

Advanced Programming with Java

Fayzullo Bakhtiyorov¹, Pedro Pereira²
, Ma Johann Aniya Luge³, John Lawrence Mendoza⁴
, and Dimo Dimchov⁵

COMP1549: Advanced Programming

University of Greenwich

Old Royal Naval College

United Kingdom

Abstract—This coursework focuses on implementing a networked distributed system for group-based client-server communication. The system can be operated via a CLI interface, and each new member inputs a unique ID and port and provided with an IP address. The first member is automatically assigned the admin role, who monitors the active state of the group and provides member details upon request. The system can handle private and broadcast messages, transfer coordination when the current admin leaves, and ensures uninterrupted communication. It also features printing out messages sent to and by members. The project demonstrates the use of design patterns, JUnit-based testing, fault tolerance, and a Version Control System using Azure DevOps.

Key words —Networked Distributed System, Client-Server Communication, JUnit Testing, Design Patterns

I. INTRODUCTION

The aim of this project is to develop a networked distributed system that allows for effective group-based client-server communication. The system can be either GUI or CLI-based and enables real-time information exchange across different members who have joined the same chat server. It meets the growing demand for smooth ways to communicate that go beyond restrictions. It understands the importance of connections within groups in modern society, that involve different sectors such as business and education.

Networked distributed systems are frequently employed in tonnes of practical situations, such as online banking, social networking, and cloud computing. They strengthen data security, enhance connectivity, and explore cloud-based solutions. We thoroughly evaluated the requirements needed for the application and implemented programming principles and practices to ensure a high performance of the system. Using design patterns visualised using UML diagrams, we were able to conceptualise and well-plan the implementation process. We also conducted extensive testing on all the important features to check their responsiveness.

II. DESIGN/IMPLEMENTATION

VS Code is a fast and open source tool that offers essential language features like built-in terminal, supports version control systems such as Git and code snippets. VS Code is ideal for Java developers who want a quick code editing tool

and a full debugging and testing cycle. It supports many languages, helps start the Java journey without installing a complex IDE, provides wide range of extension tools, team-based collaboration, and cloud services. It offers support for multiple programming languages, making it ideal for Computer Science students.

Azure DevOps is a software development platform that streamlines collaboration. In our group, we used it to manage our code testing and deployment. It offers a Git repository, version control, branching, merging, and pull requests, enabling efficient project management. We chose to divide the tasks where in other members in our group focused on the implementation side while others, manage the report writing. That will reason out the little commits we have in Azure DevOps.

PlantUML is a textual DSL used to create UML diagrams making it a convenient tool for visualising design patterns. Below are the design patterns for this project, in Server and Client implementation:

Design Patterns - Server Implementation

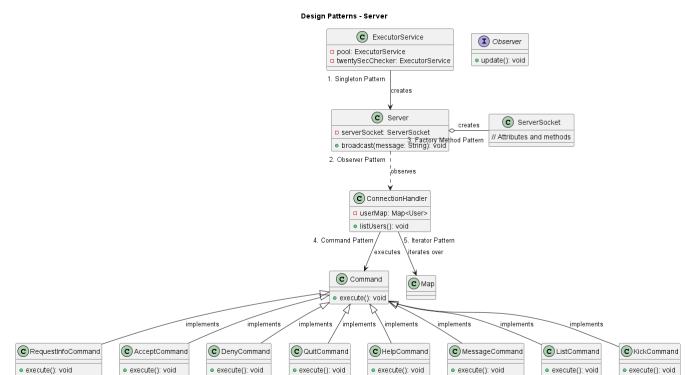


Fig. 1. UML Diagram for Design Patterns in Server Side

Client-server communication uses an architectural pattern influenced by different design patterns. We employed various design patterns in the development of a server implementation, which have been recognised for their success in solving common software design problems. We explore two creation methods and three behavioural method patterns including the Singleton, Observer, Factory Method,

Command, and Iterator Patterns.

The Singleton Pattern is a creational design pattern that ensures a class has only one instance and provides system coordination. [1] The *Executors* utility class is used to create two *ExecutorService* instances, *pool* and *twentySecChecker*, within the *Server* class. The *pool* instance is used to handle member connections, broadcast messages and check member status. While the *twentySecChecker* instance is used to monitor each member in a fixed interval of 20 seconds. To manage its global state, the variables are declared as private final, meaning they cannot be re-assigned after initialization, and are initialized in the *Server* object. Using this pattern, in a single instance we simplified coordination and prevented conflicts in a multi-user environment running concurrently.

Moreover, the Observer Pattern is a one-to-many dependency where one object (the subject) informs its observers about state changes. [2] In corresponding to the **Message Broadcasting** feature, the *Server* class *broadcast* method serves as the subject that iterates over messages being broadcasted to the *ConnectionHandler* objects represent the observers that handles connected members into the server. By implementing decoupling broadcasting logic from connection handling using the Observer Pattern successfully deliver notifications to all connected members about broadcast messages maintaining the server's modularity.

It has also contributed to the functionality of the **State Maintenance** feature in terms of the active state of members in the server. In the *ConnectionHandler* class, The active members are commented out, but its function is to indicate the activeness of the connection for each instance. The observer in this case is the *ConnectionHandler* will act as the observer and the *Server* serves as the subject. The server manages the active connections and notifies the admin instances of the changes in the connection state of the members, updating them accordingly. This pattern enhances the management of connection state facilitating better coordination based on whether the member is active or inactive. [2].

In addition, the Factory methods are creation methods that create objects from concrete classes, with the return type usually declared as either an abstract class or an interface. [4] They encourage loose coupling by deferring object instantiation to subclasses, thereby enhancing object creation flexibility. In our implementation, the code uses encapsulation and abstraction to create a *ServerSocket* in the *Server* class to prevent details from being exposed to external classes. [3] Despite that, we did not have a particular factory method, constructing a *ServerSocket* still provides the same logic and functionality.

As well as, the Command Pattern is a behavioral design pattern that encapsulates requests as objects, allowing parameterization of clients with queues, requests and operations. [7] The *ConnectionHandler* class processes various commands like */requestinfo*, */accept*, */deny*, */quit*,

/help, */msg*, */list*, and */kick*. Each command is executed by a separate method, which is a command object, based on the client's input. To maintain the independency of the commands they are encapsulated in separate methods with their defined functionality. [8] Within the *run()* method in the *ConnectionHandler* class executes the member commands with each respective methods following a consistent interface preventing duplication of logic and for easier management.

This design pattern also contributed to the functionality of **Exiting Mechanisms**. We have introduced two exiting commands to */quit* and */kick*. They are encapsulated within *ConnectionHandler* class, as command objects and part of the message loop. In the instance of a member sending a */quit* command, it initiates the *shutdown()* method. In addition, */kick* command is within the *kickUser()* that overrides to perform removal of a specified member in the server.

In addition, the concept of parsing messages and executing actions based on the command prefix ("/msg"), similar to the Command Pattern to suffice the **Private Messaging** feature. The server acknowledges a private message request when a user sends a message with the format of */msg ;username;* *message;*.

Further, the Iterator Pattern is another behavioral design pattern that allows sequential access to aggregate object elements without revealing the underlying representation.[6] The *listUsers* method in the *ConnectionHandler* class iterates over the *userMap*, which is a collection of member IDs. The map retrieves user data such as ID, nickname, IP address, and port for each user and then sends a message to the client. The *forEach* method is used to iterate over the *userMap* elements, offering convenience to the admin to access each user's information. This approach enhances code simplicity using iteration logic within a single method.

The iteration method helps to fulfil the **Dynamic Coordinator Assignment** feature. When the current admin disconnected in the chat, it iterates the user IDs using *userMap* in Java streams. [5] Then, the *filter* operation ensures that the new admin ID differs from the current admin ID *adminId* and the user associated with the current *ConnectionHandler* instance *userId*. The *findFirst* operation retrieves the first user ID that meets the filter condition, updates the *adminId* variable with the new admin ID (*newAdminId.isPresent()*), and sends a notification to the new admin.

Design Patterns - Client Side

In the context of the Observer Pattern, the *Server* class acts as the subject and the *Client* class acts as an observer. The members join the connection with the server and wait for its broadcasted messages. The server periodically sends notifications to the group members, to inform them of any changes in the system state and provide them with valuable information. [9]

In the implementation in the code, the *Client* class uses

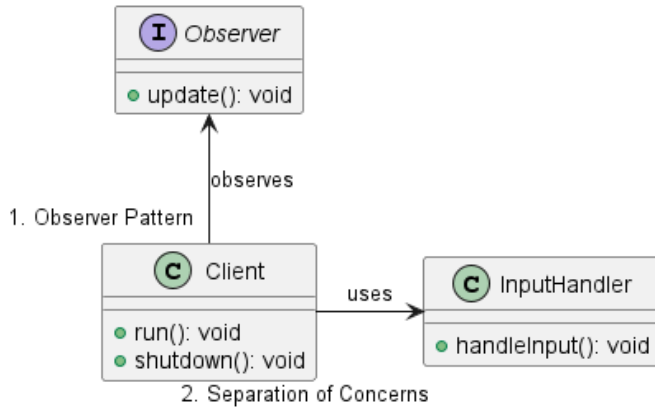


Fig. 2. UML Diagram for Design Patterns in Client Side

the *Runnable* interface to execute concurrently as a separate thread. [10] It includes a continuous loop within its run method that listens for server messages. The loop serves as an observation mechanism, allowing the members to await server notifications. Each time the server sends a message, it triggers an event in the member, (*System.out.println(fromServer)*), printing the message to the console.

The Separation of Concerns Principle is a software architecture principle that divides an application into distinct sections, each addressing a specific concern. [12] In this case, it identifies the difference of responsibilities between *Client* class and *InputHandler* class. The *Client* class manages the client-side socket connection to the server *Socket*, taking charge of creating the connection, setting up input/output streams, and overseeing communication protocols such as *BufferedReader* and *PrintWriter*. The *run* method is defined to listen for server messages and broadcast them to the members, either as a group or privately. [11]

In contrast, the *InputHandler* class handles user input from the console and sends it to the server. It encapsulates the logic for reading user input *BufferedReader* and transmitting it via *PrintWriter* for better interaction functionality. [13]

Our code complies with this principle to keep functionalities separate within their respective classes. This helps us avoid tightly coupled code, making it easier to manage without confusion.

III. ANALYSIS AND CRITICAL DISCUSSION

JUnit is a widely used Java framework for writing and running unit tests, enabling application testing by verifying the behaviour of individual components. We are now gonna show the results and discussions of our JUnit Testing:

Test for Monitoring Messages

In this JUnit testing, the test message is sent by the client to the server through a *PrintWriter*, which is initialised with the output stream of the socket. Upon receiving the message, the server proceeds to retrieve it from the *BufferedReader*, which

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL TEST RESULTS PORTS GITLENS
%TESTC 1 v2
%TSTTREE2,ServerMonitoring,true,1,false,1,ServerMonitoring,,[engine:junit-jupiter]/[class:ServerMonitoring]
%TSTTREE3,testMessageTimestamping(ServerMonitoring),false,1,false,2,testMessageTimestamping(),,[engine:junit-jupiter]/[class:ServerMonitoring]/[method:testMessageTimestamping()]
%TESTS 3,testMessageTimestamping(ServerMonitoring)
  
```

Fig. 3. JUnit-based Testing for Monitoring Message Timestamps

is stored within the message variable. The verification of a timestamp involves comparing the message with a regular expression in a format of "HH:MM:SS". If the message does not match the regular expression, a failure message is displayed. The procedure mentioned earlier guarantees that the message received conforms to the expected structure of a timestamp.

Test for Private Messaging

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL TEST RESULTS PORTS GITLENS
%TESTC 1 v2
%TSTTREE2,ServerPM,true,1,false,1,ServerPM,,[engine:junit-jupiter]/[class:ServerPM]
%TSTTREE3,testIDAssignment(ServerPM),false,1,false,2,testIDAssignment(),,[engine:junit-jupiter]/[class:ServerPM]/[method:testIDAssignment()]
%TESTS 3,testIDAssignment(ServerPM)
  
```

Fig. 4. JUnit based Testing for Private Messaging

The JUnit test simulates a scenario where a client connected with a server and inputs a nickname. To input from the client is in a *ByteArrayInputStream (inContent)* initialised with the string "TestName n". The client creates a connection by using a socket that is created with the IP address of the localhost and a given test port, *TEST_PORT*. A *BufferedReader* is used to read the server response. The received response is then stored in the response variable. The purpose of an assertion is to verify whether the response begins with the expected message that signifies a successful connection and confirms the correct assignment of the client's ID and nickname. In this test, the server properly assigns IDs and controls nickname assignments during the private messaging process.

Test for Admin Change

```

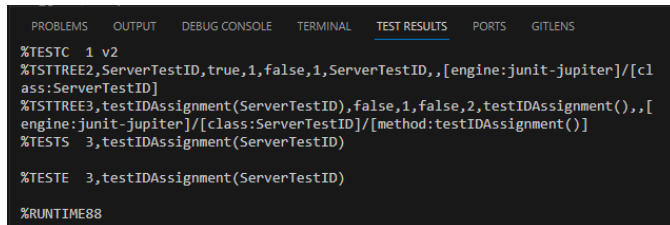
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL TEST RESULTS PORTS GITLENS
%TESTC 1 v2
%TSTTREE2,ServerTestAdminChange,true,1,false,1,ServerTestAdminChange,,[engine:junit-jupiter]/[class:ServerTestAdminChange]
%TSTTREE3,testAdminChange(ServerTestAdminChange),false,1,false,2,testAdminChange(),,[engine:junit-jupiter]/[class:ServerTestAdminChange]/[method:testAdminChange()]
%TESTS 3,testAdminChange(ServerTestAdminChange)
  
```

Fig. 5. JUnit based Testing for Change of Admin

The *testAdminChange()* method in the JUnit test is used to verify the functionality of admin changes in a server application. It requires the setting up of three sockets, creating successful connections, and identifying the administrator. Obtaining the admin's ID involves reading the server's response via the *adminSocket* and parsing it to extract the necessary information. The *adminSocket* is closed to simulate

the admin leaving the server. The remaining user responses (*userSocket1* and *userSocket2*) are analysed to determine if either user qualifies to become the new admin. If a response begins with "You are the new admin," it means that the user has now assumed the role of the admin. The ID of the new admin is obtained from this response. A test was performed to ensure that the ID of the new admin is distinct from the ID of the previous admin, thus confirming the successful assignment of the new admin.

Test for ID Assignment



```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL TEST RESULTS PORTS GITLENS
%TESTC 1 v2
%TSTTREE2,ServerTestID,true,1,false,1,ServerTestID,,[engine:junit-jupiter]/[class:ServerTestID]
%TSTTREE3,testIDAssignment(ServerTestID),false,1,false,2,testIDAssignment(),,[engine:junit-jupiter]/[class:ServerTestID]/[method:testIDAssignment()]
%TESTS 3,testIDAssignment(ServerTestID)

%TESTE 3,testIDAssignment(ServerTestID)

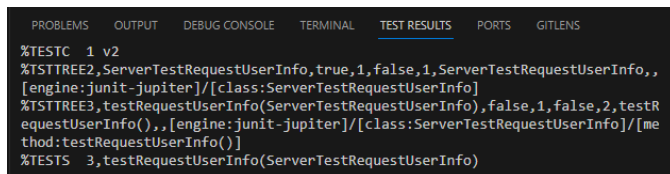
%RUNTIME88

```

Fig. 6. JUnit based Testing for ID Assignment

In a server's point of view, this test responds to a client's connection request with an assigned ID. Using a try-catch block is essential for handling any potential *IOExceptions* that may occur during the simulation. A *StringWriter* is created to capture the server's output together with a *PrintWriter*. The server's response is written to the *PrintWriter*, and a *BufferedReader* is created to be the acting client. It is important to ensure that the simulated client accurately understands the server's response and acknowledges the successful assignment of an ID. This test method confirms the server's capacity to assign an ID to a member by simulating the server's response to a client's connection request and verifying the predicted ID assignment message.

Test for Request User Information



```

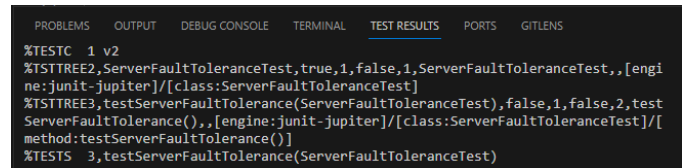
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL TEST RESULTS PORTS GITLENS
%TESTC 1 v2
%TSTTREE2,ServerTestRequestUserInfo,true,1,false,1,ServerTestRequestUserInfo,,[engine:junit-jupiter]/[class:ServerTestRequestUserInfo]
%TSTTREE3,testRequestUserInfo(ServerTestRequestUserInfo),false,1,false,2,testRequestUserInfo(),,[engine:junit-jupiter]/[class:ServerTestRequestUserInfo]/[method:testRequestUserInfo()]
%TESTS 3,testRequestUserInfo(ServerTestRequestUserInfo)

```

Fig. 7. JUnit based Testing for Request User Information

The JUnit test emulates a client connecting to a server, establishing a socket at the designated test port. Verifying a successful connection is done through an assertion. The test sends a request for user information to the server, using the *OutputStream* to write data. To send the request, you simply write "requestinfo username" to the output stream, replacing "username" with the desired username. The *flush()* method guarantees immediate transmission of data. This test method ensures that the server accurately handles user information requests by simulating a client request and verifying the validity of the responses.

Fault Tolerance Test



```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL TEST RESULTS PORTS GITLENS
%TESTC 1 v2
%TSTTREE2,ServerFaultToleranceTest,true,1,false,1,ServerFaultToleranceTest,,[engine:junit-jupiter]/[class:ServerFaultToleranceTest]
%TSTTREE3,testServerFaultTolerance(ServerFaultToleranceTest),false,1,false,2,testServerFaultTolerance(),,[engine:junit-jupiter]/[class:ServerFaultToleranceTest]/[method:testServerFaultTolerance()]
%TESTS 3,testServerFaultTolerance(ServerFaultToleranceTest)

```

Fig. 8. JUnit based Testing for Fault Tolerance

In the JUnit test, the *testServerFaultTolerance()* method is used to assess the fault tolerance capabilities of a server application in the event of an invalid request received from a client. The procedure involves setting up a socket to create a connection with the server located on the localhost, specifically at the designated test port (*TEST_PORT*), and verifying the successful establishment of the connection. For the client to send messages to the server, a *PrintWriter* is set up with the output stream of the socket. To act out the transmission of an invalid request, the client inputs the string "INVALID_REQUEST" into the *PrintWriter*. The client is able to read data sent by the server by initialising a *BufferedReader* with the input stream of the socket. The *readLine()* method is used to retrieve the response from the server, which is subsequently stored in the variable called response. An argument is posited to validate the correspondence between the server's response and the expected error message. In the connection process, *IOExceptions* are detected and regarded as failures.

While learning competence in a new language and coding through a process of trial and error, we encountered several instances where we felt disoriented and uncertain about the right path of action. Therefore, the process of implementing basic features such as timestamps on each text and conducting a 20-second assessment of the currently active users was excessively time-consuming, resulting in delays. Furthermore, during the implementation of the admin code, it functioned properly. However, the primary issue arose when attempting to assign a new administrator after the current one had departed from the chat. The occurrence of difficulties and time wastage resulted from the inability of the administrator to log in to the chat or the malfunctioning of commands whenever the code was updated. As a result, the administrator became isolated from other users, and upon exiting the chat, the new administrator also experienced separation. The aforementioned factor significantly contributed to the delay in the completion of the code. Regarding the JUnit test, the requirements do not specify that it must be functional. However, the inclusion of JUnit testing is still conducted with no proper test results. Regarding the limitation aspect, it can be stated that due to the non-professional nature of this project, which is merely a prototype, future endeavours will involve the development of practical JUnit tests and more complex features.

IV. CONCLUSIONS

In conclusion, this report details the development of a networked distributed system that enables group-based client-server communication, complete with CLI interfaces. Significant features include the capacity for users to register using a unique nickname, automatic assignment of admin privileges upon initial connection, regular state checks conducted by the admin, and the capability for members to request information regarding group details. It is essential for the system to ensure smooth communication between members, even in the event of a member leaving. Failover mechanisms ought to be implemented to guarantee uninterrupted operation. In addition, it is important to follow programming best practices when implementing the solution. This includes employing various design patterns such as Singleton Pattern, Observer Pattern, Command Patterns, and more. It is also vital to conduct thorough testing using JUnit-based Testing, implement fault tolerance measures, and use a version control system like Azure DevOps to manage and monitor the project as a team. Furthermore, using VSCode as the primary coding environment, along with its wide range of extensions for Java Programming. By addressing these requirements, the resulting solution will provide effective and reliable oversight of group interactions in a distributed computing environment.

By carefully studying and implementing the project specification for Java, we acquired deeper understanding about the complexities involved in developing and operating networked distributed systems for group-based client-server communication. This experience has greatly enhanced our skills of programming and performing technical projects. We have developed on how networked distributed system works and its key features especially understanding the fault tolerance strategies to ensure system resilience. Overall, this experience has given us the knowledge and skills to handle complex projects in distributed systems and has helped us grow as skilled Computer Scientists.

ACKNOWLEDGMENT

We would like to express our sincere gratitude to all those who have contributed to the completion of this report. First and foremost, I extend my deepest appreciation to our University and to our Advanced Programming tutors, for their invaluable guidance, support, and constructive feedback throughout the process.

To our group and other colleagues, we are grateful for all the experience and participation in our shared their time and insights, without whom this study would not have been possible.

REFERENCES

- [1] Musch, O. (2023). *Design Patterns with Java: An Introduction*. Germany: Springer Fachmedien, Imprint: Springer Vieweg, p. 24-28, March 2024
- [2] Monday, P.B. (2003). Implementing the Observer Pattern. In: *Web Services Patterns: Java™ Platform Edition*. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4302-0776-4_11
- [3] R. L. R. Maata, R. Cordova, B. Sudramurthy and A. Halibas, "Design and Implementation of Client-Server Based Application Using Socket Programming in a Distributed Computing Environment," 2017 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC), Coimbatore, India, 2017, pp. 1-4, doi: 10.1109/ICCIC.2017.8524573.
- [4] Kulkarni, N. D., & Bansal, S. (2022). Utilizing the Factory Method Design Pattern in Practical Manufacturing Scenarios. *Journal of Material Sciences & Manufacturing Research*. SRC/JMSMR-200. DOI: [doi.org/10.47363/JMSMR/2022\(3\),166,2-5](https://doi.org/10.47363/JMSMR/2022(3),166,2-5).
- [5] Ciancarini, P., Rossi, D. (1997). Jada: Coordination and communication for Java agents. In: Vitek, J., Tschudin, C. (eds) *Mobile Object Systems Towards the Programmable Internet*. MOS 1996. Lecture Notes in Computer Science, vol 1222. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-62852-5_16
- [6] Jed Liu, Aaron Kimball, and Andrew C. Myers. 2006. Interruptible iterators. *SIGPLAN Not.* 41, 1 (January 2006), 283–294. <https://doi.org/10.1145/1111320.1111063>
- [7] Cooper, J. W. (2000). *Java Design Patterns: A Tutorial*. Switzerland: Addison-Wesley.
- [8] Kin-Keung Ma and Jeffrey S. Foster. 2007. Inferring aliasing and encapsulation properties for java. *SIGPLAN Not.* 42, 10 (October 2007), 423–440. <https://doi.org/10.1145/1297105.1297059>
- [9] Simon Beloglavec, Marjan Heričko, Matjaž B. Jurič, and Ivan Rozman. 2005. Analysis of the limitations of multiple client handling in a Java server environment. *SIGPLAN Not.* 40, 4 (April 2005), 20–28. <https://doi.org/10.1145/1064165.1064170>
- [10] Cockshott, P., 2003. The Jpi interface.
- [11] Mili, H., Elkharraz, A., & Mcheick, H. (2004). Understanding separation of concerns. Early aspects: aspect-oriented requirements engineering and architecture design, 75-84.
- [12] Mr, A. K., Kumar, A., & Mr, M. I. (2016). Applying separation of concern for developing softwares using aspect oriented programming concepts. *Procedia Computer Science*, 85, 906-914.
- [13] Harold, E. R. (2010). *Java I/O*. United States: O'Reilly Media.
- [14] Appel, F. (2015). *Testing with JUnit*. United Kingdom: Packt Publishing.
- [15] Garcia, B. (2017). *Mastering Software Testing with JUnit 5: Comprehensive Guide to Develop High Quality Java Applications*. United Kingdom: Packt Publishing.
- [16] Toure, F., Badri, M. & Lamontagne, L. A metrics suite for JUnit test code: a multiple case study on open source software. *J Softw Eng Res Dev* 2, 14 (2014). <https://doi.org/10.1186/s40411-014-0014-6>
- [17] Macero, M. (2017). *Testing the Distributed System*. In: *Learn Microservices with Spring Boot*. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-3165-4_6