3K04 Assignment 2 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

2.0 Overview

The structure of the DCM has been built based around the storage, verification, and transmission of operating modes and control parameters to communicate with the pacemaker. At the base of our design is a user database, which is saved as a .json file, contains each user's personal information as well as their operating modes and parameters for the pacemaker. This information is fetched and written by the DCM via a database module. These parameters and modes are also verified for correctness via the validateentry.ParameterManager class. When it can be verified this information will be sent to the pacemaker via the serialcom.py module, and electrogram information can also be received from the pacemaker. A gui.GUI class was made to allow the user to access all this information using a graphic interface, and links all the modules into one unified interface.

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

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 for the given user in the database	username: str parameters: dict[str,Entry]	N/A	N/A
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

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.

Global State Variables

db (datatype: TinyDB):

Represents the database that stores the user information.

UserQuery (datatype: Query):

Used to perform queries on the user database.

Object Instance State Variables

The database module does not contain any instance state variables.

Private Functions

The database module does not contain any private functions.

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.

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.

user:

User object for accessing the user information with the database

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.

serial:

Reference to a serialManager class which handles serial communication between the UART connection to the pacemaker and the DCM

port_selection:

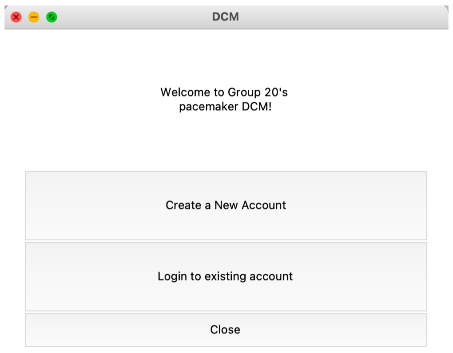
Variable for the port selected on the GUI, it is made into an instance variable to it is visible to any method in GUI.py to use for serial communication

Private Functions

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


DCM
Welcome to Group 20's pacemaker DCM!
Create a New Account
Login to existing account
Close

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.

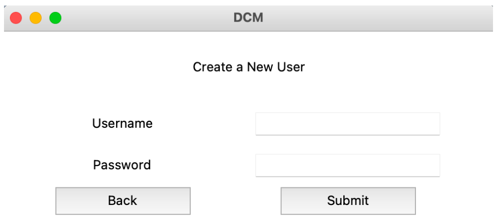
Each button in the welcome screen page will be linked to certain method. These are described in Table 4.

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

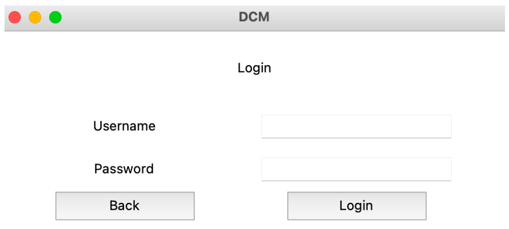
_on_submit_register is a private method which will verify the correctness of the username and password passed in and submits it to the database under the condition that they are correct. The verification that the method will do contains:

- 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

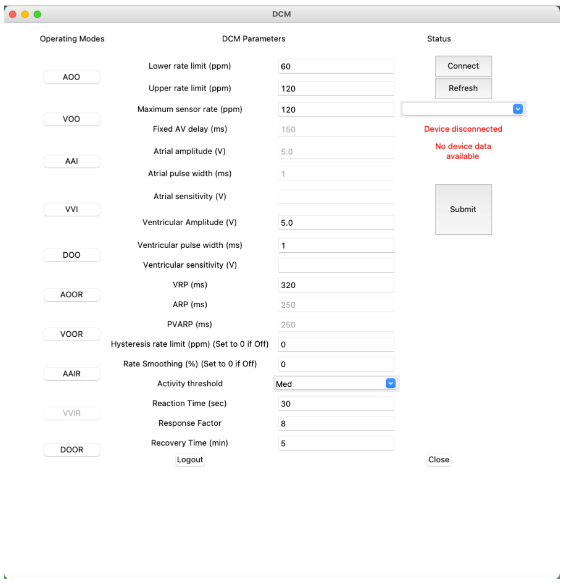
_on_submit_login is a private method which will verify the correctness of the username and password passed in and submits it to the database under the condition that they are correct. The verification that the method will do contains:

- 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



DCM Parameters

Lower rate limit (ppm)	60
Upper rate limit (ppm)	120
Maximum sensor rate (ppm)	120
Fixed AV delay (ms)	150
Atrial amplitude (V)	5.0
Atrial pulse width (ms)	1
Atrial sensitivity (V)	
Ventricular Amplitude (V)	5.0
Ventricular pulse width (ms)	1
Ventricular sensitivity (V)	
VRP (ms)	320
ARP (ms)	250
PVARP (ms)	250
Hysteresis rate limit (ppm) (Set to 0 if Off)	0
Rate Smoothing (%) (Set to 0 if Off)	0
Activity threshold	Med
Reaction Time (sec)	30
Response Factor	8
Recovery Time (min)	5

Device disconnected

No device data available

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

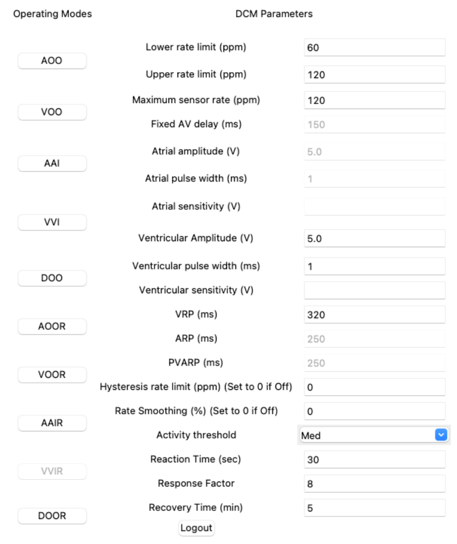
Methods	**Purpose**	**Parameters**	**Return Type**	**Return Value**
_create_dcm_screen	Creates control panel for submitting modes and parameters to the connected device and viewing device information	N/A	N/A	N/A
_load_user_defaults	Loads the user's previous values into the parameter entries, or the system defaults if the user is new	N/A	N/A	N/A
_update_parameter_entries	Updates the parameters to valid values in the GUI if the increment of the parameter was incorrect and empty the parameter if the parameter is wrong	Valid_parameters (List of valid parameters that is updated by the ParameterManager)	N/A	N/A
Self.serial._get_ports	Used for getting all available ports for the drop-down menu (see more in Table 14)	(see Table 14)		

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



DCM Parameters	
Lower rate limit (ppm)	60
Upper rate limit (ppm)	120
Maximum sensor rate (ppm)	120
Fixed AV delay (ms)	150
Atrial amplitude (V)	5.0
Atrial pulse width (ms)	1
Atrial sensitivity (V)	
Ventricular Amplitude (V)	5.0
Ventricular pulse width (ms)	1
Ventricular sensitivity (V)	
VRP (ms)	320
ARP (ms)	250
PVARP (ms)	250
Hysteresis rate limit (ppm) (Set to 0 if Off)	0
Rate Smoothing (%) (Set to 0 if Off)	0
Activity threshold	Med
Reaction Time (sec)	30
Response Factor	8
Recovery Time (min)	5

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.

_update_parameter_entries is a function that will update the GUI after the parameters are validated by the ParamaterManager when they are submitted. After the parameters are submitted using the Submit button, all the parameters will be verified for ranges and increments. If the incrementation of the parameter is wrong, the variable will be adjusted to the correct increment, or if the parameter is wrong, it will be replaced with an empty string. _update_parameter_entries will update the GUI based on these updated values.

Table 10: Methods linked to buttons in the dcm 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
_setup_device	Initialize the setup for connection to the pacemaker through the serial port selected in the dropdown	SerialManager class and updates the dcm screen with device information accordingly	None
_refresh_devices	Refreshes the DCM screen to update with new devices on ports	None	None
_disconnect_device	Resets the serial port for the connected device to none and update the dcm screen	None	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 and verify their correctness using the validateentry.ParameterManager class. If errors are returned, a popup will be prompt the user with the first error message displayed and clear the incorrect field. If no errors are returned and a device is connected, only the parameters returned valid for the mode is submitted to the database and the device. 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. The parameters will also be submitted via serial communication to the pacemaker through the connected port.

_setup_device use the serial port specified in the port selection dropdown and sets up the serial communication manager to connect to the device.

_refresh_devices will update the dropdown menu in the dcm screen with new devices that were plugged in while the GUI was running.

_disconnect_devie will update the dropdown menu and GUI dcm screen interface when disconnecting from the device and signals to the SerialManager to stop using the certain port.

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 takes the form of a manager class, which gets instantiated every time the prameters need to be checked and contains to 2 public methods to check the parameters, these are outlined in the table below. The methods test if 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, and the valid parameters of error messages can be retrieved from the class. These functions can all be found in */DCM/validateentry.py*.

The GUI also contains an associated procedure that calls the module's validation functions to assess whether all current inputs are valid in _submit_parameters, and this will display the errors as pop-up messages if the validation fails

Table 11: Parameter Validation functions

Functions	**Purpose**	**Parameters**	**Return Type**	**Return Value**
Constructor	Initializes the Parameter manager and filters out and saves the parameters to be checked for the certain mode	Valid_parameters, Parameters_dict	N/A	N/A
Run_checks	Initializes all checks and watches for errors	None	NoneType if no errors String if error occurs	None if no errors, string explanation of the invalid parameter is failed
Get_parameters	Getter for accessing the processed parameters for submission to the pacemaker	None	Dictionary	Dictionary of all valid parameters for this mode in the correct type format.

The __init__ constructor will be called every time the parameters want to be checked, passed in is a dictionary valid_parameters, which contains the information of all the parameters that are associated to the pacing mode on the pacemaker that we are running in. The other parameter passed in is the parameters_dict, which is a pointer to the unprocessed GUI entry information from the DCM parameter submission screen. These 2 arguemnts will then be passed to a private function to process the entry data information into a filtered dictionary of only the parameters for the used mode in string format.

Run_checks will set off the full range of checks created for checking the processed parameters for correctness. This includes parameter type, interval checks, range checks and cross checks between variables. If an error is found for the parameters, a ParameterError exception will be thrown, the run_checks method will be listening for these errors and if one is thrown, will catch it, convert it to a string and return it to the gui _submit_parameters function.

Get_parameters will be used to as a generic getter method to access the filtered and corrected parameters after the checks are done.

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. First, the valid parameters for the certain mode that is running will be filtered out and put in the _parameter_dict when the object is initialized. The checks will be split into 3 different types of checks. These checks will run the check for every parameter which is present and are controlled by 3 top level functions. These functions are detailed in the table below.

Justification for parameter types:

All numeric parameters that increment by 1 or more will be casted as an integer in order to avoid floating point inputs going through to the pacemaker.

All numeric parameters that increment by less than 1 will be casted as a float and converted to the correct decimal and increment to avoid rounding errors and overflow.

All string parameters have been implemented as a drop down menu and saved as string values, **however** they have been treated as enumerations and mapped to int values when passed to the pacemaker to ensure no user error can be introduced

Table 12: Top level check methods

Functions	**Purpose**	**Parameters**	**Return Type**	**Return Value**
_find_parameters_for_mode	Finds the parameters associated with the operating mode selected and returns a curated parameter_dict	Valid_parameters, Parameters_dict	Dictionary	_parameters_dict_new
_convert_parameter_types	Try to convert the types of each parameter to the needed type for testing and adjusts their interval.	None	N/A	N/A
_do_range_checks	Checks the range of all parameters for that mode to make sure	None	N/A	N/A

	that they are within a valid range			
_do_cross_checks	Checks the compatibility of all parameters to other related parameters	None	N/A	N/A

_find_parameters_for_mode will use all the keywords in the valid_parameters list representing all the needed parameters from that mode, then it will find the string in the entry objects of the parameters_dict and form a new parameter dictionary of string elements of the parameters specific to that mode.

_convert_parameter_types is a top level check method which will set off multiple parameter specific methods which will check the interval and type of the parameter based on which ones are valid for this mode. It will update the _parameters_dict with the value of the parameters with types that allow range checking to be done and corrects the intervals of the parameter values. It will throw a ParameterError exception with the corresponding error message if an error occurs.

_do_range_checks is a top level check method which will set off multiple parameter specific methods which will check the range of the parameters against the ones specified in Table 7 of the Pacemaker.pdf documentation. It will throw a ParameterError exception with the corresponding error message if an error occurs.

_do_cross_checks is a top level check method which will set off multiple parameter specifc methods which will check the value of the parameters compared to others that are related to make sure that no two parameters are conflicting. This check is not run for every single parameter, only the ones that are related. It will throw a ParameterError exception with the corresponding error message if an error occurs.

Convert Parameter Types:

These functions are the specific parameter level checks for each parameter. They take the original string value from the dict, and all return the final converted value of the parameter. Each parameter will be rounded to the nearest increment specified in the documentation and checked if it can be converted into the necessary type for checking and submission. In all cases, if an error occurs, The _parameter_dict will be updated with an empty string for the value and a ParameterError will be raised.

Table 13: Private parameter specific type checks

Functions	**Purpose**	**Parameters**	**Return Type**	**Return Value**
_convert_lower_rate_limit_value	Incrments: From 30-50 ppm: 5ppm From 50-90 ppm: 1ppm From 90-175 ppm: 5ppm	value	Int	Converted lower rate limit value
_convert_upper_rate_limit_value	Increment by 5ppm	Value	Int	Converted url value

_convert_atrial_amplitude_value	Converts the aa into float and increments by 0.1	Value	float	Converted aa value
_convert_atrial_pw_value	Coverts apw into float and increments by 0.1	Value	Float	Converted apw value
_convert_ventricular_amplitude_value	Converts va into float and increments by 0.1	Value	Float	Converted va value
_convert_ventricular_pw_value	Converts vpw into float and increments by 0.1	Value	Float	Converted vpw value
_convert_vrp_value	Converts vrp into int and incrmenets by 10	Value	Int	Converted vrp value
_convert_arp_value	Converts arp into int and incrments by 10	Value	Int	Converted arp value
_convert_max_sensor_rate_value	Converts msr into int and increments by 5	Value	Int	Converted msr value
_convert_fixed_av_delay	Converts fad into int and increments by 10	Value	Int	Converted fad value
_convert_atrial_sensitivity_value	Converts as into float and increments by 0.1	Value	Float	Converted as value
_converted_ventricular_senstivity_value	Converts vs into float and increments by 0.1	Value	Float	Converted vs value
_convert_pvarp_value	Converts pvarp into int and increments by 10	Value	Int	Converted pvarp value
_convert_hysteresis_value	Converts hyst into int and increments by: From 30-50 ppm: 5ppm From 50-90 ppm: 1ppm From 90-175 ppm: 5ppm	Value	Int	Converted hysteresis value
_convert_rate_smoothing_value	Converts rs into int and increments by 3	Value	Int	Converted rate smoothing value
_convert_activity_threshold_value	Do nothing because choices are chosen by dropdown menu	Value	String	Value
_convert_reaction_time_value	Converts rt into int and increments by 10	Value	Int	Converted reaction time value
_convert_response_factor_value	Converts rf into int and increments by 1	Value	Int	Converted rf value
_convert_recovery_time_value	Convert recoverty time into int and increments by 1	Value	Int	Converted rt value

Range check:

These functions are the specific parameter level checks for each parameter. They take the now converted value from the parameters_dict, and check against the required range. In all cases, if an error occurs, The _parameter_dict will be updated with an empty string for the value and a ParameterError will be raised.

Table 14: Private parameter specific range checks

Functions	Purpose	Parameters	Return Type	Return Value
_check_lower_rate_limit_range	Lower rate limit must be between 30 and 175 ppm	value	N/A	None
_check_upper_rate_limit_range	Lower rate limit must be between 50 and 175 ppm	Value	N/A	None
_check_atrial_amplitude_range	Must be between 0 and 5.0	Value	N/A	None
_check_atrail_pw_range	Must be between 1 and 30	Value	N/A	None
_check_ventricular_amplitude_range	Must be between 0 and 5.0	Value	N/A	None
_check_ventricular_pw_range	Must be between 1 and 30	Value	N/A	None
_check_vrp_range	Must be between 150ms – 500ms	Value	N/A	None
_check_arp_range	Must be between 150ms – 500ms	Value	N/A	None
_check_max_sensor_rate_range	Must be between 50ppm – 175ppm	Value	N/A	None
_check_fixed_av_delay_range	Must be between 70ms – 300ms	Value	N/A	None
_check_atrial_sensitivity_range	Must be between 0V – 5V	Value	N/A	None
_check_ventricular_sensitvity_range	Must be between 0V – 5V	Value	N/A	None
_check_pvarp_range	Must be between 150ms - 500ms	Value	N/A	None
_check_hysteresis_range	Must be 0 or between 30 – 175ppm	Value	N/A	None
_check_rate_smoothing_range	Must be between 0% - 25%	Value	N/A	None
_check_activity_threshold_range	Must be one of 'V-Low', 'Low', 'Med-Low', 'Med', 'Med-High', 'High', 'V-High'	Value	N/A	None
_check_reaction_time_range	Must be between 10 sec – 50 sec	Value	N/A	None
_check_response_factor_range	Must be between 1 - 16	Value	N/A	None
_check_recovery_time_range	Must be between 2 – 16 min	Value	N/A	None

Cross validation checks:

These functions are the specific parameter level checks for each parameter. They take 2 of the now converted value from the parameters_dict and related parameters against each other to ensure they don’t conflict. In all cases, if an error occurs, The _parameter_dict will be updated with an empty string for the value and a ParameterError will be raised.

Table 14: Private parameter specific cross checks

Functions	Purpose	Parameters	Return Type	Return Value
_check_lower_rate_limit_vs_upper_rate_limit	Lower rate limit must be smaller or equal to the upper rate limit	Lrl, url	N/A	None

2.4 Serial Manager

This module contains all the functions that directly involve communication with the pacemaker device. Using the Python library *serial*, the DCM can read values sent by the pacemaker and write values to the pacemaker. This is used to upload the user's selected operating mode and programmable parameter values when the user submits. At the end of communication, the Serial Manager module receives a confirmation message from the pacemaker to ensure that the values have been properly uploaded to the device. The module is also responsible for plotting the electrogram information as this data is received through serial communication with the pacemaker. The plotting is done using the *matplotlib* library.

2.4.1 Module Interface Specification

Table 15: Public functions in the serial manager module

Functions	Purpose	Parameters	Return Type	Return Value
get_ports	Gets list of available ports	N/A	List[str]	List of names of available ports
init_serial	Initializes serial communication	newport: str	N/A	N/A
serial_out	Writes values to the pacemaker via serial communication	valid_parameters: dict[str, list[str]] parameters_dict: dict[str, Any] operating_mode: str	Boolean	True: successfully wrote values to device and received confirmation back False: did not successfully write values or receive confirmation back
is_plotting_egram	Returns Boolean value describing whether DCM is in egram-plotting mode	N/A	Boolean	True: currently plotting egram

				False: not plotting egram
display_egram	Creates a plot in a new window displaying egram data	mode: str	N/A	N/A

get_ports uses Python's *serial* library to get a list of available ports (datatype: ListPortInfo). It puts the names of all the ports into a list to return.

Init_serial initializes the port state variables. It sets the *port* variable equal to the port name passed in as a parameter and the variable *serialPort* to a new object of type *Serial* created using the port. In the case of an error, it produces the error message: "Unable to establish serial connection" in an error popup.

serial_out first checks if the connection has been initialized by checking the *is_open* flag on the *serialPort* state variable. It returns False if the connection has not been opened. Otherwise, it writes the start byte b'\x16' to signal to the pacemaker to start receiving information. It first sends the operating mode as an integer value (using the global state variable OPERATING_MODE), then proceeds to go through all the parameters in a specific order. If a given parameter is one of the programmable parameters for the selected operating mode (and is therefore included in the dictionary *valid_parameters*), the value is retrieved from *parameters_dict*, cast to the appropriate data type, and written to the port. In the case that the parameter is not required for the mode, a value of zero in the appropriate datatype is sent for that parameter. All data taken from the DCM is cast into the appropriate datatype for communication using the *numpy* library and is converted to a bytes object using the *.tobytes()* function. After all the parameters have been sent, it writes the end byte b'\x17' to signify the end of the communication and checks the port for a confirmation message to be sent by the pacemaker. The confirmation byte ensures that the parameters stored in the pacemaker are what the doctor inputted on the DCM. If the pacemaker sends the confirmation byte (value b'\x18'), the function returns True. Otherwise it returns False.

is_plotting_egram returns a flag stored as a state variable (continue_plotting) that is changed to True when starting the egram plotting and changed to False when the user closes the egram window. This is used to prevent the user from submitting parameters to the pacemaker when the pacemaker is still sending egram data to simplify serial communication.

display_egram is called by the GUI module when the user requests the egram plot. It takes one parameter *mode*, which can be equal to 'Atrium', 'Ventricle', or 'Both' depending on which type(s) of egrams they request. It first attempts to send the start signal b'\x16' to the pacemaker, which as described above for the *serial_out* function signifies to the pacemaker to start receiving information. It then sends b'\x26' which is the message that alerts the pacemaker that the DCM is requesting egram data. If the serial writes are successful, it sets the state variable *continue_plotting* to True to put the SerialManager object into plotting mode, then proceeds to call the private functions *_create_single_plot* (when mode is 'Atrium' or 'Ventricle') or *_create_double_plot* (when mode is 'Both'). In the case of an error, it prints an error message to the terminal to alert the user of a broken connection and returns.

2.2.2 Module Internal Design

Global State Variables

ACTIVITY_THRESHOLD (datatype: dict[str, int]):

> This is a lookup table used to convert the activity threshold values as strings (‘V-Low’, ‘Low’, ‘Med-Low’, etc.) to integer values. These integer values are used to communicate with the pacemaker.

ACTIVITY_THRESHOLD (datatype: dict[str, int]):

> This is a lookup table used to convert the operating modes as strings (‘aoo’, ‘voo’, ‘aai’, etc.) to integer values. These integer values are used to communicate with the pacemaker.

Object Instance State Variables

port (datatype: str):

> Name of the serial connection port.

serialPort (datatype: Serial):

> Serial object describing the port used for serial communication.

continue_plotting (datatype: boolean):

> This flag is used as the condition for a while loop in the plotting functions so that the egram data continues to be plotted as long as the value remains True. It is set to False on the close event of the egram window. It is also used by the GUI to prevent users from submitting parameters while the serial communication is being used for receiving the egram data.

Private Functions

Table 16: Private functions in the serial manager module

Functions	**Purpose**	**Parameters**	**Return Type**	**Return Value**
_create_single_plot	Creates a matplotlib figure with a single plot for either the atrium or ventricle egrams	mode: str	N/A	N/A
_create_double_plot	Creates a matplotlib figure with two subplots for the atrium and ventricle egrams	mode: str	N/A	N/A
_plot	Adds the data to the plot	x: float[] y: float[] data: float[]	data: float[]	Data list provided in parameters

		ax: SubplotBase (subclass of Axes) mode: str		updated with new values
_on_close	Handles the close event of the egram window	N/A	N/A	N/A

_create_single_plot creates a matplotlib figure of size 13x6. It initializes arrays for the x- and y-values, each with a length of 100. The x-values are equally spaced floating-point values from 0 to 1 which are only used for the purpose of plotting the y-values equally spaced out (and consequently the x-axis label visibility is set to False on the actual plot). The y-values are initialized to zeros and the y-axis limits are set from -5 to 5. The data array is initialized to an empty array. In a while loop, data is constantly being read using the Serial class and appended to the y-value array. The _plot function is then called to update the plot and the arrays. The while loop runs as long as the state variable *continue_plotting* is True.

_create_double_plot creates a matplotlib figure of size 13x12. It creates the x-value, y-value, and data arrays in the same manner as _create_single_plot, but two of each are created: one for the atrium data and the other for the ventricle data. The data is received as a 16-byte block, where the first 8 bytes corresponds to atrium data and the second 8 bytes corresponds to ventricle data. The while loop also functions the same way, with the only difference being that the _plot function is called twice to update the atrium and ventricle data arrays separately.

_plot first checks if data is an empty array. In this case, the plot is initialized with the x and y arrays, sets the title based on the egram mode to 'Atrium Electrogram' or 'Ventricle Electrogram', and makes the plot visible. Outside the if-block, it updates the data array. This array is linked to the plot and consequently this adds the new value to the graph. It then pauses for 100 ms to prevent the plotting while loops from running too quickly. Finally, it returns the updated data array to be used in the next iteration of the loop.

_on_close is linked to the figure and is called on the close event (when the window is closed). It sends b'\x55' to the pacemaker, which is interpreted by the device as the stop command for the egram. It also sets the state variable *continue_plotting* to false, which stops the while loops in the plotting functions from running.

3 Testing

3.1 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.

All the unit tests described in *Tables* _____ passed.

Table 17: Unit tests for database module

Description	**Input**	**Expected Behaviour**

create_user		
Creates new user in empty database given username and password	• Username: user Password: pass • Database is empty at beginning of test	• User with username *user* and password *pass* added to database • Database has length 1 • Returns True
Creates new user in non-empty database given unique username and password	• Username: user1 Password: pass • Database contains user from previous test	• User with username *user1* and password *pass* added to database • Database has length 2 • Returns True
Doesn’t create new user in database given non-unique username and password	• Username: user Password: password • Database contains users from previous two tests	• No users added to database • Database has length 2 • Returns False
get_user		
Returns user if username is found in database	• Username: user • Database contains users from create_user tests	• Returns user with username *user* and password *pass*
Returns None if username is not found in database	• Username: user2 • Database contains users from create_user tests	• Returns None
Returns None if username has different capitalization as in database	• Username: USER • Database contains users from create_user tests	• Returns None
get_number_of_users		
Returns 0 if database is empty	• Database is empty at beginning of test	• Returns 0
Returns number of users if database is not empty	• Database contains users from create_user tests	• Returns 2
update_parameters		
Updates a single parameter without changing other values	• Username: user Parameters: {'lower_rate_limit': 65, 'upper_rate_limit': 120, 'atrial_amplitude': 3.5, 'atrial_pw': 0.4} ○ lower_rate_limit changed from default value of 60. Other values unchanged • Database contains first user from create_user tests in default state (no	• lower_rate_limit updated in database for user

	previous changes to user information)	
Updates multiple parameters	• Username: user Parameters: {'lower_rate_limit': 65, 'upper_rate_limit': 130, 'atrial_amplitude': 3.5, 'atrial_pw': 0.4} ○ lower_rate_limit and upper_rate_limit changed from default values • Database contains first user from create_user tests in default state (no previous changes to user information)	• lower_rate_limit and upper_rate_limit updated in database for user
Updates parameters for only user when there are multiple users	• Username: user1 Parameters: {'lower_rate_limit': 65, 'upper_rate_limit': 130, 'atrial_amplitude': 3.5, 'atrial_pw': 0.4} ○ lower_rate_limit and upper_rate_limit changed from default values • Database contains both users from create_user tests in default state (no previous changes to user information)	• User with username *user* remains unchanged • User with username *user1* updated with new lower_rate_limit and upper_rate_limit
	update_operating_mode	
Updates operating mode when not previously set	• Username: user Mode: aoo • Mode for user in database previously stored as blank • Database contains first user from create_user tests in default state (no previous changes to user information)	• Mode for user updated from blank to aoo in database

Updates operating mode when previously set to different mode	• Username: user Mode: voo • Database contains user from previous test (mode set to aoo)	• Mode for user updated from aoo to voo in database
Updates only operating mode of the correct user when there are multiple users	• Username: user1 mode: aoo • Mode for user1 in database previously stored as blank • Database contains both users from create_user tests in default state (no previous changes to user information)	• Mode for user with username *user* remains unchanged • Mode for user with username *user1* updated from blank to aoo

Table 18: Unit tests for validation module

Description	**Input**	**Expected Behaviour**
	Constructor	
Empty Parameters dict check	• Empty dict for parameters_dict	• Error message pops up in GUI explaining the error
More parameters than needed passed in	• Large list of parameters for every mode	• Only the needed parameters by the valid mode are outputed
Valid_parameters contains unspecified parameter	• Valid parameters list contains string for parameter that does not exist	• Error message pops up in GUI explaining the error
	_find_parameters_for_mode	
Empty Parameters dict check	• Empty dict for parameters_dict	• Error message pops up in GUI explaining the error
More parameters than needed passed in	• Large list of parameters for every mode	• Only the needed parameters by the valid mode are outputed
Valid_parameters contain unspecified parameter	• Valid parameters list contains string for parameter that does not exist	• Error message pops up in GUI explaining the error
	_convert_*_value (done for each parameter)	
Invalid string input from user	• A character value is passed in where string conversion is needed to the value parameter	• Parameter error message pops up in GUI saying input must be number

Invalid enumeration passed for activity_threshold	• Pass in a string that is not one of the enumerations	• No error thrown, will catch in next check
Negative value is inputed for numeric parameters	• Negative numeric values passed in the value	• Converted in to the correct interval with no errors, will be caught in next check
Empty string user input	• An empty string value is passed in where string conversion is needed to the value parameter	• Parameter error message pops up in GUI saying input must be number
Valid entry	• A valid number is passed in to the check	• Will save the converted value back to the parameter dict
	_check_*_range	
Out of range value	• Value outside of defined range for the parameter is passed in	• ParameterError out of range error is thrown and converted into GUI message to be displayed to the user. The parameter on the GUI is reset to empty
Negative value	• Negative value is passed into the parameter check	• ParameterError out of range error is thrown and converted into GUI message to be displayed to the user. The parameter on the GUI is reset to empty
Undefined string value for activity_threshold value	• String that is not a part of enumeration value is sent to activity threshold range check	• ParameterError out of range error is thrown and converted into GUI message to be displayed to the user. The parameter on the GUI is reset to default value
Valid entry	• Valid in range entry passed into the check	• No error is thrown and returns to top level check method
	*_cross_check_*_vs_**	
Invalid entry	• Passes in values which violates the cross-check conditions	• ParameterError out of range error is thrown and converted into GUI message to be displayed to the user. The parameter on the GUI is reset to default value for both parameters
Valid entry	• Passes in values which do not violate the cross-check conditions	• No error is thrown and returns to top level check method

String entry	• Pass in character based string entry to as one of the parameters for checking	• Will be caught by convert_*_value and will send corresponding error message

Table 19: Unit tests for serial manager module

Description	**Input**	**Expected Behaviour**
	get_ports	
Gets empty list given no device connection	• No device connected	• Returns empty list
Gets list of ports given a device connection	• Device connected (COM4, COM5)	• Returns list of strings: ['COM4', 'COM5']
	Init_serial	
Establishes new serial connection	• newport: string • String corresponds to actual available port	• Updates port address and instantiates serial object
Attempts to establish serial connection with invalid port name	• newport: string • String does not correspond to available port	• Returns error message, indicating no serial connection established
	serial_out	
Transmits parameters given device connection	• valid_parameters: dict[str, list[str]] • parameters_dict: dict[str, Any] • operating_mode: str	• Transmits appropriate parameters to pacemaker • Receives confirmation message
Attempt to pass invalid parameters	• valid_parameters: dict[str, list[str]] • parameters_dict: dict[str, Any] • operating_mode: str • parameters_dict contains invalid parameters	• Returns error message, indicating parameters not sent
Attempts to transmit parameters given no device connection	• valid_parameters: dict[str, list[str]] • parameters_dict: dict[str, Any] • operating_mode: str • the serialmanagers serialPort object has not been instantiated	• Returns error message indicating no serial connection
	display_egram	

Creates a single plot	• mode: 'Ventricle'	• A window with a single plot is created
Creates a double plot (both atrium and ventricle)	• mode: 'Both'	• A window with two plots is created
Prints message to terminal if unable to connect to device	• mode: 'Ventricle' • No connection set up (GUI blocks egram if no connection, but the test ensures that the DCM remains stable if connection is broken)	• Message printed to terminal: "Unable to receive egram data from pacemaker" • No new window created
	_create_single_plot	
Plots the atrium electrogram	• mode: 'Atrium'	• A plot is created with the title 'Atrium Electrogram' • The x-axis label is set to hidden • The y-axis limits are set to [-5,5] • The plot is updated with new data while the window remains open
Plots the ventricle electrogram	• mode: 'Ventricle'	• A plot is created with the title 'Ventricle Electrogram' • The x-axis label is set to hidden • The y-axis limits are set to [-5,5] • The plot is updated with new data while the window remains open
	_create_double_plot	
Plots both the atrium and ventricle electrograms	• N/A	• A figure with two subplots is created, the first with the title 'Atrium Electrogram' and the second with the title 'Ventricle Electrogram' • The x-axis labels on both subplots are set to hidden • The y-axis limits on both subplots are set to [-5,5]

		• Both subplots are updated with new data while the window remains open
	_plot	
Initializes plot if data is an empty array (first iteration of plotting while loop)	• x: array of floating-point values from 0 to 1 • y: array of zeros with one new value • data: empty array • ax: figure axis • mode: 'Atrium'	• Creates a plot of zero y-values • Sets title to 'Atrium Electrogram' • Data array updated with first value
Updates data array if data is a non-empty array (later iterations of plotting while loop)	• x: array of floating-point values from 0 to 1 • y: array of random floating-point values and one new y-value • data: array of values corresponding to previous y-values • ax: figure axis • mode: 'Atrium'	• Plot updated with new y-value • Data array updated with new value

3.2 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, parameter validation, and serial manager modules.

Table 20: Integration test cases starting in logged out state

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 • No mode selected • GUI displays logged in, device disconnected state	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 blank password	• Error message: Username and password cannot be empty. • No new user added to database	Pass

	• Stays on Register screen	
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 • GUI displays disconnected state	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

Table 21: Integration tests starting in logged in, device disconnected state (i.e. after test case #1 in Table 19)

Test Case	**Requirements**	**Pass/Fail**
Logged in on newly created user, connect device, click Refresh button, select port COM5, click Connect button	• Port becomes available in dropdown after clicking Refresh • After clicking Connect, DCM displays device connected state ○ "Device disconnected" message removed ○ Electrogram buttons become available	Pass
Device already connected, logged in on newly created user, select port COM5, click Connect button	• Port available immediately in dropdown • After clicking Connect, DCM displays device connected state ○ "Device disconnected" message removed ○ Electrogram buttons become available	Pass
Logged in on previously created user who previously uploaded operating mode (AOO) and parameter values (lower rate limit changed to 65 from default value of 60)	• AOO operating mode already selected upon login • Programmable parameters prepopulated with previously submitted values (lower rate limit is 65)	Pass
Logged in on newly created user, click submit	• Under status, DCM displays the message: "Device disconnected. No device data available" • All operating modes are in unselected state	Pass

	• Popup with error message: “No operating mode has been selected”	
Logged in on newly created user, click AOO, submit	• Under status, DCM displays the message: “Device disconnected. No device data available” • AOO operating mode in selected state, other operating modes in unselected state • Popup with error message: “No device connected”	Pass

Table 22: Integration tests starting in logged in, device connected state (i.e. after test case #1 in Table 20)

Test Case	**Requirements**	**Pass/Fail**
Click Disconnect button	• DCM returns to device disconnected state	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	Pass
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

The integration tests described in Table 23 were performed using an altered version of the Simulink file with two main differences from the actual file:

- LED states were added to the logic. The LED was used to determine that the device was in the correct state at any given time. This was done specifically to check that the pacemaker stops sending unnecessary data.
    - Blue: sending egram data
    - Red: not sending egram data
- The atrial signal was replaced by a constant value of 1.5 and the ventricle signal was replaced by a constant value of 3.5. This was done to test that the DCM can correctly and continuously read the exact value that the pacemaker sent.

Table 23: Integration tests involving the electrogram (starts in logged in, device connected state)

Test Case	**Requirements**	**Pass/Fail**
Start Atrium egram, wait, close egram	• Creates new window with one plot with title 'Atrium Electrogram' • Plot is a horizontal line at y=0 when window first opens • Value of 1.5 continuously gets added to the end of the plot • LED on pacemaker is blue while egram is open and red when closed	Pass
Start Ventricle egram, wait, close egram	• Creates new window with one plot with title 'Ventricle Electrogram' • Plot is a horizontal line at y=0 when window first opens • Value of 3.5 continuously gets added to the end of the plot • LED on pacemaker is blue while egram is open and red when closed	Pass
Start Both (A+V) egram, wait, close egram	• Creates new window with two plots with title 'Atrium Electrogram' and 'Ventricle Electrogram' • Both plots have horizontal line at y=0 when window first opens • Value of 1.5 continuously gets added to the end of the atrium plot • Value of 3.5 continuously gets added to the end of the ventricle plot • LED on pacemaker is blue while egram is open and red when closed	Pass
Start Atrium egram, go back to DCM home page window without closing egram window and click Submit button	• Popup with error message: "Exit egram before submitting parameters."	Pass

4 Next Steps

4.1 Requirements

More parameters or pulsing modes can be introduced in the future. When more parameters and modes are introduced, modification on the GUI and parameter checking structure is minimal because the code has been designed with scalability in mind and can be expanded with new modes simply by adding the GUI element and the corresponding requirements for checking the validity of the parameter in the checking class.

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.

4.2 Design Decisions

As the largest module, there is some need for further modularization within the GUI module. The four pages (welcome, user creation, login, home page) each have individual logic that is not related to the other pages. The DCM home page particularly has a lot of complex logic related to the pacing modes, programmable parameters, device connection, electrogram, etc. which is unrelated to the other pages. These four pages could be divided into individual modules with a fifth module that controls movement between the pages.

Additionally, especially if the DCM was to be expanded with more pacing modes and programmable parameters, simplifying the submission interface can prevent overcrowding of the page and make the DCM easier to use. For instance, rather than a list of buttons, the pacing modes could be put on a separate page from the programmable parameters, a scroll feature could be added so that not all buttons are displayed at once, or a dropdown could be used to select the operating mode instead of buttons.

The Serial Manager module currently consists of two groups of functions with no overlap between the two: those that handle the upload of parameters and operating mode to the pacemaker and those that handle the electrogram. The electrogram logic was added to this module as it relies on serial communication to obtain new values to plot in real-time. However, plotting is not inherently related to serial communication and so it can be separated into another module to which values are passed in by the Serial Manager module to plot. More than half of the functions in the Serial Manager module are solely related to plotting so there is enough complexity to justify a separate module.

References

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