CV Tool

Analysis and Design Document

Student:Lăzăreanu Sabina

**Group:30233**

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Description** | **Author** |
| 23/04/2019 | 1.0 | First Version | Lăzăreanu Sabina |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table of Contents

I. Project Specification 4

II. Elaboration – Iteration 1.1 4

1. Domain Model 4

2. Architectural Design 4

2.1 Conceptual Architecture 4

2.2 Package Design 4

2.3 Component and Deployment Diagrams 4

III. Elaboration – Iteration 1.2 4

1. Design Model 4

1.1 Dynamic Behavior 4

1.2 Class Design 4

2. Data Model 4

3. Unit Testing 4

IV. Elaboration – Iteration 2 4

1. Architectural Design Refinement 4

2. Design Model Refinement 4

V. Construction and Transition 5

1. System Testing 5

2. Future improvements 5

VI. Bibliography 5

# Project Specification

This is a web project than is helpful for those persons how want to make an easy and clean CV just by filling some fields and asking some questions.

This system has two types of users:

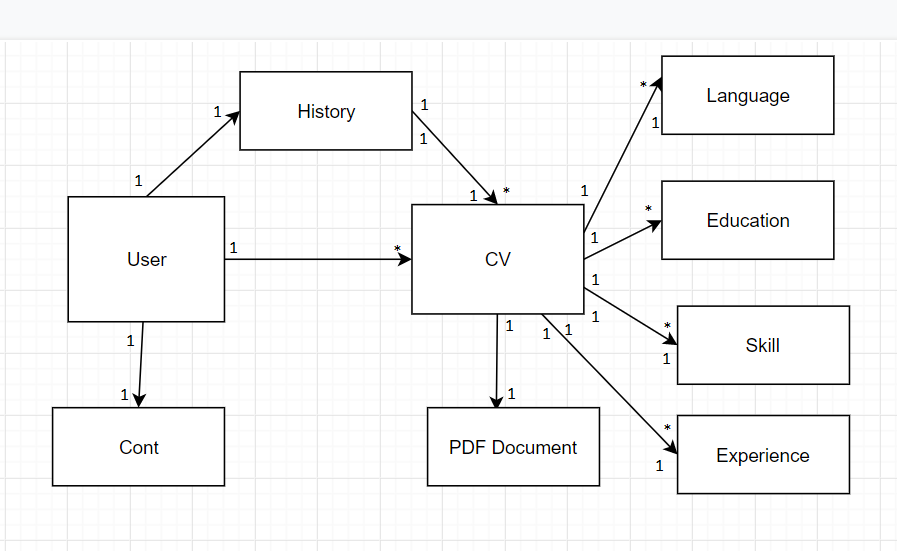
* Regular User that can create one type of CV or update an existing one
* Premium user that can choose witch type of CV want to create and can also to edit an existing CV

The generated CV will be sent on email and can be downloaded.

\*write about templates

# Elaboration – Iteration 1.1

# Domain Model



# Architectural Design

## Conceptual Architecture

This app design is based on *Layered Architecture Design Pattern an also on Model-View-Controller Designs*. It is the most common architecture pattern, otherwise known as the n-tier architecture pattern. This pattern is the de facto standard for most Java EE applications and therefore is widely known by most architects, designers, and developers. The layered architecture pattern closely matches the traditional IT communication and organizational structures found in most companies, making it a natural choice for most business application development efforts.

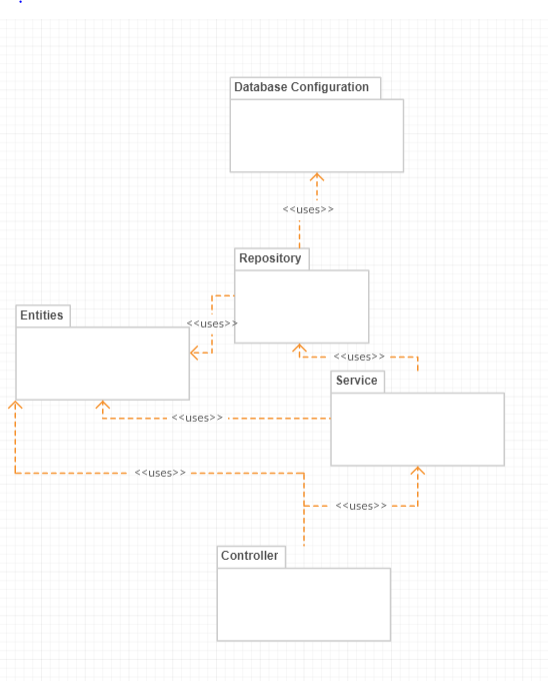
Components within the layered architecture pattern are organized into horizontal layers, each layer performing a specific role within the application (e.g., presentation logic or business logic). Although the layered architecture pattern does not specify the number and types of layers that must exist in the pattern, most layered architectures consist of four standard layers: presentation, business, persistence, and database.

Each layer of the layered architecture pattern has a specific role and responsibility within the application. For example, a presentation layer would be responsible for handling all user interface and browser communication logic, whereas a business layer would be responsible for executing specific business rules associated with the request.

This three layers are arranged on 2 tiers.

In the same time this project have a Server-Client structure, Java project being the Server component and the Angular is the Client Component.

## Package Design

o

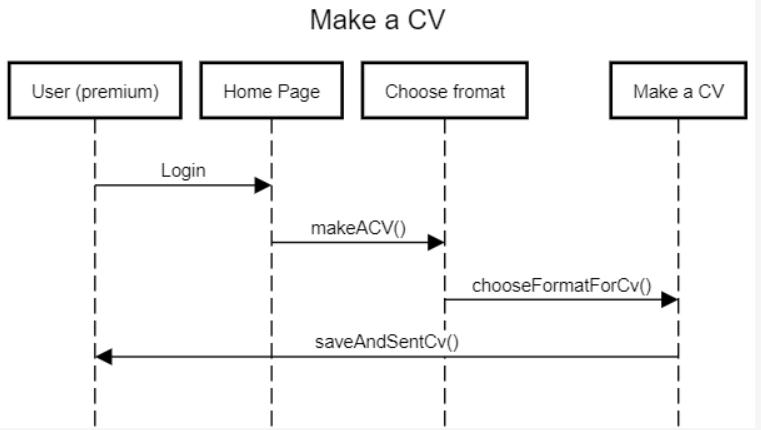
## Component and Deployment Diagrams

# 

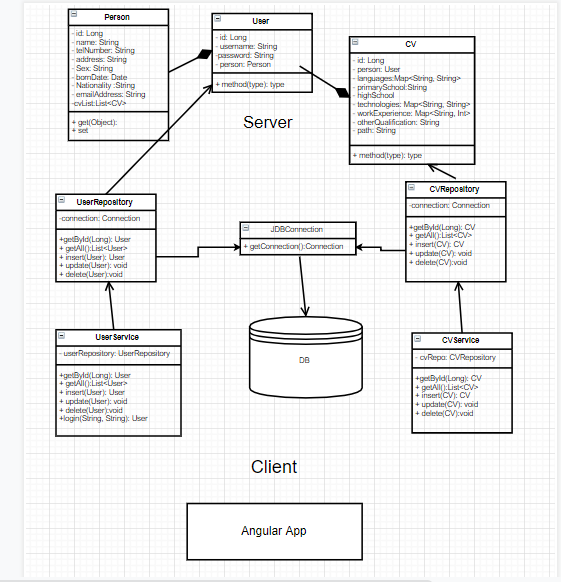
# Elaboration – Iteration 1.2

# Design Model

## Dynamic Behavior

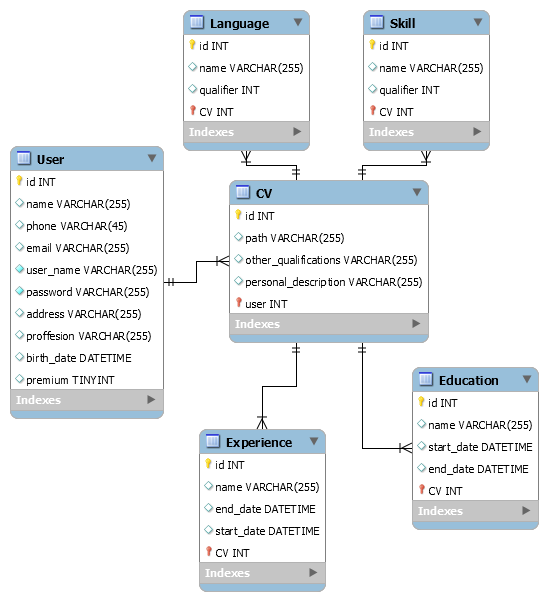


## Class Design



* **Builder** **Design Pattern**. The builder pattern is used to create complex objects with constituent parts that must be created in the same order or using a specific algorithm. An external class controls the construction algorithm.

# Data Model



# Unit Testing

A unit test should test functionality in isolation. Side effects from other classes or the system should be eliminated for a unit test, if possible.

This can be done via using test replacements (*test doubles*) for the real dependencies. Test doubles can be classified like the following:

* A *dummy object* is passed around but never used, i.e., its methods are never called. Such an object can for example be used to fill the parameter list of a method.
* *Fake* objects have working implementations, but are usually simplified. For example, they use an in memory database and not a real database.
* A *stub* class is an partial implementation for an interface or class with the purpose of using an instance of this stub class during testing. Stubs usually don’t respond to anything outside what’s programmed in for the test. Stubs may also record information about calls.
* A *mock object* is a dummy implementation for an interface or a class in which you define the output of certain method calls. Mock objects are configured to perform a certain behavior during a test. They typically record the interaction with the system and tests can validate that.

Test doubles can be passed to other objects which are tested. Your tests can validate that the class reacts correctly during tests. For example, you can validate if certain methods on the mock object were called. This helps to ensure that you only test the class while running tests and that your tests are not affected by any side effects.

Test Case Scenarios:

First scenario that will be tested using Mockito Junit Testing is login, testing best case scenario and also worst case scenario.

Second scenario that will be tested is signUp for regulary users and also creating a CV.

# Elaboration – Iteration 2

# Architectural Design Refinement

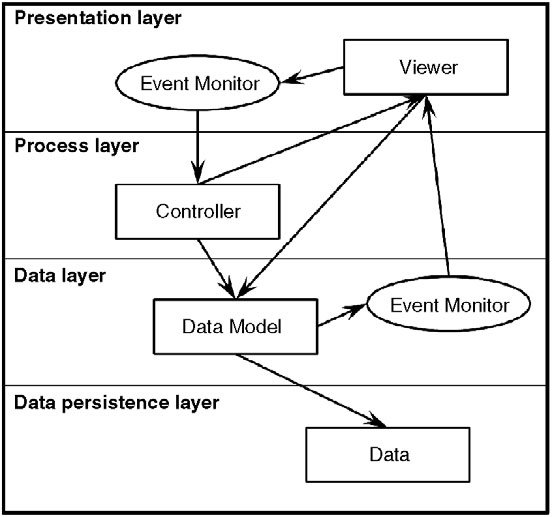
Model view controller (MVC) is an architectural pattern usually used in web-based applications. It provides three main layers; model, view, and controller. Many developers use MVC as a standard design pattern. It is a complete framework. MVC provide three types of classes:

A. Model- Model classes are used to implement the logic of data domains. These classes are used to retrieve, insert or update the data into the database associated with our application.

B. View- Views are used to prepare the interface of our application. By using that interface users interact with our application.

C. Controller- Controller classes are used to respond to the user’s requests. Controller classes perform the users requested actions. These classes work with model classes and select the appropriate view that should be displayed to the user according to user requests.

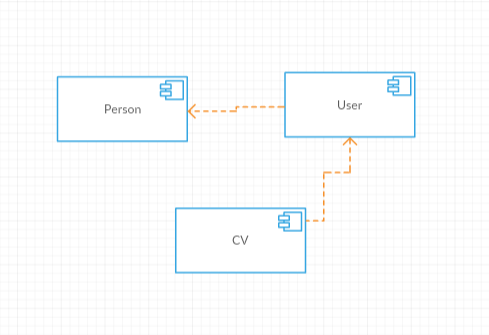
MVC pattern architecture is basically a three-layered architecture. It separates the characteristics of application. Its first layer is related to the user input logic, second layer is related to the business logic and third layer is used to implement user interface logic. MVC provide very loose coupling among these three layers. MVC pattern are used to specify the location of each logic in application . MVC patterns provide the facility of parallel development. It means that each layer of the application independent of each other i.e. three developer can work on the single application simultaneously . One developer will be working on user input logic (controller logic), other developer will be working on the user interface logic (view) and third developer will be working on the business logic (model) at the same time. Second section of our research paper will be explained about when we can use the MVC pattern architecture. In this section we will also describe about some advantages of using MVC pattern architecture. In third section of the of our research paper we will describe the features of MVC framework.



Components within the layered architecture pattern are organized into horizontal layers, each layer performing a specific role within the application (e.g., presentation logic or business logic). Although the layered architecture pattern does not specify the number and types of layers that must exist in the pattern, most layered architectures consist of four standard layers: presentation, business, persistence, and database (Figure 1-1). In some cases, the business layer and persistence layer are combined into a single business layer, particularly when the persistence logic (e.g., SQL or HSQL) is embedded within the business layer components. Thus, smaller applications may have only three layers, whereas larger and more complex business applications may contain five or more layers.

Each layer of the layered architecture pattern has a specific role and responsibility within the application. For example, a presentation layer would be responsible for handling all user interface and browser communication logic, whereas a business layer would be responsible for executing specific business rules associated with the request.





# Design Model Refinement

General Responsibility Assignment Software Patterns (or Principles), abbreviated GRASP, consist of guidelines for assigning responsibility to classes and objects in object-oriented design.[1] It is not related to the SOLID design principle.

**High cohesion** is an evaluative pattern that attempts to keep objects appropriately focused, manageable and understandable. High cohesion is generally used in support of low coupling. High cohesion means that the responsibilities of a given element are strongly related and highly focused. Breaking programs into classes and subsystems is an example of activities that increase the cohesive properties of a system. Alternatively, low cohesion is a situation in which a given element has too many unrelated responsibilities. Elements with low cohesion often suffer from being hard to comprehend, reuse, maintain and change. In my UML diagram we can easily see that this principle is respected because each class has it’s own responsibility.

Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements. **Low coupling** is an evaluative pattern that dictates how to assign responsibilities to support

* lower dependency between the classes,
* change in one class having lower impact on other classes,
* higher reuse potential.

# Construction and Transition

# System Testing

The testing is going to be done using the mokito technique, which controls the application flow and creates objects that fit the requirements.

A unit test should test functionality in isolation. Side effects from other classes or the system should be eliminated for a unit test, if possible. That is why a choose to use MOKITO for making may JUnit Tests.

I create mock objects manually (via code) and use a mock framework to simulate these classes. Mock frameworks allow you to create mock objects at runtime and define their behavior. [6]

I also choose to test my application manually and trying to take care of each use case.

# Future improvements

This kind of project can have a lot of future improvements like integrating different kind of CV formats with different fonts and so on.

Also the quality of the typeScript code can be improved.

*[Present future improvements for the system]*

# Bibliography

1. <https://www.vogella.com/tutorials/Mockito/article.html>