Updated self assessment

William Haskins

**Software Design & Engineering**

***Structure***

* The code completely and correctly implements the design.
* The code follows default coding standards.
* The code is well structured, and consistent in style and formatting.
* There are not uncalled-for or unneeded procedures nor any unreachable code.
* There are no leftover stubs or test routines in the code.
* No code can be replaced by library functions.
* There are no blocks of repeated code that could be condensed.
* Storage is inefficient. Account information could be offloaded onto a separate database.
* No symbolics are used beyond string constants.
* No modules are excessively complex.

***Documentation***

* The code is adequately documented with an easy-to-maintain commenting style.
* The comments are consistent with the code.

***Variables***

* All variables are properly defined with meaningful, consistent, and clear names.
* All assigned variables have proper type consistency.
* There are no redundant or unused variables.

***Loops and Branches***

* All loops and logic constructs are complete, correct, and properly nested.
* The most common cases are tested first in IF- -ELSEIF chains.
* All cases are covered in IF- -ELSEIF chains.
* Every case statement has a default.
* Loop termination conditions are obvious and achievable.
* Indexes are properly initialized just prior to loops.
* No statements enclosed within loops can be placed outside their loop.
* The code in the loop does not manipulate the index variable outside of the loop.

***Defensive Programming***

* No arrays or files are used.
* Input data is not tested for validity but has no effect on the program when invalid.
* All output variables are assigned.
* The correct data is operated on in each statement.
* Memory allocation is deallocated.
* No error traps are used for external device access.
* No files are accessed.

**Databases**

***Structure***

* The code completely and correctly implements the design.
* The conforms to default coding standards.
* The code is well-structured, and consistent in style and formatting.
* There are no uncalled-for or unneeded procedures nor any unreachable code.
* There are no leftover stubs or test routines in the code.
* No code can be replaced by external calls or library functions.
* No blocks of code can be condensed.
* Storage is efficient.
* No symbolics are used beyond string constants.
* No module is excessively complex.

***Documentation***

* The code is clearly and adequately documented with an easy to maintain commenting style.
* Comments are consistent with the code.

***Variables***

* All variables are properly defined with meaningful, consistent, and clear names.
* All assigned variables have proper type consistency.
* There are no redundant or unused variables.

***Defensive Programming***

* Indexes and subscripts are tested against records/files.
* Imported data is not tested for validity or completeness.
* All output variables are assigned.
* The correct data is operated on in each statement.
* Memory allocation is deallocated.
* No error traps are used for external device accesses.
* Files are checked for existence before attempting to access them.
* All files and devices are left in the correct state upon program termination.

**Algorithms and Data Structures**

***Structure***

* The code completely and correctly implements the design.
* The code conforms to default coding standards.
* The code is well structured, consistent in style, and consistently formatted.
* There are no uncalled-for or unneeded procedures, nor any unreachable code.
* There are no leftover stubs or test routines in the code.
* No code can be replaced by library functions.
* No blocks of code can be condensed.
* Storage use is efficient.
* Symbolics are used for lists. This is so that those same lists can be tested on and is vital to the design. Otherwise, no other symbolics are used beyond string constants.
* No modules are excessively complex.

***Documentation***

* The code is clearly and adequately documents with an easy-to-maintain commenting style.
* All comments are consistent with the code.

***Variables***

* All variables are properly defined with meaningful, consistent, and clear names.
* All variables have proper type consistency.
* There are no redundant or unused variables.

***Loops and Branches***

* All loops, branches, and logic constructs are complete, correct, and properly nested.
* The most common cases are tested first in IF- -ELSEIF chains.
* All cases are covered in IF- -ELSEIF blocks.
* Every case statement has a default.
* Loop termination conditions are obvious and achievable.
* Indexes are properly initialized just prior to loops.
* No statement enclosed in a loop can be placed outside of its loop.
* The code in the loops does not manipulate the index variable outside of the loop.

***Defensive Programming***

* Pointers and indexes are tested against the array.
* Input data is not tested for validity but has no effect on the program when invalid.
* All output variables are assigned.
* The correct data is operated on in each statement.
* Memory allocation is properly deallocated.
* No error traps are used for external device accesses.
* No files are accessed.

**Arithmetic Operations**

No arithmetic was used in these artifacts, but I would like to cover my approach to software that does rely on mathematics.

Firstly, it is best to avoid comparing floating-point numbers for equality, because sometimes those numbers will have a slight variation. For example, “0.01” might actually be “0.0100000000002” in the machine’s logic, and the software developer will have no way of knowing this. Using floating point numbers for greater than or less than operations is more reliable. You would still need a way to deal with an equivalent case, depending on the program’s case.

Next, it is best to avoid rounding errors. The simplest way to do this is to always round up, or always round down. This will make the program more predictable, which is ideal.

Next, it is a best practice to avoid additions and subtractions on numbers with greatly different magnitudes. This can simply be avoided altogether. It might lead to integer overflow or underflow errors if this practice is not followed.

Finally, make sure to test divisors for zero or noise. While it’s simple to just not place a 0 or a non-numeric variable as a divisor, the problem will creep up when using a numeric variable. As a variable, that number might be subject to change, and if it’s subject to change, it might be a 0 in some cases. In such examples it is best to check for a 0 just before the division operation.