<FINAL DELIVERABLE>

REFACTORING PLUTARCH PARALLEL LIVES

VERSION <1.0>

Ιορδάνης-Ραφαήλ Φυδανάκης AM:420

Ευάγγελος Δημουλής AM:421

TABLE OF CONTENTS

INTRODUCTION

The objective of this work is to refactor the legacy code of the project Plutarch Parallel Lives, which is part of the graduate course “Software & Data Evolution”. As a first step, we performed actions towards understanding the architecture of the project and reverse engineer the legacy code by generating the corresponding package diagrams, class diagrams CRC cards per class etc. Afterwards, we listed the problems detected at the architecture and design level, code and style level and prepared a preliminary plan of actions towards re-engineering the code. As a next step, we developed JUnit tests regarding general use cases of the project in order to facilitate the migration process, and we also applied some initial refactoring to the code. Finally, we proceeded to the refactoring of the legacy code using the knowledge we obtained at the fore mentioned steps and by applying refactoring patterns studied at class lectures. As a result, we solved architecture and design problems, prepared tests for future use and refactoring, and improved code quality.

*** You can find the **source code** of the project in GitHub at:

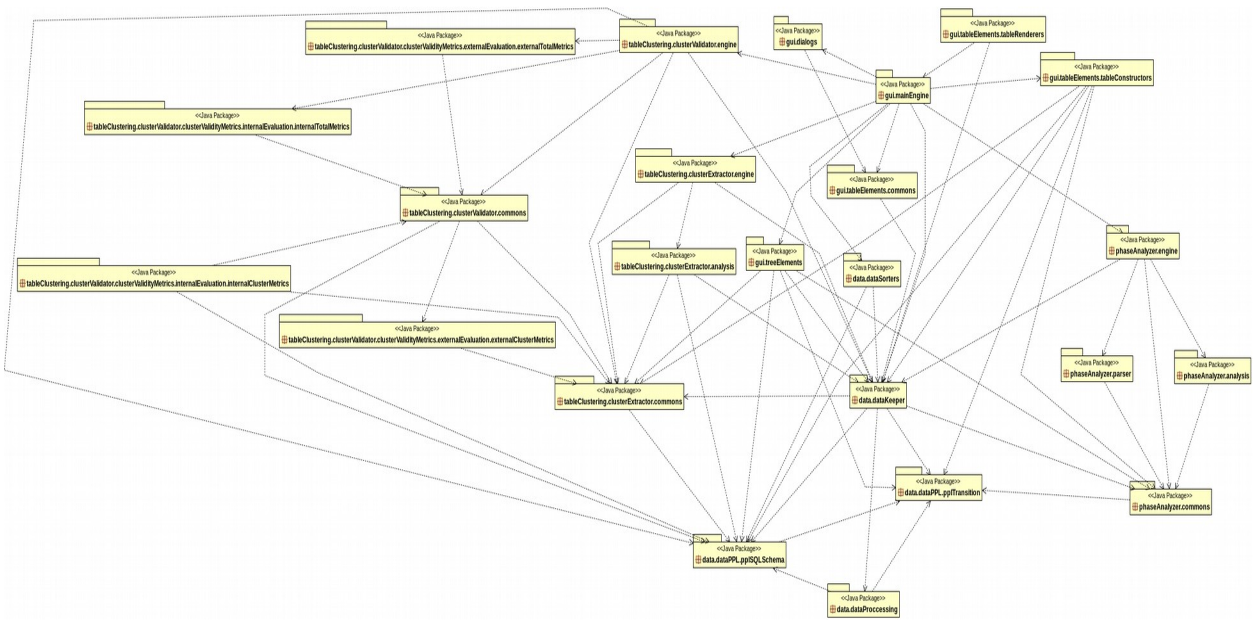
https://github.com/JordanRaphael/l1-software-data-evolution

DESIGN RECOVERY

In this section we present and analyze the architecture and design of Plutarch by providing the initial package and class diagrams before applying any refactoring to the legacy code. In order to focus better on the problems we detected, in this report we will not provide all the diagrams however they are included in the deliverable.

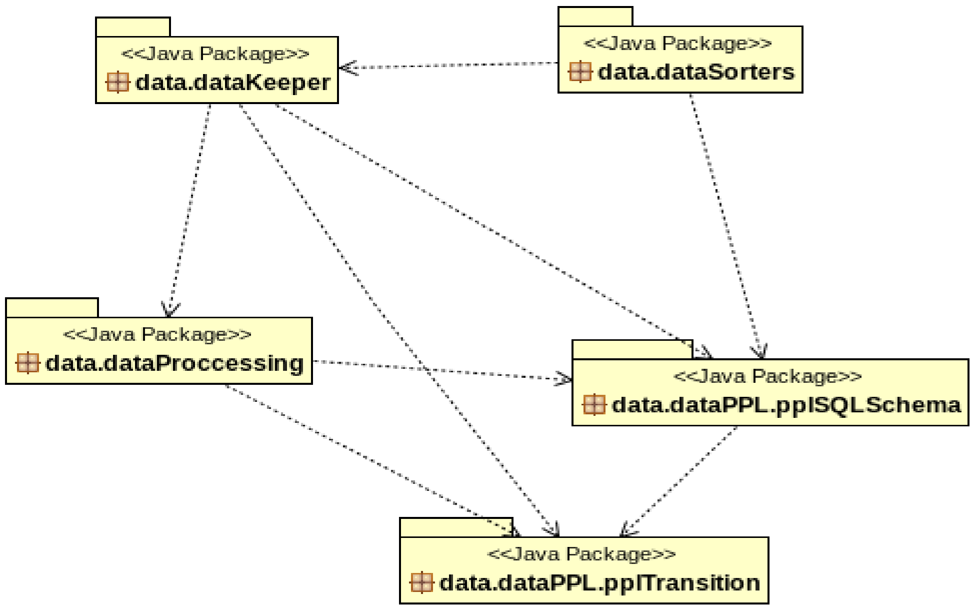
PACKAGE DIAGRAMS

Project hyper package diagram: shows structure and dependencies between all sub-packages of the system. The gui.mainEngine and data.DataKeeper packages present many dependencies with other packages of the system.

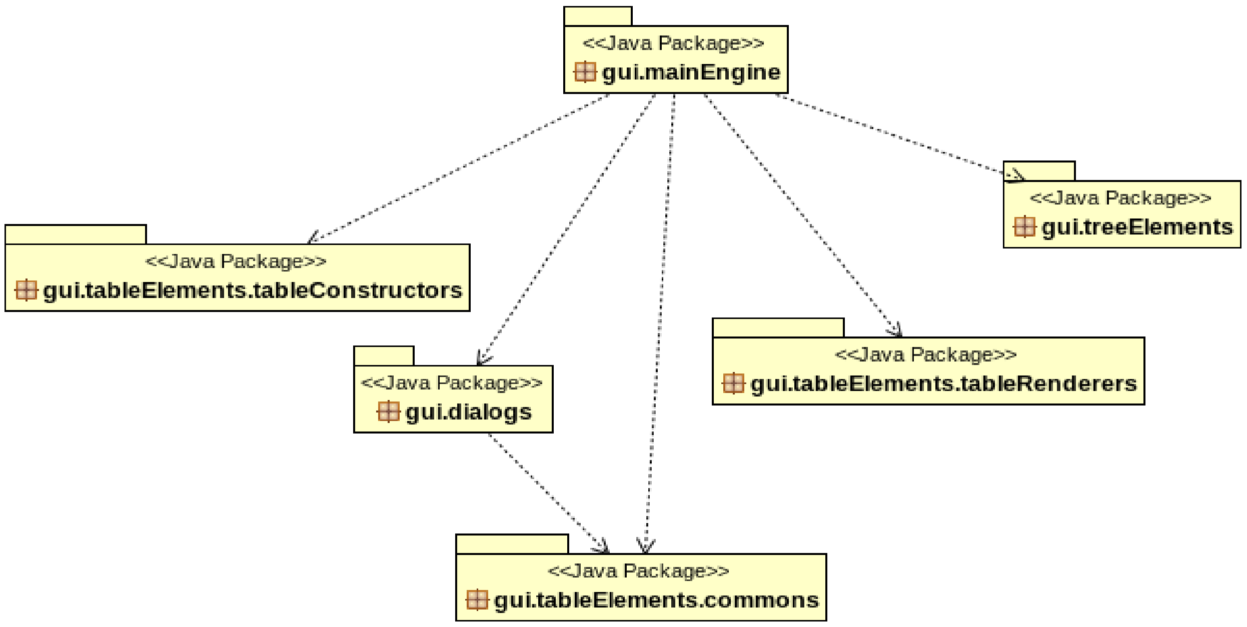


<<Java Package>> tableClustering.clusterValidator.clusterValidityMetrics.externalEvaluation.externalTotalMetrics
<<Java Package>> tableClustering.clusterValidator.engine
<<Java Package>> gui.dialogs
<<Java Package>> gui.tableElements.tableRenderers
<<Java Package>> gui.mainEngine
<<Java Package>> gui.tableElements.tableConstructors
<<Java Package>> tableClustering.clusterValidator.clusterValidityMetrics.internalEvaluation.internalTotalMetrics
<<Java Package>> tableClustering.clusterExtractor.engine
<<Java Package>> gui.tableElements.commons
<<Java Package>> tableClustering.clusterValidator.commons
<<Java Package>> phaseAnalyzer.engine
<<Java Package>> tableClustering.clusterExtractor.analysis
<<Java Package>> gui.treeElements
<<Java Package>> data.dataSorters
<<Java Package>> tableClustering.clusterValidator.clusterValidityMetrics.internalEvaluation.internalClusterMetrics
<<Java Package>> tableClustering.clusterValidator.clusterValidityMetrics.externalEvaluation.externalClusterMetrics
<<Java Package>> phaseAnalyzer.parser
<<Java Package>> phaseAnalyzer.analysis
<<Java Package>> tableClustering.clusterExtractor.commons
<<Java Package>> data.dataKeeper
<<Java Package>> data.dataPPL.pplTransition
<<Java Package>> phaseAnalyzer.commons
<<Java Package>> data.dataPPL.pplSQLSchema
<<Java Package>> data.dataProccessing

Data hyper package diagram: shows structure and dependencies between all sub-packages of the data package. Cycles between the packages can be observed here.

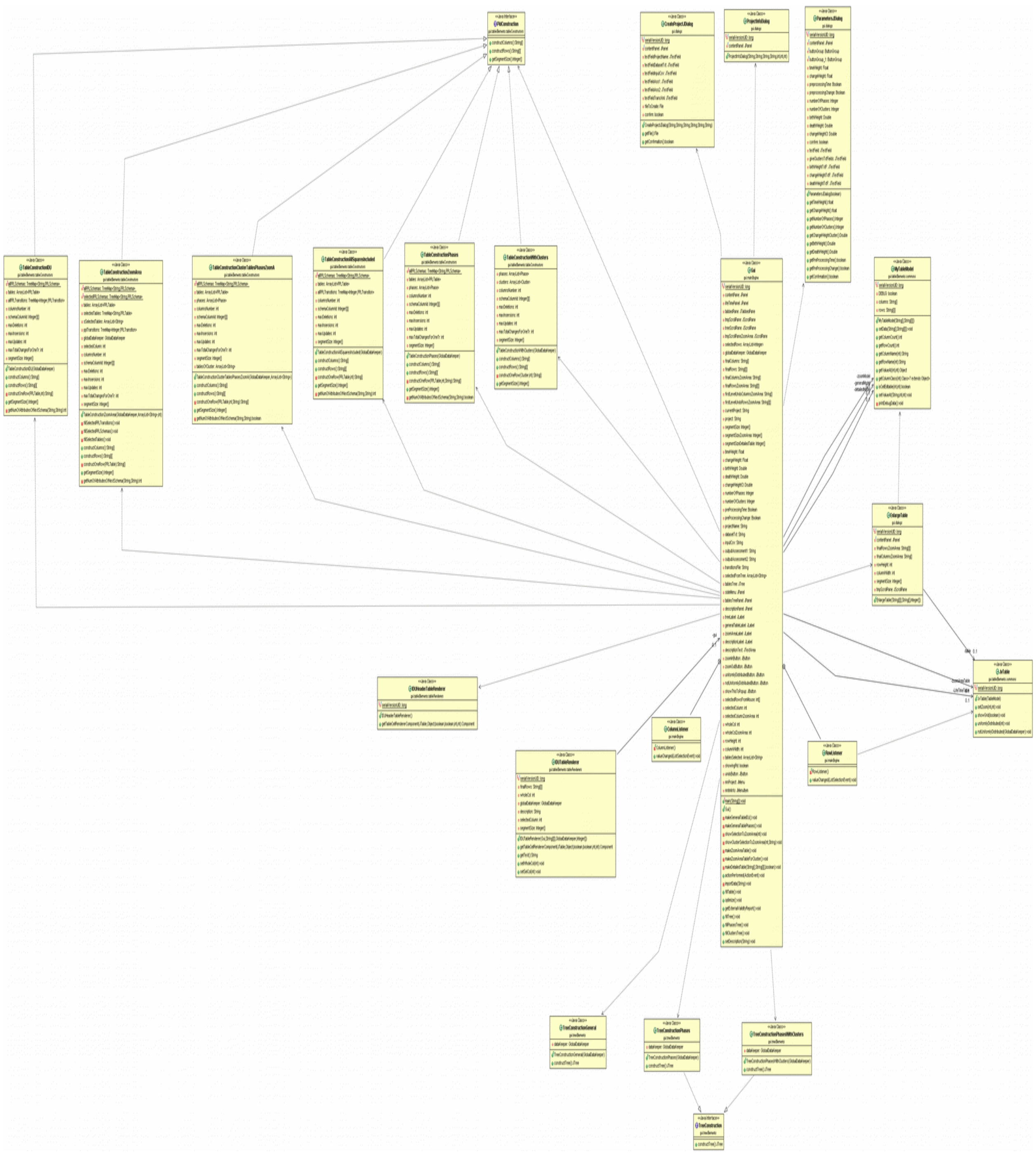


Gui hyper package diagram: shows structure and dependencies between all sub-packages of the GUI. The gui.mainEngine package depends on all other packages.

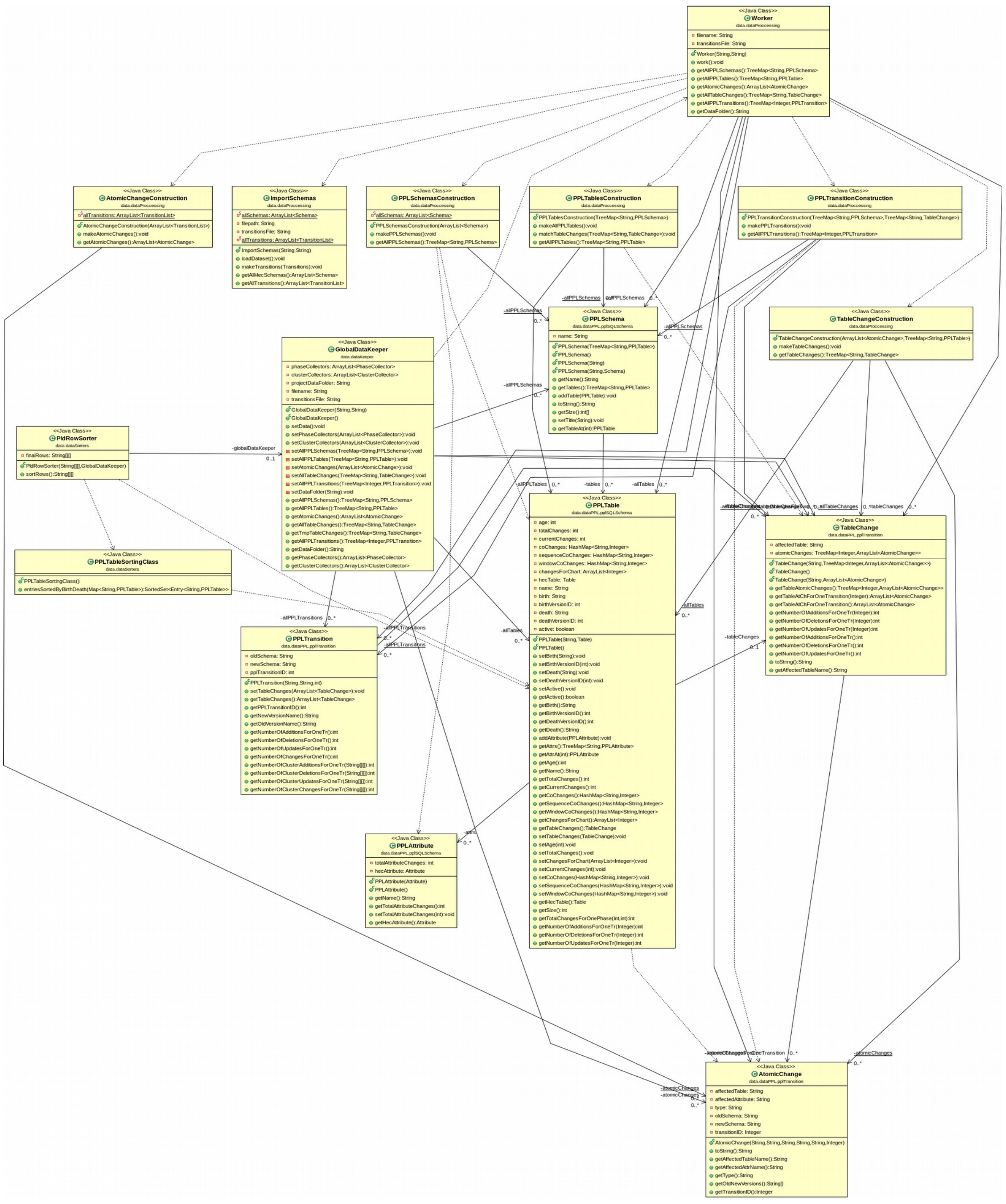


CLASS DIAGRAMS

Gui class diagram: high coupling and many dependencies between the Gui class and all other classes of the GUI package.



<u>Data class diagram:</u> many dependencies on Worker class. GlobalDataKeeper does not contain any processing logic but only accessor and mutator methods.



LIST OF PROBLEMS

In this section we report a list of architecture, design and source code style problems. After identifying and listing the problems, we provide a preliminary plan of actions towards re-engineering the legacy code.

ARCHITECTURE & DESIGN

We detected two main issues at the package level of the project.

First, there is a lack of back-end "server" package that hosts the business logic and as a result the gui.MainEngine package has dependencies to all other packages which the GUI depends on and increases the coupling. The back-end "server" package would play the role of an intermediate between the DataKeeper and the GUI populated with "load" methods, which are currently in the GUI package. Furthermore, in the GUI there are methods that do some useful processing on data and we need to move the logic to the DataKeeper i.e. "Move behavior close to data" pattern. This will result in converting the Data class to something like a "Service Provider". The DataKeeper will become a "God Class” that way. As a next step, we will divide that class into smaller subsystems that delegate work to other classes. "Facade" pattern.

The two main packages where problematic behavior has been observed are:

(1) gui.MainEngine (2) data.dataKeeper

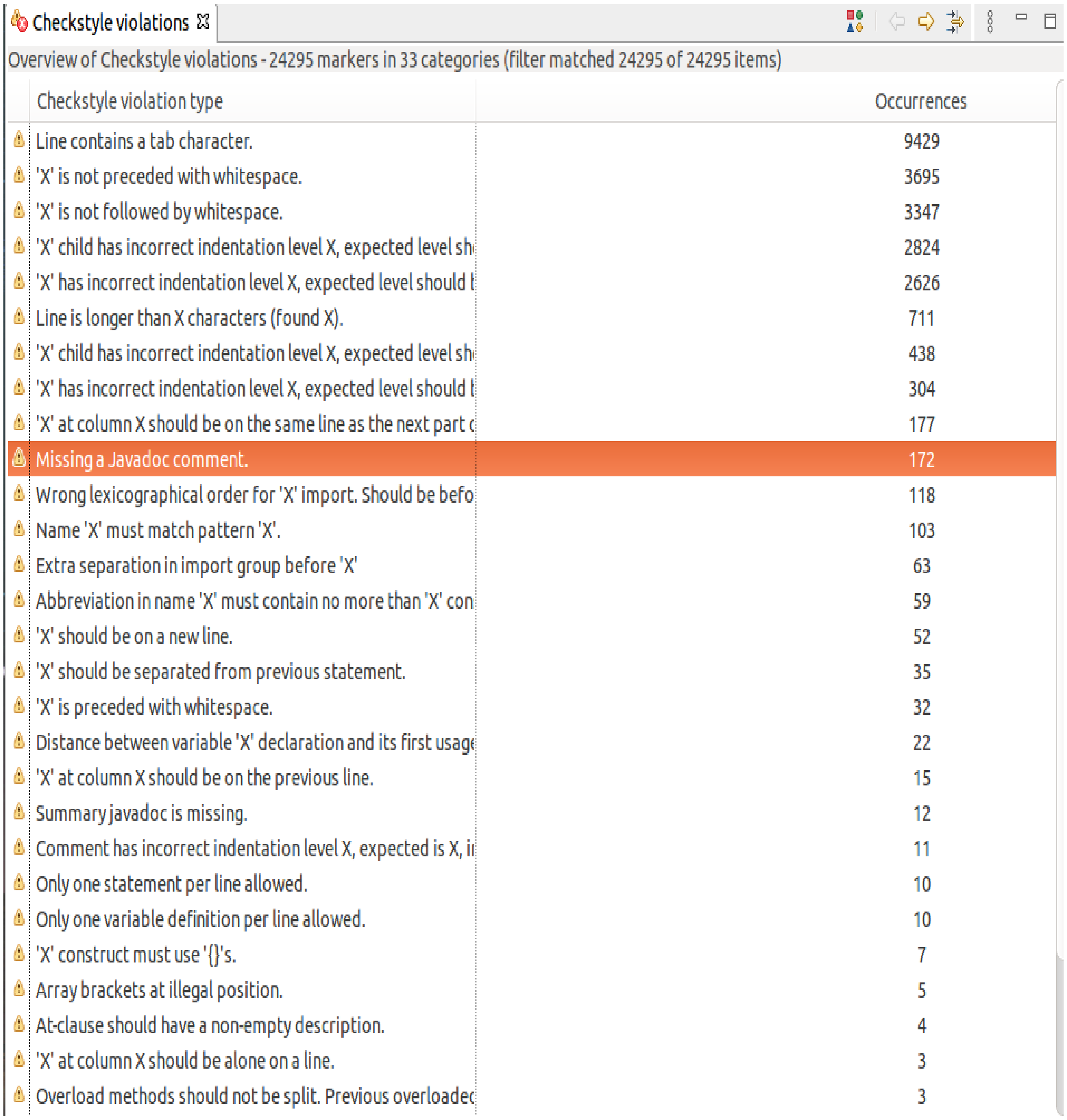
At the class level of the project we identified problems mainly at two packages:

1. At the Data package, the GlobalDataKeeper makes use of the Worker class to create the data and then copies it to his own class attributes (unnecessary step). We will remove these attributes from GlobalDataKeeper.

2. At the Gui package the Gui class acts as a "God" class, it has many dependencies inside the Gui package and also outside of it, it is composed of very long methods and contains too many attributes. As a result we will work on refactoring this class so that it is populated only with the front-end code logic.

STYLE VIOLATIONS

CheckStyle Violations (before refactoring)



Checkstyle violations

Overview of Checkstyle violations - 24295 markers in 33 categories (filter matched 24295 of 24295 items)

Checkstyle violation type	Occurrences
Line contains a tab character.	9429
'X' is not preceded with whitespace.	3695
'X' is not followed by whitespace.	3347
'X' child has incorrect indentation level X, expected level sh	2824
'X' has incorrect indentation level X, expected level should b	2626
Line is longer than X characters (found X).	711
'X' child has incorrect indentation level X, expected level sh	438
'X' has incorrect indentation level X, expected level should b	304
'X' at column X should be on the same line as the next part c	177
Missing a Javadoc comment.	172
Wrong lexicographical order for 'X' import. Should be befo	118
Name 'X' must match pattern 'X'.	103
Extra separation in import group before 'X'	63
Abbreviation in name 'X' must contain no more than 'X' con	59
'X' should be on a new line.	52
'X' should be separated from previous statement.	35
'X' is preceded with whitespace.	32
Distance between variable 'X' declaration and its first usage	22
'X' at column X should be on the previous line.	15
Summary javadoc is missing.	12
Comment has incorrect indentation level X, expected is X, ir	11
Only one statement per line allowed.	10
Only one variable definition per line allowed.	10
'X' construct must use '{}'s.	7
Array brackets at illegal position.	5
At-clause should have a non-empty description.	4
'X' at column X should be alone on a line.	3
Overload methods should not be split. Previous overloadec	3

PRELIMINARY PLAN OF ACTIONS

1. Develop tests for the existing legacy system.

2. Add a Business logic class, intermediary to the GUI and the Data packages in order to offload the GUI from data processing logic.

3. Move methods that process data to the DataKeeper in order to bring behavior closer to data.

4. Divide the DataKeeper into smaller subsystems that handle the logic depending on the type of processing we need to apply.

5. Apply general refactoring methods on the code e.g. extract methods, duplicate code,

clean dead code, spaghetti code, factory etc.

6. If needed, add more tests to verify the functionality of the refactored version of the

system.

TESTING

In this section we report the list of tests that we constructed to facilitate the migration process. In order to develop tests for Plutarch Parallel Lives tool, we had to do some basic refactoring on the legacy code so as for the code to be testable. The tests we developed involve mainly general use cases of the tool and the framework we used is JUnit. In most cases, we use the Atlas project to construct the tests and compare the output to the dumped data we generated.

We developed tests for GUI's createProject, editProject, loadProject, zoomIn and zoomOut. In each test we have a ground truth and we test that the methods give us the expected results. Additionally, we added tests regarding all table display options the tool provides such as: Show PLD, Show Phases PLD, Show Phases with Clusters PLD and Show Full Detailed LifeTime table.

As for the GlobalDataKeeper, we developed a test to verify that the method setData() works properly. To achieve that, we had to create a file with the expected results of the setData() method. Afterwards, we call the importData() which uses the setData() and other methods to insert the data to the GUI components. When this step is complete, we check if the imported data match perfectly the data that should be imported normally. The tests are configured to be automated by selecting "Run as JUnit" at the tests package.

Next we provide a list of tests we developed along with a brief description for each test.

- **testCreateProject:** populates a new dialog with test data needed to create a new project.
- **testEditProject:** imports the Atlas project and compares the dumped data we get from edit in the project to the data generated in the test.
- **testLoadProject:** imports the Atlas project using the ProjectManager class (will be introduced later) and compares the contents of the project to the dumped contents.
- **testZoomIn:** imports the phpBB project, sets constants about the size of the Gui components and compares them to what they should be.
- **testZoomOut:** same concept as in testZoomIn.

- **testSetData:** imports the Atlas project and populates the GlobalDataKeeper module with data in order to compare this data to the dumped data that we use as a ground truth.
- **testShowPLD:** imports the Atlas project, use the ShowPLD functionality and check the integrity of the updated data.
- **testShowPhasesPLD:** populates the dialog used to display the phases of the loaded project, executes the processing logic to fill the table and compares the table contents to the dumped contents.
- **testShowPhasesWithClusters:** same concept as in testShowPhasesPLD adding a number of clusters parameter.
- **testShowFullDetailedLifetimeTable:** compares the contents of the Full Detailed Lifetime to the contents of the dumped Lifetime table.

REFACTORING THE LEGACY CODE

In this section we present the steps towards refactoring the legacy code by breaking up the GUI to subsystems handling the business logic of the project and moving logic related to any useful processing on data to the DataKeeper. As a last step, we broke up the newly generated GlobalDataKeeper, which was renamed to GlobalDataManager, to subsystems handling the client requests, thus making the GlobalDataManager act as a “Service Provider”.

GUI PACKAGE

Through the class diagram and by studying the code of the Gui class, we observed that the Gui contained all the business logic of the project such as loading a project, editing a project etc., which was hidden inside action listeners. Moreover, those action listeners contained additional processing logic which applied to data generally stored in the back-end and not in the Gui making the Gui act as a “God Class”. In order to resolve the issues we observed, we took the following steps to extract functionality from the Gui class and from he GUI hyper-package:

Moving packages from GUI to data package

We moved **gui.treeElements** package to the data hyper-package, renaming it to **data.treeElements**. This package contains classes responsible for constructing JTree objects. However, even though JTree objects are part of java.swing package, these classes make use of the GlobalDataKeeper to fill a JTree with data and contain processing logic in them. As a result we moved the entire package to the data hyper-package to bring behavior closer to data.

As a next step, we moved the **gui.tableElements.tableConstructors** package to the data hyper-package, renaming it to **data.tableConstructors**. The tableConstructors package is responsible for constructing and filling table contents. using to a greater extent the back-end than the front-end similarly to the treeElements package fore mentioned.

Extracting the listeners and the business logic from the GUI

In order to split up the Gui class we created some extra classes, each with distinct responsibilities in order to extract functionality from the Gui and reduce also duplicate code.

JItemsHandler: The JItemsHandler class is responsible for the creation and initialization of the JItems which are part of the Gui. The creation of these components generates a lot of duplicate code which is reduced by creating methods at the JItemsHandler to handle the initialization and reduce duplication.

GuiController: The GuiController class acts as an intermediary between the front-end and the back-end of the application. At first, all listeners and processing logic which was part of the Gui were moved to this class. The GuiController handles the interaction between the GlobalDataManager and the Gui classes by populating the Gui with the data fetched from the back-end, and it also stores all business logic for loading a project creating a project etc.

ZoomTableListenerHandler: The ZoomTableListenerHandler class is responsible for handling and storing all the logic related to the action listeners used for manipulating the **"Zoom Table"** of the application at mouse click events mostly.

GeneralTableListenerHandler: The GeneralTableListenerHandler class is responsible for handling and storing all the logic related to the action listeners used for manipulating the **"General Table"** of the application at mouse click events mostly.

TableRenderer: The Table Renderer class is an abstract class which defines two methods for the creation of DefaultTableRenderers which are responsible for rendering the view of a table after changes to the table occur. We use this abstraction in the next classes to generalize the creation process of table renderers and add a level of abstraction to the code.

TablesFactory: The TablesFactory class acts a factory and is responsible for the creation of objects of type TableRenderer. After specifying one of the two existing table types, the class handles the creation of the corresponding TableRenderer object, which inherits from the type TableRenderer.

GeneralTableRendererHandler: The GeneralTableRendererHandler extends the TableRenderer class and implements the abstract methods defined at TableRenderer. It is responsible for the creation of DefaultTableRenderer objects which are part of displaying the **“General Table”** contents at the Gui.

ZoomTableRendererHandler: The ZoomTableRendererHandler extends the TableRenderer class and implements the abstract methods defined at TableRenderer. It is responsible for the creation of DefaultTableRenderer objects which are part of displaying the **“Zoom Table”** contents at the Gui.

Moving data processing logic to data package

After refactoring the Gui class and splitting up the logic into subsystems, the GuiController class contained methods and logic related to data processing. We moved this logic into the GlobalDataManager to bring behavior closer to data. The methods **populateWithClusters( )** and **populatedWithPhases( )** are a brief example of methods solely doing some useful processing on data, related to Cluster objects and Phase data used when the clustering algorithms are triggered at the back-end of the project. Similarly, **showClusterSelectionZoomArea( )** operates on Cluster data and prepares this data to be visualized in the Gui. More details about how the GlobalDataManager operates and what functionality is pushed there will be given in the next section.

DATA PACKAGE

One of the first problems that we observed by taking a brief look at the source code of the data package is that the global data keeper was a huge data container without any logic. So, our goal was to break it down to simpler classes with useful processing logic. At the development stage, the first thing we did was to construct tests, so that we can test that any change to the source code did not break the logic or functionality of the tool. For the data package, we developed a test for the *setData( )* method, which sets up all the necessary data for the tool to function properly. As a next step, we started moving logic from Gui to GlobalDataKeeper, in order to have a purpose besides being a data container. At the final phase of the refactoring procedure, as we moved logic to the GlobalDataKeeper, it became a god class, so we split it up to five concrete classes, the GlobalDataManager which acts as a intermediary between the GuiController and the Data, “facade” pattern, and four classes that store and process the data of the back-end.

Issues observed at the Data Package

As a first step to the refactoring of the PPL’s source code, we created the UML diagrams of the project to have a look at the higher level of the project’s architecture and design. By inspecting the UML diagrams we observed that the GlobalDataKeeper was being used as a data container by the subsystems of the project, so we had to assign responsibilities to it before splitting it up at the final phase of the project.

Testing

At the test development stage, we developed a test for the data package as well. We tested the *setData( )* method in order to have a way to test that the storing of the data and the retrieval of it, are intact after making changes to the source code. The *setData( )* method uses the other classes of the dataKeeper package, in order to store the schemas, tables, transitions, table changes, atomic changes and all the necessary project information. To test this method, we have a file that’s our ground truth, it contains all the data that should have been stored to the dataKeeper classes for a given project. So, we load the specified project, we use the *setData( )* method and recreate a file using the newly added data. If the new file is the same as the ground truth, we are certain that the *setData( )* method works as expected. If they aren't the same, the test fails.

Moving Logic Closer to Data

Next, we extracted logic and methods from the GuiController, to move to the GlobalDataKeeper. But this refactoring created another problem, the GlobalDataKeeper became a "God Class" itself with too many responsibilities.

Refactoring GlobalDataKeeper

After our changes to the GlobalDataKeeper, it became obvious to us that we had to split up this class. So we split it up to five classes. The GlobalDataManager which handles the storing of the data and four classes that actually store the data and process these data. We also moved even more logic out of GuiController and transferred it to the four classes that actually handle the processing.

GlobalDataManager

This class is now a manager class that manages the storing of the data, an intermediary between GuiController and Data packages. We used a variation of "facade" pattern here to delegate the processing to the corresponding subsystems of the data package.

DataCollectorsManager

This class manages the phase and cluster collections of the project, as well as their processing. Its purpose is to save these data and process it on demand to serve to the front-end of the application.

PPLDataManager

This class stores the schemas, tables and transitions of the project. It processes these data on demand for other subsystems of the project.

ProjectManager

This is another class that stores data, more specifically, it stores the information of the loaded project such as project name, dataset path, project folder etc. Its purpose is to parse the input file of a project, and provide to any class that needs it, the project's information.

TableChangeManager

This class stores the changes that have been made to the tables of the loaded project. Its purpose is to store these data, process them and provide them when needed.

REFACTORING PATTERNS

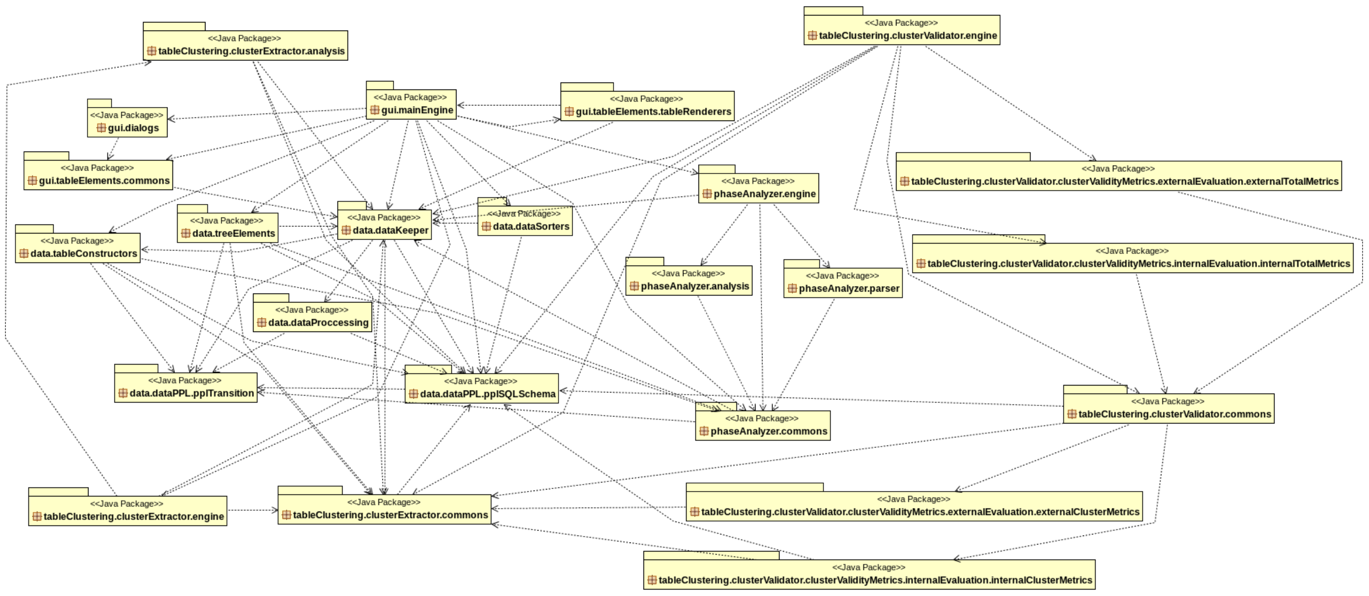
A variation of "Factor Out State" pattern was used at the Gui Package where we created a Factory class that handles the creation of the Table Renderers corresponding to each table. "Facade" was applied at the GlobalDataManager in order to delegate the processing logic from the GlobalDataManager to the subsystems that compose the manager, each with its distinct functionality on the stored data. Moreover, the pattern "Bring behavior closer to data" was used, where functionality was pushed to the back-end of the project, thus reducing the use of accessor and mutator methods to process the data with respect to the fundamentals of "Demeter's Law".

REFACTORED SYSTEM ARCHITECTURE

In this section we present and analyze the architecture and design of the refactored version of Plutarch by providing the initial package and class diagrams after refactoring the legacy code showing how responsibilities were split to subsystems and hierarchies were added.

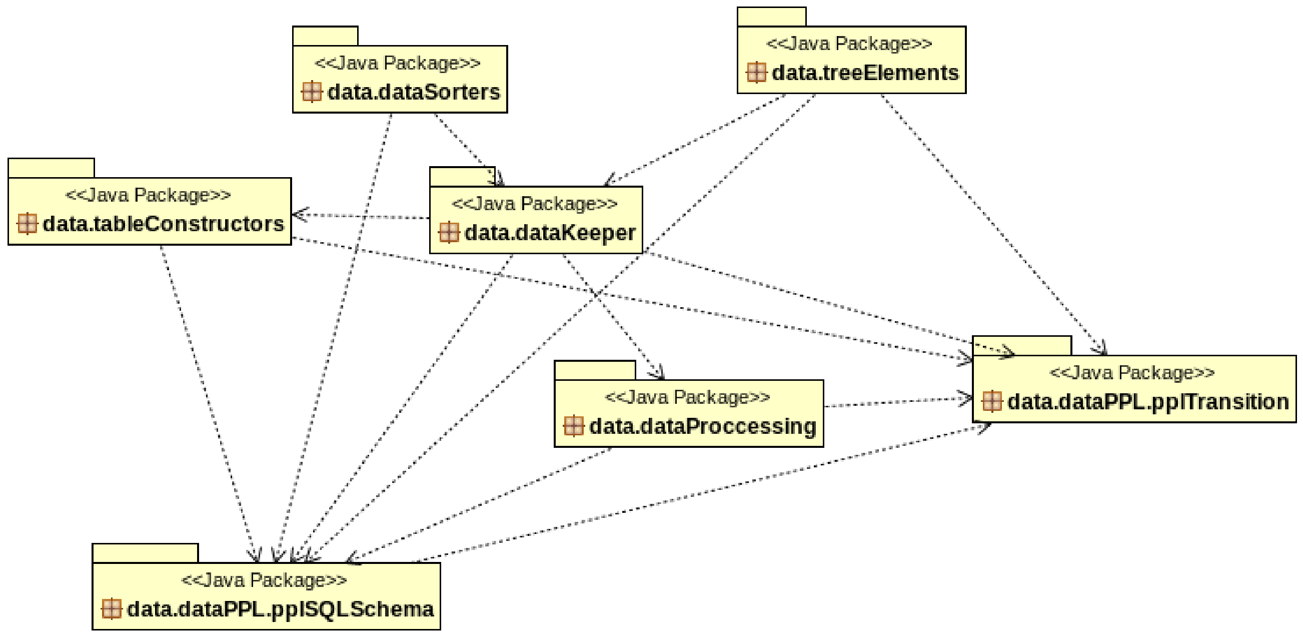
PACKAGE DIAGRAMS

Project hyper package diagram: shows the structure and dependencies between all sub-packages of the system after the refactoring.

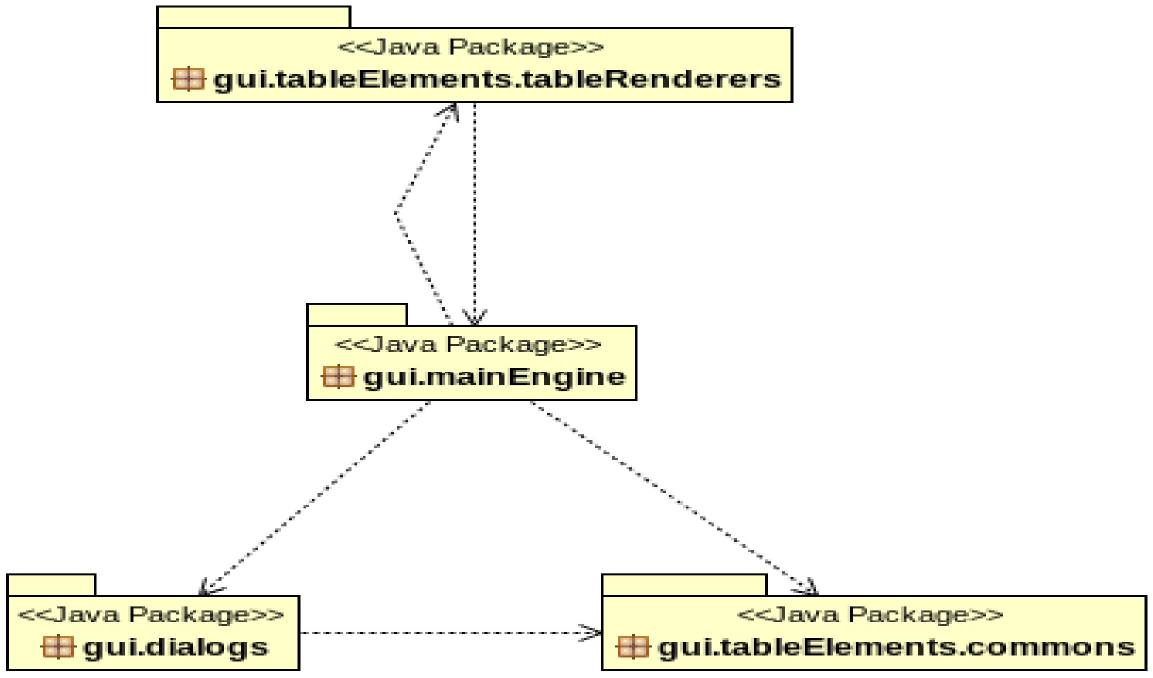


<<Java Package>> tableClustering.clusterExtractor.analysis
<<Java Package>> tableClustering.clusterValidator.engine
<<Java Package>> gui.mainEngine
<<Java Package>> gui.tableElements.tableRenderers
<<Java Package>> gui.dialogs
<<Java Package>> gui.tableElements.commons
<<Java Package>> tableClustering.clusterValidator.clusterValidityMetrics.externalEvaluation.externalTotalMetrics
<<Java Package>> phaseAnalyzer.engine
<<Java Package>> data.treeElements
<<Java Package>> data.dataKeeper
<<Java Package>> data.dataSorters
<<Java Package>> data.tableConstructors
<<Java Package>> tableClustering.clusterValidator.clusterValidityMetrics.internalEvaluation.internalTotalMetrics
<<Java Package>> phaseAnalyzer.analysis
<<Java Package>> phaseAnalyzer.parser
<<Java Package>> data.dataProccessing
<<Java Package>> data.dataPPL.pplTransition
<<Java Package>> data.dataPPL.pplSQLSchema
<<Java Package>> phaseAnalyzer.commons
<<Java Package>> tableClustering.clusterValidator.commons
<<Java Package>> tableClustering.clusterExtractor.engine
<<Java Package>> tableClustering.clusterExtractor.commons
<<Java Package>> tableClustering.clusterValidator.clusterValidityMetrics.externalEvaluation.externalClusterMetrics
<<Java Package>> tableClustering.clusterValidator.clusterValidityMetrics.internalEvaluation.internalClusterMetrics

Data hyper package diagram: shows structure and dependencies between all sub-packages of the refactored data package. Cycles that existed between the packages were broken.

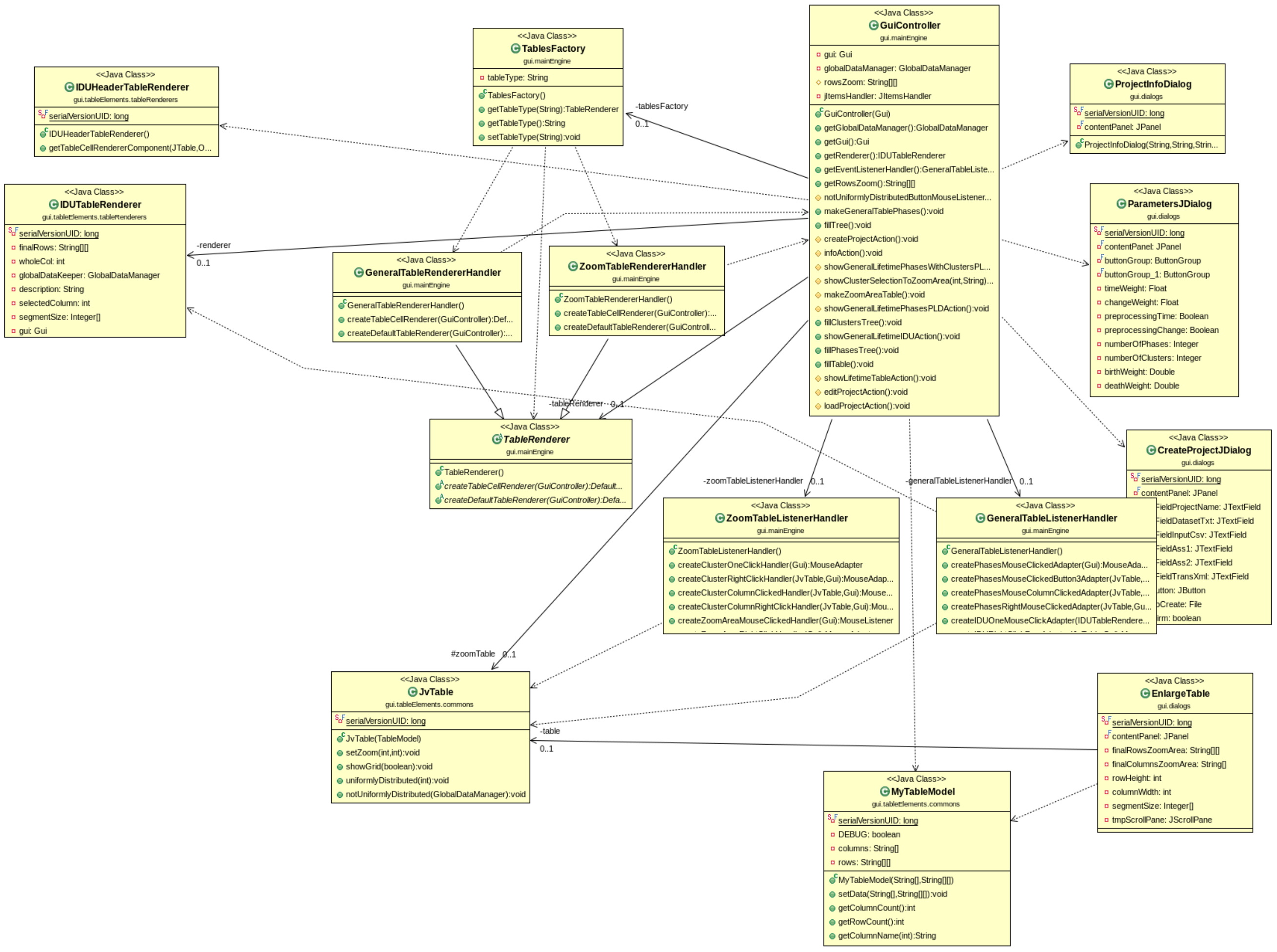


<u>Gui hyper package diagram:</u> shows structure and dependencies between all sub-packages of the GUI.



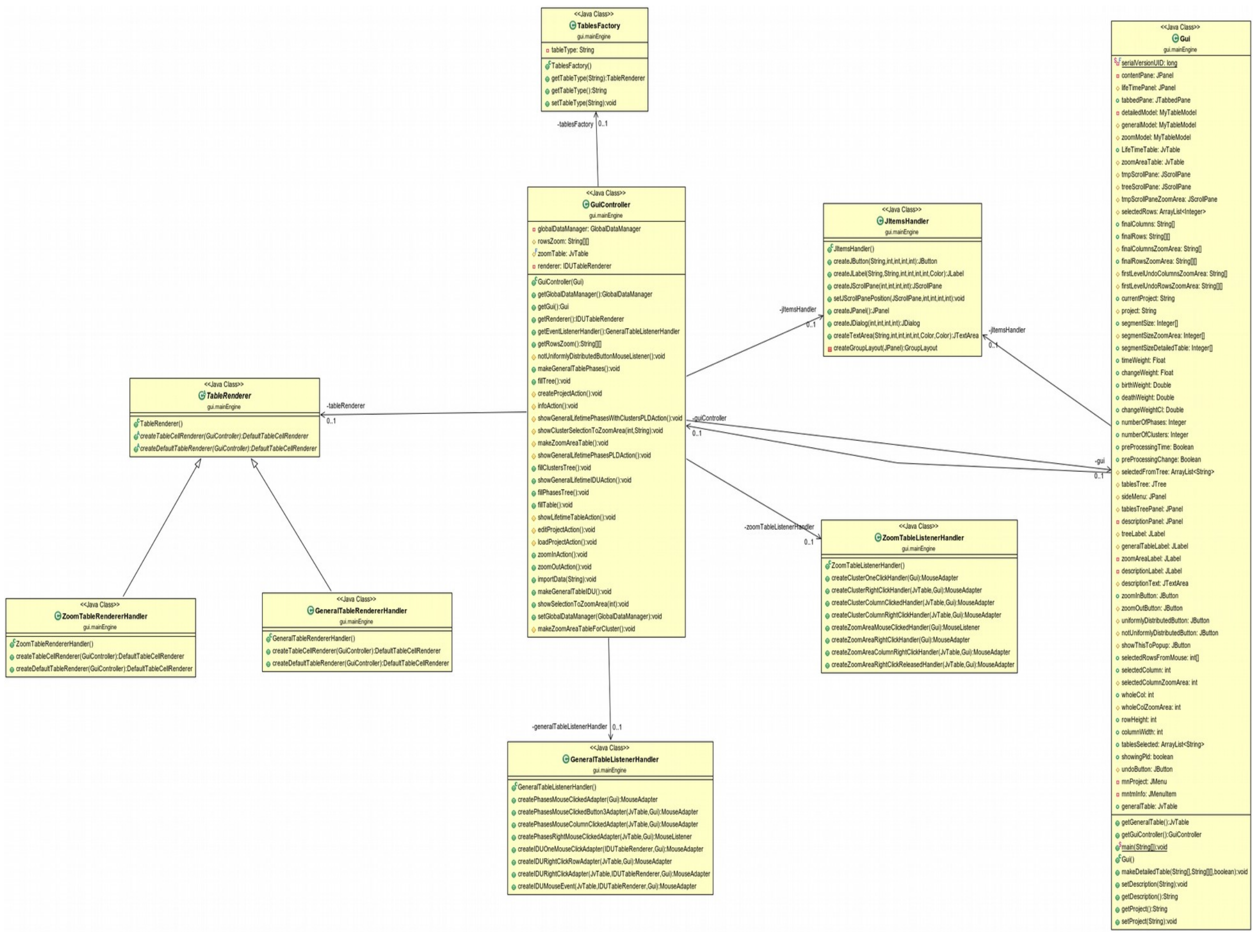
CLASS DIAGRAMS

Gui class diagram: after applying refactoring patterns, the Gui class was split up to subsystems distributing the responsibilities. As a result more classes were added to the diagram with levels of hierarchy becoming more clear.



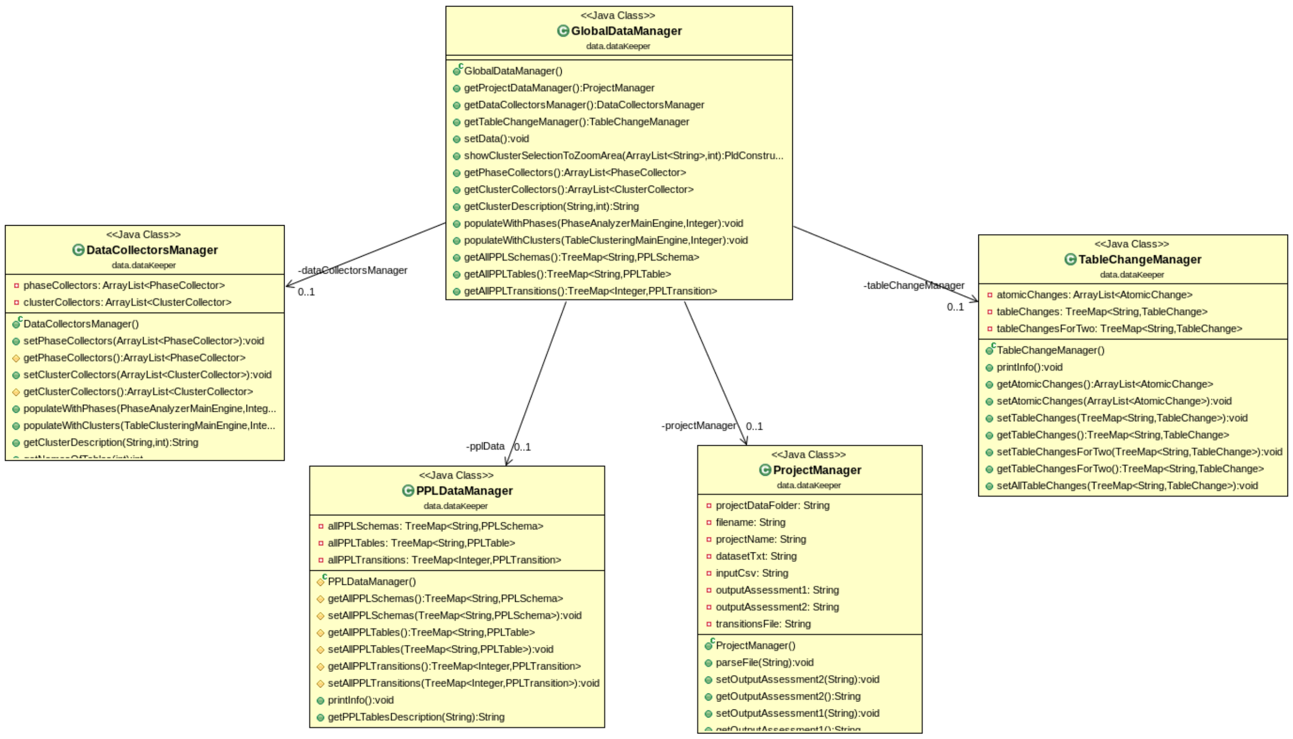
<<Java Class>> TablesFactory gui.mainEngine
tableType: String
TablesFactory()
getTableType(String):TableRenderer
getTableType():String
setTableType(String):void
<<Java Class>> GuiController gui.mainEngine
gui: Gui
globalDataManager: GlobalDataManager
rowsZoom: String[][]
jItemsHandler: JItemsHandler
GuiController(Gui)
getGlobalDataManager():GlobalDataManager
getGui():Gui
getRenderer():IDUTableRenderer
getEventListenerHandler():GeneralTableListe...
getRowsZoom():String[][]
notUniformlyDistributedButtonMouseListener...
makeGeneralTablePhases():void
fillTree():void
createProjectAction():void
infoAction():void
showGeneralLifetimePhasesWithClustersPL...
showClusterSelectionToZoomArea(int,String)...
makeZoomAreaTable():void
showGeneralLifetimePhasesPLDAction():void
fillClustersTree():void
showGeneralLifetimeIDUAction():void
fillPhasesTree():void
fillTable():void
showLifetimeTableAction():void
editProjectAction():void
loadProjectAction():void
<<Java Class>> IDUHeaderTableRenderer gui.tableElements.tableRenderers
serialVersionUID: long
IDUHeaderTableRenderer()
getTableCellRendererComponent(JTable,O...
<<Java Class>> ProjectInfoDialog gui.dialogs
serialVersionUID: long
contentPanel: JPanel
ProjectInfoDialog(String,String,Strin...
<<Java Class>> IDUTableRenderer gui.tableElements.tableRenderers
serialVersionUID: long
finalRows: String[][]
wholeCol: int
globalDataKeeper: GlobalDataManager
description: String
selectedColumn: int
segmentSize: Integer[]
gui: Gui
<<Java Class>> ParametersJDialog gui.dialogs
serialVersionUID: long
contentPanel: JPanel
buttonGroup: ButtonGroup
buttonGroup_1: ButtonGroup
timeWeight: Float
changeWeight: Float
preprocessingTime: Boolean
preprocessingChange: Boolean
numberOfPhases: Integer
numberOfClusters: Integer
birthWeight: Double
deathWeight: Double
<<Java Class>> GeneralTableRendererHandler gui.mainEngine
GeneralTableRendererHandler()
createTableCellRenderer(GuiController):Def...
createDefaultTableRenderer(GuiController):...
<<Java Class>> ZoomTableRendererHandler gui.mainEngine
ZoomTableRendererHandler()
createTableCellRenderer(GuiController):...
createDefaultTableRenderer(GuiControll...
<<Java Class>> TableRenderer gui.mainEngine
TableRenderer()
createTableCellRenderer(GuiController):Default...
createDefaultTableRenderer(GuiController):Defa...
<<Java Class>> CreateProjectJDialog gui.dialogs
serialVersionUID: long
contentPanel: JPanel
FieldProjectName: JTextField
FieldDatasetTxt: JTextField
FieldInputCsv: JTextField
FieldAss1: JTextField
FieldAss2: JTextField
FieldTransXml: JTextField
utton: JButton
oCreate: File
irm: boolean
<<Java Class>> ZoomTableListenerHandler gui.mainEngine
ZoomTableListenerHandler()
createClusterOneClickHandler(Gui):MouseAdapter
createClusterRightClickHandler(JvTable,Gui):MouseAdap...
createClusterColumnClickedHandler(JvTable,Gui):Mouse...
createClusterColumnRightClickHandler(JvTable,Gui):Mou...
createZoomAreaMouseClickedHandler(Gui):MouseListener
<<Java Class>> GeneralTableListenerHandler gui.mainEngine
GeneralTableListenerHandler()
createPhasesMouseClickedAdapter(Gui):MouseAda...
createPhasesMouseClickedButton3Adapter(JvTable,...
createPhasesMouseColumnClickedAdapter(JvTable,...
createPhasesRightMouseClickedAdapter(JvTable,Gu...
createIDUOneMouseClickAdapter(IDUTableRendere...
<<Java Class>> JvTable gui.tableElements.commons
serialVersionUID: long
JvTable(TableModel)
setZoom(int,int):void
showGrid(boolean):void
uniformlyDistributed(int):void
notUniformlyDistributed(GlobalDataManager):void
<<Java Class>> EnlargeTable gui.dialogs
serialVersionUID: long
contentPanel: JPanel
finalRowsZoomArea: String[][]
finalColumnsZoomArea: String[]
rowHeight: int
columnWidth: int
segmentSize: Integer[]
tmpScrollPane: JScrollPane
<<Java Class>> MyTableModel gui.tableElements.commons
serialVersionUID: long
DEBUG: boolean
columns: String[]
rows: String[][]
MyTableModel(String[],String[][])
setData(String[],String[][]):void
getColumnCount():int
getRowCount():int
getColumnName(int):String
-tablesFactory 0..1
-renderer 0..1
-tableRenderer 0..1
-zoomTableListenerHandler 0..1
-generalTableListenerHandler 0..1
#zoomTable 0..1
-table 0..1

<u>Gui gui.mainEngine class diagram:</u> illustrates the design of the refactored Gui where business logic is handled by the GuiController class which in turn delegates responsibilities to other subsystems as well.



<<Java Class>>
TablesFactory
gui.mainEngine
tableType: String
TablesFactory()
getTableType(String):TableRenderer
getTableType():String
setTableType(String):void
-tablesFactory
0..1
<<Java Class>>
GuiController
gui.mainEngine
globalDataManager: GlobalDataManager
rowsZoom: String[][]
zoomTable: JvTable
renderer: IDUTableRenderer
GuiController(Gui)
getGlobalDataManager():GlobalDataManager
getGui():Gui
getRenderer():IDUTableRenderer
getEventListenerHandler():GeneralTableListenerHandler
getRowsZoom():String[][]
notUniformlyDistributedButtonMouseListener():void
makeGeneralTablePhases():void
fillTree():void
createProjectAction():void
infoAction():void
showGeneralLifetimePhasesWithClustersPLDAction():void
showClusterSelectionToZoomArea(int,String):void
makeZoomAreaTable():void
showGeneralLifetimePhasesPLDAction():void
fillClustersTree():void
showGeneralLifetimeIDUAction():void
fillPhasesTree():void
fillTable():void
showLifetimeTableAction():void
editProjectAction():void
loadProjectAction():void
zoomInAction():void
zoomOutAction():void
importData(String):void
makeGeneralTableIDU():void
showSelectionToZoomArea(int):void
setGlobalDataManager(GlobalDataManager):void
makeZoomAreaTableForCluster():void
<<Java Class>>
JItemsHandler
gui.mainEngine
JItemsHandler()
createJButton(String,int,int,int,int):JButton
createJLabel(String,String,int,int,int,int,Color):JLabel
createJScrollPane(int,int,int,int):JScrollPane
setJScrollPanePosition(JScrollPane,int,int,int,int):void
createJPanel():JPanel
createJDialog(int,int,int,int):JDialog
createTextArea(String,int,int,int,int,Color,Color):JTextArea
createGroupLayout(JPanel):GroupLayout
-jItemsHandler
0..1
-jItemsHandler
0..1
<<Java Class>>
TableRenderer
gui.mainEngine
TableRenderer()
createTableCellRenderer(GuiController):DefaultTableCellRenderer
createDefaultTableRenderer(GuiController):DefaultTableCellRenderer
-tableRenderer
0..1
<<Java Class>>
ZoomTableRendererHandler
gui.mainEngine
ZoomTableRendererHandler()
createTableCellRenderer(GuiController):DefaultTableCellRenderer
createDefaultTableRenderer(GuiController):DefaultTableCellRenderer
<<Java Class>>
GeneralTableRendererHandler
gui.mainEngine
GeneralTableRendererHandler()
createTableCellRenderer(GuiController):DefaultTableCellRenderer
createDefaultTableRenderer(GuiController):DefaultTableCellRenderer
-guiController
0..1
-gui
0..1
-zoomTableListenerHandler
0..1
<<Java Class>>
ZoomTableListenerHandler
gui.mainEngine
ZoomTableListenerHandler()
createClusterOneClickHandler(Gui):MouseAdapter
createClusterRightClickHandler(JvTable,Gui):MouseAdapter
createClusterColumnClickedHandler(JvTable,Gui):MouseAdapter
createClusterColumnRightClickHandler(JvTable,Gui):MouseAdapter
createZoomAreaMouseClickedHandler(Gui):MouseListener
createZoomAreaRightClickHandler(Gui):MouseAdapter
createZoomAreaColumnRightClickHandler(JvTable,Gui):MouseAdapter
createZoomAreaRightClickReleasedHandler(JvTable,Gui):MouseAdapter
-generalTableListenerHandler
0..1
<<Java Class>>
GeneralTableListenerHandler
gui.mainEngine
GeneralTableListenerHandler()
createPhasesMouseClickedAdapter(Gui):MouseAdapter
createPhasesMouseClickedButton3Adapter(JvTable,Gui):MouseAdapter
createPhasesMouseColumnClickedAdapter(JvTable,Gui):MouseAdapter
createPhasesRightMouseClickedAdapter(JvTable,Gui):MouseListener
createIDUOneMouseClickAdapter(IDUTableRenderer,Gui):MouseAdapter
createIDURightClickRowAdapter(JvTable,Gui):MouseAdapter
createIDURightClickAdapter(JvTable,IDUTableRenderer,Gui):MouseAdapter
createIDUMouseEvent(JvTable,IDUTableRenderer,Gui):MouseAdapter
<<Java Class>>
Gui
gui.mainEngine
serialVersionUID: long
contentPane: JPanel
lifeTimePanel: JPanel
tabbedPane: JTabbedPane
detailedModel: MyTableModel
generalModel: MyTableModel
zoomModel: MyTableModel
LifeTimeTable: JvTable
zoomAreaTable: JvTable
tmpScrollPane: JScrollPane
treeScrollPane: JScrollPane
tmpScrollPaneZoomArea: JScrollPane
selectedRows: ArrayList<Integer>
finalColumns: String[]
finalRows: String[][]
finalColumnsZoomArea: String[]
finalRowsZoomArea: String[][]
firstLevelUndoColumnsZoomArea: String[]
firstLevelUndoRowsZoomArea: String[][]
currentProject: String
project: String
segmentSize: Integer[]
segmentSizeZoomArea: Integer[]
segmentSizeDetailedTable: Integer[]
timeWeight: Float
changeWeight: Float
birthWeight: Double
deathWeight: Double
changeWeightCl: Double
numberOfPhases: Integer
numberOfClusters: Integer
preProcessingTime: Boolean
preProcessingChange: Boolean
selectedFromTree: ArrayList<String>
tablesTree: JTree
sideMenu: JPanel
tablesTreePanel: JPanel
descriptionPanel: JPanel
treeLabel: JLabel
generalTableLabel: JLabel
zoomAreaLabel: JLabel
descriptionLabel: JLabel
descriptionText: JTextArea
zoomInButton: JButton
zoomOutButton: JButton
uniformlyDistributedButton: JButton
notUniformlyDistributedButton: JButton
showThisToPopup: JButton
selectedRowsFromMouse: int[]
selectedColumn: int
selectedColumnZoomArea: int
wholeCol: int
wholeColZoomArea: int
rowHeight: int
columnWidth: int
tablesSelected: ArrayList<String>
showingPld: boolean
undoButton: JButton
mnProject: JMenu
mntmInfo: JMenuItem
generalTable: JvTable
getGeneralTable():JvTable
getGuiController():GuiController
main(String[]):void
Gui()
makeDetailedTable(String[],String[][],boolean):void
setDescription(String):void
getDescription():String
getProject():String
setProject(String):void

Data data.Keeper class diagram: by moving the logic closer to data, the data class diagram is populated with more classes but each with distinct responsibilities. Requests from client classes are "forwarded" through the GlobalDataManager to the subsystems of the package.



<<Java Class>>
GlobalDataManager
data.dataKeeper
GlobalDataManager()
getProjectDataManager():ProjectManager
getDataCollectorsManager():DataCollectorsManager
getTableChangeManager():TableChangeManager
setData():void
showClusterSelectionToZoomArea(ArrayList<String>,int):PldConstru...
getPhaseCollectors():ArrayList<PhaseCollector>
getClusterCollectors():ArrayList<ClusterCollector>
getClusterDescription(String,int):String
populateWithPhases(PhaseAnalyzerMainEngine,Integer):void
populateWithClusters(TableClusteringMainEngine,Integer):void
getAllPPLSchemas():TreeMap<String,PPLSchema>
getAllPPLTables():TreeMap<String,PPLTable>
getAllPPLTransitions():TreeMap<Integer,PPLTransition>
-dataCollectorsManager
0..1
<<Java Class>>
DataCollectorsManager
data.dataKeeper
phaseCollectors: ArrayList<PhaseCollector>
clusterCollectors: ArrayList<ClusterCollector>
DataCollectorsManager()
setPhaseCollectors(ArrayList<PhaseCollector>):void
getPhaseCollectors():ArrayList<PhaseCollector>
setClusterCollectors(ArrayList<ClusterCollector>):void
getClusterCollectors():ArrayList<ClusterCollector>
populateWithPhases(PhaseAnalyzerMainEngine,Integ...
populateWithClusters(TableClusteringMainEngine,Inte...
getClusterDescription(String,int):String
-tableChangeManager
0..1
<<Java Class>>
TableChangeManager
data.dataKeeper
atomicChanges: ArrayList<AtomicChange>
tableChanges: TreeMap<String,TableChange>
tableChangesForTwo: TreeMap<String,TableChange>
TableChangeManager()
printInfo():void
getAtomicChanges():ArrayList<AtomicChange>
setAtomicChanges(ArrayList<AtomicChange>):void
setTableChanges(TreeMap<String,TableChange>):void
getTableChanges():TreeMap<String,TableChange>
setTableChangesForTwo(TreeMap<String,TableChange>):void
getTableChangesForTwo():TreeMap<String,TableChange>
setAllTableChanges(TreeMap<String,TableChange>):void
-pplData
0..1
<<Java Class>>
PPLDataManager
data.dataKeeper
allPPLSchemas: TreeMap<String,PPLSchema>
allPPLTables: TreeMap<String,PPLTable>
allPPLTransitions: TreeMap<Integer,PPLTransition>
PPLDataManager()
getAllPPLSchemas():TreeMap<String,PPLSchema>
setAllPPLSchemas(TreeMap<String,PPLSchema>):void
getAllPPLTables():TreeMap<String,PPLTable>
setAllPPLTables(TreeMap<String,PPLTable>):void
getAllPPLTransitions():TreeMap<Integer,PPLTransition>
setAllPPLTransitions(TreeMap<Integer,PPLTransition>):void
printInfo():void
getPPLTablesDescription(String):String
-projectManager
0..1
<<Java Class>>
ProjectManager
data.dataKeeper
projectDataFolder: String
filename: String
projectName: String
datasetTxt: String
inputCsv: String
outputAssessment1: String
outputAssessment2: String
transitionsFile: String
ProjectManager()
parseFile(String):void
setOutputAssessment2(String):void
getOutputAssessment2():String
setOutputAssessment1(String):void

CODE QUALITY

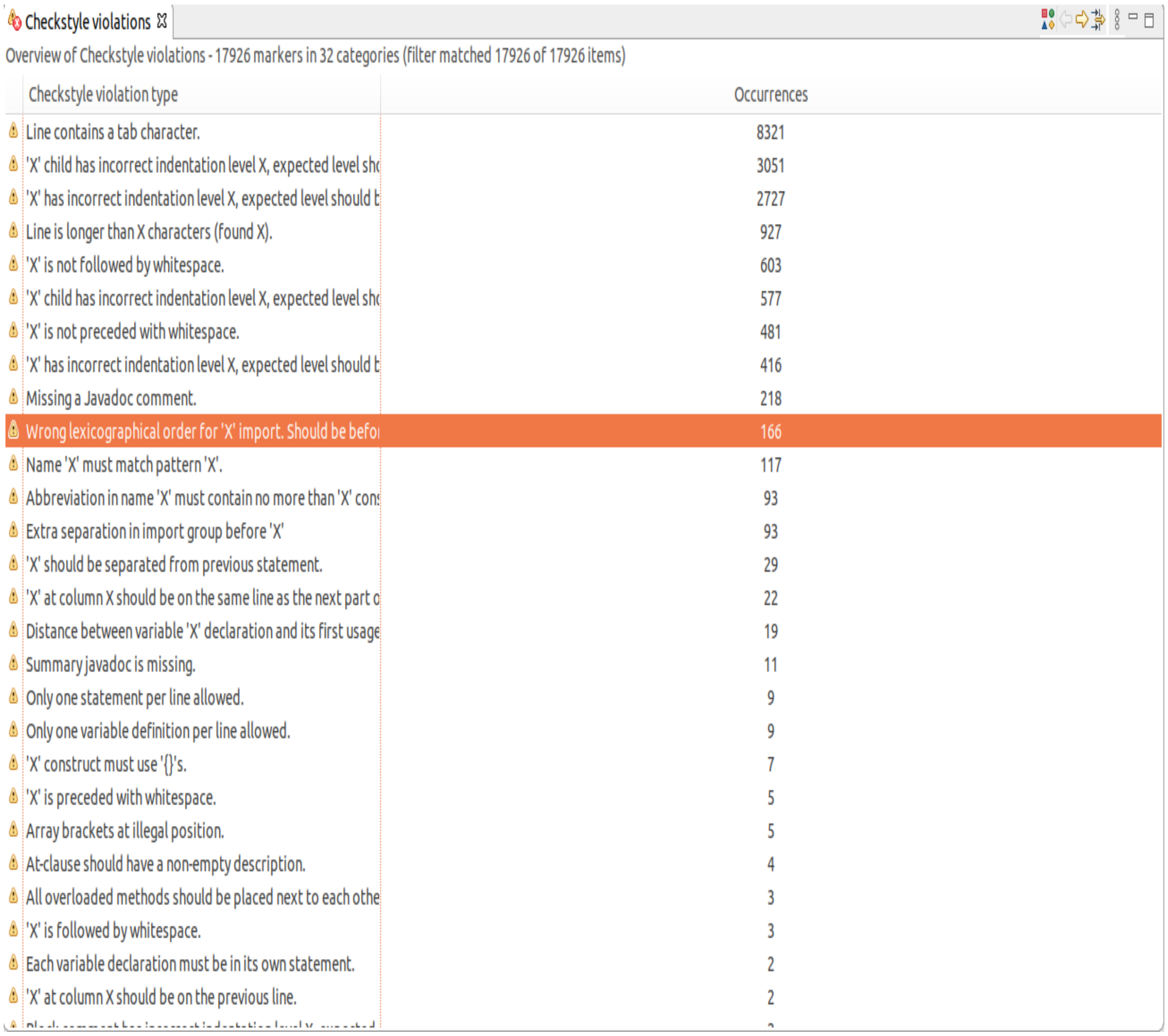
The code had complicated if statements which were extracted to separate methods in order to make the code more readable and less complex. The indentation of the code was fixed where code was not properly structured and in some cases no white spaces were followed after variable declaration or object creation. Many variable and class names were misleading and it was not obvious what purpose they served in the code so we did a lot of refactoring in order to give more proper names.

Furthermore, some methods was extremely long, they were split into smaller methods or even classes with discrete functionality, splitting up possible "God Class" cases such as the Gui class. Some methods could be moved to another class or package, so we applied method extraction techniques.

Finally, there was a lot of duplicate code, so we spent effort and time in removing duplicate code or methods with same functionality, but located in different classes. Dead code or code in comments was also removed.

STYLE VIOLATIONS

CheckStyle Violations (after refactoring)



Checkstyle violations

Overview of Checkstyle violations - 17926 markers in 32 categories (filter matched 17926 of 17926 items)

Checkstyle violation type	Occurrences
Line contains a tab character.	8321
'X' child has incorrect indentation level X, expected level sho	3051
'X' has incorrect indentation level X, expected level should b	2727
Line is longer than X characters (found X).	927
'X' is not followed by whitespace.	603
'X' child has incorrect indentation level X, expected level sho	577
'X' is not preceded with whitespace.	481
'X' has incorrect indentation level X, expected level should b	416
Missing a Javadoc comment.	218
Wrong lexicographical order for 'X' import. Should be befor	166
Name 'X' must match pattern 'X'.	117
Abbreviation in name 'X' must contain no more than 'X' cons	93
Extra separation in import group before 'X'	93
'X' should be separated from previous statement.	29
'X' at column X should be on the same line as the next part o	22
Distance between variable 'X' declaration and its first usage	19
Summary javadoc is missing.	11
Only one statement per line allowed.	9
Only one variable definition per line allowed.	9
'X' construct must use '{}'s.	7
'X' is preceded with whitespace.	5
Array brackets at illegal position.	5
At-clause should have a non-empty description.	4
All overloaded methods should be placed next to each othe	3
'X' is followed by whitespace.	3
Each variable declaration must be in its own statement.	2
'X' at column X should be on the previous line.	2
Block comment has incorrect indentation level X, expected	[illegible]

OPEN ISSUES

While refactoring the source code of “Plutarch Parallel Lives” tool we put effort in splitting up responsibilities of classes that possibly acted as “God Classes”. At the end of the refactoring process we tried to create a factory at the data package for the subsystems composing the GlobalDataManager. We tried to create an extra level of abstraction, more specifically an interface at datakeeper but at process of this step we discovered that (possibly in our implementation), no similarity existed between the subsystems of the GlobalDataManager and as a result it was not possible for us to create an interface or factory to simplify the processing logic of the data by creating the corresponding subsystem through the factory class. We tried to create as less instances of those subsystems as possible whenever they need to be used in order not to add any more dependencies between classes of the project and simplify the architecture. Another fact is that duplicate code still exists, though in much lesser quantity as in the beginning, some methods could still be simplified e.g., where description are generated to display on the tables. Refactoring the code can surely be a process that actually never ends.

P.S. Thank you for your guidance, we had a great time at class lectures even under the given circumstances of the Corona virus incident.