Technical Design Document

Virtual Letter of Life (VLOL) Application

Version 1.1

Sefanit Urgessa

Mohammed Allibalogun

Robert Garcia

Meron Getachew Debela

Charles Baisie

Augustin Mwamba

Jamal Bourne

SWEN 670 Software Engineering Project

University of Maryland Global Campus

Date: 07/21/2020

**REVISION HISTORY**

|  |  |  |
| --- | --- | --- |
| **Date** | **Version** | **Description** |
| 06/28/2020 | 1.0 | Initial Technical Design Document Release by Charles Baisie  Reviewed by EMSPlus internal team |
| 06/30/2020 | 1.1 | Populated place holders and incorporated retrospective changes by Robert Garcia  Reviewed and approved by EMSPlus internal team |
| 07/21/2020 | 2.0 | Version includes feedback from Dr. Mir Assadullah |
|  |  |  |

Table of Contents

[1. Introduction 5](#_Toc44450482)

[1.1. Purpose 5](#_Toc44450483)

[1.2. Application Description and Document Scope 5](#_Toc44450484)

[1.3. Document Organization 6](#_Toc44450485)

[1.4. Reference Material 6](#_Toc44450486)

[1.5. Definitions and Acronyms 6](#_Toc44450487)

[2. System Overview 6](#_Toc44450488)

[3. System Architecture 8](#_Toc44450489)

[3.1. Architectural Design 8](#_Toc44450490)

[3.2. Decomposition Description 9](#_Toc44450491)

[3.3. Error and Exception Handling 10](#_Toc44450492)

[3.4. Design Rationale 10](#_Toc44450493)

[4. Data Design 11](#_Toc44450494)

[4.1. Data Description 11](#_Toc44450495)

[4.2. Data Dictionary 11](#_Toc44450496)

[4.3. Entity Relationship Diagram 15](#_Toc44450497)

[5. Component Design 15](#_Toc44450498)

[6. Human Interface Design 31](#_Toc44450499)

[6.1. Overview of User Interface 31](#_Toc44450500)

[6.2. Screen Images and Actions 31](#_Toc44450501)

[7. Requirement Matrix 31](#_Toc44450502)

Table of Tables

Table 1 – VLOL Solution Stack 6

Table 2 – Allergy Data Dictionary 11

Table 3 – Illness Data Dictionary 12

Table 4 – Medication Data Dictionary 12

[Table 5 – Role Data Dictionary 12](#_Toc517588951)

[Table 6 – User Data Dictionary 12](#_Toc517588951)

[Table 7 – Allergy-User associative table 13](#_Toc517588951)

[Table 8 – Illness-User associative table 13](#_Toc517588951)

[Table 9 – Medication-User composite table 1](#_Toc517588951)4

[Table 10 – Allergy Class 1](#_Toc517588951)6

[Table 11 – Condition Class 1](#_Toc517588951)7

[Table 12 – Medication Class 1](#_Toc517588951)9

[Table 13 – Role Class 21](#_Toc517588951)

[Table 14 – User Class 23](#_Toc517588951)

[Table 15 – Requirement Matrix 31](#_Toc517588951)

Table of Figures

Figure 1 - Basic design functionality 8

Figure 2 - VLOL UML Diagram 9

Figure 3 - VLOL Decomposition Diagram 10

Figure 4 - VLOL ERD 15

# Introduction

## Purpose

This technical design document (TDD) describes the architecture and system design for release 1.1 of the Virtual Letter of Life (VLOL) to allow members of the project team to understand:

* The functionality and design of the application.
* The structures the application requires to store, process, and organize the data.
* The components the application requires to manipulate this data into information for the user.
* The components the application requires to control the flow of input and information between the application and the user.
* How the application presents information to the user.

The end state is a document that allows future development teams to understand, recreate, and improve this software without further input from the original development team.

## Application Description and Document Scope

Essentially, the VLOL application is a web-based version of the Letter of Life contained in Enclosure (1). However, it stores user input in a database, allowing authenticated users access to that information from a desktop or mobile browser. It also allows authenticated users to add, search, view, update, and delete basic medical information about themselves or others, depending on their level of authorization:

* Program Participant - View, update, or delete their individual account information, including their password, security question, or security question answer.
* Medical Services Provider (e.g., EMS, ED Staff, etc.) - Search and view the medical information of program participants (this does not include the participant's password, security question, or security question answer).
* Patient Agent (Primary Care Physician, Power of Attorney (POA) agent, etc.) - Search, view, and update the medical information of program participants (this does not include the participant's password, security question, or security question answer).
* System Administrator - Add, search, view, update, or delete all authenticated users, their roles, or their account information (this does not include the participant's password, but the system administrator may authorize a password reset).
* System Administrators may also add, search, view, update, or delete information contained in ancillary tables, such as allergies, conditions, medications, and roles.

Providers, agents, and administrators may search for program participants by entering their information in a search textbox or by scanning a Quick Response (QR) code that corresponds to the participant's identification number. In addition, the VLOL application allows unauthenticated and unauthorized users (e.g., prospective participants, etc.) to learn about the application and register (i.e., add their account information).

This document reviews the functionality mentioned above for future development teams. All other functionality or design considerations, such as client system integration, network security, virtualization, etc., are outside the scope of this document.

## Document Organization

This document is organized into several sections, beginning with a system overview, and progressing through a detailed explanation of the application's architecture, as well as the design of the data structures, components, and user interface. This document concludes with a matrix that cross-references the client's requirements with the functionality of the application. This is to ensure that the development team has met all the client's requirements; the authors encourage team members to read the VLOL SRS, before continuing. Finally, the authors intend this document to be a reference for current and future development teams, as well as project managers, marketing staff, users, testers, and documentation writers.

## Reference Material

* Assadullah, M. (2020, May 20). SWEN 670 9040 Software Engineering Project Syllabus. Retrieved from <https://learn.umgc.edu/d2l/le/content/484481/viewContent/18304635/View>
* EMS Plus (2020, June 9). Software Requirements Specification (SRS) for the Virtual Letter of Life.

## Definitions and Acronyms

While the verbs "register", "create" and "add"; "read" and "view"; and "update" and "edit" are interchangeable throughout this document, the authors intend the former to apply to system actions and the latter to user actions. The nouns "user interface" and "view" are also interchangeable, as the authors are using the Model-View-Controller (MVC) Architecture to structure this application.

# System Overview

The VLOL application was primarily written in Java, using the following solution stack:

Table 1. VLOL Solution Stack

| **Stack Item:** | **Software Used:** |
| --- | --- |
| Operating System: | Linux (Heroku-18 / Ubuntu 18.04) |
| Web Server: | Tomcat Embedded Jasper (10.0.0-M6) |
| Database Server: | H2 (1.4.200) (Development)  MySQL (8.0.20) (Development / Production) |
| Database Language: | SQL (2016) |
| Languages and Frameworks: | Java 11 and Spring Boot (2.2.1)  HTML5 and Thymeleaf (3.0.11)  CSS3  JavaScript (ECMAScript 2018) |
| Dependency Build Tool: | Maven (3.6.3) |
| Dependencies:  The software produced by maven already has these dependencies included. Therefore, the deployer does not need to deploy these dependencies separately. | commons-validator - commons-validator  org.springframework.boot:  spring-boot-starter-web  spring-boot-starter-security  spring-boot-starter-jdbc  spring-boot-starter-data-jpa  spring-boot-starter-thymeleaf  spring-boot-starter-validation  spring-boot-starter-test  org.springframework.security - spring-security-test  com.h2database - h2  org.apache.tomcat.embed - tomcat-embed-jasper  javax.servlet - jstl  org.hibernate - hibernate-validator  javax.validation - validation-api  com.google.zxing - core and javase |
| Testing and Analysis Tools: | JUnit (5.6.2)  SonarQube (7.9.3)  Selenium |

Based on the requirements in the Virtual Letter of Life SRS (EMS Plus, 2020), the basic design and functionality of the application, without authentication or authorization, is shown below (from top to bottom and back):

*Figure 1. Basic design functionality*

As stated earlier:

* Unauthenticated and unauthorized users will have access to the Main Menu and Register User functions.
* Program Participants will have access to View User Details, Edit User Details and Delete User functions for their account only.
* Medical Services Providers will have access to View and Search Users and View User Details functions for all participants.
* Patient Agents will have access to View and Search Users, View User Details, and Edit User Details functions of those participants who have authorized them as their agent.
* System Administrators will have access to all functions, except for Register User.

# System Architecture

## Architectural Design

The organizer is written using the Model-View-Controller (MVC) architectural pattern. The Model validates and loads file data into the User, Allergy, Condition, Medication, and Role objects, as well as converts data into information for the Controller on demand. The View accepts, validates, and passes on user input to the Controller, and receives information from the Model through the Controller. The Controller controls the flow of input and information between the Model and the View, allowing the data and user interfaces to act independently of each other. Please note that while requests and data can flow between objects within the Model, and Views may interact with each other through POSTs and query strings, Models and Views should interact through the Controller and not each other.

Here is the basic architectural design for the VLOL application. Due to its size, the standard getters and setters have been removed and a full-size, fold out copy has been included with this document as Enclosure (2):

Figure 2. VLOL UML Diagram

A close up of a map

Description automatically generated

## Decomposition Description

The following decomposition diagram is broken down by user authorization:

Figure 3. VLOL Decomposition Diagram

## Error and Exception Handling

Each View validates input on the front-end, and the Model validates data on the back end, to reduce the possibilities of general exceptions. In addition, each method in the model has been unit-tested against input methods to ensure data-type compatibility. Finally, all activity, including errors and exceptions, are recorded in a log for auditing.

## Design Rationale

Following several discussions with the Development and Operations (DEVOPS) Team, the authors selected a Java solution stack, specifically the Spring Boot MVC architectural pattern, for several reasons:

* First, the built-in Hibernate object-relational mapping (ORM) tool allows us to quickly link classes (DDL) and objects (DML) to data storage correctly, without having to write custom data access code based on the database server software (e.g., MySQL, Oracle, MS Server, etc.).
* Second, the Thymeleaf Java template engine uses syntax that maps server-side variables directly to common HTML-elements on the client-side (e.g., <option value="{\*userName}">, etc.), as opposed to other server-client translation technologies, such as Java Server Faces (JSF), which incorporate their own syntax and are difficult to maintain. Thymeleaf also works seamlessly with the Bootstrap CSS framework, as opposed to JSF, which requires the Bootfaces or Primefaces variants.
* Third, Spring Boot has built-in support for various unit testing frameworks, such as JUnit, allowing the easy creation of methods and unit tests for those methods.

# Data Design

## Data Description

The objects created by the VLOL application are packaged in different data structures based on their use. For example, searching for users based on a certain criterion produces a List of User objects, while Role dropdown lists in the view are populated by a Set of Role objects. Each object also contains different data types, following Java standards (e.g., Long, String, etc.). The return values of the data manipulation methods are listed in Section 5, Component Design.

One important note to remember is that the Hibernate ORM automatically converts Java data types to the appropriate database server data type. For example, Hibernate will translate Long data types to BIGINT if a MySQL database is used or String to TEXT if a SQLite database is used. Another item of note is that there are no super classes or sub-classes, therefore there is no inheritance or polymorphism.

Another note is that the term "Condition" is the appropriate term for a patient's health concern in the medical field (e.g., pregnancy is not an illness, etc.). However, "condition" is a reserved SQL99 keyword, so the term "illness" is used in the data dictionary instead.

## Data Dictionary

The main entities of the database are shown below. "PK" indicates the column is a primary key, "NN" indicates the column does not accept null values, "UQ" indicates the column values must be unique. Please remember that if these entities do not exist when the application first runs, Hibernate can create their tables in the database from the classes in the model, and apply the appropriate data type based on the database server software (e.g., MySQL, Oracle, MS Server, etc.).

Table 2. Allergy Data Dictionary

| **Field Name** | **Data Type** | **Modifiers** | **Description** |
| --- | --- | --- | --- |
| allergy\_id | BIGINT | PK, NN, UQ | The unique ID for an allergy |
| allergy\_name | VARCHAR(50) | NN, UQ | The allergy's name |

Table 3. Illness Data Dictionary

| **Field Name** | **Data Type** | **Modifiers** | **Description** |
| --- | --- | --- | --- |
| illness\_id | BIGINT | PK, NN, UQ | The unique ID for an illness |
| illness\_name | VARCHAR(50) | NN, UQ | The illness' name |

Table 4. Medication Data Dictionary

| **Field Name** | **Data Type** | **Modifiers** | **Description** |
| --- | --- | --- | --- |
| medication\_id | BIGINT | PK, NN, UQ | The unique ID for a medication |
| blood\_thinner | BOOLEAN |  | Blood thinner alert flag |
| brand\_name | VARCHAR(50) | UQ, NN | The medication's brand name |
| controlled | BOOLEAN |  | Controlled substance alert flag |
| drug\_action | VARCHAR(50) | NN | The medication's action on the body |
| generic\_name | VARCHAR(50) | NN | The medication's generic name |

Table 5. Role Data Dictionary

| **Field Name** | **Data Type** | **Modifiers** | **Description** |
| --- | --- | --- | --- |
| role\_id | BIGINT | PK, NN, UQ | The user's role unique ID |
| role\_description | VARCHAR(512) |  | The description for this database user's role |
| role\_level | INTEGER | NN, role\_level <= 20 | The user's role level; must be less than or equal to 20 |
| role\_title | VARCHAR(32) | UQ, NN | The title for this database user's role |

Table 6. User Data Dictionary

| **Field Name** | **Data Type** | **Modifiers** | **Description** |
| --- | --- | --- | --- |
| user\_id | BIGINT | PK, NN, UQ | The unique ID for a user |
| role\_id | BIGINT | NN | The role id of the user |
| admin\_comments | VARCHAR(300) |  | System administrator comments |
| adv\_dir\_type | VARCHAR(50) |  | Advance directive type |
| adv\_directive | BOOLEAN |  | Does the user have an advance directive? |
| city | VARCHAR(50) | NN | The user's city of residence |
| date\_created | TIMESTAMP | NN | The creation date for this database user account |
| dob | DATE | NN | The user's date of birth |
| doctor\_name | VARCHAR(100) |  | The user's primary care physician |
| doctor\_phone | VARCHAR(10) |  | The primary care physician's phone number |
| email | VARCHAR(320) | NN, UQ | The user's email address |
| first\_name | VARCHAR(32) | NN | The user's first name |
| ins\_company | VARCHAR(50) |  | The user's medical insurance company name |
| ins\_policy\_no | VARCHAR(32) |  | The user's medical insurance policy number |
| is\_active | BOOLEAN |  | Is the user's account active? |
| is\_locked | BOOLEAN |  | Is the user's account locked? |
| last\_login\_date | TIMESTAMP | NN | The timestamp of the user last login |
| last\_name | VARCHAR(100) | NN | The user's last name |
| password | VARCHAR(72) | NN | The generated hash value of the user's password |
| phone | VARCHAR(10) | NN | The user's phone number |
| poc\_name | VARCHAR(100) | NN | A point of contact for the user |
| poc\_phone | VARCHAR(10) | NN | The phone number for the user's point of contact |
| security\_answer | VARCHAR(72) | NN | The generated hash value of the user's security answer |
| security\_question | VARCHAR(100) | NN | Question enabling the user to re-set a forgotten password |
| ssn | VARCHAR(9) | NN, UQ | The user's social security number |
| street\_address | VARCHAR(100) | NN | The user's street address |
| us\_state | VARCHAR(2) | NN | The user's state of residence |
| user\_agent\_id | BIGINT |  | The User ID of the user's agent |
| user\_comments | VARCHAR(300) |  | User additional comments |
| username | VARCHAR(320) | NN, UQ | The user's email address |
| zipcode | VARCHAR(5) | NN | The user's zip code number |

In addition, there are two associative entities and one composite entity, indicating a many-to-many relationship with the User entity:

Table 7. Allergy-User associative table

| **Field Name** | **Data Type** | **Modifiers** | **Description** |
| --- | --- | --- | --- |
| allergy\_id | BIGINT | PK, NN, UQ | The unique ID for an allergy |
| user\_id | BIGINT | PK, NN, UQ | The unique ID for a user |

Table 8. Illness-User associative table

| **Field Name** | **Data Type** | **Modifiers** | **Description** |
| --- | --- | --- | --- |
| illness\_id | BIGINT | PK, NN, UQ | The unique ID for an illness |
| user\_id | BIGINT | PK, NN, UQ | The unique ID for a user |

Table 9. Medication-User composite table

| **Field Name** | **Data Type** | **Modifiers** | **Description** |
| --- | --- | --- | --- |
| medication\_id | BIGINT | PK, NN, UQ | The unique ID for a medication |
| user\_id | BIGINT | PK, NN, UQ | The unique ID for a user |
| dosage | FLOAT | NN | The amount of medication taken |
| frequency | VARCHAR(32) | NN | How often the medication is taken (once, twice daily, etc.) |

## Entity Relationship Diagram

Figure 4. VLOL ERD

A screenshot of a cell phone

Description automatically generated

# Component Design

The main classes of the application are shown below. Recommended constraints, such as format and size, on class attributes, and by default on generic getters and setters, are also listed:

Table 10. Allergy Class

| **Class Name:** | **Allergy** |
| --- | --- |
| Class Description/Purpose: | The Allergy class holds the attributes of an Allergy object |
| Class Modifiers: | public |
| Class Inheritance: | None |
| Class Attributes: | **Long allergyID**  Description: The unique ID for an allergy  Data dictionary name: "allergy\_id"  **String allergyName**  Description: The allergy's name  Data dictionary name: "allergy\_name"  Regex pattern for validation per RFC 3986: "^[A-Za-z0-9\\s\\-.\_~:\\/?#\\[\\]@!$&'()\*+,=]\*$"  Max length allowed: 50 |
| Exceptions Thrown: | java.lang.Exception subclasses (No custom exceptions) |
| Class Methods: | (Standard attribute getters and setters included, but not listed for brevity)  **List<Allergy> getAllAllergies()**  Description: This method retrieves all the allergies from the database.  Returns: A list of Allergy objects.  Pseudocode: No inherited methods but returns null if no allergies exist.  **List<Allergy> findAllergyByKeyword(String keyword)**  Description: This method retrieves allergies from the database using a keyword.  Parameter(s): keyword - a search term inputted by the user.  Return: A list of Allergy objects containing the keyword  Pseudocode: No parameter manipulation or inherited methods but returns null if no allergy information contains the keyword.  **Allergy getAllergy(long allergyID)**  Description: This method retrieves a specific allergy from the database.  Parameter(s): allergyID - The ID number of the allergy.  Returns: An Allergy object.  Pseudocode: Fails if allergy does not exist.  **void addAllergy(String allergyName)**  Description: This method adds an allergy to the database.  Parameter(s): allergyName - The name of the allergy.  Pseudocode: No parameter manipulation or inherited methods but fails if any parameter fails to meet the attributes length or format constraints.  **void updateAllergy(Long allergyID, String allergyName)**  Description: This method updates allergy information in the database.  Parameter(s):   * allergyID - The ID number of the allergy. * allergyName- The name of the allergy.   Pseudocode: No parameter manipulation or inherited methods but fails if any parameter fails to meet the attributes length or format constraints.  **void saveAllergy(Allergy allergy)**  Description: This method is required by Hibernate to save an allergy to the database.  Parameter(s): The Allergy object.  Pseudocode: Fails if allergy does not exist.  **void deleteAllergy(long allergyID)**  Description: This method deletes an allergy from the database.  Parameter(s): The ID number of the allergy.  Pseudocode: Fails if allergy does not exist. |

Table 11. Condition Class

| **Class Name:** | **Condition** |
| --- | --- |
| Class Description/Purpose: | The Condition class holds the attributes of a Condition object |
| Class Modifiers: | public |
| Class Inheritance: | None |
| Class Attributes: | **Long conditionID**  Description: The unique ID for a condition  Data dictionary name: "condition\_id"  **String conditionName**  Description: The condition's name  Data dictionary name: "condition\_name"  Regex pattern for validation per RFC 3986: "^[A-Za-z0-9\\s\\-.\_~:\\/?#\\[\\]@!$&'()\*+,=]\*$"  Max length allowed: 50 |
| Exceptions Thrown: | java.lang.Exception subclasses (No custom exceptions) |
| Class Methods: | (Standard attribute getters and setters included, but not listed for brevity)  **List<Condition> getAllConditions()**  Description: This method retrieves all the conditions from the database.  Returns: A list of Condition objects.  Pseudocode: No inherited methods but returns null if no conditions exist.  **List<Condition> findConditionByKeyword(String keyword)**  Description: This method retrieves conditions from the database using a keyword.  Parameter(s): keyword - a search term inputted by the user.  Return: A list of Condition objects containing the keyword.  Pseudocode: No parameter manipulation or inherited methods but returns null if no condition information contains the keyword.  **Condition getCondition(long conditionID)**  Description: This method retrieves a specific condition from the database.  Parameter(s): conditionID - The ID number of the condition.  Returns: A Condition object.  Pseudocode: Fails if condition does not exist.  **void addCondition(String conditionName)**  Description: This method adds a condition to the database.  Parameter(s): conditionName - The name of the condition.  Pseudocode: No parameter manipulation or inherited methods but fails if any parameter fails to meet the attributes length or format constraints.  **void updateCondition(Long conditionID, String conditionName)**  Description: This method updates condition information in the database.  Parameter(s):   * conditionID - The ID number of the condition. * conditionName- The name of the condition.   Pseudocode: No parameter manipulation or inherited methods but fails if any parameter fails to meet the attributes length or format constraints.  **void saveCondition(Condition condition)**  Description: This method is required by Hibernate to save a condition to the database.  Parameter(s): The Condition object.  Pseudocode: Fails if condition does not exist.  **void deleteCondition(long conditionID)**  Description: This method deletes a condition from the database.  Parameter(s): The ID number of the condition.  Pseudocode: Fails if condition does not exist. |

Table 12. Medication Class

| **Class Name:** | **Medication** |
| --- | --- |
| Class Description/Purpose: | The Medication class holds the attributes of a Medication object |
| Class Modifiers: | public |
| Class Inheritance: | None |
| Class Attributes: | **Long medicationID**  Description: The unique ID for a medication  Data dictionary name: "medication\_id"  **String brandName**  Description: The medication's brand name  Data dictionary name: "brand\_name"  Regex pattern for validation per RFC 3986: "^[A-Za-z0-9\\s\\-.\_~:\\/?#\\[\\]@!$&'()\*+,=]\*$"  Max length allowed: 50  **String genericName**  Description: The medication's generic name  Data dictionary name: "generic\_name"  Regex pattern for validation per RFC 3986: "^[A-Za-z0-9\\s\\-.\_~:\\/?#\\[\\]@!$&'()\*+,=]\*$"  Max length allowed: 50  **String drugAction**  Description: The medication's action on the body  Data dictionary name: "drug\_action"  Regex pattern for validation per RFC 3986: "^[A-Za-z0-9\\s\\-.\_~:\\/?#\\[\\]@!$&'()\*+,=]\*$"  Max length allowed: 50  **Boolean controlled**  Description: Controlled substance alert flag  Data dictionary name: "controlled"  Default value is false  **Boolean bloodThinner**  Description: Blood thinner alert flag  Data dictionary name: "blood\_thinner"  Default value is false |
| Exceptions Thrown: | java.lang.Exception subclasses (No custom exceptions) |
| Class Methods: | (Standard attribute getters and setters included, but not listed for brevity)  **List<Medication> getAllMedications()**  Description: This method retrieves all the medications from the database.  Returns: A list of Medication objects.  Pseudocode: No inherited methods but returns null if no medications exist.  **List<Medication> findMedicationByKeyword(String keyword)**  Description: This method retrieves medications from the database using a keyword.  Parameter(s): keyword - a search term inputted by the user.  Return: A list of Medication objects containing the keyword.  Pseudocode: No parameter manipulation or inherited methods but returns null if no medication information contains the keyword.  **Medication getMedication(long medicationID)**  Description: This method retrieves a specific medication from the database.  Parameter(s): medicationID - The ID number of the medication.  Returns: A Medication object.  Pseudocode: Fails if medication does not exist.  **void addMedication(String brandName, String genericName, String drugAction, Boolean controlled, Boolean bloodThinner)**  Description: This method adds a medication to the database.  Parameter(s):   * brandName - The brand name of the medication. * genericName - The generic name of the medication. * drugAction - The brand name of the medication. * controlled - Is the medication controlled? * bloodThinner - Is the medication a blood thinner?   Pseudocode: No parameter manipulation or inherited methods but fails if any parameter fails to meet the attributes length or format constraints.  **void updateMedication(Long medicationID, String medicationName)**  Description: This method updates medication information in the database.  Parameter(s):   * medicationID - The ID number of the medication. * brandName - The brand name of the medication. * genericName - The generic name of the medication. * drugAction - The brand name of the medication. * controlled - Is the medication controlled? * bloodThinner - Is the medication a blood thinner?   Pseudocode: No parameter manipulation or inherited methods but fails if any parameter fails to meet the attributes length or format constraints.  **void saveMedication(Medication medication)**  Description: This method is required by Hibernate to save a medication to the database.  Parameter(s): The Medication object.  Pseudocode: Fails if medication does not exist.  **void deleteMedication(long medicationID)**  Description: This method deletes a medication from the database.  Parameter(s): The ID number of the medication.  Pseudocode: Fails if medication does not exist. |

Table 13. Role Class

| **Class Name:** | **Role** |
| --- | --- |
| Class Description/Purpose: | The Role class holds the attributes of a Role object |
| Class Modifiers: | public |
| Class Inheritance: | None |
| Class Attributes: | **Long roleID**  Description: The unique ID for a role  Data dictionary name: "role\_id"  **String title**  Description: The role's title  Data dictionary name: "role\_title"  Regex pattern for validation per RFC 3986: "^[A-Za-z0-9\\s\\-.\_~:\\/?#\\[\\]@!$&'()\*+,=]\*$"  Max length allowed: 50  **int accessLevel**  Description: The user's role level.  Data dictionary name: "role\_level"  Max value is 20  **String description**  Description: The description for this database user's role.  Data dictionary name: "role\_description"  Regex pattern for validation per RFC 3986: "^[A-Za-z0-9\\s\\-.\_~:\\/?#\\[\\]@!$&'()\*+,;=]\*$"  Max length allowed: 300 |
| Exceptions Thrown: | java.lang.Exception subclasses (No custom exceptions) |
| Class Methods: | (Standard attribute getters and setters included, but not listed for brevity)  **List<Role> getAllRoles()**  Description: This method retrieves all the roles from the database.  Returns: A list of Role objects.  Pseudocode: No inherited methods but returns null if no roles exist.  **List<Role> findRoleByKeyword(String keyword)**  Description: This method retrieves roles from the database using a keyword.  Parameter(s): keyword - a search term inputted by the user.  Return: A list of Role objects containing the keyword.  Pseudocode: No parameter manipulation or inherited methods but returns null if no role information contains the keyword.  **Role getRole(long roleID)**  Description: This method retrieves a specific role from the database.  Parameter(s): roleID - The ID number of the role.  Returns: A Role object.  Pseudocode: Fails if role does not exist.  **void addRole(String title, int accessLevel, String description)**  Description: This method adds a role to the database.  Parameter(s):   * title - The title for this database user's role. * accessLevel - The user's role level. * description - The description for this database user's role.   Pseudocode: No parameter manipulation or inherited methods but fails if any parameter fails to meet the attributes length or format constraints.  **void updateRole(Long roleID, String title, int accessLevel, String description)**  Description: This method updates role information in the database.  Parameter(s):   * title - The title for this database user's role. * accessLevel - The user's role level. * description - The description for this database user's role.   Pseudocode: No parameter manipulation or inherited methods but fails if any parameter fails to meet the attributes length or format constraints.  **void saveRole(Role role)**  Description: This method is required by Hibernate to save a role to the database.  Parameter(s): The Role object.  Pseudocode: Fails if role does not exist.  **void deleteRole(long roleID)**  Description: This method deletes a role from the database.  Parameter(s): The ID number of the role.  Pseudocode: Fails if role does not exist. |

Table 14. User Class

| **Class Name:** | **User** |
| --- | --- |
| Class Description/Purpose: | The User class holds the attributes of a User object |
| Class Modifiers: | public |
| Class Inheritance: | None |
| Class Attributes: | **Long userID**  Description: The unique ID for a user  Data dictionary name: " user\_id"  **Role role**  Description: The unique role ID for a user  Data dictionary name: "role\_id"  **String firstName**  Description: The user's first name  Data dictionary name: "first\_name"  Regex pattern for validation per RFC 3986: "^[A-Za-z0-9\\s\\-.\_~:\\/?#\\[\\]@!$&'()\*+,=]\*$"  Max length allowed: 50  **String lastName**  Description: The user's last name  Data dictionary name: "last\_name"  Regex pattern for validation per RFC 3986: "^[A-Za-z0-9\\s\\-.\_~:\\/?#\\[\\]@!$&'()\*+,=]\*$"  Max length allowed: 100  **Date DOB**  Description: The user's date of birth  Data dictionary name: "dob"  Constraint: Date must be in the past  Date format: "yyyy-MM-dd"  Max length allowed: 50  **String SSN**  Description: The user's social security number  Data dictionary name: "ssn"  Regex pattern for validation per the SSA: "^((?!000)(?!666)(?:[0-6]\\d{2}|7[0-2][0-9]|73[0-3]|7[5-6][0-9]|77[0-2]))((?!00)\\d{2})((?!0000)\\d{4})$"  Max length allowed: 9  **String streetAddress**  Description: The user's street address  Data dictionary name: "street\_address"  Regex pattern for validation per RFC 3986: "^[A-Za-z0-9\\s\\-.\_~:\\/?#\\[\\]@!$&'()\*+,=]\*$"  Max length allowed: 100  **String city**  Description: The user's city of residence  Data dictionary name: "city"  Regex pattern for validation per RFC 3986: "^[A-Za-z0-9\\s\\-.\_~:\\/?#\\[\\]@!$&'()\*+,=]\*$"  Max length allowed: 50  **String state**  Description: The user's state of residence  Data dictionary name: "us\_state"  Regex pattern for validation per RFC 3986: "^(?-i:A[LKSZRAEP]|C[AOT]|D[EC]|F[LM]|G[AU]|HI|I[ADLN]|K[SY]|LA|M[ADEHINOPST]|N[CDEHJMVY]|O[HKR]|P[ARW]|RI|S[CD]|T[NX]|UT|V[AIT]|W[AIVY])$"  Max length allowed: 2  **String zipCode**  Description: The user's zip code number  Data dictionary name: "zipcode"  Regex pattern for validation per RFC 3986: "^\\d{5}$"  Max length allowed: 5  **String phone**  Description: The user's phone number  Data dictionary name: "phone"  Regex pattern for validation per RFC 3986: "^[2-9]\\d{2}\\d{3}\\d{4}$"  Max length allowed: 10  **String insCompany**  Description: The user's medical insurance company name  Data dictionary name: "ins\_company"  Regex pattern for validation per RFC 3986: "^[A-Za-z0-9\\s\\-.\_~:\\/?#\\[\\]@!$&'()\*+,=]\*$"  Max length allowed: 50  **String insPolicyNo**  Description: The user's medical insurance policy number  Data dictionary name: "ins\_policy\_no"  Regex pattern for validation per RFC 3986: "^[A-Za-z0-9\\s\\-.\_~:\\/?#\\[\\]@!$&'()\*+,=]\*$"  Max length allowed: 50  **Boolean advDirective**  Description: Does the user have an advance directive?  Data dictionary name: "adv\_directive"  Default value: False  **String advDirType**  Description: Advance directive type  Data dictionary name: "adv\_dir\_type"  Regex pattern for validation per RFC 3986: "^[A-Za-z0-9\\s\\-.\_~:\\/?#\\[\\]@!$&'()\*+,=]\*$"  Max length allowed: 50  **String pocName**  Description: A point of contact for the user  Data dictionary name: "poc\_name"  Regex pattern for validation per RFC 3986: "^[A-Za-z0-9\\s\\-.\_~:\\/?#\\[\\]@!$&'()\*+,=]\*$"  Max length allowed: 100  **String pocPhone**  Description: The phone number for the user's point of contact  Data dictionary name: "poc\_phone"  Regex pattern for validation per RFC 3986: "^[2-9]\\d{2}\\d{3}\\d{4}$"  Max length allowed: 50  **Long userAgentID**  Description: The User ID of the user's agent  Data dictionary name: "user\_agent\_id"  **String doctorName**  Description: The user's primary care physician  Data dictionary name: "doctor\_name"  Regex pattern for validation per RFC 3986: "^[A-Za-z0-9\\s\\-.\_~:\\/?#\\[\\]@!$&'()\*+,=]\*$"  Max length allowed: 100  **String doctorPhone**  Description: The primary care physician's phone number  Data dictionary name: "doctor\_phone"  Regex pattern for validation per RFC 3986: "^[2-9]\\d{2}\\d{3}\\d{4}$  Max length allowed: 10  **String userComments**  Description: User additional comments  Data dictionary name: "user\_comments"  Regex pattern for validation per RFC 3986: "^[A-Za-z0-9\\s\\-.\_~:\\/?#\\[\\]@!$&'()\*+,=]\*$"  Max length allowed: 300  **String email**  Description: The user's email address  Data dictionary name: "email"  Regex pattern for validation per RFC 3986: "^[\w-\.]+@([\w-]+\.)+[\w-]{2,4}$"  Max length allowed: 320  **String username**  Description: The user's email address  Data dictionary name: "username"  Regex pattern for validation per RFC 3986: "^[\w-\.]+@([\w-]+\.)+[\w-]{2,4}$"  Max length allowed: 320  **String password**  Description: The generated hash value of the user' password  Data dictionary name: "password"  Regex pattern for validation per RFC 3986: "^[A-Za-z0-9\\s\\-.\_~:\\/?#\\[\\]@!$&'()\*+,=]\*$"  Max length allowed: 72 // bcrypt maximum password length is 71 characters + 1 byte null terminator  **String securityQuestion**  Description: Question enabling the user to re-set a forgotten password  Data dictionary name: "security\_question"  Regex pattern for validation per RFC 3986: "^[A-Za-z0-9\\s\\-.\_~:\\/?#\\[\\]@!$&'()\*+,=]\*$"  Max length allowed: 100  **String securityAnswer**  Description: The generated hash value of the user' security answer  Data dictionary name: "security\_answer"  Regex pattern for validation per RFC 3986: "^[A-Za-z0-9\\s\\-.\_~:\\/?#\\[\\]@!$&'()\*+,=]\*$"  Max length allowed: 72 // bcrypt maximum input length is 71 characters + 1 byte null terminator  **Date dateCreated**  Description: The creation date for this database user account  Data dictionary name: "date\_created"  Constraint: Date must be in the past  Date format: "yyyy-MM-dd'T'hh:mm:00"  Max length allowed: 50  **Date lastLoginDate**  Description: The timestamp of the user last login  Data dictionary name: "last\_login\_date"  Constraint: Date must be in the past  Date format: "yyyy-MM-dd'T'hh:mm:00"  Max length allowed: 50  **String adminComments**  Description: System administrator comments  Data dictionary name: "admin\_comments"  Regex pattern for validation per RFC 3986: "^[A-Za-z0-9\\s\\-.\_~:\\/?#\\[\\]@!$&'()\*+,=]\*$"  Max length allowed: 300  **Boolean isActive**  Description: Is the user's account active?  Data dictionary name: "is\_active"  Default value: False  **Boolean isLocked**  Description: Is the user's account locked?  Data dictionary name: "is\_locked"  Default value: False |
| Exceptions Thrown: | java.lang.Exception subclasses (No custom exceptions) |
| Class Methods: | (Standard attribute getters and setters included, but not listed for brevity)  **List<User> getAllUsers()**  Description: This method retrieves all the users from the database.  Returns: A list of User objects.  Pseudocode: No inherited methods but returns null if no users exist.  **List<User> findUserByKeyword(String keyword)**  Description: This method retrieves users from the database using a keyword.  Parameter(s): keyword - a search term inputted by the user.  Return: A list of User objects containing the keyword.  Pseudocode: No parameter manipulation or inherited methods but returns null if no user information contains the keyword.  **User getUser(long UserID)**  Description: This method retrieves a specific user from the database.  Parameter(s): userID - The ID number of the user.  Returns: A User object.  Pseudocode: Fails if user does not exist.  **void addUser(long roleID, String firstName, String lastName, Date DOB, String SSN, String streetAddress, String city, String state, String zipCode, String phone, String email, String insCompany, String insPolicyNo, Boolean advDirective, String advDirType, String pocName, String pocPhone, Long userAgentID, String doctorName, String doctorPhone, String userComments, String username, String password, String securityQuestion, String securityAnswer, Date dateCreated, Date lastLoginDate, String adminComments, Boolean isActive, Boolean isLocked)**  Description: This method adds a user to the database.  Parameter(s):   * roleID - The role id of the user. * firstName - The user's first name. * lastName - The user's last name. * DOB - The user's date of birth. * SSN - The user's social security number * streetAddress - The user's street address. * city - The user's city of residence. * state - The user's state of residence. * zipCode - The user's zip code number * phone - The user's phone number. * email - The user's email address. * insCompany - The user's medical insurance company name. * insPolicyNo - The user's medical insurance policy number. * advDirective - Does the user have an advance directive? * advDirType - Advance directive type. * pocName - A point of contact for the user * pocPhone - The phone number for the user's point of contact * userAgentID - The User ID of the user's agent * doctorName - The user's primary care physician. * doctorPhone - The primary care physician's phone number. * userComments - User additional comments. * username - The user's email address. * password - The generated hash value of the user's password. * securityQuestion - Question enabling the user to re-set a forgotten password. * securityAnswer - The generated hash value of the user's security answer. * dateCreated - The creation date for this database user account. * lastLoginDate - The timestamp of the user last login * adminComments - System administrator comments. * isActive - Is the user's account active? * isLocked - Is the user's account locked?   Pseudocode: No parameter manipulation or inherited methods but fails if any parameter fails to meet the attributes length or format constraints.  **void updateUser(Long UserID long roleID, String firstName, String lastName, Date DOB, String SSN, String streetAddress, String city, String state, String zipCode, String phone, String email, String insCompany, String insPolicyNo, Boolean advDirective, String advDirType, String pocName, String pocPhone, Long userAgentID, String doctorName, String doctorPhone, String userComments, String username, String password, String securityQuestion, String securityAnswer, Date dateCreated, Date lastLoginDate, String adminComments, Boolean isActive, Boolean isLocked)**  Description: This method updates user information in the database.  Parameter(s):   * userID - The unique ID for a user. * roleID - The role id of the user. * firstName - The user's first name. * lastName - The user's last name. * DOB - The user's date of birth. * SSN - The user's social security number * streetAddress - The user's street address. * city - The user's city of residence. * state - The user's state of residence. * zipCode - The user's zip code number * phone - The user's phone number. * email - The user's email address. * insCompany - The user's medical insurance company name. * insPolicyNo - The user's medical insurance policy number. * advDirective - Does the user have an advance directive? * advDirType - Advance directive type. * pocName - A point of contact for the user * pocPhone - The phone number for the user's point of contact * userAgentID - The User ID of the user's agent * doctorName - The user's primary care physician. * doctorPhone - The primary care physician's phone number. * userComments - User additional comments. * username - The user's email address. * password - The generated hash value of the user's password. * securityQuestion - Question enabling the user to re-set a forgotten password. * securityAnswer - The generated hash value of the user's security answer. * dateCreated - The creation date for this database user account. * lastLoginDate - The timestamp of the user last login * adminComments - System administrator comments. * isActive - Is the user's account active? * isLocked - Is the user's account locked?   Pseudocode: No parameter manipulation or inherited methods but fails if any parameter fails to meet the attributes length or format constraints.  **void saveUser(User User)**  Description: This method is required by Hibernate to save a user to the database.  Parameter(s): The User object.  Pseudocode: Fails if user does not exist.  **void deleteUser(long UserID)**  Description: This method deletes a user from the database.  Parameter(s): The ID number of the user.  Pseudocode: Fails if user does not exist. |

# Human Interface Design

## Overview of User Interface

The user interface for the organizer:

1. Accepts input from the user
2. Validates the format of the input
3. Forwards the input to the controller for processing
4. Receives directions and information from the controller
5. Executes the directions of the controller
6. Presents the information to the user

For development, the authors are starting with a bare-bones web interface. Once the authors test and verify the organizer's components, the authors will assign a team to front-end development. However, the user interface currently allows the user to complete all the tasks required by the client. In addition, as stated earlier, the user interface performs no logic or data manipulation except to format information for the user.

## Screen Images and Actions

Please refer to Appendix A, Graphical User Interface Design, of the Software Requirements Specification (SRS).

# Requirement Matrix

Table 15. Requirement Matrix

| **Requirement \_ID** | **Requirement** | **Component** | **Data Structure** |
| --- | --- | --- | --- |
| REQ-1.3 | As an unauthenticated and unauthorized user, I want access to a secure Registration page, so I can enroll in the Letter of Life program. | addUser() | User |
| REQ-1.5 | As a registered, authenticated, and authorized Program Participant, I want to add my LOL information to the database via an online form, so medical personnel can have access to my medical history in case of an emergency. | updateUser() | User |
| REQ-1.6 | As a registered, authenticated, and authorized Program Participant, I want to view my LOL information via an online form, so I can validate my information or print my LOL. | getUser() | User |
| REQ-1.7 | As a registered, authenticated, and authorized Program Participant, I want to edit my LOL information via an online form, so I can ensure my medical history is up to date. | updateUser() | User |
| REQ-1.8 | As a registered, authenticated, and authorized Program Participant, I want to delete my LOL information from the database via an online form, so I may stop participating in the program. | deleteUser() | User |
| REQ-1.9 | As a registered, authenticated, and authorized Program Participant, I want to print a barcode or quick response (QR) code with a link to my LOL (which I may affix to a bracelet, card, or necklace), so medical personnel can quickly access my medical history in case of an emergency. | getUser() | User |
| REQ-1.12 | As a registered, authenticated, and authorized Medical Services Provider, I want to edit my profile information via an online form, so I can ensure my information is up to date. | updateUser() | User |
| REQ-1.13 | As a registered, authenticated, and authorized Medical Services Provider, I want to search for patients on a laptop or mobile device via an online form, so I can retrieve their medical history and provide correct medical care. | getAllUsers()  findUserByKeyword() | List<User>  List<User> |
| REQ-1.14 | As a registered, authenticated, and authorized Medical Services Provider, I want to search for a patient by scanning a barcode or quick response (QR) code, using a mobile device or authorized accessory, so I can retrieve their medical history and provide correct medical care. | getUser() | User |
| REQ-1.15 | As a registered, authenticated, and authorized Medical Services Provider, I want to view a patient's medical history on a laptop or mobile device via an online form, so I can provide correct medical care. | getUser() | User |
| REQ-1.23 | As a registered, authenticated, and authorized Patient Advocate, I want to edit my profile information via an online form, so I can ensure my information is up to date. | updateUser() | User |
| REQ-1.20 | As a registered, authenticated, and authorized Patient Advocate, I want to search for patients on a laptop or mobile device via an online form, so I can view, edit, and print patient information, as necessary. | getAllUsers()  findUserByKeyword() | List<User>  List<User> |
| REQ-1.21 | As a registered, authenticated, and authorized Patient Advocate, I want to search for a patient by scanning a barcode or quick response (QR) code, using a mobile device or authorized accessory, so I can view, edit, and print patient information, as necessary. | getUser() | User |
| REQ-1.22 | As a registered, authenticated, and authorized Patient Advocate, I want to view the LOL information of my patients via an online form, so I can validate their information or print their LOL. | getUser() | User |
| REQ-1.23 | As a registered, authenticated, and authorized Patient Advocate, I want to edit the LOL information of my patients via an online form, so I can ensure their medical history is up to date. | updateUser() | User |
| REQ-1.32 | As a registered, authenticated, and authorized System Administrator, I want to search for user accounts on a laptop or mobile device via an online form, so I can add, view, edit, and delete user accounts, as necessary. | getAllUsers()  findUserByKeyword() | List<User>  List<User> |
| REQ-1.28 | As a registered, authenticated, and authorized System Administrator, I want to add accounts for Medical Services Providers and Patient Advocates, so I can limit non-patient access to patient information. | addUser() | User |
| REQ-1.29 | As a registered, authenticated, and authorized System Administrator, I want to view accounts for Medical Services Providers and Patient Advocates, so I can validate their information. | getUser() | User |
| REQ-1.30 | As a registered, authenticated, and authorized System Administrator, I want to edit accounts for Medical Services Providers and Patient Advocates, so I can ensure their information is up to date. | updateUser() | User |
| REQ-1.31 | As a registered, authenticated, and authorized System Administrator, I want to delete accounts for Medical Services Providers and Patient Advocates, so I can limit non-patient access to patient information. | deleteUser() | User |
| REQ-1.32 | As a registered, authenticated, and authorized System Administrator, I want to search for patients on a laptop or mobile device via an online form, so I can add, view, edit, and delete patient information, as necessary. | getAllUsers()  findUserByKeyword() | List<User>  List<User> |
| REQ-1.28 | As a registered, authenticated, and authorized System Administrator, I want to add accounts for patients, so I can enroll them in the program. | addUser() | User |
| REQ-1.34 | As a registered, authenticated, and authorized System Administrator, I want to view accounts for patients, so I can validate their information. | getUser() | User |
| REQ-1.35 | As a registered, authenticated, and authorized System Administrator, I want to edit accounts for patients, so I can ensure their information is up to date. | updateUser() | User |
| REQ-1.36 | As a registered, authenticated, and authorized System Administrator, I want to delete accounts for patients, so I can disenroll them from the program. | deleteUser() | User |