codemanship

Test-Driven Development

Exercises & Links

TDD
Exercise #1

Roman Numerals Converter

Test-driven some code that converts integers from 1 to 3999 into their equivalent roman numerals

Videos

https://www.youtube.com/watch?v=vBJM5pzBfhY

Good Habits for TDD
Exercise #2

The Mars Rover Kata (simplified)

Test-drive code to drive a rover over the idealised surface of Mars, represented as a 2-D grid of x, y coordinates (e.g., [4, 7]).

The rover can be "dropped" on to the grid at a specific set of coordinates, facing in one of four directions: North, East, South or West.

Sequences of instructions are sent to the rover telling it where to go. This sequence is a string of characters, each representing a single instruction:

- R = turn right
- L = turn left
- F = move on square forward
- B = move one square back

The rover ignores invalid instructions.

Links

The complete Mars Rover TDD kata is described at https://kata-log.rocks/mars-rover-kata

Videos

https://www.youtube.com/watch?v=oF5jTHSV28s

What Should We Test?

Exercise #3

Community Movie Library

Members of a local community decide to create a library of the DVDs they own, allowing members to donate their copies to the library and borrow titles.

The library keeps a catalogue of the movies available, which will have one or more donated copies to borrow.

Members can find out if a movie is in the catalogue and is currently available to borrow.

When no copies of a movie in the catalogue are available, members can join a waiting list to be the next to borrow it as soon as a copy has been returned.

A movie with a waiting list can only be borrowed by the next member on the list.

When members donate movies to the library, they earn Reward Points.

- 5 points for a copy of a movie already in the catalogue
- 10 points for a copy of a movie not yet in the catalogue

Members can spend their reward points to "jump the queue" on a movie's waiting list: 5 points for every place in the queue they jump.

How to tackle this exercise:

1. *Identify the user actions we need to test, and make a Test List for each of them to drive your TDD*
2. *Don't forget to apply the disciplines of TDD and refactoring that we've learned so far!*

Videos

https://www.youtube.com/watch?v=Ji2KKShNKic

Duplication & The Rule of Three

Exercise #4

Conway's Game of Life

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, live or dead, (or populated and unpopulated, respectively). Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

- Any live cell with fewer than two live neighbours dies, as if by underpopulation.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any live cell with more than three live neighbours dies, as if by overpopulation.
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

These rules, which compare the behavior of the automaton to real life, can be condensed into the following:

- Any live cell with two or three live neighbours survives.
- Any dead cell with three live neighbours becomes a live cell.
- All other live cells die in the next generation. Similarly, all other dead cells stay dead.

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*.

Test-driven an implementation of Conway's Game of Life that uses a grid of true/false values, randomly initialised.

Videos

https://www.youtube.com/watch?v=f6KHs4aMPpU

Scaling TDD with Test Doubles
Exercise #5

Jason's Guitar Shack

Jason's Guitar Shack is in trouble! We're losing sales because we keep *running out of stock* of our most popular products.

We need you to design and develop a simple system that will alert the manager when stock of a product has fallen to a level where, unless we ordered more right away, we would sell out before replacement stock could arrive.

We have two systems you can connect to that could provide useful data about products and their historical sales.

The warehouse system contains product information – including the manufacturer's lead time - and current stock levels. The sales system has every sale of a product. Test versions of these systems have created especially for you to work with. They contain details of 4 products, and their sales history through 2018-2019.

Your system will be triggered on every new sale – *before* the sale has been processed by the warehouse (so stock levels will not have been adjusted at this point) – and will calculate how much stock we will have remaining afterwards, and compare that to a minimum restock threshold – also calculated by your system - for that product at that time (sales change throughout the year – e.g., Xmas is very busy compared to August).

The minimum restock threshold for a product is the number of units we would expect to sell within the time it would take to get more stock from the manufacturer.

Test-drive a web service that accepts sales notifications containing the following information:

- Date and time of sale
- Product ID
- Quantity sold

And sends an alert to the manager if this sale now means that more stock must be ordered immediately.

Use stubs and mocks to drive your design from the outside-in, creating placeholders for key parts of the end-to-end design. Write at least one system tests that calls your web services and connects to the test microservices linked to below.

Warehouse API

https://6hr1390c1j.execute-api.us-east-2.amazonaws.com/default/product?id=811

Sales API

https://gjtvhjg8e9.execute-api.us-east-2.amazonaws.com/default/sales?productId=811&startDate=7%2F17%2F2019&endDate=7%2F27%2F2019

Videos

Inside-Out TDD

https://www.youtube.com/watch?v=FerGg9E3p1w

Outside-In TDD

https://www.youtube.com/watch?v=p0UooferVbE

Scaling TDD with Test Doubles

https://www.youtube.com/watch?v=L2Gtln3IokQ

Outside-In TDD

Exercise #6

Solve A Problem With Software II

In a test-driven way, implement software that passes your customer's tests.

How to tackle this exercise:

1. *Work one feature, one scenario and one example at a time*
2. *Create a test fixture for each scenario. Write a unit test for each unique outcome of each example. Copy and paste the data from your example directly into the automated test.*
3. *Name your automated tests to easily identify which feature, which scenario and which outcome they are testing.*
4. *If a scenario has multiple examples, consider using a parameterized test.*
5. *Work outside-in, stubbing or mocking any dependencies you wish to make separate concerns (e.g., a database, or a web service, or another component of your system you wish to have its own tests later)*
6. *When a layer of your design is passing its tests, moving inwards to TDD the next layer (the parts you stubbed and mocked), and keep moving inwards until you have a working end-to-end core implementation that excludes any external dependencies.*
7. *Refactor when your code passes the tests but doesn't conform to the rules of Simple Design, Modular design or Conceptual Correlation. Keep extracted elements (e.g., helper methods or classes) hidden from the test code where possible.*
8. *Use version control to ensure you can easily get back to a working version if you mess up.*
9. *DON'T FORGET WHO YOUR CUSTOMER IS! Any questions about requirements should be directed to them.*

Links

Videos

Inside-Out TDD

https://www.youtube.com/watch?v=LTFboEgQjOE

Outside-In TDD

https://www.youtube.com/watch?v=TzPQ4jJ8NBs