Continuous Testing

Productivity increase through Continuous Testing.
By Bart de Best

Context:
This blog is derived from my experience as a DevOps trainer, coach, and auditor. Each application of Continuous Testing has provided more insights into this powerful concept. This blog describes both the success stories and the limitations.

Challenge:
The challenge of applying Continuous Testing is that the DevOps engineer has to switch his mind to first write a test case and then the source code. Failure to apply Continuous Testing can lead to defects being identified late in the CI/CD secure pipeline, a low testing coverage rate, sacrifice of test time to programming time and reduced performance due to a long search for the cause of the defects. The reward for applying Continuous Testing can be up to 300% performance improvement for the DevOps engineers.

Solution:
The solution to this challenge has been found in the concept of Continuous Testing in which Test Driven Development is anchored. This blog discusses the TDD approach to Continuous Testing based on the following steps:

1. Definition of Test-Driven Development (TDD)
2. Definition of Continuous Testing value stream
3. The method
4. The experiences

1. Definition of Test-Driven Development (TDD)

TDD is the core of Continuous Testing, the value stream that gives substance to test management within DevOps. TDD is an approach aimed at integrating testing and programming. This is based on the following principles:

- Shift left
- Test case first
- Incremental Iterative
- Unit test case
- Code driven testing
- Test automation
- Insulation

Shift left
TDD's method is aimed at executing 80% of the test cases in the development environment in order to find 80% of the defects in the development environment (D).

As a result, only 20% of the test effort remains in the test environment (T) and the acceptance environment (A). The testing effort therefore shifts from right to left in the D-T-A-P street.

Test case first

The 'test case first' principle refers to the fact that a test case is first written and then a small part of the source code that is just enough to successfully execute the test case. As a result, the source code must be thought about before it is written.

Incremental Iterative

Instead of writing a unit (function, etc.) in one go and then testing it in one go, a unit is built step by step, with each step starting with an additional test case. The incremental and iterative nature of Agile Scrum is therefore extended to the writing of source code. A new increment of the unit requires that all test cases are successfully closed.

Unit test cases

TDD focuses on the smallest unit of programming, which is often a function written in Python or Java, for example. This also makes the integration of testing and programming possible.

Code driven testing

Unit test cases are programmed in the language of the source code. This means that the test cases can also be included in GIT and provided with version management. GIT also allows you to establish the relationship between the unit test cases and the source code items. This means that checking out the source code in most editors also checks out the test case so that both can be modified at the same time.

Test automation

The output of the unit test cases is mandatory with each increment by immediately kicking off the unit test cases after the build. The test run of unit test cases should not take more than 5 minutes.

Isolation

The unit test case must be tested in isolation.

This means that it may not have any interfaces with databases, other applications, network traffic, message queues, e-mail services, etc. The consequence is that sometimes mocking or faking is necessary to put the unit under test through its paces.

2. *Definition of the Continuous Testing value stream*

An example of a Continuous Testing value stream is shown in Figure 1. TDD starts in step 3 in which the unit test case (UT) is created.

That test case is executed in step 4 without any source code being written and therefore fails in the first test run of the increment in question. The source code is then written in step 5 and steps 4 and 5 are carried out until the test case has been completed successfully. The source code is then cleaned up and a new increment of the unit is started by writing the new unit test case in step 3.

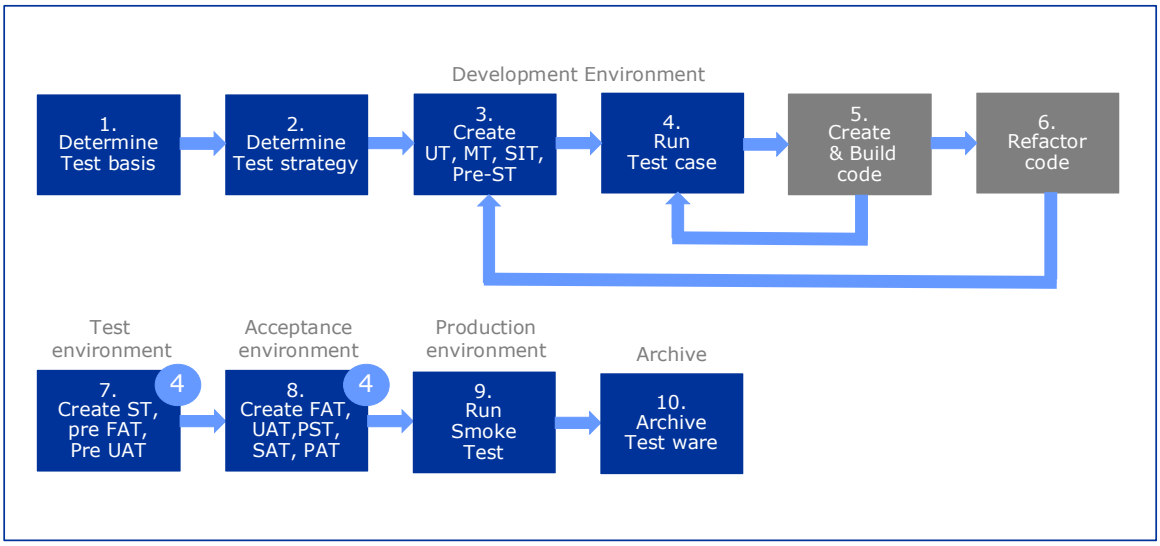


Figure 1. Continuous Testing value stream.

The abbreviations in Figure 1 are:

- MT = Module Test
- SIT = System Integration Test
- ST = System Test
- FAT = Functional Acceptance Test
- UAT = User Acceptance Test
- SAT = Security Acceptance Test
- PAT = Production Acceptance Test
- PST = Performance Stress Test

2. The Way of Working

Figure 2 provides an overview of the TDD approach. On the left, the meta data of the function is first written. The user story and requirement (BDD) were then named. These are the basis for the source code.

Before starting to work on the source code, a list is first made of the problems that need to be solved. The number of problems depends on the experience of the DevOps engineer. An experienced DevOps engineer only mentions the considerations for solutions and the choice made for the implementation. The steps describing the function are then listed, followed by the pseudocode that describes how the function should work.

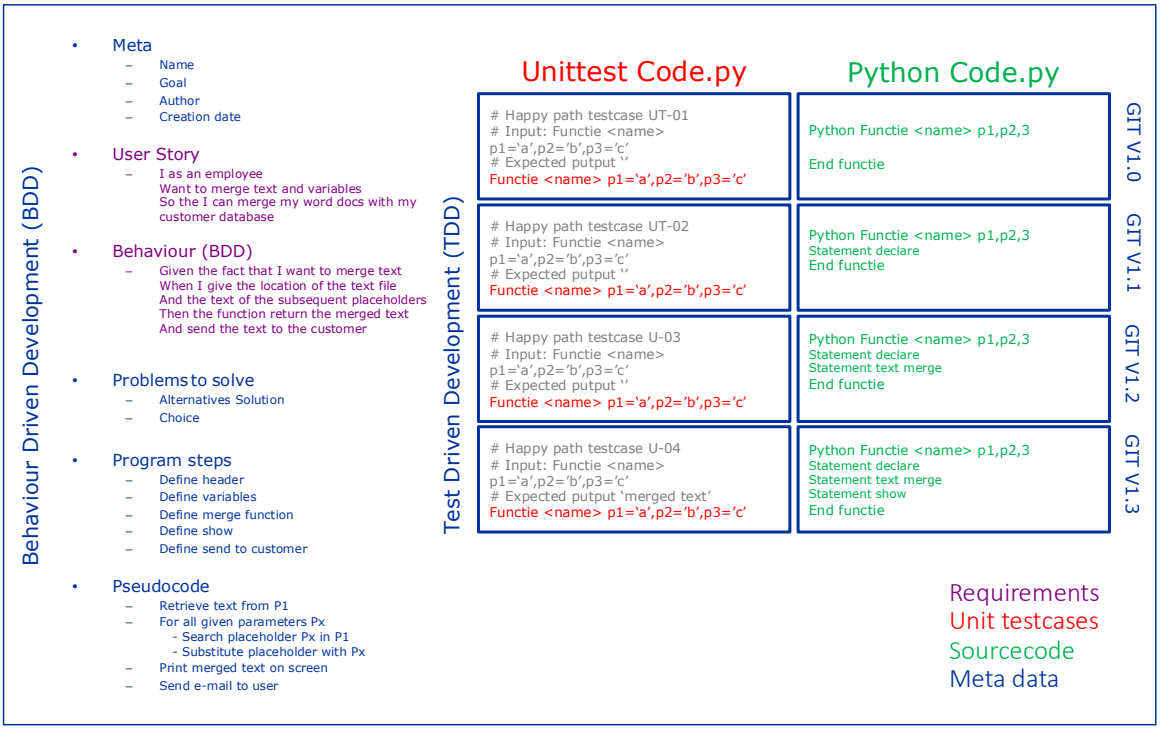


Figure 2, Example application of TDD.

In general, this definition takes about 15 minutes. However, it can save many times more time during source code modification and bug fixing. On the right you can see the unit test case in red and the source code in green. The syntax of the test case and source code is of course incorrect and is only written this way as an indication of the working method.

3. *The experiences*

Over the years I have had various experiences with TDD in the context of Continuous Testing that I would like to share with you.

Own experiences

In six months I learned to program the Python language based on the book 'Think Python' during a voluntary training outside working hours at a customer. The exercises were quite tough and programming the solution directly often led to not being able to complete the assignment in one evening. Until I applied TDD and wrote the assignment step by step based on first writing the test case and then the source code. In most cases this saved me a factor of 3 in time. The mistake I made in the beginning was trying to write all the unit test cases and then write the source code. Not only was it very difficult to define all the unit test cases in advance, but it also did not provide the feedback during programming that I needed to find a bug. Still, I was very stubborn and tried many times to complete the assignment without TDD. After a few weeks I finally left the old way of programming behind me and was able to go through my learning curve much faster.

Training experiences

The advantage of TDD is sometimes discussed in training. These are always very nice discussions and provide a clearer picture of the applicability of TDD. Difficult applications are:

1. User interface development.
   It is difficult to write a test case because the user interface is often not a separate function but directly leads to an End-2-End test. TDD can be used, but it is important to check whether the interface between the front-end and the back end of the application is used. If the front-end calls a function from the back end, the unit test case can be written against it.

2. Software that relies heavily on infrastructure services.
   The software must be tested in isolation. In this case, mocking or faking must be used. Developing this may include a significant amount of time spent. A business case must therefore be considered.

3. Legacy software.
   This is software whose code is often a monolith. This means that no individual functions can be identified and that only system testing, and the like are possible. Refactoring of the application should then be considered, such as converting it to microservices. This can be a costly exercise that is mainly done for applications that support the primary business value streams. It is true that AI can greatly accelerate this.
4. Data intensive applications.
   Applying TDD to applications whose logic is highly dependent on the data, such as data analysis and machine learning, is difficult. In that case, a number of TDD principles may have to be compromised.

Coach experiences

The question in consultancy is often whether TDD can be applied optionally, i.e. if the DevOps engineer considers this necessary. This is a cunning statement because the risk of dilution of TDD is then constantly lurking.
That is why I recommend always doing TDD, unless, for example, the mocking is too expensive, in which case TDD can still be done, but the TDD principles must be handled flexibly.

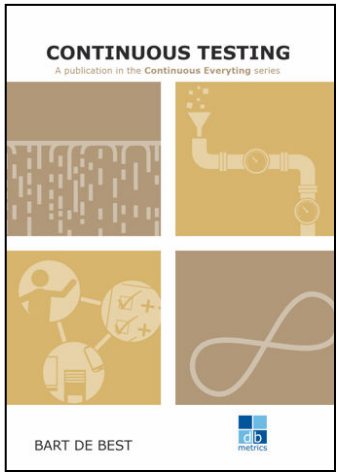
Another experience is that the test management terms are not used in accordance with common sense. For example, Selenium testing of the user interface is still defined as unit test cases. This is not wise because it involves going through the entire application. This blurring of terminology is treacherous in communication.

Audit experiences

I have had the opportunity to audit various organisations on Continuous Testing. A discussion sometimes arises as to why TDD should be seen at level 2 of maturity on the scale of 5 (CMM). Why isn't this level 3? The reason for this is that level 2 of the CMM model indicates that the flow has been adjusted, i.e. the steps of the value streams. TDD is the anchor here for Continuous Testing because it gives substance to the shift left organisation. Omitting level 2 causes a completely different flow without fast feedback.

With TDD the quality of software development can be continuously monitored. The unit test cases can also be used in regression testing. That is why TDD is a good example of implementing Continuous Testing.

By Bart de Best
DutchNordic.Group



https://www.dbmetrics.nl/ce-en/continuous-testing-en/