

Table of Contents

1. General Description	2
2. Module Descriptions	2
2.1 Login Form	2
2.1.1 Purpose of module	2
2.1.2 Black-box behaviour of module.....	3
2.1.3 State variables	3
2.1.3 Public functions and their parameters.....	4
2.1.4 Private functions and their parameters.....	4
2.2 Edit User Form.....	6
2.2.1 Purpose of module	6
2.2.2 Black-box behaviour of module.....	6
2.2.3 State variables	6
2.2.3 Public functions and their parameters.....	7
2.2.4 Private functions and their parameters.....	7
2.3 DCM	9
2.3.1 Purpose of module	9
2.3.2 Black-box behaviour of module.....	9
2.3.3 State variables	10
2.3.3 Public functions and their parameters.....	10
2.3.4 Private functions and their parameters.....	11
3. Likely Requirement Changes.....	12
4. Likely Design Decision Changes.....	13

1. General Description

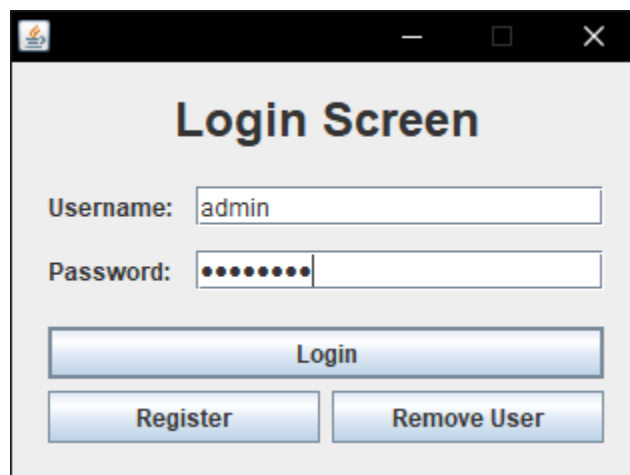
The device-controller monitor (DCM) is a top-level graphical user interface that communicates with the pacemaker and the “heart”. The DCM communicates with the hardware to send user-inputted parameters that changes the functionality of the hardware. This entire program is implemented in Java.

The DCM prompts a login screen upon start-up, and the user must either register or login with an existing account stored in a text file. The user can also remove their account if and only if they know both username and password of the account. Each account must contain a username and password, and each username must be unique. The user can login as the administrator or ‘admin’ to access further privileges in the DCM, such as opening an ‘edit user’ form that allows to see all registered users in the database, and conveniently add new users and remove any user without knowledge of their associated password.

Once a general user has access to the DCM itself, the user can freely edit any desired values in the text fields and choose to send the parameters to the pacemaker (soon to be implemented). The user can then ‘logout’ of the interface, which prompts the login form again.

2. Module Descriptions

2.1 Login Form

A screenshot of a Java Swing window titled "Login Screen". The window has a standard title bar with minimize, maximize, and close buttons. The main content area has a light gray background. At the top, the title "Login Screen" is displayed in a large, bold, black font. Below the title, there are two input fields. The first is labeled "Username:" and contains the text "admin". The second is labeled "Password:" and contains a series of black dots, indicating a password field. Below these fields, there are three buttons. The first button is labeled "Login" and is centered. The second button is labeled "Register" and is on the left. The third button is labeled "Remove User" and is on the right. All buttons have a blue gradient and a 3D effect.

2.1.1 Purpose of module

The login form accepts a username and password inputted by the user and verifies an internal database (for now, a .txt file) to check if the login is valid. The user can choose to register a new username and password, as long as the username is unique and there are less than 10 registered users in the database. The user can also remove a username and account from the database, only if both fields are valid and in the database. The purpose of this module is to limit the access of the DCM to registered users and the administrator.

2.1.2 Black-box behaviour of module

When the ‘login’ button is pressed, the module grabs the username and password and verifies it with the database. If both the username and password is in the database, then the login window closes and then the DCM window opens. If the username and password is *not* in the database, then a pop-up dialogue window appears, prompting the user that the username or password is incorrect. It is important to not let the user know if the username is in the database, so they cannot guess the password to the associated username.

When the ‘register’ button is pressed, the module checks all usernames in the database to see if there are any conflicts; if there are, a pop-up dialogue window prompts the user. Likewise, if there are not less than 10 users registered, then the user is prompted with the respective error. If everything is error-free, then the username and password is written on a new line in the text file. The user can then login with the registered username and password.

When the ‘remove user’ button is pressed, the module checks all usernames and passwords in the database and prompts the user if either username or password are incorrect—whichever is wrong is not specified to maintain privacy and obscurity. If the user attempts to remove the admin, then they are prompted that they cannot. If the username and password is found in the database, then the user is successfully removed.

2.1.3 State variables

Variable Name	Description
LOGIN_SUCCESS	A private Boolean variable. When user is successfully logs in, this variable is set true; false otherwise. This variable has a public getter function for other modules to access to ensure a successful login.
MAX_USER_COUNT	A private final variable that is unchangeable. It contains the maximum number of users that can be registered in the database.
USER_COUNT	A private variable containing the current registered users; when a new user is registered this variable increments by one, and when a user is removed this variable decrements by one.
USERNAMES	A private array containing all usernames in the database. The data gets initialized upon start-up of the login module. This array is of size MAX_USER_COUNT.
PASSWORDS	A private array containing all passwords in the database. The data gets initialized upon start-up of the login module. This array is of size MAX_USER_COUNT.
CURRENT_USER	A private string containing the current username that is logged in, upon a successful login (it is a null string otherwise). If other modules want to greet the username, then they can reference the getter function for this variable.

2.1.3 Public functions and their parameters

LoginForm()

A constructor function that references `initUserData()` and `initComponents()`. This function essentially initializes the user data/internal variables, the components of the buttons, text fields, and window of the login form. This function is required to instantiate the `LoginForm` class.

String getCurrentUser()

Returns `CURRENT_USER` if there is a successful login. Prompt error dialogue otherwise.

boolean getLoginStatus()

Returns `LOGIN_SUCCESS`. This function allows other modules to check if user has logged in.

2.1.4 Private functions and their parameters

void initUserData()

Reads the text file named “`userData.txt`” line by line and adds each username and password in the file to the `USERNAMES` and `PASSWORDS` array. `USER_COUNT` is incremented for each line in the text file, as each line represents a different user.

If no file named “`userData.txt`” is found, then the function writes a new file at the local directory with ‘`admin`’ and a default password.

void initComponents()

Initializes all buttons, text fields, and frame objects/components of the login form. Also organizes and sets the positions of all components (see screenshot at 2.1 for example).

void updateUserDataFile()

Each time a user is registered or removed, this function should get called. This function opens ‘`userData.txt`’ and writes the contents of `USERNAMES` and `PASSWORDS` into the file.

boolean userAndPassExists(String username, String password)

Accepts username and password as parameters, and then iterates through `USERNAMES` and `PASSWORDS` array. If there is a match in *both* username and password at the same index of each array, then the function returns true; if there is no match then it returns false.

boolean usernameExists(String username)

Accepts username as parameter, and then iterates through `USERNAMES` array. If there is a match of username in `USERNAMES` array, then the function returns true; if there is no match then it returns false.

int userIndex(String username, String password)

Accepts username and password as parameters, and then iterates through `USERNAMES` and `PASSWORDS` array. If there is a match in *both* username and password at the same index of each array, then the function returns the index number; if there is no match then it returns -1.

void buttonLoginActionPerformed(ActionEvent evt)

When the 'Login' button is pressed by the user, this function is called by an action listener. The passed parameter (evt) is irrelevant for this program.

The function first initializes temporary variables to store the text field inputs for username and password. If either text fields are blank, the user is prompted an error. If not blank, the username and password is sent as parameters to usernameAndPasswordExists(), and if the function return true, then it is a successful login; else, prompt the user of unsuccessful login.

Upon a successful login, a notify() method is called to wake up all suspended threads in other modules that are waiting on the login, such as the DCM program. LOGIN_SUCCESS is set to true and the CURRENT_USER is set as the username used to login. A dialogue message prompting a successful login is then displayed.

The text fields are cleared at the end of this function to let the user enter new information.

void buttonRegisterActionPerformed(ActionEvent evt)

When the 'Register' button is pressed by the user, this function is called by an action listener. The passed parameter (evt) is irrelevant for this program.

The function first initializes temporary variables to store the text field inputs for username and password. An error occurs and a dialogue prompts the user if either username or password text fields are blank, if either fields contain spaces, if username already exists in database, or if the current USER_COUNT is not less than MAX_USER_COUNT (meaning adding one more user would be over capacity).

If the text input is valid, then the current username and password stored in the temporary variables are added to the next free space in the internal USERNAMES and PASSWORDS array field. USER_COUNT is then incremented by one. Finally, the function updateUserDataFile() is called to write the new username and password the 'userData.txt'.

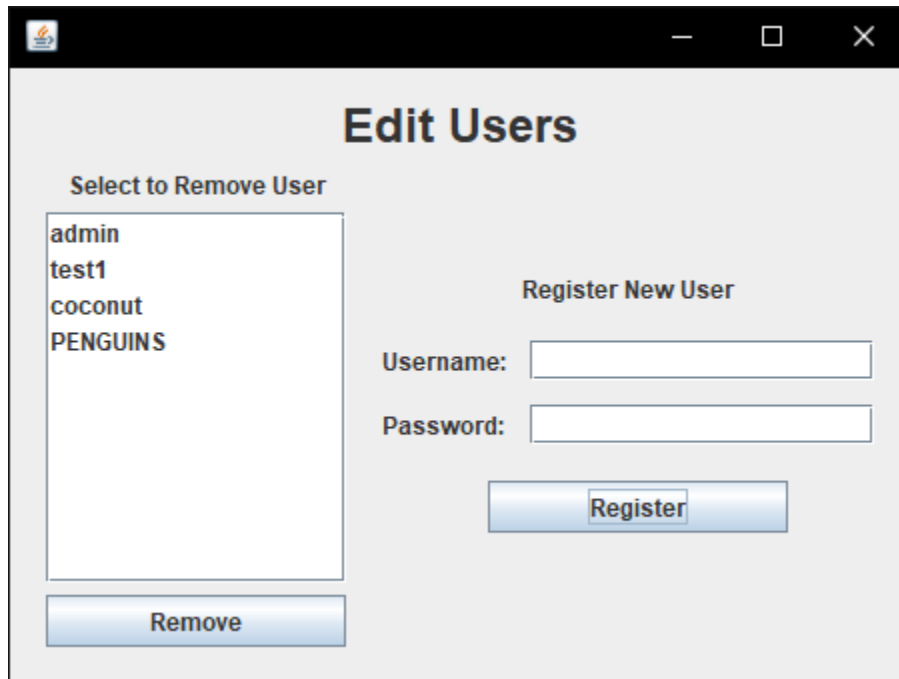
void buttonRemoveUserActionPerformed(ActionEvent evt)

When the 'Remove User' button is pressed by the user, this function is called by an action listener. The passed parameter (evt) is irrelevant for this program.

The function first initializes temporary variables to store the text field inputs for username and password. An error occurs and a dialogue prompts the user if either username or password text fields are empty, user attempts to remove admin, or incorrect username or password is entered.

If the text input is valid, then the referenced username and password in the private USERNAMES and PASSWORDS fields are deleted, and then the arrays are collapsed to account for the blank entry. Then, the global USER_COUNT variable is decremented by one, and updateUserDataFile() is called to write the new removal of the username and password in 'userData.txt'.

2.2 Edit User Form



2.2.1 Purpose of module

This module is an exclusive feature only accessible to the administrator of the program. This form is meant to easily remove/add users while seeing a live display of the users currently registered in the system. The user can click a user in the left panel and choose to remove a user (except for the admin) without knowing their password—hence, access to this form is an admin privilege. The user can also register a user on the right similar to the login form.

2.2.2 Black-box behaviour of module

When the admin clicks the user in the left panel, it is selected. If the admin then presses the ‘remove’ button, the user is removed from the panel and the database.

When the ‘register’ button is pressed, the module checks all usernames in the database to see if there are any conflicts; if there are, a pop-up dialogue window prompts the user. Likewise, if there are not less than 10 users registered, then the user is prompted with the respective error. If everything is error-free, then the username and password is written on a new line in the text file.

2.2.3 State variables

MAX_USER_COUNT	A private final variable that is unchangeable. It contains the maximum number of users that can be registered in the database.
USER_COUNT	A private variable containing the current registered users; when a new user is

	registered this variable increments by one, and when a user is removed this variable decrements by one.
USERNAMES	A private array containing all usernames in the database. The data gets initialized upon start-up of the login module. This array is of size MAX_USER_COUNT.
PASSWORDS	A private array containing all passwords in the database. The data gets initialized upon start-up of the login module. This array is of size MAX_USER_COUNT.

2.2.3 Public functions and their parameters

EditUserForm()

A constructor for the ‘edit user’ form. This method calls `initUserData()`, which initializes all user data from the database, then `initComponents()`, which initializes all the text fields, buttons and the form itself. Then, the `updateScrollPane()` method is called, which displays all registered users on the scroll pane of the window.

2.2.4 Private functions and their parameters

void initUserData()

Reads the text file named “userData.txt” line by line and adds each username and password in the file to the USERNAMES and PASSWORDS array. USER_COUNT is incremented for each line in the text file, as each line represents a different user.

If no file named “userData.txt” is found, then the function writes a new file at the local directory with ‘admin’ and a default password.

void initComponents()

Initializes all buttons, text fields, and frame objects/components of the login form. Also organizes and sets the positions of all components (see screenshot at 2.2 for example).

void updateUserDataFile()

Each time a user is registered or removed, this function should get called. This function opens ‘userData.txt’ and writes the contents of USERNAMES and PASSWORDS into the file.

void updateScrollPane()

This function updates the left scroll pane displaying all usernames. The function generates a `DefaultListModel`, a built-in class in Java, to store each username line by line in the USERNAMES array. Then, it sets the scroll pane with the `DefaultListModel`, allowing for the user to see each username and select each one.

boolean usernameExists(String username)

Accepts username as parameter, and then iterates through USERNAMES array. If there is a match of username in USERNAMES array, then the function returns true; if there is no match then it returns false.

int userIndex(String username, String password)

Accepts username and password as parameters, and then iterates through USERNAMES and PASSWORDS array. If there is a match in *both* username and password at the same index of each array, then the function returns the index number; if there is no match then it returns -1.

void buttonRegisterActionPerformed(ActionEvent evt)

When the 'Register' button is pressed by the user, this function is called by an action listener. The passed parameter (evt) is irrelevant for this program.

The function first initializes temporary variables to store the text field inputs for username and password. An error occurs and a dialogue prompts the user if either username or password text fields are blank, if either fields contain spaces, if username already exists in database, or if the current USER_COUNT is not less than MAX_USER_COUNT (in other words, adding one more user would be over capacity).

If the text input is valid, then the current username and password stored in the temporary variables are added to the next free space in the internal USERNAMES and PASSWORDS array field. USER_COUNT is then incremented by one. Finally, the function updateUserDataFile() is called to write the new username and password the 'userData.txt'.

void buttonRemoveUserActionPerformed(ActionEvent evt)

When the 'Remove User' button is pressed by the user, this function is called by an action listener. The passed parameter (evt) is irrelevant for this program.

The function first gets receives the username string from the selected name in the scroll pane. If the username is "admin", then an error message prompts the user that they cannot remove the admin, even if the user is the admin themself.

If the text input is valid, then the referenced username and password in the private USERNAMES and PASSWORDS fields are deleted, and then the arrays are collapsed to account for the blank entry. Then, the global USER_COUNT variable is decremented by one, and updateUserDataFile() is called to write the new removal of the username and password in 'userData.txt'. The function updateScrollPane() is also called to let the user show the new list of registered usernames.

2.3 DCM

DCM

p_mode ☐ VOO ☒ VVI ☐ AOO ☐ AAI

p_lower_rate_limit **p_atr_sensitivity**

p_upper_rate_limit **p_vent_sensitivity**

p_atr_pulse_amplitude **p_vrp**

p_vent_pulse_amplitude **p_arp**

p_atr_pulse_width **p_pvarp**

p_vent_pulse_width **p_hysteresis_rate_limit**

☐ **p_hysteresis_enable** ☐ **p_rate_smoothing_percent**

☐ **p_rate_smoothing_enable**

Reset Parameters **Edit Users**

Send Parameters to Pacemaker **Logout**

COM1 **Not connected** **Pacemaker Model #####**

2.3.1 Purpose of module

This is the actual interface used to communicate and send parameters to the pacemaker. The user can edit all values through this interface and can choose to send the values to the hardware. The module will then communicate to the pacemaker with the updated values. If the user is an administrator, they can access the 'edit users' form by pressing the respective button. Once a user presses the 'logout' button, then the program goes back to the login screen.

2.3.2 Black-box behaviour of module

As of now, the program doesn't do much other than store the parameters in the private instance fields from user input. In the future, it should be able to serially communicate with the pacemaker, indicate that it is connected, receive and display the model number, as well as send parameters inputted by the user to the pacemaker.

2.3.3 State variables

Variable Name	Description
p_mode	A custom datatype, PACEMAKER_MODE, that stores the modes VOO, AOO, VVI, and AAI.
p_lower_rate_limit	An integer to store lower rate limit in ppm.
p_upper_rate_limit	An integer to store upper rate limit in ppm.
p_atr_pulse_amplitude	A float to store atrial pulse amplitude in volts.
p_vent_pulse_amplitude	A float to store ventricular pulse amplitude in volts.
p_atr_pulse_width	A float to store atrial pulse width in milliseconds.
p_vent_pulse_width	A float to store ventricular pulse width in milliseconds.
p_atr_sensitivity	A float to store atrial pulse amplitude in mV.
p_vent_sensitivity	A float to store ventricular pulse amplitude in mV.
p_arp	A float to store atrial refractory period in milliseconds.
p_vrp	A float to store ventricular refractory period in milliseconds.
p_pvarp	A float to store post-ventricular atrial refractory period in milliseconds.
p_hysteresis_enable	A boolean to store whether hysteresis is enabled.
p_hysteresis_rate_limit	A float to store the hysteresis rate limit.
p_rate_smoothing_enable	A boolean to store whether rate smoothing is enabled.
p_rate_smoothing_percent	A float to store the rate smoothing percent.
ADMIN_MODE	A boolean to store if the user is logged in as admin.
IS_CONNECTED	A boolean to store if the DCM is connected to the pacemaker hardware.

2.3.3 Public functions and their parameters

DCM(String username)

This is a constructor method that initializes the DCM by calling four other methods: initComponents(), resetAllFields(), initParameters(), and initSerialPorts(). In order of the methods called, the constructor creates the text fields, labels, and buttons and assembles them onto a window, resets all text fields to the default values, stores the values in the text fields as the private instance fields, then finally attempts to communicate with the hardware to finish initialization. If a username is passed as a parameter to this method, it checks if it is equal to 'admin', and sets ADMIN_MODE true if that is the case; false otherwise.

2.3.4 Private functions and their parameters

void initComponents()

Initializes all buttons, text fields, and frame objects/components of the login form. Also organizes and sets the positions of all components (see screenshot at 2.3 for example).

void resetAllFields()

Resets all text fields to default values. For now, all the default values are zero, but this will change once we start implementing the communication with the pacemaker.

If hysteresis is disabled, then the text field for hysteresis rate limit is disabled. Likewise, if rate smoothing is disabled, then the text field for rate smoothing percent is also disabled.

void initParameters()

Sets the internal state variables of the module equal to the text field inputs.

void initSerialPorts()

This function just reads all available serial ports for now. However, once we implement communication with the pacemaker, this method will connect to a port and update the display label on the form to indicate that it is connected. Then, the model number of the pacemaker will be obtained, and be displayed on the form.

void enableSmoothingActionPerformed(ActionEvent evt)

This method updates on an action listener; if the user checks or unchecks the ‘enableSmoothing’ checkbox, then the program will respectively enable or disable the text field for rate smoothing percent.

void enableHysteresisActionPerformed(ActionEvent evt)

This method updates on an action listener; if the user checks or unchecks the ‘enableHysteresis’ checkbox, then the program will respectively enable or disable the text field for hysteresis rate limit.

void buttonEditUserActionPerformed(ActionEvent evt)

This method updates on an action listener; if the user presses the ‘edit user’ button, the method determines if the user is an ADMIN_MODE or not. If they are not an error box prompts the user that they must login as admin. If the user is the admin, then the method will initialize the ‘edit user’ form (see 2.2) and display it for the user.

void buttonResetParamActionPerformed(ActionEvent evt)

This method updates on an action listener; if the user presses the ‘reset parameters’ button, the method simply calls the resetAllFields() method.

void buttonSendParamActionPerformed(ActionEvent evt)

This method updates on an action listener; if the user presses the ‘send parameters’ button, the method simply calls the initParameters() method. This will be changed in the future, because the parameters need to be actually sent to the pacemaker through the serial port.

void buttonLogoutActionPerformed(ActionEvent evt)

This method updates on an action listener; if the user presses the ‘logout’ button, the method sends a notify() method to wake up the thread that suspends the login form, which results in the a new instance of the login form for the user to login again. Finally, the DCM form is disposed of.

3. Likely Requirement Changes

Rounding for the inputted parameters should be considered. If a user enters a value that is above a maximum or below a minimum value, then the DCM should automatically round the value down and up to the maximum and minimum value, respectively. Also, if a user enters a value that is not of the desired precision, then the program will round to the closest value. For example, if the user inputs 4.1, but the parameter requires a precision of 0.2 increments, then the input will be rounded to 4.0. Another way to counteract this problem is to implement the *spinner* component instead of text fields, so the user must increment the value at the desired increment value.

Users should be able to save their own DCM parameter values. Most likely this will be stored in their own text file in a local directory where the DCM program is stored. Also, each user should have their own nominal (default) parameter values, and they should have the option to setting their own “default”. An admin should be able to change the parameter values of all registered users.

A “help” button should be implemented to show a general user a brief guidance on how to use the program. A ‘help’ form should be implemented for this as well, which contains instructions on what each button does, and an “about” section to contact the developers/programmers.

Obviously, the DCM should be able to communicate with the pacemaker. Currently, they are disconnected, and the user will not know when the pacemaker is connected or not.

4. Likely Design Decision Changes

A hash method should be implemented to store the passwords in the text file. This is to ensure obscurity for the passwords and make the database more secure. A user should not be able to look at the text file and know all passwords associated with the username. If they are hashed using an encryption algorithm such as SHA-256, then it is impossible for the user to know the original password. Whenever a password is registered through the 'edit user' or 'login' form, the entered password is hashed and written to the text file. SHA-256 ensures 1-to-1 mapping of input and output, so only a unique password will always yield a unique hashed output.

A way to change passwords in the 'edit user' form. This would be useful if a general user requests the admin to change their password. This may not be necessary, as the general user can simply remove their account and register with the same username but with the updated password.

A design overhaul for the DCM may be required to make it more user friendly. Currently, the user will not understand the naming scheme for the parameter variables. As this is still the preliminary design, it was deemed convenient for the programmers to simply display the parameter names.

The login form should contain an image or text indicating that the user is attempting to login to the DCM program.