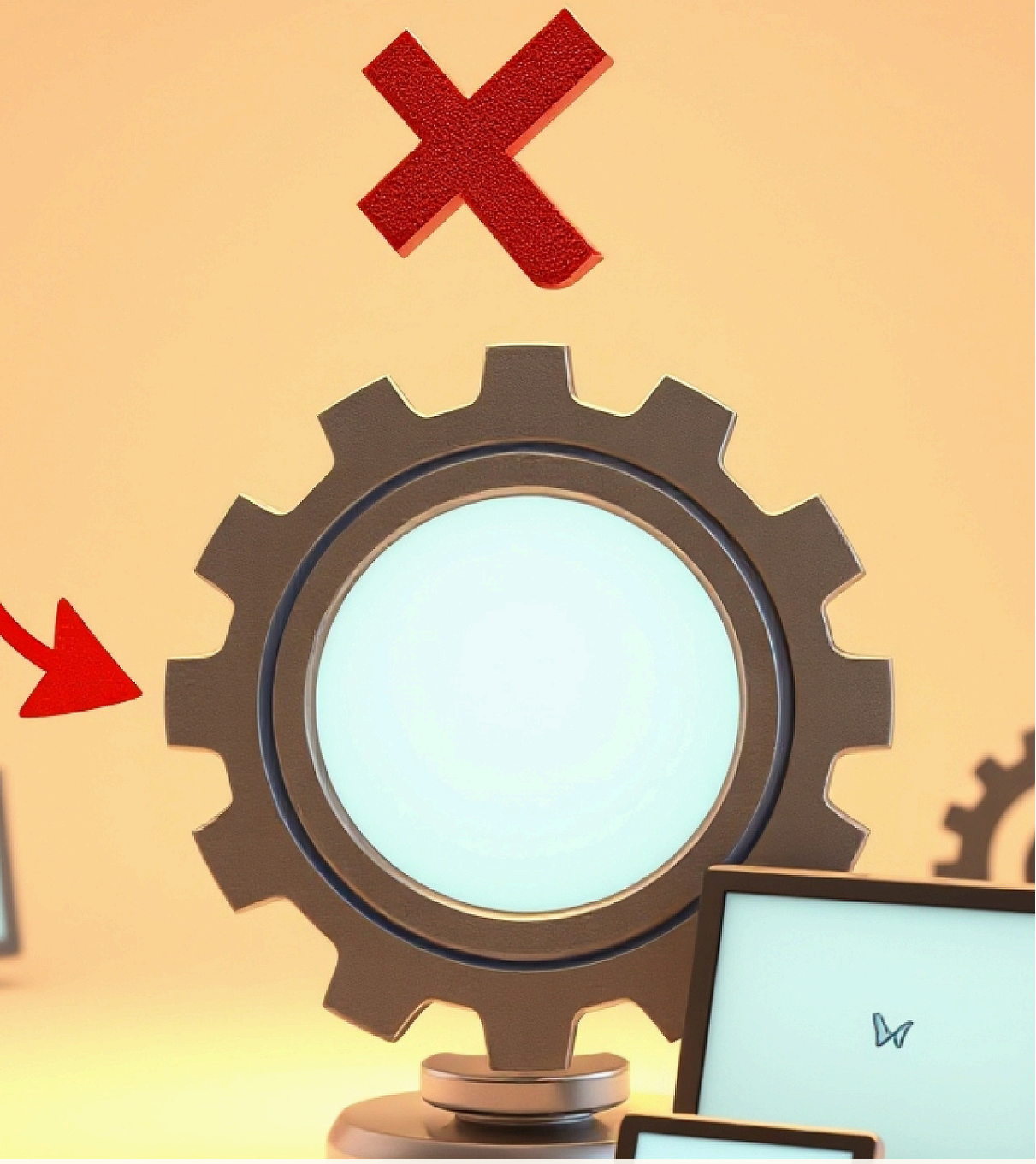
Test the discount function with several different rates as inputs. Each rate has its own expected result to compare against. For example, a 10% discount on 100 gives 90, a 20% discount on 100 gives 80, and a 50% discount on 100 gives 50. Running these tests checks the function's behavior across different cases.



Each test checks one specific input and its expected result. Several small tests together cover the function's behavior. For example, one test checks the 10% rate, another checks the 20% rate, and another checks the 50% rate. Writing separate tests for each case makes it clear which input caused a problem if one fails.



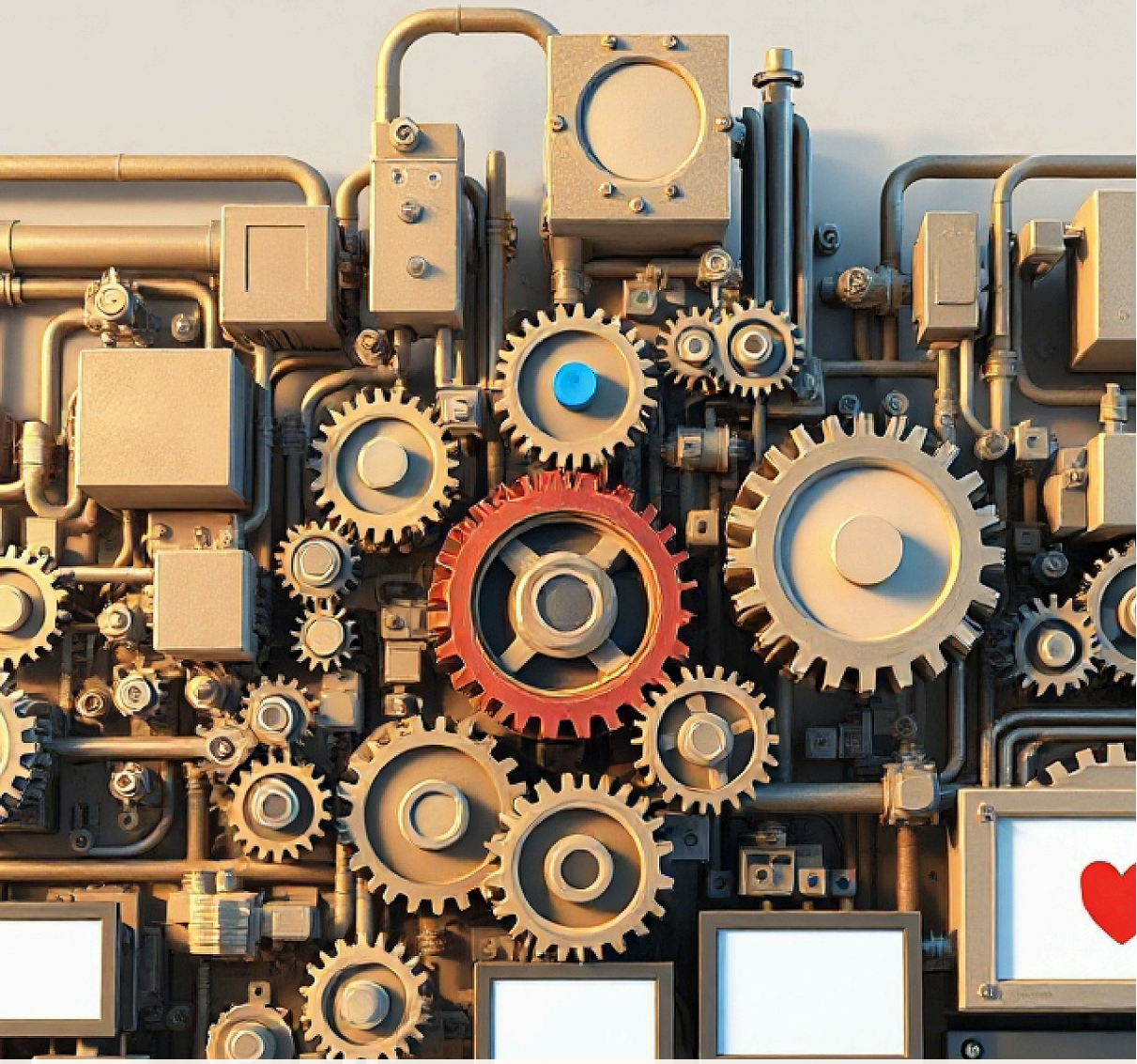
Keeping the test narrow matters because when only one small part is tested, it is easier to see what went wrong. If the discount function returns 85 instead of 90 for a 10% discount, the narrow test points directly to that function. You do not have to search through the whole program to find the mistake.



A failing narrow test points close to the error. For example, if the test for a 20% discount fails, you know the problem is in the discount function, not in other parts like formatting or data loading. This makes finding the source of the problem much easier and faster.



A failing test is useful information, not a disaster. It shows that the part's behavior does not match what you expected. For example, if the discount function returns 85 instead of 90, the failing test tells you exactly which input and which result are wrong. You can then fix the function and run the test again.



Checking whether large parts of a program work together is a different kind of testing. That job belongs to other test types, such as integration tests, which check how multiple parts interact. Unit tests stay focused on one small part at a time. For example, testing the whole checkout process belongs to integration testing, not unit testing.



Compare the narrow unit test with tests of larger parts. A unit test checks one small part, like a single function, while other tests check many parts working together. The key difference is the scope: small part versus many parts. For example, a unit test checks the discount function alone; an integration test checks the discount function plus the payment and receipt functions together.



A common beginner mistake is testing too much at once. A test that covers many parts is no longer a unit test. For example, testing the discount function together with the payment and receipt functions in one test mixes scopes. Keep each unit test focused on one small part to get clear, useful results.



Another mistake is testing without specific inputs and expected results. Without them, there is nothing to compare, so the test proves nothing. For example, calling the discount function without checking the returned value does not confirm correct behavior. Always state the input and the expected result in each test.



A unit test checks a small part of a program with specific inputs. Narrow scope makes errors easier to find because a failing test points close to the problem. Testing large parts together is left to other test types. Remember to use specific inputs and expected results in every unit test you write.