DevFluence™

Presents

TDD FUNDAMENTALS WITH TYPESCRIPT

Document Version 1.0.1

2023/01/11

Table of Contents

References

F.I.R.S.T Principles

Good tests follow five principles that form the above acronym. These are principles, or heuristics, and often we need to balance each against the others (making a test isolated can make it slower, for example), and make choices

Fast

- A developer should not hesitate to run the tests as they are slow.
- This is including setup, the actual test and tear-down should execute really fast (milliseconds) as you may have thousands of tests in your entire project.

Isolated/Independent

- A test method should do the **3 As** => **Arrange, Act, Assert**
- Arrange: The data used in a test should not depend on the environment in which the test is running. All the data needed for a test should be arranged as part of the test.
- Act: Invoke the actual method under test.
- Assert: A test method should test for a single logical outcome, implying that typically there should be only a single logical assert. A logical assert could have multiple physical asserts as long as all the, asserts test the state of a single object. In a few cases, an action can update multiple objects.
- Avoid doing asserts in the Arrange part, let it throw exceptions and your test will still fail.
- No order-of-run dependency. They should pass or fail the same way in a suite or when run individually.
- Do not do any more actions after the assert statement(s), preferably single logical assert.

Repeatable

- A test method should NOT depend on any data in the environment/instance in which it is running.
- Deterministic results - should yield the same results every time and at every location where they run.
  No dependency on date/time or random functions output.
- Each test should set up or arrange its own data.
  What if a set of tests need some common data? Use Data Helper classes that can set up this data for reusability.

Self-Validating

- No manual inspection required to check whether the test has passed or failed.

Thorough and Timely

- Should cover every use case scenario and NOT just aim for 100% coverage.
- Should try to aim for Test Driven Development (TDD) so that code does not need refactoring later.

The Three Laws of TDD

1. You are not allowed to write any production code unless it is to make a failing unit test pass.
2. You are not allowed to write any more of a unit test than is sufficient to fail, and compilation failures are failures.
3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

Red, Green, Reflect & Refactor Cycle

A revised Red-Green-Refactor cycle, designed to emphasize the importance of reading your code before refactoring it. This rhythm is key to the successful application of TDD.

It must be applied at a minute-by-minute scale; without the disciplined application of the cycle it is easy to drift off and start to problem solve at the monolithic level. Problem solving at the large scale greatly reducing the effectiveness of TDD as a thinking tool.

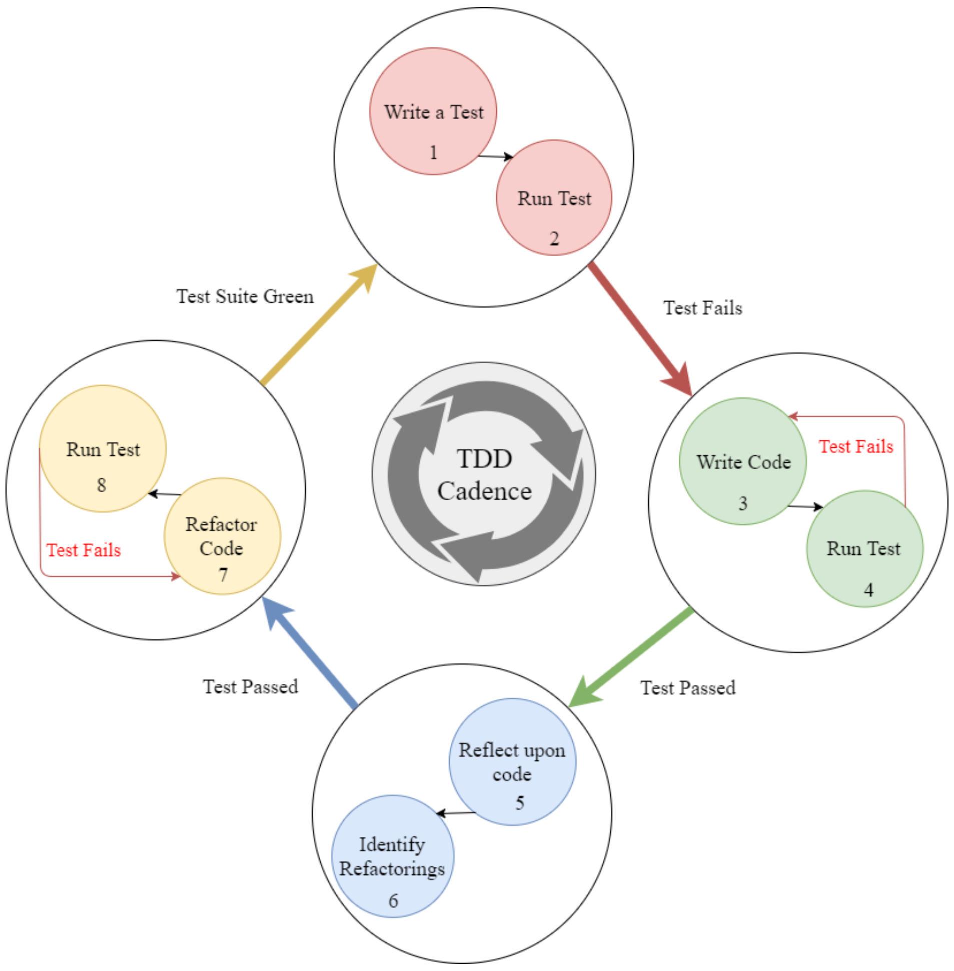


Figure 1: Red-Green-Reflect-Refactor Cycle

Green Bar Patterns

5 patterns to apply when solving a failing test. A good rule of thumb to follow when applying the strategies is to only have a red test for less than 1 minute. Anything more and you need to reverse and try again, you took too big of a step.

Fake It

This pattern is the simplest and the most utilized when building context. It is useful if you do not have examples or the examples are unclear. *If you have 2 or more good examples use triangulation instead*. The key to applying this pattern is to work in small increments until you find a generalized solution. To achieve this, start off with returning a constant, then gradually transform that into an expression with variables. Your brain is an amazing pattern matching machine, this green bar pattern makes use of this fact.

The pattern is often applied as follows:

- Start with a failing test
- Make the test pass as quickly as possible, this means returning a constant value.
  E.g.: if(value == 3) return "Fizz";
- Refactor only once all your tests are green

Triangulation

This pattern involves working with known examples and is great for building context. It feels very similar to the Fake It pattern in that when it starts off with hard-coding the specific examples in an attempt to find the general solution later.

After 2 or more specific examples, one should see a pattern emerging. It can take more than 3, if after 6 you still cannot see the pattern hit reverse gear, you are stuck and likely making a mess.

The goal of this pattern is to have used examples to guide the implementation to find a generic one that solves the solution allowing you to remove the hardcoded details.

As you move through the examples, you must vary your expected. E.g. when doing the age calculator, you should expect an age of 1, 2, and 3 respectively for the first three test. Failure to vary the expected result from one test to the next means you are not making forward progress of the implementation.

Be mindful of the need to refactor your test code as well as your production code when completing a Triangulation; this is because test code is more valuable than production and should be treated as such.
When triangulating, the most common test refactoring is to roll up related test into a single test case.

Also, be mindful of what portion of the problem you are solving, trying to use this technique to solve many different portions of the problem at once can lead to a big ball of mud. Stay focused on solving the problem one variable at a time.

One-to-Many

This pattern is useful for moving from a single item to a collection of items. This is because it is easier to write an implementation that works with a single value rather than a collection. By focusing on solving for a single value, one can focus on the core part of the problem being solved and only worry about the collection detail once the foundations have been laid.

Suppose we were creating an undo stack to track operations and allow them to be undone. We would start by only supporting a single operation to get started. Once we were comfortable that we had the concept working we can shift focus to handling many operations, thus expanding our focus from the core of the problem to the detail of the problem.

There are two common variants of this pattern. The first involves refactoring one test while the second involves using two tests. It is likely you will make use of one or both variations when applying this pattern.

Variation 1 – Evolving a public method

In this variation, I first get a test to pass dealing with a single instance of an object through a public method. I then refactor the implementation to make use of a collection, thus causing a compilation failure and a need to refactor my test to pass through a collection with 1 item.

Variation 2 – Isolating private method complexity

In this variation I first get my solution to work with a single instance of something internally, usually a private method. This allows me to solve the core problem without worrying about all the implementation details. Once I have solved the core problem, I then create a second test to address the need to deal with a collection of items in my private method.

Obvious

This is the most tempting pattern to use when applying TDD. Its application is simple: if the implementation is obvious implement it.

The trick to properly using this pattern is to realize that the obvious implementation is not so obvious. One can become stuck quickly and frequently when solving even simple problems. If you find yourself using this a lot it is likely you are overestimating your thinking capabilities. It is likely you will fail to take small steps and end up with a big ball of mud if you even manage to solve the problem.

A better way to think of this pattern is this, only follow this pattern if the implementation is trivial. If you coded an obvious implementation and you are struggling to get your test to pass, it is time to switch to reverse gear and try a new pattern.

Back Out

This is a special green bar pattern only applied in cases when you are stuck and need to use reverse gear.

The process is as follows:

- Revert to a green test run by commenting out your last test which is failing.
- Comment out or delete any production code no longer required
    - Ctrl + z works well here
- There are 2 options on how to move forward.

1. Create a learning test to verify your understanding of the 'tricky' bit of code holding you back.
    a. The 'tricky code' can be a single line or a whole method, bring it through into the test class and test your assumptions about it in a test.
    b. Once you have found the issue with the 'tricky code' roll your learning back into your production code by
        i. Uncommenting out the last test and ensure it fails
        ii. If there are no commented-out test to use, then write a new failing test
        iii. Bring your new solution through to make the test pass
        iv. Ignore or remove your learning test, it must not form part of the production test suite.
2. Revert to Fake It or Triangulation pattern to ensure you take small steps forward and remove any assumptions or gaps in your knowledge.

Learning Test

A learning test is a special type of test used to isolate and validate a snippet of code when applying the Back Out pattern. A learning test is used to bring focus to a problem area isolating it and bringing it under test. The technique can be applied to the code at any time, not just when applying the Back Out pattern. There are three well-known cases when you would want to use a Learning Test.

Case 1 – Struggling to refactor to a generalized solution

The first is when you are struggling to complete a refactor like generalizing after a Triangulation. In this case, trialing some options in a learning test can help ensure you stay green during the refactoring.

Case 2 – Struggling to get a green test

The second is when you are struggling to get a test green. This is often a sign of hidden assumptions about the problem domain or some misunderstood technical issue. When you find yourself stuck, isolate the issue with a learning test.

Case 3 – Exploring

There are often times when dealing with a new interface of technology you want to understand it before you try and use it. Using a learning test to isolate your work and clarify your goal is a great way to quickly understand the unknown.

The technique works the same for all cases it is just the circumstances driving the need to create the tests that differs.

TDD Practices and Principles

A listing of all practices and principles.

Core TDD Practices

- Rule of 3: Allow duplication to appear 3 times before you abstract it.
- Green Bar Strategies:
    - Fake It
        - Return a constant
    - Triangulation
        - Remove Fake It
            - Implement for 3 Fake It test before moving toward a generic solution after the 3rd.
    - One to Many
        - Single to collection
- 3 Laws of TDD
    - Always ensure a test fails for the correct reason before continuing. If it passes straight away make it fail first.
- 3 Stages of Naming
    - Move from meaningless to specific to meaningful names.
- Equivalence Partitions and Boundaries
- Test Factory Methods
    - Abstract common test setup per file
- Test Doubles
    - Received Asserting
        - Assert a method was called with the correct data.

Advanced TDD Practices

- Green Bar Strategies
    - Obvious Implementation: If the implementation is obvious, implement it
- Test Factory Methods
    - Avoid Default arguments in method
    - Avoid Optional Parameters in method
- Test Data Builders
    - Abstract common test setup
    - Uses Fluent syntax
- Test Doubles
    - Context Asserting
        - Assert the correct result was achieved through the correct use of various test doubles.

TDD Principles

- FIRST Principles
- SOLID Principles

TDD Gears

Intro

TDD is a complex and challenging discipline filled with many practices and principles. It can be hard to understand the reasoning behind some practices such as the rule of 3 or triangulation while solving seemingly simple katas.

At its core, TDD is a thinking tool. A tool to solve problems at the micro scale and to apply key software development principles. The TDD gears model was developed to help explain how to utilize TDD as a thinking tool.

Often as developers, we are used to solving a problem at the monolithic scale, attempting to hold all the moving parts in our head while we bash away at the keyboard. By applying TDD we are forced to start solving problems on a micro scale. That is, we need to tackle one small portion of the problem by conducting experiments to build and validate our mental models.

This seems sensible to believe: because of the micro scale problem solving, TDD was first created. In addition to helping slice up the problem, it also provides a framework for applying key development principles like SOLID, 3 Stages of Naming and Single Level of Abstraction.

This application of key Software Development principles can be achieved by purposefully reading and reflecting upon our code as we pass through each iteration of the TDD cycle. Thus, the *Red-Green-Refactor* cycle becomes the *Red-Green-Reflect-Refactor* cycle, highlighting the importance of applying software development principles iteratively as we evolve the solution.

The Gears

In the Gears model, each gear represents both a degree of adherence to the practices and scale of thought. The principles should always be strictly followed in all gears. This is because the principles are used to drive Professionalism by shaping the maintainability and quality of the solution. Professionalism must always be present regardless of the problem domain.

In low gear we are trying to build context. We move slow and deliberately, trying to understand the problem at hand strictly following our practices and applying core green bar patterns like Triangulation and One-To-Many, while we avoid the Obvious Implementation trap.

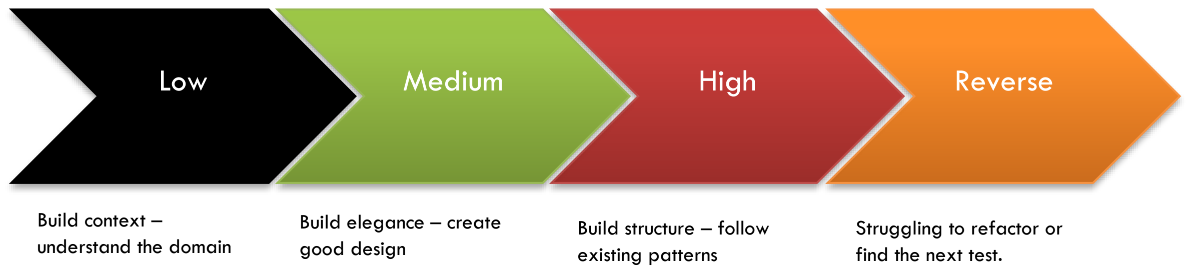
As we shift into medium gear, we focus on building elegance through good design. Having established an understanding of the problem, we look to build maintainable code. We now have access to all the green bar patterns and can start to short circuit some practices like the Rule of 3. The risk here is that we get stuck and need to make use of reverse gear.

High gear is used to either follow existing patterns or solve a large test such as an integration or BDD style test. Because the goal is much larger the scale of thought required to solve the test needs to be larger. Being good students of TDD, we will have built up to solving a large test by shifting through low and medium gear. It is because of this stepped approach we can take on more risk while still being quite relaxed about the application of various TDD practices.

Reverse gear is used to get unstuck. In any gears, it is possible to find yourself stuck, reverse offers up a solution on how to make progress when you have been struggling with a red test or tricky refactor.
A good rule of thumb for when to use reverse is red/red/reverse. That is, if after two attempts to make the failing test pass you cannot do so, reverse back to green and try again.

The Model

How the gears fit together.


Low
Medium
High
Reverse
Build context – understand the domain
Build elegance – create good design
Build structure – follow existing patterns
Struggling to refactor or find the next test.

- **Low**
    - Unsure of Domain or algorithm being implemented
        - Evolve slowly to **build context -** understand the problem
    - Strict following of all core practices
    - No advanced practices used
    - Low risk of reversing
- **Medium**
    - Some familiarity with domain or algorithm being implemented
        - Evolved to **build elegance** - create good design
    - Short circuit some TDD practices
    - Mixes advanced and core practices
    - Moderate risk of reversing
- **High**
    - Familiarity with domain or algorithm being implemented
        - Evolve to **build substance** - follow existing patterns or solve a larger test
    - Heavy short-circuiting of TDD practices
    - Mixes advanced and core practices
    - Moderate / High risk of reversing
- **Reverse**
    - Like getting stuck in the mud it is best to reverse and try a different angle.
    - A good rule of thumb for when to use reverse is red/red/reverse.
        - If after two attempts to solve the test it is still red, reverse.
    - When to use:
        - Struggling to complete a refactoring
        - Struggling to find next test
        - Feels like the solution is stuck and cannot move forward
    - How to use:
        - Apply the Back Out pattern
        - Restart in **Low gear**
        - Find a new point of attack: Use your test to take small steps to build a "virtual stack trace" validating your understanding of the code at each step. Use a Learning Test to validate a portion of the problem if struggling to understand the inner workings of a private method.

Software Development Workflows

The following is a visualization of two styles of software development expressed as influence diagrams as per Gerry Weinberg's Quality Software Management.

An arrow between nodes means in an increase in the first node implies an increase in the second. An arrow with a circle means an increase in the first node implies a decrease in the second.

Below in figure 1 the effects of stress on the development workflow are expressed as a positive feedback loop. The more stress increases the less likely it is the developer will test their changes by launching the application and poking around. The more errors they will make, thus the more stress they will feel. This type of loop self-reinforces, thus the more stress you feel, the less testing you will do. The less testing that was done, the more errors made. The more errors made, the higher stress levels rise. Wash, rinse, repeat and watch the overtime required to complete increase.

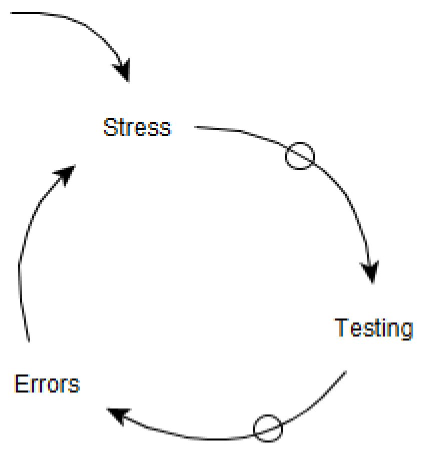


Figure 2: Non-TDD Developer Workflow

The only way to break this cycle is to introduce a new element, replace one of the elements or change the arrows.

Below in figure 2, the impact of TDD on a developer's workflow can be seen. TDD being the practice of writing a failing automated test before writing production code. This mechanism allows the developer to build up a suite of automated test suite used to check program correctness at any moment.

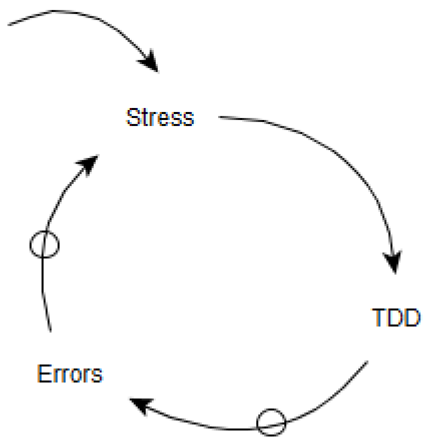


Figure 3: TDD Developer Workflow

Now when you feel stressed, simply run the test suite. The more stress you feel, the more you should run your test suite. This gives you a good feeling, building your confidence, thus combating stress and reducing the number of errors made when writing code.

Why does stress feature so prominently in our models?

Stress releases a hormone called cortisol, it affects brain function, mood and generally decreases your decision-making abilities while impacting your working memory, thus reducing your programming capabilities.

It is nearly impossible to notice the degradation of abilities, you will only notice the side effects, eventually.

The reason for this brain behaviour is rooted in survival, when faced with a life and death situation instinct kicks in and shuts down the thinking portion of your brain, thus defaulting to automated behaviors such as fight or flight. Back in history this was used to keep you from being eaten by a saber tooth tiger, today it is rarely useful.

Thus, a non-TDD workflow in one designed to increase stress and decrease overall development efficiency leading to longer working days as you stay stuck in the cycle of increased stress increasing errors increasing stress.

A TDD workflow is one designed to work with the constraints of our neurobiology, thus limiting the impact of stress. This stabilizes your overall development efficiency leading to shorter working days and higher quality code.

Refactoring Priorities

The order in which one should evaluate refactoring when reflecting upon their code in order of importance.

1. Identify concepts
   a. Think Single Responsibility of SOLID
      i. Each collection of responsibilities should only have 1 reason to change.
   b. Collect responsibilities at the method level
   c. Collect responsibilities at the class or concept level
      i. Group related methods into new classes – Be sure to create a related test class and migrate the affected test.
2. Name things well
   a. 3 Stages of Naming as applied to variables, methods and class names
      i. Drive for meaningful names, avoid specific names in tests.
3. OLID and SOLID principles
   a. Open/Closed
   b. Liskov substitution principle
   c. Interface segregation
   d. Dependency inversion
4. DRY (Don't repeat yourself)
   a. Every piece of knowledge must have a single unambiguous, authoritative representation in the system.
   b. Often this principle is miss-understood with developers building abstractions for things that look similar, but represent different concepts. It is only repeated if the code represents the same concept. That is the same sentence can be used to describe what it represents.

Test Smells

As wonderful as TDD is, there are times when tests become poorly designed, thus they may need some refactoring to improve their design. Because development is such a creative process, both newbies and veterans can fall into these traps. Knowing what they look like and how to handle them are a developer's best tools to combating them.

Here are a few common test smells to watch out for.

1. **Exposing internal details** – When a test needs to make use of the internal details of the object being tested is a clear sign that encapsulation is broken.
   a. An example of this would be making properties public to check state in a test
   b. Or relying too heavily on Mock assertions that methods were called in the correct order. Test should be ignorant of internal details, they only care about results.
2. **Passing too much information** – When the object being testing has many parameters being passed into its constructor it is time to break it up into several new objects.
   a. A good rule of thumb is if the object has 5 or more items being passed into the constructor it is time to explore the single responsibility principle of SOLID and break up the object.
   b. If any of the data being passed in is state for the object to operate on, try and move towards stateless objects. This means passing the state in to the method that needs to operate on it, not in via the constructor.
3. **Tests replicate production code** – Code should not be replicated. If you find yourself replicating production code in test, stop and reverse, you are about to fall into a trap.
4. **Highly coupled code** – This is often an artefact of failing to reflect and refactor after each TDD cycle.
   a. It can result in fragile test – when a test fails to run or compile due to unrelated changes to the object being test.
   b. It is often driven by poor/quick design choices or lack of reward structure that encourages cleaner more maintainable code over short term optimized feature production.
5. **Test code duplication** - Often tests need to do similar things, thus code can be come quickly duplicated causing maintenance problems in the future.
   a. To help manage the duplication it is recommend to make use of builder patterns such as factory methods and test data builders to help centralize the duplication with making use of Test Setup and Teardown attributes.
6. **Test case abuse** – Developers look to minimize the amount of work they need to do. One useful feature for being to reuse a test is a test case, it allows the developer to vary the input into a single test.
   a. The danger with this is that an entire method becomes tested with a single test that has many test cases. When this happens the test name is no longer meaningful about the scenario being tested. The test becomes blunt instrument with which the developer bludgeons the code in the name of efficiency.

b. Test Cases are often an artefact of Triangulation and Fake It green bar patterns. They exist to roll up test related to a single concept. They do not exist as an efficiency tool for developers to avoid ‘duplication’ in code. When in doubt, be verbose and make a new test.

S.O.L.I.D Principles

The SOLID principles as a group were collected, explained and popularized by Robert C. Martin about 20 years ago – his first articles on them were circulated in the late 90s and the principles were covered extensively in his book, Agile Software Development: Principles, Patterns and Practices, published in 2002. They are another attempt to create a theory of programming – a set of principles to guide how we write and structure code. SOLID is a mnemonic formed from the first letter of each principle:

- Single-Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency-Inversion Principle

The benefit of SOLID is that these are principles that are more concrete than the abstract ideas of coupling and cohesion. They allow us to aim at reducing coupling and increasing cohesion through more practical methods.

These principles are simple in formulation but are extremely deep in their application. Simply knowing the definition does not suffice to embed the ideas into what we do – we need to really engage with these principles, discuss them as we do our jobs and especially as we do deliberate practice as through this we will be able to utilize these principles well.

Knowing each principle in isolation is good. But applying all these principles as a solid block is when they become truly powerful.

The Single-Responsibility Principle

"A class should have only one reason to change"

Robert C. Martin himself acknowledges that this is very similar to the concept of cohesion developed by many other authors from the late 70s on, but his formulation is subtly different. Instead of thinking about the cohesion of a class and applying various analyses to figure out a cohesion score, he says that if there is more than one reason for a class to change then it is doing too much and should be split into two classes that each does one thing.

While this sounds fairly simple, it's also probably the most difficult of these principles to do in practice. We don't always know the way things are going to change, for example. It's not a good idea to segregate out things that are never going to change because then we are doing extra work that is unnecessary. So we need to think about and possibly predict the way things may change in the future. This is tough, and we will likely not get it right most of the time. Then a year down the road someone will ask "why didn't you separate out this part of the class – you didn't follow the SRP!"

The key here is to realize when you are changing a class in more than one way and look to separate out the parts that are changing. One way to discover which classes are changing is to use your source control history to generate a heat-map of your code – the classes that are being changed often over a long period are likely doing too much (they could also be violating the Open-Closed Principle).

The Open/Closed Principle

"Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification."

This principle followed to the extreme, would mean that every time we want to add functionality we create a new class, or a new function and never change existing code. Of course, this is not a realistic dream, but it is an idea that we can work towards. Why? Because using this approach we are able to add functionality to a system without breaking existing functionality and without causing knock-on changes. If we are able to recompose things in different ways to achieve different outcomes without rewriting the pieces we are using to compose our solution, then we have achieved a level of flexibility that is very powerful.

When modules are aligned with the Open/Closed Principle, they exhibit two attributes as referred to in the principle definition. They are:

- Open for extension. This means the module can be extended in some fashion to change its behavior.
- Closed for modification. While the module can be extended to change its behavior, it is not necessary to change its source code. Ideally, we would be able to extend the behavior of the module without even needing to recompile the binary.

This principle is not actually possible to follow in a procedural language because it depends on some form of abstractions and polymorphism to achieve.

The classic example of this principle is a switch statement that switches on an enum or another variable. This switch can almost always be replaced by a class hierarchy that uses polymorphism to execute the correct method – this is the one that follows the OCP because you can inject new behavior by adding another class with a new polymorphic method.

Like our caveat with the SRP, it is not possible to always anticipate the ways in which a class will need to be extended. The idea is to use our experience to anticipate the ways a class/module will most likely need to be extended and ensure that it is open for extension in that area. Once again using the example of a deserializer – it is very likely that we will want to extend this by adding new types of sources for the bytes to deserialize, so we need to ensure that it is open for extension in that area by providing a way of plugging in a strategy.

The Liskov Substitution Principle

"Subtypes must be substitutable for their base types"

Anyone who has ever used inheritance has likely violated this principle many times unknowingly. On a number of occasions, I've inherited from a base class and provided overrides that simply throw a NotImplementedException() or the like. This subclass is violating the Liskov Substitution Principle because this subtype is not substitutable for the base type in all circumstances. However, there are still times when this is expedient. A good example is the ADO.NET library – not all database providers implement all the features defined by ADO.NET, but most of them implement most of the features, and it would be very difficult to separate out every feature into different interfaces, so it makes more sense to simply have some NotImplementedExceptions() thrown here or there than to try and ensure that the LSP is never violated.

One way to ensure that LSP is followed is to separate out interfaces – that is: follow the Interface Segregation Principle. Then your subtypes will only need to implement the interfaces that directly apply.

The Interface Segregation Principle

"Clients should not be forced to depend on methods they do not use"

This principle is essentially an acknowledgment that cohesive classes are an ideal and not always possible. There are times when we need to build a class that has a large set of methods on its interface – for example when building a façade that acts as the adapter to a number of different back-end systems. However, the Interface Segregation Principle says that this large interface should be split into sections depending on the needs of the various types of clients that are going to connect to it. That is, if there are a set of methods that will be used by one type of client, and a different set of methods used by another, then these methods should be split into two different interfaces so that each client type can depend on a specific abstract interface that only contains the methods they need, instead of depending on the fat interface itself.

The segregation of interfaces provides a way of decoupling client classes. If two different client classes depend on one concrete API then changes made to the API for the benefit of one client class will impact on the other client class, even if those changes were not made to methods that client class needs. This means that the two client classes are coupled to each other through their common dependence on the API. If that API is broken up into two different segregated interfaces and each client class depends on a different one, then changes made to the concrete implementations will only affect the client that uses the affected interface. This means the two client classes are no longer coupled to each other.

The Dependency-Inversion Principle

"A. High-level modules should not depend on low-level modules. Both should depend on abstractions.

B. Abstractions should not depend on details. Details should depend on abstractions."

I would argue that this is the most widely implemented of these SOLID principles. The idea is embedded in every “container” out there and is now much more baked into frameworks like ASP.NET MVC.

The idea in the Dependency-Inversion Principle is to invert the natural dependencies that normally arise when coding; for example, if you are building a user interface and need to call a data layer to access a database you would ordinarily simply instantiate a data layer object and call it. This is what Robert C. Martin refers to as the traditional procedural method. He argues that the well-factored Object-Oriented method, however, is for the user interface to depend on an abstraction of the data layer and have the concrete implementation given to it so that this traditional dependency is inverted.

This has many benefits:

- Reducing the coupling between concrete classes, which means each of them can change in isolation
- Allowing for new implementations of the data layer without affecting the user interface (the Open/Closed principle)
- Allowing for new insertion of decorators or other classes even at runtime or depending on configuration – for example, we could create a LoggingDataLayer that uses the original data layer implementation but adds logging to it
- Depending on abstractions means our classes are less brittle as abstractions change less often than implementation details
- Testing each of the classes can be done in isolation – we don’t need a concrete data layer in order to test the user interface

Equivalence partitions and Boundaries

An equivalence partition is a group of input values where the expected behavior/logic is the same. A boundary is where two equivalence partitions meet, said in other words, where adjacent input values belong to different equivalence partitions. This is best illustrated with an example.

Given the following function:

```
function print(input: number)
{
    if (input <= 0) return "TDD";
    if (input > 0 && input <= 99) return "is";
    return "awesome"; //Anything 100 or above
}
```

There are 3 equivalence partitions:

1. "TDD", which groups all input values that are less than or equal to 0
2. "is", which groups all input values from 1 to 99
3. "awesome", which groups input values greater than or equal to 100

There are 2 boundaries:

1. "TDD" - "is"
2. "is" - "awesome"

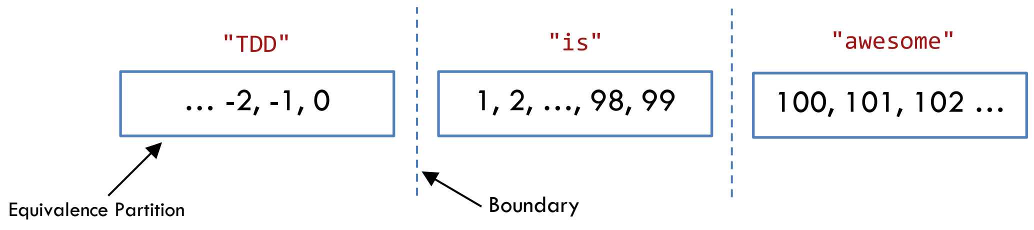


Figure 4: boundaries and equivalence partitions for the print() function.

Identifying boundaries and equivalence partitions allow for a more systematic and comprehensive approach to testing, automated or manual. The input values before, after and on each boundary, should be used to test with. Sometimes there is no actual value on the boundaries, in these cases, the values before and after each boundary are used.

Looking at the above example you would test using 0 and 1 as input values from the "TDD" - "is" boundary and 99 and 100 from the "is" - "awesome" boundary. Selecting your tests this way increases the chances of finding the infamous, and far too common, off-by-one error that happens around boundaries. It also means your tests have a higher chance of covering all the different behaviors that exist in the code being tested.

In the above print() function the values in the equivalence partitions are linear/sequential. This isn't a rule. Value in an equivalence partition can also be non-linear/non-sequential. To illustrate this let's use another example.

Given the following function:

```
function string getFizzBuzz(input: number)
{
    if (input % 15 === 0) return "FizzBuzz";
    if (input % 3 === 0) return "Fizz";
    if (input % 5 === 0) return "Buzz";
    return input.toString();
}
```

There are 4 equivalence partitions:

1. "FizzBuzz", which groups all input values that divisible by 3 and 5 (15).
2. "Fizz", which groups all input values that divisible by 3.
3. "Buzz", which groups all input values that divisible by 5.
4. Other, which groups all input values that aren't divisible by 3, 5 or 15.

There are 3 boundaries:

1. Other - "Fizz"
2. "Fizz" - "Buzz"
3. "Buzz" - "FizzBuzz"

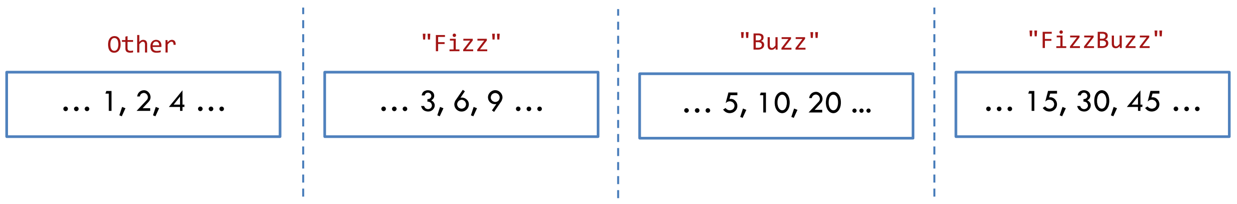


Figure 2: boundaries and equivalence partitions for the getFizzBuzz() function.

When identifying equivalence partitions and boundaries look for where behaviour changes. A behaviour change indicates a boundary. Once the boundaries are identified, group all input values causing a behaviour into an equivalence partition.

Breaking Dependencies

In TDD it is common to use objects that look and behave like production objects but are simplified for testing. These simplified objects are called Test Doubles. They are used to break production dependencies so the code is more testable and does not cause side effects like updating a DB.

Often people use the term Mock to refer to different types of test doubles. Mixing test double implementations will influence test design and can increase the fragility of the tests, potentially making it difficult to refactor later.

XUnit patterns[1] advocates for five test double types. We will talk about the three most common types: Fakes, Stubs and Mocks.

Fake

"An object with a simplified working implementation"

An example of a fake object could be a user repository that does not connect to a DB but instead uses an in-memory list to store data, thus allowing us to test the fetch user use case without performing time-consuming DB activities.

Beyond making it easier to test, fake objects can be used to prototype functionality by deferring key decisions.

[1] http://xunitpatterns.com/Test%20Double.html

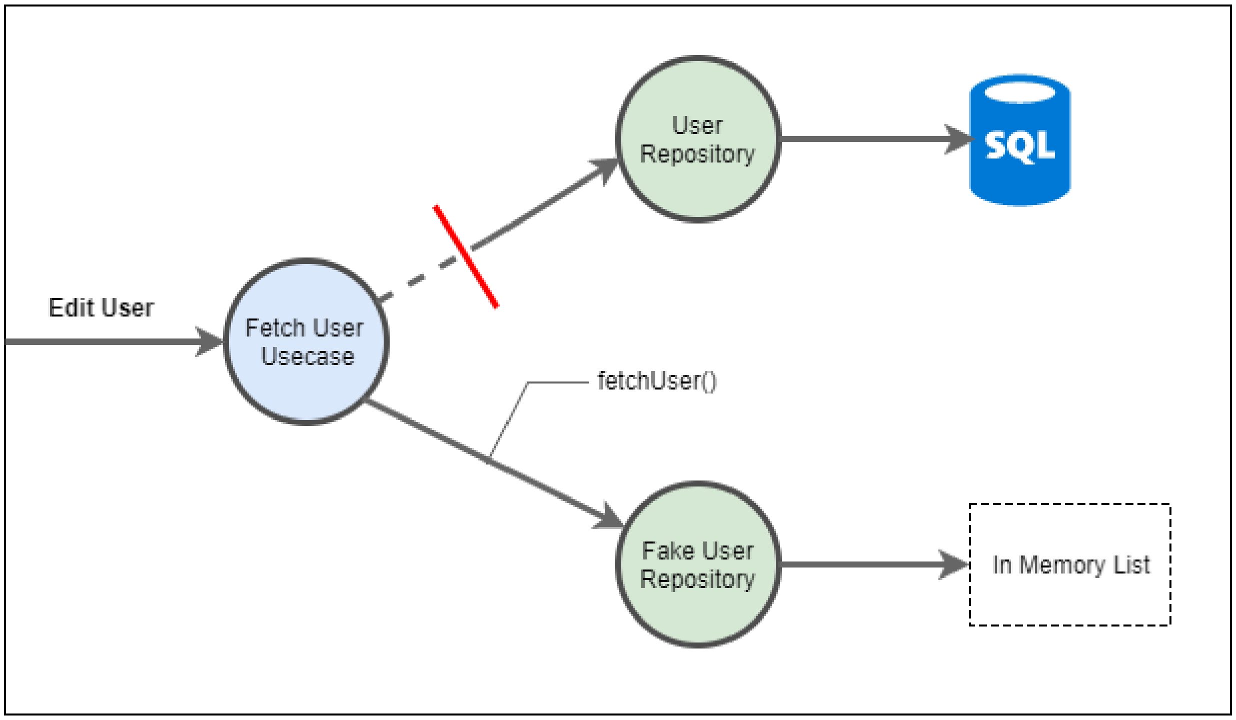


Figure 5: Fake Object Diagram

Stub

"An object that holds predefined data and uses it to respond to method calls during a test run"

An example of this is the Character Copy Kata. In this Kata a Copier object needs to read from a source and write to a destination without using production objects.

To achieve this one would then Sub out the source, as per the interface, to respond with specific data when called. This concept differs from Fake Object in that no state is saved to be used later in the test execution. Stubs are simple in that they merely respond to request with predefined data.

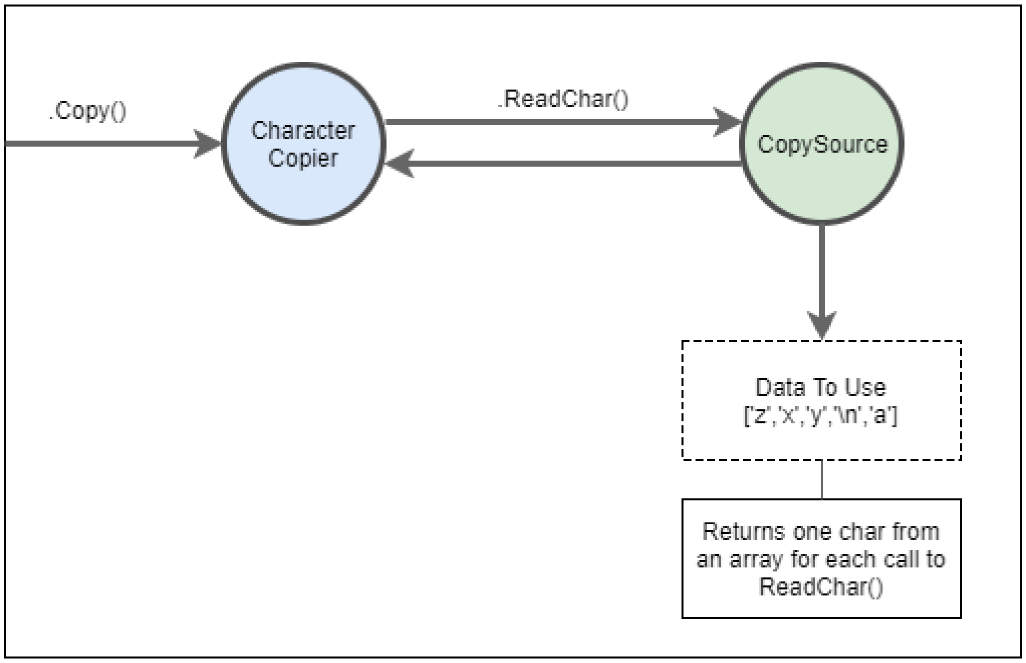

.Copy()
Character Copier
.ReadChar()
CopySource
Data To Use ['z','x','y','\n','a']
Returns one char from an array for each call to ReadChar()

Figure 6: Stub Object Diagram

Mock

"An object that registers calls received. The test then verifies that the expected method calls where performed"

Mocks are used when we do not wish to invoke production code or when there is no easy way to verify the intended code was executed. An example is a password reset feature where we would never want to send an actual email, instead, we would want to verify that the email service was called.

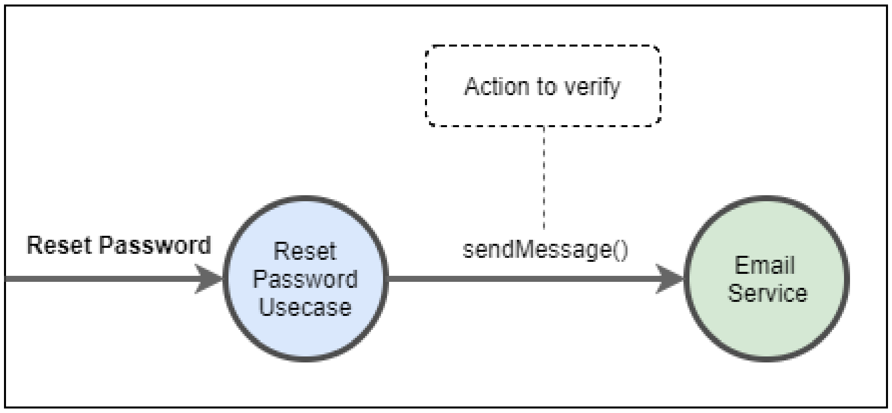


Figure 7: Mock Object Diagram

Leaky Abstraction Trap

It is very easy to misuse this type of Test Double. One common mis-use is called the 'Leaky Abstraction Trap'. An example would be a test that asserts methods where called and in a specific order. This makes it hard to change internal structure without refactoring test.

This is a trap because of the fact that a test should have two reasons to change there was a bug or the requirements changed. If you find yourself making structural refactoring's in your code and refactoring your test at the same time you have leaked too many implementation details, thus making your test brittle.

The use of Mocks is perfectly acceptable and a normal thing to do while engaging with TDD. Making almost exclusive use of them as your primary mechanism to assert test failure or success leads to expensive and brittle test. Your test should know as little as possible about the internals of what is being tested.

Using Mocks to check that an orchestration service called all the correct dependencies in the correct order is an example of the 'Leaky Abstraction Trap.' Instead, use Stubs to configure the scenario as expected, then ensure the correct result was achieved little or no reliance on mock asserting. Instead favor asserting that the correct result was achieved, not that the correct methods were called in the correct order.

Katas

Chillisoft

Rock Paper Scissors Kata

The Kata

Rock Paper Scissors is a game involving two players making pre-defined hand gestures at each other. The gesture that each player uses is played against the other, with a winner being decided based on the rules being used.

The three gestures used in base Rock Paper Scissors are... well... rock, paper, and scissors. The way these are scored is as such: Rock beats Scissors, Scissors beats Paper, and Paper beats Rock. It gets a lot more complicated when you introduce new gestures, but let's keep it simple for now.

We want you to create a backend for the game that we can use to hook up to our many game clients we're going to be creating.

Test Cases

Player Move	Opponent Move	Result
Paper	Rock	Player Wins
Paper	Scissors	Player Loses
Paper	Paper	Tie
Rock	Scissors	Player Wins
Rock	Paper	Player Loses
Rock	Rock	Tie
Scissors	Paper	Player Wins
Scissors	Rock	Player Loses
Scissors	Scissors	Tie

Bonus

Extend the game engine to include the rules for Rock, Paper, Scissors and Spock.

The new moves to include are:

- Spock smashes Scissors
- Paper disproves Spock
- Rock crushes Lizard
- Lizard poisons Spock
- Scissors decapitates Lizard
- Lizard eats Paper
- Paper disproves Spock
- Spock vaporizes Rock

Extra bonus:

After implementing the additional rules make sure your code has no if statements.

Fizz Buzz Kata

The Kata

Return “Fizz”, “Buzz” or “FizzBuzz”.

For a given natural number greater zero return

- “Fizz” if the number is divisible by 3
- “Buzz” if the number is divisible by 5
- “FizzBuzz” if the number is divisible by both 3 and 5

Test Cases

Input	Result
1	1
2	2
3	Fizz
4	4
5	Buzz
6	Fizz
9	Fizz
10	Buzz
15	FizzBuzz
20	Buzz
30	FizzBuzz
75	FizzBuzz

Bonus

Add the following new rule, if a number is prime return Whiz. Only worry about numbers up to 100.

Input	Result
1	1
2	Whiz
3	FizzWhiz
4	4
5	BuzzWhiz

Age Calculator Kata

The Kata

Calculate a person's age at a given date. Do this by creating a single public method that takes a person's birth date and a target date to compare to and returns their age as an integer.

Example

Brendon was born on 30 September 1982 how old was he on 5 October 2001? Answer 19.

Bonus

In the age of Social Media birthday, weeks are a big deal. To allow for this, add a new method to the Calculator that returns the start date of the person's birthday week, as a string, according to the following rules.

- If my birthday is Thursday – Saturday, my birthday week starts on the Sunday of that week. E.g. I was born on September 9, 2017, my birthday week will start September 3, 2017.
- If my birthday is Sunday-Wednesday, my birthday week starts 6 days back. E.g. I was born on September 3, 2017, my birthday week will start August 28, 2017.

String Calculator Kata

Roy Osherove (with modifications by Chillisoft)

Rules

1. Strictly practice TDD: Red, Green, Refactor
2. No use of the debugger or any console writes are allowed.
   2.1. Make use of a learning test to focus on the troublesome code.

The Kata

1. Create a simple String calculator with the signature: **function add(input: string): number**
   1.1. The method can take 0, 1 or 2 numbers, and will return their sum (for an empty string it will return 0) for example **"" or "1" or "1,2"**
   1.2. Start with the simplest test case of an empty string, then move to one and two numbers
2. Allow the add() function to handle an unknown amount of numbers
3. Allow the add() function to handle new lines between numbers (in addition to commas).
   3.1. the following input is ok: "1\n2,3" (will equal 6)
   3.2. the following input DOES NOT need to be handled: "1,\n" (not need to prove it - just clarifying)
4. **Support different delimiters**
   4.1. To change a delimiter, the beginning of the string will contain a separate line specifying the custom delimiter. This input looks like this: "//{delimiter}\n{numbers…}" (Note that the curly braces are representing the sections of the input and are not input formatting).
   4.2. For example: "//;\n1;2" should return a result of 3 because the delimiter is now ';'.
   4.3. The first line is optional (all existing scenarios should still be supported).
   4.4. Do not worry about supporting the specification of '\n' as an explicit custom delimiter. New lines should always be supported as delimiters in your number string.
5. Calling add() with a negative number in the input will throw an exception "negatives not allowed" - and the negative that was passed, if there are multiple negatives, show all of them in the exception message
6. Numbers bigger than 1000 should be ignored, so adding 2 + 1001 = 2

Bonus

1. Delimiters can be of any length with the following format: "//[{delimiter}]\n{numbers…}"
   1.1. For example: "//[***]\n1***2***3" should return 6.
   1.2. Note that the square brackets are required around the multiple character delimiter.
   1.3. A square bracket is not a valid delimiter.
2. Allow multiple delimiters like this: "//[{delim1}][{delim2}]\n{numbers…}"
   2.1. For example "//[*][%]\n1*2%3" should return 6.
   2.2. Note that once again the square brackets are required around each custom delimiter.
3. Make sure you can also handle multiple delimiters with length longer than one char

Character Copy Kata

The Kata

In the interest of better understanding using test doubles, write a character copier class that reads characters from a source and copies them to a destination. It must copy and write one character at a time.

To do this create a `Copier` class that takes in a `Source` and `Destination`. `Source` has one method `readChar(): string` and `Destination` has one method `writeChar(c: string): void`. The `Copier` class has one method `copy(): void` that when called will read one character from the `Source` and write it to the `Destination`. The relationship beeen the artefacts can be seen in Figure 1 below.

The copying and writing are done one character at a time until a newline ('\n') is encountered. Then the processing stops without writing the newline. For this kata, only the `Copier` class may exist as a concrete. You are to use a test doubles for `Source` and `Destination`.

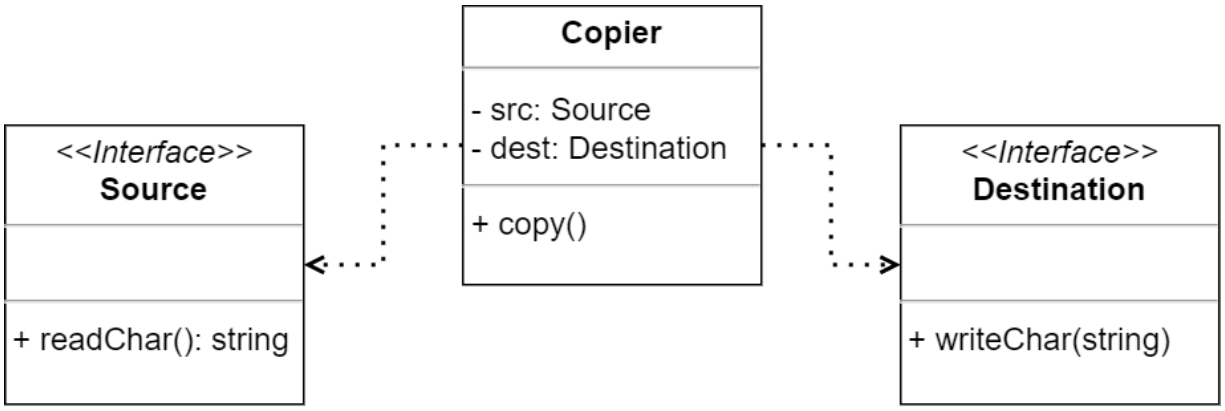


Figure 1: Class Diagram

CSV File Kata – Requirement 1

Rules

1. Use SOLID principles.
2. Use provided interfaces and classes.

You are given the following code snippets (these files cannot be changed by you):

```
export interface FileWriter {
    writeLine(fileName: string, line: string): void;
}

export class Customer {
    private _name: string = "";
    private _contactNumber: string = "";

    constructor(name: string, contactNumber: string) {
        this._name = name;
        this._contactNumber = contactNumber;
    }

    public get name() {
        return this._name;
    }
    public set name(name: string) {
        this._name = name;
    }

    public get contactNumber() {
        return this._contactNumber;
    }
    public set contactNumber(contactNumber: string) {
        this._contactNumber = contactNumber;
    }

}
```

The Kata

Your team has come to you with an urgent requirement. They need to dump `Customer` objects, as per the `Customer` class, to disk in CSV format for a nightly job that imports the data into the CRM system for sales. You do not need to worry about writing headers as the CRM system has NO need for headers. You do NOT need to worry about catering for commas in the data, or quoting the content. You do NOT need to worry about checking for existing files or data as the customers will always be dumped to a clean folder. You MUST NOT write to actual disk, use the `FileWriter` interface provided and use a test double when building your unit tests.

Hint

You should end up with a single public method after completing this requirement. Use the one-to-many green bar pattern to move from handling a single customer to a collection of customers mutating your single public method as you progress through the stages of the pattern. To assist you a UML diagram of how the various parts are related is given below.

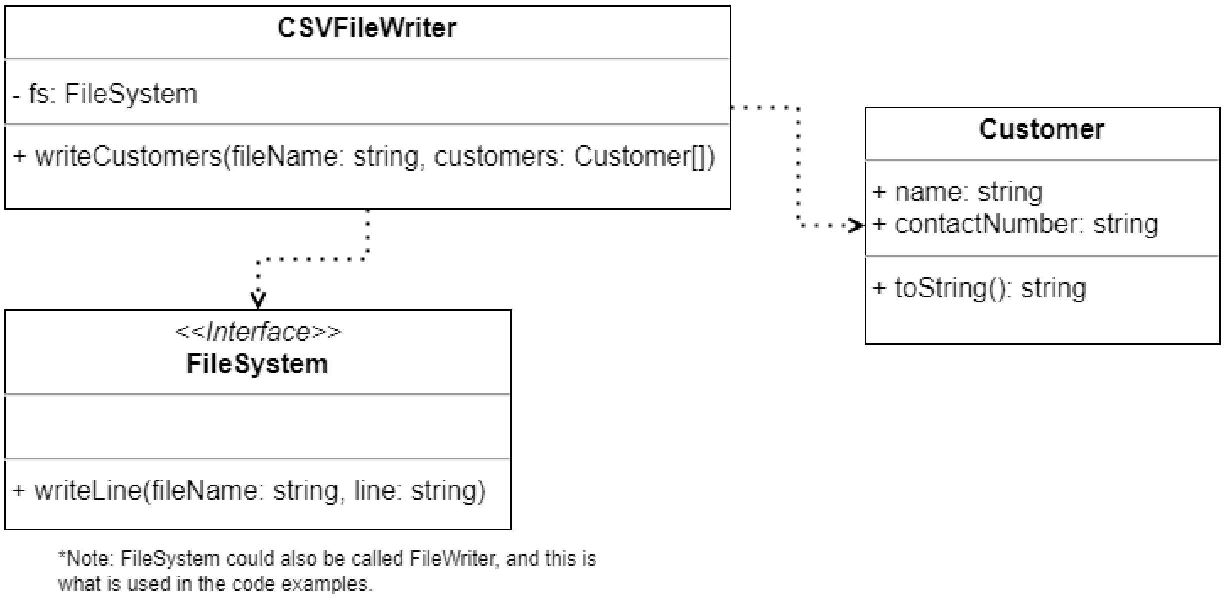

CSVFileWriter
- fs: FileSystem
+ writeCustomers(fileName: string, customers: Customer[])
Customer
+ name: string
+ contactNumber: string
+ toString(): string
<<Interface>>
FileSystem
+ writeLine(fileName: string, line: string)
*Note: FileSystem could also be called FileWriter, and this is what is used in the code examples.

Figure 8: UML Diagram of how the components

Note:

This is part 1 of a five-part Kata, but please *do not* look ahead. The idea is to simulate the real world a little more in that requirement 2 would be something that is asked for after going live with requirement 1, so we are trying to simulate not knowing what requirements will be asked for in the future.

CSV File Kata – Requirement 2

Rules

1. Use SOLID principles.
2. Use provided interfaces and classes.

The Kata

The system admin team is using your CSV generator but are having issues with importing a large single CSV file. The nightly job is hanging and someone needs to get up at 2 am to restart it. To assist in resolving this issue they have requested that the CSV generator is amended to write out CSV files in batches of 10. E.g. if the array of customer contains 12 records then two files will be written. The first with 10 lines and the second with 2.

Hint

When amending the code to allow for batching, pay close attention to the open/close principle. Avoid using default arguments in methods and think how to best to EXTEND the class. Avoid changing any existing public methods, instead, extend by creating an overload or another new method to handle the requirement.

CSV File Kata – Requirement 3

Rules

1. Use SOLID principles.
2. Use provided interfaces and classes.

The Kata

The system admin team have resolved the issue of importing large files and now need the CSV generator to be amended to make batches of 15,000 for the nightly job.

Hint

You need to ensure the system can continue to support batches of 10 in addition to batches of 15,000.

CSV File Kata – Requirement 4

Rules

1. Use SOLID principles.
2. Use provided interfaces and classes.

The Kata

The Sales team has come to you and asked that the CSV generator is amended to remove duplicate entries from the files created for import as it is causing havoc with their CRM system. A duplicate is defined as the record having the same Name.

Hints

This is the 3rd time you've had to modify the same section of your code because of a new requirement, consider generalizing now if you haven't already.

Specifically look to:

- Extract responsibilities into their own class
  - This is the S in SOLID
    - Single Responsibility
- Extract interfaces for your public methods
  - This is the I in SOLID
    - Interface Segregation
- Ensure your interfaces are as granular as possible
  - This is the L in SOLID
    - Liskov Substitution
- Inject the functionality extracted into the CSV writer class via the constructor
  - This is the D in SOLID
    - Dependency Inversion

CSV File Kata – Requirement 5

Rules

1. Use SOLID principles.
2. Use provided interfaces and classes.

The Kata

The time is 16:45. (*Pretend it is*)

The system admin team has contacted you again to say there is an issue importing the CSV files into the nightly job.

To help address this newest issue, they would like two sets of files to be written:

1. The current CSV files as generated in batches of 15,000 and a debug set.
2. The debug set shall have all data, even duplicates, in batches of 20.

They need this fix before you go home at 5. (*You have 15 minutes to finish this requirement*)

Note:

You can create new classes. You may NOT modify your existing classes in any way.

You have 15 minutes. If you fail to meet this requirement in that timeframe, delete all your code and start over.

Bonus Katas

Conway's Game of Life Kata

The Kata

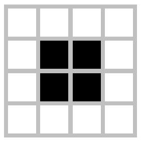
The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, alive or dead. Every cell interacts with its eight neighbors, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

1. Any live cell with fewer than two live neighbours dies as if caused by underpopulation.
2. Any live cell with two or three live neighbor's lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by overcrowding.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

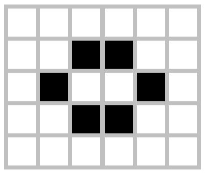
The initial pattern constitutes the seed of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed—births and deaths occur simultaneously, and the discrete moment at which this happens is sometimes called a tick (in other words, each generation is a pure function of the preceding one). The rules continue to be applied repeatedly to create further generations.

Still Patterns

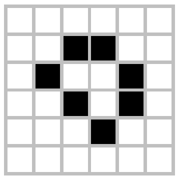
Block



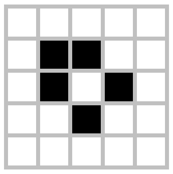
Beehive



Loaf

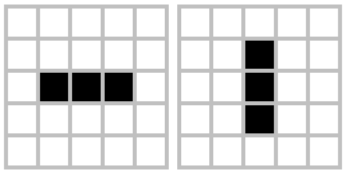


Boat

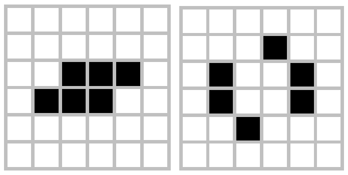


Oscillating Pattern

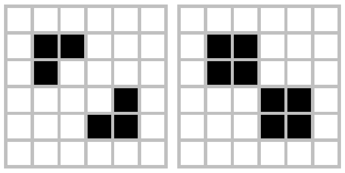
Blinker



Toad



Beacon



Roman Numerals Kata

The Kata

The Romans wrote numbers using letters - I, V, X, L, C, D, M, which represented the following values:

Numeral	Value
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

Table 1: Roman numeral values

The Kata says you should write a function to convert from Arabic Numerals to Roman Numerals. E.g. 2 → II

There is no need to be able to convert numbers larger than about 3999. (The Romans themselves didn't tend to go any higher)

Note that you can't write numerals like "IM" for 999. Wikipedia says: Modern Roman numerals ... are written by expressing each digit separately starting with the leftmost digit and skipping any digit with a value of zero. To see this in practice, consider the ... example of 1990. In Roman numerals, 1990 is rendered: 1000=M, 900=CM, 90=XC; resulting in MCMXC. 2008 is written as 2000=MM, 8=VIII; or MMVIII.

Bonus

Write a function to convert in the other direction, i.e. numeral to digit

Clues

- Can you make the code beautiful and highly readable?
- Does it help to break out lots of small named functions from the main function, or is it better to keep it all in one function?
- If you don't know an algorithm to do this already, can you derive one using strict TDD?
- Does the order you take the tests in affect the final design of your algorithm?
- Would it be better to work out an algorithm first before starting with TDD?
- If you do know an algorithm already, can you implement it using strict TDD?
- Can you think of another algorithm?
- What are the best data structures for storing all the numeral letters? (I, V, D, M etc)
- What is the best way to verify your tests are correct?

Mars Rover

The Kata

Develop an API that moves a rover around a grid on Mars.

- You are given the initial starting point(x,y) of a rover and the direction (N,S,E,W) it is facing.
- The rover receives a character array of commands
    - Implement commands that:
        - Move the rover forward(f)
        - Move the rover backward(b)
        - Turn the rover left(l)
        - Turn the rover right(r)
- Implement wrapping from one edge of the grid to another (Planets are spheres after all)

Hint

Your constructor should look like **MarsRover(location, direction, grid)**
E.g var rover = new MarsRover([0,0],'e',[50,50]);

Example

The rover is on a 100x100 grid at location (0, 0) and facing SOUTH. The rover is given the commands "fflff" and should end up at (2,2).

Bonus

Implement obstacle detection before each move to a new square. If a given sequence of commands encounters an obstacle the rover moves up to the last possible point and reports the obstacle. You will need to amend your code to take in an array of obstacles.

Ten Pin Bowling Kata

The Kata

Create a program, which, given a valid sequence of rolls for one line of American Ten-Pin Bowling, produces the total score for the game. Here are some things that the program will not do, for the purposes of the kata:

- It does not need to check for valid rolls.
- It does not need to check for a correct number of rolls and frames.
- It does not need to provide scores for intermediate frames.

We can briefly summarize the scoring for this form of bowling:

- Each game, or "line" of bowling, includes ten turns, or "frames" for the bowler.
- In each frame, the bowler gets up to two tries to knock down all the pins.
- If in two tries, the bowler fails to knock them all down, their score for that frame is the total number of pins knocked down in their two tries.
- If in two tries the bowler knocks them all down, this is called a "spare" and their score for the frame is ten plus the number of pins knocked down on their next throw (in their next turn).
- If on the bowler's first try in the frame they knock down all the pins, this is called a "strike". Their turn is over, and their score for the frame is ten plus the simple total of the pins knocked down in their next two rolls.
- If the bowler gets a spare or strike in the last (tenth) frame, they get to throw one or two more bonus balls, respectively. These bonus throws are taken as part of the same turn. If the bonus throws knocks down all the pins, the process does not repeat: the bonus throws are only used to calculate the score of the final frame.
- The game score is the total of all frame scores.

More info on the rules at http://www.topendsports.com/sport/tenpin/scoring.htm

Clues

What makes this game interesting to score is the look ahead in the scoring for a strike and spare. At the time, we throw a strike or spare, we cannot calculate the frame score: we have to wait for one or two frames to find out what the bonus is. Some Suggested Test Cases are:

(When scoring "X" indicates a strike, "/" indicates a spare, "-" indicates a miss)

- "XXXXXXXXXXXX" (12 rolls: 12 strikes) = 10+10+10 + 10+10+10 + 10+10+10 + 10+10+10 + 10+10+10 + 10+10+10 + 10+10+10 + 10+10+10 + 10+10+10 + 10+10+10 = 300
- "9-9-9-9-9-9-9-9-9-9-" (20 rolls: 10 pairs of 9 and miss) = 9 + 9 + 9 + 9 + 9 + 9 + 9 + 9 + 9 + 9 = 90
- "5/5/5/5/5/5/5/5/5/5/5" (21 rolls: 10 pairs of 5 and spare, with a final 5) = 10+5 + 10+5 + 10+5 + 10+5 + 10+5 + 10+5 + 10+5 + 10+5 + 10+5 + 10+5 = 150 10+5 + 10+5 + 10+5 + 10+5 + 10+5 + 10+5 + 10+5 = 150

Document References

- *Test Driven Development By Example*, Kent Beck, Addison-Wesley, 2003
  - Chapter 28: Green Bar Patterns
  - Appendix 1: Influence Diagrams
- *Test Doubles - Fakes, Mocks and Stubs,* Michal Lipski 2017, Accessed 02/11/2022 <https://dev.to/milipski/test-doubles---fakes-mocks-and-stubs>
- *Agile Principles, Patterns, and Practices in* C#, Robert C. Martin & Micah Martin, Prentice Hall 2007:
  - Chapters 8 to 12, pp115 to 175
- *Pablo's SOLID Software Development* – free PDF collection of blog posts from LosTechies.com – Accessed from http://lostechies.com/wp-content/uploads/2011/03/pablos_solid_ebook.pdf (last checked 02 Nov 2022)
- *S.O.L.I.D. Software Development, One Step at a Time* by Derick Bailey http://www.codemag.com/article/1001061 (long form article - 13000 words)
- *Adaptive Code: Agile coding with design patterns and SOLID principles 2nd Edition,* Gary McLean Hall, Microsoft Press, 2017
  - Chapters 7 to 11, pp215 to 343
- *http://xunitpatterns.com/* - this is a wiki that was a working area for Meszaros when writing his *xUnit Test Patterns* book. It contains a wealth of information available without having to buy the book (although we do recommend it!).