Textual explanations

Design patterns used

The mediator pattern design pattern was used. It was used inorder to allow classes to communicate with each other without having a lot of subclasses. The mediator object in this case was DCS. Using this design pattern allowed us to reduce dependencies and sub classes. While most of our classes used DCS as the mediator object some of them did not. Thus, we did not strictly follow the mediator pattern design pattern. Also, the command design pattern inspired the design of our device. We used the command design pattern due to the fact that we used various UI tools in order to send requests to our program.

Design decisions made

Heart Rate Graph Design Decisions
Instead of simply reading in from a file, we chose to abstract the heart rate data generation to the Device class, as this mimicked a physical device reading in a pulse from a user. Our heart rate data implementation leveraged the <cmath> library's *sine* method. We used this method to generate heart rate data whose values resembled a sinusoidal function. In order to achieve various levels of coherence in the data, we added a *noise* modifier to add some chaos to the sinusoidal values. Since our control class (DCS) had access to the Device, it simply had to poll the Device class for the heart rate data when it needed it.

Session History/Log Design Decisions
We chose to maintain a sort of database for our sessions. Instead of a file, we leveraged the Active Record Pattern and created a class for data storage, allowing us to have queries and mutations for our data when we needed it. This DataStorage class was polled anytime we needed persistent data from an execution of the program. We made use of this in our Session History feature - when the user asked to view their history, we retrieved their sessions from the DataStorage, and passed them into our Display class to handle the front-end work. Our Display class abstracted the viewing of an individual session and all the sessions, so the device control system simply needed to call the required view. When the user wished to delete a session, we captured their request in the Display class, but signaled to the DCS class to handle that deletion request. The DCS class then used the deletion mutation in our DataStorage class to remove the session, and refreshed the UI by once again calling to the appropriate Display class view.

Session data design decisions
We initially had our session serve as a bonafide data store, containing the final values for the session - however after deciding to use the Active Record Pattern, we pivoted to having our

Session class be a sort of in-memory object record, representing a Session in our database, but also providing accessor methods and and mutator methods for that record. This allowed us to abstract much of the Session functionality to its own Session class, and keep the rest of the code neater and more streamlined.

Breath Pacer Design Decisions

We chose to have a BreathRegulation class that contains a QProgressBar. While the project specifications said that the breath pacer is in the form of a strip of lights on the machine itself, or a ball going back and forth on the session screen, we choose to implement the breath pacer as a QProgressBar. The progress bar would be visible if the user indicated they wanted to use the breath pacer in settings. Also, the user is able to select the breath level using a QSpinBox. Then in BreathRegulation the progress bar is initially set to 100% and then the progress bar gradually decreases. The progress bar gets to zero after the indicated amount of time in the settings tab.

Battery Design Decisions

There is a battery charge indicator visible on all of the screens rather than just the session screen in order to give the user a more clear indication of the state of the device. The Battery class contains a QProgress bar, which starts at 100% and decreases by 1% every second. Upon reaching 10%, an alert is displayed to the user warning them of low battery, and can only be dismissed after the user presses the selector button. If the warning occurs during a live session, the session continues. However, if the battery reaches 0% during a live session, the session is saved, ends, and the device “dies” (is powered off). The latter behavior is the same for all other screens. The battery level can be recharged to 100% with the help of the “Recharge Battery” simulation button.

Session Disconnect Design Decisions

The sensor connect symbol is visible only on the session frame as that is what is most intuitive/necessary/ the only frame that actually depends on the sensor being connected in order to function. At any time, if the HR sensor is disconnected, an alert will display that can only be resolved with the sensor being reconnected (this can be done with the help of the simulation buttons). If this alert is triggered during a live session, the session ends and is saved, and the HR sensor connected symbol (which is a red heart) disappears.

Display Class and UI Design Decisions

Battery, BreathRegulation, DCS, Display, Session, all inherit from QObject in order to make use of slots and signals, which helped greatly with abstracting UI elements to their appropriate classes in a clean and streamlined fashion. All QObjects that come from mainwindow.h that need to be used during runtime, (e.g., QFrames, QGroupBoxes, QPushButtons, etc.) with the exception of Battery and BreathRegulation, are declared as attributes within the Display class. Other classes that need these objects can obtain them from the Display class with appropriate getter methods. This cleans up the code a lot, and keeps all UI elements in 1 respective class, and separates them from actual logic while still being able to utilize them. The vast majority of all slot and signal connections are in mainwindow.cpp, in order to keep the code clean.