System Maintenance

Environment

Software

During the development of this system I used various software tools to create the system, the names and versions of these are listed below:

- Python 3.4
- Atom 1.4.1
- PyQt 4
- Sqlite 3
- DB Browser for SQLite
- git 2.5.0 & GitHub
- LibreOffice 5.0.5.2

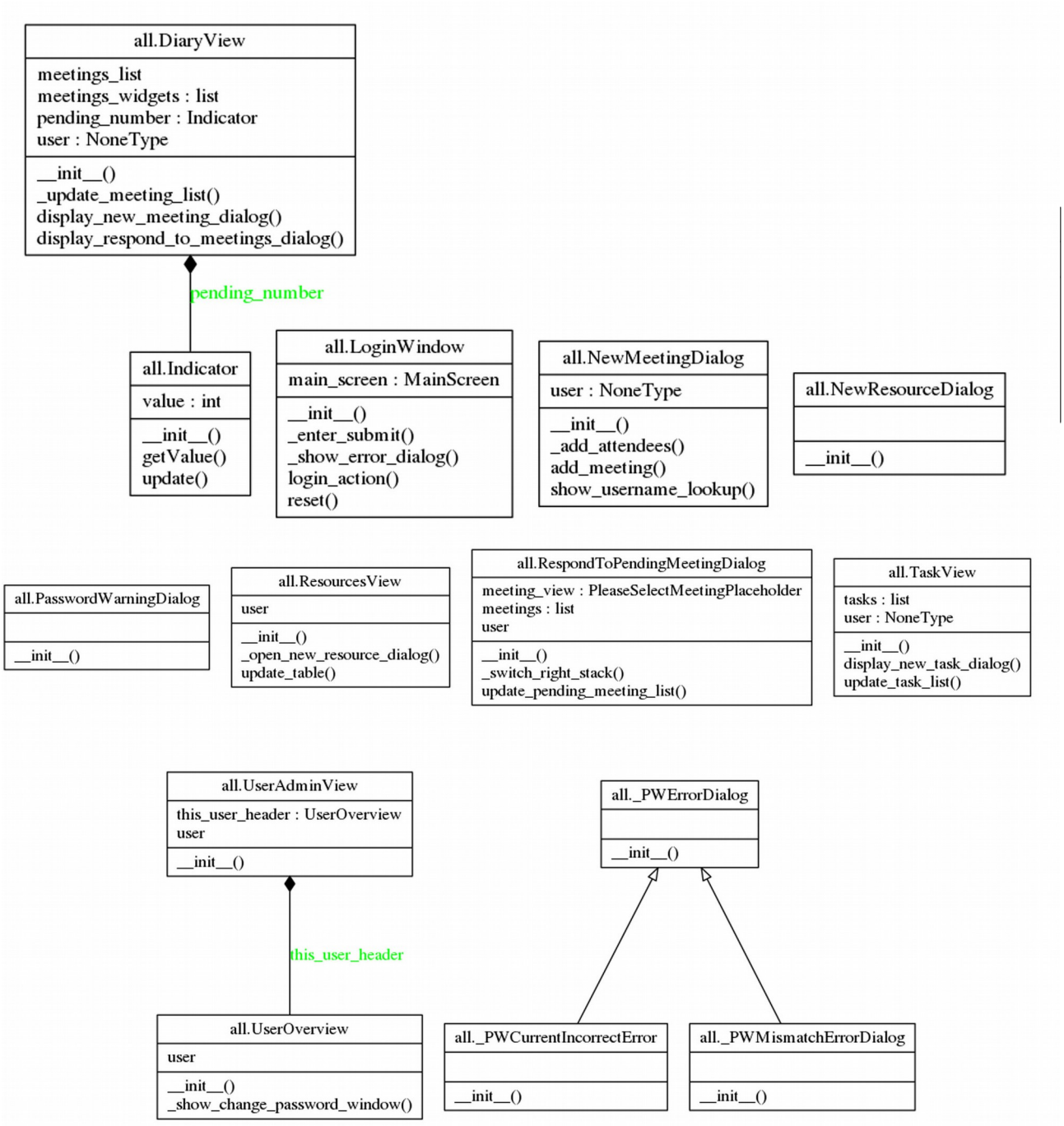
Usage Explanation

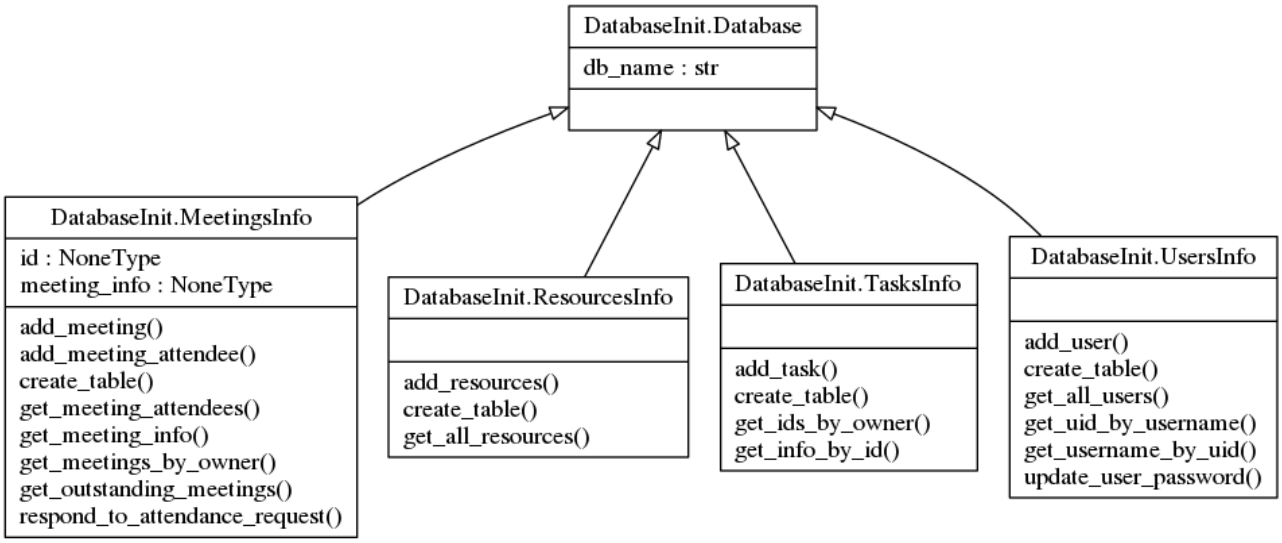
Below are the explanations for the software I have used, all of the software I have used is either open sourced and/or freely available at no cost for developer or client.

Software	**Usage**
Python 3.4	Python is the underlying language that the software is written in. Version 3.4 is the latest stable version and contains the best balance of security, features and stability of all currently available versions.
Atom 1.4.1	Atom by GitHub is the text editor that I chose to use throughout the project because of it's customizable syntax highlighting and the various extensions that it includes to make writing code easier.
PyQt 4	The software library for defining graphical user interfaces and converting python into an event driven language.
Sqlite 3	Support for this standard is included with the python language and is used for all the database interactions in this project.
DB Browser for SQLite	This is not required for the client's system but it is required for testing the validity of the data held in the database.
Git 2.5.0 & GitHub	These are not strictly required by the user, I use them for version control and synchronisation of code and resources however they could be used as a way to distribute updates.
LibreOffice 5.0.5.2	For the production of the documentation

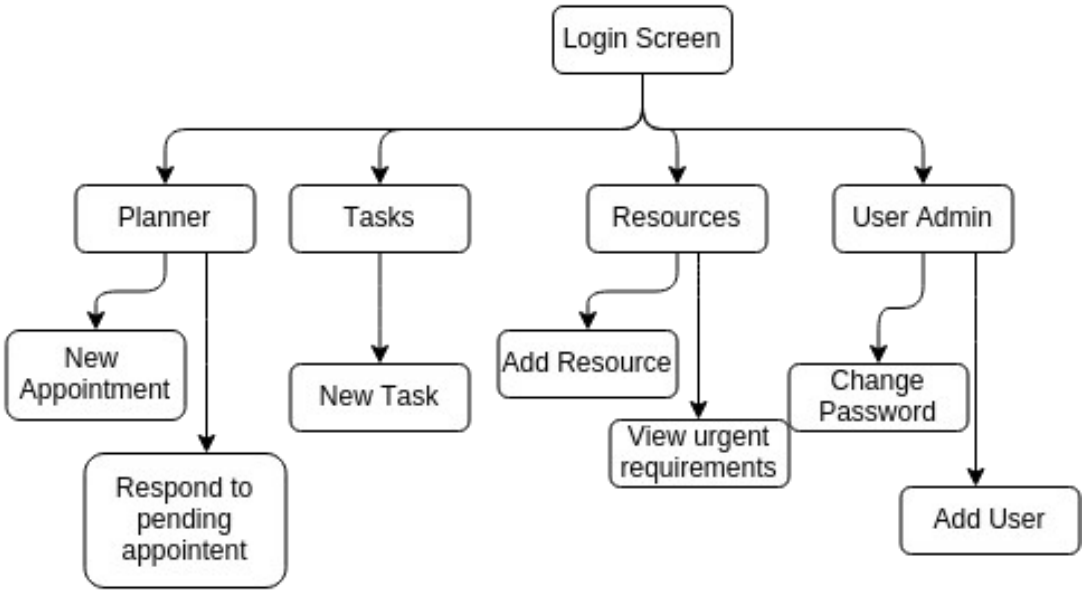
System Overview

Class Diagrams




DatabaseInit.Database
db_name : str
DatabaseInit.MeetingsInfo
id : NoneType
meeting_info : NoneType
add_meeting()
add_meeting_attendee()
create_table()
get_meeting_attendees()
get_meeting_info()
get_meetings_by_owner()
get_outstanding_meetings()
respond_to_attendance_request()
DatabaseInit.ResourcesInfo
add_resources()
create_table()
get_all_resources()
DatabaseInit.TasksInfo
add_task()
create_table()
get_ids_by_owner()
get_info_by_id()
DatabaseInit.UsersInfo
add_user()
create_table()
get_all_users()
get_uid_by_username()
get_username_by_uid()
update_user_password()

System Navigation



Code Structure

The class shown below is the 'Database' class which is the derivative class for all the database interactions from the file “DatabaseInit.py”.

```
#This tidies all of the SQL queries into another name space.
import SqlDictionary
import random
import hashlib
import sqlite3

def gen_pw_hash(password):
    phash = hashlib.md5()
    phash.update(bytes(password, "UTF-8"))
    return phash.hexdigest()

class Database:
    """This is the general database wrapper that I'll use throughout the system"""
    def __init__(self, child, database_name="cmsdb.db"):
        #print("[INFO] Created database object")

        self.db_name = database_name

    def _connect_and_execute(self, sql="", database_name=None):
        #print(sql)
        if database_name == None:
            database_name = self.db_name

        with sqlite3.connect(self.db_name) as dbcon:
            cursor = dbcon.cursor()
            cursor.execute(sql)
            results = cursor.fetchall()
        return results
```

This code is structured as a class so the more specific classes within the code can inherit it's properties. This prevents the duplication of code across the classes as well as reduces the risk of errors. This code provides all of it's child classes with a routine to access the database.

Below is the code for the interaction with the table 'Users' within the database:

```
class UsersInfo(Database):
    def __init__(self, uid = 0):
        super().__init__(self)
        self.create_table()

    # NB: all input MUST be sanitized at this point.
    def create_table(self):
        self._connect_and_execute(SqlDictionary.CREATE_USERS)
        self._create_initial_admin_user()

    def _create_initial_admin_user(self):
        # This is to create an initial administrative user in case something happens to the database
        pwd = "".join([chr(random.choice(range(ord('A'), ord('z')))) for c in range(10)])

        new_user_info = {"Name":"ADMIN - TMP", "Username":"default_admin", "Password": gen_pw_hash(pwd), "Permissions": 29}

        if len(self.get_all_users("WHERE(Username = 'default_admin')")) == 0:
            print("[INFO] Empty users table detected, adding default user...")
            self.add_user(new_user_info)
            print("[INFO] Default user added,\n\tUsername: 'default_admin'\n\tPassword: '{0}'".format(pwd))

    def get_all_users(self, condition = ""): # Add a SQL condition? maybe? TODO: refactor this bit
        return self._connect_and_execute(SqlDictionary.GET_ALL_USERS.format(condition))

    def get_uid_by_username(self, username=""):
        return self._connect_and_execute((SqlDictionary.GET_USER_ID.format(username)))[0][0]

    def get_username_by_uid(self, uid=None):
        return self._connect_and_execute(SqlDictionary.GET_USERNAME_BY_UID.format(uid))[0][0]

    def add_user(self, info):
        #info follows the format {"SQL value":Data value}
        values = "'{0}', '{1}', '{2}', {3}".format(info["Name"], info["Username"], info["Password"], info["Permissions"])

        return self._connect_and_execute(SqlDictionary.ADD_USER.format(values))

    def update_user_password(self, password, uid):
        sql = SqlDictionary.UPDATE_PASSWORD.format("'{0}'".format(password), uid)
        return self._connect_and_execute(sql)
```

I wrote this code as a class so it could inherit properties from a parent class, in this case, “Database”. This class contains all the functions required for passing data in and out of the table “Users” in the database, as well as some of the management and administrative functions such as '*create_table*' and '*_create_initial_admin_user*'.

System Structure

Log in

This part of the system controls overall access to the system and prevents unauthorised users from accessing some or all of the system. When the user inputs a username and password combination, a hash is generated of the entered password and the user name is used to lookup the hash of the password stored in the database, the two hashes are compared. If the hashes match then the system prepares the GUI based on the permissions associated with the user.

Graphical User Interface

This part of the system provides the user with a graphical way of interacting with the software. It is designed to be easy and intuitive for users to use without or with minimal training. Command buttons and the various input components and positioned around the interface in an appropriately logical structure. The interface also includes various labels and hints to provide guidance to the user and to assist in their operation of the system.

Meetings Overview

The meetings overview tab contains all the controls and output regarding the meetings subsystem. It contains the functionality to add meetings, respond to attendance requests and, of course, view a list of meetings

Add Meeting

This dialogue box contains the functionality for adding meetings, this is where the user will add all of the information to be input into the database. The data that the user inputs will also be validated before it is committed to the database.

Respond

This dialogue box contains a list of meeting requests on the left-side, when the user clicks on one of the meeting requests, more information about that meeting is shown on the right-side, along with controls to delete, confirm or reject the request for meeting.

Tasks Overview

The tasks overview tab contains the functionality for dealing with the tasks section of the database, it contains a list of tasks taken from the database and the option to add a new task.

New Task

This dialogue box contains the form to add a new task to the tasks section of the database, it includes all the code for validating the tasks information before that information is committed to the database.

Resources Overview

The resources overview contains a 2-dimensional grid containing a direct copy of the resources table in the database. There is also the button to open the 'new resource' dialogue box.

New Resource

This window contains the form for adding a new resource to the database table 'Resources'

Admin Area

This area is designed to contain all the miscellaneous tools for administration of the 'users' database table. It includes a button which launches the 'change password' dialogue box.

change password

This dialogue box contains one place to enter the user's current password, which is then hashed and checked against the database and then two places to enter the new password, which are then hashed and checked that they match eachother.

Variable Listing

Class *Meeting* (does not inherit)

Variable Name	Purpose
self.title	Attribute for the meeting title taken from the database
self.place	Attribute for the meeting location taken from the database
self.attendees	Contains a python list of attendees' usernames
self.when	Attribute to contain the data and time of the meeting, as fetched from the database.

Class *NewMeetingDialog* (Inherits from Qdialog)

Variable Name	**Purpose**
Self.user	Contains the user information passed to this class from the parent window

Variable Name	Purpose
Self.main_layout	Contains the Qt layout used for organising the components of the window
Self.meeting_title_label	Contains the Qlabel widget that is used as a title for the window
Self.meeting_title_entry	Contains the QlineEdit into which the user inputs the meeting title.
Self.attendees_label	Contains a Qlabel to identify the entry point for attendees information
Self.attendees_container	Contains a Qwidget for logically organising the customized input structure for attendees
Self.attendees_layout	The layout used for organising the above container
Self.attendees_entry	Contains the manual entry box for usernames to be added as attendees
Self.username_lookup_button	Contains a button which provide functionality to the username lookup feature
Self.attendees_info_label	Contains a small piece of text to explain the process of entering usernames as attendees to this meeting
Self.where_label	Contains a Qlabel to distinguish the entry point of the meeting's location
Self.where_entry	Contains a QlineEdit for the meeting's location
Self.button_container_widget	Contains a customized Qwidget for organising the control buttons in a logical format.
Self.button_container_layout	The layout used for organising the above container
Self.save_button	Provides a button for the user to save the completed meeting to the database.
Self.cancel_button	Provides a button for the user to discard the information they input and to close the dialogue
Info	Contains a python dictionary containing all the information associated with the meeting
Meeting	Contains a *MeetingsInfo* object to interact with the database
Valid	Keeps track of if the attendee is valid or not
raw_attendee_list	Contains the text from Self.attendees_entry
Pattern	Provides a python regex object for dealing with regular expressions

Class *DiaryView* (inherits from Qwidget)

Variable Name	Purpose
Self.main_layour	Contains the layout for organising the highest level elements of the page
Self.title	Shows a large title text at the top of the 'page'
Self.middle_widget	Provides a container for the central elements of the pagr
Self.middle_layout	Organises the central elements of the 'page'
Self.left_side_widget	Provides a container for all the left-side elements
Self.left_side_layout	Organises the elements in the above container
Self.meetings_widget	Provides a container for all the meetings objects
Self.meetings_layout	Organises the elements in the above container
Self.meetings_widgets_container	Provides a scrollable area for all the meetings objects
Self.right_side_widget	Provides a container for all the right-side elements
Self.right_side_layout	Organises the elements in the above container
Self.button_container	Provides a container for the command buttons
Self.button_layout	Organises the elements in the above container
Self.new_appointment_button	Provides a button to open the 'New Meeting' dialogue.
Self.response_container	Provides a container for all the response options
Self.response_layout	Organises the elements In the above container
self.respond_to_pending_appointments_button	Provides a button to open the 'Respond to pending appointments' dialogue
Self.pending_number	Shows the number of pending appointments the user has

Class *PasswordWarningDialog* (inherits Qdialog)

Variable Name	Purpose
Self.main_layout	Contains the Qt layout used for organising the components of the window
Self.title	Shows some text as the title to the page
Self.text	Shows the error message for this dialogue
Self.dismiss_button	Provides a button to get rid of the dialogue box
Self.subtext	Shows a little information message at the bottom of the window

Class *LoginWindow* (inherits Qdialog)

Variable Name	Purpose
Self.main_title	Shows the text “Welcome” at the top of the page
Self.username_item	A container for the username entry set of widgets
Self.username_layout	The layout used for organising the above container
Self.username_label	Shows short descriptive text for the username entry field.
Self.username_input	Provides a place for the users to input their username
Self.password_item	Provides a container for the various elements associated with password entry
Self.password_layout	The layout used for organising the above container
Self.password_label	Shows a short descriptive text for the password entry box
Self.password_input	Provides a place for the user to enter their password
Self.button_layout	Organises the control buttons for this section
Self.submit_button	Submits the data the user entered into the entry points and begins the login subroutine
Self.help_button	Provides the users with help when clicked
Self.quit_button	Provides the user with a button that will exit the software immediately

(note: do the variable listing of more classes)

Explanation of detailed Algorithms

LoginAction.py

```
import hashlib

from DatabaseInit import *
import SqlDictionary

class User:
    def __init__(self, uid=0):
        self.info = {} # info to be retrieved from the database
        self.permissions = {}
        self.user_id = uid

        self.dbinterface = UsersInfo()

        self.update_user_info()
```

```
    def gen_pw_hash(self, password):
        # Create a md5 hash of the password
        phash = hashlib.md5()
        # (Encode the password – the python md5 implementation only accepts
binary data.)
        phash.update(bytes(password, "UTF-8"))
        # return a hexadecimal representation of the md5 hash.
        return phash.hexdigest()

    def password_hash_cmp(self, password_input):
        currenthash = self.info["Password"]
        return currenthash == self.gen_pw_hash(password_input)
```

In this code I have used Python's builtin hashing library to secure the passwords with one-way encryption so that if the user or someone else accesses the database containing the user names and passwords, they will not be able to use that information to gain access to the system. In this instance, I have used the md5 algorithm.

Permissions Array

```
def gen_permissions(self):
        # Get the denary integer value for the user's permissions
        perm = self.info["Permissions"]

        # Create a list of the default values for the user's permissions
        blist = [False, False, False, False, False]

        # For each item in a list of the individual binary digits
        # The python `bin` function outputs a string in the format '0b10101'
        # which is why we need to get rid of the first two characters
        for index, digit in enumerate(bin(int(perm))[2:]):
            # set each item in the b(inary)list as a python Bool so it can easily
            # be used in selection statements
            blist[index] = (bool(int(digit)))
        permissions = {}
        permissions["Meetings"] = blist[0]
        permissions["Tasks"] = blist[1]
        permissions["Resources"] = blist[2]
        permissions["ChangeOwnData"] = blist[3]
        permissions["Admin"] = blist[4]

        # Overwrite the existing permissions number with a dictionary.
        self.permissions = permissions
        return permissions
```

For this code I have used the python integer-to-binary conversion as well as python dictionaries[1] (a basic key-value database system that is stored in memory). This algorithm converts an integer value taken from the global database into a binary string (taking the format *0b10101*) and then converting that into a list of boolean data (taking the format *[True, False, True, False, True]*) which can then easily be used elsewhere in the system.

1 https://docs.python.org/3.4/library/stdtypes.html?highlight=dict#dict

User name Parsing

```
def _add_attendees(self, meeting):
        valid = True
        raw_attendee_list = self.attendees_entry.text()
        #Parse this into a list of attendees, separated by either semicolons or
commas.
        pattern = re.compile("([a-zA-Z]+;?)")
        # Iterate through the list of attendees
        print(pattern.findall(raw_attendee_list))

        for string in pattern.findall(raw_attendee_list):

            if string[-1] == ";":
                string = string[0:-1]
            try:
                attendeeID = UsersInfo().get_uid_by_username(string)
            except IndexError:
                print("[WARN] Username '{0}' not recognised".format(string))
                attendeeID = 0
                valid = False
                # Show a warning dialog and prevent the form from completing.
            if attendeeID:
                meeting.add_meeting_attendee(attendeeID)
        return valid
```

From the file "*NewMeetingDialog.py*" in the class "*NewMeetingDialog*". I used python's built in support for regular expressions, in this instance, I wanted the user names input as text and separated by any non-alphabet character, for this I used the regex *([a-zA-Z]+;?)* which is designed to separate strings based on any characters which are not in the alphabet (a-Z) and are either upper or lower case. It then checks each of the strings it finds against the database. Because of the way the database code is written, if it can't find a user with a matching username, it raises an *IndexError*, which is caught by the "***except** IndexError*" and processed appropriately.