# COMP 6411: ASSIGNMENT 2

***Important:***

1. *Assignments must be submitted on time in order to received full value. Appropriate late penalties (< 1 hour = 10%, < 24 hours = 25%) will be applied, as appropriate.*

2. *It is the student's responsibility to verify that the assignment has been properly submitted. You can NOT send an alternate version of the assignment at a later date, with the claim that you must have submitted the wrong version at the deadline. Only the original submission will be graded (though you can certainly re-submit multiple times before the due date).*

3. *These are individual assignments. Feel free to talk to one another and even help others understand the basic ideas. But the source code that you write must be your own.*

4. *You may use any modules found in the Clojure Standard Libraries. You can NOT utilize any external third-party libraries. The graders will not have these when evaluating your submissions.*

5. *Do not use build automation tools like Leiningen. These tools are very useful for building big projects but will only complicate matters for a simple assignment like this. You should be able to run your code from the command line simply by invoking the main clojure executable.*

6. *This is an assignment designed to expose you to traditional functional programming style. As such, any imperative style iteration macros – such as **for, doseq, dotimes, while** – cannot be used. Instead, iteration can either be done with your own recursive functions, or with sequence application function such as **map, reduce**, and **filter**, where Clojure effectively applies an expression recursively for you.*

DESCRIPTION: So it's time to try a little functional programming. In this case, your job will be to develop a very simple Sales Order application using the Clojure language. REALLY simple. In fact, all it will really do is load data from a series of three disk files. This data will then form your Sales database. Each table will have a "schema" that indicates the fields inside. So your DB will look like this:

**cust.txt**: This is the data for the customer table. The schema is

```
<custID, name, address, phoneNumber>
```

An example of the cust.txt disk file might be:

```
1|John Smith|123 Here Street|456-4567
3|Fan Yuhong|165 Happy Lane|345-4533
2|Sue Jones|43 Rose Court Street|345-7867
```

Note that no error checking is required for any of the data files. You can assume that they have been created properly and all fields are present. Each field is separated by a "|" and contains a non-empty string. All text is case-sensitive so "John" and "john" are different people. There are no duplicate records/customers. Finally, note that the records can be stored in any order. In this case, for example, customer 3 is listed before Customer 2.

**prod.txt**: This is the data for the product table. The schema is

```
<prodID, itemDescription, unitCost>
```

An example of the prod.txt disk file might be:

```
4|gum|1.25
5|eggs|2.98
1|shoes|14.96
2|milk|1.98
3|jam|2.99
6|jacket|42.99
```

Again, the data is valid – no duplicates and text is case sensitive.

**sales.txt**: This is the data for the main sales table. The schema is

```
<salesID, custID, prodID, itemCount>
```

An example of the sales.txt disk file might be:

```
2|2|2|3
3|2|1|1
1|1|1|3
4|3|3|4
```

The third record (salesID 1), for example, indicates that John Smith (customer 1) bought 3 pairs of shoes (product 1). Again, you can assume that all of the values in the file (e.g., custID, prodID) are valid.

So now you have to do something with your data. You will provide the following menu to allow the user to perform actions on the data:

```
*** Sales Menu ***
------------------

1. Display Customer Table
2. Display Product Table
3. Display Sales Table
4. Total Sales for Customer
5. Total Count for Product
6. Exit

Enter an option?
```

**Note**: We will want to clear the screen each time the menu is displayed. With our docker version of Clojure, we will do this simply by printing an *ansi escape sequence* to the screen, as in:

```
(print "\u001b[2J")
```

The options themselves will work as follows:

1. You will display the contents of the Customer table. The output should be similar to the following:

   1: ["John Smith" "123 Here Street" "456-4567"]
   2: ["Sue Jones" "43 Rose Court Street" "345-7867"]
   3: ["Fan Yuhong" "165 Happy Lane" "345-4533"]

   Note that exact formatting does not matter. You can use commas as separators or round brackets instead of square brackets. The important thing is that each record lists the ID first, followed by the data associated with the ID. Records should be sorted by ID (in this case 1, 2, and 3).

   As noted earlier, the records are NOT guaranteed to be sorted in the data file. In addition, ID numbers are not guaranteed to be consecutive numbers (e.g., the IDs could be 7, 3, 2, 9, 14)

2. Same thing for the Product table – it will be sorted by Product ID (again, the data file may not be sorted)

3. The Sales table is a little different. ID values aren't very useful for viewing purposes, so the custID should be replaced by the customer name and the prodID by the product description, as follows:

   1: ["John Smith" "shoes" "3"]
   2: ["Sue Jones" "milk" "3"]
   3: ["Sue Jones" "shoes" "1"]
   4: ["Fan Yuhong" "jam" "4"]

   Again, the list should be sorted by Sales ID (the data file may not be sorted)

4. For option 4, you will prompt the user for a customer name. You will then determine the total value of the purchases for this customer. So for Sue Jones you would display a result like:

   Sue Jones:  $20.90

   This represents 1 pair of shoes and 3 cartons of milk (in our simple example). As noted previously, names are case sensitive, so in our simple application "John" and "john" are different people. (This just makes it a little easier for you).

5. Here, we do the same thing, except we are calculating the sales count for a given product. So, for shoes, we might have:

   Shoes: 4

   This represents three pairs for John Smith and one for Sue Jones.

   **Note** that if a given customer or product does not exist when using menu option 4 or 5, an appropriate response is given (either a "cust/prod not found" message or total sales/count = 0 ). Your program should not crash or give false results.

6. Finally, if the Exit option is entered the program will terminate with a "Good Bye" message. Otherwise, the menu will be displayed again.

One final note about the menu listing. The menu and option output should not scroll down the display. Instead, there should be a confirmation prompt, as illustrated below:

```
*** Sales Menu ***
------------------

1. Display Customer Table
2. Display Product Table
3. Display Sales Table
4. Total Sales for Customer
5. Total Count for Product
6. Exit

Enter an option? 1


** Customer Table **
1: ["John Smith" "123 Here Street" "456-4567"]
2: ["Sue Jones" "43 Rose Court Street" "345-7867"]
3: ["Sue Smith Yuhong" "165 Happy Lane" "345-4533"]
4: ["Dan Woo" "46 Boose Street" "345-9080"]
5: ["Billy Boo" "19 Gomer Court" "345-1113"]
6: ["Tom Jones" "4445 Sue Lane" "356-7895"]
7: ["Mohammed Falah" "4560 Gaff Road" "456-4533"]
8: ["Tammy Zifa" "19 Threp Road" "345-4443"]
9: ["Moo Doo" "2 Pizza Place" "345-5677"]
10: ["Sam Bo" "456 Dale Place" "345-9872"]
11: ["Mike Faiza" "234 Call Road" "456-5647"]


Press any key to continue...
```

So that's the basic idea. Here are a few final points to keep in mind:

1. You do not want to load the data each time a request is made. So before the menu is displayed the first time, your data should be loaded and stored in appropriate data structures (you can do this any way you like but something like `(def myFoo (build_data_structure))` would make sense).
2. This is a Clojure assignment, **not** a Java assignment. So Java should not be embedded in the Clojure code for any important functionality. It might be necessary to use Java classes, for example, to convert text to numbers in order to do the sales calculations. That's OK, but Java should not be used for much more than that.
3. The I/O in this assignment is trivial. While it is possible to use complex I/O techniques, it is not necessary just to read the text files. Instead, you should just use *slurp*, a Clojure function that will read a text file into a string (there are also `split` and *split-lines* string functions that are quite useful. For the input from the user, the *read-line* function can be used.
4. Do not worry about efficiency. There are ways to make this program more efficient (both the data management and the menu), but that is not our focus here. I just want you to use basic functionality to try to get everything working.
5. Do NOT try to code the full assignment from the start. You will likely create a mess that simply doesn't work. Start simple and slowly add additional features. So play with the menu first and provide the functions to display file content. Then try sorting the basic data. Only then should you start adding additional complexity to the process.
6. Keep your functions small. Do not try to create functions that do 4 or 5 different things. This works for imperative programming but is awkward for functional programs. Instead, use *function composition* to produce a sequence of operations. This might be something like

`(add_foo_logic (add_baz_logic (add_bar_logic input_data)))` This will essentially act as a pipeline that processes the input_data with `add_bar_logic`, then pipes this intermediate output to `add_baz_logic` and then `add_foo_logic`.

7. Use `let` expressions to prepare data/input in some way and assign this logic to a label/binding. You can then use the bound name in the subsequent expression. This will make the code easier to read and simpler to debug/implement.
8. When possible, use the *apply-to-all* functions to encapsulate complex logic in a simpler expression. This includes `map, reduce, filter, apply`, etc. These functions are very powerful, relatively simple, and completely avoid the requirement to explicitly implement recursive functions. Of course, you may still need to write one or two recursive functions yourself.

DELIVERABLES: The code will be organized into two source files, one containing the menu, the other containing the logic for loading and manipulating the data files. The first source file is called `menu.clj` and the second will be `db.clj`.

In order for `menu` to be able to access the functions in `db.clj,` we must define a *namespace* for `db.clj`. If you have any library *aliases*, they can also be added here. So something like the following in `db.clj` would be fine:

```
(ns db
  (:require [clojure.string :as string]))
```

You can also provide a namespace for `menu`. The name doesn't actually matter, however, since `menu` will run directly from the command line.

**Important**: The Clojure interpreter will NOT automatically look for imported modules in the current folder. So we must specify this explicitly. There are a number of ways to do this (including a `deps.edn` file), but the Clojure distribution on docker is a little more limited. Plus, we want to keep things very simple.

So we are simply going to set the `CLASSPATH` environment variable so that it includes the current folder. From the command line in docker you can type:

```
export CLASSPATH=./
```

This would have to be set each time you logged in. If you want this to be set permanently – and you are working on the docker Linux distribution - you can simply add the `export` line to the `.bashrc` file in the `/root` folder.

If the Classpath has been set properly, you can now run your application as:

```
clojure -M menu.clj
```

or just `clojure menu.clj` for older Clojure versions

Once you are ready to submit, compress the two .clj source files into a zip file. Do not submit any .txt files – the graders will provide their own. The name of the zip file will consist of "a2" + last name + first name + student ID + ".zip", using the underscore character "_" as the separator. For example, if your name is John Smith and your ID is "123456", then your zip file would be combined into a file called a3_Smith_John_123456.zip". The final zip file will be submitted through the course web site on Moodle. You simply upload the file using the link on the assignment web page.

*Good Luck*