HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

LOGIC EXPRESSION NORMALIZER

FINAL PROJECT REPORT

Object Oriented Languge and Theory - Lab

lecturer:

Nguyen Thi Thu Trang, Ph.D.

group 3:

Tran Trung Hieu – 20194764

To Xuan Hung – 20194772

Nguyen Tran Minh Tuan - 20194877

June 2nd, 2021

Table of Contents

[**A.** **Description** 4](#_Toc73280779)

[I. Mini-project requirements 4](#_Toc73280780)

[1. Overview: 4](#_Toc73280781)

[2. Specifications: 4](#_Toc73280782)

[II. Use case diagram 4](#_Toc73280783)

[**B.** **Design idea** 6](#_Toc73280784)

[I. General class diagram 6](#_Toc73280785)

[II. Class diagrams and explanation 7](#_Toc73280786)

[1. Minterm package: 7](#_Toc73280787)

[2. Column package: 8](#_Toc73280788)

[3. Table package: 9](#_Toc73280789)

[4. Output package: 11](#_Toc73280790)

[5. Controller package: 13](#_Toc73280791)

**MEMBERS ASSIGNMENT**

|  |  |  |
| --- | --- | --- |
| **Full name** | **Student ID** | **Assignments** |
| Tran Trung Hieu | 20194764 | * Package: column, table * Controller and GUI: Main menu * Report |
| To Xuan Hung | 20194772 | * Package: minterm * Controller and GUI: Input interface * Report and Slides |
| Nguyen Tran Minh Tuan | 20194877 | * Package: output * Controller and GUI: Output interface * Report |

# **Description**

## Mini-project requirements

### Overview:

Create an application to normalize a logic expression using the implementation of the Quine-McCluskey method. The final expression must have the least number of terms. The fewer the number of implicants are, the higher reliability and lower manufacturing cost. This application accepts three-variable or four-variable inputs without don’t care terms.

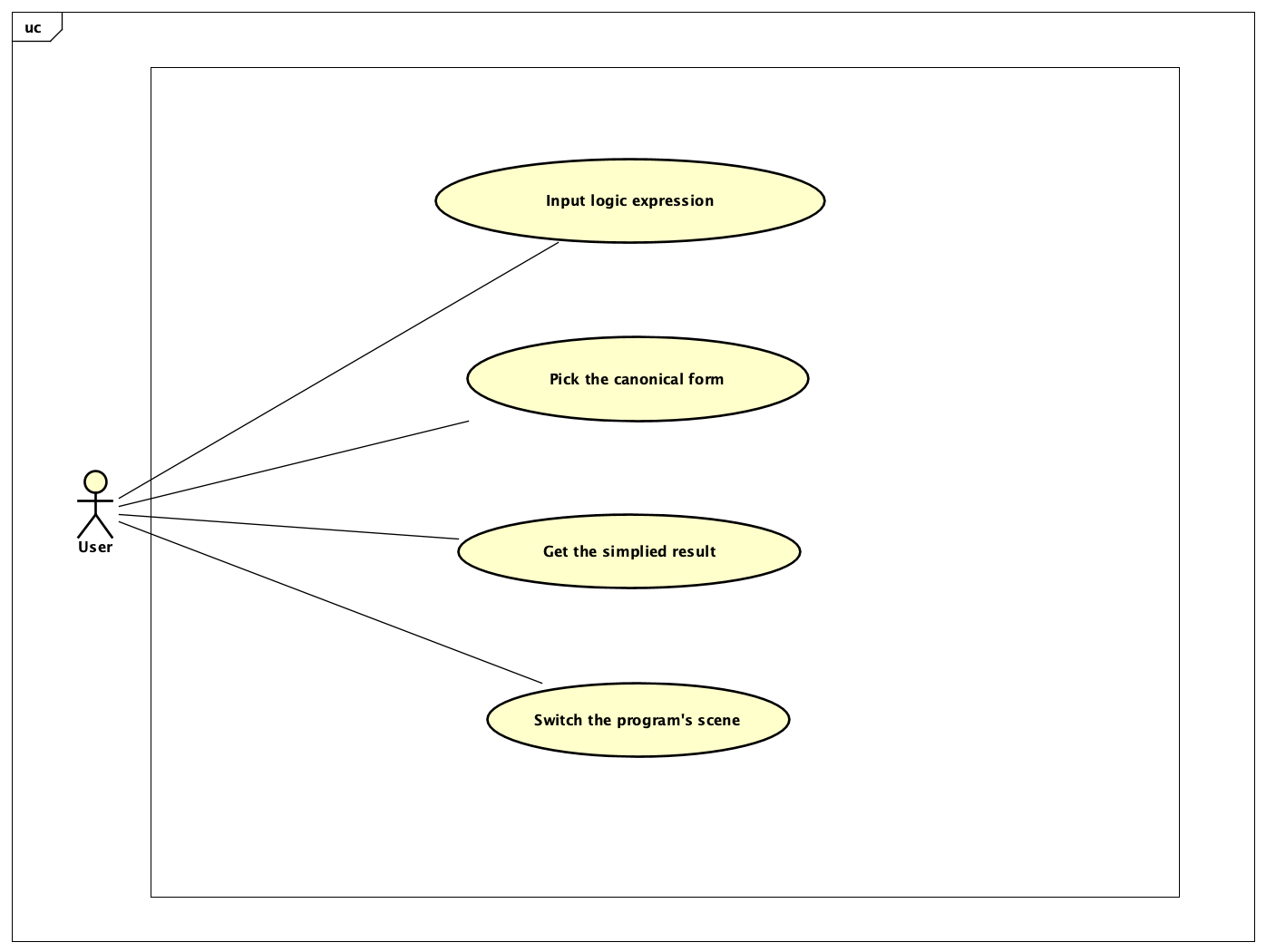
### Specifications:

This application receives a boolean expression from the user through an interface simulating a truth table. The user interface includes:

* Main menu: The user can choose one of the two cases 3 variables or 4 variables expression from the navigation bar.
* Input interface: The user can input a logic expression through a truth table. Don’t care values are not allowed. They can also pick a canonical normal form for the simplified expression – either SOP (Sum of products) or POS (Product of Sums). After the user has finished picking values for the truth table, they can press submit to see the output. *The user can also choose to reset all the inputs to 0 and choose to go back to the Main menu*.
* Output interface: The application will show to the user the intermediate table (contains intermediate columns), the PI table, *the make equation table (contains EPIs and their character equations)* and the final simplified expression.

## Use case diagram

* Get the simplified result including user submits the selected input, and the result is shown up after.
* Pick the canonical form including user select the canonical result form ( in SOP or POS).
* Inputing logic expression includes user choose a sequence of 0s or 1s through the truth table, in which 0 means not chosen and 1 means chosen.
* Switch the program scene including user click the button back or select from the menubar option (3-variable, 4-variable, home button).



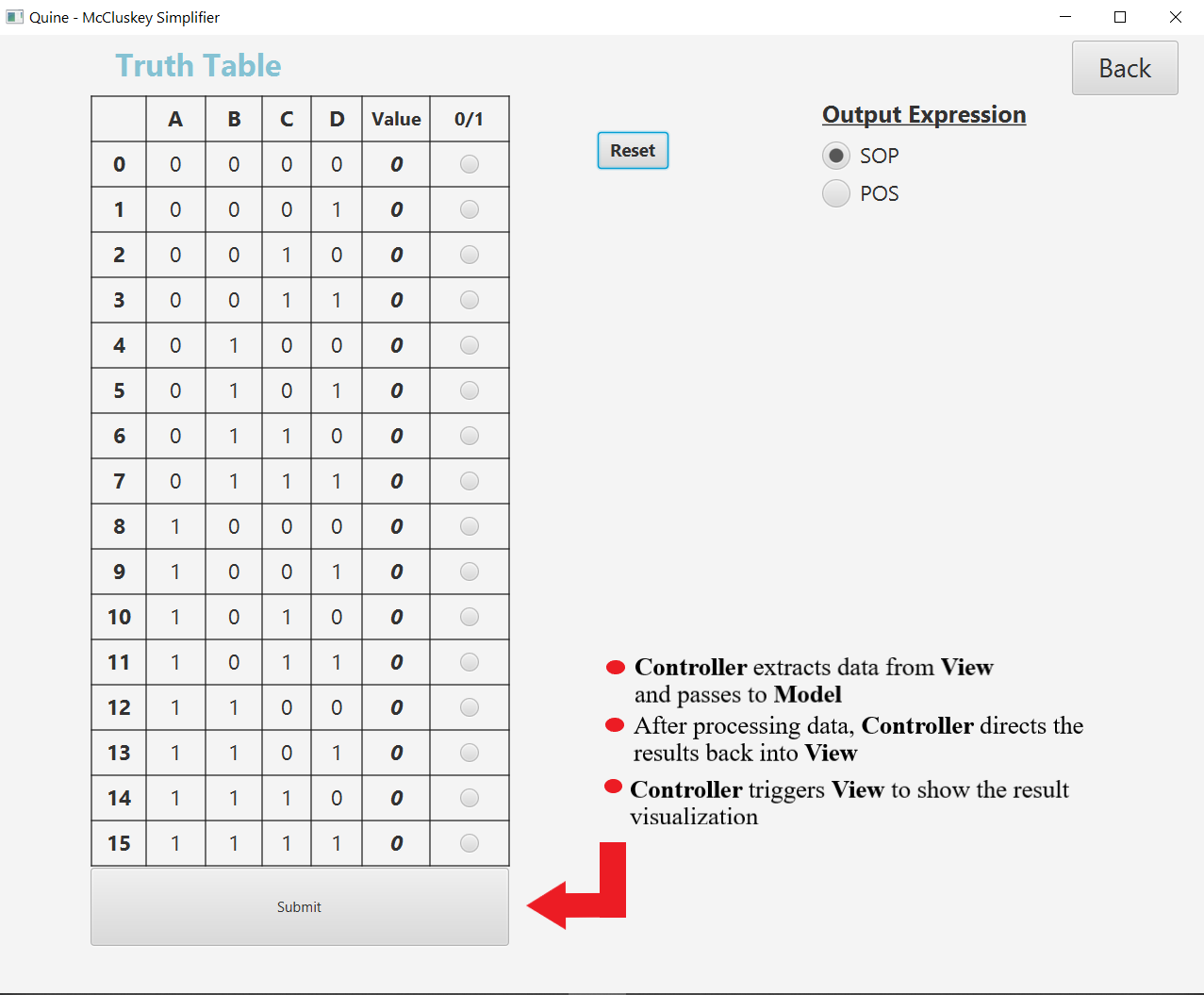
*Figure 1. Logic expression normalizer use case diagram*

# **Design idea**

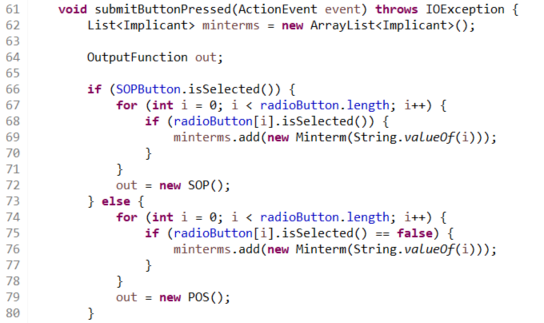
## The Model – View – Controller design patterns

In this mini project, we decided to design our program based on the MVC Pattern or Model – View – Controller Pattern. By this way, the code is organized into different sections with their own purposes and the code will be much easier and cleaner for others to review.

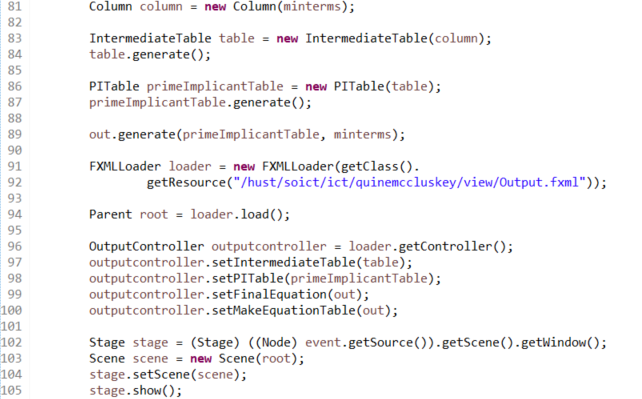
The **Model** is a package that holds raw data and all components used for the Quine-McCluskey algorithm. In the **View** package, there are several FXML files and each file name represents for a specific scene in JavaFX GUI. Simply this package is used to show the visualization of the data that the **Model** package contains. The last of three is **Controller** packge. This acts as a connection between the **Model** and **View** package. It controls the data flow input from user, directs into **Model** to process and updates the **View** whenever data changes separately.



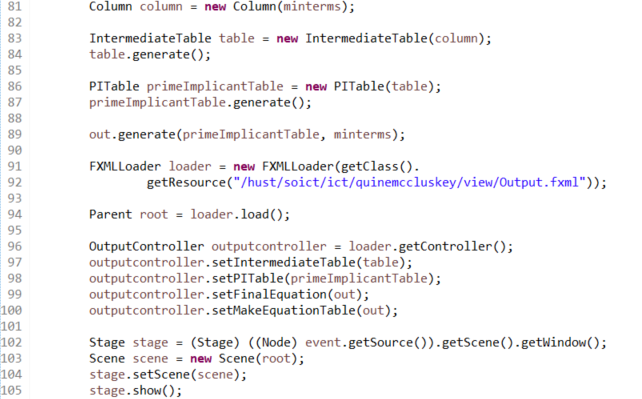
*Figure 1. The connection between Model – View - Controller*



*Figure 2. Extract data from View*

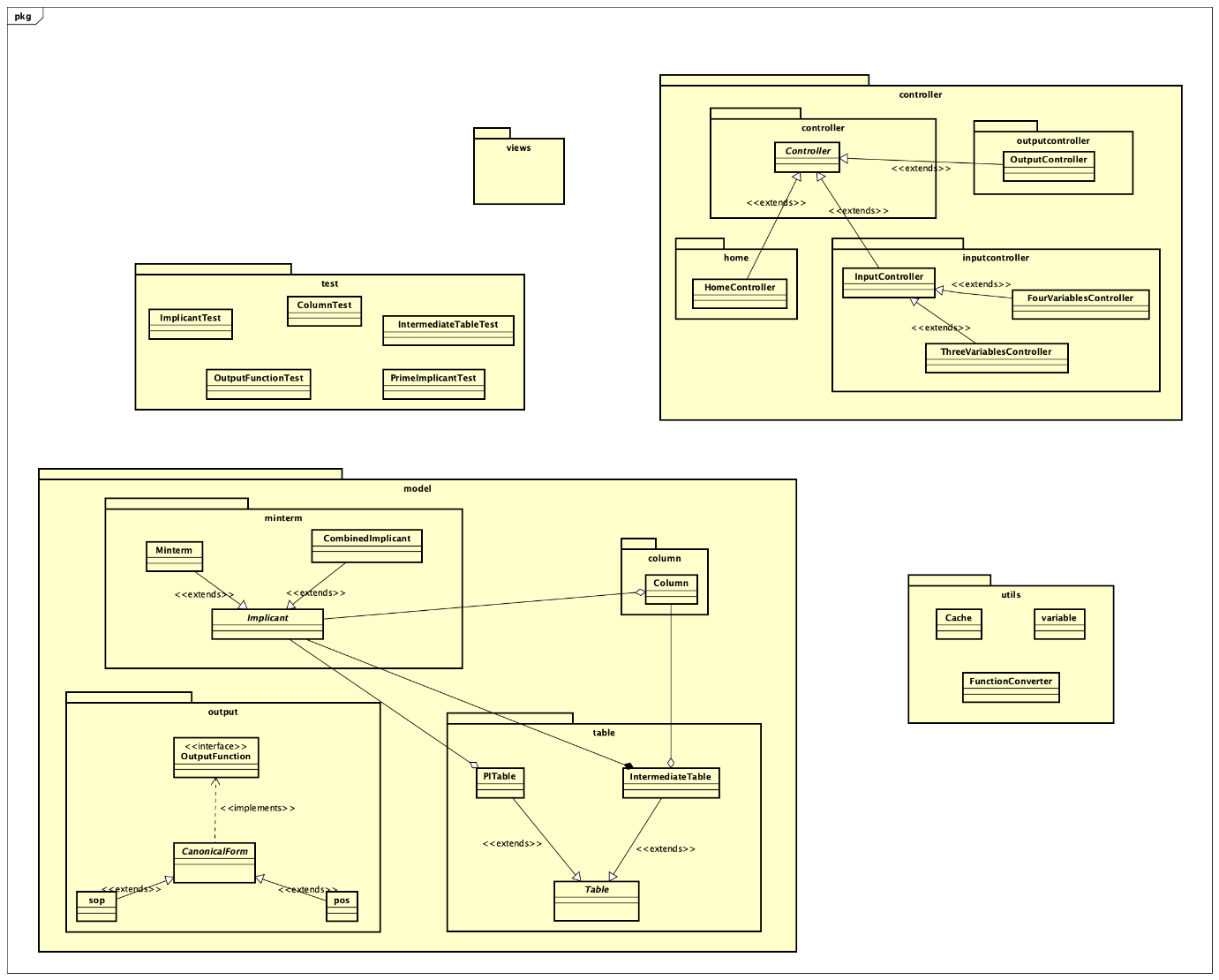


*Figure 3. Process data in Model*



*Figure 4. Direct results and trigger View to show data visualization*

## General class diagram



*Figure 5. Logic expression normalizer class diagram*

A column possibly contains implicants, the intermediate table contains one or more columns. Based on that idea, the aggregation relationship is used among most of the classes, specially among the Implicant class, the Column class, the PITable and the IntermediateTable class. However, the IntermediateTable must contain at least one column, so the relationship between those two is a strong composite relationship.

The Implicant class is a parent class which is inherited by the Minterm class and CombinedImplicant class since they have same atributes of an implicant.

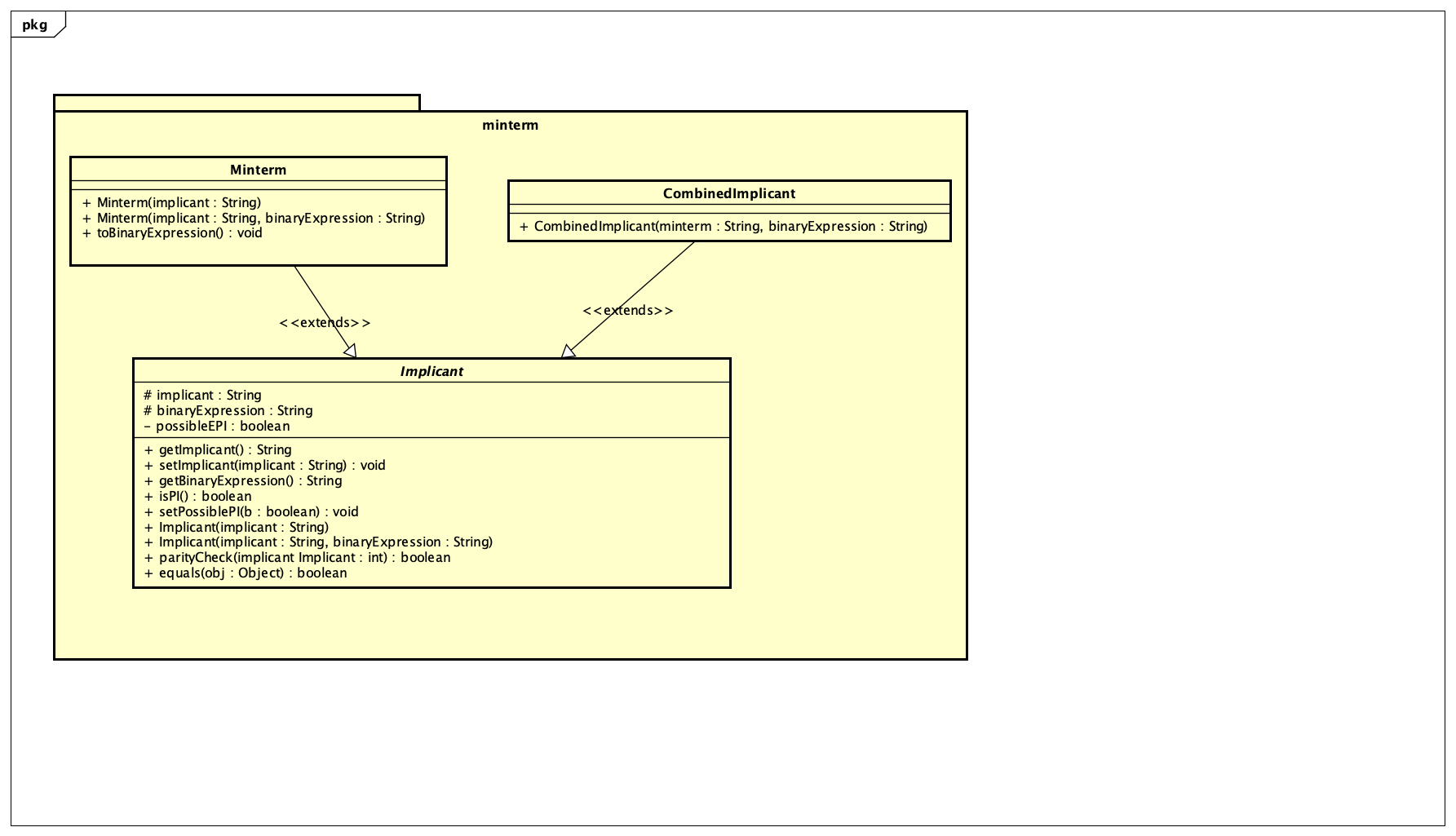
There are also two types of tables, the IntermediateTable and PITable. Both inherit the Table class.

The user can choose two types of the final canonical expression, POS or SOP. Therefore, the SOP and POS class should inherit the OuputFunction class and itself override the method **generate()** in order to normalize the input expression.

As three views main menu, input interface and output interface all include some buttons to switch to another view, we create a parent class Controller to store that Button variable and functions to handle the event in which the button is clicked. So, the child classes can inherit this button and its functions.

## Class diagrams and explanation

### Minterm package:

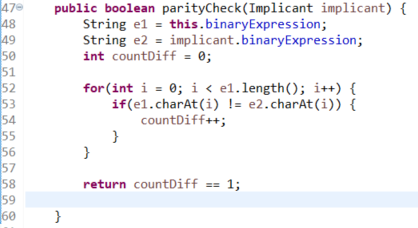


*Figure 6. Minterm package’s class diagram*

This package consists of three classes:

* Implicant class: This is an abstract class defines the common behaviors shared between Minterm (the initial implicant which is represented in its binary expression) and CombinedImplicant (the implicant obtained from combining some Minterm pairs, represented by their binary expression and a symbol “-” where there is a change in one-bit position). This class contains the constructor to create objects with Minterm and CombinedImplicant which will be used alongside the other two classes within this package throughout the program. This class implements the method **parityCheck(Implicant** implicant**)** to examine whether the current implicant can be combined with the implicant argument or not. Besides, there is a method override the **equals(Object** obj**)** method to compare the equality of two implicants.
* Minterm class: This class inherited the Implicant class to use its constructors and all of its methods. The Minterm class implements a method called

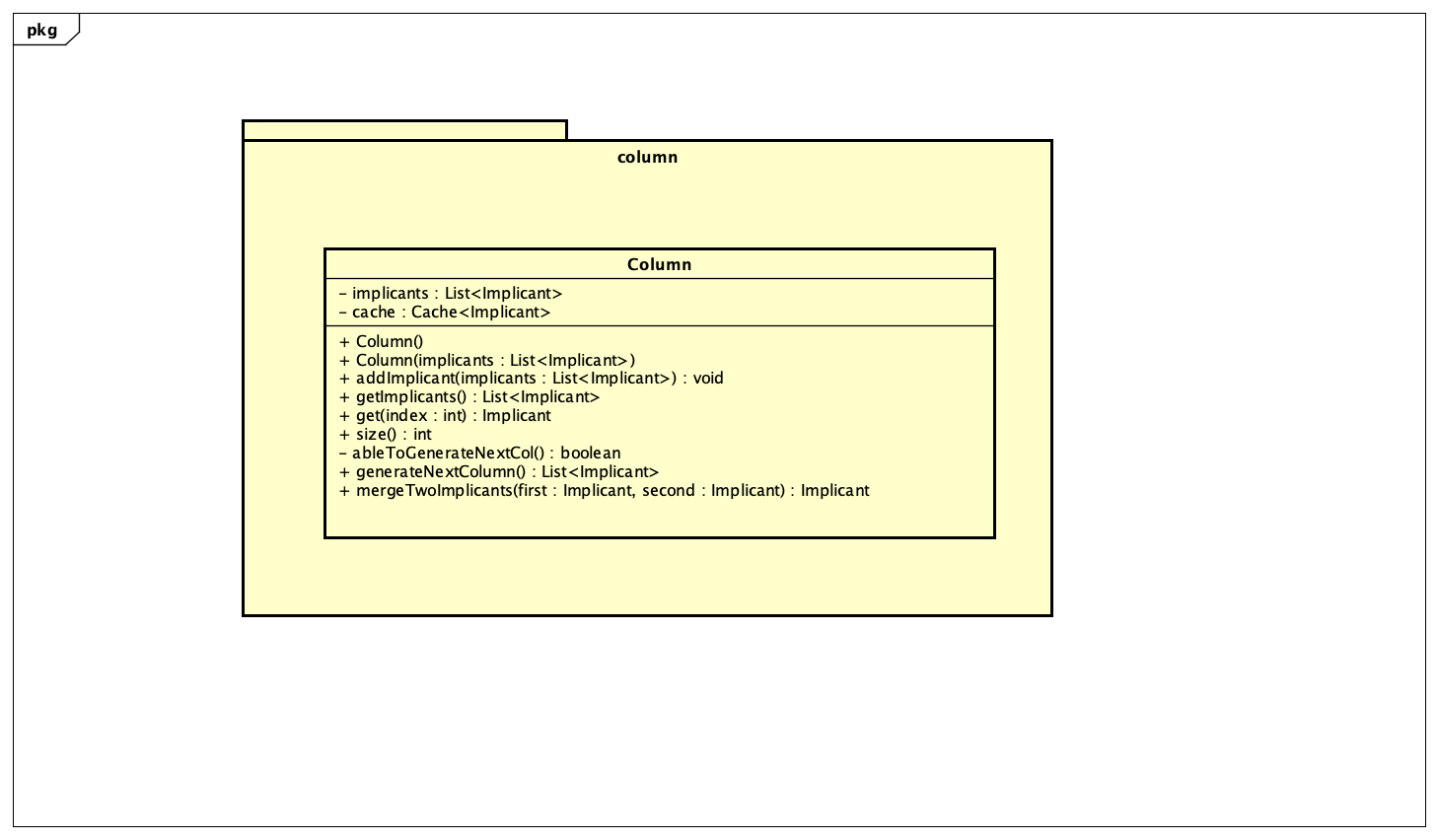
**toBinaryExpression()**, which is used in the constructor of the Implicant class when Implicant is an instance of Minterm to convert the decimal input into its binary expression.



*Figure 7. parityCheck() implementation*

* CombinedImplicant class: this is a subclass inherits the Implicant class to demonstrate the combined implicant and its combined expression.

### Column package:



*Figure 8. Column package’s class diagram*

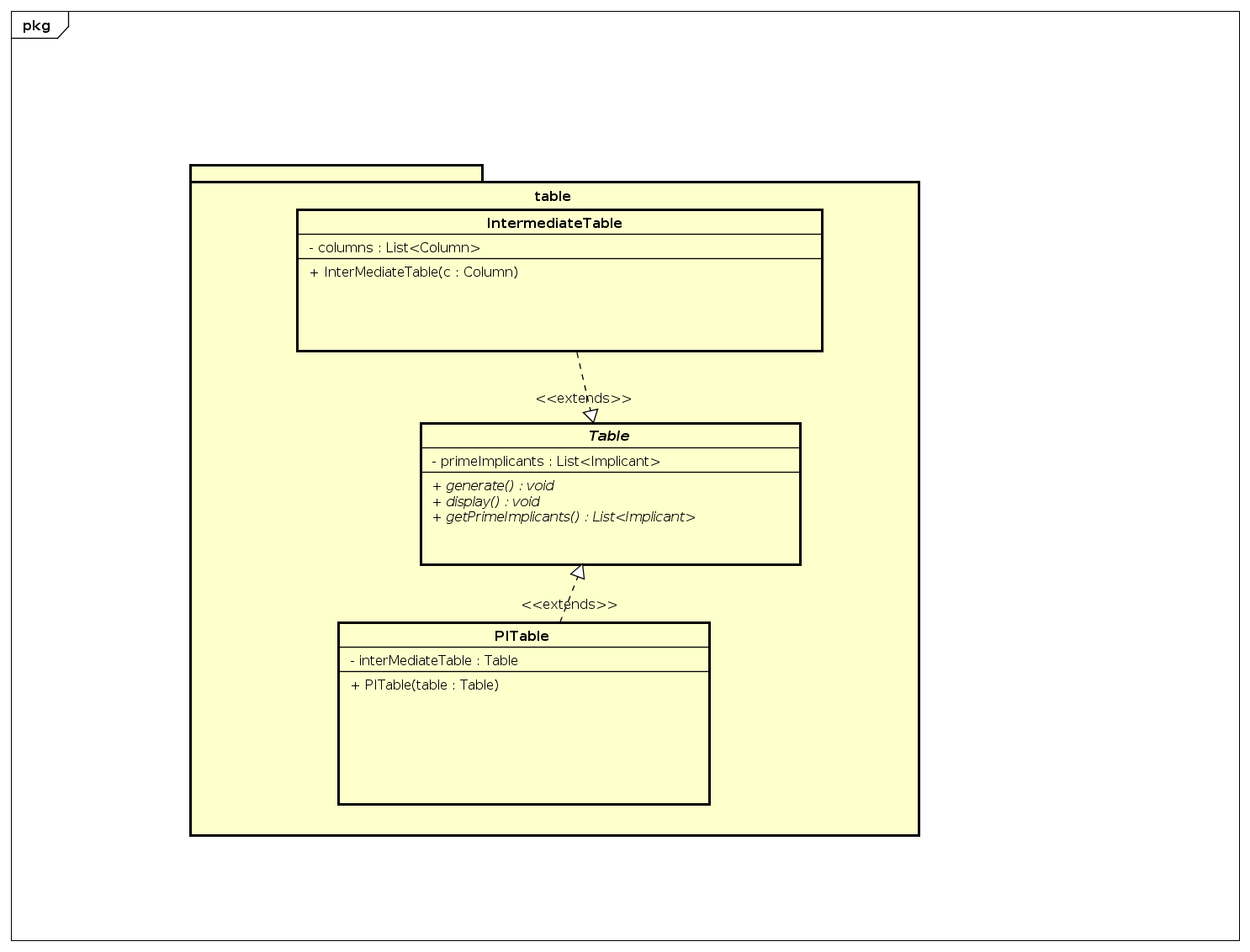
A column is a group of implicants 🡪 we use aggregation property.

**The addImplicant()** is an overloaded function. It accepts a single Implicant or a List<Implicant> as an argument. This is where **polymorphism** works. As mentioned before, we have two child classes of the parent class Implicant (these two classes are **Minterm** and **CombinedImplicant).** Hence, when we pass an argument to this function **addImplicant()**, the argument can be bounded by **Minterm** or **CombinedImplicant** (note that we cannot pass an **Implicant** argument because this class is abstract).

The heart of this class is the combination of 2 classes: generateNextColumn() and mergeTwoImplicants()

* **generateNextColumn()**: using double loop, traverse through all implicants in **List<Implicant> implicants**. Then, it checks for parity possibility of 2 random implicants (to check parity, it will call **parityCheck()** defined in Implicant class). If **parityCheck()** returns true, then 2 implicants can be merged, and then call function **mergeTwoImplicant()**. Otherswise, it continues its loop.
* **mergeTwoImplicants()**: accepts 2 Implicants as 2 arguments, returns an Implicant object obtained from merging 2 arguments. These 2 arugments are able to be merged if there is only one diffrence in their binaryExpression (again, binaryExpression is already defined in class Implicant). For example, binary expression 0011 can be merged with 0010 but not with 0110.

### Table package:

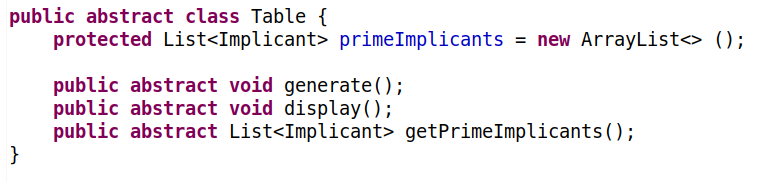


*Figure 9. Table package’s class diagram*

There are two types of table: Intermediate Table (which contains multiple columns), and Prime Implicant Table (which takes all prime implicants from Intermediate Table to generate a final result).

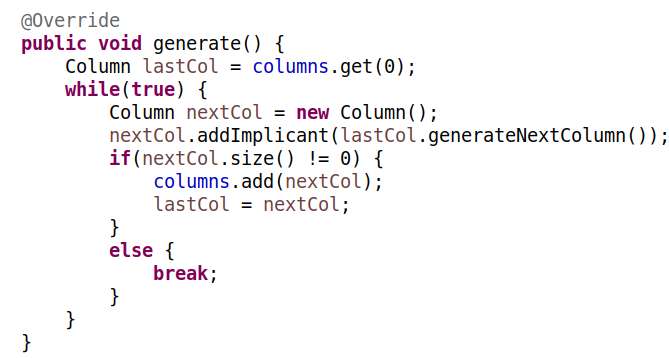
Each table has functions: **generate()**, **getPrimeImplicants()**, and **display()**. Note that the fucntion **display()** is only used for command-line interface program. Although we have moved to GUI, we still decide to keep that function for debugging. Hence, it is better and more "OOP" to have an abstract ckass called Table, and let these 2 tables extend:

* Abstract class Table:



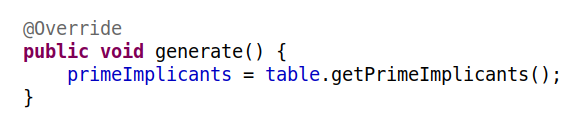
*Figure 10. Table class*

* This class contains a protected variable because its child classes will use it: **List<Implicant>** primeImplicant.
* Besides, we have two abstract classes **generate()** and **getPrimeImplicants()**. Its child classes must take responsibility to define these functions.
* Class IntermediateTable:
* This class contains a list of columns List<Column> columns.
* The overrided function **generate()** executes an infinite loop. In each loop, it will generate a new column by calling function **generateNextColumn()** through the last element in the list columns. This infinite loop will only be broken when there is no column generated.



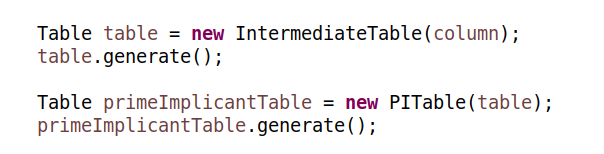
*Figure 11. Override the method generate() in IntermediateTable class*

* Class PITable:
* This class has a variable Table table. Note that Table is an abstract class, hence it will be binded by its child classes.
* The **generate()** function simply calls table.getPrimeImplicants() (this function is an abstract function as mentioned above). This is where polymorphism is exploited. If table is binded by type IntermediatTable, then this function will call IntermediateTable::**getPrimeImplicants()**. Otherwise, it will call PITable::**getPrimeImplicants()**, the same getter method.



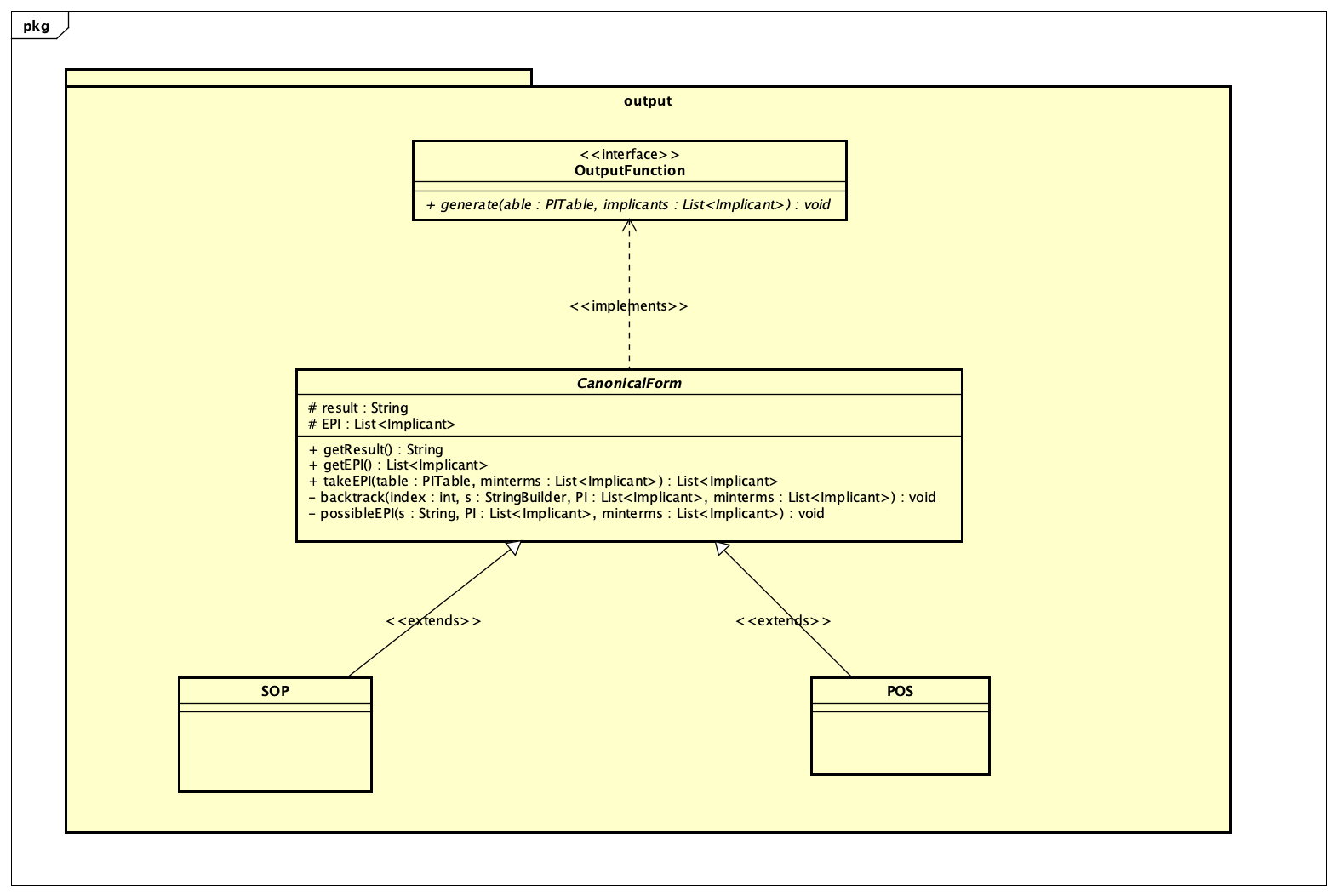
*Figure 12. Override the method generate() in PITable class*

The function of **generate()** is decided only when the program is running. This is where we obtained run-rime polymorphism.



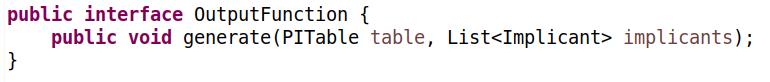
*Figure 13. generate() is used in InputController*

### Output package:



*Figure 14. Output package’s class diagram*

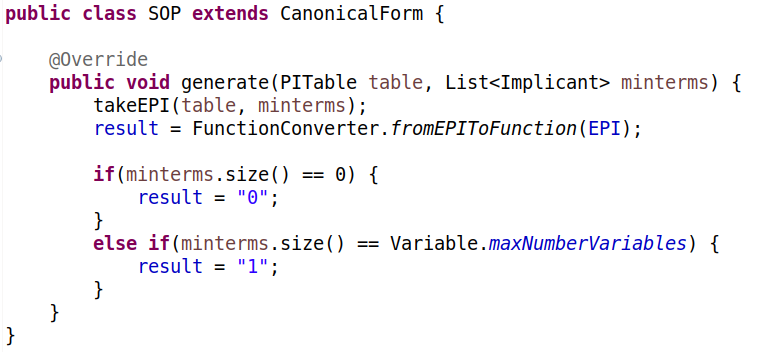
The outputfunction package is directly responsible for the makeEquation table and the final normalize expression. In this package, we have an interface OutputFunction and 2 classes SOP and POS which extend an abstract class called CanonicalForm. Below is the method of the interface OutputFunction:



*Figure 15. generate() method in OutputFunction*

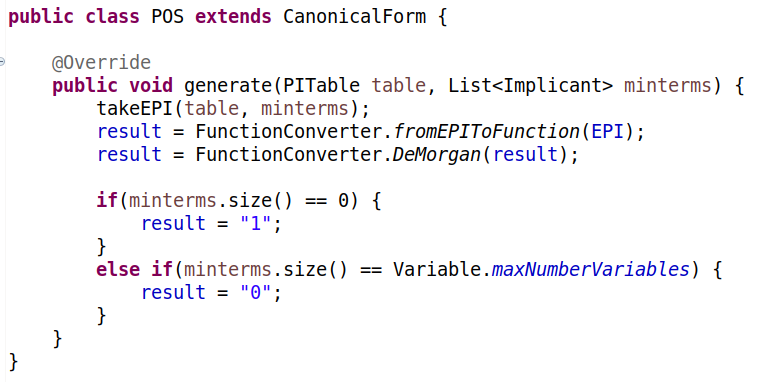
Here is the overriden *generate()* method in 2 subclass that extends CanonicalForm:

* In SOP:



*Figure 16. generate() implementation of SOP*

* In POS:



*Figure 17. generate() implementation of POS*

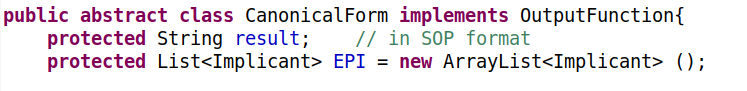
The method **generate()** is decide at runtime. In the InputController class, depending on whether user chooses SOPButton or POSButton, method **generate()** of either class will be invoked correspondingly

Description: Graphical user interface, text, application

Description automatically generated

*Figure 18. inside the submitButtonPressed() method*

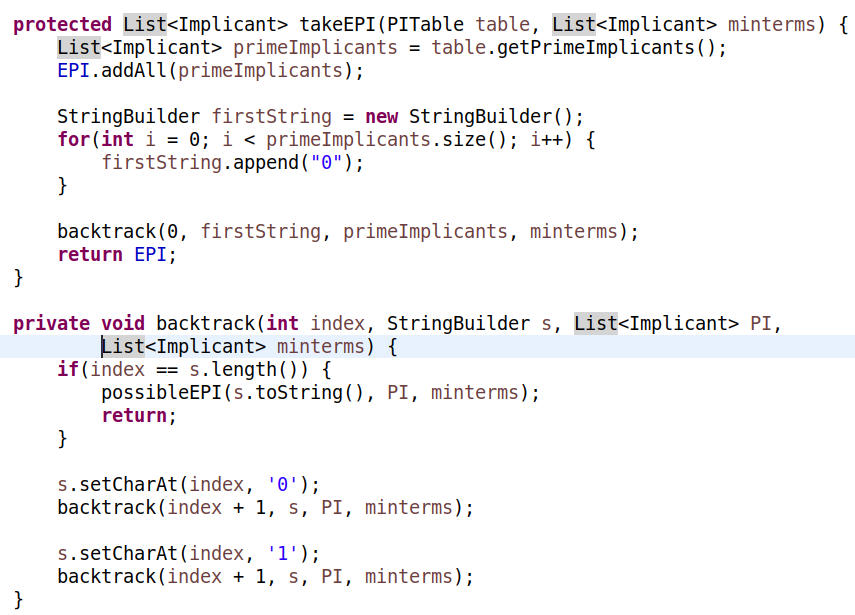
* OutputFunction interface:
* There is a method, namely **generate()** simply to invoke class to be used in the InputController class.
* CanonicalForm class:



*Figure 19. CanonicalForm abstract class*

Note that we do not override the abstract function **generate()** of OutputFuction interface. This is because all of its children must have different **generate()** functions. So all child classes have to override this method themselves.

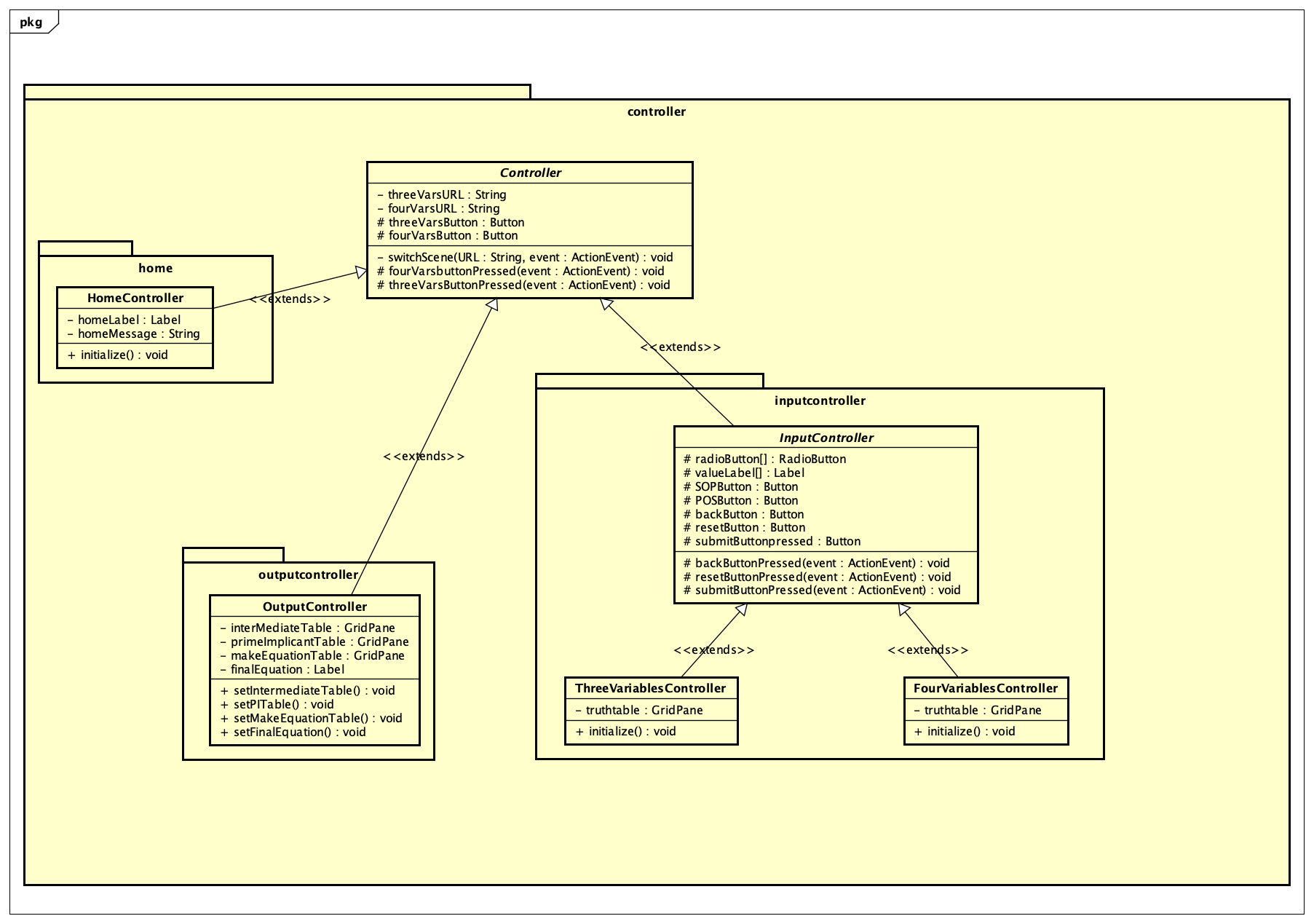
* The two protected attributes **String result** and **List<Implicant> EPI** are used with the intention of re-using these 2 properties in both the abstract super class and its 2 subclasses.
* The most crucial tasks of OutputFunction class are taking possible EPI from the PITable then providing algorithms to minimalize logic expression which is executed by the **takeEPI()** method.
* **takeEPI()** method: Uses backtracking algorithm to check the coverage of the possible EPI, meanwhile sort out the most fitted one out of those chosen (EPIs that has the fewest implicants).



*Figure 20. Output package’s class diagram*

* SOP and POS class: When the user selects the canonical form (SOP or POS), either one of these 2 classes will be used to invoke an object in the InputController class (check out the InputController class description for more detail).
* If the POS class is invoked, the program performs the takeEPI method in the abstract class and transforms the minimalize implicants into normalize (character) expression. The remaining task is applying DeMorgan 's Law through the DeMorgan method to get the POS final expression.
* If the SOP class is invoked, the program does the same thing as the POS class except this time the **DeMorgan()** method are unused.

### Controller package:



*Figure 21. Controller package’s class diagram*

The abstract class Controller has a switch scene method that transition to the three variables input scene when the 3-variable button is pressed by user or to the four variables input scene when the 4-variable button is pressed.

Both these buttons are indispensable parts of all scenes since all of the class below extends this abstract super class.

* Controller.home package:

Only comprises of the HomeController class. Other than inheriting 2 buttons, HomeController also has an **initialize()** method to show the program manual for user.

* Inputcontroller package:

ThreeVariablesController and FourVariablesController that extends the parent class InputController. Inside the InputController class, there are integrated buttons, labels that appear on the user interface and several listeners to their pressed button.

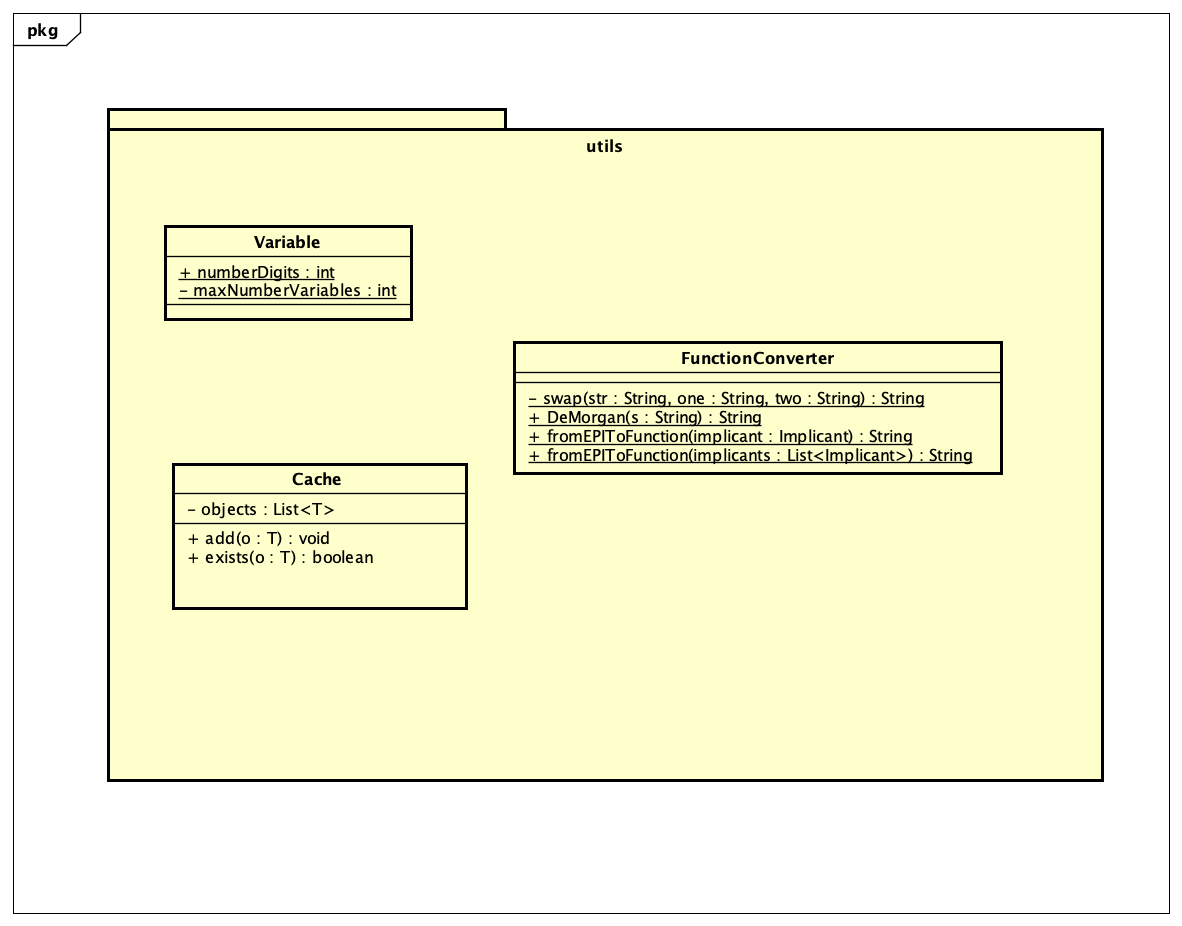
* **backButtonPressed()**: the scene is switched to home screen.
* **submitButtonPressed()**: handle all the processes that happened after the submit button is pressed.

The core principle of this method is when the SOP button is pressed, all the expression that are set to 1 are directly used as minterm. On the other hand, if the POS button is pressed, all the remaining expressions (expressions that have the value of 0) are used as minterm for further investigation.

* Outputcontroller package:

OutputController class involves 4 methods: **setIntermediateTable(), setMakeEquationTable()**, **setPITable()**, **setFinalEquationTable ()**.

### Util package:



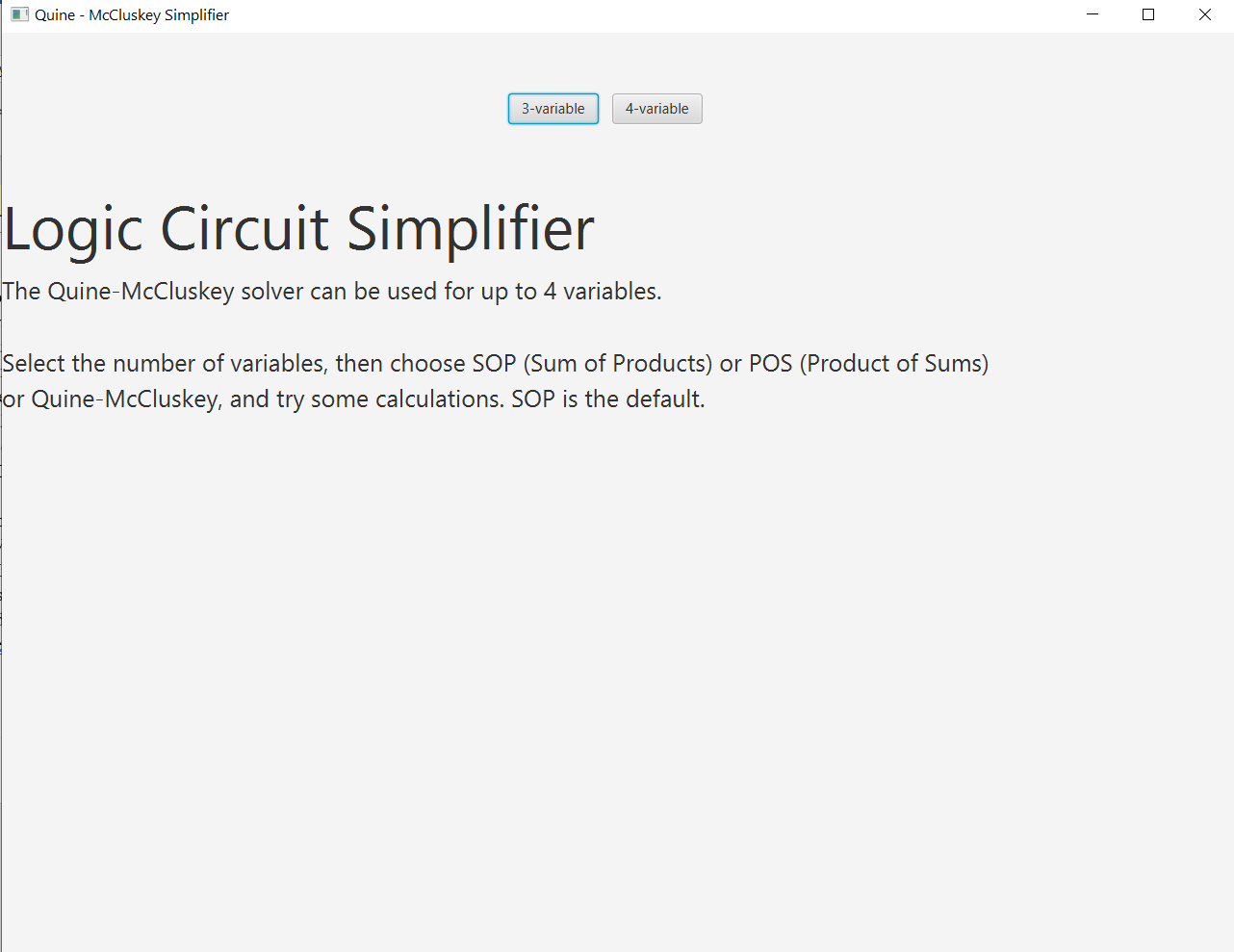
*Figure 22. Util package’s class diagram*

This package consists of 3 classes:

* Variable class: Contains the number of digits of an implicant and the maximum numbers of implicants can reach.
* Cache class: Stores temporary variables. We applied generic programming so the Cache class can cast with any type of object.
* FunctionConverter class: Implements the method **DeMorgan()** and a method to convert the EPIs to string represention.

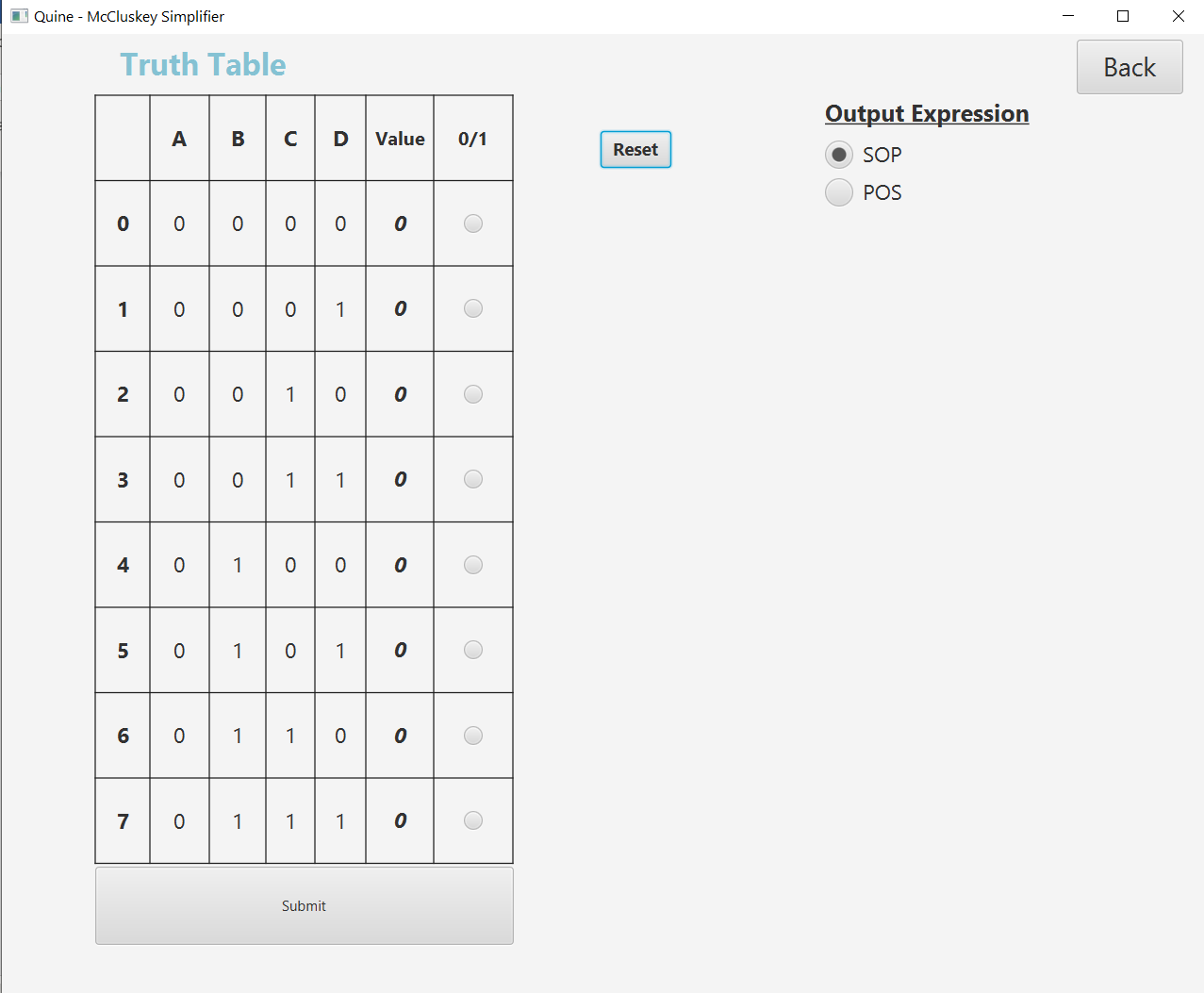
### View package:

This packages only contains FXML files to visualize the GUI through JavaFX using SceneBuilder.

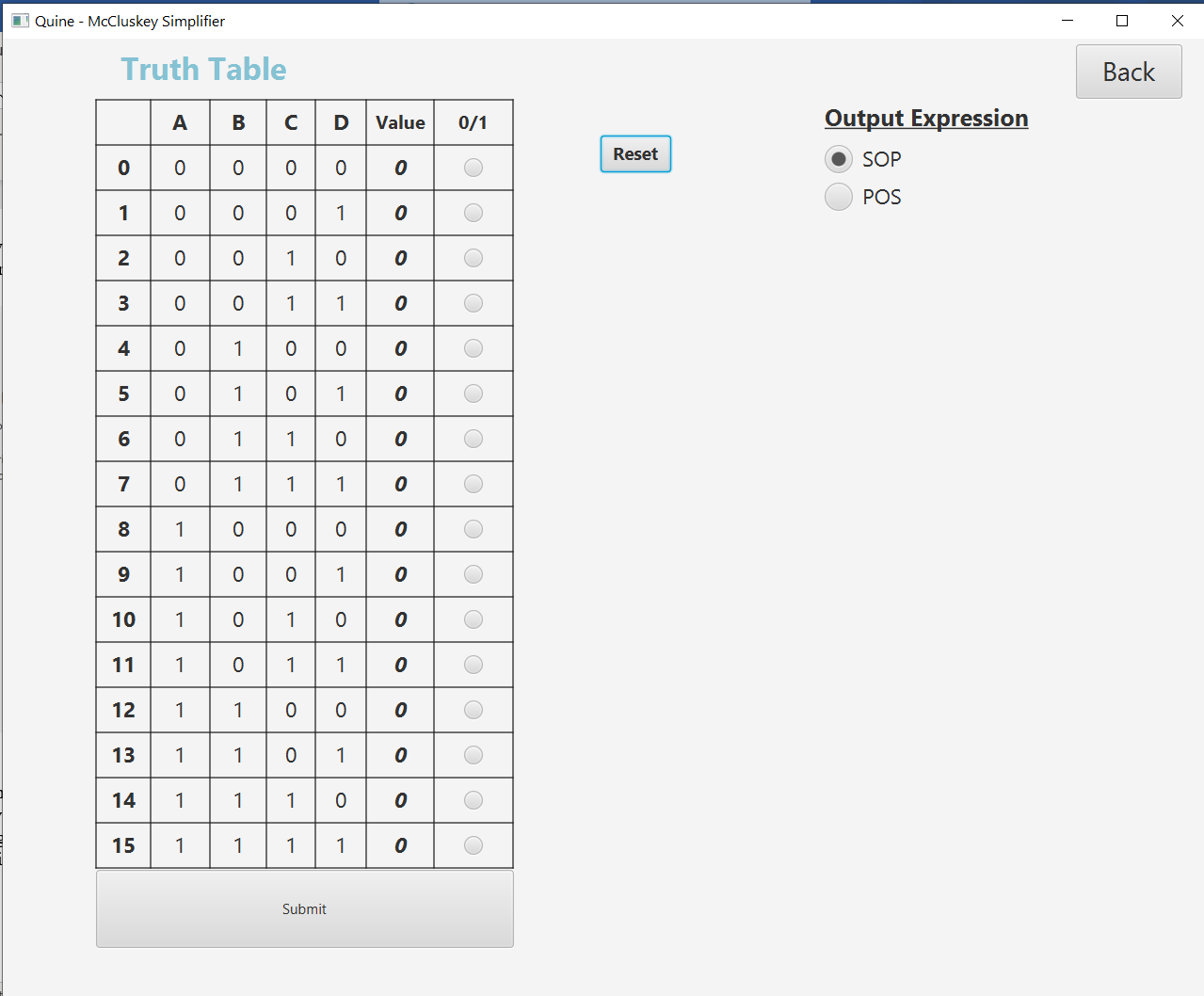


*Figure 23. Main menu*

When the user runs the program, a main menu appears on the screen as a 1080x720 resolution window. When the user clicks on the 3-variable or 4-variable button, the program will trigger switching scene to direct the user to the corresponding input interface as following:

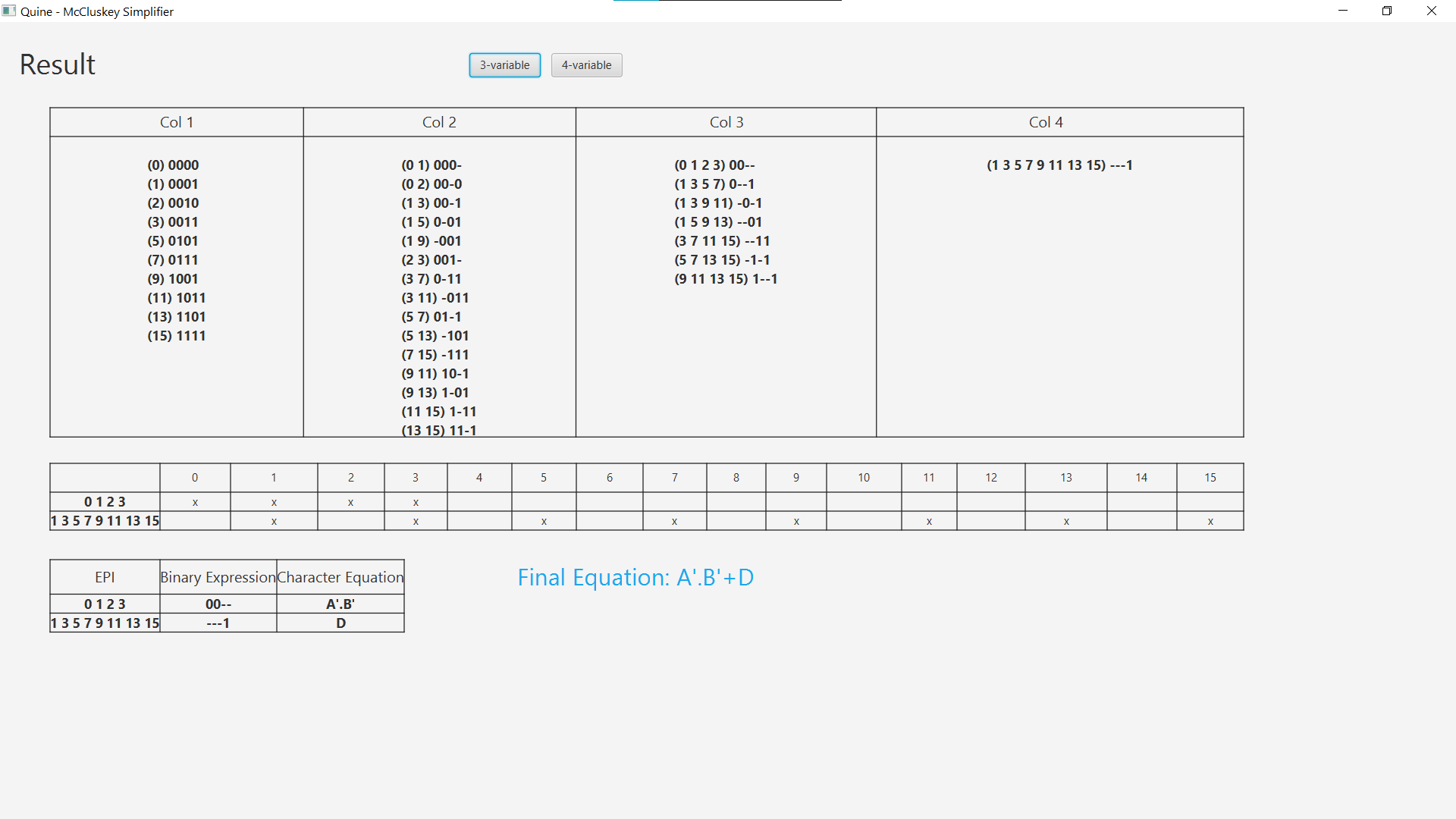


*Figure 24. 3-variable input interface*

**

*Figure 24. 4-variable input interface*

All the components inside these two scenes will share the same listener which are implemented in InputController claas, the parent class of ThreeVariablesController and FourVariablesController as mentioned in section 5. The output interface appears after the user press the “Submit” button.



*Figure 24. Output interface*

The frame size of the output interface above can alternate depending on the size of its components.