**GLIMPSE Software Design Document**

**Michael Wu**

**August 5th, 2019**

**1. Introduction**

This document is meant to provide documentation and insight on the current code and implementation of the GLIMPSE graphical interface. The intention of this report is that it will be passed onto future developers looking to modify and improve the GLIMPSE code. In this report, I will not only explain how the code currently functions, but also provide suggestions into how to fix and modify the code into an optimal state.

GLIMPSE is the graphical interface developed for the Global Change Assessment Model (GCAM) which aims to make the GCAM model more intuitive and accessible for both scholars and policymakers. GCAM is developed by the U.S. Environmental Protection Agency (EPA) and aims to model air emissions in order to predict specific emissions amounts by considering a multitude of factors.

GCAM utilizes XML files as the input for the model. Since it is somewhat complex to understand and modify the XML files in the format that GCAM properly understand, GLIMPSE modifies the XML files through the graphical interface. The goal of GLIMPSE is that it will eventually be utilized by policymakers and students to help understand and predict future air emissions. Since it takes so long to develop expertise in understanding GCAM inputs and outputs, GLIMPSE hopes to remedy it by simplifying the process.

**Note**

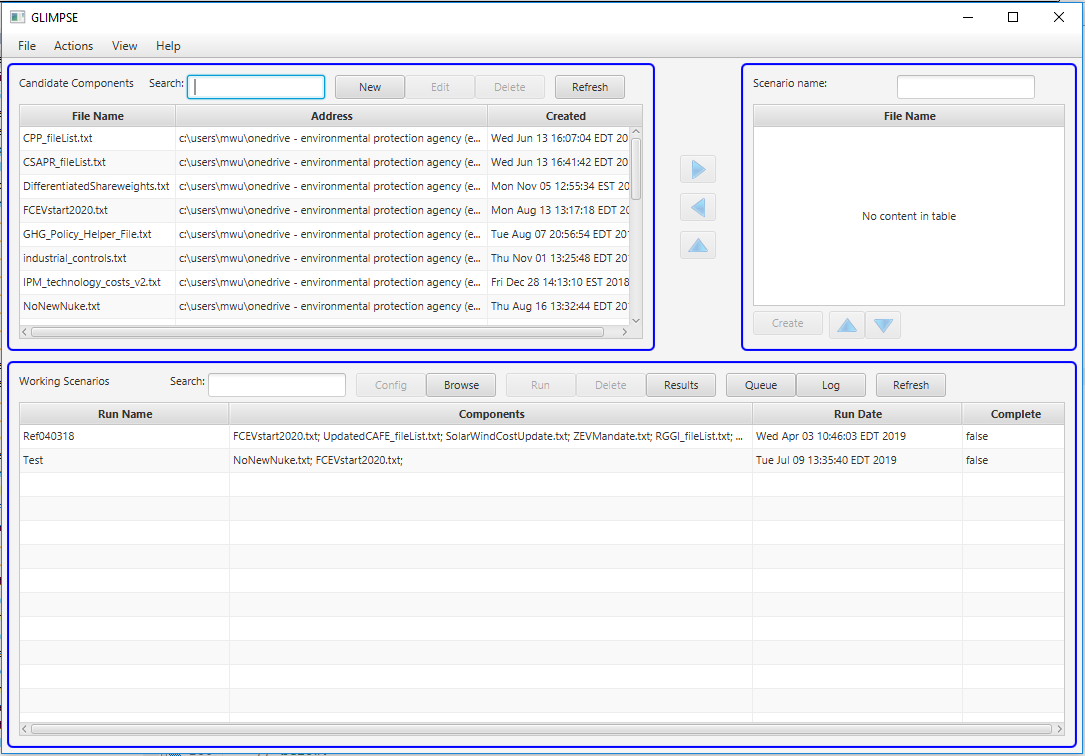
An important note to make before continuing is that the author of this document is an undergraduate student and will be attending his third year of college the following semester. I have worked on the GLIMPSE code the summer of 2019 as part of my summer internship through ORISE. Due to my relative inexperience, any of my suggestions made here in the document should be taken with a grain of salt. If there are any issues or questions that arise, feel free to contact me at [michael.wu@emory.edu](mailto:michael.wu@emory.edu).

**2. Design Overview**

**2.1 Overview**

In this section I will cover how the code generally functions and the important methods and classes to pay attention to. I will primarily focus on the current iteration of the functionality as opposed to its initial implementation.

Below is an image of the current graphical interface.



**2.2 Application Functionality**

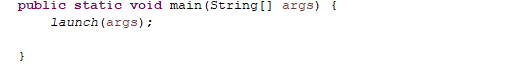
The application that launches is from the Application class. Documentation of the Application class can be found online. The top bar of the application contains a menu with four tabs. Additionally, each blue outlined box is a pane. The top left pane is referred to as PaneCandidateComponents throughout the code, the top right pane is referred to as PaneCreateScenario, and the bottom pane is referred to as PanRunTable. Each of the panes contain a Table. The rows and columns containing the information are the tables. On each of the panes are several different buttons. Additionally, in the middle between the top left and the top right pane are 3 arrow buttons. These buttons move scenarios from the top left to the top right pane. The up-arrow button pulls a created scenario up to the top right pane for modification.

**2.3 Run Configurations**

The run configurations argument is a string. This string is just the filepath to the options file named options\_ORD-GCAM-USA-4p3.txt. If you load up GLIMPSE and it fails to create any scenarios and appears blank, try to make sure that the run configurations parameter is input correctly.

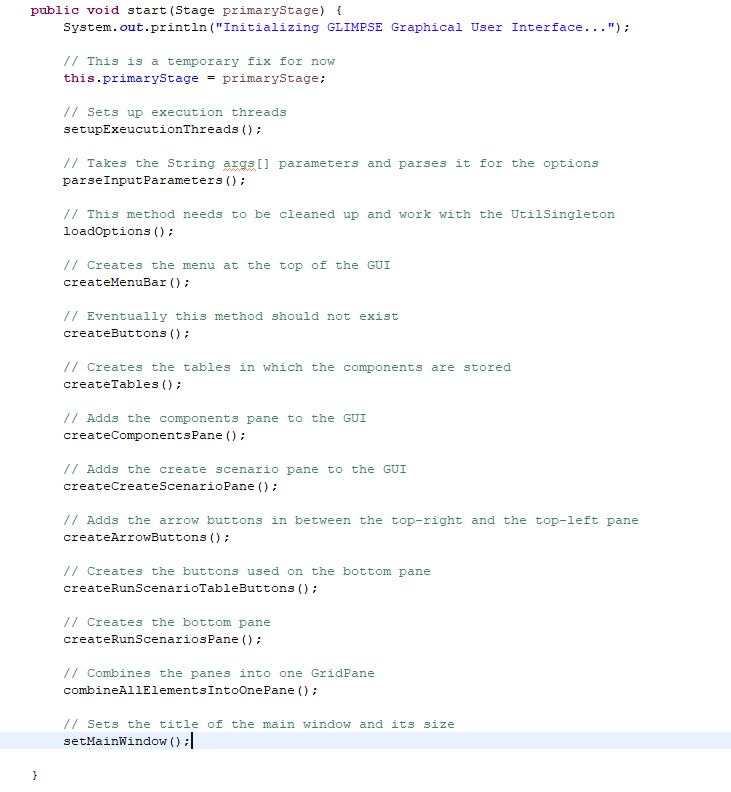
**2.4 main() Method**

The main method has one line of code which simply just states:



The launch method is a method inherited from the Application class that calls the start() method and creates the application that you see.

**2.5 start() Method**



The start method is the best location to begin analyzing the code to try and understand it. I broke down this method to only contain method calls such that it is clear what each action does. setupExecutionThreads() is a method that creates execution threads. This is to ensure that when multiple runs are going on, they execute in the proper order. The method parseInputParameters() takes in the run configurations args to load the options. In the future this can be changed so that instead of having to manually enter the location of the options file, there could be a pop-up dialogue box in which you enter it instead.

**2.6 loadOptions() Method**

The method loadOptions() loads the options using the parsed parameters from above. At the moment the loadOptions() method is not ideally coded but it is functional and works. In the future it may be worth cleaning up.

**2.7 createMenuBar() Method**

This method creates the tabs on the menu. Each separate tab is its own class. For example, the file tab’s class is named SetupMenuFile. Below is an image of the method.



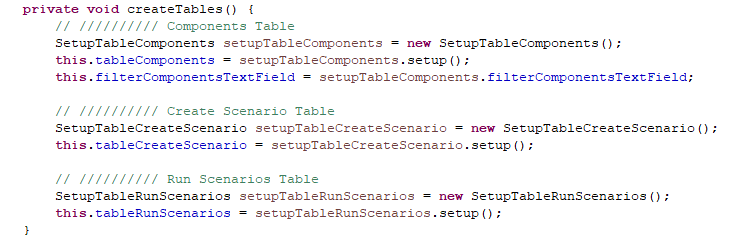
This is a decent example of what my work during my internship attempted to achieve. Previously, all four tabs of the menu were all congregated in the start() method which created a lot of clutter. By separating them into another method which in turn calls their class to instantiate them, it makes it so that editing the tabs is a lot more straightforward. Hopefully

**2.8 createButtons() method**

This method hopefully will not exist once somebody finishes working on GLIMPSE. To understand why, it’s important to understand how the buttons were originally coded. At the beginning, each button was declared above at the start of the file. Later, the createButtons() method instantiates some (not all) of the buttons in the graphical interface. Then later down the line in the start() method, the buttons were added to the graphical interface. Finally, the buttons were then given their functionality in the Pane inner classes. However, some of the buttons had their functionality given in the start() method, half of the buttons on the bottom pane did not follow this format, and many of them were inconsistent in their form.

To remedy this, the buttons should all be their own class. For example, the Results button should be its own class that simply instantiates and creates the results button. All the button’s functionalities should lie in one area. It’s declaration, functionalities, and instantiation would all be together. Therefore, having a method that only declares the buttons should be shifted away from.

**2.9 createTables() method**



This method creates the tables in which the scenarios and their components are stored. As you can see it follows the same format as I have described above in which each table is its separate class. In this way it is more evident what object performs which duty.

**2.10 Pane Creation Methods**

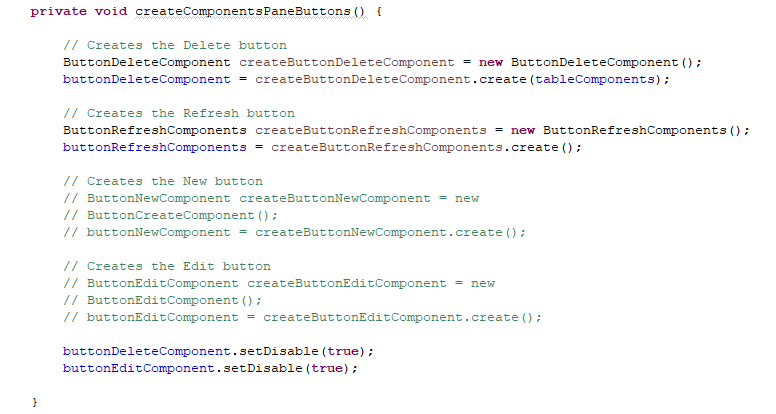
This covers the three pane creation methods: createComponentsPane(), createCreateScenarioPane(), createRunScenarioPane()

The top left pane is referred to as the components pane. The top right pane is referred to as the create scenario pane. The bottom pane is referred to as the run scenario pane. Each method creates their respective pane. An important note to make here is that these methods make use of the inner classes such as PaneRunTable/PaneCandidateComponents/PaneCreateScenario.

**2.11 Button Creation Methods**

This covers the following methods: createArrowButtons(), create componentsPaneButtons(), createCreateScenarioPaneButtons, createRunScenarioTableButtons()

The createArrowButtons() method creates the three arrow buttons that move the components between the top two panes. The following three methods create the buttons for their respective panes. However, these methods are incomplete since the button instantiation and the inner class decoupling is the most difficult aspect of this code.



Above is an example of one of the button creation methods, in this case the components pane. As we see here, the buttons are instantiated. However, the New button and the Edit button are commented out. This is because they are heavily coupled with the inner classes. In the future this should be hopefully how the buttons work.

**2.12 combineAllElementsIntoOnePane() method**

This just adds all three panes into one GridPane.

**2.13 setMainWindow() method**

This sets the main window name and sets the default size of the window.

**3. Initial Implementation**

**3.1 Overview**

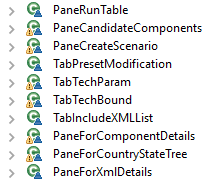
In this section I will describe the code in the state that I received it in at the start of my appointment at the EPA. Since the work that I have done is a work in progress it is important to understand where my work initially began and the issues that I have resolved and some of the issues I have yet to resolve.

**3.2 Code Length**

The primary issues with the initial code was the length of the code. The primary class I received, originally named Final53.java had 5500 lines of code. This is generally regarded as poor coding practice as any one class should not have this much responsibility. Since Final53.java performs too many duties, it is colloquially known as a “God” class. Initially the start method began on line 277 and ended on line 775. By breaking down this code into more methods and separate classes that perform only one duty, it would make editing and modifying the code much more straightforward. Originally, the start method began on line 277 and ended on line 755. These 500 lines of code made it really confusing to follow what each line of code did.

**3.3 Inner Classes**

Composed of the last 3000 lines of the code are the following inner classes:



These inner classes are the primary reason why the length of the code is extremely long. It is the most difficult issue to resolve. The reason being is that there is such tight coupling between the inner classes and methods in the Glimpse file. Ideally all these inner classes would be moved into separate classes such as PaneRunTable.java. However, due to the difficulty of removing the inner classes this was something that I was unable to complete.

**3.4 Refactoring**

Several variable names were originally inconsistent. To remedy this, I worked on refactoring and renaming the variables. Final53.java was additionally renamed to Glimpse.java. However, there has been some talk about changing its name to ScenarioBuilder.java instead. Additionally, the

**3.4 Version Control**

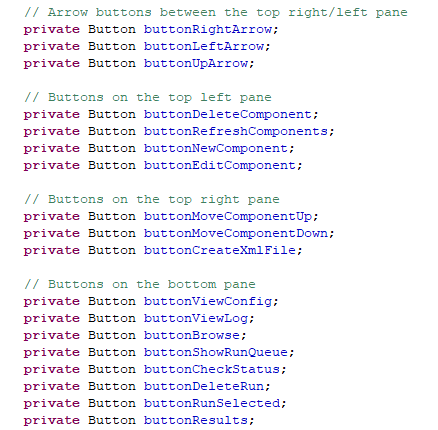
Additionally, there are some issues regarding good development practice. When I arrived, there was no version control in place.

**4. Current Implementation**

**4.1 Overview**

This details how the current code functions. Specifically, I will go more into detail on some topics

**4.2 Buttons**

****

Above is a picture of the declared buttons at the top of the Glimpse.java code. Currently, here are the buttons that I believe I have correctly decoupled and separated into their respective class.

Top left pane – DeleteComponent, RefreshComponent

Top right pane – MoveComponentUp, MoveComponentDown

Bottom pane – ViewConfig, Browse, ShowRunQueue, ViewLog

Arrow buttons – RightArrow, LeftArrow

Therefore, the following buttons are unfinished – UpArrow, NewComponent, EditComponent, CreateXMLFile, CheckStatus, Run, DeleteRun, Results

If the buttons are not being instantiated through their own class, typically they will be created through one of the various inner classes.

**4.3 Inner Classes**

A button to try and decouple that might help you understand the issue with the inner classes it the NewComponent button. This class makes use of several of the tab inner classes. To create this button, it is necessary to instantiate about four different tab inner classes. Since the inner classes would ideally be shifted away from, this means that one would have to first extract those four inner classes and place them into their own class. Not only that, this button also interacts with the PaneCandidateComponents inner class. Understanding why the inner classes are troublesome will help you a lot with understanding where to proceed with the code.

**4.4 UtilMethods**

UtilMethods is the name of the class that contains a lot of helper methods that are used throughout the code. It contains things such as the warming message method or other various useful methods.

**4.5 UtilSingleton**

UtilSingleton is a singleton class that contains all the filepath strings. This is meant to help organize the filepaths instead of having all the strings public and static. However, this is implemented only partially throughout the code and it may be better to shift away from this.

**5. Suggestions for Future Implementation**

**5.1 Decouple Inner Classes**

The number one issue with the code is its current length. In order to break down the buttons on the graphical interface into separate classes, the inner classes should be broken down and decoupled. This will be the primary task that will take the longest time. If the inner classes are removed from the main file, the length of the code will be shorted significantly. Additionally, if a helper method is utilized by the inner classes or by one specific method, it can be moved into that class file, further reducing the file length. Ideally, the final length of Glimpse.java should lie around 500-1000 lines of code.

**5.2 Organize Button Classes**

As mentioned above, the buttons on the graphical interface should be broken down into their respective classes. In order to compartmentalize the button classes, decoupling the inner classes is necessary first since some of the inner classes add the button functionalities. Check the description above to better understand the individual button classes.

**5.3 Properly Implement Version Control**

Bitbucket was somewhat set up for this project. In the future it should be standardized and used by anybody who plans to work on the project. At this point in time, there is a Bitbucket repository online. However, it should be better utilized from the start and be constantly updated.

**5.3 Software Design Documentation**

This document is somewhat of a strange version of a software design document. In the future documentation of GLIMPSE code should be done with either JavaDoc or a proper SDD website.

**5.4 Platform Independence**

Ideally GLIMPSE should be platform independent so that it can be run on both mac and windows. Since most of the filepath strings are hard coded into GLIMPSE currently, it should be looked at.

**5.5 File Organization**

Current GLIMPSE folder organization is somewhat messy. It is split into two folders named GLIMPSE and GCAM-GUI. Strangely enough, the GLIMPSE folder contains the GCAM model, while GCAM-GUI contains GLIMPSE and its code. Additionally, the project is named commandPrompt which is somewhat confusing. Reducing the number of extraneous files needed, such as files that need to be deleted or moved elsewhere would be key.