Extracted from article at https://codurance.com/2015/05/12/does-tdd-lead-to-good-design/

Not all TDDs are the same

There are two main styles of TDD with significant differences between them, mainly when it comes to design.

Classicist

The Classicist approach is the original approach to TDD created by Kent Beck. It’s also known as Detroit School of TDD.

Main characteristics

* Design happens during the refactoring phase.
* Normally tests are state-based tests.
* During the refactoring phase, the unit under test may grow to multiple classes.
* Mocks are rarely used, unless when isolating external systems.
* No up-front design considerations are made. Design completely emerges from code.
* It’s a great way to avoid over-engineering.
* Easier to understand and adopt due to state-based tests and no design up-front.
* Often used in conjunction with simple design upfront and BDD.
* Good for exploration, when we know what the input and desired output are but we don’t really know how the implementation looks like.
* Great for cases where we can’t rely on a domain expert or domain language (data transformation, algorithms, etc.)

Problems

* Exposing state for tests purpose only.
* Refactoring phase is normally bigger when compared to Outside-In approach (more on that below).
* Unit under test becomes bigger than a class when classes emerge during the refactoring phase. This is fine when we look at that test in isolation but as classes emerge, they create life of their own, being reused by other parts of the application. As these other classes evolve, they may break completely unrelated tests, since the tests use their real implementation instead of a mock.
* Refactoring (design improvement) step is often skipped by inexperienced practitioners, leading to a cycle that looks more like RED-GREEN-RED-GREEN-…-RED-GREEN-MASSIVE REFACTORING.
* Due to its exploratory nature, some classes under test are created according to the “I think I’ll need this class with this interface (public methods)”, making them not fit well when connected to the rest of the system.
* Can be slow and wasteful since quite often we already know that we cannot have so many responsibilities in the class under test. The classicist advice is to wait for the refactoring phase to fix the design, only relying on concrete evidence to extract other classes. Although this is good for novices, this is pure waste for more experienced developers.

Outside-In

Outside-In TDD, also known as London School or mockist, is a TDD style developed and adopted by some of the first XP practitioners in London. It later inspired the creation of BDD.

Main characteristics

* Different from the classicist, Outside-In TDD prescribes a direction in which we start test-driving our code: from outside (first class to receive an external request) to the inside (classes that will contain single pieces of behaviour that satisfy the feature being implemented).
* We normally start with an acceptance test which verifies if the feature as a whole works. The acceptance test also serves as a guide for the implementation.
* With a failing acceptance test informing why the feature is not yet complete (no data returned, no message sent to a queue, no data stored in a database, etc.), we start writing unit tests. The first class to be tested is the class handling an external request (a controller, queue listener, event handler, the entry point for a component, etc.)
* As we already know that we won’t build the entire application in a single class, we make some assumptions of which type of collaborators the class under test will need. We then write tests that verify the collaboration between the class under test and its collaborators.
* Collaborators are identified according to all the things the class under test needs to do when its public method is invoked. Collaborators names and methods should come from the domain language (nouns and verbs).
* Once a class is tested, we pick the first collaborator (which was created with no implementation) and test-drive its behaviour, following the same approach we used for the previous class. This is why we call outside-in: we start from classes that are closer to the input of the system (outside) and move towards the inside of our application as more collaborators are identified.
* Design starts in the red phase, while writing the tests.
* Tests are about collaboration and behaviour, not state.
* Design is refined during the refactoring phase.
* Each collaborator and its public methods are always created to serve an existing client class, making the code read very well.
* Refactoring phases are much smaller, when compared to the classicist approach.
* Promotes a better encapsulation since no state is exposed for test purposes only,
* More aligned to the tell, don’t ask approach.
* More aligned to the original ideas of Object Oriented Programming: tests are about objects sending messages to other objects instead of checking their state.
* Suitable for business applications, where names and verbs can be extracted from user stories and acceptance criteria.

Problems

* Much harder for novices to adopt since a higher level of design skill is necessary.
* Developers don’t get feedback from code in order to create collaborators. They need to visualise collaborators while writing the test.
* May lead to over-engineering due to premature type (collaborators) creation.
* Not suitable for exploratory work or behaviour that is not specified in a user story (data transformation, algorithms, etc).
* Bad design skills may lead to an explosion of mocks.
* Behavioural tests are harder to write than state tests.

Knowledge of Domain Driven Design and other design techniques are required while writing tests.