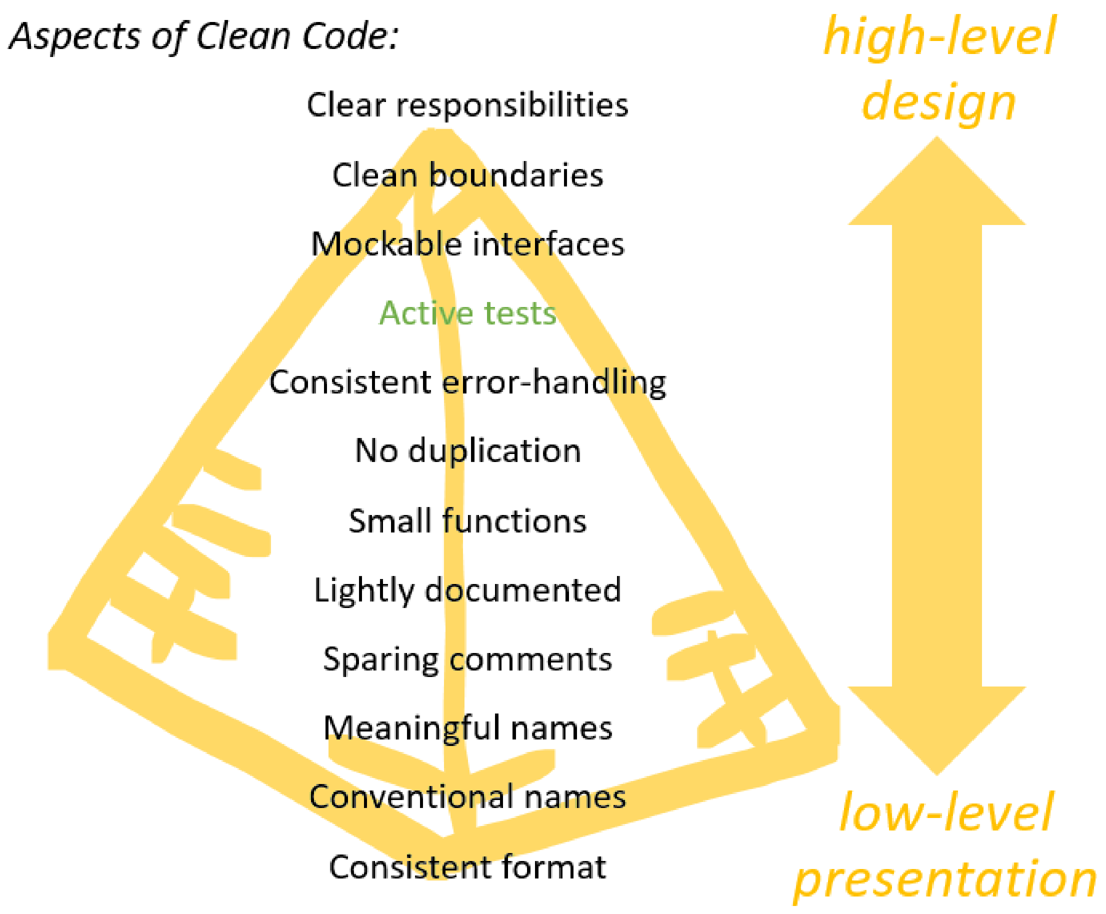
Test Driven Development

The 'School of Clean Code'

Test-driven development is one component of what some call the 'School of Clean Code'. The proponents of this school assert that it is important to keep all aspects of our code 'clean', in order to deliver reliable, maintainable software. These aspects range from low-level presentation of the source code and variable names, through to the high-level architectural design of the project:



The benefits of TDD

Why go to all this trouble of creating unit tests, refactoring into smaller functions, and thinking up meaningful (yet concise) variable names?

Broadly, we can identify three types of benefits that accrue from Clean Code (and Test-Driven Development in particular):

1. Day-to-day improvements in productivity. By having a set of automated tests to verify that the software is working, the development team can work in parallel on adding features, fixing bugs and refactoring code, confident that their changes will not introduce any unseen problems. In contrast, without any such tests, as the project gets bigger, it takes longer and longer to manually verify that changes have not broken anything.

2. Reduced risks of catastrophic error. We want to reduce or eliminate the chances of major errors that endanger lives (or business viability).

3. Enhanced lifetime for source code. What eventually 'kills' a software project, and how do we know when software is showing signs of ill-health? Proponents of Clean Code talk about the 'smell' of the code, caused by not adhering to the above aspects of clean code. Examples of 'code smell' include: functions and classes that are too big, duplicated code, variable names that don't mean anything (or still have an old meaning, despite a change of purpose), lack of testing, and tangled responsibilities. If nothing is done about this 'smelly code', then it starts to rot. It becomes impractical to add significant new features or to solve issues without creating problems elsewhere. The development team start to hate the project and leave. The business stakeholders decide to either withdraw from this business area, buy in technology from elsewhere, or start again from scratch. By keeping our code clean, we can eliminate the chance of it rotting, which means that it has a longer lifetime (assuming it still has business value).

Incremental code delivery

Test-Driven Development does NOT mean that we write all our tests and only then write the code. Instead, there is an incremental development of testing code and production code, summarized in these 'laws' below:

- First Law: You may not write production code until you have written a failing unit test.
- Second Law: You may not write more of a unit test than is sufficient to fail, and not compiling is failing.
- Third Law: You may not write more production code than is sufficient to pass the currently failing test.

Tests are written just before the production code, so that all production code can be tested as it is being written.

TDD Workflow

We also need to integrate refactoring into our workflow. When writing our production code so that it passes the test, the advice is to first work on just passing the test, and only then start refactoring the code to make it more elegant. These activities should be kept separate: we don't try to refactor when we have failing tests. We only refactor when we have a set of passing tests, so that we can make changes and check that all the tests still pass.

1. Part 1: Write a failing unit test.
2. Part 2: Write/edit the code so that all tests pass.
3. Part 3: Refactor all code to make it clean

There's a scenario here that is worth drawing attention to. What if, after writing the next unit test, we find that we cannot make the test pass without extensive refactoring? We seem to be caught between a rock and hard place: we are told we should only refactor against a suite of passing tests, but to pass the failing test we need to do the refactoring.

The answer here is to skip the failing test, so that it is temporarily removed from the set of tests. Now, with all tests passing, we can do the necessary refactoring so that we are ready to write the code to pass the test (which is currently being skipped.) Our next step ('write the failing test') consists of simply re-enabling the test which was being skipped. Once again, this test fails, yet we are now in a better position to 'make the test pass' (now that the code has been refactored, the required changes to the production code are more straightforward).

Characteristics of Good Unit Tests

There are several characteristics of unit tests. Some of these are identified by the acronym FIRST:

- Fast
- Independent
- Repeatable
- Self-validating
- Timely

In addition, we can make the following observations about well-written unit tests:

- Small: each test should only make one assertion (or a small number of assertions).
- Clean: the tests should be as clean as the production code.
- Readable: so that others understand the point.
- Contextualised: given an initial context, when this call us made, then we expect the following output / effects.

Summary

We've seen how test-driven development is part of the School of Clean Code, with claims of improved productivity, reduced chance of catastrophic errors, and enhanced software lifetime. The main workflow consists of repeatedly applying three steps: i) write a failing test; ii) make all tests pass; iii) refactor the code.

Not all teams use test-driven development. However, it's rare to find a team that does not have (or would like to have) some level of unit testing. The test-driven development methodology is quite an extreme position, and in some way represents an ideal that we can aim for, even if our actual software writing processes don't always match up.