# TEST-DRIVEN DEVELOPMENT

*Outcome*

Test-Driven Development (TDD) is a software development technique. Developers write a unit test before writing or changing just enough code to enable the test to pass. This leads to a very short feedback cycles.

*Function*

Test-Driven Development benefits include:

- Defects are caught as early as possible in the development cycle when they are cheapest to fix.
- Simple design is encouraged, since code has a crystal clear goal – passing the defined tests.
- Quality regression test suites are evolved that continuously assure the integrity of the product.
- Supports predictable and sustainable progress through step-by-step incremental extension of a product.

*Benefit*

Test-Driven Development in its original form and application is focused at the lowest level of code change and integration, i.e. unit tests and code changes being written at an individual workstation on a minute-by-minute basis. Its use has now been extended to offer benefits at further levels:

- Specifying a major component interface. Writing the tests that the component needs to pass against that interface before implementing the component helps us control the development and integration of complex modular systems – even when the modules are being developed separately.
- System-level tests. Writing tests for a product requirement before we start to design and implement the requirement enables us to clarify what behaviour is required without writing detailed requirements documentation.

*Who*

Test-Driven Development was first popularised as part of the Extreme Programming method (XP rule: 'Code the Unit Test First' and XP practice: 'Test-First Programming') but it is also a stand-alone practice, which does not require the adoption of any other XP practices.

## Implementation

**Prerequisites**

- An environment in which we can write and run unit tests and code.
- All unit tests for the code being changed or extended must pass before further code is written.

*Scaling Factors*

**Doing Test-Driven Development**

- Write a new test that will test the operation of the new code behaviour that is required.
- Run all the tests – only the new test should fail.
- Write the software required to make the new test pass.
- Re-run all the tests to make sure they all pass.

*Difficulty*

- Refactor the code as required to remove duplicates or inefficiencies and ensure that it is maintainable.
- Re-run the tests to make sure that they all still pass.

## Potential pitfalls

- Test-Driven Development is a highly disciplined approach. Some developers find it laborious and time consuming to begin with. This means it is more often talked about than consistently applied.
- If the business goal does not demand quality, longevity or maintainability of the code, then Test-Driven Development may not be the best use of a developer's time in all circumstances. Be warned though – it is easy to believe code will be thrown away – only for it to turn into a legacy system with nightmare maintenance problems that could have been avoided by using Test-Driven Development up front.
- Developers must find the right balance between algorithmic programming and 'programming by example'. Sometimes it is easier to think through the requirement and the solution 'in the general case first' rather than on a blinkered individual unit-test-by-unit test basis. In many examples, this is common sense – imagine a function to multiply 2 numbers together. It would be absurd to write a test to ensure that when 2 and 2 are passed as inputs, the return result is 4; then write code to get this test to pass; then try with a test to see if passing in 3 and 3 returns 9, and so on. Instead, it makes sense to code the function and then think up and write a reasonable set of test cases to check it is working.

If you want to learn more, consider reading:
*Test-Driven Development by Example* by Kent Beck
*Extreme Programming Explained* by Kent Beck and Cynthia Andres
*Growing Object-Oriented Software, Guided by Tests* by Steve Freeman and Nat Pryce