DCM-Pacemaker Documentation

Part 1

Requirements

The requirements for the pacemaker project are carefully divided between the graphical user interface (GUI) and the system implementation, which is carried out through Simulink. This deliverable places a specific focus on the GUI requirements.

In terms of the GUI, our objectives are multifaceted. We aim to create a welcoming screen that offers two fundamental functionalities: user registration and user login. Users will have the option to register themselves using a chosen username and password. The system is designed to locally store information for up to 10 users, providing a practical and efficient experience. Additionally, the GUI will facilitate access to essential pacing modes, including AOO, VOO, VVI, and AAI.

Furthermore, the GUI requirements for the final product encompass several key components. Firstly, it must possess the capability to create and manage various windows, accommodating the display of text and graphics. It should seamlessly process user interactions and button inputs, making the user experience intuitive and user-friendly. The interface should offer access to all programmable parameters for users to review and modify, ensuring complete control and customization. An indicator will be prominently displayed, providing real-time feedback on the device's connectivity status, whether it is connected or not. In cases where telemetry is lost, possibly due to the device being out of range or encountering interference, the GUI should communicate this to the user. Additionally, it will be able to detect when a different pacemaker comes into range, allowing users to distinguish between different devices and make informed choices.

In terms of the Python requirements, they are conveniently outlined within a requirements.txt file that contains all the necessary packages and dependencies. The README of the project provides clear instructions for users to install these packages, excluding those that are already built-in. To ensure a seamless installation process, it is recommended to run the installation as a Python module to maintain consistency with the Python interpreter in use.

Our collaborative development approach adheres to a set of strict formatting guidelines. We employ tools such as the Black code formatter and linting utilities like Pylint to enforce industry-standard coding practices and maintain adherence to PEP 8 conventions. This approach ensures that our codebase remains consistent and well-structured across all contributors involved in the project. Also, issues were utilized to assign and document all tasks needed to be completed with their labels added.

Specific instructions were written in a README for the user to be able to easily run the program without any confusion. The project paths were configured relative to the root directory of the entire project so that the user never needs to change their directory to run anything.

For a streamlined version control environment, we rely on a well-defined .gitignore and gitattributes template. This configuration is adept at filtering out unnecessary build and cache files, allowing us to maintain a clean and focused codebase. Additionally, binary file types are explicitly outlined, ensuring that version control operations are consistent and clearly understood by Git. In this setup, we prioritize uniform line endings, particularly by setting them to "auto," a practice that minimizes potential conflicts with different file types and platforms, such as bash scripts and Unix-based files, which are stored in LF format and can be automatically converted to CRLF to prevent any Git-related issues. Furthermore, the main branch was protected to require a pull request each time something was to be contributed to avoid corruption of the code base. For this, different branches were made with a naming convention of dcm or simulink as the prefixes.

Design Decisions

More requirements outlined with their design decisions:

The choice of Python as the programming language for our application was driven by the availability of rich libraries and its ease of implementation. In this context, there was no compelling need for strong performance requirements, negating the necessity for low-level languages. Operating the application natively on a computer, as opposed to web-based deployment, was favoured for security considerations.

For user management, the application supports up to 10 registered users. A simple entry mechanism was implemented for user input. During the registration process, stringent criteria were enforced to ensure data integrity and security. Usernames were required to exclude special characters to maintain data consistency. Case insensitivity was enforced to prevent users from registering under the same name with variations in capitalization, which could lead to database ambiguities. Passwords were expected to meet specific security standards, permitting special characters and disallowing the use of the username. Minimum length requirements were imposed to enhance password security.

The user interface leveraged the Tkinter library to enable a user-friendly interaction, incorporating clickable buttons and input fields. To enhance the user experience, keybindings were added, enabling users to perform common actions with ease by mapping keys such as Enter and ESC to continue and navigate the application.

Additionally, the application was designed to include a dedicated egram tab. This new screen class served as a foundation for accommodating future egram-related functionality. This approach allows for seamless updates in response to evolving requirements while maintaining a robust prototype for future stages of development.

To provide clear and informative feedback, indicators were integrated into the application. These indicators employed colour-coding to convey important information. For instance, the program and the user can readily identify the connected pacemaker by its unique ID. An associated label changes colour (red for newly detected devices and green for previously interrogated ones) to alert the user to any changes in the connected device. In the absence of a connection, a neutral grey status is displayed. Furthermore, the application provides real-time connection status feedback, with a green label indicating a successful connection and a red label signifying the absence of a connection.

In summary, our choice of programming language, user management protocols, user interface design, and indicators were all implemented with precision to fulfil our application's requirements effectively and deliver a secure and user-friendly experience. The design principles incorporated into the application ensure adaptability to future requirements and maintain data integrity and security.

The implementation of the next set of requirements, pertaining to pacing modes, was realized through the settings page. This was executed by initially allowing the user to select their desired pacing mode from a user-friendly drop-down menu. This approach provides an accessible overview of the available pacing modes. Subsequently, the pacing mode entries, complete with units for clear comprehension and conversion reference, were presented to the user.

Recognizing that most users might not be well-versed in the specific valid intervals for these parameters, a thoughtful inclusion was made. Users were provided with a drop-down menu containing predefined ranges. This intuitive feature empowers users to increment or decrement values using a custom widget tailored for this purpose. By implementing fixed increments, we ensured that users are not burdened with the complexities of manual entry and the potential frustration of discovering that their input was invalid.

The design philosophy also encompasses a thoughtful user experience. Users have the option to either apply their changes, which updates their database and makes these values accessible for future sessions, or to close the settings page, effectively discarding any unintended modifications. Additionally, an "OK" button was included, mirroring the behaviour found in the Windows settings interface. In cases where users leave these entries empty, we have chosen a more user-friendly approach. The default values, based on the documentation, are automatically populated, providing users with a nominal starting point to refine according to their preferences.

To ensure robust validation of these viewable parameters, a dedicated widget was instantiated for each parameter, allowing for precise control over parameter limits while reusing the same class to create different parameter entries. Within this widget, a drop-down menu was thoughtfully designed to present the available ranges, including specific increments tailored for precision in different intervals. The parameter checker efficiently manages these increments, relying on a dictionary of tuples that package interval-increment pairs. This approach allows the widget to add the corresponding increment based on the value selected. To accommodate floating-point values and maintain clarity for the user, the numpy library was judiciously employed to facilitate floating-point ranges and increments when required. This meticulous design not only minimises user errors but also offers an intuitive and seamless parameter entry experience.

	Pygame	Tkinter	Web technologies
Accessibility of interface	7	8	10
Support for necessary features	6	6	5
Ease of implementation	5	5	7
Security	10	10	2

A secondary decision matrix was used to determine the significance of each defining characteristic used in the initial decision matrix

Parameter	Weighting factor	Justification

Accessibility of interface	0.3	The DCM only needs to be accessed in specific clinical situations where supply of the necessary infrastructure to handle inaccessibility is non-concern. High accessibility may also cause security concerns for users; Recent implementations of web-based DCMs have caused pacemakers to get hacked.
Support for necessary features	1	The used technology has to be capable of doing everything needed of a DCM, including communication with the pacemaker. Popular, well supported software environments and libraries ensure higher reliability and future compatibility
Ease of implementation	0.8	Simpler implementations minimise the incurrence of tech debt, improving reliability of software and minimising development time.
Security	1	Hacking of pacemakers is a real problem faced by users. The safety of the device is critical to widespread adoption.

These decision matrices were combined to compute the final scores of each respective candidate technology.

	Pygame	Tkinter	Web technologies
Accessibility of interface	7*0.3 = 2.1	8*0.3 = 2.4	10*0.3 = 3
Support for necessary features	6	6	5
Ease of implementation	5*0.8 = 4	5*0.8 = 4	7*0.8 = 5.6
Security	10	10	2
Total	22.1	22.4	15.6

Therefore, we elected to use Tkinter for the DCM module due to its higher score.

In shaping the design of our application, we have carefully orchestrated a user-centric experience. Our application opens to a login screen, offering users the choice to either log in with their credentials or embark on the registration process for new users. Upon successfully logging in, users are seamlessly transitioned to the main homepage, serving as a central hub for accessing the program's comprehensive array of features. These features include settings configuration, egram data, pacemaker connection status, pacemaker identification, pacing mode selection, and a readily available logout button for user convenience.

To optimise user interaction, our design architecture includes a dedicated settings page. This separation was deliberate, given that each pacing mode entails distinct and unique settings. Consequently, users can efficiently tailor their pacemaker's parameters on the main screen by selecting their pacing mode and subsequently accessing the relevant settings.

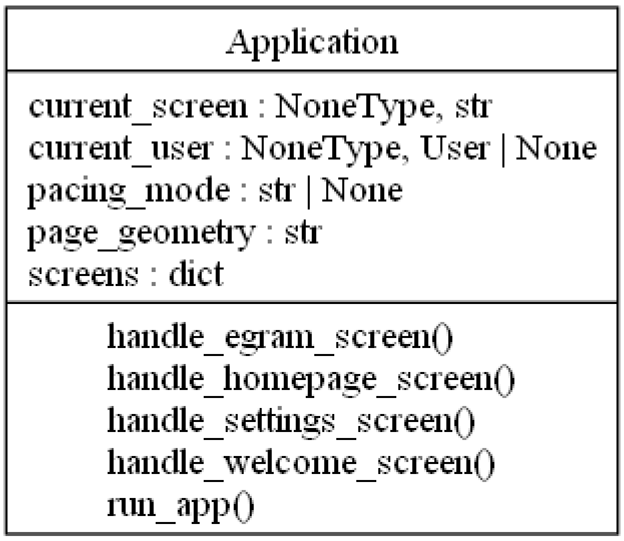
In terms of organising the user interface (UI) elements, our design strategy favours Tkinter's grid layout functionality over the 'pack' method. This choice has enabled us to maintain an organised and structured UI layout, enhancing the overall user experience. Furthermore, we've implemented a dynamic button sizing mechanism based on the screen's dimensions. This approach ensures a responsive UI that adapts to various screen sizes, although it necessitates intricate calculations for precise element placement.

Another important design consideration involves our approach to list enumeration. Recognizing the need for efficiency, we've implemented simultaneous enumeration across multiple lists, optimising data processing and traversal. However, this technique requires meticulous attention to the lengths of these lists and synchronisation to function effectively. Conditional checks have also been implemented to handle enumerated values intelligently, enhancing the robustness of our application.

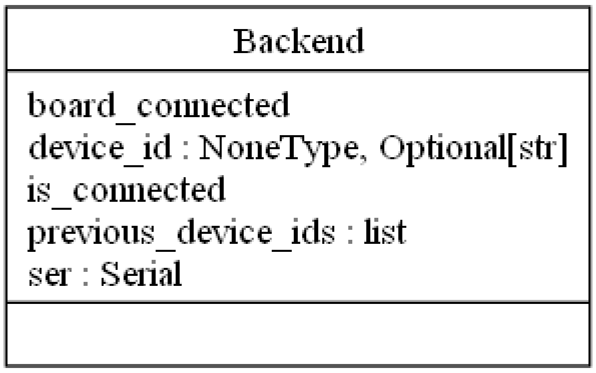
To manage the flow and structure of our application, we've integrated a Finite State Machine (FSM). This feature restricts users from opening multiple windows simultaneously, ensuring a well-defined approach to state transitions and user interactions.

In anticipation of future requirements for the second part of this project, we've adopted an object-oriented design for our application. Each facet of our solution has been meticulously designed to accommodate scalability and adapt to evolving demands. This modularity empowers us to easily debug, test, and modify code when necessary. Our application is organised around a class-based structure, with each screen inheriting from a base class aptly named "Screen."

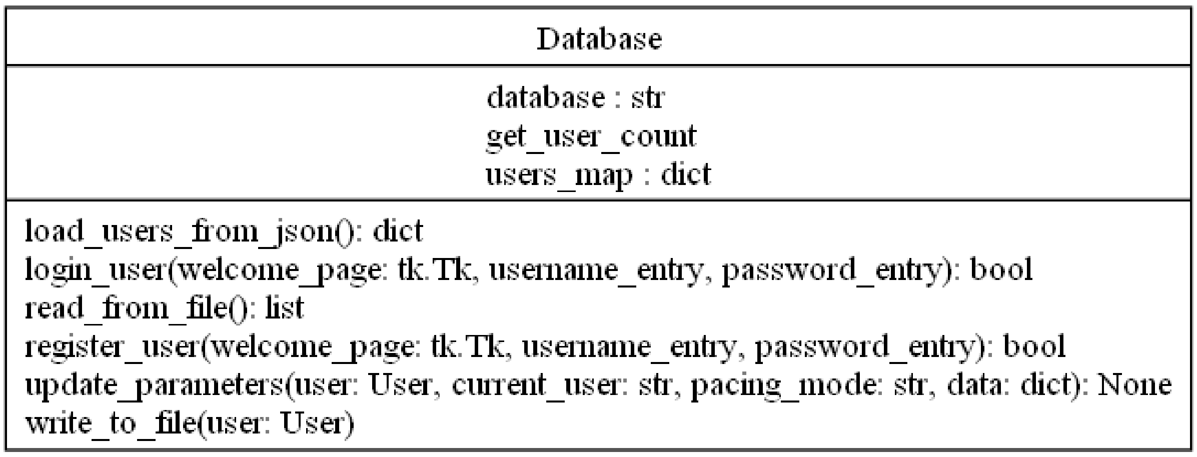
For a more visual representation of our object-oriented application, we've crafted UML diagrams to illustrate our design approach. In these diagrams, each class is identified at the top of a box, followed by the variables and their corresponding data types specific to that class. Beneath that, you'll find a concise list of the functions encapsulated within each class, providing a comprehensive overview of our design.



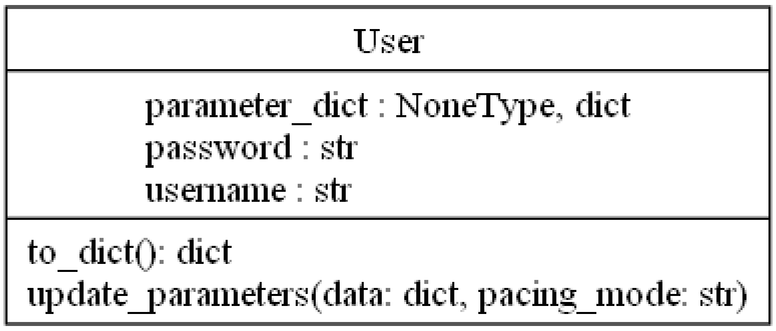
Within our application class, we've made a judicious decision to incorporate robust error handling mechanisms. This strategic choice serves a pivotal role in shielding the application from abrupt crashes by gracefully managing unexpected issues. By doing so, we fortify the system's stability, as this proactive approach is instrumental in addressing potential error scenarios. Our commitment to error handling underscores our dedication to delivering a resilient and reliable application that remains steadfast even when confronted with unforeseen challenges. This class contains the handlers for all of the screens and works as an FSM for the application by executing the screen flow from the screens class in a manner suitable for the application. A consistent init was used to initialise these screens with a standard format so that all can match as intended and transition based on the screens imported.


Backend
board_connected
device_id : NoneType, Optional[str]
is_connected
previous_device_ids : list
ser : Serial

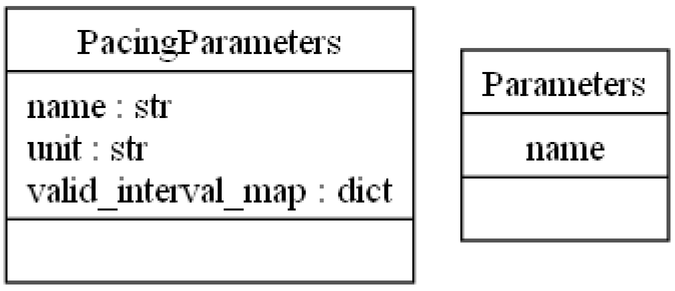
The backend class of the application was set up to check for serial connections and determine properties such as if the board is connected or not. Based on this other labels were set up on the application to note down different requirements. This is left in a class to make the functionality for those labels very easy to adjust. These will require changes as the next stage approaches hence this standard class template was set up in a manner that it can be modified when needed.

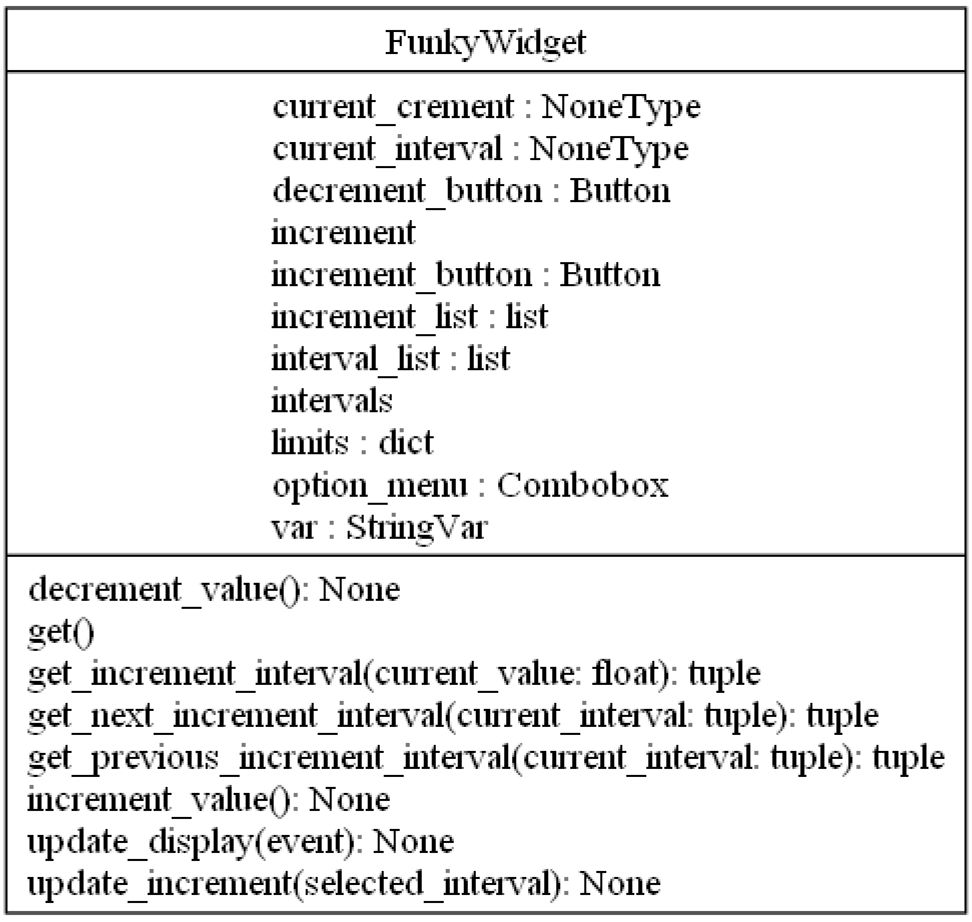

Database
database : str
get_user_count
users_map : dict
load_users_from_json(): dict
login_user(welcome_page: tk.Tk, username_entry, password_entry): bool
read_from_file(): list
register_user(welcome_page: tk.Tk, username_entry, password_entry): bool
update_parameters(user: User, current_user: str, pacing_mode: str, data: dict): None
write_to_file(user: User)

For our database we also opted to use a json file to store all user information instead of a traditional text file. With json files, it was much easier to store user information under a user key than a text file, hence why we chose to use a json file.

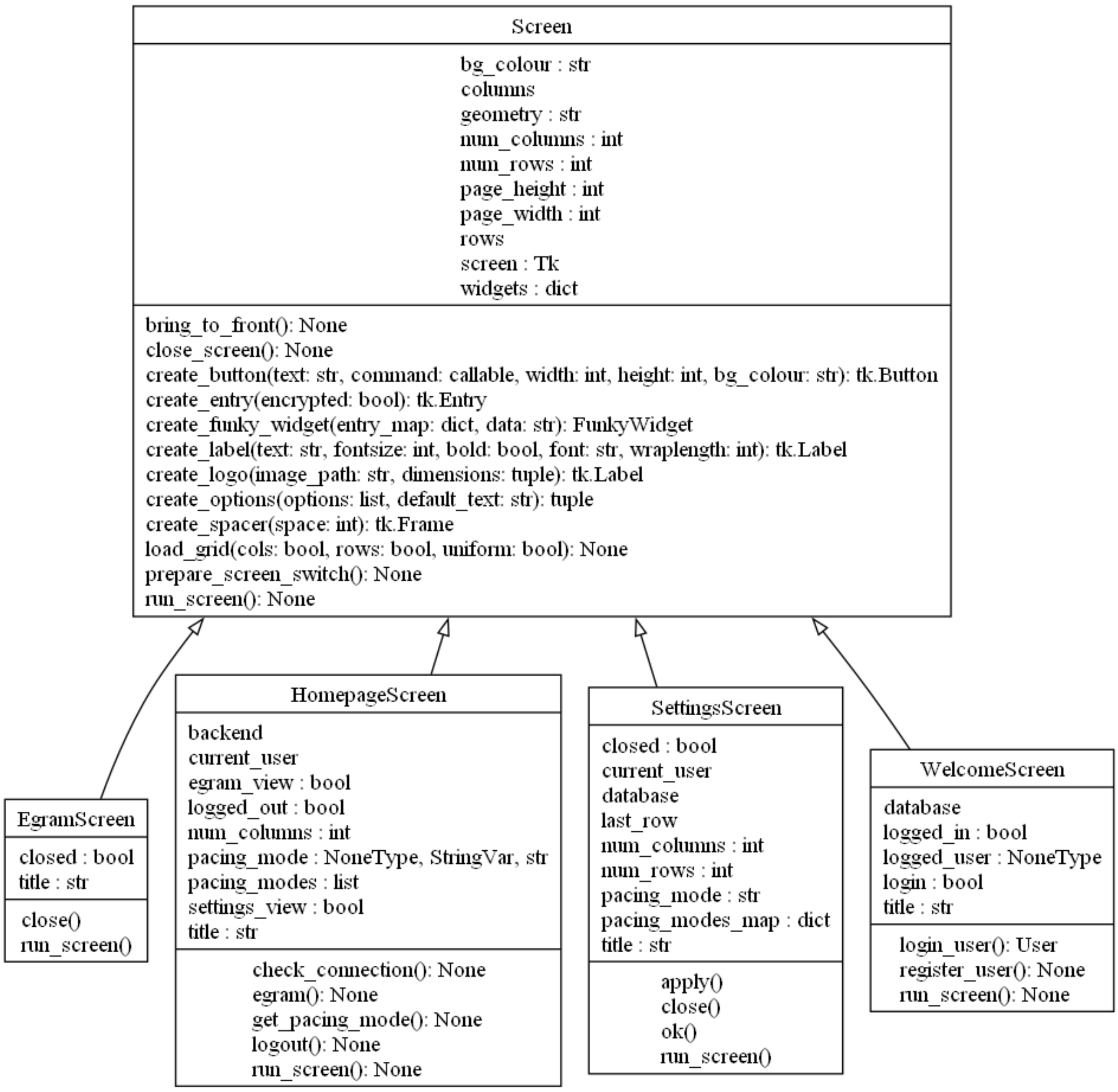

User
parameter_dict : NoneType, dict
password : str
username : str
to_dict(): dict
update_parameters(data: dict, pacing_mode: str)

We decided that a user class was necessary to organise the user data, and the associated parameters effectively. This provided us with a structured approach for managing user-specific information.

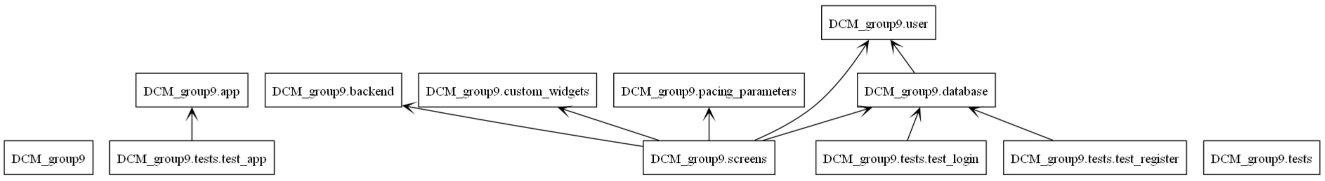



FunkyWidget
current_crement : NoneType
current_interval : NoneType
decrement_button : Button
increment
increment_button : Button
increment_list : list
interval_list : list
intervals
limits : dict
option_menu : Combobox
var : StringVar
decrement_value(): None
get()
get_increment_interval(current_value: float): tuple
get_next_increment_interval(current_interval: tuple): tuple
get_previous_increment_interval(current_interval: tuple): tuple
increment_value(): None
update_display(event): None
update_increment(selected_interval): None

The "FunkyWidget" class is used for the custom widgets we have implemented inside our "SettingsScreen." These custom widgets include the increment and decrement buttons. This class provides a tailored solution for the project's unique requirements, but it required additional implementation effort compared to standard widgets. We also needs to efficiently store valid intervals and increments for the various parameters. This streamlined access to parameter data for validation and processing, but handling these floating-point precision can be intricate, requiring specific considerations, so we also added additional code to manage these precision concerns.



Since each screen shares some of the same basic properties, we decided to have each unique screen inherit those properties from our base class called "Screen." This means we can cut down on how many lines of code we have and how much memory the application uses.


DCM_group9.user
DCM_group9.app
DCM_group9.backend
DCM_group9.custom_widgets
DCM_group9.pacing_parameters
DCM_group9.database
DCM_group9
DCM_group9.tests.test_app
DCM_group9.screens
DCM_group9.tests.test_login
DCM_group9.tests.test_register
DCM_group9.tests

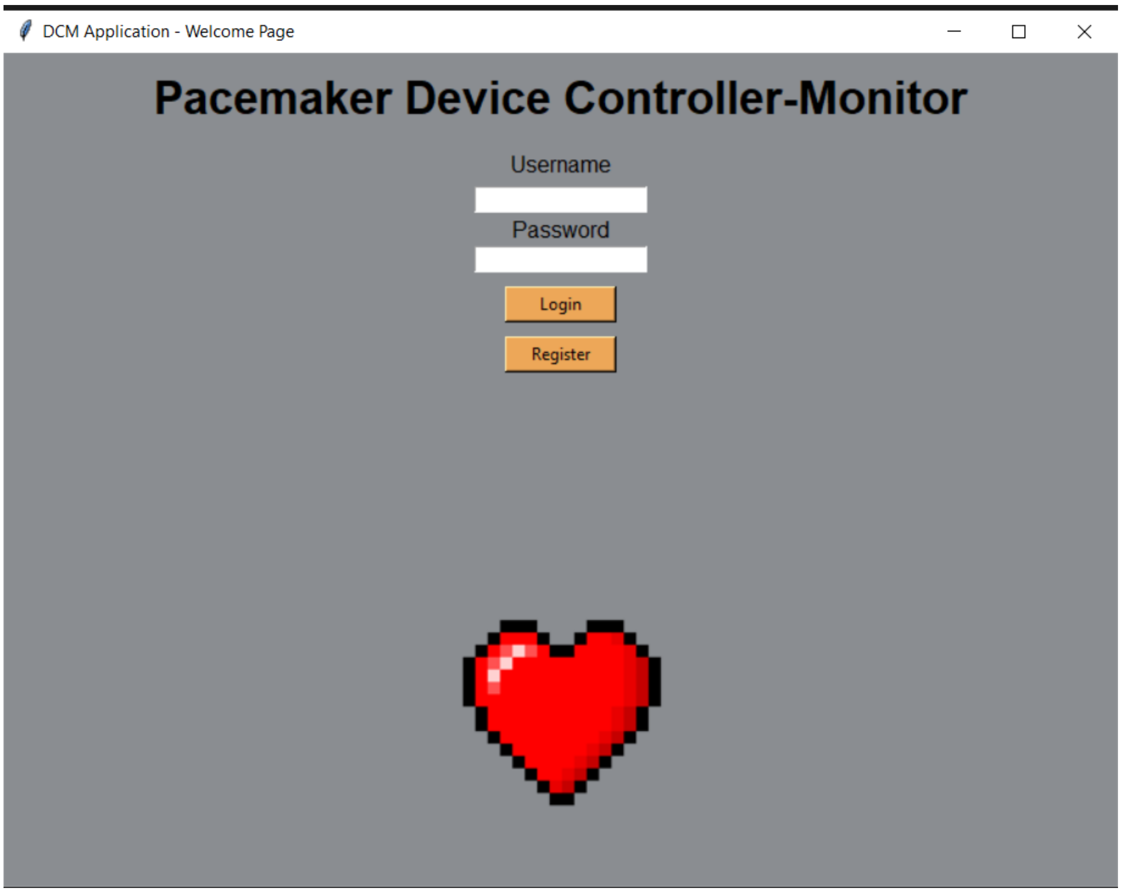
The above figure represents the modules and their communication with each other. Each file here was initialised with an __init__ file to allow python to detect these as modules. The submodules are noted with the dot before the name. We can see that the tests module exists on its own without being imported to anything else as they stand alone. We then have the screens class that is receiving all of the classes required to operate each screen, using information from the backend, pacing parameters, user, database, and widgets classes. The user is at the top of the diagram as it is also used within the database and then in classes that have Database as well for screens.

App Screenshots

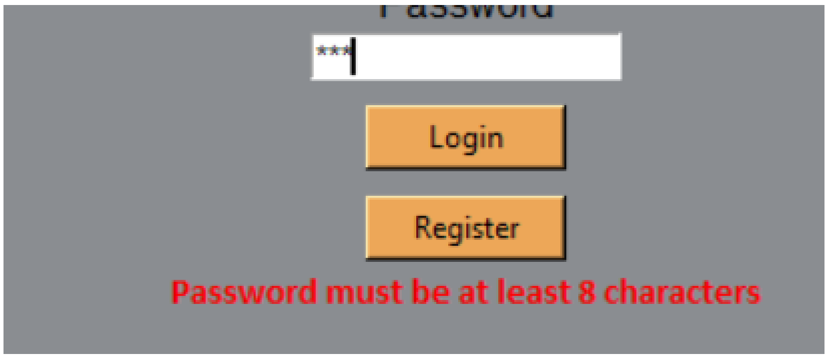
An example entry in the JSON database:

```
{
        "user": {
            "password": "password",
            "pacing_mode_params": {
                "AOO": {
                    "Lower Rate Limit": 60.0,
                    "Upper Rate Limit": 120.0,
                    "Atrial Amplitude": 3.5,
                    "Atrial Pulse Width": 0.4
                },
                "AAI": {
                    "Lower Rate Limit": 60.0,
                    "Upper Rate Limit": 120.0,
                    "Atrial Amplitude": 3.5,
                    "Atrial Pulse Width": 0.7,
                    "ARP": 250.0
                },
                "VOO": {
                    "Lower Rate Limit": 60.0,
                    "Upper Rate Limit": 120.0,
                    "Ventricular Amplitude": 3.2,
                    "Ventricular Pulse Width": 1.2
                },
                "VVI": {
                    "Lower Rate Limit": 125.0,
                    "Upper Rate Limit": 120.0,
                    "Ventricular Amplitude": 3.5,
                    "Ventricular Pulse Width": 1.5,
                    "VRP": 300.0
                },
                "AOOR": {
                    "Lower Rate Limit": 60,
                    "Upper Rate Limit": 120,
                    "Atrial Amplitude": 3.5,
                    "Atrial Pulse Width": 0.4
                },
                "AAIR": {
                    "Lower Rate Limit": 60,
                    "Upper Rate Limit": 120,
                    "Atrial Amplitude": 3.5,
                    "Atrial Pulse Width": 0.4,
                    "ARP": 250
                },
                "VOOR": {
                    "Lower Rate Limit": 60,
                    "Upper Rate Limit": 120,
                    "Ventricular Amplitude": 3.5,
                    "Ventricular Pulse Width": 0.4
                },
                "VVIR": {
                    "Lower Rate Limit": 60,
                    "Upper Rate Limit": 120,
                    "Ventricular Amplitude": 3.5,
                    "Ventricular Pulse Width": 0.4
                }
            }
        }
    }
```

Welcome page of the app:


DCM Application - Welcome Page
Pacemaker Device Controller-Monitor
Username
Password
Login
Register

Error message for failed registration or login: (this example is of entering a short password)



Password

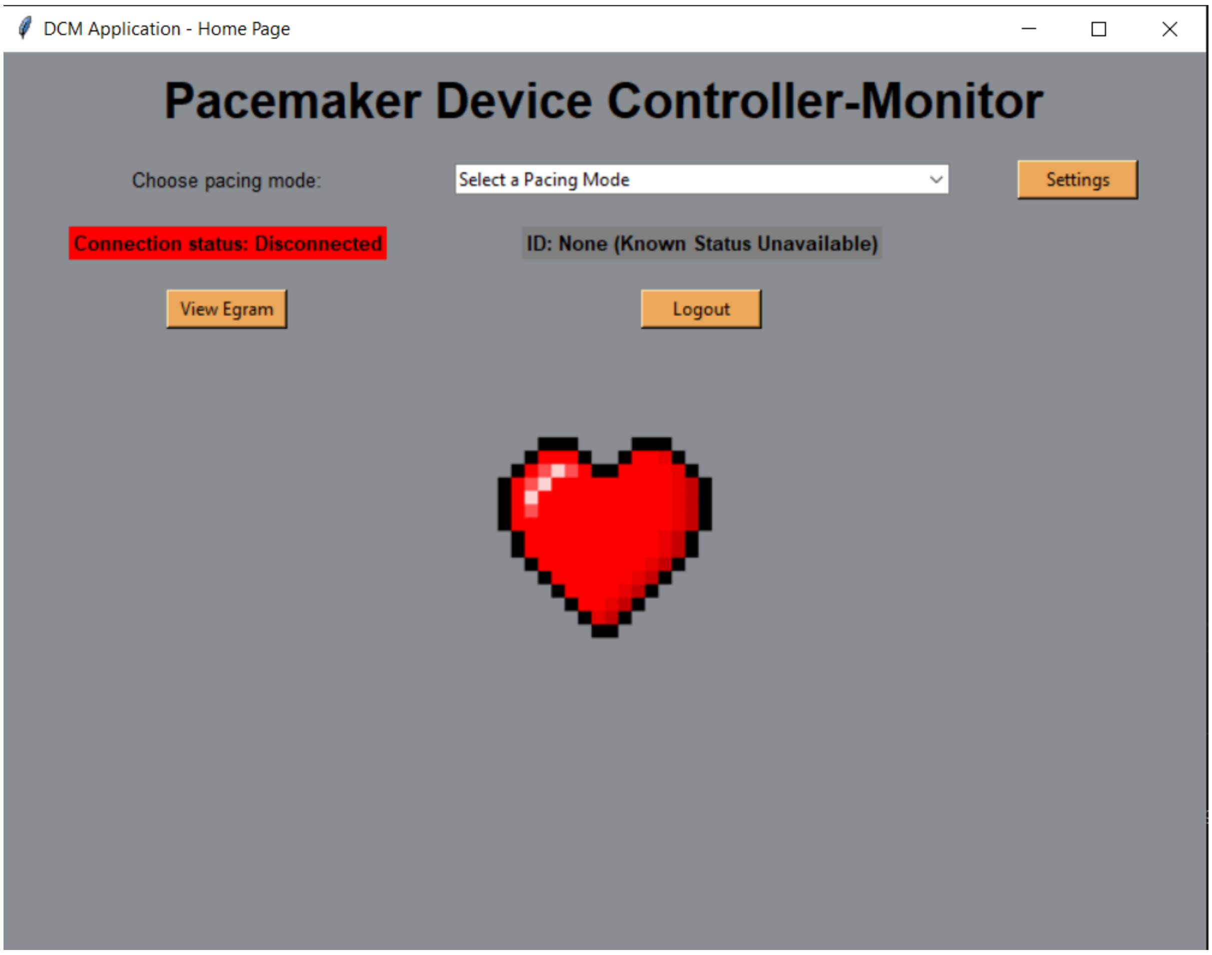
***

Login

Register

Password must be at least 8 characters

Home page of the app featuring connection status, previously interrogated pacemaker indicator, a way to log out, and a way to enter egram and settings pages:



DCM Application - Home Page

Pacemaker Device Controller-Monitor

Choose pacing mode: Select a Pacing Mode

Settings

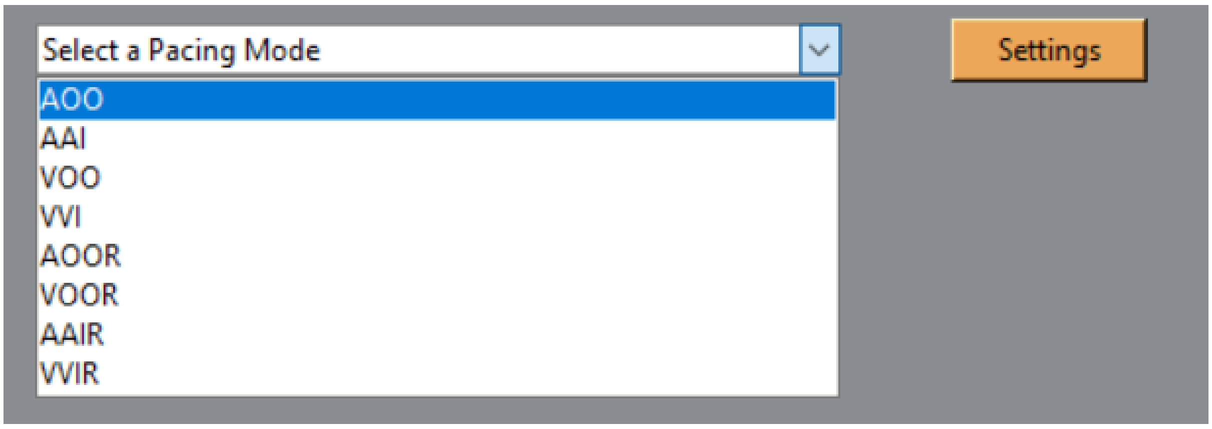
Connection status: Disconnected

ID: None (Known Status Unavailable)

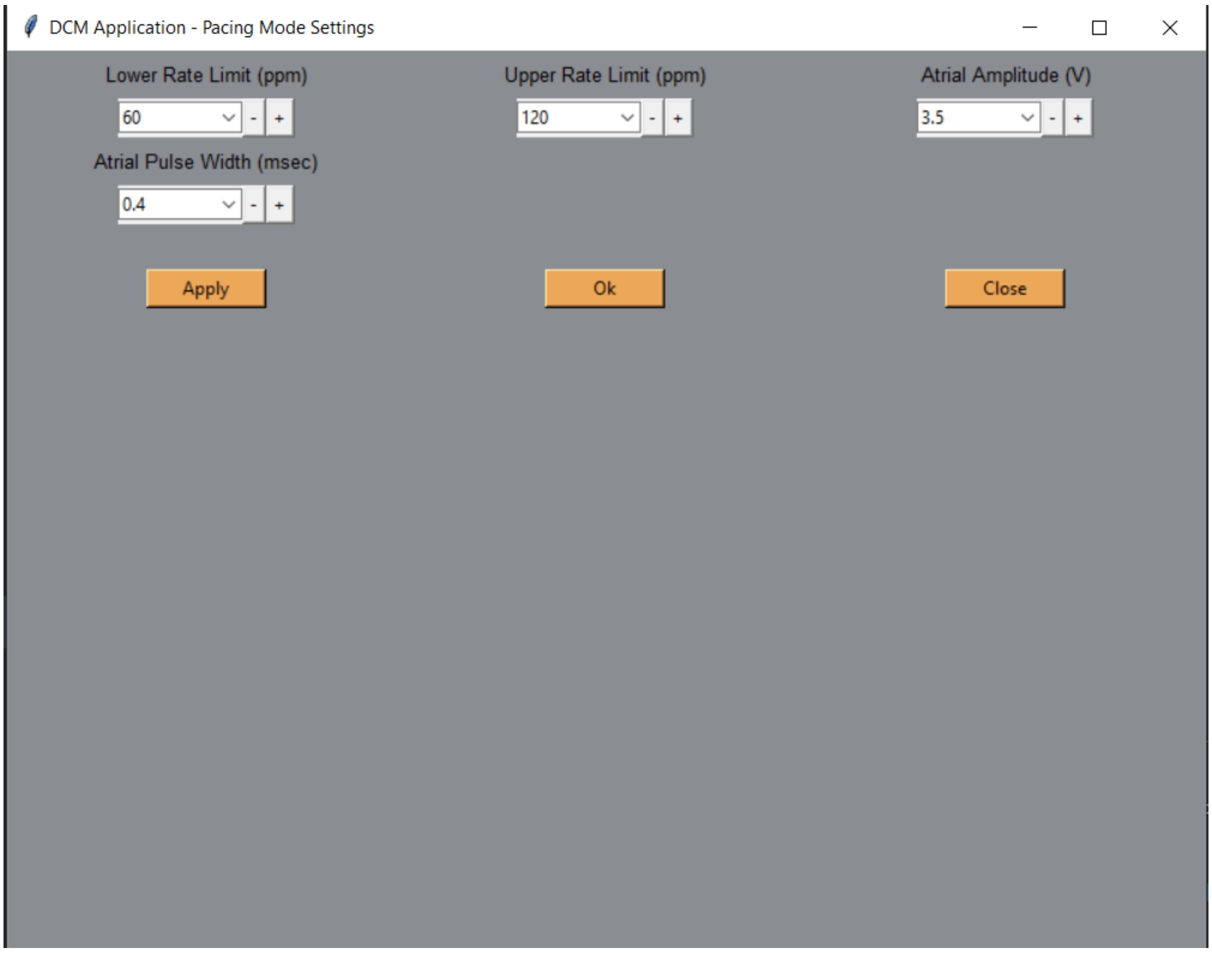
View Egram

Logout

Entering a pacing mode:

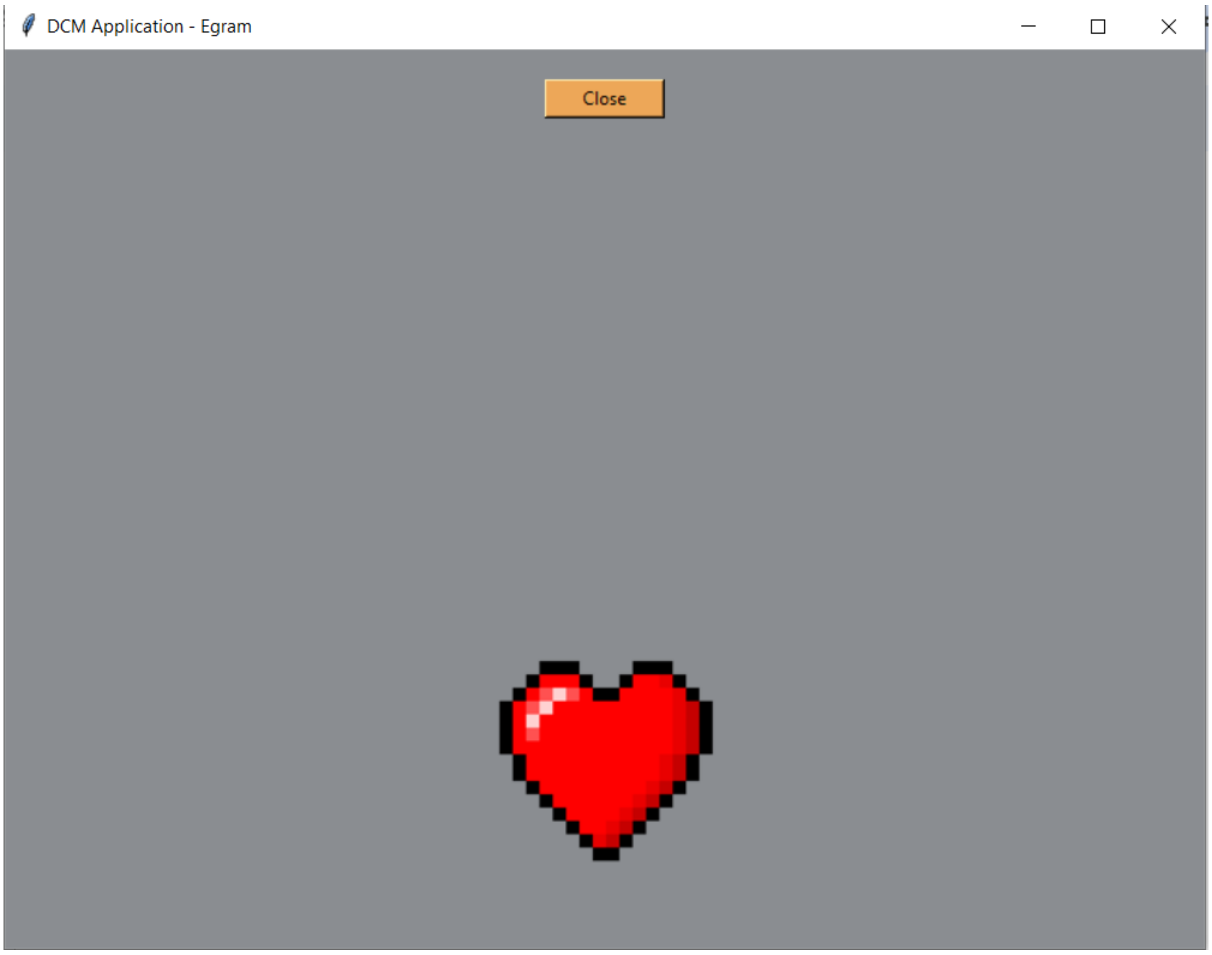


Settings screen for a pacing mode: (parameters vary depending on selected pacing mode)



Lower Rate Limit (ppm) 60
Upper Rate Limit (ppm) 120
Atrial Amplitude (V) 3.5
Atrial Pulse Width (msec) 0.4

Reserved screen for Egram:



Part 2

Likely Requirement Changes

The current backend class primarily functions as a template, making use of the pyserial library for serial communication. At present, it initialises essential parameters, including port and connection fields. It's worth noting that this configuration is expected to evolve as additional backend requirements emerge. One prospective adjustment involves transitioning to a more generalised backend class with child classes that inherit from it. This approach would allow for more nuanced and focused behaviours as needed.

Furthermore, the process of detecting other devices is slated for improvement. Instead of relying on a static list, we plan to enhance this by maintaining a dynamic list of devices, tracking their IDs, and detecting changes in IDs. This means that the application will consult an external file for the latest device information, ensuring it's up to date. Implementing a mechanism for devices to communicate their IDs with the application is also on the agenda.

Regarding egram data, the current structure is designed to store data at specific intervals when it's available from the port. We are considering a shift towards a more data-centric approach, where data will be processed in a more granular manner to ensure different data types are accurately recognized. This will pave the way for an update to the egram data tab, which currently remains blank as it awaits the influx of data from the backend. The class's flexible format makes it easy to adapt and implement these changes as needed.

Regarding the implementation of R-type pacing modes (AOOR, VVIR, etc), this could be easily implemented, because we currently have all the dicts setup, therefore the only changes we would need to implement are in the parameters.

Modules

Welcome Screen

The welcome module is the entry point into the pacemaker GUI. This module provides a login and registration option for users who want to access the pacemaker. If a new user needs to register, they can fill the username and password information, then click "Register" to be enlisted as one of the 10 accepted users. If an existing user is trying to login, they can input their username and password and click "login" instead. This module also has error handling protocols such as error prompts that appear if the username/passwords are invalid.

Homepage Screen

The homepage module allows users to choose their next steps with the interface. Users have the option to enter the settings page or view the status of their pacemaker. The connection status allows users to know when the pacemaker is connected to the GUI. A feature that lets the user know when a new pacemaker is within range will also be available in this module. Users are able to select the pacing mode they want to edit through a dropdown menu button, then click "settings" for editing capabilities. This module also has a "Logout" button which lets the user logout.

Settings Screen

Once the user has chosen the pacing mode they wish to adjust, they enter the settings module. Depending on the mode, the user will have the option to adjust the parameters and apply them to the pacemaker. The setting module also accounts for invalid inputs such as negative values and non-integer values.

Public Functions

Class Application:

- __init__ (self)
- run_app (self)
- handle_welcome_screen (self)
- handle_homepage_screen (self)
- handle_settings_screen (self)
- handle_egram_screen (self)

Class FunkyWidget:

- __init__(self)
- update_increment (self, selected_interval)
- update_display (self, event)
- get_increment_interval (self, current_value: float)

- get_next_increment_interval (self, current_interval:tuple)
- get_previoust_increment_interval (self, current_interval:tuple)
- increment_value (self)
- decrement_value (self)
- get (self)

Class Backend:

- __init__(self, port: str = None, device_id: str = None)
- is_connected (self)
- board_connected (self)
- Get_egram_dict (self)

Class Database:

- __init__(self, database: str = DATABASE, users_map: dict | None = None)
- load_users_from_json(self)
- get_user_count(self)
- write_to_file(self, user: User)
- register_user(self, welcome_page: tk.Tk, username_entry, password_entry)
- login_user (self, welcome_page: tk.Tk, username_entry, password_entry,)
- update_parameters(self, user: User, current_user: str, pacing_mode: str, data: dict)
- read_from_file(self)

Class PacingParameters:

- __init__(self, name: str, valid_interval_map: dict, unit: str = "")

Class Screen

- __init__(self, geometry: str, bg_colour: str = "#8a8d91")
- create_button(self, text: str, command: callable, width: int = 10, height: int = 1, bg_colour: str = "#eda758")
- create_entry(self, encrypted: bool = False)
- create_label(self, text: str, fontsize: int, bold: bool = False, font: str = "Helvetica", wraplength: int = 0)
- create_options(self, options: list, default_text: str = None)
- create_funky_widget(self, entry_map: dict, data: str)
- create_logo(self, image_path: str, dimensions: tuple)
- create_spacer(self, space: int)
- bring_to_front(self)
- load_grid(self, cols: bool, rows: bool, uniform: bool = False)
- prepare_screen_switch(self)
- run_screen(self)
- close_screen(self)

Class WelcomeScreen(Screen)

- __init__(self, geometry: str, bg_colour: str = "#8a8d91")
- run_screen(self)
- login_user(self)
- register_user(self)

Class HomepageScreen(Screen)

- __init__(self, geometry: str, current_user: User, bg_colour: str = "#8a8d91")
- run_screen(self)
- get_pacing_mode(self)
- logout(self)
- egram(self)
- check_connection(self)

Class SettingScreen(Screen)

- __init__(self, geometry: str, current_user: User, pacing_mode: str, bg_colour: str = "#8a8d91")
- run_screen(self)
- apply(self)
- ok(self)
- close(self

Class EgramScreen (Screen)

- __init__(self, geometry: str, bg_colour: str = "#8a8d91")
- run_screen(self)
- close(self)

Black Box Behaviour

Application.__init__(self): The __init__ method initialises the Application class. It sets attributes for the application's page geometry, current user, pacing mode, and the current screen being displayed. It also initialises a dictionary that maps screen names to their corresponding handling methods.

Application.run_app(self): The run_app method is responsible for running the application. It continually executes the current screen's handling method until the current screen becomes None, which indicates that the application should exit. The specific screen to display is determined by the handling methods for different screens.

Application.handle_welcome_screen(self): This method handles the behaviour for the "WelcomeScreen" state. It creates a WelcomeScreen instance with specific page geometry, runs the screen, and updates the page_geometry based on the screen's geometry. If a user logs in on the WelcomeScreen, it sets current_screen to "HomepageScreen" and assigns the logged-in user to current_user. Otherwise, it sets current_screen to None.

Application.handle_homepage_screen(self): This method manages the "HomepageScreen" state. It creates a HomepageScreen instance with the page geometry and the current user, runs the screen, and updates the page_geometry. Depending on user actions, it transitions to different screens: "WelcomeScreen" on logout, "SettingsScreen" for settings, "EgramScreen" for e-gram viewing, or None if no specific action occurs.

Application.handle_settings_screen(self): Handling the "SettingsScreen" state, this method creates a SettingsScreen instance with page geometry, the current user, and the pacing mode. It runs the screen, updates the page_geometry, and sets current_screen to "HomepageScreen" on closing the settings screen or None otherwise.

Application.handle_egram_screen(self): For the "EgramScreen" state, this method creates an EgramScreen instance with the page geometry, runs the screen, updates the page_geometry, and sets current_screen to "HomepageScreen" on closing the e-gram screen or None otherwise.

Backend.__init__(self, port: str = None, device_id: str = None):
The __init__ method initialises the Backend class, taking two optional parameters: port for the serial port and device_id for the device ID. It also initialises an empty list previous_device_ids. Depending on the provided port, it establishes a serial connection with specific settings. If a port is not provided, an empty connection is created.
Backend.is_connected (property): The is_connected property checks if the serial port is open and returns True if it is open and False if it's closed. It determines the status of the serial connection.

Backend.board_connected (property): The board_connected property checks what board is currently connected to the system by examining the Vendor ID (VID) and Product ID (PID) of connected devices. It initialises VID and PID as placeholders and checks if any device in the list of connected devices matches the given VID and PID. If a match is found, it assigns the device's name to the device_id. If no match is found, device_id is set to None.

Backend.get_egram_dict: This function is a prototype from the backend to utilise the serial ports in a way that it can read the data in a templated format that can be modified later on. Currently it checks for voltages at certain times to simulate what Heartview does. The data is received in a dictionary format for simple parsing. This way when data is read and as long as data is received, the backend will construct this dict and then different voltages at certain timestamps can be received for plotting. It can also just grab voltage levels and not worry about time stamps so that it can plot this data during live connection. For this, it is possible that the format can be changed to a dictionary.

Database.__init__(self, database: str = DATABASE, users_map: dict | None = None):
Initialises the class with a database file path (\`database\`) and optionally loads user data from the specified database file.

Database.get_user_count\` (property):
Returns the number of users in the database.

Database.write_to_file(self, user: User)`:
Writes user registration data to the database, adding the user and updating the database file.

Database.register_user(self, welcome_page: tk.Tk, username_entry, password_entry) -> bool`:
Registers a user based on provided username and password, performing input validation and displaying success or failure messages.

Database.login_user(self, welcome_page: tk.Tk, username_entry, password_entry) -> bool`:
Logs a user into the application by verifying the username and password, returning `True` on successful login.

Database.update_parameters(self, user: User, current_user: str, pacing_mode: str, data: dict)`:
Updates user's pacing mode parameters in the database and within the `User` object.

Database.read_from_file(self) -> list`:
Reads user data from the database file and returns it as a list.

FunkyWidget.__init__(self, screen: tk.Tk, limits: dict, default: float, **kwargs): Initialises the widget with a given screen, limits, and a default value. Creates a user interface component with adjustable values within specified limits.

FunkyWidget.get_increment_interval(self, current_value: float) -> tuple: Determines the increment and interval for the current value. Maps the widget's value to the appropriate interval and increment settings.

FunkyWidget.get_next_increment_interval(self, current_interval: tuple) -> tuple: Determines the increment and interval for the next value. Helps facilitate transitions between different interval settings.

FunkyWidget.get_previous_increment_interval(self, current_interval: tuple) -> tuple: Determines the increment and interval for the previous value. Supports transitions between different interval settings in the reverse direction.

FunkyWidget.get(self): Retrieves and returns the current value of the widget. Provides access to the widget's current value for external use.

PacingParameters.__init__(self, name: str, valid_interval_map: dict, unit: str = ""): Initialises an instance of the PacingParameters class with a name, valid interval map, and optional unit. Allows for the creation of pacing parameters with specific attributes.

Screen Class:

 __init__(self, geometry: str, bg_colour: str = "#8a8d91)"`: Initialises a screen object with geometry and background colour. Prepares an empty screen layout.

create_button(self, text: str, command: callable, width: int = 10, height: int = 1, bg_colour: str = "#eda758")`: Creates a button widget with specified text, command, width, height, and background colour. Returns the created button widget.

create_entry(self, encrypted: bool = False)`: Creates an entry widget for text input, with optional encryption. Returns the created entry widget.

create_label(self, text: str, fontsize: int, bold: bool = False, font: str = "Helvetica", wraplength: int = 0)`: Creates a label widget with specified text, fontsize, and formatting options. Returns the created label widget.

create_options(self, options: list, default_text: str = None)`: Creates an option menu widget for selecting from a list of options. Returns the created option menu widget and its associated string variable.

create_funky_widget(self, entry_map: dict, data: str)`: Creates a custom "FunkyWidget" and returns it.

create_logo(self, image_path: str, dimensions: tuple)`: Creates a logo widget using an image file and specified dimensions. Returns the created logo widget.

create_spacer(self, space: int)`: Creates a spacer widget to add space in the layout. Returns the created spacer widget.

bring_to_front(self)`: Brings the current screen to the front.

load_grid(self, cols: bool, rows: bool, uniform: bool = False)`: Configures the grid layout by setting column and row weights.

prepare_screen_switch(self)`: Prepares the screen for switching.

run_screen(self)`: Creates and runs the screen, setting its geometry and background color.

close_screen(self)`: Closes the current screen.

WelcomeScreen Class:

__init__(self, geometry: str, bg_colour: str = "#8a8d91")`: Initialises the welcome screen with geometry and background colour. Manages user login and registration.

login_user(self) -> User`: Handles the login process and sets the current user. Returns the logged-in user.

register_user(self)`: Handles the registration process.

HomepageScreen Class:

__init__(self, geometry: str, current_user: User, bg_colour: str = "#8a8d91")`: Initialises the homepage screen with geometry, a current user, and background colour. Manages pacing mode selection and user logout.

get_pacing_mode(self)`: Retrieves the selected pacing mode and navigates to the settings screen.

logout(self)`: Logs the user out.

check_connection(self)`: Checks and displays the connection status.

SettingsScreen Class:

__init__(self, geometry: str, current_user: User, pacing_mode: str, bg_colour: str = "#8a8d91")`: Initialises the settings screen with geometry, a current user, pacing mode, and background colour. Manages the user's pacing mode settings.

apply(self)`: Applies the user's input settings.

ok(self)`: Applies the settings and closes the screen.

close(self)`: Closes the settings screen.

EgramScreen Class:

__init__(self, geometry: str, bg_colour: str = "#8a8d91")`: Initialises the Egram screen with geometry and background colour.

close(self)`: Closes the Egram screen.

User Class:
**__init__(self, username: str, password: str, data: dict = None)`:** Initialises a `User` object with a username, password, and pacing mode parameters. The `data` parameter is optional and used to set pacing mode parameters. If not provided, default parameters are used.

**to_dict(self) -> dict`:** Returns a dictionary representation of the `User` object for writing to a JSON file. The dictionary includes the username, password, and pacing mode parameters of the user.

**update_parameters(self, data: dict, pacing_mode: str = None)`:** Updates pacing mode parameters of the user. If `pacing_mode` is specified, it updates the parameters for that specific pacing mode. If `pacing_mode` is not specified, it updates all the pacing mode parameters.

<u>Global Variables</u>

The GUI runs on an Object-Oriented model which reduces the need for global variables. The program currently uses no global variables.

The Pacemaker GUI uses maps and dictionaries to store, organise and retrieve data. There are many instances of their use. The User class leverages dictionaries to store user-specific pacing mode parameters. Each user's parameters are represented as a dictionary, where pacing mode names, such as "AOO" or "AAI," act as keys. The associated values are another dictionary, encapsulating details like "Lower Rate Limit," "Upper Rate Limit," and more. This hierarchical structure efficiently groups and manages user-specific parameters, allowing easy access and modification.

In the context of the SettingsScreen class, a map is employed to map pacing mode names to lists of pacing parameters. This mapping guides which parameters are displayed based on the chosen pacing mode. The map simplifies the process of dynamically rendering the appropriate parameters on the settings screen. By structuring the parameters in this way, the code becomes more modular and adaptable to different pacing modes.

The PacingParameters class showcases the use of dictionaries as well. Specifically, the class takes a dictionary named valid_interval_map as an attribute. This dictionary is instrumental in mapping interval names to their respective values, which define valid intervals for the given pacing parameter. Such mapping simplifies the validation of user inputs for this parameter.

<u>Private Functions</u>

- __update_increment(self, selected_interval)
- __update_display(self, event)
- __increment_value(self)
- __decrement_value(self)

- __load_users_from_json(self)
- __add_user(self, user: User)
- __remove_user(self, username: str)
- __write_to_file(self, user: User)

- def __repr__(self)

Internal Behaviour

User Class

When a new `User` object is created, the constructor __init__ receives a `username`, a `password`, and an optional `data` dictionary that contains pacing mode parameters. The `username` and `password` are immediately assigned to the object's attributes, `self.username` and `self.password`. If `data` is provided during initialization, it's set as the `parameter_dict` attribute, which stores the user's pacing mode parameters. If `data` is not provided, the `parameter_dict` is initialised with default pacing parameters for different pacing modes. This function establishes and maintains the state variables `username`, `password`, and `parameter_dict`.

The private method `__repr__` is responsible for providing a human-readable string representation of the `User` object when it's printed. It returns a string that includes the `username` and `password` of the user. Importantly, this method does not alter any state variables; its purpose is solely to provide information.

The public method `to_dict' has a specific purpose, which is to convert the user's information, including the `username`, `password`, and pacing mode parameters, into a dictionary format. It returns a dictionary where the `username` serves as the key, and within this key, there's a nested dictionary containing `"password"` and `"pacing_mode_params"` keys. The state variables `username`, `password`, and `parameter_dict` are not modified by this method.

The public method `update_parameters` is designed for updating the user's pacing mode parameters. It takes in a `data` dictionary containing the new pacing mode parameters and an optional `pacing_mode` as an argument. If `pacing_mode` is specified, the method updates the `parameter_dict` for that specific pacing mode using the provided data. If `pacing_mode` is not specified, the method replaces the entire `parameter_dict` with the provided data. This function efficiently updates the state variable `parameter_dict`.

As for the constructor `__init__`, it closely mirrors the public `__init__` method. When creating a `User` object, this method initialises it with a `username`, a `password`, and an optional `data` dictionary. The `username` and `password` values are assigned to the object's attributes, maintaining the state variables. If `data` is provided, it sets the `parameter_dict` attribute with this data. If `data` is not provided, it initialises the `parameter_dict` with the default pacing parameters for various pacing modes, preserving the state variables `username`, `password`, and `parameter_dict`.

Screen Class:

__init__(self, geometry: str, bg_colour: str = "#8a8d91"): Initialises the `Screen` class with the provided geometry and background colour. It sets the `bg_colour` and `geometry` attributes, which maintain state information about the screen's appearance.

create_button(self, text: str, command: callable, width: int = 10, height: int = 1, bg_colour: str = "#eda758"): Creates a button widget with the given properties and appends it to the `widgets` dictionary under the "Button" key. The button's command is specified as an argument, allowing it to be executed when the button is clicked. This function doesn't directly alter state variables but helps to populate the `widgets` dictionary.

create_entry(self, encrypted: bool = False): Generates an entry widget and adds it to the `widgets` dictionary under the "Entry" key. If the `encrypted` parameter is `True`, the entry is configured to display a password. This function maintains the `widgets` state variable.

create_label(self, text: str, fontsize: int, bold: bool = False, font: str = "Helvetica", wraplength: int = 0)`**: Creates a label widget with the provided properties and includes it in the `widgets` dictionary under the "Label" key. The function helps to maintain the state of the `widgets` dictionary.

create_options(self, options: list, default_text: str = None)`**: Creates an option menu widget and returns it. The selected option is associated with a `StringVar` that's also added to the `widgets` dictionary. This function maintains the `widgets` state variable.

create_funky_widget(self, entry_map: dict, data: str)`**: Produces a custom `FunkyWidget` and adds it to the `widgets` dictionary under the "FunkyWidget" key. It helps maintain the state variables in the `widgets` dictionary.

create_logo(self, image_path: str, dimensions: tuple)`**: Constructs a label widget for displaying a logo image and appends it to the `widgets` dictionary under the "Label" key. This function helps maintain the `widgets` state variable.

create_spacer(self, space: int)`**: Generates a spacer widget (an empty frame) and returns it. This function doesn't directly alter state variables but can be used for layout and spacing purposes.

bring_to_front(self): Brings the screen to the front, making it the active window. This function doesn't directly modify state variables but affects the screen's behaviour.

load_grid(self, cols: bool, rows: bool, uniform: bool = False)`**: Configures column and row weights for the screen's grid, which affects the layout. It doesn't alter state variables directly but impacts the screen's appearance and responsiveness.

prepare_screen_switch(self): Prepares for a screen switch by storing the current geometry and triggering the `close_screen` function to close the screen. It doesn't directly alter state variables but sets the stage for transitioning between screens.

run_screen(self): Creates the screen with the specified geometry and background colour. It sets the `page_height` and `page_width` attributes based on the screen's dimensions. This function doesn't directly change state variables but sets up the initial state of the screen.

close_screen(self): Closes the screen by destroying it. This function doesn't directly affect state variables, but it's essential for removing the screen from view.

WelcomeScreen Class:

__init__(self, geometry: str, bg_colour: str = "#8a8d91")`: Initialises the `WelcomeScreen` class with the specified geometry and background colour, and additional attributes like the title, database connection, login status, and the currently logged-in user. These attributes help maintain the state of the welcome screen.

run_screen(self): Runs the welcome screen with a login interface. It checks the user's login credentials, and upon successful login, it updates the state variables `logged_in` and `logged_user`.

login_user(self): This function acts as a wrapper for the `Database.login_user` method. It handles user login, updates the state variable `logged_user`, and ensures the state variable `logged_in` is set appropriately.

register_user(self): Acts as a wrapper for the `Database.register_user` method, allowing users to register.

HomepageScreen Class:

`__init__(self, geometry: str, current_user, bg_colour: str = "#8a8d91")`: Initialises the `HomepageScreen` with attributes for the current user, pacing mode, and other UI elements like buttons. These attributes help maintain the state of the homepage.

run_screen(self): Runs the homepage screen, allowing users to select pacing modes and manage their interaction with the application. It checks the connection status and maintains the state of the application's UI.

get_pacing_mode(self): Retrieves the selected pacing mode from the dropdown menu and

PacingParameters Class:

__init__(self, name: str, valid_interval_map: dict, unit: str = "")`: The constructor of the `PacingParameters` class initialises the class with a name, a valid interval map, and a unit. This class is used to represent a pacing parameter. It sets the `name`, `valid_interval_map`, and `unit` attributes for each instance of the class, which help maintain state information for individual pacing parameters.

Parameters Enum:

LOWER_RATE_LIMIT = PacingParameters("Lower Rate Limit", L_RATE_INT, "ppm")`: This line defines an enum constant `LOWER_RATE_LIMIT` with an associated `PacingParameters` instance. It specifies the name, valid interval map, and unit for the "Lower Rate Limit" parameter. The enum `Parameters` collects various pacing parameters, and this specific line adds an enum constant for the lower rate limit. The state information is stored in the enum constant, including the associated `PacingParameters` instance.

UPPER_RATE_LIMIT = PacingParameters("Upper Rate Limit", U_RATE_INT, "ppm")`: Similar to the previous line, this defines an enum constant for the "Upper Rate Limit" parameter. It associates the parameter with a specific name, valid interval map, and unit. State information is stored within this enum constant.

ATRIAL_AMPLITUDE = PacingParameters("Atrial Amplitude", AMPLITUDE_INT, "V")`: This line is similar to the previous ones but defines an enum constant for "Atrial Amplitude."

ATRIAL_PULSE_WIDTH = PacingParameters("Atrial Pulse Width", PULSE_WIDTH_INT, "msec")`: Defines an enum constant for the "Atrial Pulse Width" parameter.

ARP = PacingParameters("ARP", ARP_INT, "msec"): Creates an enum constant for the "ARP " parameter.

VENTRICULAR_AMPLITUDE = PacingParameters("Ventricular Amplitude", AMPLITUDE_INT, "V"): This line defines an enum constant for "Ventricular Amplitude."

VENTRICULAR_PULSE_WIDTH = PacingParameters("Ventricular Pulse Width", PULSE_WIDTH_INT, "msec"): Defines an enum constant for "Ventricular Pulse Width."

VRP = PacingParameters("VRP", VRP_INT, "msec"): Similar to the previous lines, this defines an enum constant for "VRP."

Database Class

`__init__(self, database: str = DATABASE, users_map: dict | None = None): Initialises the `Database` class with a database file path. It loads the users from the JSON file using the `__load_users_from_json()` method and stores them in the `users_map` attribute.

__load_users_from_json(self) -> dict`: Loads users from a JSON file and returns a dictionary of users and passwords. It reads the JSON data from the database file, processes it to extract usernames and passwords, and returns a dictionary of users.

get_user_count(self) -> int: A property method that returns the number of users in the database. It accesses the `users_map` attribute and returns its length.

__add_user(self, user: User): A private method for adding a user to the `users_map`. It takes a `User` object and adds the user's username and password to the `users_map`.

`__remove_user(self, username: str): A private method for removing a user from the `users_map`. It takes a username as input and deletes the corresponding user from the `users_map`.

__write_to_file(self, user: User): A private method that writes user data to the database when registered. It adds a user to the `users_map`, updates the JSON data with the new user's information, and writes it back to the database file.

register_user(self, welcome_page: tk.Tk, username_entry, password_entry) -> bool`**:
Registers a user and writes their information to the database if the input is valid. It checks the validity of the username and password, handles various error cases, and, if successful, creates a new `User` object and calls `__write_to_file()` to save the user's data.

login_user(self, welcome_page: tk.Tk, username_entry, password_entry) -> bool`**:
 - Logs a user into the homepage if the user exists and the password is correct.
 - It checks the validity of the provided username and password, verifies the login information, and returns `True` if the login is successful.

update_parameters(self, user: User, current_user: str, pacing_mode: str, data: dict): Updates the pacing mode parameters of a user and writes them to the JSON database. It takes a `User` object, the current username, pacing mode, and data to be updated, and it modifies the user's pacing mode parameters in the database.

read_from_file(self) -> list: Reads users from the database file and returns a list of users. It reads the JSON data from the database file and returns the list of users.

FunkyWidget Class

__init__(self, screen, limits, default, **kwargs)`:
  Initialises the widget with the provided screen, limits, and default value. Create a dropdown menu with values corresponding to the defined intervals. Set the initial value of the widget to the default value. Define `current_crement` and `current_interval` based on the default value.

__update_increment(self, selected_interval)`: Update the increment value (`current_crement`) when the user selects a different interval. Determine the current interval based on the selected interval from the dropdown menu.

get_increment_interval(self, current_value)`: Calculate the increment and interval for the current value. Loop through the defined intervals and find the one that contains the current value. If the value is within the interval, determine the increment for that interval. Return the increment and interval for the current value.

get_next_increment_interval(self, current_interval)`: Get the increment and interval for the value that follows the current interval. Determine the index of the current interval in the list. Calculate the next interval's index, considering the possibility of looping back to the beginning of the list. Return the increment and interval for the next value.

get_previous_increment_interval(self, current_interval)`: Get the increment and interval for the value that precedes the current interval. Determine the index of the current interval in the list. Calculate the previous interval's index, considering the possibility of looping back to the end of the list. Return the increment and interval for the previous value.

__increment_value(self): Handle value increment when the "+" button is clicked. Check if the current value is at the upper bound of the current interval. If it is, move to the next interval and adjust the value accordingly. Otherwise, increment the value within the current interval.

__decrement_value(self): Handle value decrement when the "-" button is clicked. Check if the current value is at the lower bound of the current interval. If it is, move to the previous interval and adjust the value accordingly. Otherwise, decrement the value within the current interval.

get(self)`: Retrieve the currently selected value within the widget. Convert the value from the dropdown menu (as a string) to a numeric value and return it.

Backend Class

__init__(self, port: str = None, device_id: str = None): This constructor initialises a `Backend` object with optional parameters `port` and `device_id`. It sets the instance variable `device_id` to the provided `device_id` and creates an empty list named `previous_device_ids`. Within the "TODO" section, it attempts to read device IDs from a file and appends them to the `previous_device_ids` list. It then checks if the `port` is provided. If `port` is `None`, it creates an empty serial connection (`ser`) to be used for future communication. If `port` is provided, it establishes a serial connection using the specified port and sets the connection parameters (baud rate and timeout). Finally, it flushes the input and output buffers of the serial connection.

is_connected(self) -> bool: This property method checks the status of the serial port connection by accessing the `is_open` attribute of the `ser` serial connection. If the serial port is open, indicating an active connection, it returns `True`. If the port is not open, it returns `False`, indicating that there is no active connection. This function does not modify any state variables but provides information about the connection status.

board_connected(self) -> str: This property method determines which board is connected to the system by specifying the Vendor ID (`VID`) and Product ID (`PID`) of the target board. It retrieves a list of connected devices using the `list_ports.comports()` function. The method then iterates through the list of devices, checking if any device's VID and PID match the specified values. If a matching device is found, it sets the `device_id` instance variable to the identifier of the device, such as a COM port on Windows. If no matching device is found, it sets `device_id` to `None`. The function does not modify the `previous_device_ids` list in this version. A "TODO" section suggests checking if the connected board's device ID is present in the `previous_device_ids` list, ensuring it's not a previously interrogated board.

Application Class

__init__(self): The constructor initialises an `Application` object. It sets the initial state of the application, including `page_geometry`, `current_user`, `pacing_mode`, and `current_screen`. The `page_geometry` stores the initial window size, while `current_user` and `pacing_mode` are initialised as `None`. The `current_screen` is set to the "WelcomeScreen," indicating the initial state. Additionally, a dictionary named `screens` is created, where keys represent screen names, and values are functions responsible for handling each screen state.

run_app(self): This method is responsible for running the application. It enters a loop that continues as long as `current_screen` is not `None`. Within the loop, it retrieves the handler function associated with the current screen from the `screens` dictionary. It then calls this handler function to manage the state of the application.

handle_welcome_screen(self): This method handles the behaviour when the application is in the "WelcomeScreen" state. It creates an instance of the `WelcomeScreen` class, passing the current `page_geometry` as an argument. It runs the `WelcomeScreen` using the `run_screen()` method. After the screen is closed, it updates the `page_geometry` based on the new screen geometry and checks if a user has logged in. If a user has logged in, it transitions to the "HomepageScreen" state, sets the current user, and updates the `current_screen` accordingly. If no user logs in, it sets `current_screen` to `None`.

handle_homepage_screen(self): This method handles the "HomepageScreen" state. It creates an instance of the `HomepageScreen` class, passing the current `page_geometry` and `current_user`. It runs the `HomepageScreen` using the `run_screen()` method. After the screen is closed, it updates the `page_geometry`, checks if the user has logged out, accessed settings, or requested to view the Egram. Depending on these conditions, it transitions to the "WelcomeScreen," "SettingsScreen," or "EgramScreen" state, and updates the `current_screen` and `pacing_mode` as needed.

handle_settings_screen(self): This method handles the "SettingsScreen" state. It creates an instance of the `SettingsScreen` class, passing the current `page_geometry`, `current_user`, and `pacing_mode`. It runs the `SettingsScreen` using the `run_screen()` method. After the screen is closed, it updates the `page_geometry`. If the screen was closed (settings were updated), it transitions back to the "HomepageScreen." If it is not closed, it sets `current_screen` to `None`.

handle_egram_screen(self): This method handles the "EgramScreen" state. It creates an instance of the `EgramScreen` class, passing the current `page_geometry`. It runs the `EgramScreen` using the `run_screen()` method. After the screen is closed, it updates the `page_geometry`. If the screen was closed, it transitions back to the "HomepageScreen." If it was not closed, it sets `current_screen` to `None.

Testing

Testing procedures involved a comprehensive evaluation of the application's functionality. Initially, we conducted manual tests by interacting with the application as different users, both new and existing. Varied parameters were configured to assess the application's performance. These manual assessments were pivotal in identifying any potential issues or bugs within the application.

Subsequently, a dedicated test database was established to automate the testing process. A script was developed to facilitate the clearing and updating of this database when running tests. We leveraged the pytest framework to craft a suite of tests that rigorously checked various aspects of the application. This encompassed extensive coverage of edge cases, ensuring that all functions were thoroughly validated.

Manual testing played an essential role in the quality assurance process, serving as a critical checkpoint before automation. This allowed us to confirm the absence of any inherent bugs within the application itself.

Tests were meticulously written to address different scenarios for user logins and registrations, examining various instantiations and behaviours. Automation was further enhanced with the creation of a script capable of setting up the database and seamlessly transitioning to the testing environment. Additionally, a YAML file was authored to outline the testing procedure for GitHub. This framework enabled continuous integration, offering real-time validation of all code contributions.

A crucial aspect of our testing philosophy was ensuring extensive test coverage. While GitHub Actions couldn't execute Tcl and load GUIs, it was adept at testing other facets of the code. It verified the logic within the codebase that was directly testable without Tkinter, notably the private methods of FunkyWidget. The automated tests were conducted within an Ubuntu environment that mirrored the setup used by our contributors.

The primary focus during testing involved an in-depth examination of widget behaviours. We employed debugger tools and strategically placed print statements to facilitate the debugging process. This approach allowed us to verify that various features operated as intended and delivered the expected results. Code reviews played a pivotal role in the process, as pull requests were subject to thorough scrutiny and provided valuable second opinions.

Linting was also used to ensure the code was functional, compliant to standards, and was consistent and not redundant.

Results

During the development phase, a significant number of tests failed, primarily due to underlying issues related to the database structures and parsing mechanisms. This presented an invaluable opportunity to revamp the database format to ensure compatibility with various parameter values based on different pacing modes. The testing process shed light on these issues and drove essential improvements.

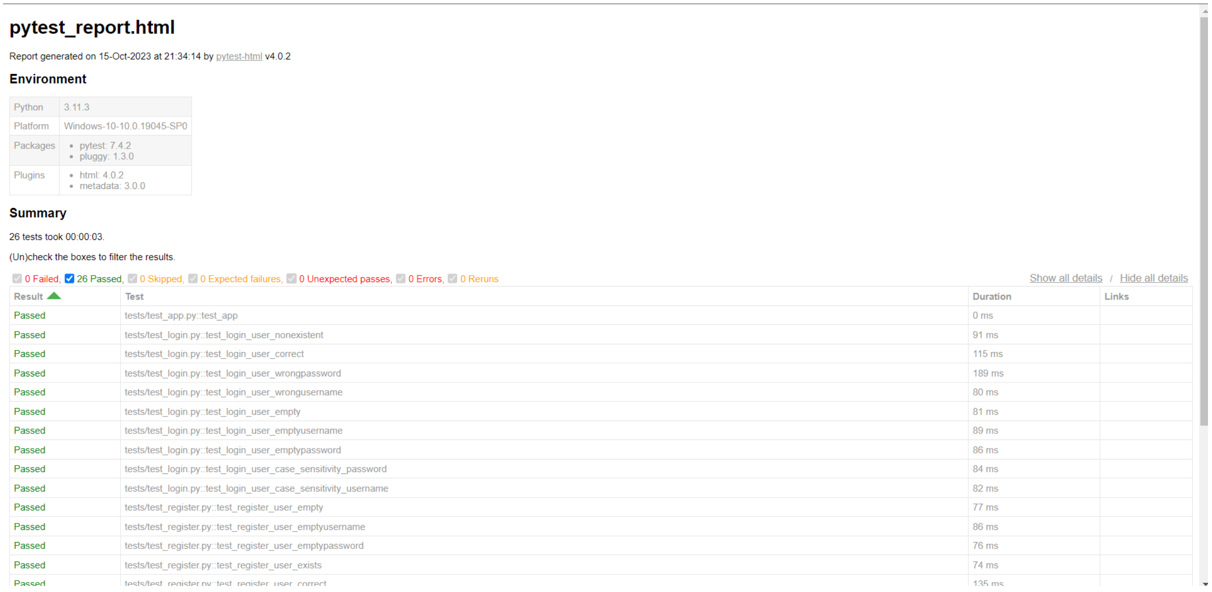
In the absence of GUI support, we had to resort to running testing scripts locally, as CI/CD pipelines could not execute GUI-related test cases, resulting in errors (though not in assertions). This workflow allowed us to scrutinise each push, verifying that it adhered to linting standards and then manually confirming that the functionality remained intact.

The testing approach involved substantial mocking of entries to ascertain whether the anticipated results were achieved. This process unveiled additional errors related to widget functionality, compelling us to conduct manual testing to establish a structured format for returning intervals and their increments. This experience guided us to leverage tools like OrderedDicts, facilitating the meaningful storage of data using the Parameters and Pacing Parameters classes.

One of the critical takeaways from this testing phase was the adaptability and extensibility of the Parameters class, which can readily accommodate other parameter types for potential future modifications. The utilisation of Enums to maintain a consistent format for storing parameters allowed us to ensure future scalability.

In essence, our testing methodologies proved to be highly effective in uncovering and rectifying bugs, enabling us to enhance the robustness and reliability of our application. The testing process not only addressed existing issues but also contributed to the evolution of our application, making it better equipped to handle future requirements.

The results were further observed through a report test interface as seen below:



pytest_report.html

Report generated on 15-Oct-2023 at 21:34:14 by pytest-html v4.0.2

Environment

Python	3.11.3
Platform	Windows-10-10.0.19045-SP0
Packages	• pytest: 7.4.2 • pluggy: 1.3.0
Plugins	• html: 4.0.2 • metadata: 3.0.0

Summary

26 tests took 00:00:03.

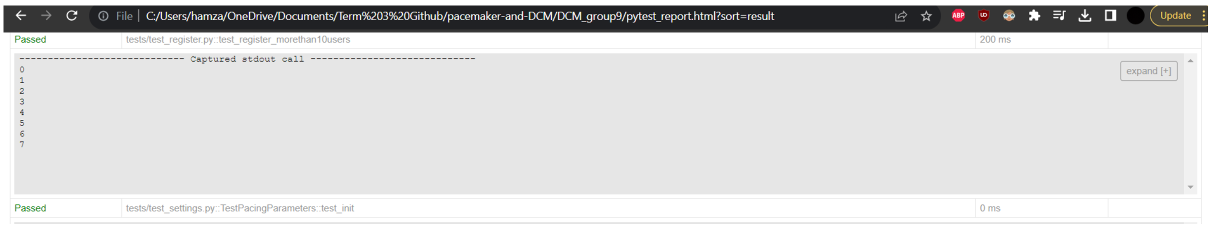
(Un)check the boxes to filter the results.

☑ 0 Failed, ☑ 26 Passed, ☑ 0 Skipped, ☑ 0 Expected failures, ☑ 0 Unexpected passes, ☑ 0 Errors, ☑ 0 Reruns

Show all details / Hide all details

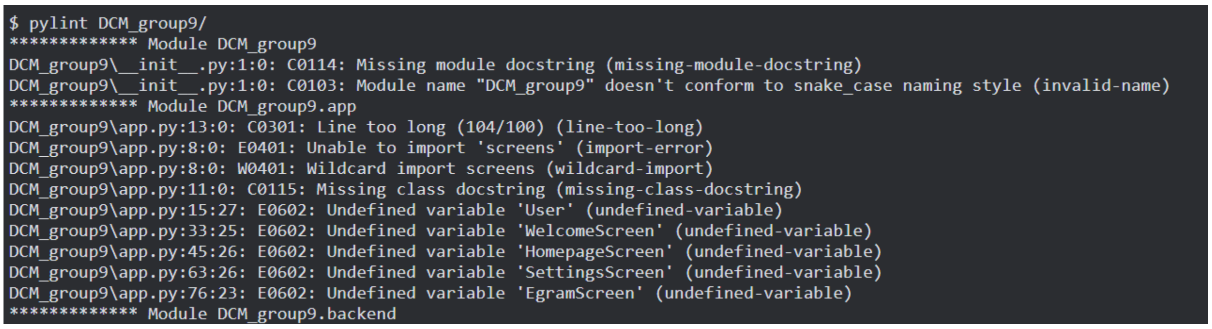
Result ▲	Test	Duration	Links
Passed	tests/test_app.py::test_app	0 ms	
Passed	tests/test_login.py::test_login_user_nonexistent	91 ms	
Passed	tests/test_login.py::test_login_user_correct	115 ms	
Passed	tests/test_login.py::test_login_user_wrongpassword	189 ms	
Passed	tests/test_login.py::test_login_user_wrongusername	80 ms	
Passed	tests/test_login.py::test_login_user_empty	81 ms	
Passed	tests/test_login.py::test_login_user_emptyusername	89 ms	
Passed	tests/test_login.py::test_login_user_emptypassword	86 ms	
Passed	tests/test_login.py::test_login_user_case_sensitivity_password	84 ms	
Passed	tests/test_login.py::test_login_user_case_sensitivity_username	82 ms	
Passed	tests/test_register.py::test_register_user_empty	77 ms	
Passed	tests/test_register.py::test_register_user_emptyusername	86 ms	
Passed	tests/test_register.py::test_register_user_emptypassword	76 ms	
Passed	tests/test_register.py::test_register_user_exists	74 ms	
Passed	tests/test_register.py::test_register_user_correct	135 ms	

This was made using HTML and was an interactive way for us to see what failed and how long different tests took with explicit diagnostic information for our tests.
This also let us capture stdout and observe what occurred in detail for each test in an easier format than reading from console and guessing where it came from:



```
C:/Users/hamza/OneDrive/Documents/Term%203%20Github/pacemaker-and-DCM/DCM_group9/pytest_report.html?sort=result
Passed    tests/test_register.py::test_register_morethan10users    200 ms
---------------------------- Captured stdout call ----------------------------
0
1
2
3
4
5
6
7
Passed    tests/test_settings.py::TestPacingParameters::test_init    0 ms
```

As for linting results, we noticed many issues with the code and were able to go back and update these things:



```
$ pylint DCM_group9/
************* Module DCM_group9
DCM_group9\__init__.py:1:0: C0114: Missing module docstring (missing-module-docstring)
DCM_group9\__init__.py:1:0: C0103: Module name "DCM_group9" doesn't conform to snake_case naming style (invalid-name)
************* Module DCM_group9.app
DCM_group9\app.py:13:0: C0301: Line too long (104/100) (line-too-long)
DCM_group9\app.py:8:0: E0401: Unable to import 'screens' (import-error)
DCM_group9\app.py:8:0: W0401: Wildcard import screens (wildcard-import)
DCM_group9\app.py:11:0: C0115: Missing class docstring (missing-class-docstring)
DCM_group9\app.py:15:27: E0602: Undefined variable 'User' (undefined-variable)
DCM_group9\app.py:33:25: E0602: Undefined variable 'WelcomeScreen' (undefined-variable)
DCM_group9\app.py:45:26: E0602: Undefined variable 'HomepageScreen' (undefined-variable)
DCM_group9\app.py:63:26: E0602: Undefined variable 'SettingsScreen' (undefined-variable)
DCM_group9\app.py:76:23: E0602: Undefined variable 'EgramScreen' (undefined-variable)
************* Module DCM_group9.backend
```

For example a report like this mentioned a lot of errors with the code that we were able to go back to and ensure all were handled. Of course a perfect linting score is very difficult to obtain for such a large code base, hence keeping a reasonable score became our standard. With very large code bases, scores can even become 0.1/10 usually. We decided therefore to be at atleast a 6 given our size.