

Final Report

Implementation of a Test Tool for Privacy Protection Algorithms

Author: Matthew Connop

Supervisor: Dr Jianhua Shao

Moderator: Prof. Stephen Hurley

CM0343 40 Credits

Cardiff School of Computer Science and Informatics

Cardiff University



May 2012

Abstract

Privacy protection is not an easy task in today's modern society, with the use of technology sending personal data from hospital records to bank details across the country new methods of securing it are needed every day. I hope by creating this test environment in which two algorithms can be measured, compared and then analysed, that it will lead to more secure privacy protection algorithms to be implemented.

The tool allows the user to compare and analyse two algorithms by using a set of metrics to compare them to each other, this will hopefully lead to better privacy protection standard. I used prototype construction to improve on the design as the project evolved in an agile approach.

The final tool created should be a good platform from to produce a valuable aid in helping researchers improve privacy protection.

Table of Contents

1 Introduction.....	1
2 Design	2
2.1 File Management	3
2.2 Algorithms	6
2.3 Measurement tools	7
2.4 Test tool GUI	9
3 Implementation	10
3.1 File Management	10
3.2 Run an Algorithm	14
3.4 Test tool GUI	19
3.4.1 Managing Datasets	19
3.4.2 Managing Results.....	22
3.4.3 Managing Algorithms	23
3.4.4 Comparison Tool	25
4 Results and Evaluation.....	28
5 Future Work.....	32
6 Conclusions.....	33
7 Reflection on Learning	34
Appendix A: Old Process Diagrams	35
Appendix B: Old GUI Design.....	39
Appendix C: Testing	40

Table of Figures

Figure 2.1 <i>Basic packages</i>	2
Figure 2.2 A dataflow diagram to describe the flow of information	6
Figure 2.3 A process diagram to show a possible flow for a measure.....	7
Figure 3.1 <i>Screenshot of the main window</i>	19
Figure 3.2 <i>Screenshot of the manage dataset window</i>	19
Figure 3.3 <i>Screen shot of NewDatasetPane.java</i>	20
Figure 3.4 <i>Screen shot of CSVEditor.java</i>	21
Figure 3.5 <i>Screen shot of NewResultPane.java</i>	22
Figure 3.6 <i>Screen shot of ManageAlgorithmPane.java</i>	23
Figure 3.7 <i>Screenshot of NewAlgorithmPane.java</i>	24
Figure 3.8 <i>Screenshot of NewAlgorithmPaneParameterPane.java</i>	24
Figure 3.9 <i>Screen shot of RunAlgorithmPane.java</i>	25
Figure 3.10 <i>Screen shot of ComparisonPane.java</i>	25
Figure 3.11 <i>Screen shot of DisplayComparisonResults.java</i>	26

Acknowledgments

I would like to thank Dr J Shao for taking his time to describe the problem in great detail and help me understand the very difficult aspects of the privacy protection world, also Dr G Loukides and Huong Ong for allowing me to use their algorithms and help in getting them to work.

1 Introduction

The goals of the project were to develop a java based test tool that can store datasets, run algorithms to produce output and then compare algorithm outputs using a set of metrics; this will assist in the research of privacy protection algorithms.

The approach taken for the final system has been one of design-implementation cycles, this proved a valuable approach as a solution was discovered and documented to allow a user to import algorithms.

The design while not all encompassing shows the approach to the project and how the system has been split into 'modules' to keep the code manageable. The implementation shows key ways in which the goals have been reached, even though some of these implementations may not be perfect.

The assumption this project works on, is that the system is more of a platform to show that a more robust system could be developed. The outcome being that this can be developed and cover a larger section of the wider problem, than just the problems in this projects scope.

2 Design

Some old process diagrams from the prototype design are included in Appendix A; this shows how the original system was one large process and will offer a comparison with the final design which has been changed into more manageable packages.

For the design of the final system the requirements have been split into 4 main components as seen in figure 2.1.

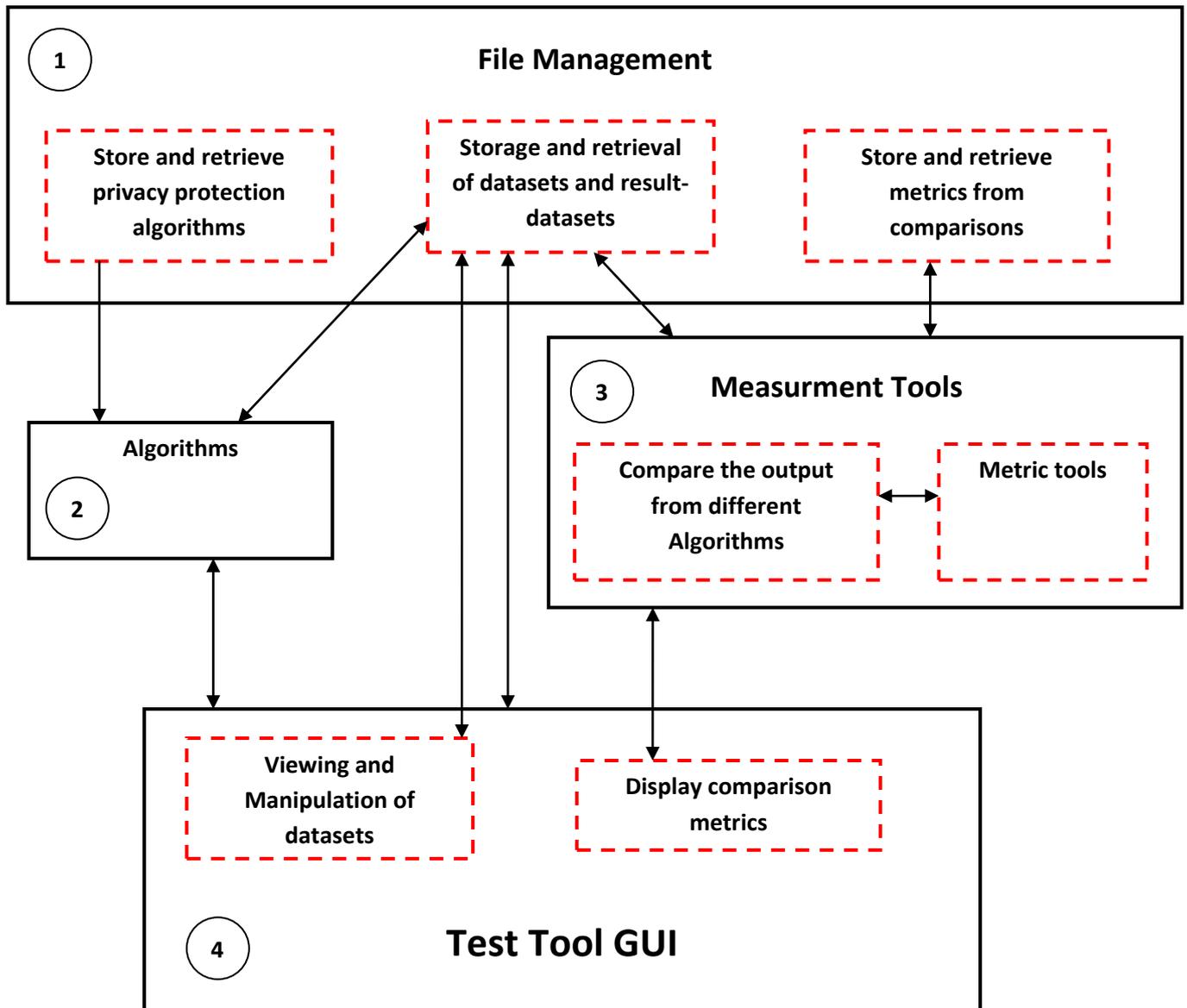


Figure 2.1 Basic packages

In order for the software package to be used as a standalone API (Application Programming Interface) and a test tool with a Graphical User Interface (GUI), the system has been divided into the following sub packages:

- file management

- measurement tools
- algorithms
- GUI

These packages will hold the relevant classes that will do the appropriate tasks. This design ensures that classes can be easily managed and added to for future improvements.

File Management package

This package requires that the system allows the user to add datasets (non-anonymised CSV files) and result-datasets (anonymised CSV files) to a repository that can then be retrieved and used for algorithms, manipulation or comparisons. It will also handle storing and retrieving algorithms for use in the GUI and in other applications; additionally the task of saving results from the measurement tests will also be managed in this package so that if a dataset has already been measured with a certain metric it can retrieve the result instead of running the measure again.

Measurement tools package

This package requires that the user can compare the outputted datasets from algorithms using a set of metrics of their choice.

Algorithm package

This package requires that algorithms can be configured then run to produce some result-dataset files, these result-dataset files can be used by the comparison tools and saved to the repository for future comparisons.

Test tool GUI package

This package requires that an interface is created for the user to interact visually with the tools available in the other packages and include a graphical tool for manipulation of datasets.

2.1 File Management

After implementing the prototype for the interim report it became clear that meta-data about any files imported into the system needed to be stored and easily retrieved. This meta-data would help form a standard way to describe the different files e.g. whether a dataset type is relational or transactional, the system would need this to understand how to process the data for additional tasks.

As mentioned in the initial report the types of files that the system will handle will be in CSV format, but to increase the scope of the system slightly the dataset meta-data should contain a delimiter value, this way space delimited files could also be used.

A lesson learned after creating the prototype was that it may be hard to find a working algorithm that can take CSV files and output anonymised CSV files. To overcome this

problem it was decided that anonymised CSV files (result-datasets) could be added directly instead of through an algorithm, with the assumption that the original dataset is available and linked to the result-dataset.

The following CRC cards describe the classes needed to accomplish the tasks this package has been given.

DatasetFileType	
Knows number (primary key) Knows name Knows dataset type Knows Delimiter Knows file location Knows analysis file number	DatasetFileManager AnalysisFileManager FileCounter

ResultFileType	
Knows number (primary key) Knows name Knows original file number Knows Delimiter Knows file location Knows analysis file number	ResultFileManager AnalysisFileManager FileCounter

AlgorithmFileType	
Knows number (primary key) Knows name Knows file location Knows dataset type it accepts Knows file location	AlgorithmFileManager AnalysisFileManager FileCounter

AnalysisFileType	
Knows number (primary key) Knows name Knows file location Knows original dataset	DatasetFileManager ResultFileManager AnalysisFileManager FileCounter

DatasetFileManager	
Keeps a list of DatasetFileTypes Writes list to file Reads list from file Retrieves DatasetFileType from list Removes DatasetFileType from list	DatasetFileType FileLocations

ResultFileManager	
Keeps a list of ResultFileTypes Writes list to file Reads list from file Retrieves ResultFileType from list Removes ResultFileType from list	ResultFileType FileLocations
AlgorithmFileManager	
Keeps a list of AlgorithmFileTypes Writes list to file Reads list from file Retrieves AlgorithmFileType from list Removes AlgorithmFileType from list	AlgorithmFileType FileLocations
AnalysisFileManager	
Keeps a list of AnalysisFileTypes Writes list to file Reads list from file Retrieves AnalysisFileType from list Removes AnalysisFileType from list	AnalysisFileType FileLocations
FileLocations	
Knows location where to save DatasetFileManager List Knows location where to save AlgorithmFileManager List Knows location where to save AnalysisFileManager List Knows location where to save ResultFileManager List Reads locations from a file	AnalysisFileManager DatasetFileManager AnalysisFileManager ResultFileManager
FileCounter	
Keeps track of primary keys Saves next primary key to use to file Reads next primary key to use from file	AnalysisFileType DatasetFileType AnalysisFileType ResultFileType

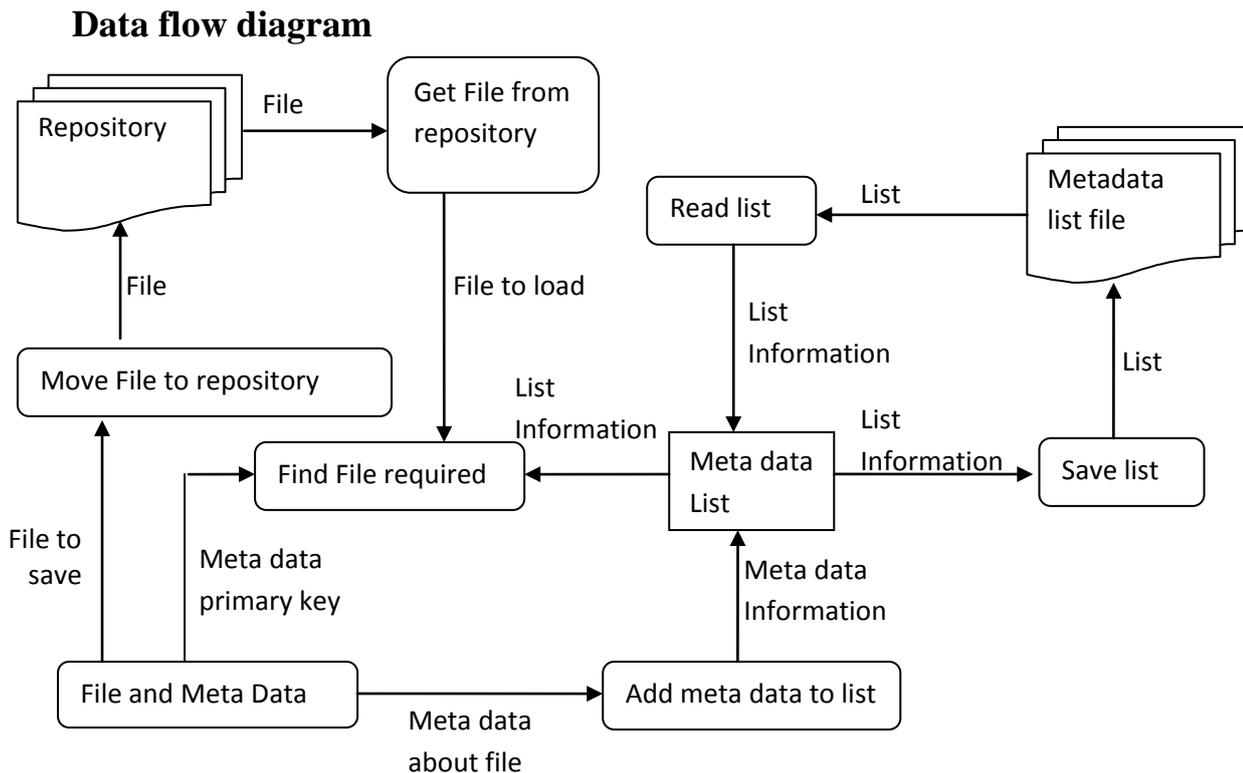


Figure 2.2 A dataflow diagram to describe the flow of information

Figure 2.2 shows how the data will flow for the managers that control the meta-data, this is an abstract diagram that describes the general behaviour of each manager.

The FileCounter and FileLocations are helper classes that will store information vital to the running of the system; each class will read a file then be able to inform the system of primary keys in the case of the FileCounter, and information about where the repository is on the computer system in the case of FileLocations.

By doing this it will make the package easier to distribute as the two files the system needs can be kept in a resources folder with the final executable jar file.

2.2 Algorithms

During the implementation of the prototype an algorithm was hard coded into the system, this was going to be the main way a user could run an algorithm, this seemed a very impractical way of managing algorithms but would have been enough to meet the requirements specified.

But during the implementation of the final system the idea of running a command line argument through java was approached. If the system could run a command line argument then it could run an algorithm that had been packed into an executable jar file.

To do this a class would need to be created that handles the command line arguments, with an added benefit of being able to measure the time it took to run. One of the requirements was to design a method to allow a user to add algorithms; this would not only satisfy that requirement but would additionally implement the design to see if is workable.

2.3 Measurement tools

The measurement tools package is to contain all the classes needed to compare and measure a dataset. As mentioned before this will only deal with actual files that have already been output from an algorithm, instead of running an algorithm and then measuring the output.

While I am unsure about the design of this, the constraints upon it are to create a class that takes a CSV file, measures it and returns the value for each different measure type.

This should be split up into a selection of classes that completes a single measure, a threaded class that runs each of these selected measures in turn and a class that can save the values that have been outputted by the measure classes to a file for later retrieval.

A possible process model would be

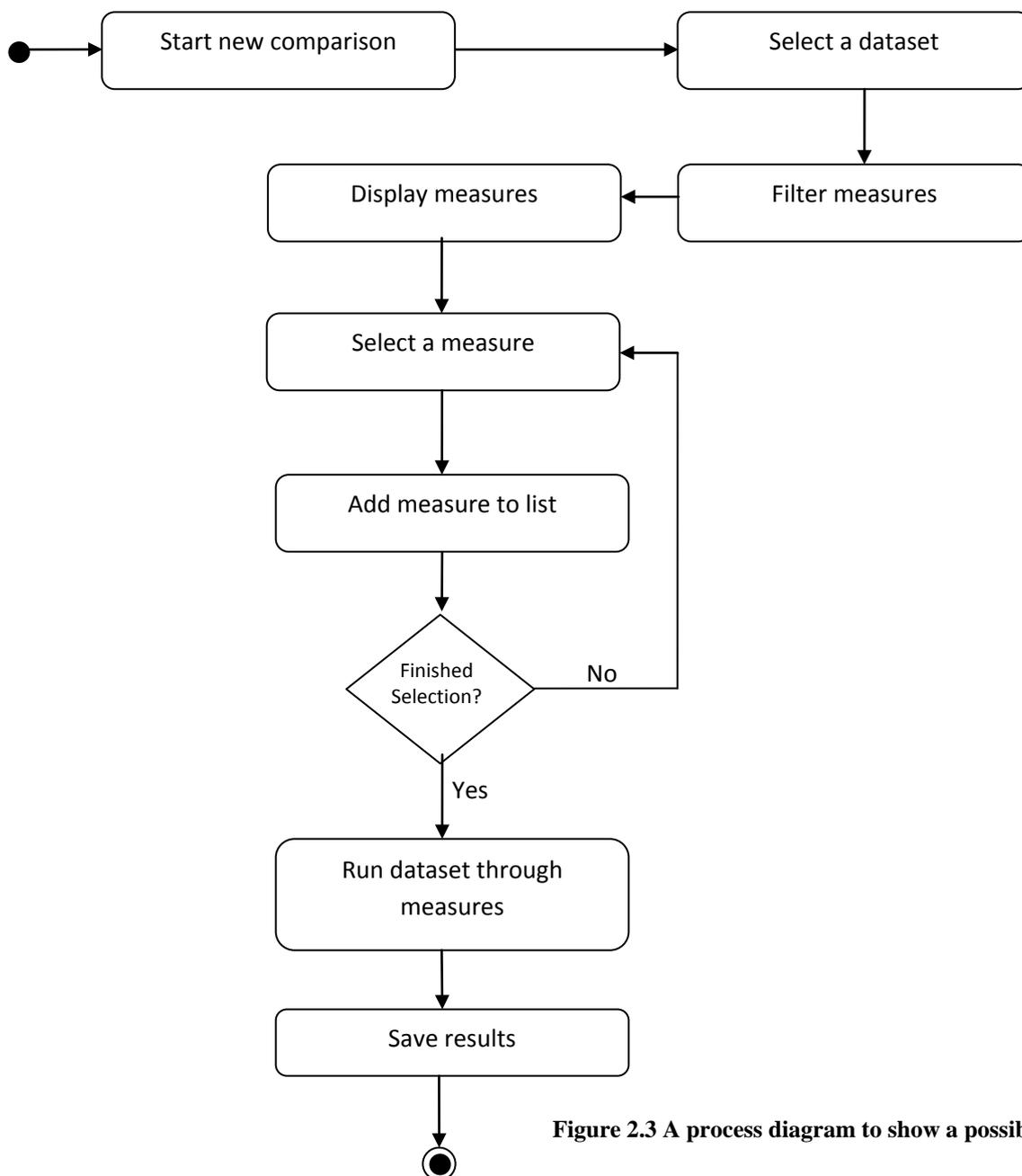


Figure 2.3 A process diagram to show a possible flow for a measure

The design only deals with measures that have been hard coded in the system, but for the requirements it was necessary to come up with another design that would give the system the ability to add new measurements.

Having implemented the final system, the following design was created towards the end of the project and could be implemented at a later date.

After coming up with one solution for the algorithm problem mentioned earlier, a similar system could be used for the measurements.

The following CRC cards show the classes needed to describe a measure and save its details they will be very similar in design to the already implemented classes in the source code and should be relatively simple to code.

MeasureFileManager	
Keeps a list of MeasureFileType Writes list to file Reads list from file Retrieves MeasureFileType from list Removes MeasureFileType from list	MeasureFileType FileLocations

MeasureFileType	
Knows number (primary key) Knows name Knows file location Knows dataset type it accepts Knows number of parameters	MeasureFileManager AnalysisFileManager FileCounter

These two classes can be added to the FileTools Package to deal with adding and removing measures, the system will then know how to run the executable jar which takes a result-dataset as its first argument, and any other arguments it may need i.e. original file, and could return a value using `System.out.println()`.

A threaded class can then be created that loops through the selected measurement jars and catches the `System.out.println()`, parses it to an Integer and saves it to a file.

2.4 Test tool GUI

A design was originally drawn to demonstrate the GUI side of the system this can be seen in Appendix B.

No official design was agreed upon before the final system was implemented but it should have a fairly simple interface so the goals can be accomplished. As discussed during one of the weekly meetings, when a user wants to accomplish a task a new window should open in which the task can then be completed.

Standard Javax Swing components will be used for the windows

- JFrame
- JPanel
- JButton
- JLabel

A form window will be created that allows the user to enter the meta-data using further components like:

- JComboBoxes – a drop down box
- JCheckBoxes - a check box
- JTextFields – a text field
- JFileChooser - a file chooser window
- JTextAreas – a text area

The classes in control of all the windows should be kept in organised packages to allow easier upgrades of the system.

3 Implementation

3.1 File Management

The file management package has two sub packages one for dataset tools and one for file tools.

Dataset Tools

This sub-package was created to manage the conversion of datasets, although the system only currently handles CSV files, any other classes added in the future that can parse other file formats can go here. This was an addition to the design as several other classes depended on converting a CSV.

CSVTools class

This class was created to read a CSV file and return it in a format the system can use for measuring and manipulating, I have chosen the `ArrayList` data type as it has faster access than an array and it will be easier to remove any elements when it comes to the manipulation of the CSV's.

The methods in this class use `au.com.bytecode.opencsv` API which when given a CSV file returns it as a `List<String[]>`, this package was chosen as it was easy to understand and use, but I had to put in a few lines of code to convert the `List<String[]>` which the API outputs to an `ArrayList<ArrayList<String>>` which the system uses.

While this method is not ideal and requires more computation time, the implementation of a better package will not take too much effort. Another problem that may occur is if large datasets are used, this would take up a lot of memory when read in, but for the implementation of this system it is adequate.

File Tools

This sub-package handles the different types of files that the system uses and how it retrieves and stores them.

FileLocations class

Any datasets, result-datasets, algorithms and comparison results that are stored, and their locations will need to be known for the other classes in the system.

To accomplish this, a data file was created which kept the locations of the required folders these are:

- Result folder - where all the output files from algorithms are placed

- Dataset folder - where the dataset files will be kept
- Analysis folder – where the comparison metric files are stored
- Algorithm Folder – where the algorithms will be stored

By designing the system this way the application when installed on another computer will allow the user to easily change the dedicated folder locations.

To implement this, a class called `FileLocations.java` was created that would read a file with the appropriate locations stored in them, and return the desired location when queried by any other class.

An XML file was chosen to store this data as it is user friendly to read and change if need be, it also would give me the opportunity to learn about reading and writing XML files in java.

This class uses the `DocumentBuilderFactory` interface in java to build and read an XML document, it then has static methods to return the file locations as a string.

During this I initially had problems understanding how relative locations worked when running an executable jar but after doing some reading the solution was to use:

```
MyClass.class.getProtectionDomain().getCodeSource().getLocation().getPath();
```

This allows the jar to find its current location and from there it can find the XML file containing the file locations.

FileCounter class

For the addition of datasets, result-dataset, algorithms and comparison results files that the system stores, each file has been assigned an individual number (primary key), by implementing this a user can retrieve any file by using its number, which is easier to remember than a full file name.

This class `FileCounter.java` simply reads an `ArrayList` that has been stored in a text file of the form [dataset Count, comparison result file count, algorithm count], its methods include saving the file by using the `org.json.simple.JSONValue` API to turn the array into a string and the `org.apache.commons.io.FileUtils` API to write the string to a file in the application data folder.

The method to read the file uses `org.codehaus.jackson` API, this API maps the values in the file back into an `ArrayList`, get methods then use this `ArrayList` to return the appropriate numbers.

```
private static ArrayList<Integer> read() throws IOException {
    ObjectMapper mapper = new ObjectMapper();
    ArrayList<Integer> temp = mapper.readValue(
        new File(fileName),
        new TypeReference<ArrayList<Integer>>() {
        });
    return temp;
}
```

The class uses considerably less code than programming a specialist parser hence the use of the API's, it also helped in understanding how to implement external API's, the issues with understanding how they work and which API's are appropriate to use.

FileType class

This was a change to the original design, as a dataset and result-dataset have common properties. This super class was created to hold those common values, the `DatasetFileType` and `ResultFileType` can then extend this.

In order keep track of the datasets in the repository (original datasets and result-datasets), a data structure was created which describes a dataset, and could be used to retrieve information about that dataset.

This class describes the dataset using:

- File Number – primary key.
- File Name – the name of the file.
- Delimiter – the delimiter used.
- Analysis File Number – the number of the file associated with the storing of the comparison result file.
- Description – a description of the dataset so the user can see what is contained in the dataset at a glance.

As this project only deals with CSV data file types I placed the delimiter value in this super class, if any other types of datasets were to be implemented it may be better to have the delimiter variable in a sub class for CSV-datasets.

The constructors for this class are protected so it can only be constructed with files in the same package and will prevent a user of the API from creating the super-type.

DatasetFileType and ResultFileType class

These are the sub classes of the super `FileType` class, the `DatasetFileType` has an additional variable of `DatasetType`, this stores the type of the dataset i.e. relational or transactional, the system only handles these two but more could be added later.

The `ResultFileType` has an additional variable of `OriginalDatasetNumber`, this is so any result-dataset files that are added to the repository have a link to the original dataset it was created from for use in the comparison tests.

Each class has a public constructor so a user can create a new file with the file name, description, delimiter used and the file type or original file number as parameters. It will use the file counter class described earlier to set the file number and sets the analysis file number to -1 to signify it has not been measured yet. A protected constructor is also available so the

system can use this for building the list of files in the repository, but has extra parameters of file number and analysis file number.

Each type class has a method `getFilepath()` to return the file location using the `FileLocation` class described earlier.

AlgorithmFileType class

This data structure describes an algorithm that a user would like to add to the repository using:

- Algorithm Number – primary key
- Algorithm File Name – the name of the algorithm file
- Description – a short description of the algorithm
- Input File Type – the type of CSV it can accept (relational or transactional)
- Number Of Arguments – number of parameters it can take
- Argument Type And Description – A 2-dimensional array that stores the argument type i.e. string, integer and a description i.e. input file, k-value.

This also has public and protected constructor methods similar to those described above and appropriate get methods.

It keeps a list of the parameter types and arguments so a user can find out what arguments the algorithm will require to run.

AnalysisFileType class

This data structure describes a comparison result file (Analysis file) that the user would like to add to the repository using:

- File number – primary key
- File name – file name
- Original file number – the original file that was used to create these metrics

This also has public and protected constructor methods similar to those described above and appropriate get methods.

DatasetFileManagement class

The system was required to allow a user to add datasets to a repository for later use, in order for the system to know what datasets are available it was necessary to keep some kind of list of these datasets and move any selected datasets to a folder where they will be stored.

For this class the XML format was chosen again to store the list, this gave me experience parsing a more complicated XML file. That way a user could simply add a file to the repository manually (drag and drop) and they could then update the XML file, though this is not recommended, as an error in the file will make it unreadable.

The format of the XML file is:

```
<DataSetList>
  <DataSet>
    <DataSetID ID="37">
      <DatasetName>data.csv</DatasetName>
      <Delimiter>,</Delimiter>
      <Description>test data set</Description>
      <FileType>Transactional</FileType>
      <AnalysisFileNumber Num="-1"/>
    </DataSetID>
  </DataSet>
</DataSetList>
```

The methods implemented in the class include reading and writing the file using the java tool `DocumentBuilderFactory`, like before both are private access so only this class can perform these tasks.

The other methods implemented include:

`addfileToList()`, which allow a user to add a dataset to the repository with two parameters, the file to be moved and the `DatasetFileType`.

`addArrayToList()` to add an `ArrayList<ArrayList<String>>` to the repository using the `CSVTools` class to convert it to a CSV before adding it.

`getFile()` for retrieving a file using the file number or name to locate it.

`removeFileFromList()` for removing a file from the list.

When parsing the more complicated file difficulties arose with the reading and writing of the XML file, some of the other problems that came up were possible file write problems. If different threads in the GUI tried to write to the file a correct document was not produced. To overcome this a lock has been added to the file write that has to be acquired before writing and if the write lock is locked then any thread wanting to read or write to the file will have to wait.

AlgorithmFileManager class, AnalysisFileManager class and ResultFileManager class

These three classes handle reading and writing the lists that keep track of their relative files that have been added to the repository. They are all similar in design to the `DatasetFileManager`.

3.2 Run an Algorithm

During the implementation of the prototype, it became clear that it would not be easy to import a new algorithm that the system could use and they may have to be hard coded individually. This was unsatisfactory but would have been enough to fulfil the requirements.

During the implementation of the system the problem arose again as it was an important part, to solve the issue of algorithms being added to the system and run by the user without any

real knowledge of how that algorithm worked, it became necessary to have some kind of standard.

As it is possible to run an executable file from within a java application, the algorithms will have to be imported in the form of an executable file (jar or exe). The standard decided upon was, an input file as argument 0, an output file as argument 1 and then any further arguments needed.

An `AlgorithmTools` package was created to hold the class needed, if the system is then built upon, this package would then be concerned with the running of algorithms. It may be necessary to add new classes if future algorithms require it.

Within this package `RunAlgorithm.java` is a class that implements `Runnable`, so if it is a long running algorithm it will not block the main thread it is run from. Its constructor takes a string array of arguments (in the standard described above), an `AlgorithmFileType` and a `ResultFileType` as its parameters.

When the thread is run the method `setCommandLineString()` then builds a command line string starting with:

```
java -jar "c:/algorithmlocation/algorithmname.jar"
```

The algorithm location can be obtained from the `AlgorithmFileType`, it then adds the arguments that have been passed to the constructor to the end of the command line string.

This command is then passed to a `Process` using:

```
Process pr = Runtime.getRuntime().exec(command);
```

The time it took the process to run can be measured and added to the metrics associated with the `ResultFileType` that has been passed to the constructor.

This class at the moment only can handle executable jar files but it will be possible to extend this to other executable files as long as the executable algorithms keep to the standard of input file as first argument and output file as second argument.

3.3 Comparison tools

The requirements for the system included the ability to measure the result-datasets output from algorithms and compare these measures. A `MeasurementTools` package was created to hold all the classes that deal with these tasks.

Hard coding the measures into the system was the easiest way to do this, although it makes it difficult to create more measures, the only way to add a measure would be to hard code them into the source code and recompile, but for the basic requirements it would be adequate.

Firstly a few measures would be needed, one had already been included in the run algorithm package and that was 'Time To Run', the system needed to know what measures are available so a class that holds these list of measures was created.

This class contains Array lists of measures and the names of the available measures in the form of

```
public static final String UTILITYMEASURE = "Utility Measure";  
public static final String TIMETORUN = "Time To Run";
```

This was done so standard names are used throughout the system; they are then simply added to an `ArrayList` of different types of measures.

"Time To Run" would come under the list of general measures as it can be used for relational and transactional datasets that the system handles.

"Utility Measure" would come under the list of relational measures as it requires a relational dataset.

A list for transactional measures has also been implemented although no class has been written that can measure transactional datasets, these lists can be retrieved using the appropriate get methods.

`MeasurementInformation.java` is the class that saves and loads the measures produced, it contains a `HashMap<String, Float>`. `HashMap` is the java hash map class, it can use the measure name as a key and `Float` as the value of that measure, once all the measures have been added to the `HashMap` using the `add` method, the `HashMap` can be written to a file using the `saveFile()` method and a `HashMap` can also be read from a file using the `readFile()` method, both methods use the JSON parser as described in the `FileCounter` class.

To measure a dataset the `UtilityMeasure` class was created, this measure accepts a dataset in the form of `ArrayList<ArrayList<String>>` in its constructor, which the `CSVTools` class can provide, and the list of QID's (columns that contain the quasi-identifiers).

The utility measure is a measure that for each QID row in a dataset 1 is multiplied by the number of other rows with the same values and are added together to form a total, that is the same as counting the number of items in a group and multiplying by itself.

The class runs though the `ArrayList` selecting the appropriate columns and adding them to a new `ArrayList` that contains just the selected QID's, this will make it easier to count the number of similar entries. It then takes each row of the `ArrayList` containing just the QID values and adds them to a `HashMap<ArrayList<String>, Integer>` the rows can be used as keys in the `HashMap` and then every time it comes across that key the value is incremented.

This can be seen in the following code segment:

```
// method to count the number of times a row appears in a dataset
private static HashMap<ArrayList<String>, Integer>
    getQIDCount(ArrayList<ArrayList<String>> inDataset, int[] QIDs) {
    // creat an new hashmap
    HashMap<ArrayList<String>, Integer> wordCount = new HashMap<>();
    // get an ArrayList that just contains the QID values
    ArrayList<ArrayList<String>> dataset = getQIDsAsArrayList(inDataset,
                                                            QIDs);
    // llop through the arraylist and count the number of duplicate entries
    for (ArrayList<String> row : dataset) {
        Integer count = wordCount.get(row);
        wordCount.put(row, (count == null) ? 1 : count + 1);
    }
    // return the HashMap
    return wordCount;
}
```

The class has then a public static method that will use the above action to return the utility value, this method is static so it can be used by the system and through the API.

```
public static float returnUtilityMeasure(
    ArrayList<ArrayList<String>> dataset,
    int[] QIDs) {
    float temp = 0;
    // get the HashMap containing the count of entries
    HashMap<ArrayList<String>, Integer> tempMap = getQIDCount(dataset, QIDs);
    // loop through the HashMap and calculate the measure
    for (ArrayList<String> key : tempMap.keySet()) {
        temp += tempMap.get(key)*tempMap.get(key);
    }
    return temp;
}
```

Once this measure had been implemented it was necessary to then compare two datasets using this measure.

A class `CalculateComparisonMeasures.java` was created to handle running the measures; this constructor takes a result-dataset number, the original dataset number, an `ArrayList` of Measures, and QID numbers as parameters.

This class will then look and see if the datasets have been measured before, if they have it will not run the measure it will just return the value from the saved measurement information file, as long as the measures use the same QIDs as before

The `CalculateComparisonMeasures.java` starts a thread which goes through the list of measurements it has been set with the use of a switch case (a type of java if statement). It will see if it recognises that measure, if it does it will run the code required to do that measure. An example of the switch case for the Utility Measure:

```

// if utility measure is in the list
case ListOfMeasures.UTILITYMEASURE:
    // if the measure information has not been measured for the result-
    dataset
    if (!resultInfo.hasMeasure(currentMeasure)) {
        // add the measure name to the information file and the run the
        // utility measure on the dataset to get the float value
        resultInfo.addMeasure(currentMeasure,

            UtilityMeasure.returnUtilityMeasure(CSVTools.getCSVAsArray(
                resultFile.getFilePath(),
                resultFile.getDelimiter()),
                QIDs));

        // else if the measure has already been recorded but the QIDs selected are
        // different set the new value
    } else if (resultInfo.hasMeasure(currentMeasure) &&
        !checkSameQIDsSelected(resultInfo,QIDs) ) {
        resultInfo.setMeasure(currentMeasure,
            UtilityMeasure.returnUtilityMeasure(CSVTools.getCSVAsArray(
                resultFile.getFilePath(),
                resultFile.getDelimiter()),
                QIDs));
    }
    // if the measure information has not been measured for the original dataset
    if (!originalInfo.hasMeasure(currentMeasure)) {
        // add the measure name to the information file and the run the utility
        // measure on the dataset to get the
        // float value
        originalInfo.addMeasure(currentMeasure,
            UtilityMeasure.returnUtilityMeasure(CSVTools.getCSVAsArray(
                originalFile.getFilePath(),
                originalFile.getDelimiter()),
                QIDs));

        // else if the measure has already been recorded but the QIDs selected are
        // different set the new value
    } else if (originalInfo.hasMeasure(currentMeasure) &&
        !checkSameQIDsSelected(originalInfo,QIDs) ) {
        originalInfo.setMeasure(currentMeasure,
            UtilityMeasure.returnUtilityMeasure(CSVTools.getCSVAsArray(
                originalFile.getFilePath(),
                originalFile.getDelimiter()),
                QIDs));
    }
;
break;

```

While implementing this into the system a threading problem occurred. In the GUI after the comparisons have been run it is necessary to show the results of those comparisons, the display pane that has been implemented has to wait for these comparison threads to finish before the results can be displayed. The first attempt at this had the display panel trying to read the results before the comparison threads had finished.

To solve this problem a `CountDownLatch` was used, a `CountDownLatch` is a java method to that counts the number of threads that have called a countdown and when it reaches zero any thread waiting for the countdown to reach zero may continue

The updated constructor for this method now requires a `CountDownLatch` in the parameters and when the thread has finished its measures, it calls `threadCounter.countDown()` ;

3.4 Test tool GUI

For the graphical user interface side of the system most of the implementation is fairly standard java Swing components throughout, as most of this is basic java the full classes are available in the GUI package, which can be seen in the source code. The screenshots will show the different elements that have been added.

A fairly simple interface was created as the entry point into the application.

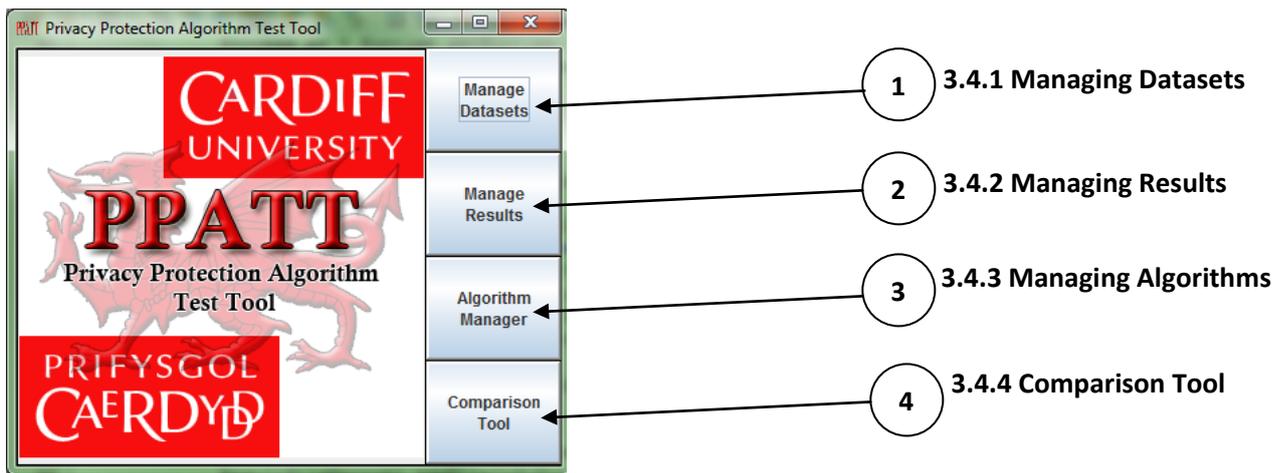


Figure 3.1 Screenshot of the main window

Figure 3.1 shows the MainWindow.java output, this is a simple JFrame with four JButtonS added that allow the user to manage the various goals that need to be achieved.

3.4.1 Managing Datasets

Button ① when pressed opens up a new JFrame as seen in Figure 3.2 this is the manage dataset panel.

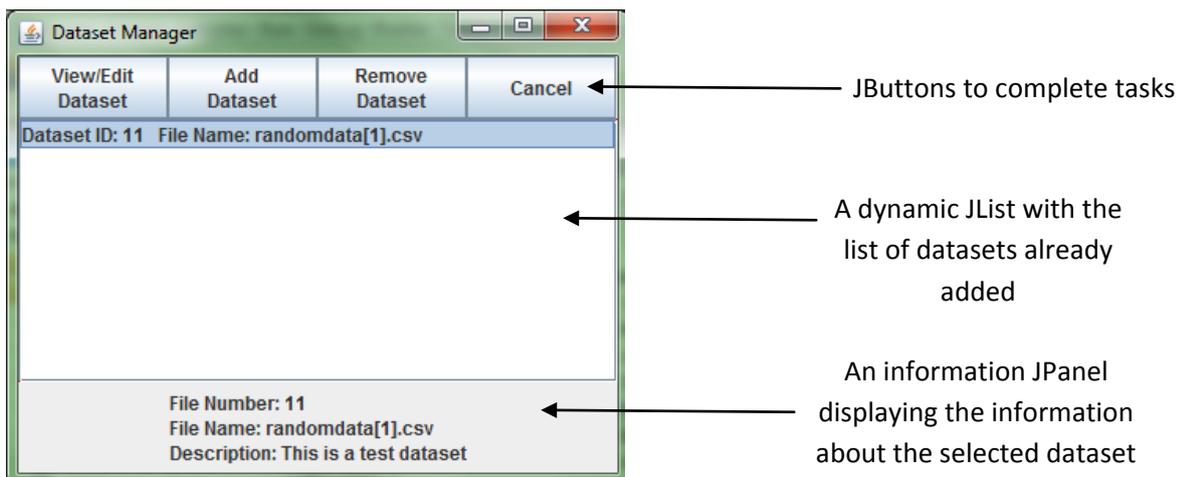


Figure 3.2 Screenshot of the manage dataset window

This JFrame in figure 3.2 contains a dynamic JList which updates when datasets have been added or removed, an information panel, which is a simple implementation of a

JPanel with the ability to set text to the label (InformationBar.java) allowing the user to see basic information about the dataset at a glance and a selection of buttons to complete the various tasks that the system is required to do.

NewDatasetPane.java

When the Add Dataset button is pressed this opens a form as seen in figure 3.3.

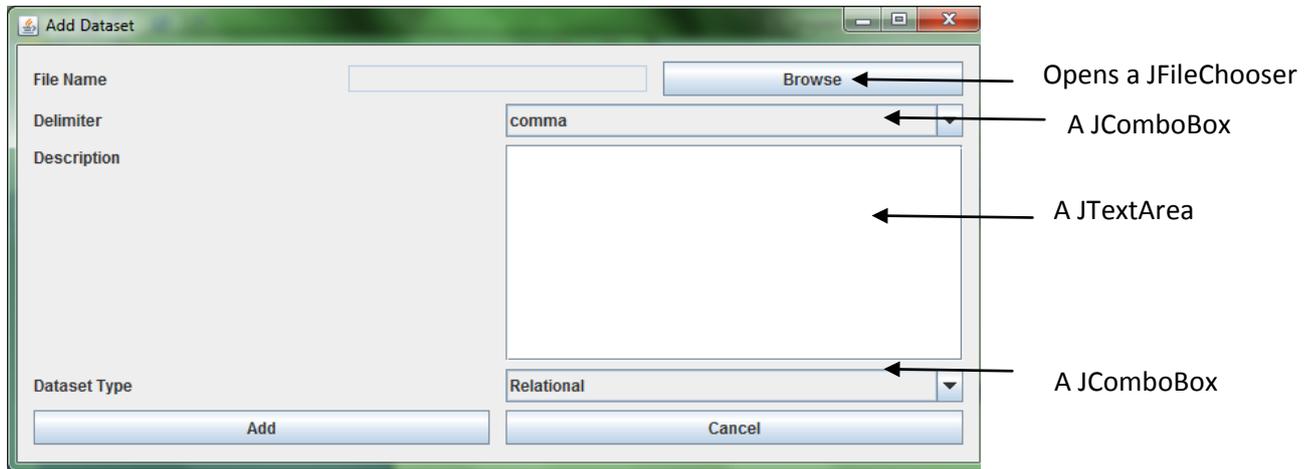


Figure 3.3 Screen shot of NewDatasetPane.java

For the layout of the form entry, the `net.java.dev.designgridlayoutAPI` has been implemented, initially a `GridLayout` (a standard java layout manager) was used but it was difficult to ensure the components added were laid out neatly. After some research this API was decided upon as it is easy to implement and produced neat looking forms. The instantiation operation is:

```
DesignGridLayout layout = new DesignGridLayout(yourJPanel);
```

And to add a component to the layout you can use

```
layout.row().grid().add(new JLabel()).add(component1);
layout.row().grid().add(new JLabel()).add(component2);
```

These two lines add components row by row and the API takes care of the layout of the form.

The user can select a file from their computer system using the browse button and then they simply have to fill out the rest of the data and press the add button. Upon pressing the add button a new `DatasetFileType` is created with the information entered and it is added to the list of datasets and repository using the `DatasetFileManager` class which handles the rest.

```
DatasetFileType newFile = new DatasetFileType(inFileName.getText(),
                                             delimiter,
                                             description.getText(),
```

```
fileType);
```

```
DatasetFileManager.addFileToList(file, newFile);
```

To remove a dataset from the list, use the Remove Dataset button which uses the following code to remove it when a file has been selected.

```
DatasetFileManager.removeFileFromList(fileNumber);
```

CSVEditor.java

Finally to do some manipulation of the datasets which is also a requirement, the View/Edit button runs the class `CSVEditor` as seen in figure 3.4.

To create an editor for CSV files basically meant implementing a spreadsheet where rows or columns could be removed. In the interim report it was discussed about using Open Office as the spread sheet editor, after reviewing the API for open office its complexity quickly made it a major task to implement.

After doing some research on open source packages that can manage this task, a far simpler package was selected and used. This package `quicktable` allows a `Vector<Vector<String>>` to be set as a `JTable` (a java Swing component to mimic a table) within a `JPanel`, this package is quite old and so still uses `Vectors`, this will have to be dealt with at a later date.

In order to use this some helper methods have been written to convert the `ArrayList` used by the system into a `Vector`, once this is completed it can be loaded with the data in an array using:

```
DBTable dBTable1 = new DBTable();
dBTable1.refresh(array);
```

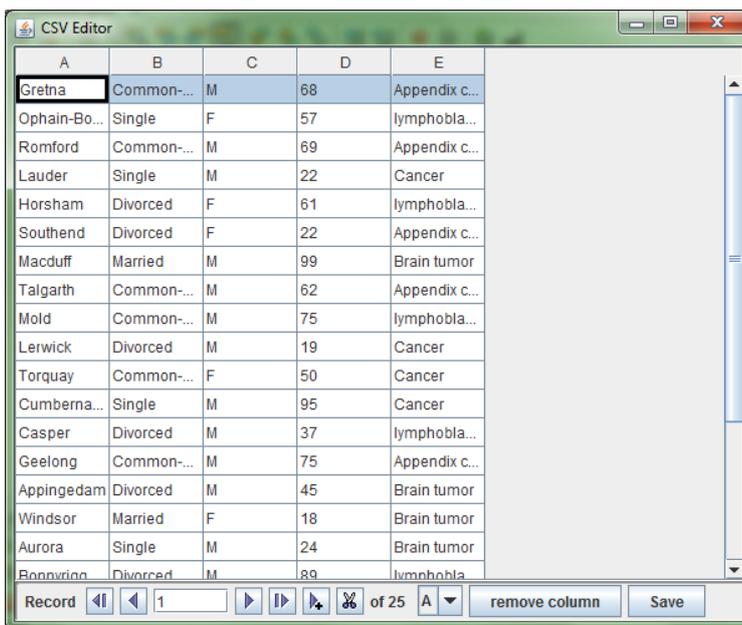


Figure 3.4 Screen shot of `CSVEditor.java`

One problem that cropped up is when using any transactional data to load into the table, it did not accept uneven arrays. A relational dataset is a square table with every tuple the same length, but transactional datasets may have differing tuple length. To overcome this another

helper method was used which made a transactional dataset ‘square’ by adding empty strings to the end of the short tuples, and removing them before saving. While not an ideal method as it uses more computation and if larger datasets are used it may seem slow, it is effective for now.

Another component of the DBTable is the control panel, this panel can be seen at the bottom of the screenshot in figure 3.4, it is a built in control panel that can remove rows. The DBTable control panel did not however have a control for removing columns.

The solution to this was to use a feature the control panel had to add more components, so a drop down box was added to select a column and a JButton which removes the selected column when pressed. This column removing feature is removed when transactional datasets are loaded as they have no columns as such.

Also added to the control panel was a save button, which opens up the new dataset form (fig 3.3) as seen earlier.

3.4.2 Managing Results

To manage the results that have been outputted from an algorithm, a very similar method to the manage dataset panel was implemented as seen in figure 3.2. The only differences being that when viewing the result-datasets in the CSVEditor the control panel has been removed so no entries can be changed.

The other difference is when adding a new result-dataset; this also opens up a new form using the same methods implemented in the NewDatasetPane.java.

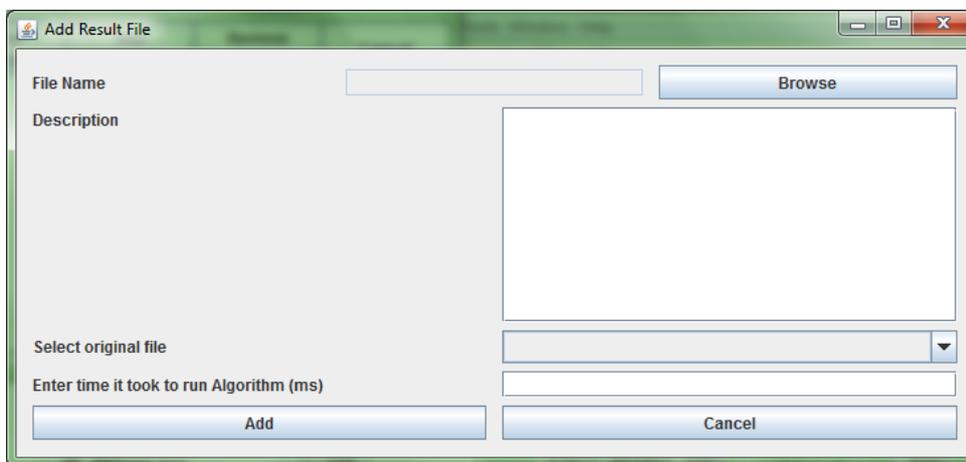


Figure 3.5 Screen shot of NewResultPane.java

This was required so a user could add a result-dataset to the repository after manually running an algorithm, some basic data has to be filled in, the original dataset that was used has to be selected and the time it took to run so it can be compared to other result-datasets. Once this has been filled in the add button uses the following commands to add it to the repository.

```
ResultFileType newFile = new ResultFileType(originalFileNum,
```

```

        inFileName.getText(),
        delimiter,
        description.getText());
ResultFileManager.addFileToList(file, newFile);

```

A new analysis file is also created to store the new “Time To Run” metric the user has entered with the following lines of code.

```

// create a new measurement info data structure
MeasurementInformation info = new MeasurementInformation();
// add the measure Time To Run to the info data structure
info.addMeasure(ListOfMeasures.TIMETORUN, new Float(timeToRun.getText()));
//create a filename for where the measures will be stored
String filename = FilenameUtils.removeExtension(newFile.getFileName()) +
        "analysis.txt";
// create the meta data data structure that will be added to the list
AnalysisFileType infoFile =
        new AnalysisFileType(filename, newFile.getFileNum());
// set the result-file analysis number so the file can be found for future
// use
newFile.setAnalysisFileNumber(infoFile.getFileNum());
// save the info data structure file
info.saveFile(infoFile.getFilePath());
// add the meta data associated with info data structure to the list
AnalysisFileManager.addFileToList(infoFile);

```

3.4.3 Managing Algorithms

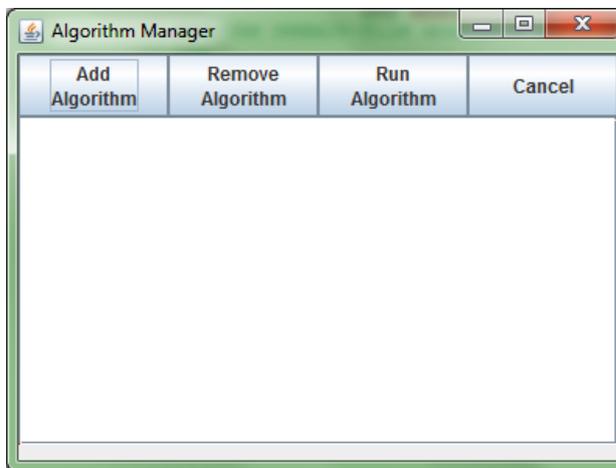


Figure 3.6 Screen shot of *ManageAlgorithmPane.java*

Figure 3.6 shows a screen shot of the panel that will appear when button ③ is pressed; this is similar in design to the previous panels used to manage datasets and algorithms. The main difference here is the Run Button, this allows the user to run an algorithm that has been imported into the system.

Firstly to add a new algorithm, a new form was created using the same methods as *NewDatasetPane.java*, this pane has all the form entry components needed to add a new algorithm to the system.

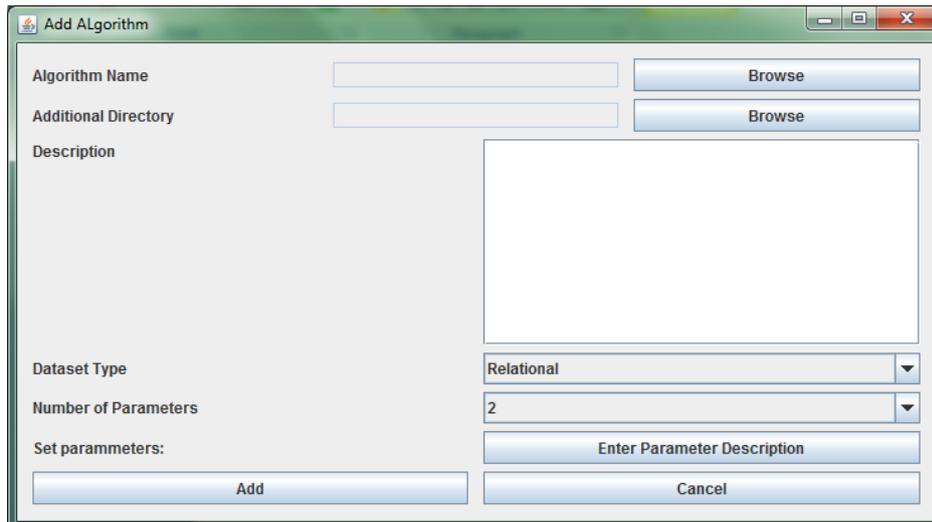


Figure 3.7 Screenshot of *NewAlgorithmPane.java*



Figure 3.8 Screenshot of *NewAlgorithmPaneParameterPane.java*

Figure 3.7 shows the entry form for a new algorithm. Figure 3.8 shows the panel where the parameter descriptions have to be entered, as you can see the first two parameters entered are set to the standard of input and output file.

Figure 3.9 shows the panel that deals with running an algorithm, continuing to use a simple form design with familiar Swing components. Once the user has entered the data the run button uses the following command to run the thread.

```
RunAlgorithm runningThread = new RunAlgorithm(getParamValues(),
                                             algorithm,
                                             resultFile);
SwingUtilities.invokeLater(runningThread);
```

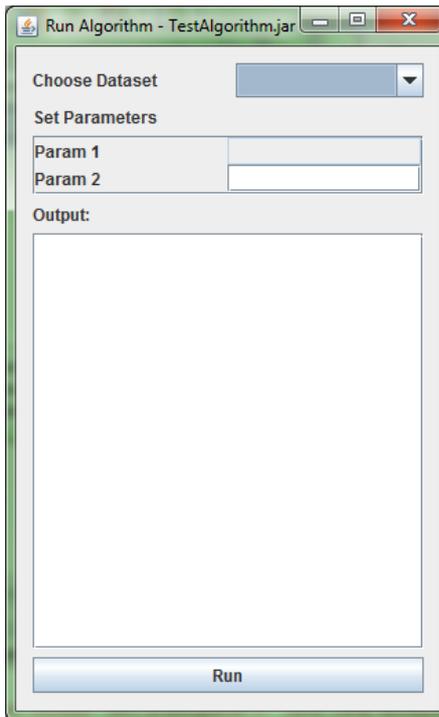


Figure 3.9 Screen shot of RunAlgorithmPane.java

3.4.4 Comparison Tool

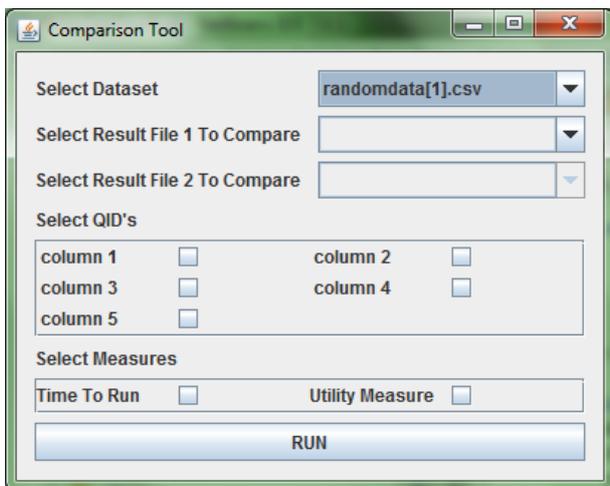
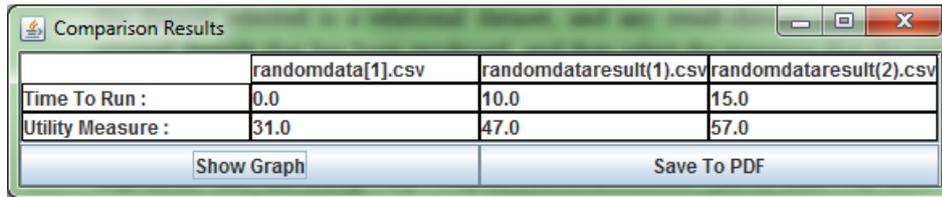


Figure 3.10 Screen shot of ComparisonPane.java

For the ability to run comparisons from the GUI another simple form was used to gather the data required, as seen in figure 3.10. This form has the ability to select the QIDs to measure (if the dataset selected is a relational dataset), any result-dataset file associated with the original dataset and measures that the user would like to compare.

After the comparison has run the system required that the results are shown to the user, to implement this, the DisplayComparisonResults.java class was created, this class creates a new JFrame which displays the results in a table format as shown in figure 3.11.



	randomdata[1].csv	randomdataresult(1).csv	randomdataresult(2).csv
Time To Run :	0.0	10.0	15.0
Utility Measure :	31.0	47.0	57.0

Buttons: Show Graph, Save To PDF

Figure 3.11 Screen shot of `DisplayComparisonResults.java`

Its constructor takes the parameters of the measurement information files from the original and two results files that are being compared, the list of measures that have been chosen and the QID's selected.

Initially when first implementing this, the layout on the central `JPanel` did not format the results very well. This problem was overcome with a `JPanel` component as it can display HTML, so the class loops through the results that have been saved in the measurement info files, adds them to a HTML table and writes it to the `JPanel`, as can be seen in figure 3.11.

An addition to the system was the ability to save the results to a PDF, to accomplish this task the `com.itextpdf` API has been used. This API seemed very suitable as it can accept the HTML string that has been used to write to the `JPanel` and convert it to a PDF with the following commands:

```
// create a new document
Document document = new Document();
// create a file from the JFileChooser
File file = fc.getSelectedFile();
// get the instance of PDFWriter
PdfWriter.getInstance(document, new FileOutputStream(file));
// open the document
document.open();
// create a list from the HTML string
List htmlarraylist = HTMLWorker.parseToList(
    new StringReader(infoHTMLString.toString()),
    null);

//add the collection to the document
for (int k = 0; k < htmlarraylist.size(); k++) {
    document.add((Element) htmlarraylist.get(k));
}
```

As mentioned earlier this class is where the threading problem occurred and the solution was a `CountDownLatch`. The code below shows the `CountDownLatch` and how it is used.

```
// create a new countdown latch
CountDownLatch waitForThreadLatch = new CountDownLatch(2);
// construct a new comparison measure thread for result-dataset 1
CalculateComparisonMeasures calculate = new CalculateComparisonMeasures(
    resultFile1, originalDatasetNumber, measures, QIDs, waitForThreadLatch);
// construct a new comparison measure thread for result-dataset 2
CalculateComparisonMeasures calculate2 = new CalculateComparisonMeasures(
    resultFile2, originalDatasetNumber, measures, QIDs, waitForThreadLatch);
// construct a new display pane thread
DisplayComparisonResults display = new DisplayComparisonResults(
```

```
        originalDatasetNumber, resultFile1, resultFile2, measures, QIDs);
try {
    // run comparison thread 1
    SwingUtilities.invokeLater(calculate);
    // run comparison thread 2
    SwingUtilities.invokeLater(calculate2);
    // wait for those threads to countdown the latch to zero
    waitForThreadLatch.await();
    // then display results
    SwingUtilities.invokeLater(display);
}
```

Figure 3.11 also shows a button Display Graph, this would have displayed the results as a graph. This would be a nice addition to the system but unfortunately there was not enough time to implement this.

4 Results and Evaluation

During the implementation of the final system continuous tests were performed to try and find as many bugs as possible.

After a section had been completed checks were performed to see if it functioned correctly. During the GUI development after a component had been implemented like the new dataset form for example, the system would be run and then checks were made to see if the file manager had written properly to the XML file and the dataset was moved to the correct folder.

For the purposes of the formal main testing cases below, the requirements from the interim report have been numbered as follows:-

1. The system shall allow a user to create a new test.
2. The system shall allow a user to choose from a selection of datasets.
3. The system shall allow a user to choose from a selection of algorithms.
4. The system shall allow a user to review the results according to a selection of measurements.
5. The system shall allow a user to specify algorithm configuration.
6. The system shall allow manipulation of datasets.
7. The system shall allow a user to import more datasets.
8. The system shall make any imported datasets available for future tests.
9. The system shall produce a set of metrics from the test.
10. The system design shall implement a method for admin to import new algorithms for testing.
11. The system design shall implement a method for admin to import new measurements to review results.
12. The system should contain a suitable API for use in other systems.

Test case ID: 001			
Test Purpose: To see if a dataset can be added to the system through the GUI to Fulfil requirement 7 and 8			
Environment: Windows 7 JDK 1.7.0_01-b08			
Pre-conditions: none			
Test Case Steps:			
Step Number	Procedure	Expected Results	P/F
1	Start Application	Main window appears	P
2	Select dataset manager button	Dataset manager window appears	P
3	Select Add Dataset button	New dataset form window appears	P
4	Fill out the meta data	Form entry components can be filled properly	P
5	Press the add button	The file is moved to the correct folder and the XML file is populated correctly	P
6	Return to dataset manager window	The new dataset should appear	P
Comments:			

Test case ID: 002			
Test Purpose: To see if a dataset can be chosen through the GUI to Fulfil requirement 2 and 6			
Environment: Windows 7 JDK 1.7.0_01-b08			
Pre-conditions: none			
Test Case Steps:			
Step Number	Procedure	Expected Results	P/F
1	Start Application	Main window appears	P
2	Select dataset manager button	Dataset manager window appears	P
3	Select A Dataset from the list	A dataset is high lighted	P
4	Select the View/Edit Button	The CSVEditor window should appear with the data loaded	P
5	Edit the data using the buttons made available	The appropriate rows and columns should be removed	P
6	Press the save button	A new dataset window should appear	P
7	Press the add button	The adjusted dataset should be added to repository and XML file updated properly	P
Comments:			

Test case ID: 003			
Test Purpose: To see if an algorithm can be added to the system through the GUI to Fulfil requirement 3 and 5			
Environment: Windows 7 JDK 1.7.0_01-b08			
Pre-conditions: none			
Test Case Steps:			
Step Number	Procedure	Expected Results	P/F
1	Start Application	Main window appears	P
2	Select algorithm manager button	Algorithm manager window appears	P
3	See if algorithms are available	A list of algorithms	F
Comments:			
No algorithms have been added to the system – to test at a later date acquire some algorithms to add to the system and see if they can be configured.			

Test case ID: 004			
Test Purpose: To see if a comparison test can be started through the GUI to Fulfil requirement 1,4,8,9			
Environment: Windows 7 JDK 1.7.0_01-b08			
Pre-conditions: Test datasets have been added and test result-datasets have been added to the system			
Test Case Steps:			
Step Number	Procedure	Expected Results	P/F
1	Start Application	Main window appears	P
2	Select comparison tool button	Comparison tool window appears	P
3	See if datasets are available are available in the JComboBoxes	A list of dataset that have been added	P
4	Pick JCheckBoxes to select measurements	The appropriate measures that were selected appear in the window showing the results	P
Comments:			

As there was no way to really test the ability to actually run a genuine algorithm, as it was hard to procure a working one, a ‘fake’ algorithm was created. This executable jar takes an input in the form of a CSV file and then copies the file to the output, this was to test the theory that a jar could run from the system and that the process worked correctly. Further future tests with some real algorithms would be an invaluable addition to the project.

For requirements 10 and 11, it was only necessary to come up with a method to add a new algorithm or measure and not actually implement it. These have been discussed earlier.

Requirement 12 was to make a suitable API that can be used by other applications, to actually pass this requirement is almost impossible, as what is a suitable API? I have tried to make the code as usable possible from other applications but to actually test the API rigorously and develop it further would require lots of testing by other users.

Further test and error logs have been completed which can be found in Appendix C including tests to see if an “algorithm” can be run, this is by no means an exhaustive list. Further testing will be required especially on thread management and exception handling.

Upon evaluation of the project, I feel the design could have been more detailed with clear goals on what classes to implement for every section, but when starting this project the thought of knowing exactly what classes would be needed seemed very daunting and goes against the agile philosophy, designs can always be bettered in hindsight.

Looking back at the requirements from the interim report after the project has been completed makes them seem very naive and lacking in enough detail. Not all the requirements can be measured in a quantifiable way. Who is to say what is suitable? And how can it be measured?

The approach to the project was a relaxed more agile development as this allows change quite easily. Ideas on a better implementation can come during the project as seen when a solution to the algorithm problem was discovered. Had a rigid class diagram been designed at the start and stuck to, the way of solving the algorithm problem may never have been seen.

As for the project scope, this was a bit too large on reflection, but I tried to implement as robust a system as possible, that could be easily distributed. The system meets all the goals it was intended to do from the point of view of the GUI and it is possible to use the jar created from the project in other applications i.e. to retrieve and save datasets.

The system also only has two measures, running time and a utility measure. If there was more time, an inclusion of the ILoss measure and Discernability measure would make the tool a more complete package.

A few more measures that could be used on transactional datasets would have been a nice addition, but this would require parsing the result-datasets and knowing how each algorithm anonymises the data.

5 Future Work

The design to add more measures described previously would be a good place to start on expanding this tool. It would be necessary to create the classes that store and retrieve the files, in a similar way to how the other methods have been implemented in the `FileTools` package. A new class in the `MeasurementTools` package will then have to be created that can run these new measures.

More work could also go into creating other ways to measure datasets; a way to measure the transactional datasets would make the tool better at assisting researchers. A possible starting point to this would be to see how a variety of algorithms anonymise data and create a general parser for the output file; it may be required to expand the `AlgorithmFileType` to include a regular expression string that can be used by the general parser.

Another element that could be implemented would be a more graphical file selection window, containing icons of the files and tooltips that pop up with the information, instead of the current `JList` method.

A more robust method to prepare the data, instead of the simple CSV Editor that has been used in this system would be an ideal addition. This could manipulate data in more ways than current row or column removal method used, maybe in a similar way to the data preparation tools used in WEKA.

The ability to run exe files from within the system would also be an advantage; this would mean the system could run algorithms written in different languages. This would save the time consuming process of converting from another language to java.

To increase the scope of the system the ability to handle other file formats that datasets come in should be added. A starting point here would be to introduce the ARFF format the WEKA tool uses. Being able to process other file formats would mean data can be used from a variety of sources other than CSV.

A database connection ability may also be a useful addition, as it would collect data from a database. This data could then used in the algorithms contained in the system, saving the user time as they could query the database directly.

6 Conclusions

The goals of the project were to develop a java based test tool that can store datasets, run algorithms to produce output and compare algorithm outputs using a set of metrics, through a process of research-specification then design-implementation cycles.

The research and specification demonstrates the scope of the problem and how this project only tackles a few goals in this area. The design and implementation shows that it is possible to create a tool for these purposes. It would be feasible to expand the project to encompass more goals, though more research and better specification/requirements would be needed to accomplish this.

The testing of the tool shows that the goals have been achieved but more rigorous testing is needed to fully say that the tool is fit-for-purpose.

The final system produced allows a user to complete the goals set in a basic way, and will serve as a suitable platform to carry on future work and make the tool robust and useful for researchers to critically examine their algorithms performance.

7 Reflection on Learning

When I first started the course as a whole, I thought the main focus would be to learn programming languages and create programs, it turned out that this is a minor and an ‘easy’ task to accomplish, the ‘real’ skills are the ability to manage projects from requirement gathering to delivering results. I had no idea that so much documentation was needed for this task and how useful that documentation is.

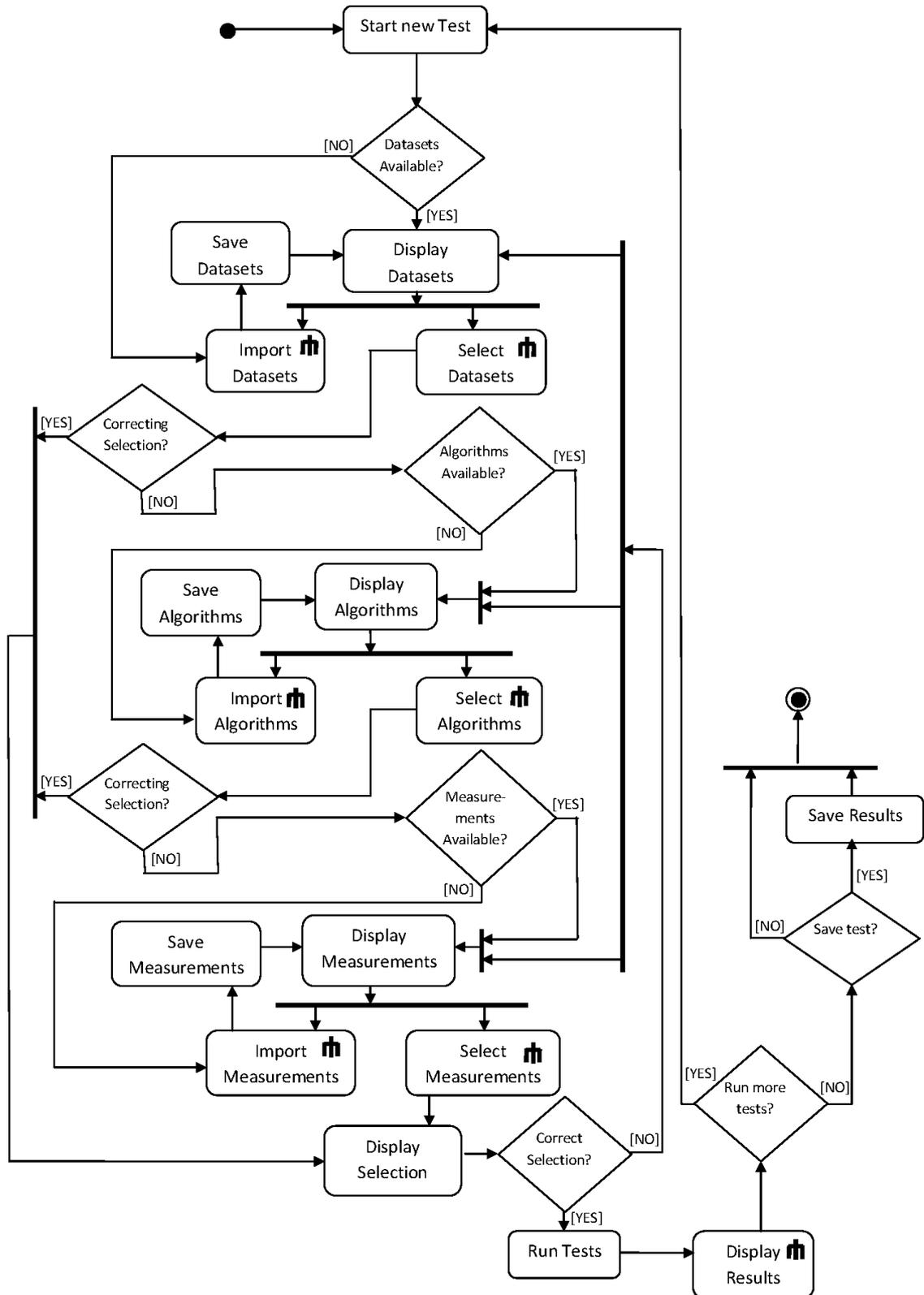
When approaching this project it was difficult to imagine the scale of the work that was needed and the time it would take to accomplish the goals satisfactorily. All the other projects accomplished so far were small and required little documentation or was a group effort where you were given small tasks to complete each week.

Although we have had many lectures on requirement gathering and many examples to work from, to actually produce said requirements from a real life problem proved difficult for me. This skill definitely needs working on but the experience of doing this will be useful to apply to other projects.

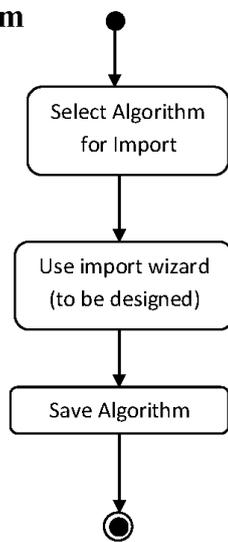
This project has forced me to learn to manage time, document everything, review any project on a regular basis and make sure requirements can be measured. The ability to use this experience to gauge time constraints on further projects is also an invaluable experience that will be useful in the future.

Appendix A: Old Process Diagrams

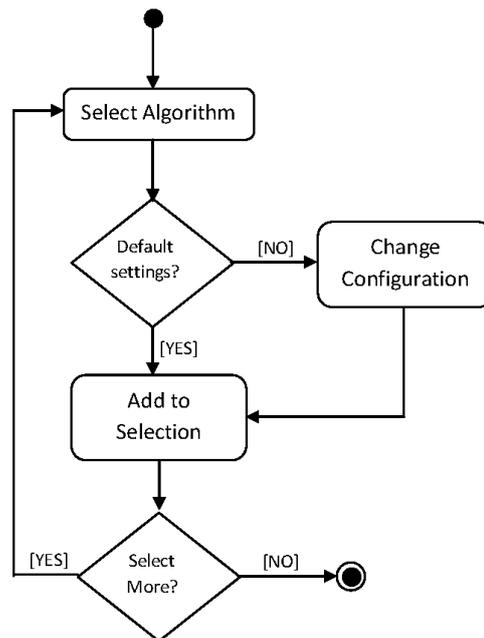
This appendix shows how the new design has changed from my original idea.



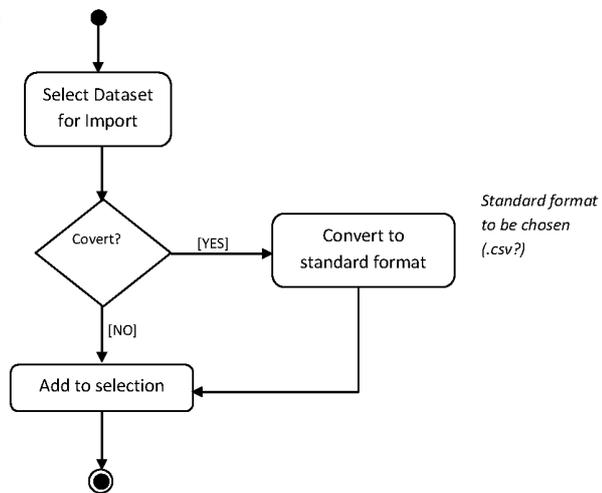
Import Algorithm



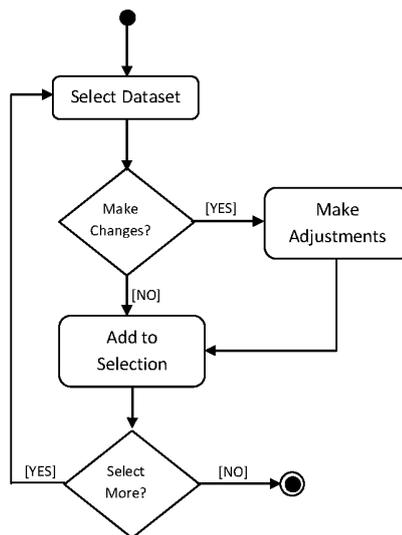
Select Algorithm



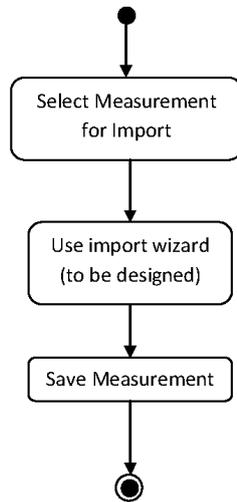
Import Datasets



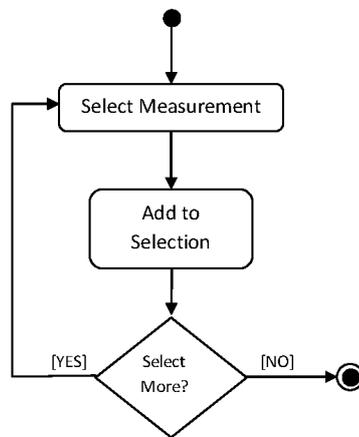
Select Datasets



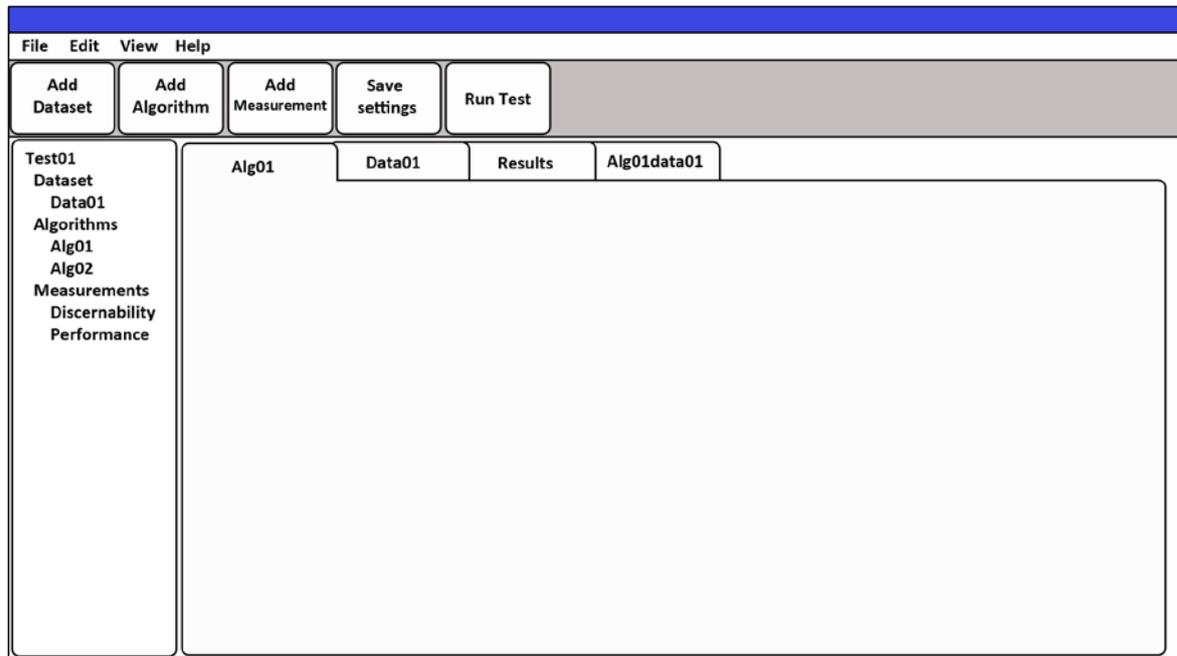
Import Measurement



Select Measurement



Appendix B: Old GUI Design



This design was discarded as it would have been far too complicated to create in the time frame.

Appendix C: Testing

Bug/Error	Date Found	Date Solved
Continues problems with Hung algorithm	20/02/12	Not solved
Relative locations	02/03/12	Not solved at this time
Not being able to import algorithms	03/03/12	04/03/12
XML data management file not working	04/03/12	06/03/12
Layout of JPannels	06/02/12	06/02/12
Database does not refresh table location 0,0	20/02/12	20/02/12
Database does not like uneven arrays.	03/03/12	04/03/12
Run algorithm problems with string using test algorithm	10/03/12	14/03/12
Creating the HTML text	15/03/12	16/03/12
Threading problem	18/03/12	20/03/12
Relative locations	19/03/12	21/03/12