Assignment 3

Analysis and Design Document

Student: Rares-Sebastian Moldovan

**Group: 30433**

Table of Contents

1. Requirements Analysis 3

1.1 Assignment Specification 3

1.2 Functional Requirements 3

1.3 Non-functional Requirements 3

2. Use-Case Model 4

3. System Architectural Design 5

4. UML Sequence Diagrams 8

5. Class Design 9

6. Data Model 10

7. System Testing 10

8. Bibliography 11

1. Requirements Analysis

# Assignment Specification

Use Java/C# API to design and implement a client-server application for a news agency. The application has three types of users: the readers, the writers and an administrator. The readers can view a list of articles, read an article and do not need to login in order the use the application. The writers ​need to authenticate in order to ​create, update or delete articles. The admin ​is the only one who can create writer accounts, but cannot create new admin accounts. So the admin accounts are preset by the application developer and cannot be altered. An article has the following components:

* Title
* Abstract
* Author
* Body
* List of related articles

# Functional Requirements

When reading an article, the user should be able to see the title and the abstract of the related articles. By clicking on the title of the related article, he will be taken to a page that displays the full article. The application must support multiple concurrent users. If a writer posts a new article, the readers must see it in the list of articles in real time, without performing any refresh operation.

The reader users can:

* View the list of articles.
* Read articles.

The writer users can:

* Write new articles with associated related articles.
* Update own articles.
* Delete own articles.

Any writer can also be a reader by not logging in.

The administrator can:

* Create new writer accounts.
* View the list of writer accounts.
* Delete writer accounts.
* Update writer accounts.

# Non-functional Requirements

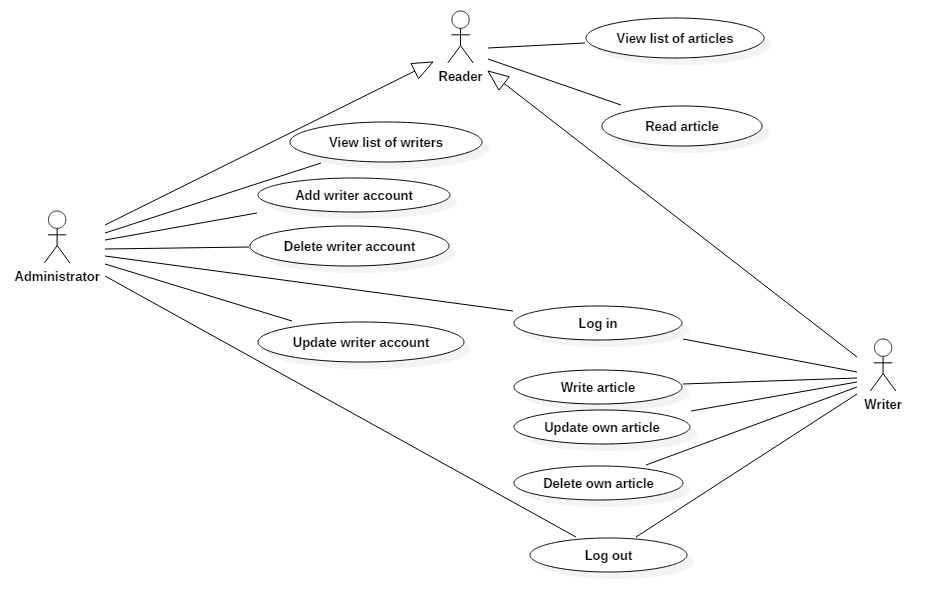
The application must be client-server, implemented using sockets in Java (making use of the java.net.Socket library).

Use the Observer design pattern for updating the list of articles in real time.

For sending data from the client to the server use JSON serialization.

When writing an article, show a list that supports multi-select for choosing the related articles.

2. Use-Case Model

**Use case:** Write article

**Level:** user-goal level

**Primary actors:** Writer

**Main success scenario:**

* The user successfully launches the application.
* The application displays the list of articles together with an authentication form.
* The user successfully logs in.
* The user fills in the content for the article: title, body, abstract.
* The user selects the related articles from the associated list.
* The user hits the save button.
* The new article is saved in the database and automatically displayed to all online users.

**Extensions:**

* If the user does not select any related article, it is automatically considered that the list of related articles should be empty.
* If the user provides wrong credentials, an error message is shown.

3. System Architectural Design

**3.1 Architectural Pattern Description**

The application relies on the **Client-Sever** architectural pattern. This architecture was used because of the non-functional requirement of the assignment that stated the use of this pattern.

The main idea is to split the functionality into two main parts: client side and server side. The client side is responsible for providing the user the interface with the system and send request messages to the server. The server side is responsible for fulfilling these requests by responding and/or accessing the database.

The communication between the two components is standardized (the request/receive message encoding is known to both of them and they both respect the same format of the sent data).

1. **Client**

The client represents the component which allows the user to interact with the system. There are two types of clients : thin and fat client. A thin client is a client which provides only the interface with the user and has no business logic. A fat client provides some business logic besides the means of communicating with the system.

This system relies on a relatively thin client which provides the user interface and a listener object (a thread which continually listens for server requests and provides the interface for sending messages).

1. **Server**

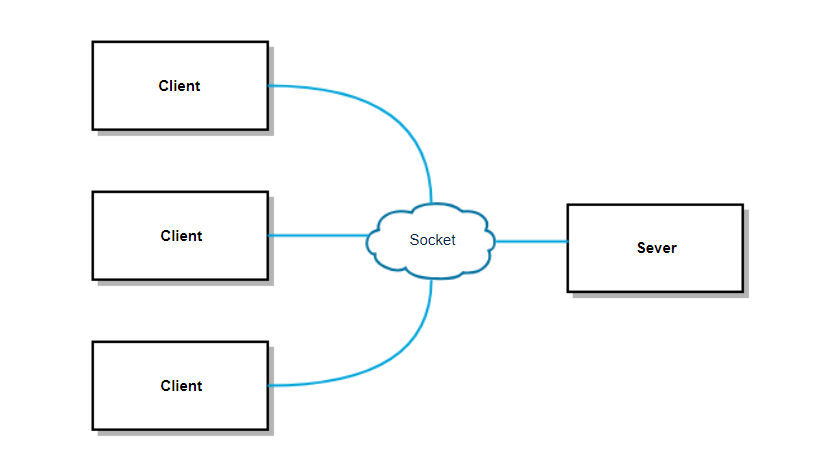
The server is the “common ground” for all clients to share. The server generally takes care of the business logic and data access. A server can be specialized (database server) or it can store the logic for different types of transactions. The server provides its interface to the clients via messages. Messages can be of both types: from client to server and from server to client. Any message generally has a response.

This system relies on the server to store the data, retrieve it and perform CRUD operations on the data.

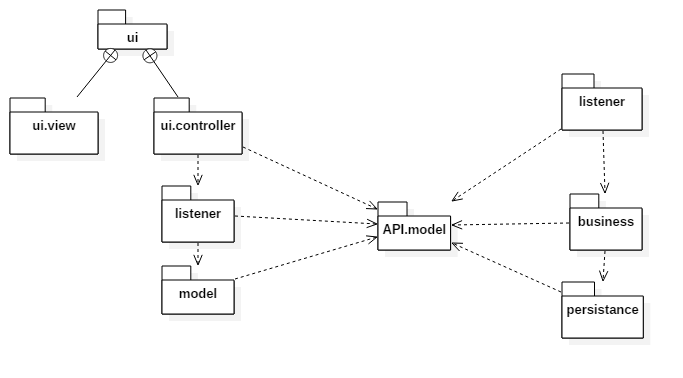
Communication is possible using some data streams (input and output streams) on which messages are stored. The most basic way for obtaining such a functionality is through the use of sockets. The client and the server must both know the port and the address and must access the socket. After obtaining such a socket object, they can send and receive messages at any time. However, the communication must be synchronous (the server must be able to acquire client requests from numerous clients at a moment in time), therefore the server has multiple running threads that wait for requests.

**3.2 Diagrams**

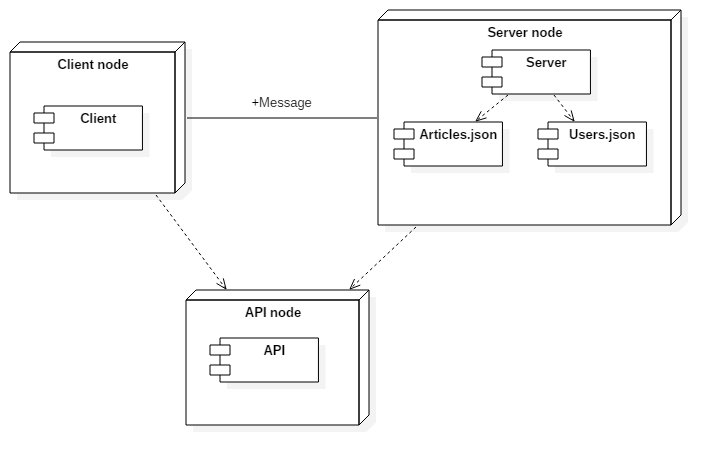
**3.2.1 Conceptual Architecture**



**3.2.2 Package diagram**

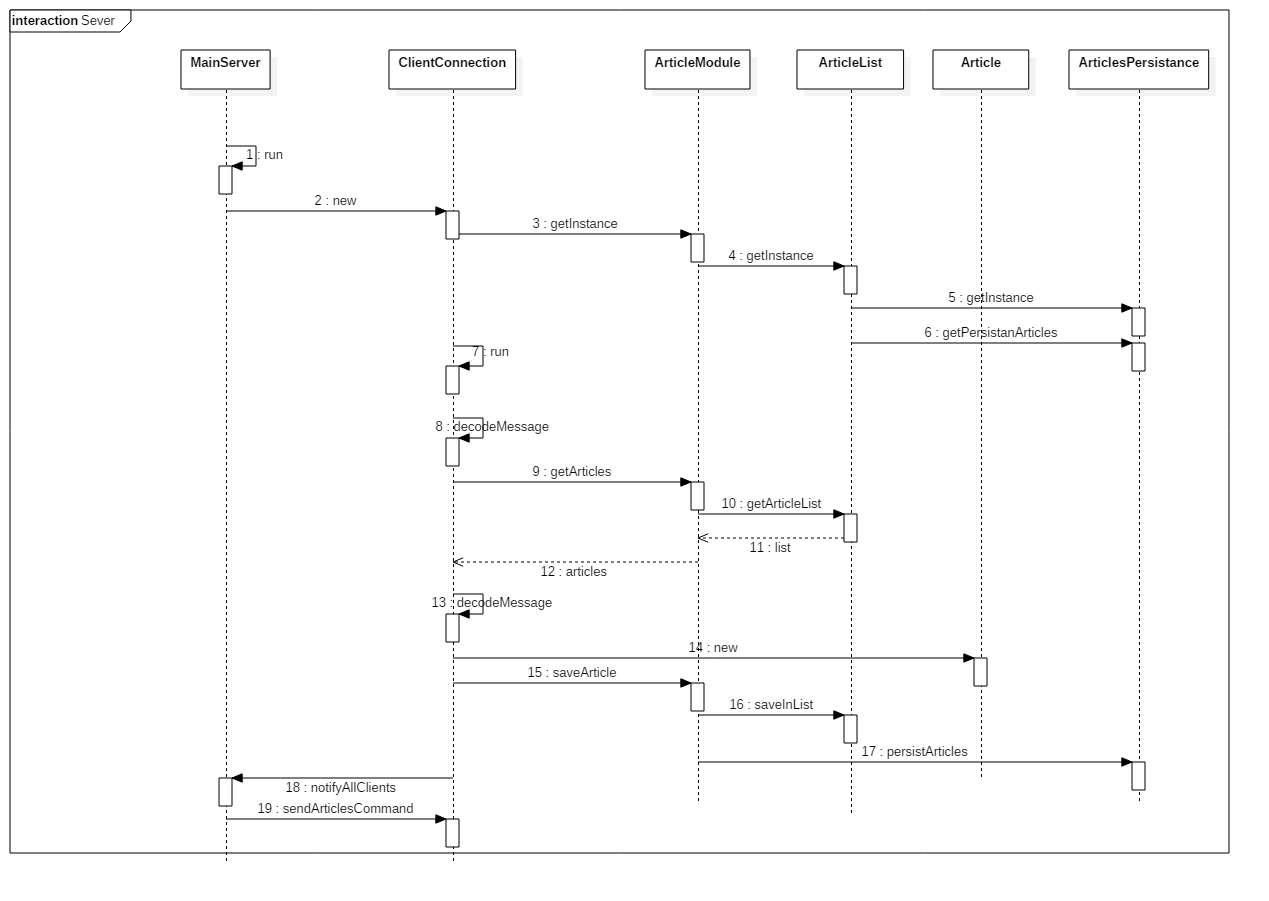
****

**3.2.3 Deployment diagram**

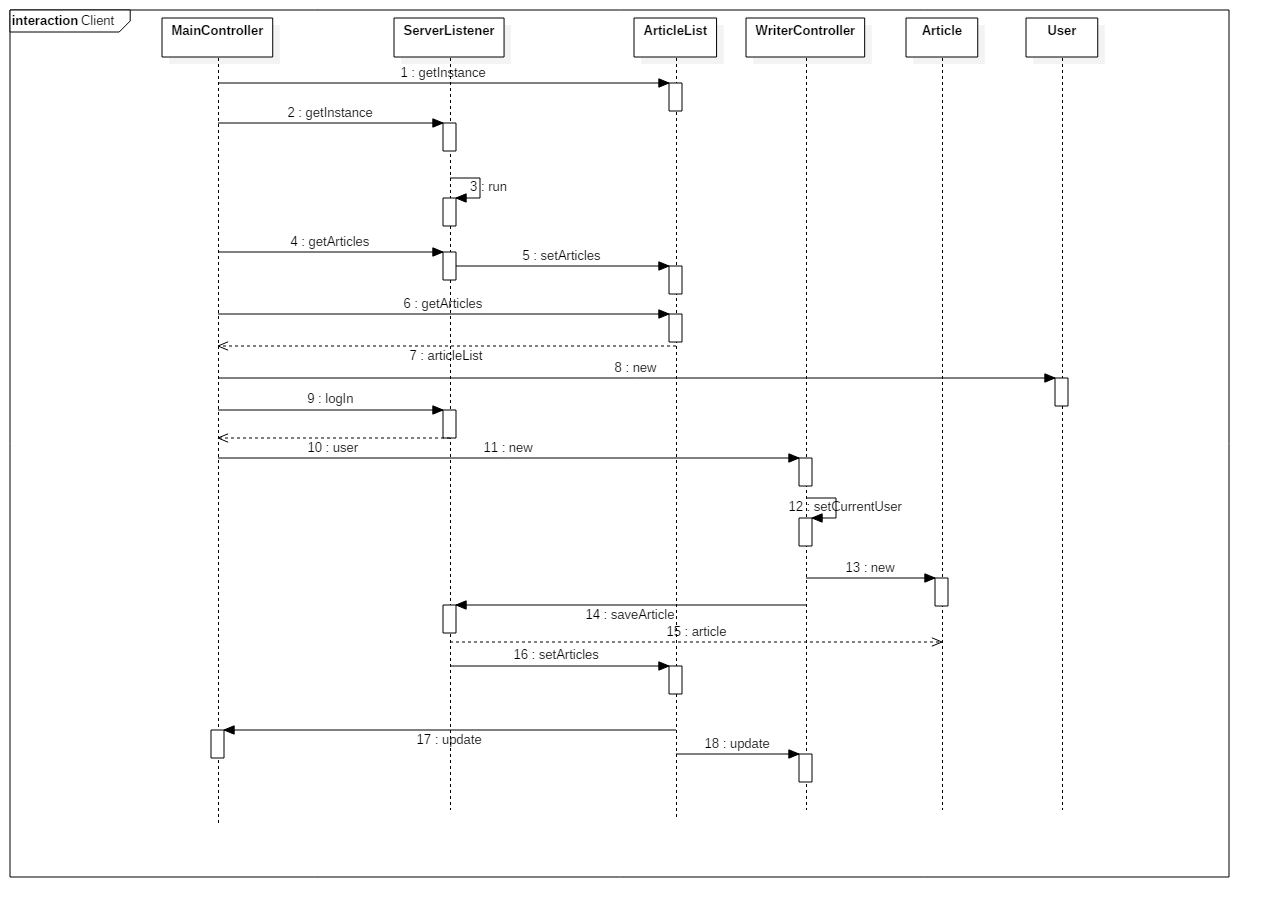
****

4. UML Sequence Diagrams

Sequence diagram for Sever – save article



Sequence diagram for Client– save article



5. Class Design

**5.1 Design Patterns Description**

This assignment focused on the Client-Sever architectural pattern and did not impose many design patterns to work with. However, one design pattern had a very important role to play in the automatic update of the article lists in all clients: Observer Design pattern.

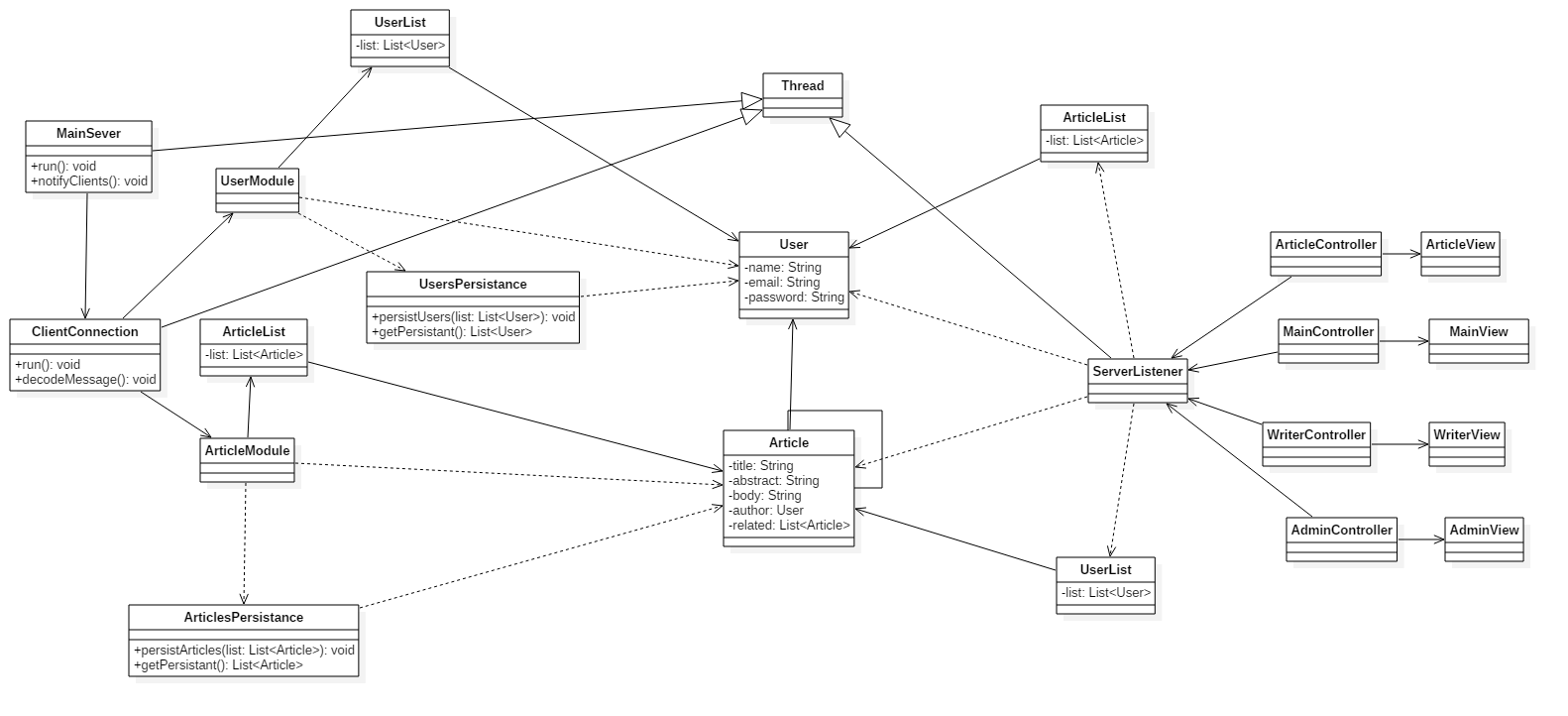
**The Observer Design Pattern**

The observable design pattern was induced by the MVC architecture used for constructing the GUI. The view is an observer of the model, the model being observable. This design pattern allows notification of one object that another has changed. Any observer can add observable objects to an internal list.

Whenever a client saves/updates/deletes an article, all clients (including the sender client) are notified via an update message. When this message is received, the internal article list of the client is changed. This list is observable, and it is observed by the main view and the writer view. The content of the article list in both views is updated with the necessary changes using the update method inherited from the Observer interface.

The article list class makes use of notification methods to announce the observers that the content of the list has changed. This notification is performed in methods for adding an article and setting the article list.

**5.2 UML Class Diagram**



6. Data Model

The application did not use a database. However, persistence was accomplished by using JSON serialization in local files. The two files: Articles.json and Users.json store the list of article and user objects.

As seen in the class diagram, an article is characterized by:

* *Title*
* *Author*
* *Abstract*
* *Body*
* *List of related articles.*

A user is characterized by:

* *Name*
* *Email*
* *Password*

There is a “one to many” relationship for Article entity itself. One article has many related articles.

There is a “one to many” relationship between the User entity and the Article entity. One user has many articles.

This functionality was accomplished by storing the list of related articles as a java.util. List<Article> and the author as a User entity. The same look would be present even if the project was to use a database since the foreign key constraints in the database translate to associations in the object-oriented language.

7. System Testing

The business logic for the application was not very complicated since it supposed only simple operations (CRUD). The following were the most interesting testing scenarios:

1. Log in with the same credentials from two different clients.
2. Test for real-time update of the list when saving an article (testing the reliability of the observer pattern).
3. Test basic CRUD operations on users.
4. Test for real-time update of the “related” list for the writer frame.
5. Test for client sudden disconnection.
6. Test for server disconnection.

The most informative test scenarios were the ones regarding client/server disconnections since they yielded exception problems in both components. These problems were treated afterwards using exception handling mechanism and, most importantly, adding a “close” command between the two components. This command informs one component about the other one not being available anymore and the system has the necessary functionality for treating such an issue (disconnect the client or display an informative message to the user that the server is no longer available for operations and it must try a later time or try to reconnect).

8. Bibliography

1. Client Server architecture

<https://en.wikipedia.org/wiki/Client%E2%80%93server_model>

<https://github.com/buzea/Software-Design>

<https://docs.oracle.com/javase/tutorial/networking/sockets/clientServer.html>

1. Observer design pattern

<https://www.javaworld.com/article/2077258/learn-java/observer-and-observable.html>

<https://docs.oracle.com/javase/7/docs/api/java/util/Observer.html>

1. Sockets in Java

<https://docs.oracle.com/javase/7/docs/api/java/net/Socket.html>

<http://www.baeldung.com/a-guide-to-java-sockets>

1. JSON serialization

<https://www.mkyong.com/java/json-simple-example-read-and-write-json/>

<https://www.journaldev.com/2315/java-json-example>