Multithreaded-dictionary server

YUNTAO LU 1166487

Problem Context

The goal of this assignment is to design and implement a multi-threaded server that allows concurrent clients to search the meaning(s) of a word, add a new word, and remove an existing word using client-server architecture.

The requirements for the server program include the following:

Server Side

- Explicit use of socket and threads have to be made to allow multiple clients to connect to a (single) multi-threaded server and perform operations concurrently.
- All communication will take place via sockets, and the communication between clients and server is required to be reliable.
- The errors must be properly managed through exception handling.

Client Side

- A Graphical User Interface (GUI) is required for the client
- The user has to be informed of any errors that occur while performing any operations

Functional Requirements

The system should handle different scenarios in GET, CREATE, UPDATE, DELETE operation both in server side and client side.

Client

The input should be validated from the client side and be sent to the server if it's valid. When the results are returned, it should present it in GUI.

	Get	Create	Update	Delete
Success	Present word definition	Success! The word def created	Success! The word def updated	Success! The word deleted

Failure	a. The input invalid b. “The word can not be found in the system”	a. The input is invalid b. The word already exist	a. The input is invalid b. The word does not exist in the dictionary	a. The input invalid b. The word does not exist in the dictionary

Server

The exception should be handled in unexpected circumstances and the result should be returned to the client in json format

	Get	Create	Update	Delete
Success	Return the definition of the value in the dictionary	Add the word-def in the dictionary and return success for the client	Update the word-def in the dictionary and return success for the client	Delete the word from the dictionary and return success
Failure	Word not exist (404 Not Found)	Invalid Request(400 Bad Request) Duplicate Record(409)	Word Not Found in the dictionary(404) Invalid request body(400)	Word not found(404)

System Components

The whole system consists of two independent but interacting module: the server module and the client module

Client

- **ClientResultHandler**
  Generate the result based on the content returned by the server
- **ClientServer**
  - Build connection to the server
  - Validate the input
  - Send the request to the server
  - Get the response from the server
  - Present the result using GUI

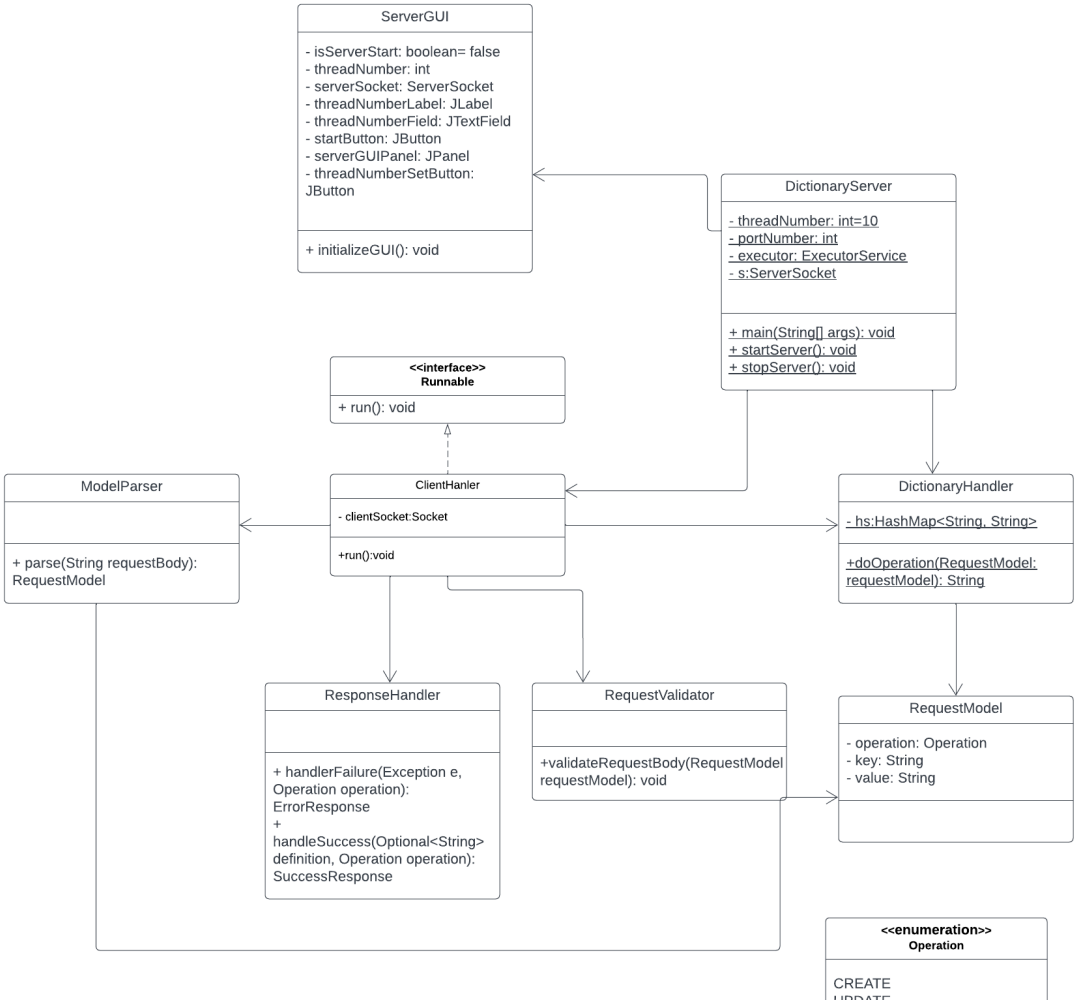
- **ClientGUI**
  - The user interface of the client
  - The action performed for each button

Server

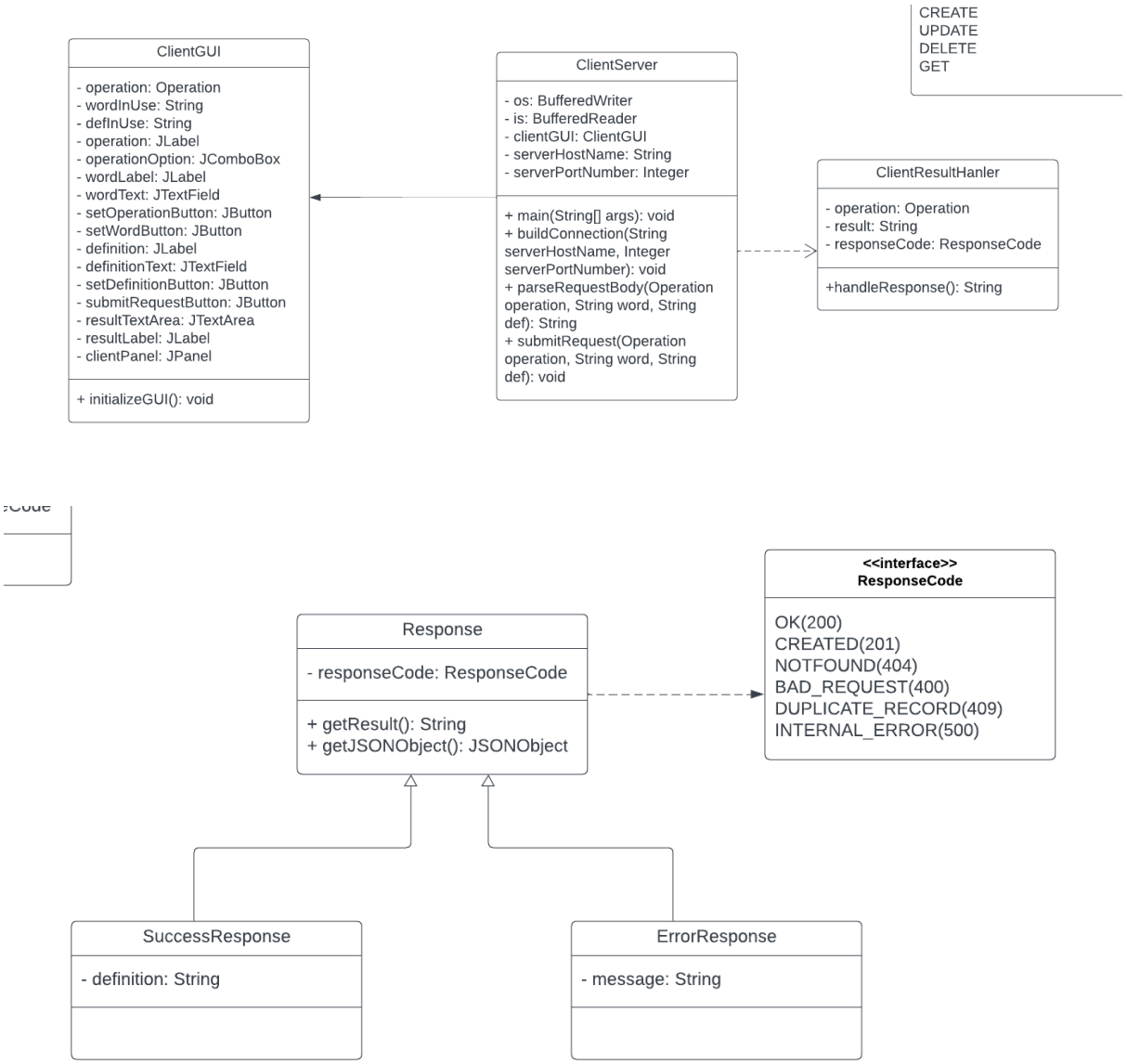
- **DictionaryServer**
  - Read file specified in args
  - Initiate a new thread for GUI
  - Start and stop the server with the specific port
- **ClientHandler**
  - Handle the input and output stream for each connection
  - Process the request and generate the result
- **DictionaryHandler**
  - Doing CRUD operation in the dictionary
- **ModelParser**
  - Transform the JSONObject to RequestModel
- **RequestValidator**
  - Validate the request and throw the exception for the unexpected scenarios

Class diagram

Server

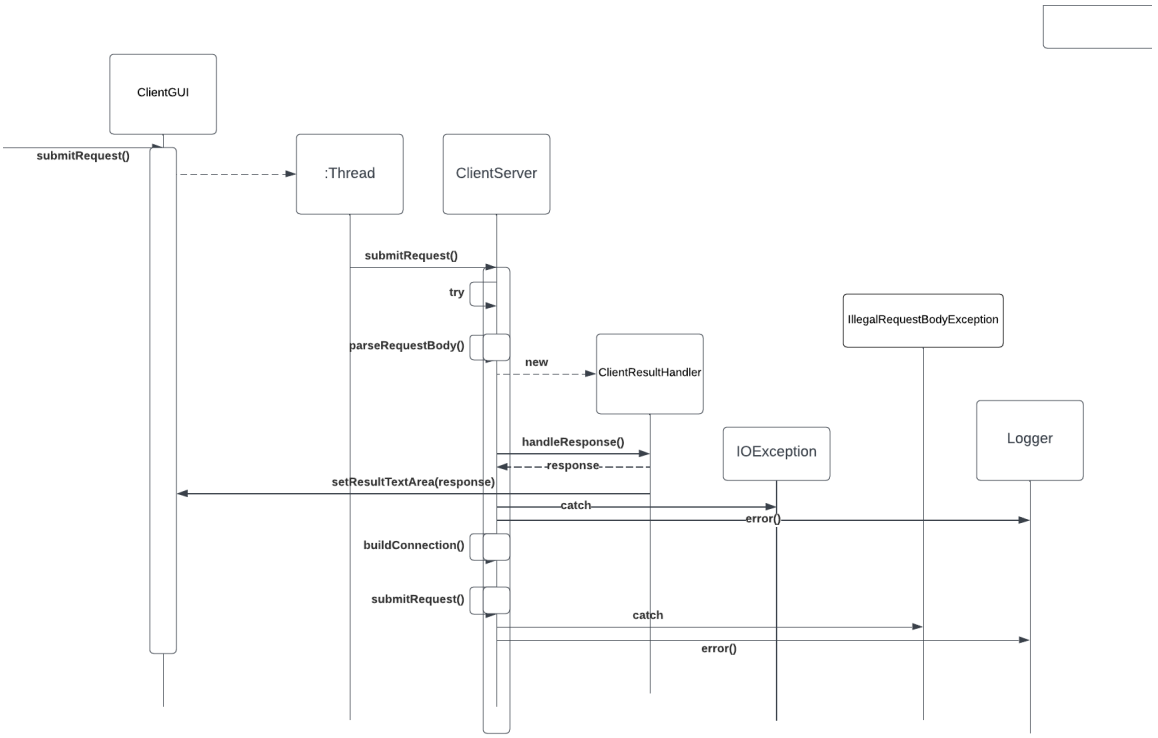


Client

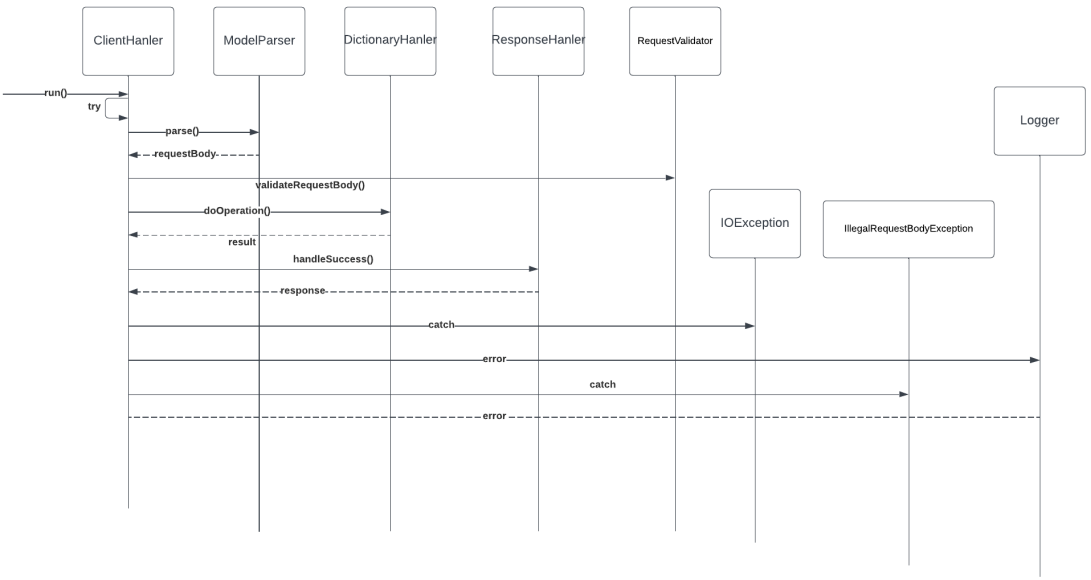

CREATE
UPDATE
DELETE
GET
ClientGUI
- operation: Operation
- wordInUse: String
- defInUse: String
- operation: JLabel
- operationOption: JComboBox
- wordLabel: JLabel
- wordText: JTextField
- setOperationButton: JButton
- setWordButton: JButton
- definition: JLabel
- definitionText: JTextField
- setDefinitionButton: JButton
- submitRequestButton: JButton
- resultTextArea: JTextArea
- resultLabel: JLabel
- clientPanel: JPanel
+ initializeGUI(): void
ClientServer
- os: BufferedWriter
- is: BufferedReader
- clientGUI: ClientGUI
- serverHostName: String
- serverPortNumber: Integer
+ main(String[] args): void
+ buildConnection(String serverHostName, Integer serverPortNumber): void
+ parseRequestBody(Operation operation, String word, String def): String
+ submitRequest(Operation operation, String word, String def): void
ClientResultHanler
- operation: Operation
- result: String
- responseCode: ResponseCode
+handleResponse(): String
<<interface>>
ResponseCode
OK(200)
CREATED(201)
NOTFOUND(404)
BAD_REQUEST(400)
DUPLICATE_RECORD(409)
INTERNAL_ERROR(500)
Response
- responseCode: ResponseCode
+ getResult(): String
+ getJSONObject(): JSONObject
SuccessResponse
- definition: String
ErrorResponse
- message: String

Sequence Diagram

Client



Server



System analysis

Protocols

The protocol we use in the project is TCP protocol. The reason I use it is because it is a connection-oriented communication protocol that provides a reliable flow between two servers. This guarantees the reliability of the system. Also, the dictionary server is not sensitive to time, so the reliability is more important compared with latency.

The message exchange protocol between the client and the server is chosen to be in JSON format. This is because JSON is very light-weight and flexible. This can guarantee efficient communication between the server and the client. It is a very popular data format and many languages can support this data type. This means the client implemented with other languages or framework can still communicate with the server. In conclusion, JSON is a very desirable choice for us.

Architecture

Worker Thread Pool

The architecture we employ for the server side is a worker-thread pool architecture. This can make it possible to reuse threads, so that it can reduce the thread creation and destruction time. On the other hand, the performance can be affected negatively if an improper thread number is set, as the only limited number of threads are available at a time. This means that once all worker threads in the pool are occupied, subsequent requests would have to wait until one of the worker threads is available.

Synchronisation

We allow the DictionaryHandler to handle the operation related to the dictionary. The public methods in it are synchronised. This means we can avoid the race condition scenario. If multiple threads try to write to the dictionary at the same time, the server may not know what to do. In this case, we use synchronise to lock the access to the shared resource each time.

ClientGUI Concurrency

We created a separate thread for clientGUI. Firstly, multithreading can make the system asynchronous and remain responsive while the client server is handling complicated tasks. Also, through modularity, we can break the application into small parts, so that each part can focus on its own job. Through this means, we can improve the operation efficiency. Apart from that, it can save the CPU resource as well.

Excellence

- The input is validated both in server side and client side
- All errors are caught and handled properly and be presented in a human understandable manner through UI.
- The errors and status are logged in the console, and makes it easy to debug
- The system follows the open-closed principle and we reduce coupling and increase cohesion as much as possible.
- The report includes all the necessary components and a full analysis of the design including advantages and disadvantages.

Creativity

- A GUI for the server has been implemented, allowing people to adjust the number of threads based on clients of the system
- The communication exchange between the server and the client has been designed to follow a standardised format. In this way, we can allow other client servers to connect and communicate with our server.
- The operation on the dictionary is implemented using HashMap. This can make our system more scalable with the increasing dictionary volume. It can also reduce duplicate words.

Conclusion

All the requirements have been covered and implemented in the project. Proper choice of protocols and architecture structure has been made and provide reliability and scalability for concurrent client requests. Both server GUI and client GUI are developed to make the system more user friendly.