TEST-DRIVEN DEVELOPMENT

Test-Driven Development (TDD)

- Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle:
  - first the developer writes an (initially failing) automated test case that defines a desired improvement or new function,
  - then produces the minimum amount of code to pass that test,
  - and finally refactors the new code to acceptable standards.
- Kent Beck, who is credited with having developed or 'rediscovered' the technique, stated in 2003 that TDD encourages simple designs and inspires confidence.
- Test-driven development is related to the test-first programming concepts of extreme programming, begun in 1999.

TDD Cycle

1. Add a test
2. Run all tests to see if new one fails
3. Write some code
4. Run test
5. Refactor code
6. Repeat

1. Add a test

- In TDD, each new feature begins with writing a test. This test must inevitably fail because it is written before the feature has been implemented. To write a test, the developer must clearly understand the feature's specification and requirements.
- The developer can accomplish this through use cases and user stories to cover the requirements and exception conditions, and can write the test in whatever testing framework is appropriate to the software environment. This could also be a modification of an existing test.
- This is a differentiating feature of test-driven development versus writing unit tests after the code is written: it makes the developer focus on the requirements before writing the code, a subtle but important difference.

2. Run all tests to see if new one fails

- This validates that the test harness is working correctly and that the new test does not mistakenly pass without requiring any new code.
- This step also tests the test itself, in the negative: it rules out the possibility that the new test always passes, and therefore is worthless. The new test should also fail for the expected reason. This increases confidence (though does not guarantee) that it is testing the right thing, and passes only in intended cases.

3. Write some code

- The next step is to write some code that causes the test to pass. The new code written at this stage is not perfect, and may, for example, pass the test in an inelegant way. That is acceptable because later steps improve and hone it.
- At this point, the only purpose of the written code is to pass the test; no further (and therefore untested) functionality should be predicted and 'allowed for' at any stage.

4. Run tests

- If all test cases now pass, the programmer can be confident that the code meets all the tested requirements. This is a good point from which to begin the final step of the cycle.

5. Refactor code

- Now the code should be cleaned up as necessary. Remove any duplication you can find. Make sure that variable and method names represent their current use. Clarify any constructs that might be misinterpreted. Use Kent Beck's four rules of simple design to guide you, as well as anything else you know about writing clean code.
- By re-running the test cases, the developer can be confident that code refactoring is not damaging any existing functionality.
- The concept of removing duplication is an important aspect of any software design.

6. Repeat

- Starting with another new test, the cycle is then repeated to push forward the functionality.
- The size of the steps should always be small. If new code does not rapidly satisfy a new test, or other tests fail unexpectedly, the programmer should undo or revert in preference to excessive debugging.

Development style

- There are various aspects to using test-driven development, for example the principles of "keep it simple stupid" (KISS) and "You aren't gonna need it" (YAGNI). By focusing on writing only the code necessary to pass tests, designs can often be cleaner and clearer than is achieved by other methods.

- To achieve some advanced design concept, such as a design pattern, tests are written that generate that design. The code may remain simpler than the target pattern, but still pass all required tests. This can be unsettling at first but it allows the developer to focus only on what is important.

Development style

- Write the tests first. The tests should be written before the functionality that is being tested. This has been claimed to have many benefits. It helps ensure that the application is written for testability, as the developers must consider how to test the application from the outset, rather than worrying about it later. It also ensures that tests for every feature get written.
- Additionally, writing the tests first drives a deeper and earlier understanding of the product requirements, ensures the effectiveness of the test code, and maintains a continual focus on the quality of the product. When writing feature-first code, there is a tendency by developers and the development organisations to push the developer on to the next feature, neglecting testing entirely. The first test might not even compile, at first, because all of the classes and methods it requires may not yet exist. Nevertheless, that first test functions as an **executable specification**.

Benefits

- A 2005 study found that using TDD meant writing more tests and, in turn, programmers who wrote more tests tended to be more productive. Hypotheses relating to code quality and a more direct correlation between TDD and productivity were inconclusive.
- Programmers using pure TDD on new projects reported they rarely felt the need to invoke a debugger. Used in conjunction with a version control system, when tests fail unexpectedly, reverting the code to the last version that passed all tests may often be more productive than debugging.
- Test-driven development offers more than just simple validation of correctness, but can also drive the design of a program. By focusing on the test cases first, one must imagine how the functionality is used by clients. So, the programmer is concerned with the interface before the implementation. This benefit is complementary to Design by Contract as it approaches code through test cases.

Benefits

- Test-driven development offers the ability to take small steps when required. It allows a programmer to focus on the task at hand as the first goal is to make the test pass. Exceptional cases and error handling are not considered initially, and tests to create these extraneous circumstances are implemented separately. Test-driven development ensures in this way that all written code is covered by at least one test. This gives the programming team, and subsequent users, a greater level of confidence in the code.
- While it is true that more code is required with TDD than without TDD because of the unit test code, the total code implementation time could be shorter. Large numbers of tests help to limit the number of defects in the code. The early and frequent nature of the testing helps to catch defects early in the development cycle, preventing them from becoming endemic and expensive problems. Eliminating defects early in the process usually avoids lengthy and tedious debugging later in the project.

Benefits

- TDD can lead to more modularized, flexible, and extensible code. This effect often comes about because the methodology requires that the developers think of the software in terms of small units that can be written and tested independently and integrated together later. This leads to smaller, more cohesive classes, looser coupling, and cleaner interfaces.
- Because no more code is written than necessary to pass a failing test case, automated tests tend to cover every code path. For example, for a TDD developer to add an else branch to an existing if statement, the developer would first have to write a failing test case that motivates the branch. As a result, the automated tests resulting from TDD tend to be very thorough: they detect any unexpected changes in the code's behaviour. This detects problems that can arise where a change later in the development cycle unexpectedly alters other functionality.

Shortcomings

- Test-driven development reliance on unit tests does not perform sufficient testing in situations where full functional tests are required to determine success or failure.
- Examples of these are user interfaces, programs that work with databases, and some that depend on specific network configurations. TDD encourages developers to put the minimum amount of code into such modules and to maximize the logic that is in testable library code, using fakes and mocks to represent the outside world.
- Management support is essential. Without the entire organization believing that test-driven development is going to improve the product, management may feel that time spent writing tests is wasted.

Shortcomings

- Unit tests created in a test-driven development environment are typically created by the developer who is writing the code being tested. The tests may therefore share the same blind spots with the code:
  - If, for example, a developer does not realize that certain input parameters must be checked, most likely neither the test nor the code will verify these input parameters.
  - If the developer misinterprets the requirements specification for the module being developed, both the tests and the code will be wrong, as giving a false sense of correctness.
- The high number of passing unit tests may bring a false sense of security, resulting in fewer additional software testing activities, such as integration testing and compliance testing.

Shortcomings

- Tests become part of the maintenance overhead of a project. Badly written tests are themselves prone to failure, are expensive to maintain. There is a risk that tests that regularly generate false failures will be ignored, so that when a real failure occurs, it may not be detected. It is possible to write tests for low and easy maintenance, and this should be a goal during the code refactoring phase described above.

- Overtesting can consume time both to write the excessive tests, and later, to rewrite the tests when requirements change. Also, more-flexible modules (with limited tests) might accept new requirements without the need for changing the tests. For those reasons, testing for only extreme conditions, or a small sample of data, can be easier to adjust than a set of highly detailed tests.

Shortcomings

- The level of coverage and testing detail achieved during repeated TDD cycles cannot easily be re-created at a later date. Therefore these original, or early, tests become increasingly precious as time goes by. The tactic is to fix it early. Also, if a poor architecture, a poor design, or a poor testing strategy leads to a late change that makes dozens of existing tests fail, then it is important that they are individually fixed. Merely deleting, disabling or rashly altering them can lead to undetectable holes in the test coverage.

Designing for testability

- Complex systems require an architecture that meets a range of requirements. A key subset of these requirements includes support for the complete and effective testing of the system. Effective modular design yields components that share traits essential for effective TDD.
  - High Cohesion ensures each unit provides a set of related capabilities and makes the tests of those capabilities easier to maintain.
  - Low Coupling allows each unit to be effectively tested in isolation.
  - Published Interfaces restrict Component access and serve as contact points for tests, facilitating test creation and ensuring the highest fidelity between test and production unit configuration.

Designing for testability

- A key technique for building effective modular architecture is Scenario Modeling where a set of sequence charts is constructed, each one focusing on a single system-level execution scenario.
- The Scenario Model provides an excellent vehicle for creating the strategy of interactions between components in response to a specific stimulus.
- Each of these Scenario Models serves as a rich set of requirements for the services or functions that a component must provide, and it also dictates the order that these components and services interact together. Scenario modeling can greatly facilitate the construction of TDD tests for a complex system

Managing tests for large systems

- In a larger system the impact of poor component quality is magnified by the complexity of interactions.
- This magnification makes the benefits of TDD accrue even faster in the context of larger projects. However, the complexity of the total population of tests can become a problem in itself, eroding potential gains. It sounds simple, but a key initial step is to recognise that test code is also important software and should be produced and maintained with the same rigor as the production code.
- Creating and managing the architecture of test software within a complex system is just as important as the core product architecture.