SIEMENS ASSIGNMENT 1 REPORT

✨ *AMIR KEDIS* ✨

Introduction

First assignment is focused on testing a "client-server set configuration" scenario. The task is to design how you would test this feature and implement, in Python, a program to automate the generation of all possible combination and their expected results.

In this report, I will explain the approach, assumptions, implementation, input and output formats, test cases, validation and results of my solution.

Approach / System Design

Introduction and problem statement

Not having the implementation details, I will assume that we have to test them as a black box. In some sort of Exhaustion testing that covers all possible combinations of the input parameters and their expected results.

Which I think is fair to assume, as the second part of the task is to implement a program to automate this exact process.

Assuming we had the implementation details, I would have used Unit Testing to test the individual components and their interactions. maybe testing

more aspects like the validity of the packets transferred between the client and the server. the times and delays of the responses, etc.

For the implementation, I will follow something like *top-down* approach, I will implement *stubs* for server client and the main program that will generate the combinations and test them.

Generation of test cases

Based that all server options are boolean, It was intuitive to think of all possible combinations as a truth table. Just the difference here might have 1 of 3 values (True, False, None). and for every server-option we have two desired inputs one for the master-client and one for the slave-client.

Truth Table for $(p \rightarrow q) \vee (q \rightarrow p)$				
p	q	$p \rightarrow q$	$q \rightarrow p$	$(p \rightarrow q) \vee (q \rightarrow p)$
T	T	T	T	T
T	F	F	T	T
F	T	T	F	T
F	F	T	T	T

So the number of test-cases will be calculated as follows:

$$3^{2*ServerOptions}$$

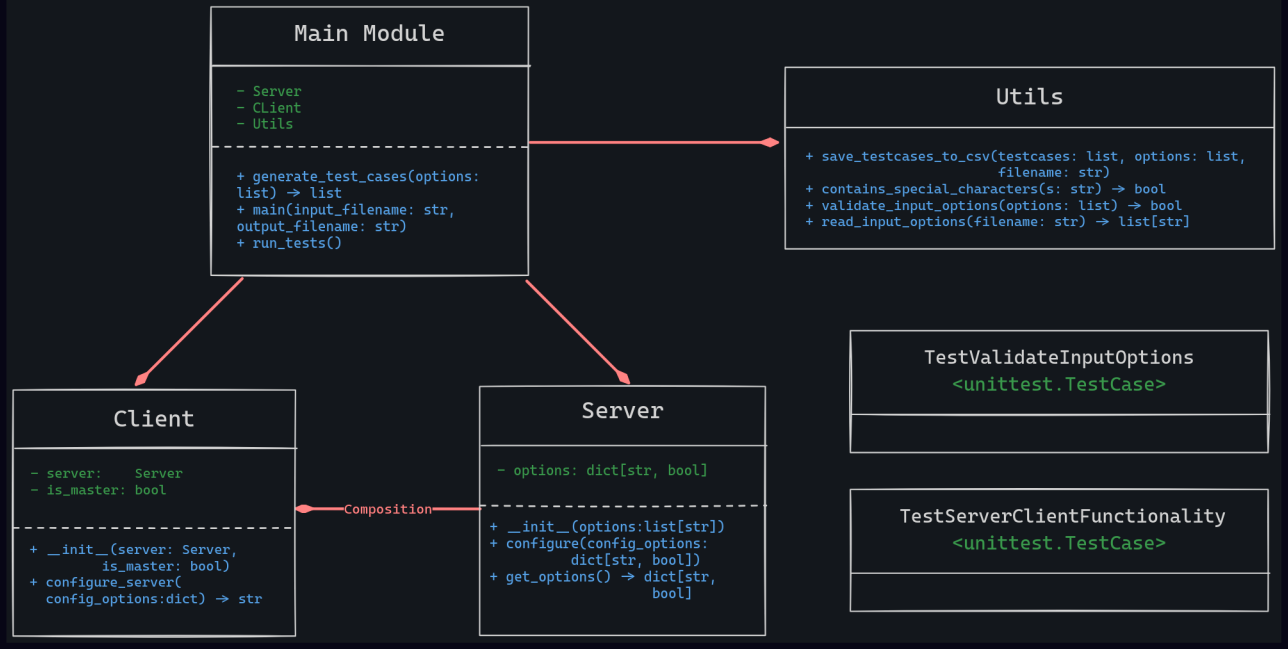
having 2 server options will result in 81 test-cases.
3 options will result in 729 test-cases. and so on.

Implementation

For the system design the system consists of 5 main components:

- server: responsibly of setting of configuration
- client: handles the logic of validating the configuration
- Utils: handles util functions like IO and input validation.
- Main Driver program: generates the test-cases and tests them.
- Unit tests: validates the server and client configuration and the input options handling logic and input errors.

> Note
> it wasn't clear how much Unit testing is required or if it is required at all. So I will assume that it is required and I will implement some important tests.



Additionally, there are 2 unit tests classes that validates the server and client configuration. and one for testing the input options handling logic and input errors.

> **Note**
> for generating the input combination it was really suitable here to use the `iterools.product` function to generate the combinations.

File structure

```
.
├── docs
│   ├── report.md
│   └── sample_output.csv
├── input.txt
├── output.csv
├── README.md
├── requirements.txt
└── src
    ├── client.py
    ├── __init__.py
    ├── main.py
    ├── server.py
    ├── tests
    │   ├── inputs
    │   │   ├── duplicate_test.txt
    │   │   ├── no_test.txt
    │   │   ├── one_test.txt
    │   │   ├── special_chars_test.txt
    │   │   ├── three_test.txt
    │   │   └── two_test.txt
    │   ├── outputs
    │   │   ├── one_test_out.csv
    │   │   ├── three_test_out.csv
    │   │   └── two_test_out.csv
```

```
    │   ├── test_input_validation.py
    │   └── test_server_client.py
    └── utils.py

8 directories, 22 files
```

Input and Output Formats

The main driver program have 2 modes of operation:

- First one runs all the system test cases found in the directory `tests/inputs` and puts the output in `tests/outputs`
- Sample output:
  - std

```
$ python src/main.py test

======================================================================
            RUNNING the program with different inputs
======================================================================

Running one_test.txt
output file created successfully

Running duplicate_test.txt
Invalid Input.

Running two_test.txt
output file created successfully

Running three_test.txt
output file created successfully

Running special_chars_test.txt
Invalid Input.

Running no_test.txt
Invalid Input.
```

  - and output files generated in `test/outputs`
- Second mode is normal mode, where the user can input the server options and the program will output the expected results.
  - sample input:

```
BufferData
TimeOut
```

- sample output:

```
TestCase ID,Master Option For BufferData,Master Option For TimeOut,Client Option For BufferData,Client Option For TimeOut,Valid TC,Expected BufferData,Expected TimeOut
1,True,True,True,True,YES,True,True
2,True,True,True,False,NO,NA,NA
3,True,True,True,NA,YES,True,True
4,True,True,False,True,NO,NA,NA
5,True,True,False,False,NO,NA,NA
6,True,True,False,NA,NO,NA,NA
....
```

Assumptions

- When generating the test cases: I assumed that if the test case is invalid, the expected output will be `NA` for all the fields. this inferred from this test case in the document

```
2 TRUE FALSE NA TRUE NO NA NA
```

- Also When generating the test cases: I assumed that if master is None it is equivalent to True, as it is the default. so a test cases like these will be valid:

```
18 NA NA NA TRUE YES TRUE TRUE
19 NA NA TRUE TRUE YES TRUE TRUE
```

Test cases

- Function unit tests
  - Input Validation test-cases in `TestValidateInputOptions` class
    - no input
    - duplicate input
    - special chars input
    - valid input with 1 option
    - valid input with 2 options
    - valid input with 3 options
  - Server Client configure test-cases in `TestServerClientFunctionality` class
    - if both none
    - if master true and slave are none
    - if master none and slave are true

- ↗ if master and salve are different
- ↗ if master and slave are the same

★ I also made some sort of manual integration testing at first that runs using `python src/main.py test` and this is the test cases:

1. `duplicate_test.txt`: test for duplicate server options
   - input:

```
BufferData
BufferData
```

   - output:

```
Innvalid Input.
```

2. `no_test.txt`: for invalid logic
   - input: empty file
   - output:

```
Innvalid Input.
```

3. `special_char_test.txt`: for invalid input
   - input: input containing special chars

```
BufferData$
Delay
TimeOut
```

   - output:

```
Innvalid Input.
```

4. `one_test.txt`: testing giving only 1 input
   - input:

```
BufferData
```

   - output:

```
TestCase ID,Master Option For BufferData,Client Option For BufferData,Valid
TC,Expected BufferData
1,True,True,YES,True
2,True,False,NO,NA
3,True,NA,YES,True
4,False,True,NO,NA
```

```
5,False,False,YES,False
6,False,NA,YES,False
7,NA,True,YES,True
8,NA,False,NO,NA
9,NA,NA,YES,True
```

> 🔥 Important
> This testcase can show all of my assumptions in action.

5. `two_test.txt`: testing giving two valid inputs

- input:

```
BufferData
TimeOut
```

- output:

```
TestCase ID,Master Option For BufferData,Master Option For TimeOut,Client Option
For BufferData,Client Option For TimeOut,Valid TC,Expected BufferData,Expected
TimeOut
1,True,True,True,True,YES,True,True
2,True,True,True,False,NO,NA,NA
...
```

6. `three_test.txt`: testing giving 3 valid inputs

- input:

```
BufferData
TimeOut
Delay
```

- output:

```
TestCase ID,Master Option For BufferData,Master Option For TimeOut,Master Option
For Delay,Client Option For BufferData,Client Option For TimeOut,Client Option For
Delay,Valid TC,Expected BufferData,Expected TimeOut,Expected Delay
1,True,True,True,True,True,True,YES,True,True,True
2,True,True,True,True,True,False,NO,NA,NA,NA
3,True,True,True,True,True,NA,YES,True,True,True
4,True,True,True,True,False,True,NO,NA,NA,NA
...
```

> 🔥 Important

All input files and output files mentioned can be found in the `tests/inputs` and `tests/outputs` directories.