Vocal Maturity Coding (VMC) Application Design Document

03/15/2021

Prepared by Alex Gheith

email: gheith@purdue.edu

VMC Application Design Document

# Introduction

This document describes the VMC application design along with the description of the files and folders included in the codebase. It is intended for future software developers who will be tasked with maintaining and evolving the application.

# Architecture

The application is build using Python 3.7, the Qt 5 framework, and the SQL-Alchemy over a PostgreSQL database hosted on the Amazon AWS. It follows the design of N-tier software architecture, as shown in figure 1.

**Presentation Layer**

**Core Logic Layer**

**Data Access Layer**

Front layer is responsible for “presenting” the data to the user AND acquiring/managing the user input.

The Logic layer performs requirement management and adds some wrapping logic.

**Data Models**

Data Models are common objects across all layers.

Backend layer is responsible for accessing data stores, and outside resources, mostly, the database.

**Database**

Figure 1: VMC Software Acrchitecture

In the coming sections, we will describe the design of each of these layers, as well as the database and the cloud storage.

## The Presentation (GUI) Layer

There are two UI components in the application: a Login UI and a Coding UI, both of which are created using simple forms from the Qt Designer and are styled using a modified version of the QDarkStyle library. Figure 2 below shows the UI forms with and without styling.

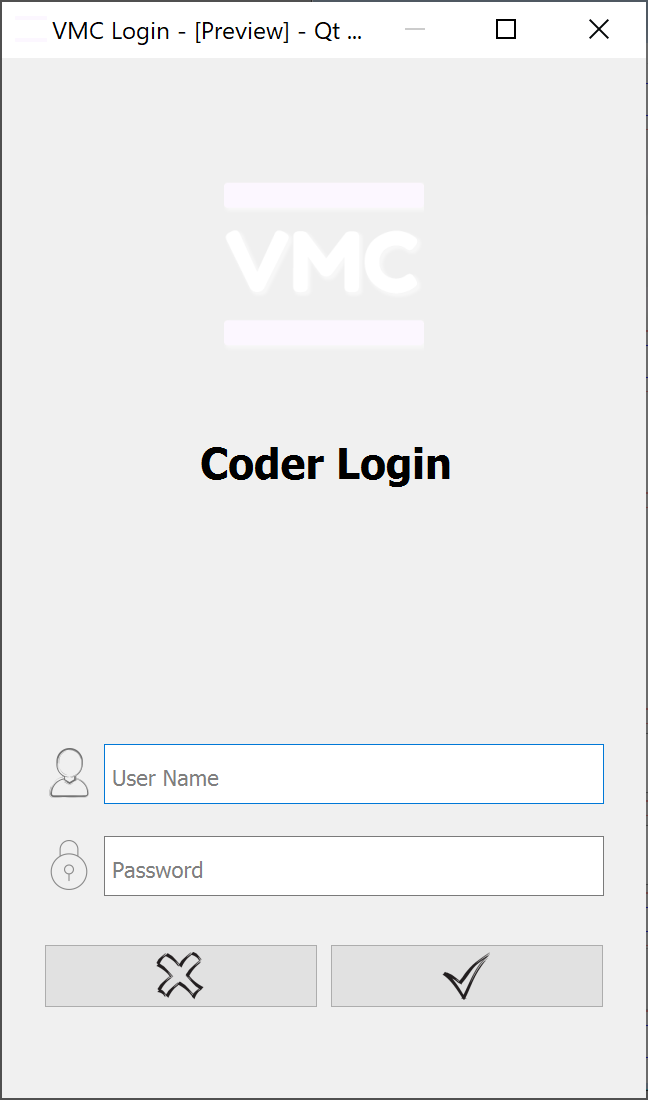
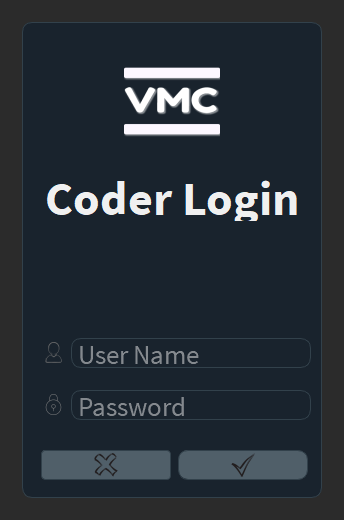
 

Figure 2: The UI Window with (Right) and without (Left) styling.

Both UI forms have only two functionality types: managing UI events and communicating with the database through the Core Logic Layer. It is \*very\* important to note that the forms do not connect directly to the Data Access. They must go through the Core Logic to maintain architectural integrity. We opted not to use the Model-View-Controller (MVC) Pattern, as the list of UI functions is very short.

## The Core Logic Layer

The Core Logic Layer follows the Command Pattern, which aims to hide the complexity of database interactions, and adds some wrapping logic to handle exceptions and logging to the commands. Note that this layer in its current form might be redundant. However, it is intended to simplify managing the growth of functionality of the application as time goes on.

In this layer, the design uses three types of classes:

* A Request class: This class contains one or more nested classes, each of which contains the relevant properties that pertain to that specific request. These properties are the data elements needed to pass on through the Core Logic to fulfill the actions of the command. These properties are populated by the invoking class from the Presentation Layer.
* A Response class: This class contains one or more nested classes, each of which contains the relevant properties that pertain to that specific response. These properties are the results of executing the command, upon successful execution, or the error that occurred from the execution attempt. This class is intended to handle DB errors and prevent their propagation to the Presentation Layer.
* A Command class: This class contains one or more "execute" methods. Each method must take in an instance of a "request" class and returns an instance of a "response" class.

Note that all of these classes are children of their respective parent classes, where common code is expected to reside to avoid code duplication and centralize logic. Also, note that this layer uses the Data Models to pass data back and forth.

## The Data Access Layer

The Data Access Layer utilizes the Repository Pattern to communicate with the database using the SQL-Alchemy ORM (Object Relational Mapper) approach. This design avoids direct queries for a myriad of reasons, including better security, easier extensibility, and better maintenance. All repositories must be inherited from the Base Repository and must \*not\* propagate the ORM's database types to the Core Logic but instead use Data Models to send and receive data.

Note that, ideally speaking, each database table should have its own repository. However, we did not need to do that as UI functionality was only required for some of the tables.

## The Data Models

The Data Models are simple classes that only house properties or attributes. They are intended to standardize the types of objects being communicated across the layers, which eliminates the need to reference the Data Access Layer and its DB-only types.

All of the Models are built using the Python "attrs" library, which offers an excellent and consistent way of representing property-based classes.

## The PostgreSQL Database

The database used in this application is a PostgreSQL database hosted on Amazon Web Services (AWS) Relational Database Service (RDS). The PostgreSQL engine is a much better choice than the common MySQL engine as it is better supported and has better enterprise adoption.

In addition to defining the relations between tables and the constraints needed for some columns, each table must include the following columns:

* An "ID" column that acts as the primary key.
* An "AddedOn" that captures the timestamp of the row addition.
* A "ModifiedOn" that captures the timestamp of the row modification.

The current DB design has evolved and is expected to require more changes and additions. The current design does not support coding versioning, for example, which will require an extensive modification to several tables. Another feature worthy of consideration is the addition of "IsDeleted" flag that allows for soft deletes.

## The Amazon S3 Cloud Storage

The VMC application performs a remote update via checking the Amazon S3 Storage for the latest version at startup. If a newer version exists, it is downloaded and compiled on the user’s machine. The structure of the S3 storage is as follows:

**ndd-family-lab/ # Bucket Name**

**VMC/ # VMC Application Folder**

**1.2.0/ # Release 1.2.0 Files**

**1.3.0/ # Release 1.3.0 Files**

**. . . # Next release goes here.**

**latest.config # Contains latest release info.**

There is only one folder, "**VMC**", under the bucket "**ndd-family-lab**" that contains all the releases, which allows for adding more lab-related applications down the line. Furthermore, the use of the config file allows for uploading different releases and controlling which one should be used.

# Folders and Files

This section will go over the files and folders in the application codebase and briefly describe their content and purpose.

## File: **VmcLoader.py**

This file is the entry point of the Coding Application. As the application starts, it checks the S3 Cloud storage for the latest version, and it downloads it if needed, at which point it needs to restart. Note that this file also holds information about the LENA data folder and the database connection string, which means it needs to be referenced by all standalone scripts that would require that information.

**NOTE:** We recommend extracting these elements into a separate file to avoid that dependency and have cleaner architecture.

## File: **CloudUpdater.py**

This file houses the code that checks for and downloads any new version of the application in the Amazon cloud. It is invoked at the beginning of each run, and it is also used to perform an initial application download at the first deployment.

## File: **AddParticipantsToDB.py**

This file is a standalone administrative script that checks the file "Participant.csv" from the Data Folder and adds the new participants (subject children) to the database. It should be run first before running the script that adds new recordings to confirm that no new participants are missing from the database, which will stop the addition of any dependent recordings.

## File: **AddRecordingsToDB.py**

This file is also a standalone administrative script that checks the file "Recording.csv" from the Data Folder and adds new recordings to the database. While the recording audio is checked for existence, this script does not load that file just yet. Only the segments and the ITS file are read and stored in the database.

## File: **DirectAccess.py**

This is the main standalone administrative script that contains all administrative functions. Before any function is run, you need to uncomment that function then comment it out after completion. This is very important to avoid running more than one function at a time.

## File: **UploadToCloud.py**

This file is a standalone administrative script that uploads a pre-defined list of files to the cloud storage under a defined folder name.

NOTE:

\* If the list of files in the application changes, the defined list in the script must be updated.

\* You must update the file "release.config" to the latest desired release value \*before\* using the script.

## File: **log.config**

This file contains the settings needed for all logging used in the application. It will rarely change over the lifetime of the application as new logging policies are not often introduced.

## File: **release.config**

This file contains the release version of the latest application deployment. It should be updated with every new release.

## File: **cloud.config**

This file contains the cloud credentials used to both upload new releases or download them. You will need to have these credentials assigned to you before you run the application or the scripts.

## File: **MasterStyleSheet.py**

While a Python file, it only contains the updated styling information for the application, stored in a string.

## File: **requirements.txt**

This file contains a list of the required libraries for the application. It uses a standard format that allows for an easy update to the virtual environment.

## Folder: **Presentation**

This folder contains the presentation layer components of the application. There are two "<FormName>.ui" files that contain the GUI layout. These files are converted to "<FormName>Base.py" files using the **pyuic5** tool, which are then extended by inheritance into "<FormName>.py" file where the behavior implementation resides.

The resources are converted into a Python file using the **pyrcc5** tool to bypass all dependent images' need to exist alongside the code files. Please review the “readme” file in the root folder for how the tools are used.

## Folder: **CoreLogic**

This folder contains the middle layer components of the application. The architecture requires the presentation, or the GUI, layer to go through the logic layer to communicate with the database. As mentioned before, this layer follows the Command Pattern in implementing its components and requires all new commands to inherit from the BaseCommand class.

## Folder: **DataAccess**

This folder contains the classes that communicate with the database. First, the database tables’ classes are autogenerated using the SQL Alchemy Code Generator **sqlacodegen** tool and saved in the “BaseDB.py” file. This file must match the database schema design, so it must be updated after any design change, or the code will break.

This layer uses the Repository Pattern, which creates a repository class for every main database table. Note that we do not strictly follow this pattern as the interaction with the database is not complex in this application.

While it should be self-contained, this layer breaks this rule and requires access to the hard drive when adding utterances to the database to be encoded. This only happens from the administrative script access, but we recommend extracting that code from this layer to keep it clean.

## Folder: **Models**

This folder contains the data objects (objects that only contain data attributes) that are used for communicating between the layers. This is very important as it allows for layers to be self-contained and not require references from either the invoking or the invoked layer.

All data models utilize the **attrs** library, which offers many enhancements to simple Python classes or slots.

## Folder: **Database**

This folder is not directly a part of the VMC application. It contains two PostgreSQL scripts:

1. The script that creates the database schema, which must be updated with any design changes.
2. A script that contains a list of maintenance and administrative queries.

A database IDE, e.g., DBeaver, is required to use these scripts.

## Folder: **Support**

This folder contains the documentation associated with the VMC application. The subfolder **Deployment** is the folder needed to perform a new application deployment to a new machine. Remember to update these files if you modify the source files. The files in this folder are:

* **cloud.config**: An empty config file that coders must populate with their credentials.
* **GetAppFromCloud.py**: This is a copy of the “CloudUpdater.py” file, but renamed.
* **requirements-PC.txt**: This is a copy of the “requirements.txt” file from the root folder.
* **requirements-Mac.txt**: This is a copy of the “requirements.txt” file from the root folder but has one modification needed for one of the libraries on Mac machines.

## Folder: **Legacy**

This folder contains the code files used to add the information from the legacy system into the current database. That code was only used once and is not expected to be needed again. It is only included here for future investigations or code improvements.