3K04 Assignment 1 Part 2

DCM Documentation

Emily Ham, 400068761, hamj1

Aaron Billones, 40012394, billonea

Thomas Yoo, 400261798, yoot

Tingyi Ridvan Song, 400208112, songt7

Hadi Daniali, 400353520, mohamh46

Hanxiao Chang, 400259468, changh28

Table of Contents

1 Device Controller Monitor

The device controller monitor (DCM) is a user interface for the pacemaker. It visually communicates to the physician when the correct device is connected and allows them to select a pacing mode and specify parameter values. The DCM allows up to 10 users to be created.

2 Modules

2.1 Database

The database module is used to create, store, and retrieve user information. Each user document contains the username, password, a device ID, a dictionary of parameters containing either initial default values or previously set values, and the previously set operating mode. It was built using the Python library *TinyDB*, which contains basic database commands and stores information in a json file.

2.1.1 Module Interface Specification

The module contains three public functions outlined in Table 1. There are two global variables: *db* (datatype: TinyDB) which represents the database and *UserQuery* (datatype: Query) which is used to perform database queries. Without regarding the internal behavior, the database module functions as a dictionary factory where the module returns a dictionary for you to access user information such as password, DCM parameters and operation modes.

Table 1: Public functions in the database module

Functions	**Purpose**	**Parameters**	**Return Type**	**Return Value**
create_user	Creates a new user in the database if a user with the same username doesn't already exist	username: str password: str	Boolean	True: user created False: user not created

get_user	Gets the user from the database with the given username	username: str	User: {
			username: str,
			password: str,
			device_id: int,
			parameters:
			dict[str,str],
			operating_mode: str
			}
				User if user exists
				
				None if user doesn't exist

get_number_of_users	Gets the number of users currently stored in the database	N/A	Int	Number of users in database
update_parameters	Updates parameters with the provided values	username: str parameters: dict[str,Entry]	N/A	N/A

	for the given user in the database			
update_operating_mode	Updates operating mode for the given user in the database	username: str operating_mode: str	N/A	N/A

create_user first does a check if there is an existing user with the given username using the function *get_user*. If no existing user is found, a new user is inserted into the database with the given username and password and the function returns true. The user is also prepopulated with blanks for all of the programmable parameter values. If an existing user is found, the function returns false.

get_user creates a database query that searches for users with the given username. The query returns a list of users if any are found or an empty list if none are found. Because there is a check for unique usernames in *create_user*, the list should either be empty or have a single user, and so if the list is not empty, the first element is returned.

get_number_of_users returns the size of the database using the function *len()*.

update_parameters finds the user in the database that matches the given username by calling *get_user*. It then loops through the key values in the parameter dictionary parameter to perform the value replacements. This allows the user to keep any existing values for parameters that were not included in the dictionary. A database update is then performed with the new parameter object.

2.1.2 Module Internal Design

Although the Database module does not include any private functions, there is still internal behavior of the module that is not detectable from its interface. Internally, every time a user's data is updated or retrieved, the information is retrieved from or written to a text-based json file from the database and converted to a python dictionary datatype to be used by the caller. This internal behaviour is fully managed by TinyDB.

2.2 GUI

The GUI module was developed to enable users to easily interact with the pacemaker device, it is the only access point for the user to access the information in the database, and it is also the access point for users to interact with the pacemaker device. It provides a front-end user interface in the form of a desktop application. It was built using the Python library T*kinter*.

2.2.1 Module Interface Specification

The interface which the rest of the code interacts with the GUI is through the gui.GUI object. The functions that the main function can use GUI is through its public methods. The main function of the GUI is that it provides the mode of creating and populating a user interface which represents the data of the user and acts as a control panel for the device. It allows you to make entries specifying the mode of operation of your pacemaker and enter all parameters associated with that operating mode. It also specifies ID information of your pacemaker device and allows you to submit your parameters and operating mode to the connected device.

Table 2: Public methods of the GUI object

Methods	Purpose	Parameters	Return Type	Return Value
Constructor	Initializes the object when it is created in a different module	None	N/A	None
quit_win	Exits the GUI window	None	N/A	None
loop	Puts the GUI in a looping state where it updates the interface when any action is detected	None	N/A	None
update	A manual update to the interface state when you would like to cause a change to the interface without performing any action	None	N/A	None

__init__ constructor will initialize all the GUI instance variables listed in section 2.2.2, initialize the window of the application, and call the *_create_welcome_screen* method to populate the welcome screen.

quit_win will call the associated tkinter method to close the window and the application, exiting the GUI loop

loop will call the associated tkinter method which puts the GUI in an infinite loop which looks for events in the application

update will call the associated tkinter method which manually update the GUI application interface once, it is equivalent to one iteration of the *loop* function

2.2.2 Module Internal Design

Global State Variables

VALID_PARAMETERS:

The global variable valid parameters were a dictionary used to hold the information about which DCM parameters were valid for each operating mode as specified in page 28 of PACEMAKER.pdf. Each dictionary key was the name of an operating mode which held a list of parameter names which were valid under that mode.

GUI Object Instance State Variables

window:

A tkinter TK object which represents the window of the application.

frame:

A tkinter Frame object which represents a single display within the window of the application.

state:

Indicates the present page that the GUI is on as a string.

user:

A dictionary holding information from the database of the present user which is logged in. Initialized as a None object and populated after login or registration of a user.

mode:

String type variable holding the string enumeration of the operation mode (‘aai’, ‘vii’, ‘aoo’ etc.), for easier access to the operation mode.

modes_dict:

A dictionary containing all the button objects associated with the operating modes accessible using string enumerations of the mode names. Used for disabling and enabling operation mode buttons when they are selected.

parameters_dict:

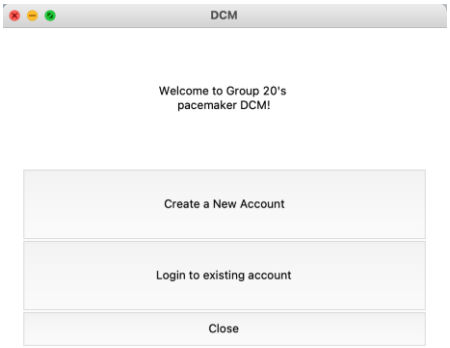
A dictionary containing all the entry objects associated with the DCM parameters accessible using string enumerations of the parameter names. Used for storing and accessing parameter values.

Private Methods

The following private methods will be categorized by their relation to a GUI page since the functionality of each page unifies the functionality of the methods. It is easier to understand if private functions were grouped by GUI frames

Welcome Screen

Fig. A: GUI appearance of the menu screen



The purpose of the welcome screen is to provide the user with a menu to register or login in with their information. This ensures each user’s DCM parameters are registered under their profile and saved. The welcome screen features buttons for creating a new user and for logging into an existing user. There is also a button to close the application. There are no public methods for the welcome screen, and the private methods are as outlined in Table 3.

Table 3: Private methods in the GUI module for the welcome screen

Methods	Purpose	Parameters	Return Type	Return Value
_create_welcome_screen	Creates a new frame with a welcome message, create user button, and login button	N/A	N/A	N/A

The _create_welcome_screen function will be called every time the welcome screen is needed by the program. This method will populate the application window with the necessary buttons for registration, login and closing the application. This method will change *state* instance variable to the corresponding page name, it will also resize the window using the *window* instance variable and replace the *frame* instance variable with a new frame containing the information of the welcome screen.

Each button in the welcome screen page will be linked to certain method. These are described in Table 4.

Table 4: methods linked to buttons in the welcome screen

Methods	Purpose	Linked Button	Parameters
_create_register_screen	Described in Table 5	Create a New Account	None
_create_login_screen	Described in Table 7	Login to Existing Account	None

Register User Screen

Fig. B: GUI appearance of the registration screen

The create user screen was the frame which allowed the user to register themselves as another user in the database. The create user screen allows for up to 10 users to be created. It features two entry fields for the user to enter a username and password and two buttons to submit the user information or go back to the welcome screen. The private methods related to the _create_user_screen are shown in Tables 5 and 6.

Table 5: Private methods in the GUI module for the _create_user_screen

Methods	Purpose	Parameters	Return Type	Return Value
_create_register_screen	Creates a new frame with entry fields and buttons for registering a new user	N/A	N/A	N/A

The _create_register_screen function will be called every time the register screen is needed by the program. It is linked to the *Register a New Account* button in the welcome screen. This method will populate the application window with the necessary buttons and entry field for registering yourself with a username and password. This method will change *state* instance variable to the corresponding page name, it will also resize the window using the *window* instance variable and replace the *frame* instance variable with a new frame containing the information of the register screen. The two buttons will be linked to the corresponding functionality for registering a user (more detail in Table 6) and returning to the home screen using the _create_welcome_screen.

Table 6: Private methods linked to buttons in the welcome screen

Methods	Purpose	Linked Button	Parameters
_create_welcome_screen	Described in Table 3	Back	None
_on_submit_register	Takes the entered username and password, verifies it is valid and saves it to the database	Submit	usernameEntry, passwordEntry
_create_dcm_screen	Described in Table 9	Submit	None

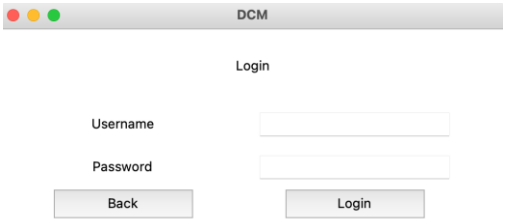
_on_submit_register is a private method which will verify the correctness of the username and password passed in and submits it to the database under the condition that they are correct. The verification that the method will do contains:

- The user is not registering under a device which has more than 10 users
- No fields are left empty
- The username has not previously been taken

It will then register the user using the db.create_user described in Table A, then set the *user* instance state variable as the newly created user, and proceed to the DCM screen as the new user using the method _create_dcm_screen.

Login Screen

Fig. C: GUI appearance of the login screen



The login screen was the frame which allowed the user to login to their account and access the DCM interface using their parameters and settings. It features two entry fields for the user to enter a username and password and two buttons to submit the user information or go back to the welcome screen. The private methods related to the _create_login_screen are shown in Table 7 and 8

Table 7: Private methods in the GUI module for the _create_login_screen

Methods	**Purpose**	**Parameters**	**Return Type**	**Return Value**
_create_login_screen	Creates a new frame with entry fields and buttons for logging in	N/A	N/A	N/A

The _create_login_screen method will be called every time the login screen is needed by the program. It is linked to the *Login to Existing Account* button in the welcome screen. This method will populate the application window with the necessary buttons and entry field for registering yourself with a username and password. This method will change *state* instance variable to the corresponding page name, it will also resize the window using the *window* instance variable and replace the *frame* instance variable with a new frame containing the information of the register screen. The two buttons will be linked to the corresponding functionality for logging in as a user (more detail in Table 8) and returning to the home screen using the _create_welcome_screen.

Table 8: methods linked to buttons in the welcome screen

Methods	**Purpose**	**Linked Button**	**Parameters**
_create_welcome_screen	Described in Table 3	Back	None
_on_submit_login	Takes the entered username and password, verifies it is valid, and the password is correct, and enters DCM screen if valid	Submit	usernameEntry, passwordEntry
_create_dcm_screen	Described in Table 9	Submit	None

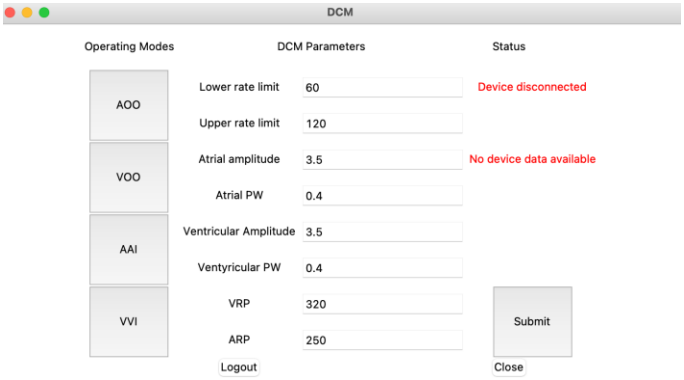
_on_submit_login is a private method which will verify the correctness of the username and password passed in and submits it to the database under the condition that they are correct. The verification that the method will do contains:

- The user exists
- The password is correct

It will then login the user using the db.get_user described in Table A, check if the password is correct, then set the *user* instance state variable as the logged in user and proceed to the DCM screen using the method _create_dcm_screen.

DCM Home Screen

Fig. D: GUI appearance of the DCM screen



The DCM screen is the frame which serves as the main control panel of controlling the pacemaker device. It contains 3 columns of containing selections for choosing the operation modes, entries for all related parameters and information about the device. Additionally, there are buttons for uploading your parameters to the device and your account, logging out of your account, and closing the application. The private methods related to the _create_dcm_screen are listed below in Table 9 and 10

Table 9: Private methods in the GUI module relating to _create_dcm_screen

Methods	**Purpose**	**Parameters**	**Return Type**	**Return Value**
_create_dcm_screen	Creates control panel for submitting modes and parameters to the connected device and viewing device information	N/A	N/A	N/A
_load_user_defaults	Loads the user's previous values into the parameter entries, or the	N/A	N/A	N/A

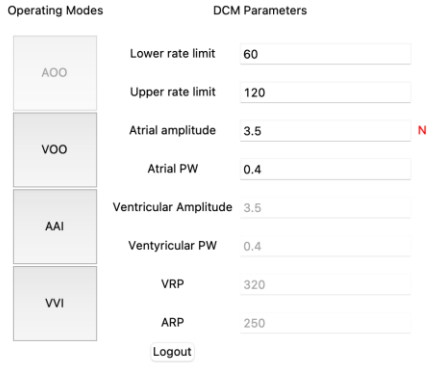
	system defaults if the user is new			
_find_parameters_for_mode	Find the valid parameters for the valid mode and curates the full dict of operating modes into a dict containing only the parameters valid to this mode	parameter_dict mode	Dictionary	parameters_for_mode
_validate_parameters	Takes the mode and validates the parameters valid for that mode and returns all error messages	mode	Boolean Set	Valid errormessageset

The _create_dcm_screen method will be called every time the pacemaker control panel is needed by the program. It is called whenever a new user is registered of logged in. This method will populate the application window with entries and buttons to select the operating modes and parameters. It will also populate a column of entries for parameters.

The default values of the operating modes are set to none if the user is new and will default to the previously used operating mode of the user if the user is registered. The values of the DCM parameters will be initialized as a set of defaults specified in page 34 of the PACEMAKER.pdf document if the user is newly registered, or it will be initialized as the previously used values of that user if the user previously logged in.

When an operating mode is selected, the button for that mode will be disabled while the rest remain enabled. The programmable parameters which are not applicable for this mode will be disabled according to the global variable VALID_PARAMETERS for the certain operating mode, As shown in Fig. E.

Fig. E GUI appearance of the operation modes and parameters when a selection is made



When you submit, the function will verify that you are have an operating mode selected, and that all the DCM parameters are in a valid range as specified in page 34 of the PACEMAKE.pdf. When you submit,

only the valid parameters for that mode will be submitted to the device and updated to the datebase while the others remain unchanged.

This method will change *state* instance variable to the corresponding page name, it will also resize the window using the *window* instance variable and replace the *frame* instance variable with a new frame containing the information of the DCM screen. The self.parameters_dict and self.modes_dict will be initialized with their corresponding buttons and entry fields in this page.

_load_user_defaults is a method called within the _create_dcm_screen method which initializes the operating modes buttons and parameter entries settings into the user defaults stored in the database. If the user is newly registered and has no prior entries, the defaults stored in the database for operation_mode is an empty string, and the entries are the defaults specified in page 34 of PACEMAKER.pdf. It will set the state instance variable *mode* of the operating mode of the user.

_find_parameters_for_mode is a method which uses the VALID_PARAMETERS global state variable dictionary to curate a list of parameter entries that are valid to the mode given when passed in a parameter entry dict.

_validate_parameters is a method which will validate the instance state variable parameters_dict to ensure that all the parameters valid for that given mode is within the valid range as specified in page 34 of the PACEMAKER.pdf. The method will only check the parameters valid for the specified mode. The state instance variable *parameters_dict* is used to get the parameter entries. It uses the validate parameter module validateentry namespace of functions to validate the range of the parameters and returns all the error messages generated and a Boolean stating whether it passed validation.

Table 10: Methods linked to buttons in the welcome screen

Methods	**Purpose**	**Linked to**	**Parameters**
_create_welcome_screen	Described in Table 3	Logout	None
quit_win	Described in Table 2	Close	None
_set_mode	Sets the mode of the DCM for the user and disables invalid fields for that operating mode	Will link to each operation mode button	Operating mode string
_submit_parameters	Verifies the correctness of all the selections and submits only the valid parameters of the mode into the database and device	Submit	None

_set_mode will disable the button for that mode will be disabled while the rest remain enabled given the mode. The programmable parameters which are not applicable for this mode will be disabled according to the global variable VALID_PARAMETERS for the certain operating mode, As shown in Fig. E. This method will set the *mode* instance state variable to the current mode being selected

_submit_parameters will run the procedures of validation for the parameters entered in the entry fields, handle errors as popup windows when they error and submit the valid parameters of that mode into the database and device if they are correct. Submit parameters will determine the parameters valid for this

mode using the _find_parameters_for_mode method, verifies them using the _validate_parameters method. If errors are returned, a popup will be prompt the user with the first error message displayed and indicate the total number of errors present and will not submit to the device or database. If no errors are returned, only the parameters returned from _find_parameters_for_mode is submitted to the database and the device when a device is connected. Currently device submission is not implemented as it is not required for this assignment, in the future, errors and checks associated to the device will also be checked. When a successful submission is made, the mode and parameter data will be saved to the database using the db.update_parameters and db.update_operating_modes functions, a popup using tkinter messagebox will appear verifying that the settings have been submitted. Instance state variable *user* will be used to read and write the user information on parameters and modes to and from the database, state variables *mode*, *mode_dict and parameter_dict* will also be read but not modified to verify their correctness.

2.3 Parameter Validation

The parameter validation module is used to ensure that the user only inputs valid information, and that only valid information is passed into the database. Every time the user tries to submit information through the DCM, depending on the selected mode the validity of the corresponding parameters is assessed. In the case that some input is invalid, error messages are returned to the user and the data is not submitted, and in the case that the input is valid, the new values overwrite the old values in the database.

2.3.1 Module Interface Specification

The module contains 8 public functions outlined in Table 11. Each function takes the users input as a parameter, and returns a Boolean and a string, indicating the validity of the input and any relevant error messages. Each function tests that the input is of the correct type (*int* or float) and then ensures that the value is in the correct range, and of the correct interval. These functions can all be found in */DCM/validateentry.py*.

The GUI also contains an associated function that calls the module's validation functions to assess whether all current inputs are valid. This function, _validate_parameters, takes the mode of the pacemaker as a parameter and calls the individual validation functions while tracking the validity of the inputs using the all_valid Boolean, and holding all the error messages in the errormessageset array.

Table 11: Parameter Validation functions

Functions	**Purpose**	**Parameters**	**Return Type**	**Return Value**
validate_lrl	Validates lower rate limit input. Checks input data type, and value.	lrl: int	Boolean, String	True, empty string: input passed all checks False, error message string: input failed some check(s)
validate_url	Validates upper rate limit input. Checks input data type, and value.	url: int	Boolean, String	True, empty string: input passed all checks

				False, error message string: input failed some check(s)
validate_regulated_atrial_amp	Validates regulated atrial amplitude input. Checks input data type, and value.	aa: float	Boolean, String	True, empty string: input passed all checks False, error message string: input failed some check(s)
validate_atrial_pw	Validates atrial pulse width input. Checks input data type, and value.	apw: float	Boolean, String	True, empty string: input passed all checks False, error message string: input failed some check(s)
validate_regulated_ventricular_amp	Validates regulated ventricular amplitude input. Checks input data type, and value.	va: float	Boolean, String	True, empty string: input passed all checks False, error message string: input failed some check(s)
validate_ventricular_pw	Validates ventricular pulse width input. Checks input data type, and value.	vpw: float	Boolean, String	True, empty string: input passed all checks False, error message string: input failed some check(s)
validate_vrp	Validates ventricular refractory period input. Checks input data type, and value.	vrp: int	Boolean, String	True, empty string: input passed all checks False, error message string: input failed some check(s)
validate_arp	Validates atrial refractory period input. Checks input data type, and value.	arp: int	Boolean, String	True, empty string: input passed all checks False, error message string: input failed some check(s)
_validate_parameters	Performs appropriate validation checks depending on current selected mode	mode: string	Boolean, Array	True, empty array: All validations passed, no error messages False, array: At least one validation failed, array contains error messages

2.3.2 Module Internal Design

Internally, the validate parameters checks the type of the input along with the range and increment of the input.

Type check:

validate_lrl, validate_url, validate_vrp, validate_arp:

These functions perform their type check by casting the input as an integer and excepting ValueErrors.

validate_regulated_atrial_amp, validate_regulated_ventricular_amp, validate_atrial_pw, validate_ventricular_pw:

These functions perform their type check by vesting the input as a float, and recasting it as an int multiplied by some integer for later use. They except ValueErrors.

Value and Increment check:

Each function tests the inputs range using logical operators and comparative statements. They then check the increment of the input by taking the modulus of it against the valid increment, and determining whether the result is 0. The ranges and increments can be seen in Table 12.

Table 12: Valid input ranges and increments

Functions	**Valid Range(s)**	**Valid Increment(s)**
validate_lrl	30-175 ppm	From 30-50 ppm: 5ppm From 50-90 ppm: 1ppm From 90-175 ppm: 5ppm
validate_url	50-175 ppm	5ppm
validate_regulated_atrial_amp	0, 0.5-3.2, 3.5-7 V	From 0.5-3.2V: 0.1V From 3.5-7V: 5V
validate_atrial_pw	0.05, 0.1-1.9 ms	From 0.1-1.9ms: 0.1ms
validate_regulated_ventricular_amp	0, 0.5-3.2, 3.5-7 V	From 0.5-3.2V: 0.1V From 3.5-7V: 5V
validate_ventricular_pw	0.05, 0.1-1.9 ms	From 0.1-1.9ms: 0.1ms
validate_vrp	150-500 ms	10ms
validate_arp	150-500 ms	10ms

GUI validation check:

The validation function that the GUI calls receives the current active mode as a string and calls the individual validation functions accordingly. Every time a validation check occurs, the *all_valid* Boolean is updated, where it is set to false in the case that any of the checks fail. Additionally, if a validation check returns a non-empty string, it is added to the *errormessageset* array to be displayed to the user.

3 Testing

Unit Tests

These tests are currently manually done because there are not enough edge cases to justify the need to create automated testing procedures. However for assignment 2 we will be creating automated unit testing using the python unittest construct.

Table 13: Unit tests for database module

create_user	
Creates new user in database and returns True given username that doesn't already exist and password	Pass

Doesn't create new user in database and returns False given username that already exists and password	Pass
get_user	
Returns user if username is found in database	Pass
Returns None if username is not found in database	Pass
Returns None if username has different capitalization as in database	Pass
Returns None if password has different capitalization as in database	Pass
get_number_of_users	
Returns 0 if database is empty	Pass
Returns number of users if database is not empty	Pass
update_parameters	
Updates a single parameter without changing other values	Pass
Updates multiple parameters	Pass
Updates parameters for only user when there are multiple users	Pass
update_operating_mode	
Updates operating mode when not previously set	Pass
Updates operating mode when previously set to different mode	Pass
Updates only operating mode of the correct user when there are multiple users	Pass

Table 14: Unit tests for database module

_find_parameters_for_mode	
If given an invalid mode that does not exist, error given and will not crash	Pass
Given list or parameters, only the valid parameters for the mode is returned	Pass
_validate_parameters	
Given an invalid mode, will not crash and proper error message is displayed	Pass
If given correct entries valid is True and no error messages are returned	Pass
If given incorrect entries valid is returned False and correct error messages are returned	Pass

Table 15: Unit tests for validation module

validate_lrl	
Returns True and empty string on valid entry	Pass
Returns False and appropriate error message on invalid type	Pass
Returns False and appropriate error message on incorrect range	Pass
Returns False and appropriate error message on invalid increment	Pass
validate_url	
Returns True and empty string on valid entry	Pass
Returns False and appropriate error message on invalid type	Pass
Returns False and appropriate error message on incorrect range	Pass
Returns False and appropriate error message on invalid increment	Pass
validate_regulated_atrial_amp	
Returns True and empty string on valid entry	Pass
Returns False and appropriate error message on invalid type	Pass

Returns False and appropriate error message on incorrect range	Pass
Returns False and appropriate error message on invalid increment	Pass
validate_atrial_pw	
Returns True and empty string on valid entry	Pass
Returns False and appropriate error message on invalid type	Pass
Returns False and appropriate error message on incorrect range	Pass
Returns False and appropriate error message on invalid increment	Pass
validate_regulated_ventricular_amp	
Returns True and empty string on valid entry	Pass
Returns False and appropriate error message on invalid type	Pass
Returns False and appropriate error message on incorrect range	Pass
Returns False and appropriate error message on invalid increment	Pass
validate_ventricular_pw	
Returns True and empty string on valid entry	Pass
Returns False and appropriate error message on invalid type	Pass
Returns False and appropriate error message on incorrect range	Pass
Returns False and appropriate error message on invalid increment	Pass
validate_vrp	
Returns True and empty string on valid entry	Pass
Returns False and appropriate error message on invalid type	Pass
Returns False and appropriate error message on incorrect range	Pass
Returns False and appropriate error message on invalid increment	Pass
validate_arp	
Returns True and empty string on valid entry	Pass
Returns False and appropriate error message on invalid type	Pass
Returns False and appropriate error message on incorrect range	Pass
Returns False and appropriate error message on invalid increment	Pass
_validate_parameters	
Returns True and empty array on valid entries	Pass
Returns False and populated array on invalid entries	Pass

Integration Tests

Integration testing is needed for testing the user flow of the GUI, these cannot be done through unit testing because they require visual verification of the design and functionality of the GUI interface. They also ensure that the GUI properly interacts with and responds to the database and parameter validation modules.

Test Case	**Requirements**	**Pass/Fail**
Create user with unique non-blank username and non-blank password	• User created in database • Redirect to home page • Programmable parameters prepopulate with default values	Pass
Create user with non-unique username	• Error message: User with that username already exists. • No new user added to database • Stays on Register screen	Pass

Create user with blank username or password	• Error message: Username and password cannot be empty. • No new user added to database • Stays on Register screen	Pass
Create user with 10 existing users for given device	• Error message: Maximum number of users reached. • No new user added to database • Stays on Register screen	Pass
Login with correct username and password	• Redirect to Home page • Programmable parameters prepopulate with values from database	Pass
Login with incorrect username	• Error message: User not found. • Stays on Login screen	Pass
Login with incorrect password	• Error message: Incorrect password. • Stays on Login screen	Pass
Select operating mode, enter valid values, and submit	• Entry fields not related to selected operating mode are disabled • Values get saved to database on submit • Popup informing user that values were saved	Pass
Select operating mode, enter invalid values, and submit	• Entry fields not related to selected operating mode are disabled • Values do not get saved to database on submit • Popup informing user of validation errors	P
Select operating mode, enter valid values, change operating mode, enter valid values, and submit	• Entry fields not related to selected operating mode are disabled • Values related to operating mode selected on submit get saved to the database • Values only related to the first selected operating mode do not get saved to the database • Popup informing user that values were saved	Pass
Submit without selecting operating mode	• Popup with error message: No operating mode has been selected • No data saved to database	Pass

4 Next Steps

The highest priority next step is to implement new features based on the requirement changes. The pacemaker connection will be implemented to allow for communication between the pacemaker device

and the DCM. This will include developing new modules to handle the device connection. Device information, likely in the form of a serial number, also needs to be correctly added to the user database, as currently there is only a hardcoded device id associated with every user due to the connection not having been implemented. A state variable which holds information on the connected device will likely also be added so that the DCM can correctly inform the user that the DCM is connected to the correct pacemaker. There will also be support added for additional pacing modes (DOO, AOOR, VOOR, VVIR, and DOOR). This will involve developing new buttons and new parameter validations.

As the requirements become more complex, there are also some design decisions that may be changed. Adding new pacing modes could lead to changes in the UI as there is currently a button for each pacing mode and this could overcrowd the page, making the DCM more difficult to use. Potential solutions to this problem include putting these on a separate page from the programmable parameters, adding a scroll feature so that not all buttons are displayed at once, or using a dropdown to select the operating mode. Additionally, as the GUI module becomes increasingly large and complex, it may be modularized in the near future to separate the logic for each of the pages.

The design of the parameter validation module may also be changed as more pacing modes are added. With new validation functions added to the module, a validation class could be created rather than the module consisting only of the individual validation functions.

Some other changes that may be implemented to handle the increasing complexity of the code is creating automated tests and converting the Python script into an executable. Automated tests would reduce the risk of bugs in existing features caused by new code changes. As the application becomes more complex, it will become increasingly more difficult to manually test the application with each major change, and so automated tests would help make the application more reliable. Converting the program to an executable would make it easier for the user to run by removing the dependency on pre-installed libraries.

References

"PACEMAKER System Specification." Boston Scientific, Marlborough, Massachusetts, United States, 03-Jan-2007.