Team Mike Comprehensive Metrics table – CodeMR Plugin

Package Util			LOC	Coupling	Complexity (general)	LCOM	DIT	McCabe	WMC	NOM	Nested Block Depth
	DataBaseResults		24	Low - 1	Low - 1	0.571	1		8	8	1
		<Method>DataBaseResults.public DataBaseResults( ): void	4	1	1			1			1
		<Method>DataBaseResults.public DataBaseResults(String, String, double): void	4	1	1			1			1
		<Method>Utilities.DataBaseResults.public getGeos( ): String	2	1	1			1			1
		<Method>Utilities.DataBaseResults.public getRefDates( ): String	2	1	1			1			1
		<Method>Utilities.DataBaseResults.public getValues( ): double	2	1	1			1			1
		<Method>Utilities.DataBaseResults.public setRefDates(String): void	2	1	1			1			1
		<Method>Utilities.DataBaseResults.public setValues(double): void	2	1	1			1			1
	Forecasting		223	Low-medium – 2 **CBO: 7**	Low-medium – 3	1.0	6		13	5	6
		<Method><Anonymous> : LUtilities/Forecasting$2466;.public actionPerformed(ActionEvent): void		Very-High – 5 **CBO: 12**	0					Number of methods called: 114 (java function)	
		<Method>Utilities.Forecasting.private createDataset(String): DefaultCategoryDataset	12	Low – 1 **CBO: 3**	1			3		NOM called: 12	3

Team Mike Comprehensive Metrics table – CodeMR Plugin

		<Method>Utilities.Forecasting.public Forecasting( ): void	166	1 **COB: 1**	1			3		NOM called: 26	7
		<Method>Utilities.Forecasting.public prepareData(ArrayList): Instances	17	Low-medium – 2 **COB: 4**	1			2		NOM called: 12	2
		<Method>Utilities.Forecasting.public static main(String): void	6	1	1			1		NOM called: 4	1
		<Method>Utilities.Forecasting.public updateBasedOnGeo(String): void	2	1	1			1		NOM called: 1	1
		<Method>Utilities.Forecasting.public updateDataset(String): ArrayList	9	1 **COB: 2**	1			3		NOM called: 7	3
	MainWindow		234	Low – 1	Mid-High - 3	1.0	6		24	4	6
		<Method>Utilities.MainWindow.private createDataset(String): DefaultCategoryDataset	10	1	1			3		NOM called: 11	3
		<Method>Utilities.MainWindow.public MainWindow(String): void	152	1	1			2		NOM called: 37	3
		<Method>Utilities.MainWindow.public setTabularViews(String, String, String): TabularView	7	1	1			1			1
		<Method>Utilities.MainWindow.public static getMonthNumber(String): String	26	1	2 (low-mid)			13		NOM called: 12	2
		<Method>Utilities.MainWindow.public static main(String): void	14	1	1			1		NOM called: 2	1
		<Method>Utilities.MainWindow.public updateDataset(String, String, String, String): void	13	1	1			4		NOM called: 12	3
	MySQLAccess		62	Low – 1	Low - 1	0.563	1		15	6	1

Team Mike Comprehensive Metrics table – CodeMR Plugin

		<Method>Utilities.MySQLAccess.private close( ): void	9	1	1			5		NOM called: 3	3
		<Method>Utilities.MySQLAccess.public connectToDataBase( ): void	7	1	1			2		NOM called: 2	2
		<Method>Utilities.MySQLAccess.public readEntireDataBase( ): ArrayList	11	1	1			2		NOM called: 5	2
		<Method>Utilities.MySQLAccess.public sendQuery(String, String, String): ArrayList	11	1	1			2		NOM called: 5	2
		<Method>Utilities.MySQLAccess.public sendQuery(String): ArrayList	9	1	1			2		NOM called: 3	2
		<Method>Utilities.MySQLAccess.private writeResultSet(ResultSet): ArrayList	10	1	1			2		NOM called: 6	2
	NHIPComparison		130	Low - 1	Mid-High - 3	0.0	6		15	4	6
		<Method>Utilities.NHIPComparison.private calculateMannWhitneyU(ArrayList, ArrayList): String	10	1	1			3		NOM called: 9	2
		<Method>Utilities.NHIPComparison.public calculateTtest(ArrayList, ArrayList): String	21	1	1			6		NOM called: 14	2
		<Method>Utilities.NHIPComparison.public NHIPComparison( ): void	68	1	1			2		NOM called: 26	4
		<Method>Utilities.NHIPComparison.public static main(String): void	7	1	1			1		NOM called: 5	1
		<Method>Utilities.NHIPComparison.public updateDataset(String, String, String): ArrayList	12	1	1			3		NOM called: 8	3
	TabularView		169	Low – 1	Mid – High – 3	1.0	6		12	6	6
		<Method>Utilities.TabularView.private createToggleButton( ): JToggleButton	12	1	1			1		NOM called: 8	1
		<Method>Utilities.TabularView.private getDBResults(String): ArrayList	11	1	1			3		NOM called: 4	2

Team Mike Comprehensive Metrics table – CodeMR Plugin

		<Method>Utilities.TabularView.public getTable( ): JPanel	2	1	1			1		NOM called: 0	1
		<Method>Utilities.TabularView.private rawData(ArrayList): JPanel	36	1	1			2		NOM called: 48	2
		<Method>Utilities.TabularView.public static main(String): void	8	1	1			1		NOM called: 4	1
		<Method>Utilities.TabularView.private summaryTable(ArrayList): JPanel	58	1	1			2		NOM called: 56	2
		<Method>Utilities.TabularView.public TabularView(String, String): void	37	1	1			2		NOM called: 17	3

Note to TA: This table above provides the basic metrics needed for this report that were listed on the handout. For some calculations in which we need other metrics that CODEMR provides we will be providing screen shots attached separately to maintain accuracy and transparency.

Also it may seem strange that we have 1 as complexity a lot but honestly, our code is very simple.

More info on CodeMR Eclipse Plugin: [CodeMR | Static Code Analysis and Software Quality Features]

Team Mike Comprehensive Metrics table – CodeMR Plugin

Please read the following code smells and refactoring fixes, then navigate to the github link at the end of the doc to completely understand the before and after!

Code Smells

1) Large Class

This code smell is present within two of our classes. MainWindow.java and Forecasting.java.

a) MainWindow.java (whole class)

To prove that we have large class code smell we consider four metrics. First is **LOC** (Lines of Code) which for this class is **234**, the largest amount in the project. The **WMC** (Weighted Method Count) is also needed. For each class it can be seen from our table above as **24**, which in comparison to the other classes is the highest. CodeMR calculates TCC as LTCC (Lack of TCC), in this case a higher metric means lower cohesion. Below is a table of all LTCC for our classes.

Class	LTCC (higher means lower)
DataBaseResults	0.429
Forecasting	1.0
MainWindow	0.9
MySQLAccess	0.0
NHIPComparison	1.0
TabularView	0.667

In this case the **LTCC** of MainWindow is **0.9**, which is second highest in our project. The last metric we need for our explanation is **ATFD** (Access to Foreign Data) which is also provided by CodeMr. Since our project is small, we at most access one other class.

Class	ATFD
DataBaseResults	0
Forecasting	1
MainWindow	1
MySQLAccess	0
NHIPComparison	1
TabularView	1

Our ATFD is **1** in this case which can be neglected because all other ATFD are also 1. Our WMC is Very High, in fact it is the highest, and our TCC is very low, definatly lower than one third. Based on Unit 4 slide 127, these metrics combined prove that MainWindow has the large class code smell.

b) NHIPcomparison.java (whole class)

The reasoning for this follows the above with some changes. The LOC are clearly not the largest, but this code smell does not entirely depend on LOC, but instead the responsibilities of the class. It has the second highest WMC meaning that the methods are very heavy and complex. It is also tied for lowest cohesion score.

- **LOC** = 130, **WMC** = 15, **LTCC** = 1.0, **ATFD** = 1

This code smell is a problem because it indicates that our classes are designed poorly and have low cohesion. Low cohesion itself indicates errors in design and high complexity. It makes our code harder to understand to the third party which can slow development and other tasks in the future.

2) Feature Envy

c) rawData() and summaryTable() in TabularView.java [Lines 54-75]

These two methods purely based on the return of getDBResults() which is a method inside TabularView.java and uses methods from MySQLAccess.java. These three methods combined have less to do with the making of the tabular view and access more methods and attributes outside the TabularView class. A metric we can use to prove this smell is similar to ATFD but instead we would like to draw upon Number of **Method Calls (NOM Called)** on the table for these methods. Both have the highest amount of method calls in the whole project (**56** and **48)** and the WMC for this class is not even equivalent to them. They are calling more methods and therefore attributes from outside this class than the class they exist in.

3) Shotgun Surgery

d) TabularView.java

If there was a need for changes in the GUI components for the table views this class would need to change in many places. We already used **metrics in c)** for this class to explain how many methods it calls (Similar to CM and CC metric from Unit 4 Slide 128) but on top of that each component is in the same class and is implemented differently. For example, A change in the toggleButton would require manual changes within rawData(), summaryTable(), and eventListeners. The fix for this is to implement each component in their own class. This code smell makes our code less organized, prone to duplication (which can be seen in the creation of the table components), and harder to maintain (exact example of GUI change).

4) Long Method

The bottom three methods carry the Long Method code smell. The metrics that we took into consideration were **LOC** to judge the size of the method**, McCabe Cyclomatic Complexity** to gauge how complex the method is to see if we have presence of spaghetti code, and **Nested Block Depth** (Maximum Nesting Level) to judge the number of statements blocks that are nested within the method creating a chain of calls. The reason that this code smell should be eliminated is for the sole purpose to make the code easier to understand. This general reasoning applies to f, g, and h below.

e) calculateTtest() in NHIPcomparison.java

This method is not the largest by size as compared to the upcoming two but it definitely is complex and the largest method within its class, hence us considering it a Long Method. This method alone calls upon 14 other methods (NOM Called).

- Method is excessively large
  - LOC = 21, making it the largest method minus constructor on the table
- Method had many conditional branches
  - CYCLO (McCabe) >= High
  - 6 >= 4 (4 is considered high complexity for class in CodeMR)
- Method has deep nesting
  - 2, not the largest but this is alongside the high complexity score

f) summaryTable() in TabularView.java

- LOC = 58, largest method in class.
- CYCLO = 2, its in the middle
- Nesting Depth = 2, not the deepest but this method calls 56 other functions which have deeper nests

g) Forecasting() in Forecasting.java

- LOC = 166
- CYCLO = 3, almost high
- Nesting Depth = 7, very deep

5) Duplicate Code

This code smell is self explanatory, we feel that metrics were not needed in this case. Getting rid of this code smell will make our project have less LOC and simplify the code. This can make the project more efficient, cheaper, easier to test, and easier to onboard a dev. The risks with keeping it are the opposite. Within i) the method in question returns dataset objects but the method is declared in each class which can be fixed. Within j) the string is created over and over again to create the chart title which is redundant and takes up space. String builder can be used to refactor it.

h) NHIPComparison, Forecasting & MainWindow classes all share date updating methods which store string data structures

i) Update Titles of charts in MainWindow.java constructor

```
"NHIP " + selectedCountry + " VS " + selectedCountry2, // Chart title
"Date", // X-Axis Label
"NHIP Value", // Y-Axis Label
dataset
```

j) There exists duplicated code within Mainwindow.java and NHIPComparison.java in the form of string array data structures. The later refactoring's show will do a better job at explaining.

6) Long Parameter List

There is no need for metrics for this smell, but essentially within our MainWindow.java and NHIPComparison.java classes we had a lot of methods that would call upon different types of string, the parameter list would hit 4. This was clearly not good design and an obvious antipattern. The reason that we need to fix this is because it will allow us to find code that is duplicated, and we can use the extracted parameter list to make the code even shorter. This is exactly what happened when you read our refactoring's.

k) UpdateDataset() within MainWindow.java

l) TabularView() within MainWindow.java

m) ActionPreformed() within NHIPComparison.java

All of the methods use the same type of string parameters of startDate, endDate, selectedCountry1, and selectedCountry2. Not only that but they also accessed the String date data structures mentioned before.

Q3) Some of the code smells identified have not been refactored because they were only calculated based off metrics to show understanding (we showed more than 10 smells). The descriptions of the 10 refactoring's are below, when pictures and updated metrics at the end of the document.

<u>Fixing Large Class code smell</u>

1. Mainwindow.java

- What we did to fix this large class is extract as many methods as possible to make the code cleaner. The first change we made was to extract the getMonthNumber() method and call it from a different class. This makes this part of the code easier to manipulate and locate for future changes. This also enforces the single responsibility principle as there is now a specific class that changes the month string into a number for the queries. (We noticed that we have to do a lot of shotgun surgery, this point will be extrapolated within that section). Pictures are below.

2. NHIPcomparison.java

    - The methods that we extracted within this class is calculateTTest() and calculateMannWhitneyU(). The logic follows a) in the sense that we moved large chucks of code that could be organized better by giving them their own class to enforce SRP. The calculation methods logic is now spit up form the creation of the GUI so if either or need to be switched they can be done so from the respective and properly labeled class. This change can be explained as extracting class and interface as the GUI and logic for the statistical comparisons now have their own classes.

Fixing Long Method Code Smell

3) calculateTTest() in StatisticalCalculations.java (Originally NHIPComparison.java, but previous refactoring extracted the method)

    - Before we had created this new class to store our statistical tests. Within this class we also have long method of calculateTTest() which does two things, it implements logic and updates GUI text. We Moved the updating of the GUI text to its own method so it can be easier to modify the text in the future, enforce SRP, and make our design more modular. The picture for 2) can be used to explain this.

Fixing Long Parameter List

4) Creation of TimeSeriesStartEnd()

    - Within the MainWindow.java we had a long parameter list code smell within the updateDataset() method. We used the refactoring principle of creating an new object class instead of having a long list of parameters, this allowed us to access this objects using dot notation and again further enforcing SRP as now the dataset has its own callable objects that live in their own new class called TimeSeriesStartEnd().

5) Another long method that we can fix is also within MainWindow.java and it is for the tabularView() methods. The fix and rational follows above. The TabularView method now accepts a single parameter which is an object stored in a different class called TimeSeriesStartEnd(). We use dot notation to access the needed variables that the function needs. (NOTE: this smell was first listed as long method but previous refactoring's removed that smell)

6) actionPreformed() within NHIPComparison.java

- for update button, update ttest, fixed parameters, passing ttest method. Recall the SelectedMonth class we made before. This method used to first check for every instance but now we can just call the method in the SelectedMonth class and reference it as an object.

Fixing Duplicate Code

Fixing the passing of the same parameters, now we are just having one object and updating the country of the object

7) Settabularview, before we were passing selectedcountry and the same start and end date but now we are creating a new object and just updating the country mainwindow.java lines 198-211
    a. This refactor builds upon 5) because the long parameter list was used in many methods and calls within the methods as well. We reduced the amount of redundant code by extracting the class and calling the object.

8) Month and year string arrays shifted to their own class. The MainWindow.java and NHIPcomparison.java had a copy of a large list of string arrays, both were the same and held the same purpose in each class. The fix to this was to extract that code into its own class and allow the arrays to be accessed globally between the classes. This again enforces SRP because this new class called DatesetYearsMonths.java only has one function.

Fixing Shotgun Surgery

9) ActionPreformed() and other queries within MainWindow.java

    - Like mentioned before this code would require the input of dates from the database and then used a manual string builder in many locations. The moving of the SelectedMonth.java class allows us to be able to change anything to do with the date from one class rather than having to jump through lines of code.

10) ActionPreformed() in NHIPComparison.java

    - The rational follows 9), but in this case we also use extracted class TimeSeriesStandEnd.java which was first created to reduce the parameter list, but instead this allowed us to be able to change the start and end times from a single class rather than having to jump across lines within this method. Now the code only needs to be changed in one place rather than many.

Team Mike Comprehensive Metrics table – CodeMR Plugin

mehtabjalaf/TeamMikeD3: This is a repository for EECS3311 D3 to show our refactoring's (github.com)

Please click the link above to visualize the refactors. The commit is named D3 REFACTOR. The metrics after refactoring are also present on the repo as PNG in a folder.