Assignment Seven | Week Three Assignment

Jacob A. Laqua & Kenny Still

IT 372 Software Maintenance and Evolution | Green River College

Assignment Seven | Week Three Assignment

**KWIC Technical Summary**

Note – The revised design used for this assignment is modeled closely after the provided revised design provided in class.

Link to repository: <https://github.com/kenstill46/KWIC-Project>

The primary tactic used to decompose the first design from the KWIC program description was the clear need for File IO implementation. Strong File IO implementation is a necessary portion of both the first and revised designs. Knowing this, beyond reading and writing to the files the program simply consists of parsing lines down to each word of a given line to enable circular shifts. The second design implementation is much more resilient than the former. This is in part to the modular aspect and reduction of dependencies. In the first design, there are two main classes which perform all of the program tasks. With this, any changes run the risk of breaking some portion of the program. If one of the classes changes, the other will have to change as well to accommodate. This is a poor design which requires much more time and resources to make simple updates to the program and will force JUnit testing to be more involved.

For the first implementation to accept database use in lieu of File IO, the entire filer.java class would have to be re-written. Better yet, an entire new class should be designed to interact with the database. Ideally, recycling the same method names and return types in the filer.java class there shouldn’t be much conflict if any with the KWIC.java class. This is of course dependent on the data from the database being massaged in such a manner that once brought into the system, it can act just like before when using the circulate method and so on. For the revised design to permit database use, the only thing that would have to change is the logic in the LineIO.java file. The rest of the system does not care how the internal logic of this class derives the data because whether the lines were received from a database or file because it simply interacts with methods that run off the line data found. The system simply assumes that the lines were read in properly and interacts with the LineManager class.

In either deign, the main methods could remain in place. However, in the first design, the main method would launch some scene, likely using JavaFX. In the second design, the main method would also launch a scene. However, with this design, the method calls that are below the main method in the Main.java class should be able to remain as well since they are static public methods that can be called easily from anywhere. It may be a little unorthodox but the scenes for the GUI would call these static methods (unorthodox because a controller should decide what data to pass around not the view) and display results accordingly, much like the command-line version. Instead of simple System.out logic for feedback from the system, there would be Text objects that would receive data to show accordingly.

Another possible change to this system would be more features in terms of processing the line data. For the revised design, there would be the addition of methods in the LineManager class to handle new operations. Consequently, there would have to be related methods added to the Main class to interact with the internal logic of the LineManager which we are trying to keep hidden. In the first design, since there are only two classes, adding additional line features would be as simple as writing new methods in the Filer class. The KWIC class simply calls Filer methods accordingly. So, all in all, either design with the addition of new line features would have similar impact. With the information hiding aspects of the revised design, there would be more time spent adding logic to the LineManager class and the Main class, however, the design is more sturdy and secure so it’s justifiable.

While the first design may seem easier to understand because it’s less files to research, the drawbacks that occur as the system grows would be increasing in number. It wouldn’t take long for the Filer class in the first design to get huge and hard to read, taking up a lot of vertical space. In contrast, the modular pattern in the revised design, promote keeping methods and classes small as recommended by modern design pattern and clean code books. More files which are easier to digest are much more maintainable and useful for new developers coming in compared to minimal files that are large in size. With all this being said, the revised design, reads much more like human language (like a good program should) whereas some of the logic in the former design can appear cryptic, especially as it gets more advanced.

In the short term, the original design is likely better performing because there are less files and thus less overhead in compilation with the Java Virtual Machine. However, as these few files grow, the design would become slower than the revised version because essentially all of the files have to be complied every time. For example, if a feature is added to the Filer class which keeps growing, the entire file would need to be re-compiled. In the revised design, since there are more files that are smaller, a change in one file which is small would re-compile much quicker with all of the untouched files remaining compiled. This smaller impact in compilation can make a world of difference as the system is scaled to the enterprise level and re-compilation of the entire system would be extra time-consuming, as discussed in class. With all this being said, the overall performance of the revised design is better, and more scalable, which is an integral part of good software.

As touched on earlier in this write-up, the revised version incorporates information hiding for the internal LineManager logic, using high-level method calls which are static in the Main class. This ensures that all a user directly interacts with during run time is the forward facing static methods which are essentially simple translations of the business logic in the LineManager class. This way, the user and the program understand the convenience of using the circulate method for example, but how that method actually works isn’t important. This information hiding also reduces the chances of outer classes like the Main class altering the instance variables and other concerns in the internal structure (LineManager, LineIO). In contrast, the original design, doesn’t incorporate information hiding and the internal mechanics of the implementation are not quite safe.