# Design Approach for Checkout Kata

This document looks at the decisions taken in developing the Checkout Kata, it also looks at the reasoning into the solution setup, patterns, and coding style.

## Technology Used

The provided solution is build using Visual Studio 2019 and is made up of:

* C# .NET Core 3.1 Class Library
* C# .NET Core Test Library

Third Party Libraries:

* NUnit
* Moq
* FluentAssertions

## Approach Taken

The general approach taken was to follow a fairly strict TDD approach. This allowed me to create a solution that meets the requirements of the Kata instructions with a simple and precise solution, with writing just enough code to meet the AC.

As I progressed through the requirements, I have taken opportunities to refactor code, extracting and moving ownership of certain needs into more defined classes. This can be seen between commits 2 & 3, when the PricingEngine was introduced. The PricingEngine could have been added earlier but seemed overkill at the time and was only added when the pricing calculation requirements got more complex. This leans towards a clean and ‘keep is simple’ approach which prevents time being spend on unnecessary requirements and therefore unnecessary code.

### Patterns & Clean Coding

The main SOLID principles adopted are:

* SRP – in the separation of concern into classes, each function within the app whether that’s products, pricing or promotions etc we have separate classes for each.
* ISP – I’ve developed a implementation of each requirement, i.e. ProductDemoService, which implements IProductService – If the requirement changed and we wanted to load products from somewhere else then we could implement that interface and swap out the implementation.

The remaining Solid principles I have not been used.

The Promotion implementation of the requirement provides a simple example of a strategy pattern, which wraps/ hides the decision of the which promotion is applicable. This means the calling code is dumb to the actual solution. Also allowing the code to be easily extended – if you want a new promotion then implement IPromotion and add it to the provider and you’re done.

### Testing

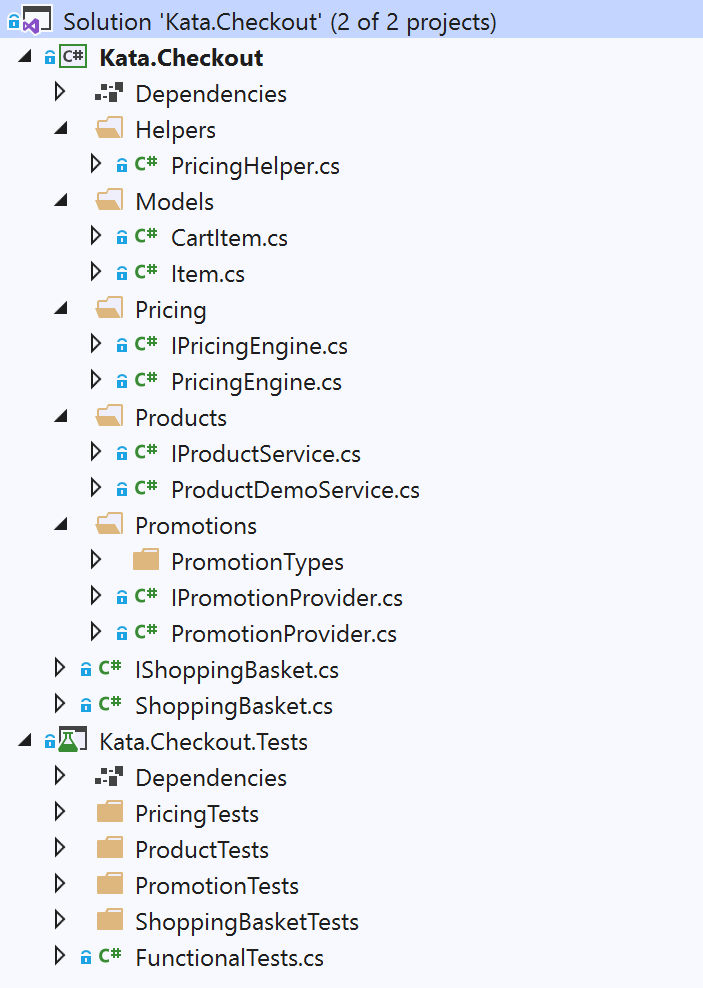
The Unit Testing structure is standard, each folder relates to the structure of the actual implemented classes. This allows easy navigation. The addition of FunctionTests.cs at the root, and although not true functional/ e2e testing it contains four tests, one for each AC in the Kata requirements. These tests contain the real implementations of each class (no mocking) this ensures full AC are met. In a larger application the functional testing would be separated out into another project.

## Software Architectures Decisions

I was undecided if I should try and implement a more complex architecture pattern DDD or CQRS for the Checkout Kata. However, I came to an early decision that this would be overkill for the given task. The Kata instructions does not provide any understanding of what the future roadmap for the product would likely be so implementing a complex architecture would be the incorrect decision in my opinion.

I decided against splitting the responsibility into separate class libraries i.e. Pricing Library, Promotion Library, Product/ Catalog Library etc. In larger applications these could easily be separate API’s, however with the current requirements I do not feels it is necessary to separate into libraries and if anything, right now would make readability harder.

## Final Solution Structure



The overall solution is simple but that is because the requirements are simple. Could we easily convert the current solution into DDD, Onion or CQRS? yes, should we? It depends.