Title: Data Analysis Tools
Project Part 6: Final Report
Team: Alex Sheehan, Elias Bezanis, Tyler Glotz

1. List the features we implemented (table with ID and title)

UR-03	As a user, I need to import .txt and .csv files, so that I can use the tool to manipulate data	Functional	High
UR-04	As a user, I need to store data from files into a database, so that I can access them and combine them	Functional	High

UR-08	As a user, I need to analyze table data using standard visualization and mathematical functions (min, rotate, etc.)	Functional	High

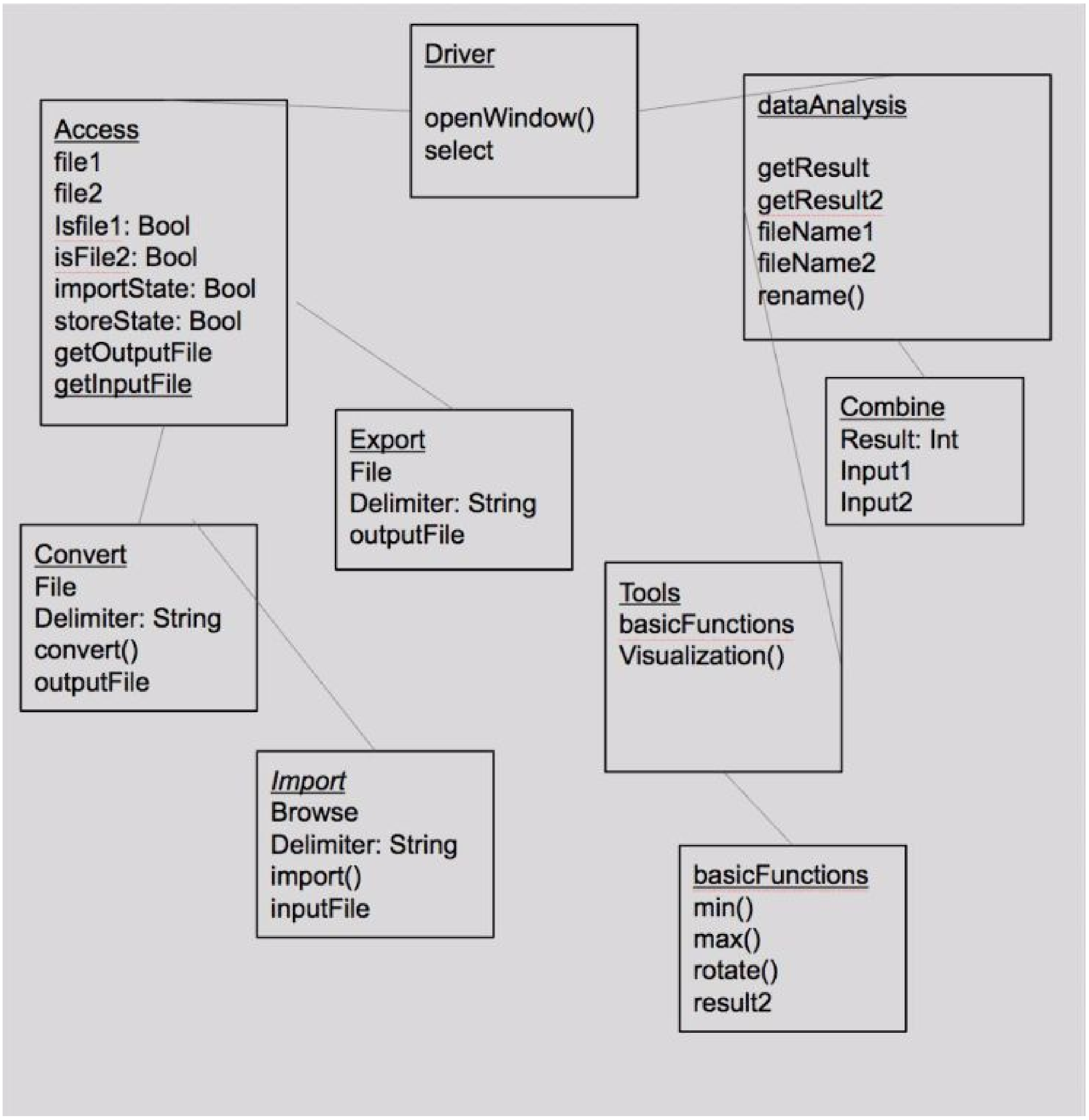
2. List the features not implemented (table with ID and title)

UR-05	As a user, I need to compare data from between two files	Functional	High
UR-06	As a User I need to be able to keybind shortcuts for functions	Functional	Low
UR-07	As a user, I need to combine rows and columns in csv and txt files	Functional	High

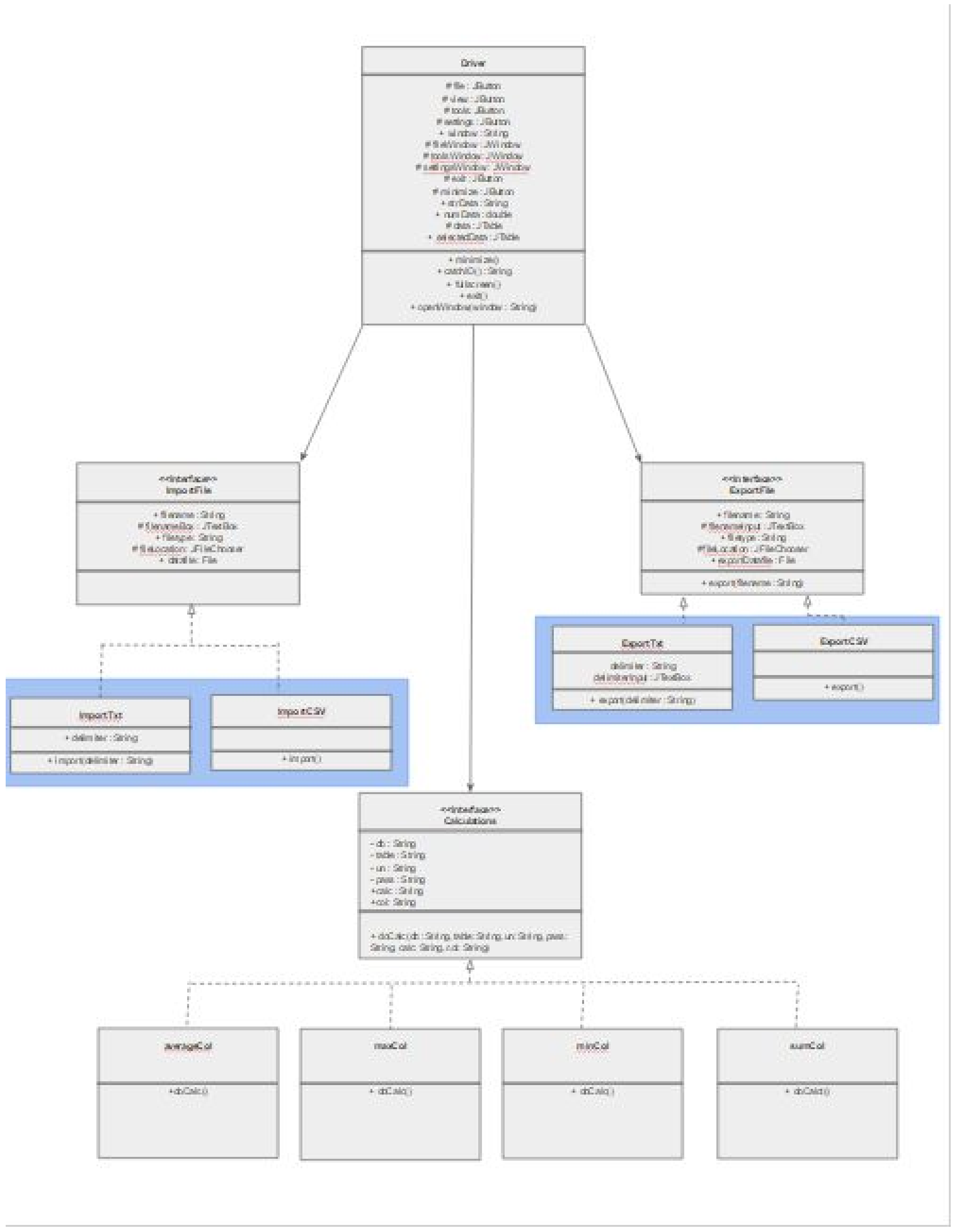
UR-09	As a user, I need to be able to pop-out the current table into a new window so I can view multiple tables at the same time	Functional	High
UR-10	As a user, I need to be able to rename data tables	Functional	High

3. **Show Part 2 diagram and final diagram and explain what changed:**

Part 2 Diagram:



Final Class Diagram:

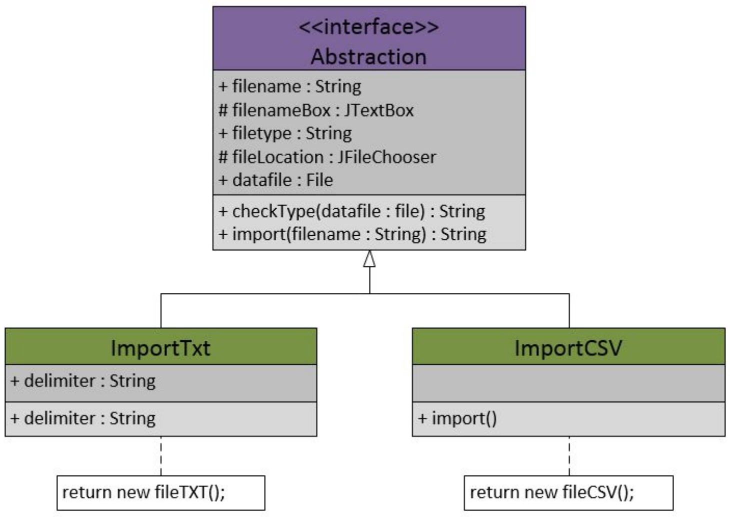


Our part 2 class diagram was incorrect, we focused too much on gui we never implemented, and did not make use of any design patterns. Our final diagram shows that we did not implement the gui (because it is optional and due to time constraints), and that we used both the Strategy and Factory Design Patterns. This proved extremely useful because we can have many different versions of an algorithm to select at runtime. We utilized these patterns for most of our interfaces of the program, which is great for scalability of our project. For example, we implemented it in our Import functions, if we needed or wanted to be able to export JSON files, then we can just add Add a new export class that implements ExportFileStream interface and add the appropriate function call in the driver.

4. We Implemented two design patterns primarily(Strategy and Factory). We use these design patterns in conjunction to make our program functional and scalable:

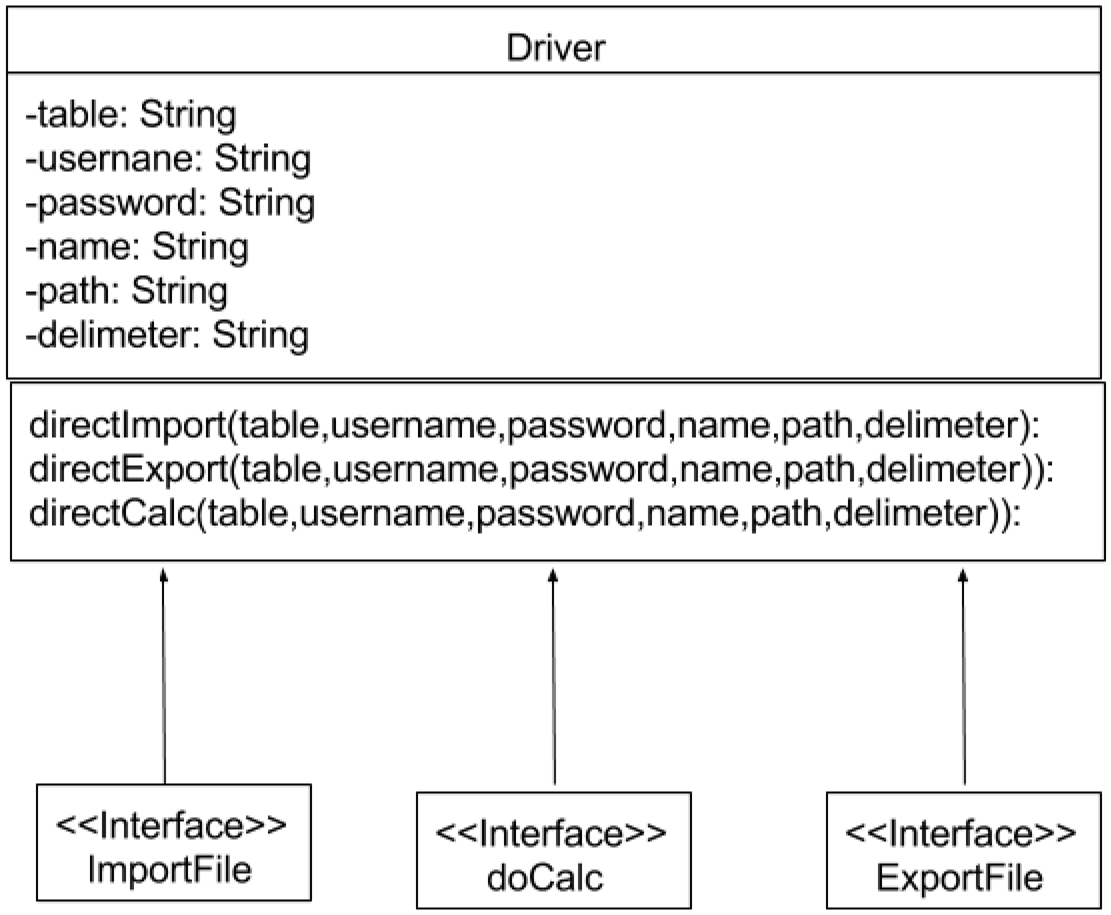
    1). Strategy

Strategy was necessary in our implementation, and our whole program revolves around its effective use. There is a general “Import,” “Export,” and “Calculation” strategy that we use for every instance of an import, export, or calculation we will always pass in the same values. This is represented by each functions respective interface. An importTXT(), for example, will implement the same interface as importCSV() and call the same function/parameters. This makes our program scalable in a large way and functional from an Object Oriented Standpoint. Specifically, strategy is implemented because both ImportCSV and ImportTXT operate using the same interface, and this holds true for all other interfaces we implement.



    2). Factory

Factory was essential in making our program dynamic. Based on input, we needed to have multiple possible behaviors that could be reached from one place, the driver. To implement factory, we used abstraction by making a “driver” object that is passed in as a parameter for every driver method. The driver catches the appropriate parts of the driver object and produces a desired, unique, behavior. Once again, this makes our program scalable to a large degree. If a new class were implemented that takes a new file type, there are very obvious and accessible points in the driver where you can trivially implement these new methods. The large amount of architecture we put down early on makes for much easier extension of the program. Specifically, Factory is implemented in driver when we parse strings to produce a desired behavior and call a specific package instance.


Driver
-table: String
-usernane: String
-password: String
-name: String
-path: String
-delimeter: String
directImport(table,username,password,name,path,delimeter):
directExport(table,username,password,name,path,delimeter)):
directCalc(table,username,password,name,path,delimeter)):
<<Interface>> ImportFile
<<Interface>> doCalc
<<Interface>> ExportFile

5. The main thing we have learned is the value of Object Oriented Design and scalability. After working on such a small program and seeing how much the design decisions we make affect the entire architecture, we realize that when working on larger projects in our field the choices we make are extremely important. In this sense, we used the DRY (Don't Repeat Yourself) technique, it saves time and coding expense.

A small change to a parameter type could lead to having to edit multiple files and methods, and that was in our small program alone. A change like that in a large data package could take hours, or days to implement. So once again, the choices you make are extremely important, and you have to be very careful when laying out the architecture of your program.

We also discovered scalability is extremely beneficial and abstraction is an invaluable tool. While designing our program, the saying “Closed for Modification, Open for Extension” really hit homeYou have to design towards scalability, or else your code will not be functional in the long term and you design towards scalability with abstraction. Also, designing towards scalability creates a very large ceiling for growth in your program. Because of the design patterns we used, if we implemented our program completely, it could become a very powerful tool to work with multiple data types, but the catch is we wouldn’t have to change the architecture of our program every time we wanted to implement a new data type. This makes new implementations trivial from an architectural standpoint. All you need to do is extend the correct interface and implement a new class.