
dojo®
A Path to an easier life:
TDD with Code Craftsmanship
COME IN WE'RE
OPEN

“

The process of building software should also be considered an engineering discipline, just like with hardware.

”

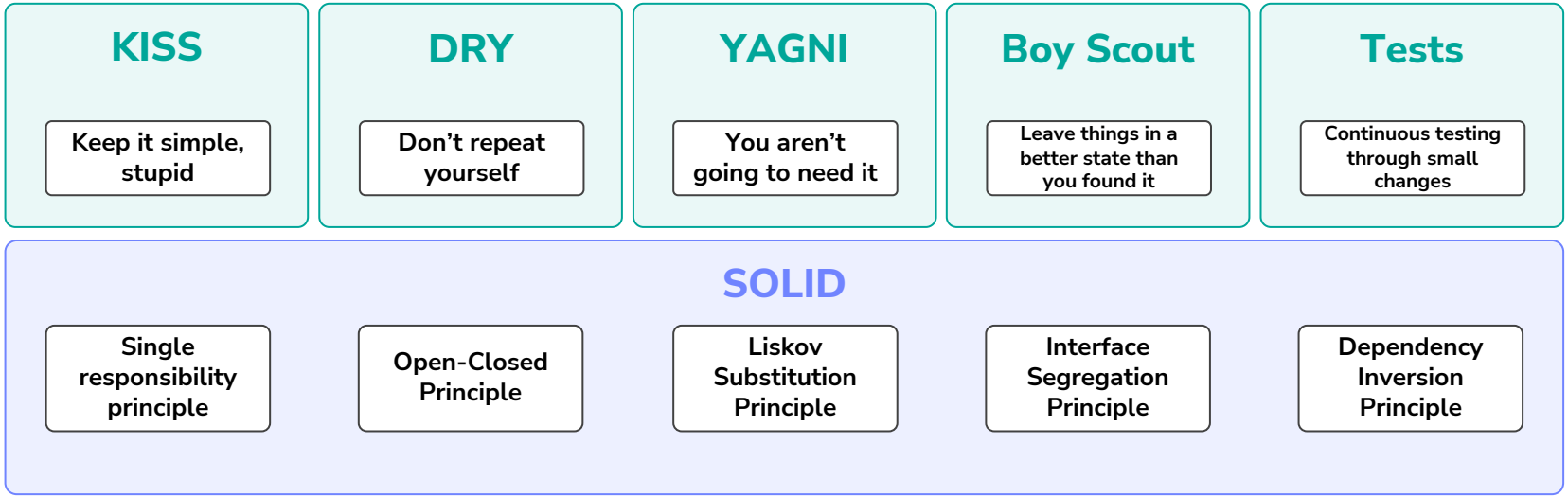
Margaret H. Hamilton

dojo

What is Code Craftsmanship?

A set of disciplines and principles to achieve

Pillars of code craftsmanship



What does that really look like?

- Break down your problems into small, manageable chunks
- Write small methods that have one job (helper methods are our friend!)
- Don’t over complicate it - take a step back and keep your goal in mind
- Tests!
- Write reusable and extendable classes and methods through interfaces

What does that achieve?

- Less spaghetti code
- Thoroughly tested codebases where you can confidently add functionality
- Code that is easier to read
- A better understanding of what your code is actually doing

dojo

Code Craftsmanship Examples

```
public void FizzBuzz()
{
    for (int i = 1; i <= 100; i++)
    {
        if(i % 3 == 0 && i % 5 == 0)
        {
            Console.WriteLine("FizzBuzz");
        }
        else if (i % 3 == 0)
        {
            Console.WriteLine("Fizz");
        }
        else if (i % 5 == 0)
        {
            Console.WriteLine("Buzz");
        }
        else
        {
            Console.WriteLine(i);
        }
    }
}
```

Kiss Example

Keep it simple, stupid

To the left is a classic solution to the known ‘FizzBuzz’ technical test:

- Go through every number from 1 to 100
- Print ‘Fizz’ if the number is divisible by 3
- Print ‘Buzz’ if the number is divisible by 5
- Print ‘FizzBuzz’ if the number is divisible by 3 & 5
- Print the current number if it is not divisible by either

How much more complicated can it be?

```
private readonly string[] outputs = new string[]
{
    null,
    null,
    "Fizz",
    null,
    "Buzz",
    "Fizz",
    null,
    null,
    null,
    "Buzz",
    null,
    "Fizz",
    null,
    null,
    "FizzBuzz"
};

public void FizzBuzz()
{
    for (int i = 1; i <= 100; i++)
    {
        Console.WriteLine(outputs[14 - (i % 15)] ?? i.ToString());
    }
}
```

Kiss Example

Keep it simple, stupid

This is the same method, with the same outcome, written in a ‘fancy’, complicated way.

To properly understand the code, you have to really think about it, and possibly step through it.

This is the heart of KISS - keeping things as simple and straightforward as possible.

DRY Example

Don't Repeat Yourself

To the right is some code for a basic API. There are 2 GET methods which grab something, and then log a message.

Notice how both methods log a message that is very similar and follows the same logic.

```
public async Task<HttpResponseMessage> GetObjects()
{
    var result = await _testRepository.GetObjects();
    var message = "";
    if (result.IsSuccessStatusCode)
    {
        message = $"Success! Status Code: {result.StatusCode}, Response: {result.ReasonPhrase}";
    }
    else
    {
        message = $"Fail! Status Code: {result.StatusCode}, Response: result.ReasonPhrase}";
    }
    Console.WriteLine(message);

    return result;
}

public async Task<HttpResponseMessage> GetObject(string key)
{
    var result = await _testRepository.GetObject(key);
    var message = "";
    if (result.IsSuccessStatusCode)
    {
        message = $"Success! Status Code: {result.StatusCode}, Response: {result.ReasonPhrase}";
    }
    else
    {
        message = $"Fail! Status Code: {result.StatusCode}, Response: {result.ReasonPhrase}";
    }
    Console.WriteLine(message);

    return result;
}
```

DRY Example

Don't Repeat Yourself

As you can now see, our code is much shorter, and easy to read.

We have extracted the previous if/else logic into a **reusable** method, which can be independently tested, and used across our solution.

```
public async Task<HttpResponseMessage> GetObjects()
{
    var result = await _testRepository.GetObjects();
    _logger.Log(result);
    return result;
}

public async Task<HttpResponseMessage> GetObject(string key)
{
    var result = await _testRepository.GetObject(key);
    _logger.Log(result);
    return result;
}

public class Logger : ILogger
{
    public void Log(HttpResponseMessage result)
    {
        var prefix = result.IsSuccessStatusCode ? "Success!" : "Fail!";
        var message = $"{prefix} Status Code: {result.StatusCode}, Response:
{result.ReasonPhrase}";
        Console.WriteLine(message);
    }
}
```

YAGNI Example

You aren't gonna need it

Take our previous, new and improved, logger. The current requirements of the API are GET methods only. We know that there is a chance that we will add a POST method in the future however the below code **violates** YAGNI - whilst we might need the change in the future, we might not. We do not **currently** need it, so we shouldn't add it.

```
// Might be adding Post in future implementation
public void Log(HttpResponseMessage result, HttpMethod method)
{
    var prefix = result.IsSuccessStatusCode ? "Success!" : "Fail!";
    if (method == HttpMethod.Post)
    {
        prefix = result.IsSuccessStatusCode ? "Your upload succeeded!" : "Your upload failed!";
    }
    var message = $"{prefix} Status Code: {result.StatusCode}, Response: {result.ReasonPhrase}";
    Console.WriteLine(message);
}
```

dojo

Where does TDD fit in?

“

It is better to define your system up front to minimise errors, rather than producing a bunch of code that then has to be corrected with patches on patches.

”

Margaret H. Hamilton

What really is TDD?

*The key principle is "**Test then implementation**"*

- A better understanding of your requirements
- Confidence to make changes
- Clean and modular code
- Increase in code and test quality
- Tests are documentation
- Fast feedback
- Less bugs

TDD: Not just writing tests

Think of the behaviour and outcome before you write the code

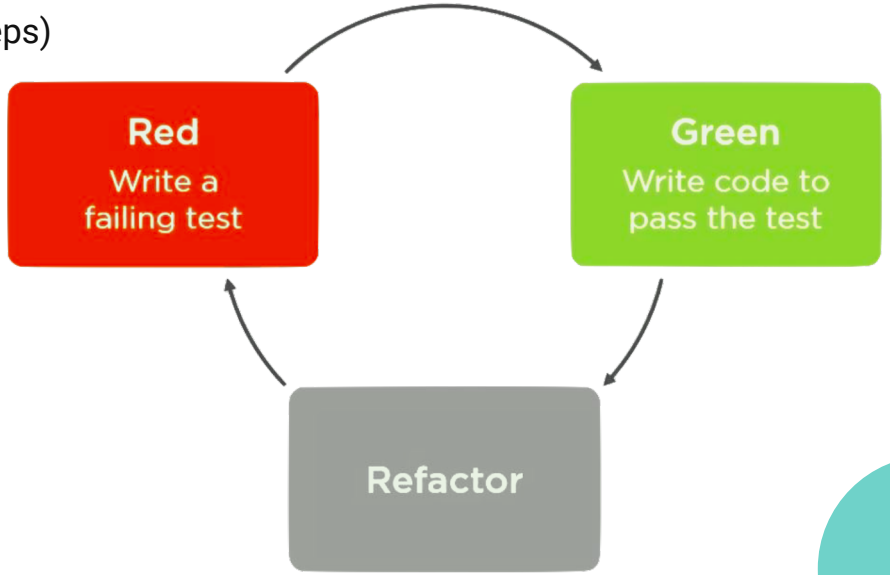
When working on development tasks, the first thing that we do is define requirements, as that way we know what to build. By writing our tests first, we are doing the same thing; we need to really understand our requirements, and then define what our code should do upfront - then we go and build it. This applies to every level of testing - from granular tests (unit) to behavioural tests and full system tests.

It’s about changing the way you think when solving problems - not just bashing out the code as quickly as possible, but truly knowing your acceptance criteria, understanding what are we getting out of adding this feature, and making sure that the customer is getting the wanted behaviour from the beginning.

What does this look like?

1. Add the test (including classes/methods you haven't yet written)
2. Run the test (it has to fail)
3. Write the minimum code needed to pass (and yes...it will be ugly)
4. Run the tests (they pass)
5. Refactor with confidence (and repeat above steps)
6. Party 🎉

You need to ensure your tests cover multiple scenarios - your implementation needs to support all/most expected inputs. As you add more test scenarios, you may need to refactor your tests.



Advanced TDD?

Behaviour Driven Development (BDD)

- Written plainly/Gherkin
  - Given = arrange / current state
  - When = act / what to test
  - Then = assert / what should happen
- Tests written by business analyst/acceptance criteria
- Tests the behavior of the service
- Each ‘step’ carries out the the action performed
- Written before unit tests - these are the **last** tests that will pass

```
Feature: Guess the word

  # The first example has two steps
  Scenario: Maker starts a game
    When the Maker starts a game
    Then the Maker waits for a Breaker to join

  # The second example has three steps
  Scenario: Breaker joins a game
    Given the Maker has started a game with the word "silky"
    When the Breaker joins the Maker's game
    Then the Breaker must guess a word with 5 characters
```

Purpose:

- Clear executable specification
- Document how the system behaves
- Tests come directly from acceptance criteria

When TDD isn’t possible...

- For emergency fixes with an extreme time limit
- When a system has no means to have automated tests
- Legacy codebases that violate Code Craftsmanship
- Legacy codebases that rely on technology that don’t work with unit tests (i.e. don’t support mocking)

How to make it possible...

- Backlog / implement tests once the emergency has been fixed
- Follow Boy Scouts (leave it in a better state than you found it)
- Refactor (both the tests and codebase!) where possible
- Investigate & **plan** how to modernise your tech stack and/or be able to add automated tests - break it down into small, manageable chunks
- Write new features following TDD & Code Craftsmanship where possible in legacy systems
- Sometimes you will need to just play around to see how much work will it be to retroactively implement Code Craftsmanship principles (i.e. adding abstractions) - write your tests and see if you can do it. If you can but it will be a long process, **document** how and backlog tickets - the smaller, the better!

Things to watch out for

- Meaningful tests over 100% code coverage
- Remove/refactor tests when they are no longer relevant / don't make sense anymore
- Remove/refactor badly written code - if you have your tests then you can reimplement easily
- Overtesting - you can end up with the same code tested multiple times - if you have passing tests that cover the logic, it is often an indicator that you do not need to write more complex code
- Have someone else review your test cases to ensure they fit your requirements
- Overhead - adopting the practice takes time and practise
- Short-lived, non-critical/not important projects (i.e. a quickly written tool to help that won't be used much or relied on) might not need TDD

How does Code Craftsmanship fit in?

A key part of TDD is writing **granular** tests. TDD and Code Craftsmanship goes hand in hand - by following the principles you end up with small, easily testable methods, and by following TDD you write small, granular methods for your tests.

dojo

Overview of SOLID

Single-responsibility principle

- “There should **never** be more than one reason for a class to change” - Robert C. Martin
- Every class or method should be focused on a single concern
- Don’t couple logic that has different responsibilities

Imagine you have a method that produces a document with formatted data. This can be changed in two ways:

1. The data can change
2. The formatting can change

These can change for different reasons. Therefore, each **concern** - data, formatting - should be in it’s own method. This way they can be **independently** changed and tested.

Open-closed principle

- "Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification" - Bertrand Meyer
- If a module is open for extension, it should be possible to add fields to the data structures it contains, or new elements to the set of functions it performs.
- If a module is closed it is available for use by other modules. This assumes that the module has been given a well-defined, stable description.

This usually involves inheritance through interfaces & abstraction, polymorphism or common design patterns such as the strategy pattern.

This means that we want to write code that we can easily add functionality to when we have new requirements, but not have to change/modify existing unless bugs are found.

Liskov Substitution principle

- If S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program (e.g. correctness, task performed, etc).
- You can replace an instance of a class with an instance of a **more** specific class without changing the behavior or causing errors
- New exceptions **cannot** be thrown by the methods in the subtype, unless they are subtypes of exceptions thrown by the method of the supertype.

For example:

A dog(T) can walk, wag their tail and bark. A puppy(S) can also walk and wag, but instead of barking, they breathe fire. This violates Liskov as the subclass (Puppy) has contract breaking behaviour.

Interface Segregation principle

- "Many client-specific interfaces are better than one general-purpose interface."
- A class should never be forced to implement an interface that it doesn’t use
- A class shouldn’t be forced to depend on methods they do not use.

To achieve this:

- Split interfaces that aren’t granular into smaller, more specific ones
- Ensure that the interface has single responsibility
- Keep things decoupled, allowing for easier changes and more tests

Dependency Inversion principle

- Depend upon abstractions, not concrete types.
- This means that as many classes & methods as possible should rely on interfaces and abstract classes
- Create loosely coupled classes and methods

This achieves:

- Reusable code which is easier to change and extend
- Easier to test as you can mock (fake) interfaces, meaning you can test the method **without** testing its dependencies