Siemens Assessment Report

Presented To

Eng. Mohamed Elsamanoudy

Eng. Noha Gamal

Eng. Ahmed Seif

July 2024

Contents

List of Figures

1. Assignment 1

A program implemented in **Python** to automate the generation of all possible system configuration combinations and their expected results.

1.1. Assumptions

If the master client doesn't configure the system while the client configures it, the client's options are compared with the default server's values instead of the master's unconfigured system options.

Master Options	None	None	None
Server's Default	True	True	True
Client Options	True	None	None
Expected Options	True	True	True

Figure 1: No master config edge case assumption

1.2. Program Input

1.2.1. According to the Assignment

- Server Options.

1.2.2. In the Application

- The input is provided via the `ARGS` variables in the `MakeFile`.
- The first argument presents the output CSV file name, while the rest are the Server Options names.
- In any case where the user enters invalid values (no arguments, no options, duplicate options, special characters), the program will throw a `ValueError` and print the error to the user before exiting.

1.3. Program Output

1.3.1. According to the Assignment

- CSV file in the following format.

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	NA	NA	NA	NA	YES	TRUE	TRUE
2	TRUE	FALSE	NA	TRUE	NO	NA	NA
3	FALSE	TRUE	FALSE	NA	YES	FALSE	TRUE

Figure 2: Assignment 1 CSV File Format

1.3.2. In the Application

- The CSV file is saved after evaluating all the possible options combinations between the master and the client.
- If the CSV file is already opened the program will tell the user to close the file to update it.

1.4. UML Design

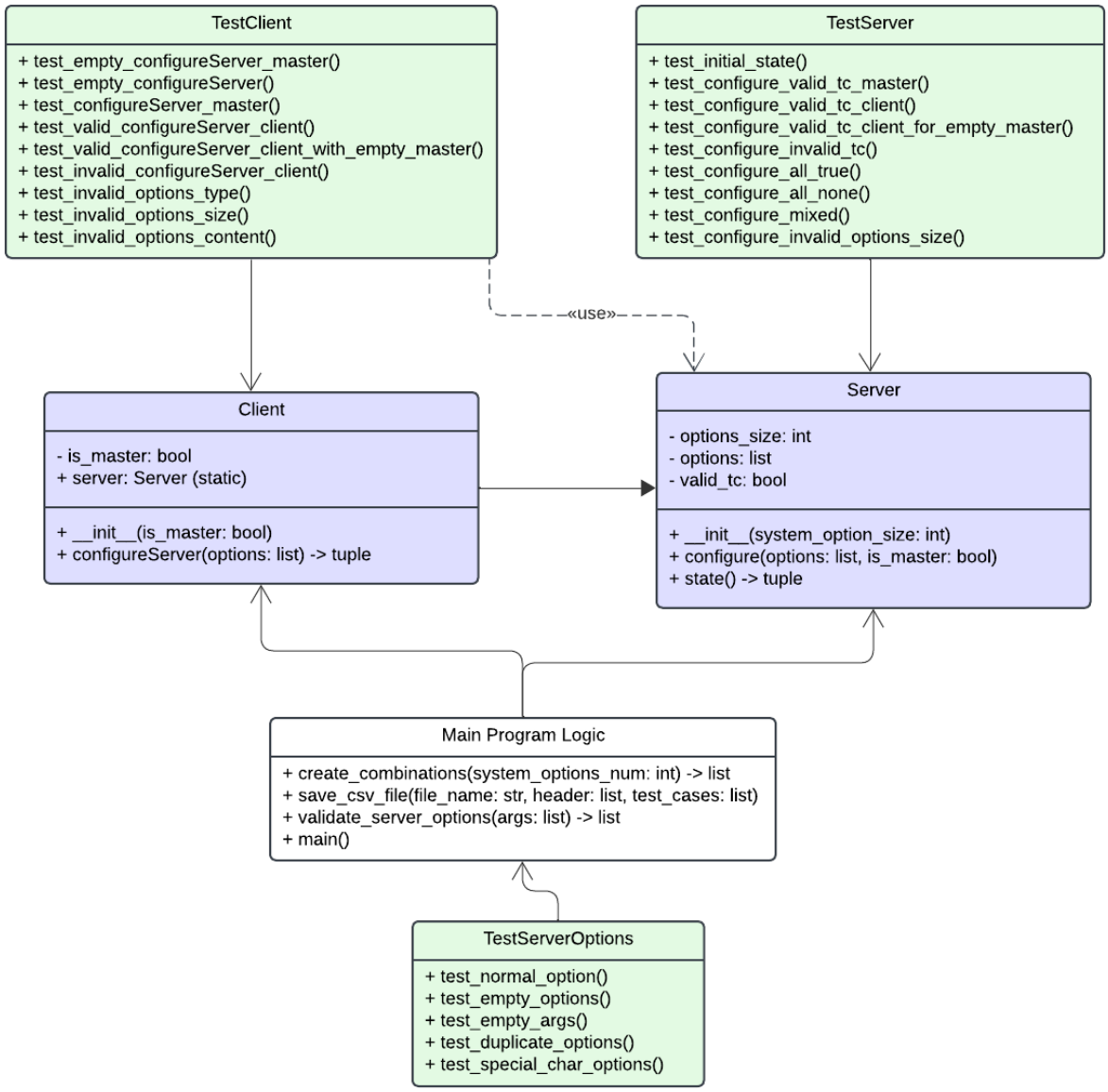

Figure 3: Assignment 1 UML Diagram

1.5. Code Structure

1.5.1. Main Program `app.py`

- The main program validates the user to input from `sys.argv`.
- The program generates all possible system configuration combinations, then evaluate them by creating master and client Class objects to configure the Server.
- The program saves the expected results data to a CSV file.

Functions:

- **create_combinations(system_options_num: int) -> list:** Creates combinations of boolean values and None for a given number of system options.
- **save_csv_file(file_name: str, header: list, test_cases: list) -> None**: Saves test cases to a CSV file, handling potential IOErrors.
- **validate_server_options(args: list) -> list:** Validates the input system options provided by the user in the `sys.argv`.
- **main():** The main function initializes the server and clients, creates test cases, and saves them to a CSV file by calling their respective functions.

1.5.2. Client Class `client.py`

- The `Client` class provides functionality to configure a shared server instance with other clients.

Attributes:

- **is_master (bool)**: A Boolean flag indicating if the client is the master.
- **server (Server)**: A static variable to store the server instance.

Methods:

- **__init__(self, is_master: bool) -> None**: Initializes the client with a master flag.
- **configureServer(self, options: list) -> tuple:** Configures the server with the client's options. It raises exceptions if the options are invalid and returns the server's state after configuration.

1.5.3. Server Class `server.py`

- The `Server` class simulates a configurable server with options and validation states.

Attributes:

- **options_size (int)**: The number of system options given by the user.
- **options (list):** The list of system options, initialized with `True`.
- **valid_tc (bool):** A boolean flag indicating if the system configuration is valid.

Methods:

- **__init__(self, system_option_size: int) -> None**: Initializes the server with a given number of system options.
- **configureServer(self, options: list, is_master: bool) -> None:** Configures the server with a list of options from the client. If the client is the master, it sets the options directly, replacing `None` with the default value `True`. If the client is not the master, it compares the options with the current server options (which reflect the master's configuration) and updates the server's options to `None` if there is a mismatch, as it indicates an invalid test case.
- **state(self) -> tuple:** Returns the current state of the server, which is the test case validation state and the expected system options of this test case.

1.5.4. Test Cases `test Folder`

- The `Unit test` operates using `unittest`
- The test cases ensure the functionality of the `Server` and `Client` classes, and the input validation function in the main program.

Tests:

For more information about the test cases used, check out the TEST.md document.

2. Assignment 2

A program implemented in **C++** to automate the verification of all possible packets' assigned module numbers scenarios and their expected results.

2.1. Assumptions

Since the last module `M` isn't given as a Program input, it is indicated as the largest module number in the given module numbers array.

2.2. Program Input

2.2.1. According to the Assignment

- Number of packets.
- Module number of each packet.

2.2.2. In the Application

- The input is provided via the console.
- In any case where the user enters invalid values (characters, strings, etc), the program will throw an `invalid_argument` error and prompt the user to try again without restarting.
- Any float number is floored to the nearest integer.

2.3. Program Output

2.3.1. According to the Assignment

- CSV file in the following format.

PacketID	ModuleNumber	ValidModule
1	1	Yes
2	3	No
3	4	Yes
4	4	Yes

Figure 4: Assignment 2 CSV File Format

2.3.2. In the Application

- The CSV file is saved by calling the `saveValidation` method of the `PacketValidator` class.
- The CSV file is automatically saved in the `src/output` folder.

2.4. UML Design

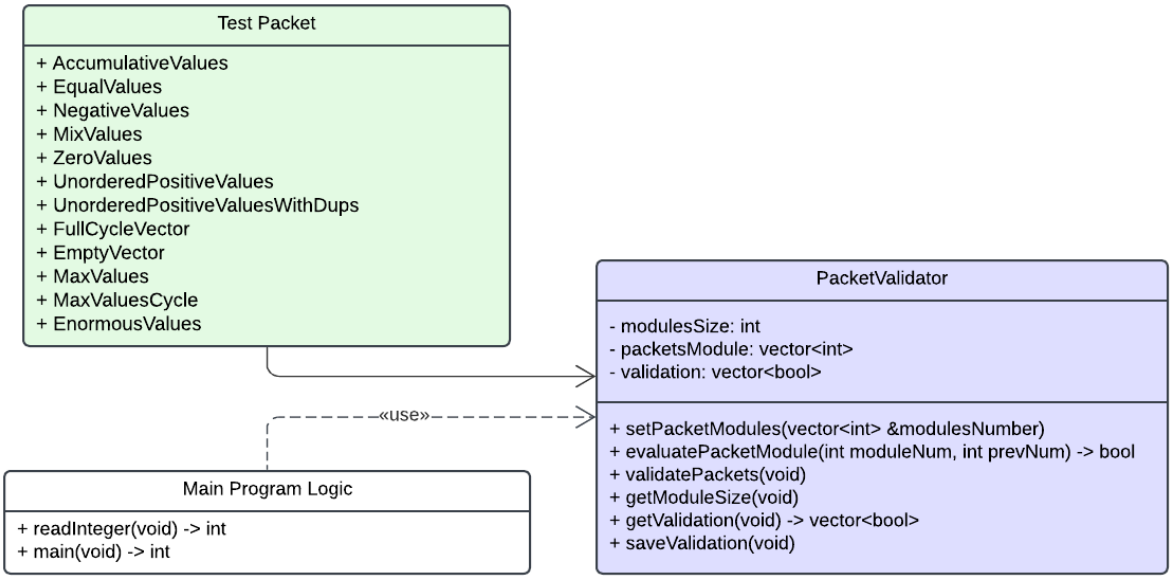

Figure 5: Assignment 2 UML Diagram

2.5. Code Structure

2.5.1. Main Program `main.cpp`

- The main program prompts the user to input the number of packets and their module numbers.
- The program creates an object from the `PacketValidator` class to validate the module numbers and save this data to a CSV file.

Functions:

- **readInteger**: Reads and returns an integer or rounds a float from the console and throws an error if invalid input was given, and then tries again.

2.5.2. PacketValidator Class `packet.h`

- The `PacketValidator` class is designed to validate packets based on their assigned module numbers.
- The `PacketValidator` is designed to save the validation result to a CSV file in the `src/output` folder.

Methods:

- **setPacketModules**: Sets the packet modules for validation and initializes the validation

vector.

- **evaluatePacketModule**: Evaluates if a packet module number is valid based on the previous module number.
- **validatePackets**: Validates all packets based on the packets' assigned module numbers.
- **getModuleSize**: Retrieves the largest module number in the cycle, indicating the last module `M` .
- **getValidation**: Retrieves the validation results of each packet.
- **saveValidation**: Saves the validation results to a CSV file.

2.5.3. Unit Test `test_packet.cpp`

- The `Unit test` operates using `googletest`
- The `Unit test` is intended for testing the `PacketValidator` class.

Tests:

For more information about the test cases used to test the `PacketValidator`, check out the TEST.md document.