Test-driven development
TDD in a nutshell

The development cycle in Test driven development follows a simple Red-Green-Refactor cycle

• Write a test that fails
• Write the simplest code that passes the tests
• Refactor (tests and code!), and make sure tests still pass
• Repeat
Benefits

Ensures:

• Testability
• High test coverage
• Fast and clear development cycle
• Tests also document usage
Tests

• Write simple and short tests
  • Usually we start the simplest tests and work our way up
  • Use the IDE to automatically generate class and method signatures for us

• Tests are the specification of your code
  • Once you write down a test, you know the feature is done when it's passing

• You can write multiple tests to cover a single feature
  • But always make sure your tests can fail
Code

- Write the **simplest** code you can to make **all** the tests pass
  - Including **code duplication**
  - Including **hard coding** the results
  - Including writing code that works only for the **specific** test input
- This part is done as soon as the tests pass
Refactor

This is the most important part!

• Remove any code duplication from the previous part
  • Both in production code and in the tests
• Redesign your class and tests
  • Refactoring is easy since our tests cover all of our functionality by definition
Productivity boosts

By focusing on writing tests first, we get a very efficient production cycle

• Tests are (usually) easier to write than production code
  • If it's harder, then there might be something wrong with your design
  • We pick the lower hanging fruit first, but once we got a test we know what our next goal is

• The coding part is usually pretty fast too
  • We always write the simplest code to make the tests pass
  • We're working incrementally, adding a small feature every time

• Refactoring is easier and safer (and therefore faster), since by the definition of TDD all our features are covered by tests
  • Impossible to break existing features without tests failing
  • But refactoring tests should be done more carefully
It's not (only) about the tests

Despite its name, the point of TDD isn't writing tests

• TDD is a methodology for writing production code
  • You should write tests regardless of whether you use TDD or not
• By writing the tests before there is any implementation, we can write our ideal implementation without uglying it (yet) with the necessary details.
• And, as a side-effect, it improves our unit tests
  • Our tests tend to be more declarative and readable, since they are the specification of our code
  • Units tests written after we wrote (a lot of) production code tend to be poorer
    • It's common to "be-little" the importance of tests if we already got our code working
    • Easy to forget what the code was supposed to be doing in the first place
    • Hard to remember all the edge cases

Author: Gal Lalouche - Technion 2017 ©
Testability and YAGNI

• TDD puts testability to the front of center
  • You can't even write code if it's not testable
  • Code tends to be more testable than usual: If our code is hard to test, it will be difficult to add new features
  • By writing the tests before the implementation, we focus on our external API instead of internal details

• Takes YAGNI to an extreme level
  • Not only are we not writing extra features, we're not even writing any more code than the bare necessities to make the tests pass
    • Even after refactoring, we don't usually add complex logics to our code
  • TDD takes the mentality of "Not tested? Not needed!"
Pros and Cons

• Use TDD when you:
  • Know what your inputs and output are, but you don't know what the ideal implementation will be
  • Have a lot of edge cases that you want to cover
  • Want to ensure high test coverage of your code

• TDD can be problematic when
  • Dealing with legacy code
  • Dealing with non-deterministic code
    • Either random or arbitrary (e.g., non-stable sort)
  • Testability isn't always the right choice
    • See Test Induced Design Damage
Live Coding Demo: Bowling scoring

Bowling rules:

- 10 frames
- Each frame has one or two rolls, trying to knock down 10 pins
  - Last frame can have 3 rolls, if you score a strike or a spare in the 10th frame
- Every knocked down pin awards a point
- Knocking down all pins in 2 rolls is called a **Spare**
  - Spares double your next roll
- Knocking down all pins in 1 roll is called a **Strike**
  - Strikes double your next two rolls