



SFWRENG 3K04 – Software Development

PACEMAKER DCM

Sunday November 29, 2020

TA: Sayed Ahmed

Lab 1 Group 8

Muntakim Ali – 400210016
Theodor Aoki – 400202020
Sameer Marok – 400201508
Boris Samardzic – 400204693
Gurkaran Sondhi – 400193807

Table of Contents

1. General Description	3
2. Module Descriptions	3
2.1 Top state module (RunClass)	3
2.1.1 Purpose of module	3
2.1.2 Black-box behaviour of module	3
2.1.3 Modules in use	3
2.2 Login Form (LoginForm)	4
2.2.1 Purpose of module	4
2.2.2 Black-box behaviour of module	4
2.2.3 Modules in use	5
2.2.3 State variables	5
2.2.4 Public functions and their parameters	5
2.2.5 Private functions and their parameters	6
2.3 Edit User Form (EditUserForm)	8
2.3.1 Purpose of module	8
2.3.2 Black-box behaviour of module	8
2.3.3 State variables	8
2.3.4 Public functions and their parameters	9
2.3.5 Private functions and their parameters	9
2.4 Device Controller-Monitor (DCM_Form)	11
2.4.1 Purpose of module	11
2.4.2 Black-box behaviour of module	11
2.4.3 Modules in use	12
2.4.4 State variables	12
2.4.5 Public functions and their parameters	13
2.4.6 Private functions and their parameters	13
2.5 DCM Serial Communication (DCM_SerialCOM)	17
2.5.1 Purpose of module	17
2.5.2 Black-box behaviour of module	18
2.5.3 Modules in Use	18
2.5.4 State Variables	18
2.5.5 Public functions and their parameters	19
2.5.6 Private functions and their parameters	21
2.6 Electrogram (EGRAM)	21
2.6.1 Purpose of module	22
2.6.2 Black-box behaviour of module	22
2.6.3 Modules in use	22
2.6.4 State variables	22
2.6.5 Public functions and their parameters	22
2.6.6 Private functions and their parameters	22
3. Testing	23
4. Likely Design Decision Changes	25
4.1 Hash passwords before storing them	25
4.2 Option to change passwords in 'Edit User' form	25
4.3 View singular electrogram signals	25
4.4 Change sampling rate in electrogram	25
4.5 Log atrial and ventricular signal samples from electrogram	25
5. Likely Requirement Changes	26
5.1 Memory management	26

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 must connect to a serial port to communicate with the pacemaker. Once connected, the user can view current parameters loaded onto the pacemaker as well as send parameters to overwrite the current parameters. Parameters can be edited from the user interface. User can also view the electrogram to monitor the heart’s atrial and ventricular signals.

2. Module Descriptions

2.1 Top state module (RunClass)

2.1.1 Purpose of module

This is the module that ties the whole program together. It is simply a top state finite state machine that circularly loops from the login screen to the DCM interface until the user closes any of the forms.

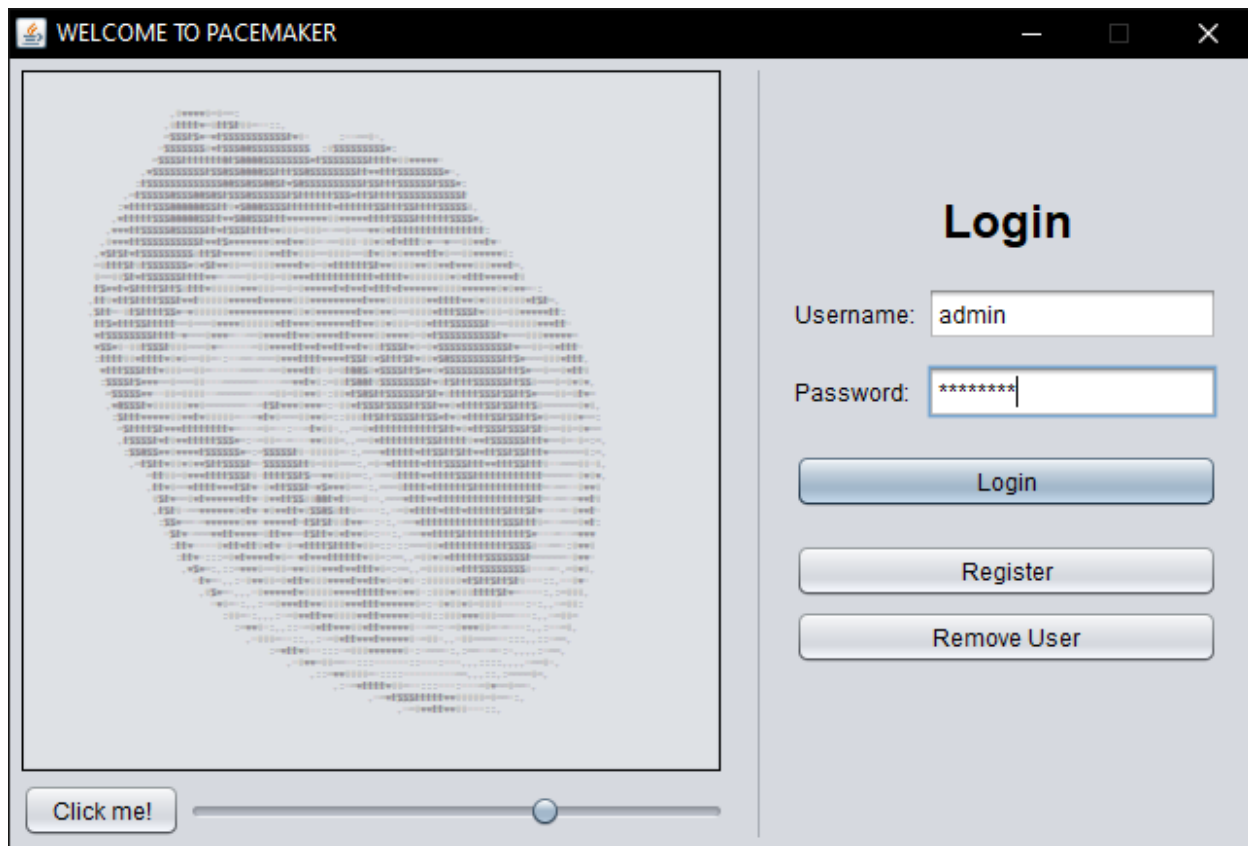
2.1.2 Black-box behaviour of module

The module starts by constructing a login form. Once the user logs in, the login form is disposed, and the DCM form is constructed. Likewise, when the user logs out of the DCM, the DCM is disposed, and the login form is constructed again, repeating the process.

2.1.3 Modules in use

Module name	Description
LoginForm (see 2.2)	The login module.
DCM_Form (see 2.4)	The main DCM module.

2.2 Login Form (LoginForm)



2.2.1 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. There is also a fun ASCII animation to let the user easily identify that this is a pacemaker program (source:

https://en.wikipedia.org/wiki/File:CG_Heart.gif).

2.2.2 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.

When the “Click Me” button is pressed, the ASCII heart animation will start playing. The slider underneath the animation adjusts the speed at which the heart beats. The user can pause the animation by pressing the button again.

2.3.3 Modules in use

Module name	Description
ASCII_Animation	The module used to animate the heartbeat.

2.2.3 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.
anim	A variable to store the ASCII_Animation object.

2.2.4 Public functions and their parameters

LoginForm()

A constructor function that references initData() and initComponents(). This function essentially initializes the user data/internal variables, the components of the buttons, text fields, and window of the login form. Also reads the assets folder to initialize the ASCII animation object. 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.

2.2.5 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.

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 userAndPassExists(), 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 to the 'userData.txt'.

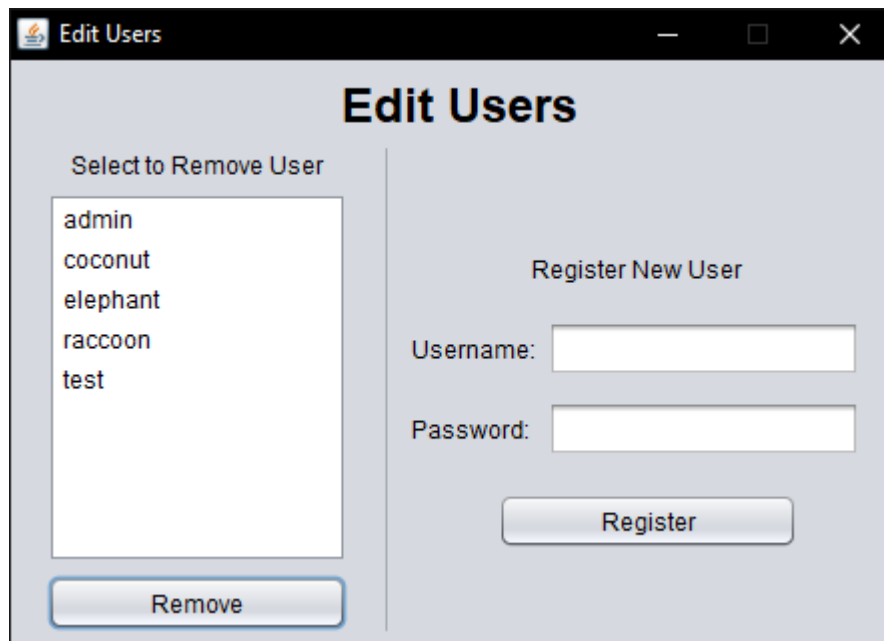
void buttonRemoveUserActionPerformed(ActionEvent evt)

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 to 'userData.txt'.

2.3 Edit User Form (EditUserForm)



2.3.1 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 admin can click a user in the left panel and choose to remove a user (except for the admin themselves) without knowing their password—hence, access to this form is an admin privilege. The admin can also register a user on the right similar to the login form.

2.3.2 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.

2.3.3 State variables

Variable Name	Description
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.
soleInstance	A private variable that stores the singleton instance of this (EditUserForm) module.

2.3.4 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.

static EditUserForm getInstance()

Returns a singleton instance of the module (the private `soleInstance` variable). This is to ensure that there is only one instance of this class instantiated, because there is no reason for there to be anymore. If the

2.3.5 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 that is hard-coded.

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 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 to 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 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 themselves.

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 to 'userData.txt'. The function updateScrollPane() is also called to let the user see the new list of registered usernames.

2.4 Device Controller-Monitor (DCM_Form)

2.4.1 Purpose of module

This is the actual interface used to communicate and send parameters to the pacemaker. The user can connect to a serial port which is connected to the device through this interface. The user can then send parameters to the pacemaker and overwrite the values on the board. Values can then be viewed later. User can also open the electrogram through this interface.

The user can save parameters by exporting the values or saving user default by pressing ‘Export Settings’ or ‘Save User’s Default’, respectively, which stores it in a local directory folder. User can also load the values at any time by pressing their corresponding buttons.

If the user is an administrator, they can access the “edit users” form by pressing ‘Edit Users’. Once a user presses the ‘logout’ button, then the program goes back to the login screen.

2.4.2 Black-box behaviour of module

The user can change the serial port they are connecting to by changing the COM in the combo box. Once the user presses “Connect” the icon in the bottom left changes to a loading circle, indicating that the interface is trying to connect to the device. Once connected, the icon should turn into a checkmark and a dialog success window should appear—otherwise, a crossed-out box will appear, and a dialog error window will appear, indicating what went wrong.

Once connected, the user can choose to “Send Parameters to Pacemaker” or “View Parameters in Pacemaker”. If the user tries pressing either one of these when the interface is not connected, an error window will appear. Otherwise, a success window will appear, indicating that the selected action has been performed.

“Export Parameters” and “Save User’s Default” will create a text file containing all current parameters in the GUI in the local working directory. “Load Parameters” will open a file chooser window that will allow the user

The user can choose to view the EGRAM by pressing “View Electrogram”. This will open a form that shows a real-time graph of the atrial and ventricular signals of the heart.

2.4.3 Modules in use

Module Name	Description
DCM_SerialCOM	A singleton instance of the DCM Serial COM module. Required to send/read parameters to/from the pacemaker.
EditUserForm (see 2.3)	A singleton object containing Edit User Form module. To ensure only one instance of the object is created.
EGRAM	A singleton object containing the EGRAM module. To ensure only one instance of the object is created.
ASCII_Animation	To animate the loading circle when connecting to a port.
jSerialCOM	The jSerialCom module which handles port communication (see https://fazecast.github.io/jSerialComm/)

2.4.4 State variables

Variable Name	Description
PacingModeList	A list of strings containing the pacing modes.
ActivityThresholdList	A list of strings containing the activity thresholds.
PacingMode	A byte containing the enumerated index of the pacing mode in PacingModeList.
LowerRateLimit	An integer containing lower rate limit in ppm.
AtrAmplitude	A float containing atrial amplitude in V.
VentAmplitude	A float containing ventricular amplitude in V.
AtrSensitivity	A float containing atrial sensitivity in V.
VentSensitivity	A float containing ventricular sensitivity in V.
AtrPulseWidth	An integer containing atrial pulse width in ms.
VentPulseWidth	An integer containing ventricular pulse width in ms.
VentRefractoryPeriod	An integer containing the ventricular refractory period in ms.
AtrRefractoryPeriod	An integer containing the atrial refractory period in ms.
MaxSensorRate	An integer containing the maximum sensor rate (lower rate limit can't be larger than this value) in ppm.
ActivityThreshold	A byte containing the enumerated index of the activity threshold in ActivityThresholdList.
ReactionTime	An integer containing the reaction time in seconds.
ResponseFactor	An integer containing the response factor (unitless).
RecoveryTime	An integer containing the recovery time in minutes.
ADMIN_MODE	A boolean storing whether the current user is an admin or not.
MODEL_NUMBER	A string containing the pacemaker model number.
USERNAME	A string containing the current username.

2.4.5 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.

2.4.6 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 2.4 for example).

boolean initParameters()

Sets the internal state variables of the module equal to the text field inputs. If the lower rate limit is larger than the max sensor rate, an error is thrown, and function returns false. Otherwise the values from the input fields are converted to be stored in the private instance fields.

void buttonSendParametersToPacemakerActionPerformed(ActionEvent evt)

Function is called on an action listener, when "Send Parameters to Pacemaker" is pressed.

First, a check is done to make sure the `DCM_SerialCOM` is connected. If it is not, an error dialog pops up and function returns. Then, `initParameters()` is called, and if it returns false, then the function returns.

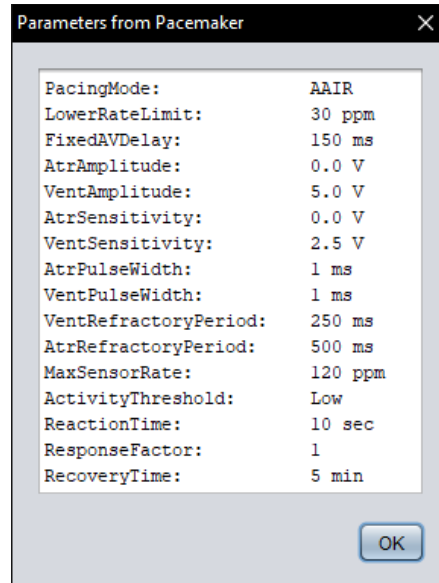
Once all checks passed, then the parameters are sent to a function in the `DCM_SerialCOM` module to be written. The module function returns false if there was an error in writing the parameters, and true if they were successfully written.

void buttonLoadParametersInPacemakerActionPerformed(ActionEvent evt)

Function is called on an action listener, when "Load Parameters in Pacemaker" is pressed.

First, a check is done to make sure the `DCM_SerialCOM` is connected. If it is not, an error dialog pops up and function returns.

If check passes, then a function is called from the `DCM_SerialCOM` module to request a copy of the byte parameters stored in an array. Once obtained, the byte array is parsed and put into a string, which is then displayed in a dialog window for the user to view.



boolean isValidSerialPort(String portName)

Given a port name, it searches if the port is still in the list of available ports seen by the OS. Returns true if there is a match, false otherwise.

void refreshSerialPorts()

Refreshes the combo box containing all the port names with the updated ports. This method is called on an action listener when the user presses the refresh ports button.

void safelyCloseConnectedPorts()

Sets the connection icon to a cross, indicating that the communication port is disconnected. Disables the “View Electrogram” button because ports will be disconnected. A disconnect method is called from DCM_SerialCOM to close the port and remove action listeners. Finally, a dialog window indicates to the user that port has been disconnected.

void buttonConnectPortActionPerformed(ActionEvent evt)

Function is called on an action listener, when “Connect” button is pressed.

This entire function is performed on a separate thread, as to not stall the program. First, the “Connect” button is disabled, so the user can’t spam this function request. Then, the loading circle animation starts playing to indicate to the user that the DCM is trying to connect to a port.

First the port name is obtained from the combo box, and it is determined to be a valid port by isValidSerialPort() function—an error window pops up and function returns if it isn’t valid. Then, the isConnected() method of DCM_SerialCOM is called to make sure that the DCM isn’t already connected to a port—if it is, then the port is disconnected to *only if the user is trying to connect to a different port*. If the user trying to connect to a port they are already connected to, then an error message pops up and the function returns. Then, the initPort() function is called from DCM_SerialCOM to check if there was a successful initialization—if there wasn’t an error window prompts user and function returns. Then, if port is successfully opened, the returnSerialCode() function from DCM_SerialCOM is called to make sure that the connected device is actually the pacemaker—if a serial code is returned from the request, then the function proceeds, and prompts error window then returns otherwise.

Once all the checks have been passed, the loading circle is paused, and the connection icon is set to a checkmark. The “View Electrogram” button is enabled so the user can view the EGRAM, and the obtained pacemaker serial number is set in user interface. The “Connect” button is also enabled again so the user can choose to connect to a different port.

Finally, a “secret” of this function is that it starts a new thread once a port is successfully opened. In this thread, it constantly polls the `isConnected()` function from the `DCM_SerialCOM` module to ensure that the port is still connected to. The poll occurs at a rate of 1 Hz, as to not overload the requests for the `DCM_SerialCOM` module. Now, if the user does something brash like yank the USB cord for this program, then all ports are safely disconnected by calling the `safelyCloseConnectedPorts()` function, and the user is prompted that the port is disconnected.

`void buttonLoadNominalActionPerformed(ActionEvent evt)`

This function is called on an action listener when the user presses “Load Nominal”.

This function simply sets all input fields to the hard-coded nominal values. These values will not ever change, so the user can always have a set of values to fall back onto.

`void loadParamsFromDirectory(String filePathDir)`

Function opens a file at `filePathDir`, the accepted parameter—if the file path does not exist, an error is thrown and the function returns. The file then reads through the file line by line and sets the input fields of the GUI as the values stored in the text file. This was implemented with a switch/case statement, which reads the parameter name in the text file, and assigns the corresponding value to the input field.

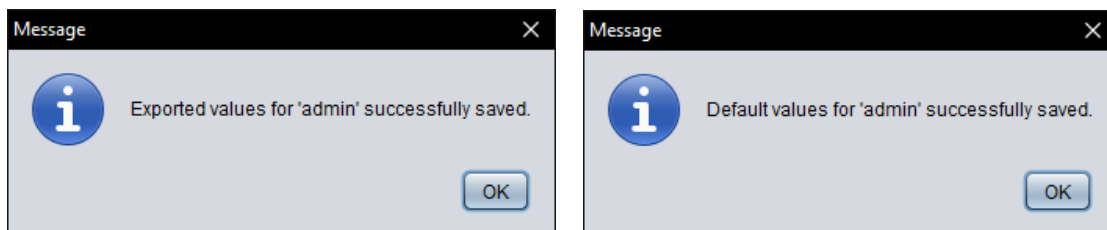
The text file that’s being read should be the files generated by the `saveParamsToDirectory()` function, since the function outputs the file in a specific format which this function was designed to read from.

`void saveParamsToDirectory(String dir, String filename, String saveType)`

Function initializes all parameters by calling `initParameters()` to load the private instance variables from the input text fields. If `initParameters()` returns false, there was an error, so the function returns.

Then, a file directory is created from the received parameters of this function (if the directory does not exist). Then a text file is created to write each parameter name and value on a separate line, separated by spaces.

Finally, a prompt window is shown to display to the user that the values were successfully saved. Depending on if the `saveType` is “Default” or “Exported”, the message is customized to notify the user of whichever save type was performed. Example:



void buttonSaveUserDefaultActionPerformed(ActionEvent evt)

Function is called on an action listener when the user presses “Save User’s Default”.

A file path directory to “/DefaultParameters/fileName.txt” is saved as a string, with the file name being the currently logged in user’s username. The file path directory is then sent to saveParamsToDirectory with the saveType being “Default”.

void buttonExportSettingsActionPerformed(ActionEvent evt)

Function is called on an action listener when the user presses “Export Settings”.

A file path directory to “/ExportedParameters/fileName.txt” is saved as a string, with the file name being the currently logged in user’s username plus a Unix time stamp to include version history and keeping the file names unique. The file path directory is then sent to saveParamsToDirectory with the saveType being “Exported”.

void buttonLoadUserDefaultPerformed(ActionEvent evt)

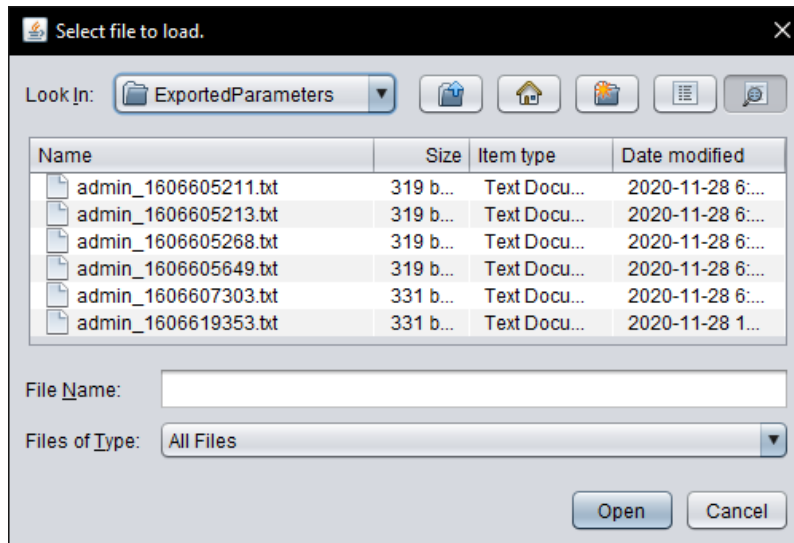
Function is called on an action listener when the user presses “Load User’s Default”.

Function creates a file path directory in /DefaultParameters/, and a file name containing the username that is currently logged in. If the file path does not exist then an error window prompts the user and the function returns. Else, the file path is then passed as a string to loadParamsFromDirectory().

void buttonLoadSettingsActionPerformed(ActionEvent evt)

Function is called on an action listener when the user presses “Load Settings”.

Function first determines if /ExportedParameters/ exists and prompts the user with an error window and returns if it does not. Then, a file chooser window is launched, as shown here:



The user can choose what file to load through this window, which is native to the Java Swing Toolkit. Once they double clicked a file or pressed “Open” with a file selected, the directory of the file is sent to loadParamsFromDirectory().

void buttonEditUserActionPerformed(ActionEvent evt)

This function is called on an action listener when the user presses the “Edit Users” button.

The method function will return and prompt the user to login as an admin if they are not. Otherwise, if the user is the admin, then the method will grab the singleton instance of the “Edit User” form and the location is set relative to the DCM form.

void buttonViewEGRAMActionPerformed(ActionEvent evt)

This function is called on an action listener when the user presses the “View Electrogram” button.

This method will only perform if it is connected to a serial port. This is just an added safety check upon the disabled button when port is disconnected. If it is connected, then the EGRAM singleton instance is grabbed, and the location is set relative to the DCM form.

void buttonHelpActionPerformed(ActionEvent evt)

This function is called on an action listener when the user presses the “Help” button.

The file path for “help.txt” is generated, which should always be consistent in the working directory. If the file does not exist, the user is prompted an error message. If the file does exist, the file is opened using the user’s default text file viewer.

void disableAllInputFields()

All input fields are disabled, and the user cannot interact with them.

void enableInputFieldsBasedOnPacingMode()

Input fields are enabled depending on what the user selected in the Pacing Mode combo box. This was implemented with a switch case statement.

void inputPacingModesItemStateChanged(ActionEvent evt)

This function is called on an action listener when the Pacing Mode combo box changes item states.

The function disableAllInputFields() is called, and then enableInputFieldsBasedOnPacingMode(). This is to make sure that the user inputs data relevant only to the pacing mode and is aware of what the pacing modes do.

void buttonLogoutActionPerformed(ActionEvent evt)

This function updates on an action listener when the user presses the ‘Logout’ button.

The function sends a notify() method to wake up the thread in the RunClass module (which is controlling this module, see 2.1). This DCM form should be disposed of and then a new login form would be generated.

2.5 DCM Serial Communication (DCM_SerialCOM)

2.5.1 Purpose of module

This module is what handles the communication requests with the pacemaker, as well as the initialization of the ports. All other modules that want to communicate with the pacemaker should do it through this module. This is because the serial port is sensitive to requests and can

only handle one request at a time—this module ensures that this criterion is met. This module implements an action listener for when UART transmissions are received, and cycles until the input buffer is fully received, so it is important that only one request's input is received, and multiple requests aren't filling the same buffer (which would result to corrupt data).

This module also does the integer/float to byte conversion for transmission over the UART protocol for when parameters are being sent. It also contains static functions such as retrieving the port name and a boolean on whether the serial port is connected to or not. Also, a disconnect function to safely close the port and remove action listeners.

2.5.2 Black-box behaviour of module

When repeatedly trying to open a port, the attempts are printed to console. A failure to open a port is also printed to console. Attempts to send or read parameter data are also printed. When the EGRAM is initialized, the atrial and ventricular values are printed to console each time the requests are made. There are other behaviours of this module, but they are all “under the hood”, and not observable from the “black-box”.

2.5.3 Modules in Use

Module Name	Description
jSerialCOM	The jSerialCom module which handles port communication (see https://fazecast.github.io/jSerialComm/)
ByteBuffer	A module native to Java that allows for converting floating point numbers to bytes, and vice versa.

2.5.4 State Variables

Variable Name	Description
IS_CONNECTED	Boolean variable storing whether the port is initialized and connected to.
GLOBAL_LOCK	Boolean that stores whether the communication port is under use.
RECEIVED	Boolean flag that is set true when input buffer is full.
OUTPUT_BUFFER[17]	A byte array to store data to send to pacemaker. The first element is always for the instruction code to send to the pacemaker, so that it knows what to return.
INPUT_BUFFER[16]	A byte array that stores input from pacemaker. It is of length 16 mainly because there are 16 parameters to be read. Serial number of the pacemaker was adjusted to be length 16.
BUFFER_INDEX	An integer storing the index for the INPUT_BUFFER.
READ_SERIAL_NUMBER WRITE_PARAMETERS READ_PARAMETERS READ_ATR_SIGNAL READ_VENT_SIGNAL READ_ATR_VENT_SIGNAL	These are the enumerated byte codes for the instructions for the pacemaker. The purpose of each variable is self explanatory by their name. These were deemed necessary to improve legibility of the program.
soleInstance	A private variable that stores the singleton instance of this (DCM_SerialCOM) module.

2.5.5 Public functions and their parameters

static DCM_SerialCOM getInstance()

Returns the singleton instance of the DCM_SerialCOM object. If the object is null, a new one is generated and returned. This is to ensure that all modules share the same instance and doesn't create a new instantiation and have multiple handles on a single port.

boolean initPort(SerialPort port)

Function initializes a port. A loop was implemented to attempt the openPort() method from the jSerialComm module, which returns a true/false depending on whether the port was successfully opened or not. If it hasn't opened and the loop occurs 5 times, the function returns false. Other modules using this method should handle this with their own implementation.

If the port was successfully opened, the port's baud rate is set to 115200, and an action listener is added to the port, which executes serialEvent(). Basically, whenever a byte has been received by the module, this serialEvent() function is called. Finally, IS_CONNECTED is set to true, and the function returns true.

The serialEvent() method is an *anonymous* function implemented solely for initialization of the port. When a byte buffer is received by the module, it will not always be 16 bytes. Sometimes it sends data in partial bytes at a time—this is a severe problem. This is why it is important to have a static INPUT_BUFFER variable to store each byte as it comes, and a BUFFER_INDEX variable to store where in the INPUT_BUFFER to next store the incoming byte data. When the BUFFER_INDEX reaches the length of the INPUT_BUFFER length, that is when we know that the incoming data is fully received, and we can set the RECEIVED flag to true, which lets the threads in other modules using this module know when they can use the data. The BUFFER_INDEX is set to 0 to reset for the next chunks of incoming data.

boolean writeParameters(...all 16 parameters)

This function suspends the current thread until GLOBAL_LOCK is set to false (when another process let goes of the lock). Then, the lock is enabled again by this thread, so other threads can't interfere.

The OUTPUT_BUFFER is set to all the 16 parameters that were passed as parameters to this function. The type conversion is handled here, as all variables are converted to bytes so that they can be sent over to the pacemaker using the UART protocol. Each parameter will occupy one byte in OUTPUT_BUFFER.

Then, sendPacemakerCode(code) is sent twice. First with WRITE_PARAMETERS as the code, then READ_PARAMETERS. This will write the parameters to pacemaker and receive them for verification. A loop iterates until the RECEIVED flag is set to true by the serialEvent() handler, and then a check is performed. Another loop iterates to verify that INPUT_BUFFER and OUTPUT_BUFFER match, and a success variable is set to false if there's a mismatch.

Finally, GLOBAL_LOCK is set to false and RECEIVED flag is set to false so other threads can use the process. The success is then returned by the function to indicate whether the parameters were successfully written to the parameter or not.

byte[] returnPacemakerParameters()

This function suspends the current thread until GLOBAL_LOCK is set to false (when another process let goes of the lock). Then, the lock is enabled again by this thread, so other threads can't interfere.

Then, sendPacemakerCode(READ_PARAMETERS) is called to read parameters, and a loop iterates until the RECEIVED flag is set to true. Then, the INPUT_BUFFER array elements are copied to another byte array which will be returned.

GLOBAL_LOCK is set to false and RECEIVED flag is set to false so other threads can use the process. The byte array is returned.

String returnSerialCode()

This function suspends the current thread until GLOBAL_LOCK is set to false (when another process let goes of the lock). Then, the lock is enabled again by this thread, so other threads can't interfere.

Then, sendPacemakerCode(READ_SERIAL_NUMBER) is called to read parameters, and a loop iterates until the RECEIVED flag is set to true. Then, a loop iterates through the INPUT_BUFFER and appends the elements to an empty string, while type-casting each byte to a character.

GLOBAL_LOCK is set to false and RECEIVED flag is set to false so other threads can use the process. The string is returned.

double[] returnAtrVentSignals()

This function suspends the current thread until GLOBAL_LOCK is set to false (when another process let goes of the lock). Then, the lock is enabled again by this thread, so other threads can't interfere.

Then, sendPacemakerCode(READ_ATR_VENT_SIGNAL) is called to read the atrial and ventricular signals. INPUT_BUFFER is read; bytes 0-7 contain the atrial signal, and 8-15 contain ventricular signal, as per the protocol for the pacemaker. However, before we read these bytes, the INPUT_BUFFER must be reversed because Java uses big endian, and the pacemaker sends the byte data in little endian. Then, the wrap() function from ByteBuffer is used to convert the byte data into double datatypes and stored in an array. The values are printed to console.

GLOBAL_LOCK is set to false and RECEIVED flag is set to false so other threads can use the process. The double array is returned.

static String getPortName()

Returns the name of the serial port. If IS_CONNECTED is false (meaning the port is not connected, and hence not initialized), then the function returns "NULL".

static boolean isConnected()

Returns IS_CONNECTED flag, which tracks whether the serial port is open or not.

static void disconnect()

If the serial port is not null, the data listener is removed, and the port is closed. Then the port is set to null. The IS_CONNECTED flag is set to false, and BUFFER_INDEX is set to 0, so that next time the port is initialized the index starts at 0.

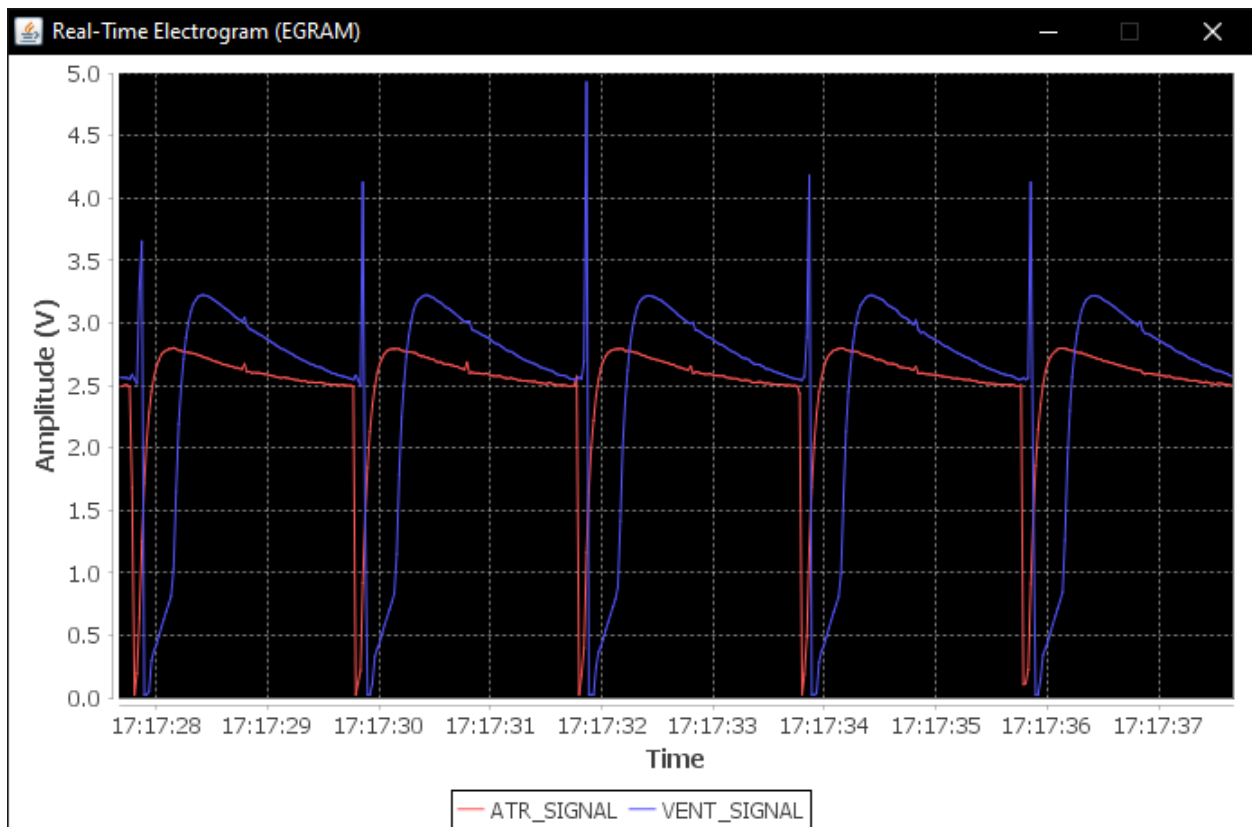
2.5.6 Private functions and their parameters

synchronized void sendPaceMakerCode(byte code)

If IS_CONNECTED is true, this function executes—it does nothing otherwise. The function sets the first variable of OUTPUT_BUFFER as the code, which was received as a parameter. The code is essential in telling the pacemaker what instruction to perform. The codes are READ_SERIAL_NUMBER, WRITE_PARAMETERS, READ_PARAMETERS, and READ_ATR_VENT_SIGNAL. The OUTPUT_BUFFER is sent to the open port for the pacemaker to receive.

The synchronized keyword is an added caution. In Java, this keyword basically ensures that only one thread has access to this method at a time. We do not want to repeatedly write data to the pacemaker, or else we may end up sending junk and get mismatched requests. The keyword may be unnecessary, because all functions that use this private method are globally locked to ensure that only one thread is using this function at a time.

2.6 Electrogram (EGRAM)



2.6.1 Purpose of module

The electrogram graphically displays all the digital atrial and ventricular signal samples received from the heart (which the pacemaker provides). The graph displays the most accurate representation of the analog signal by interpolating the missing data. The times at each sample are represented on the domain axis, and the amplitude (in volts) is represented on the range axis. The graph automatically pans from right to left to show a 10 second interval of the most recent samples.

2.6.2 Black-box behaviour of module

Atrial and ventricular signals are displayed at a sampling rate of 100 Hz.

2.6.3 Modules in use

Module Name	Description
JFreeChart	A module used to graph data (see https://www.jfree.org/jfreechart/)
DCM_SerialCOM	A singleton instance of the DCM_SerialCOM module used to communicate with the pacemaker to get atrial/ventricular signals.

2.6.4 State variables

Variable Name	Description
ATR_SIGNAL	A variable of type TimeSeries (a submodule of JFreeChart), which stores the atrial samples and their associated time.
VENT_SIGNAL	A variable of type TimeSeries (a submodule of JFreeChart), which stores the ventricular samples and their associated time.
SAMPLE_PERIOD	The period at which the samples are being taken. Currently set at 10 ms, but can be changed with future functionality of the module.
soleInstance	A singleton instance of this (EGRAM) module.

2.6.5 Public functions and their parameters

static EGRAM getInstance()

Returns the singleton instance of this module. If the instance is null, then a new instance is created.

static void destroyInstance()

Destroys the instance of this module as null. Java garbage collector will remove this object from memory.

2.6.6 Private functions and their parameters

EGRAM()

The constructor for this class. The class is an inherited version of a JFrame in Java, so it is essentially a form. In this constructor, the title of the form is set, both ATR_SIGNAL and VENT_SIGNAL are initialized as TimeSeries objects (which essentially is a list of data samples containing the time of the samples and the sample values).

A chart is then created by calling `createChart(ATR_SIGNAL, VENT_SIGNAL)`, which returns a `JFreeChart`. The colouring of the chart is set and then added to the EGRAM form. Then, a thread is initialized to call the `update()` method at a period of 10 ms (or 100 Hz), which updates the `ATR_SIGNAL` and `VENT_SIGNAL` variables to include new samples. As the variables include new samples, the chart displaying the data updates.

It is important for this constructor to be private, otherwise other classes can create as many instances of this class as possible. If other modules want to use EGRAM, then they must obtain the singleton instance through the public function `getInstance()`.

void update()

Temporarily stores the values returned by `returnAtrVentSignals()` from `DCM_SerialCOM`. Adds the new atrial signal sample the `ATR_SIGNAL`, and the ventricular signal sample from `VENT_SIGNAL`. The graph should update accordingly to the new data, which is handled by `JFreeChart`.

JFreeChart createChart(XYDataset dataSet1, XYDataSet dataSet2)

The function returns a `JFreeChart` that graphically displays `dataSet1` and `dataSet2`, which in our case would be `ATR_SIGNAL` and `VENT_SIGNAL`. The method sets the renderers and the ranges of the axis and adds the renderers to the plots. Then created chart (handled by `JFreeChart`) is then returned.

3. Testing

#	Test Case	Expected Result	Actual Result	Pass/Fail
1	Registering 11 users.	Error on 11 th registration.	Error on 11 th registration.	Pass
2	Blank text field inputs.	Error dialog prompt.	Error dialog prompt.	Pass
3	Registering existing username.	Error dialog prompt.	Error dialog prompt.	Pass
4	Incorrect username OR password.	Error dialog prompt.	Error dialog prompt.	Pass
5	Removing admin.	Error dialog prompt.	Error dialog prompt.	Pass
6	'userData.txt' does not exist.	Creates new text file with 'admin' and default password.	Creates new text file with 'admin' and default password.	Pass
7	Exporting settings or saving default settings when /ExportedParameters/ or /DefaultParameters/ does not exist.	Create the directory and notify user.	Creates the directory and notifies the user.	Pass
8	Loading settings or saving default settings when /ExportedParameters/ or /DefaultParameters/ does not exist.	Error dialog prompt.	Error dialog prompt.	Pass
9	Connecting to an already open port used by another device.	Error dialog prompt.	Error dialog prompt.	Pass
10	Connecting to an available port,	Connection timeout	Connection timeout	Pass

	but no device is connected.	error.	error.	
11	Connecting to port for pacemaker.	Update serial code, icon becomes checkmark, and success notification window.	Update serial code, icon becomes checkmark, and success notification window.	Pass
12	Sending parameters to pacemaker.	Success notification window.	Success notification window. But, if the user continuously spams the button, then an error will pop up. This just means you just have to try sending again though.	Overall pass.
13	Loading parameters from pacemaker.	A dialogue window containing the variables loaded onto pacemaker.	A dialogue window containing the variables loaded onto pacemaker.	Pass
14	Pressing spinner arrow out of range of values.	The spinners should hold value when increasing from max value or decreasing from min value.	The spinners holds value when increasing from max value or decreasing from min value.	Pass
15	Refreshing ports when already connected to a port.	Disconnect from port.	Disconnects from port.	Pass
16	Connecting to the same port again.	Error dialog prompt.	Error dialog prompt.	Pass
17	Connecting to different port from current port.	Dialog prompt indicating port disconnection and start connecting to new port.	Dialog prompt indicating port disconnection and start connecting to new port.	Pass
18	Pressing help button with missing 'help.txt' file.	Error dialog prompt.	Error dialog prompt.	Pass
19	Changing pacing modes in GUI.	Enable/disable appropriate input fields.	Enables/disables appropriate input fields.	Pass
20	Inputting higher lower rate limit than max sensor rate.	Error dialog prompt.	Error dialog prompt.	Pass
21	Sending/viewing parameters to/from pacemaker when disconnected to a port.	Error dialog prompt.	Error dialog prompt.	Pass
22	Logging out when connected to a port.	Dialog window indicating port disconnected, then login form is shown, and DCM is disposed.	Dialog window indicating port disconnected, then login form is shown, and DCM is disposed.	Pass
23	Yanking the USB cable when port is already connected to.	Error dialog prompt, and icon changes to cross.	Error dialog prompt, and icon changes to cross.	Pass

24	Manually editing the saved parameters stored in text files, then loading them.	Loads them regardless if they are out of range.	Loads them regardless if they are out of range. This shouldn't be an issue for the doctor, as they can simply load the nominal values (which are hard-coded).	Pass
25	Loading parameters from a text file that are not in format.	Does nothing.	Does nothing. This is good because we don't want to read junk data.	Pass

4. Likely Design Decision Changes

4.1 Hash passwords before storing them

- 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 a unique password will always yield a unique hashed output.

4.2 Option to change passwords in 'Edit User' form

- This would be useful if a general user requests the admin to change their password.
- May not be necessary, as the general user can simply remove their account and register with the same username but with the updated password.

4.3 View singular electrogram signals

- Option to view only atrial or only ventricular signals in the electrogram could be beneficial
- This could be useful for the doctor if they only care about one signal.

4.4 Change sampling rate in electrogram

- Currently the sampling rate is at a fixed rate of 100 Hz.
- This could be bad for people with poor performance computers, and the electrogram may lag their system because they cannot handle that many requests.
- A slider to change the sampling period could be nice.

4.5 Log atrial and ventricular signal samples from electrogram

- A way to log the data into a .csv file could be useful for doctors that want to save the graphical data numerically.
- Also, a good way to view and manipulate the data outside of the electrogram.

5. Likely Requirement Changes

5.1 Memory management

- Java is an extremely memory hungry program. Since all variables and objects are allocated on the heap, size on RAM can get huge.
- This can be fixed by reimplementing the program on a lower-level language, such as C++, and properly deallocating memory when not in use.
- Or perhaps there is a memory leak in the program which has yet to be investigated.