Some developers who are passionate about testing are sometimes also fanatical advocates of Test Driven Development (TDD), to the point that they believe TDD is **the only correct way** to test applications. Strict adherence to TDD requires a test to be created first before any code has been written. As such proponents of TDD tend to see TDD as a design mechanism rather than as a test mechanism. Irrespective of whether TDD is used as a design and/or test mechanism, it has a number of pitfalls.

In this post I will:

1. give a brief overview of the TDD cycle,
2. explain why TDD as software design mechanism is inadequate,
3. explain why TDD as software test mechanism is inadequate,
4. explain the conditions under which TDD can be applied successfully to projects.

1. TDD

In TDD the steps to add new code are as follows [1]:

1. Add a (failing) test that serves as a specification for the new functionality to be added.
2. Run all the tests and confirm that the newly added test fails.
3. Implement/correct the new functionality.
4. Run all tests to confirm that they all succeed.
5. Refactor the code to remove any duplication.

The aim of TDD is to provide a simple way to grow the design of complex software one decision at a time.

2. TDD as a Design Mechanism

From a design perspective TDD emphasizes m**i**rco level design rather than m**a**cro level design [2]. The weaknesses of this approach are:

- TDD forces a developer to necessarily focus on a single interface. This often neglects the interaction between interfaces and leads to poor abstraction. Bertrand Meyer gives a balanced review of the challenges regarding TDD and Agile in this regard [3].
- Quality attributes (like performance, scalability, integrability, security, etc.) are easily overlooked with TDD. In the context of Agile, where an upfront architecture effort is typically frowned upon, TDD is particularly dangerous due to poor consideration of quality attributes.

3. TDD as a Testing Mechanism

From a testing perspective TDD has the following drawbacks:

- Interfaces/classes may be polluted in order to make them testable. A typical example is that a private method is made public in order to test it. This obfuscates the intended use of the class which will cause developers to more easily digress from the intended use of the class. In the long term this creates an unmaintainable system.
- Often when TDD is used on projects unit testing is used to exclusion, with limited or no regard for the broad spectrum of testing, which should at least include integration testing and systems testing.
- TDD has no appreciation for the prioritization of the testing effort: Equal amounts of effort are expended to test all code irrespective of the associated risk profile. Typically TDD expects all further development to be blocked until all tests pass. This ignores the reality that some functionality has greater business value than others.

4. Guidelines for using TDD Effectively on Projects

TDD can be used with no side effect if the following process is adhered to:

1. The architecture has to be complete which should include details as to how testability of the system at all levels (unit-, integration- and systems testing) will be achieved.
2. The risk profile of the sub system (module/use case) has been established, which informs the testing effort that must be expended on the testing of the sub system.
3. The design for the sub system (module/use case) should be complete in adherence to the architecture and risk profile. Since testability is considered as part of the architecture, and the design is informed by the architecture, the design should be by definition testable.
4. Only at this point is the developer is now free to follow a TDD approach in implementing the code.

5. Conclusion

The very positive thing that TDD emphasizes is the need for testing. However, to naively embrace TDD, is to do it at the peril of your project.

References

1. Kent Beck, *Test-Driven Development by Example*, The Addison-Wesley Signature Series, Addison-Wesley, 2003.
2. Cédric Beust and Hani Suleiman, *Next Generation Java Testing : TestNG and Advanced Concepts*, Addison-Wesley, Upper Saddle River, NJ, 2008.
3. Bertrand Meyer, *Agile!: The Good, the Hype and the Ugly*, Springer, 2014.