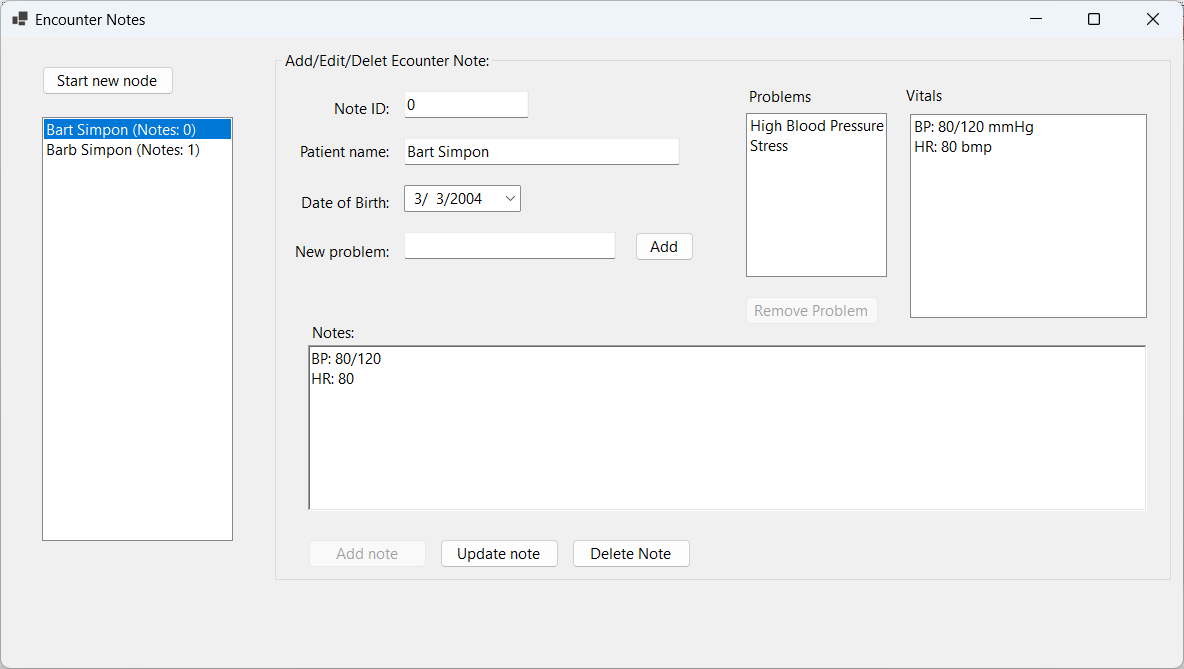
# Assignment #3: More UI, OOP, Collections, File I/O, and Regexes (30 marks)

## Introduction:

The goal for this assignment is to build a basic app to create, edit, read, and delete clinical encounter notes that will be stored in a simple text file. As such, in this assignment you will be practicing:

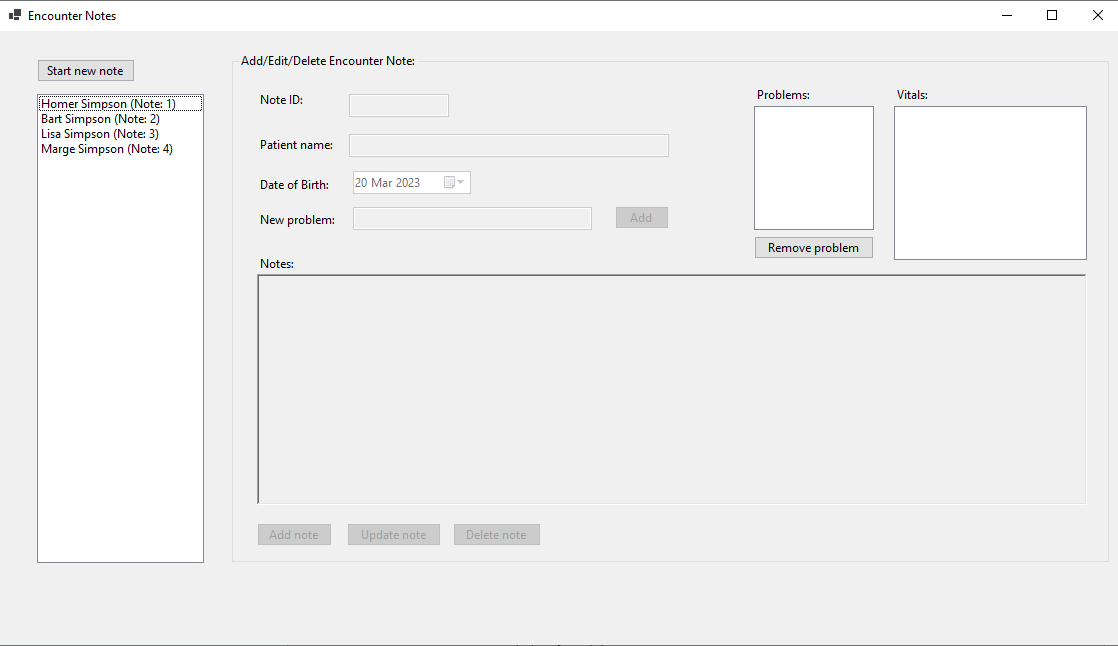
* Some object-oriented design and defining classes in a separate class library
* File I/O
* More Win Forms UI interactions and application state management
* And Regexes to extract vitals (e.g. Heart rate (HR), Blood Pressure (BP), etc. readings from free text clinical notes.

I have provided start code solution (in the assignment folder). And I will do a demo of a working solution in class but also the following screenshot provides you with a sample UI to aim for:



## The core functionality:

When the app starts it will read the currently stored encounter notes from a file and load a view of them into the listbox on the left hand side showing only the patient name and note ID, i.e. as such:



It will also place the right hand side, i.e. the "*Add/Edit/Delete Encounter Note*" side, in “*awaiting note mode*” by disabling most of the controls. From there the user can do 1 of 2 things:

* Highlight a past note (i.e. by selecting it in the listbox)
* Start a new note by clicking on the button

### Highlighting a past note:

* This loads the properties of that note into the appropriate fields of the "*Add/Edit/Delete Encounter Note*" side
* And places that side in “*edit mode*” - i.e. the “Update note” and “Delete note” buttons are enabled but the “Add note” button is disabled.
* The user can then make any edits to the fields (except of course the ID field which is always disabled from user editing) and save those changes using the “Update note” button.
  + If the fields are valid then the updated note should be saved immediately to the file.
  + However, if the fields are invalid then the update should not happen and instead the user should be informed of the validation errors.
  + See more below for validation requirements.
* Or the user could delete the note by pressing the “Delete note” button:
  + This should delete the note immediately from the file
  + And remove it from the listbox on the left hand side
  + You can choose whether the app should just highlight another note in the listbox (if there is one) or put the right hand side in the “*awaiting note*” mode mentioned above.

### Starting a new note:

* This places the "*Add/Edit/Delete Encounter Note*" side in “*add mode*” which means that the “Add note” button is enabled whereas the “Update note” and “Delete note” buttons are disabled.
* The user can then enter the required fields and save this note by clicking on the “Add note” button:
  + If the fields are valid then:
    - The note should be saved to file
    - Added to the left hand listbox
    - And the right hand side placed in “*edit mode*”
  + Otherwise the user should be informed of the validation errors
    - See more below for validation requirements
    - And the right hand side should stay in “*add mode*”

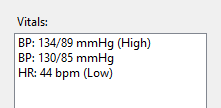
### Managing a list of patient problems:

* As indicated in the demo video, the tool allows to manage, separately from the note, a list of problems the patient might have, e.g. Diabetes.
  + When a problem is added using the button the problem should end up in the problems listbox and the textbox cleared.
  + The current problems should always be saved to file when a note is saved.
  + The delete problem button should delete the currently selected problem in the listbox or, if none are selected, then there should be an error message that a problem was not selected.
    - Deleting must be from both the listbox and the stored note when it is saved.

## Extracting Vitals (e.g. Blood Pressure, Heart Rate, etc.) readings from Note Text:

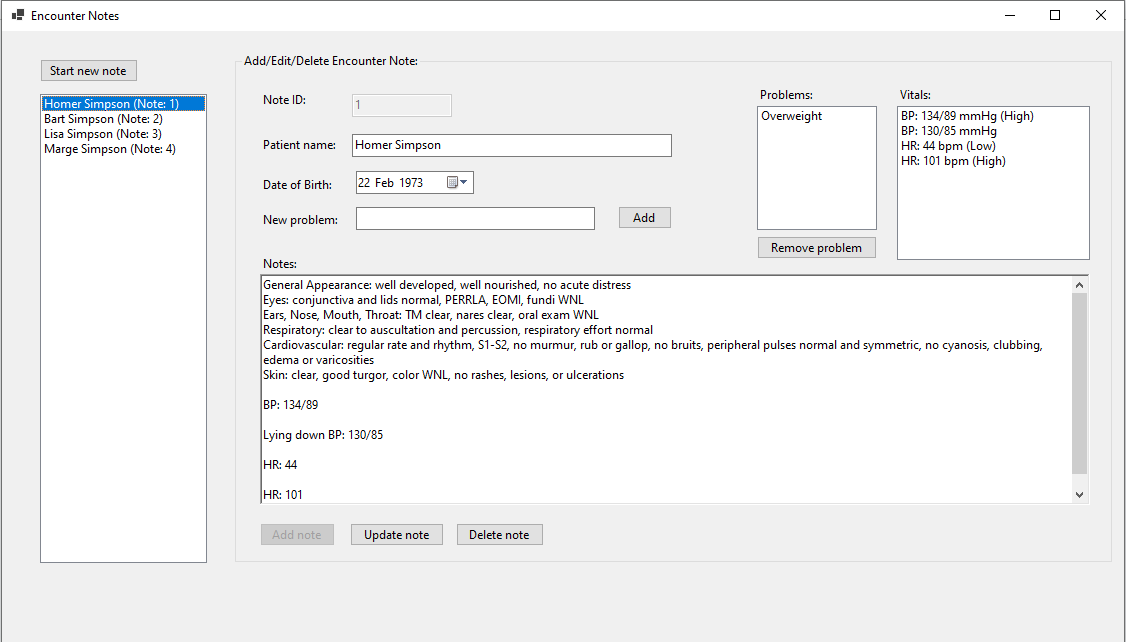
As text is entered in the rich text box for “Notes” your solution should look for text matching patterns that represent vitals, e.g. a blood pressure (BP) reading, a heart rate (HR), etc.

* For BP measurements this traditionally means:
  + Starts with BP
  + Optionally a colon
  + A space
  + 2-3 digits for the systolic (i.e. first)value
  + A slash
  + 2-3 digits for the diastolic (i.e. second) value
* For HR measurements this means:
  + Starts with HR
  + Optionally a colon
  + A space
  + 2-3 digits for the beats per minute (bpm) value
* For temperature (T) and respiratory rate (RR) the pattern will be the same or similar.
* You can read more about vitals but for simplicity let’s use the following:
  + Normal HR range is 60 to 100
  + For BP measurements, a measurement is:
    - Low if systolic is below 90 ***and*** diastolic is below 60
    - High if systolic is above 130 ***and*** diastolic is above 80
  + Normal RR range is 12 to 16 breaths per minute (bpm)
  + Normal T range is 36.5 to 37.2 degrees Celsius
* Once matches are found, they should be added to the “*Vitals*” listbox.
* Also, when an existing encounter note is loaded in the "*Add/Edit/Delete Encounter Note*" side, its vitals should be extracted from the text and also loaded in that same listbox.
* Also, when displaying vitals in the “*Vitals*” listbox be sure to:
  + Show the type of vital, e.g. BP for Blood pressure, HR for Heart rate, RR for respiration rate, and T for temp
  + Show the units
  + Highlight/show when the measurement is High or Low
  + For example, the following shows 2 BP and 1 HR measurements with their units, one 1 low and 1 high.



* To implement these vitals you need to consider how best to do this - do you need any classes? If so, how many? One per vital type? If multiple classes, do inheritance and polymorphism play a role?

The following is another screenshot showing the app in edit mode:

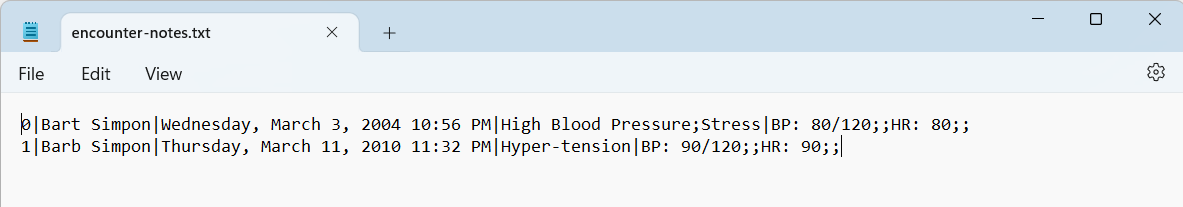


## File I/O:

In terms of reading and writing encounter notes to/from a file, keep the following points in mind…

* All edits - i.e. adding a new note, updating an existing one, or deleting an existing note should be reflected in the file immediately.
* As mentioned earlier, when the app starts it should load the notes from the file and show them in the left hand listbox by highlighting the patient’s name and note ID.
* As for what file format, you are free to explore other formats, but the following represents a viable solution:
  + 1 encounter note per line
  + The 5 properties of each encounter note should be delimited by a special character, e.g. the | (i.e. pipe or vertical slash) character.
    - Why only 5 properties? Because the BP readings are derived from the note content they do not need to be stored separately.
  + Then, among those 5 properties, the following need to be split further still, using a different special character, e.g. the ; (i.e. semicolon) character: Problems and Note Content.
    - For the Note Content, the ; (or other delimiter of your choosing) delimits the lines of text in the note content - i.e. it effectively replaces the newline character.

Thus, if you adopt this file format, the encounter note for Lisa Simpson in the first screenshot above would look like this in a text file:



## Validation of User Input for both Add & Editing Encounter Notes:

When adding/editing an encounter note there is some validation of the data required, namely:

* The note ID is an integer that must be unique.
  + As such, it is likely best that the app controls assigning these values and not let the user define/edit them. For example, the textbox for it should always be disabled and the app then picks the next available ID and pre-populates the textbox with that value when the user clicks on the "Start new note" button.
* Patient name is required
* Date of birth is required and cannot be in the future.
* And the clinical note content is required.

## Implementation details:

For this assignment you must practice some OOP skills so at the very least you must:

* Define at least 3 classes:
  + One representing a clinical note
  + One that is responsible for managing clinical notes, including the reading and writing of them from/to a file
  + One that represents a vital (e.g. BP, HR, etc.) measurement
    - As discussed previously, this can be done in multiple ways but one possibility is to have a hierarchy of Vitals classes and polymorphism used at the very least on the ToString method.

## What to hand in?

Bundle and upload your solution folder to the assignment drop-box on eConestoga ***as a single zip file***. You may submit multiple versions but only the last (i.e.most recent) submission will be looked at and graded.

### How will it be graded?

Please see the marking sheet that accompanies the assignment. It will be used to calculate your grade so be sure that you are meeting all the criteria included there. Please ensure that you submit only your own work! Remember - all your code submissions will be run through [MOSS](https://theory.stanford.edu/~aiken/moss/), the software plagiarism detection tool.