Hanoi University of Science & Technology

School of Information & Technology

======================

Object-Oriented Programming

Project Report

***Logic Expression Normalizer***

Students:

1. Le Quang Hieu 20210335
2. Pham Trung Hieu 20214899
3. Hoang Minh Hoang 20214900
4. Phan Duc Hung 20214903

Class: 141177

Supervisor: Prof. Nguyen Thi Thu Trang

*Table of Contents*

1. **Assignment of members**
2. **Mini-project description**
3. Requirements
4. Use case Diagram
5. **Design**
6. General Class Diagram
7. Algorithm Class Diagram
8. User Interface Class Diagram
9. Explanation of Design

**IV**. **Result**

1. Result
2. *Assignment of members*

Assignments:

1. Design

+) General class diagram

+) Algorithm class diagram

+) Solver Diagram

+) Use Case Diagram

+) User Interface Diagram

1. Sourcecode

* Solver: class Solver
* ui:

1. class ‘Choice\_Panel’
2. class ‘Input\_Panel’
3. class ‘Output\_Panel’
4. class ‘GUI\_Frame’
5. class ‘Main’

* algorithm:

1. class ‘MinTerm’
2. class ‘MinTermGrouper’
3. class ‘ShowColumn’
4. class ‘TransformColumn’
5. package ‘algorithm.column’:

* class ‘Column’
* class ‘FirstColumn’
* class ‘IntermediateColumn’

1. Demo video
2. Slide
3. Report

| Name | Student’s ID | Tasks |
| --- | --- | --- |
| Phan Duc Hung | 20214903 |  |
| Le Quang Hieu | 20210335 | GUI Implementation. |
| Pham Trung Hieu | 20214899 |  |
| Hoang Minh Hoang | 20214900 |  |

1. *Mini-project Description*
2. Requirements
3. What is Quine-McCluskey method?

The quine-McCluskey method also called the **tabulation method** is a very useful and convenient method for simplification of the Boolean functions for a large number of variables (greater than 4). This method is useful over K-map when the number of variables is larger for which K-map formation is difficult. This method uses prime implicants for simplification.

In this method, we construct multiple tables according to the question and at the last, we make a prime implicant table which is used to obtain essential prime implicants which are present in the simplified boolean expression. This method requires prior knowledge of decimal to binary representation and the basics of boolean algebra. It is a suitable method for a large number of input variables which can be easily solved by this method but the computation complexity is high. Majorly, this method includes the use of minterms, and prime implicants and obtains essential prime implicants which are further used in the simplified boolean functions.

***\*Step for Quine-McCluskey method:***

1. From the given minterms, arrange them according to the number of 1 presented in the binary representation in the ascending order.
2. In 2 adjacent groups, take the min terms if there is only a one-bit difference to make their pair.
3. Place the ‘-’ symbol where there is a bit change accordingly and keep the remaining bits the same. In our project, we use a ‘9’ instead of ‘-’ for convenience.
4. Repeat step 2 and 3 until we can merge anymore. At this time, we will get all prime implicants.
5. Make a prime implicants table.
6. Place ‘1’ in the minterms (cell) which are covered by each prime implicant.
7. Observe the table, if the minterm is covered by only one prime implicants then it is an essential to prime implicants.
8. Add the essential prime implicants to the simplified boolean function.

*\*Now let’s go to an example to see how the Quine-McCluskey work:*

Simplify using tabulation method : F(A,B,C,D) =∑ m(0,1,2,4,6,8,9,11,13,15).

Let’s start by doing this step by step.

***Step 1***: Convert the given minterms into their binary representation and arrange them according to the number of ones present in the binary representation.

| Table 1 | | | | | |
| --- | --- | --- | --- | --- | --- |
| Group | Minterm | A | B | C | D |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 |
| 2 | 0 | 0 | 1 | 0 |
| 4 | 0 | 1 | 0 | 0 |
| 8 | 1 | 0 | 0 | 0 |
| 2 | 6 | 0 | 1 | 1 | 0 |
| 9 | 1 | 0 | 0 | 1 |
| 3 | 11 | 1 | 0 | 1 | 1 |
| 13 | 1 | 1 | 0 | 1 |
| 4 | 15 | 1 | 1 | 1 | 1 |

***Step 2***: Now, for table-2 take minterms from successive groups(simultaneous group only) which have only a 1-bit difference in their representation and form their pair by merging them and making a group of the pairs which are from the same groups that are merged.

| Table 2 | | | | | |
| --- | --- | --- | --- | --- | --- |
| Group | Minterm | A | B | C | D |
| 0 | (0,1) | 0 | 0 | 0 | - |
| (0,2) | 0 | 0 | - | 0 |
| (0,4) | 0 | - | 0 | 0 |
| (0,8) | - | 0 | 0 | 0 |
| 1 | (1,9) | - | 0 | 0 | 1 |
| (2,6) | 0 | - | 1 | 0 |
| (4,6) | 0 | 1 | - | 0 |
| (8,9) | 1 | 0 | 0 | - |
| 2 | (9,11) | 1 | 0 | - | 1 |
| (9,13) | 1 | - | 0 | 1 |
| 3 | (11,15) | 1 | - | 1 | 1 |
| (13,15) | 1 | 1 | - | 1 |

***Step 3***: Repeat the previous step again

| Table 3 | | | | | |
| --- | --- | --- | --- | --- | --- |
| Group | Minterm | A | B | C | D |
| 0 | (0,1,8,9) | - | 0 | 0 | - |
| (0,2,4,6) | 0 | - | - | 0 |
| 1 | (9,11,13,15) | 1 | - | - | 1 |

***Step 4***: As we can see, the table 3 cannot be grouped anymore. Now we will take the Prime Implicants Table:

| PRIME IMPLICANT TABLE | | | | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Min terms | | 0 | 1 | 2 | 4 | 6 | 8 | 9 | 11 | 13 | 15 |
| Prime Implicant | |
| B’C’(0,1,8,9) | | 1 | 1 |  |  |  | 1 | 1 |  |  |  |
| A’D’(0,2,4,6) | | 1 |  | 1 | 1 | 1 |  |  |  |  |  |
| AD(9,11,13,15) | |  |  |  |  |  |  | 1 | 1 | 1 | 1 |

B’C’ is simplified as minterm 1 is only covered by B’C’. Similarly, minterms 2,4,6 are only covered by A’D’ and minterms 11,13,15 are only covered by AD.

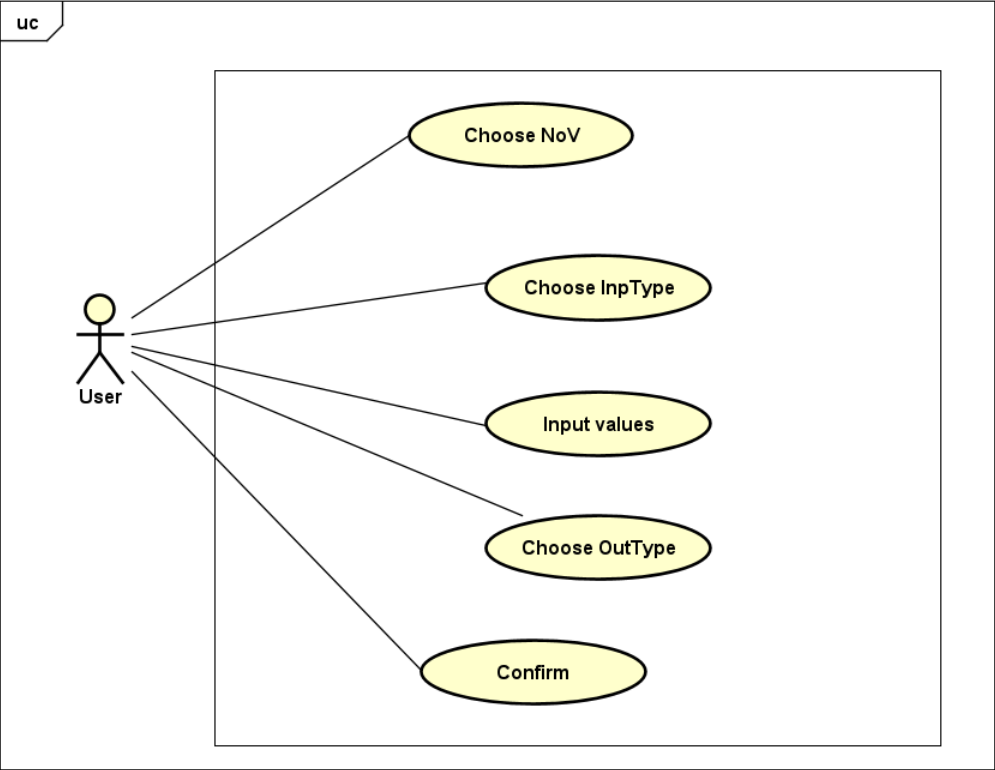
>>Simplified boolean function: B’C’+A’D’+AD.

***Approaching to Quine-McCluskey method***

In our project, we want to do a GUI that allows people to choose the number of variables, type of input( Truth table or Kmap), choose which values are contained and type of output(SOP or POS). By clicking confirm, user cannot only get the logical expression but also the visualization of the map of getting the expression.

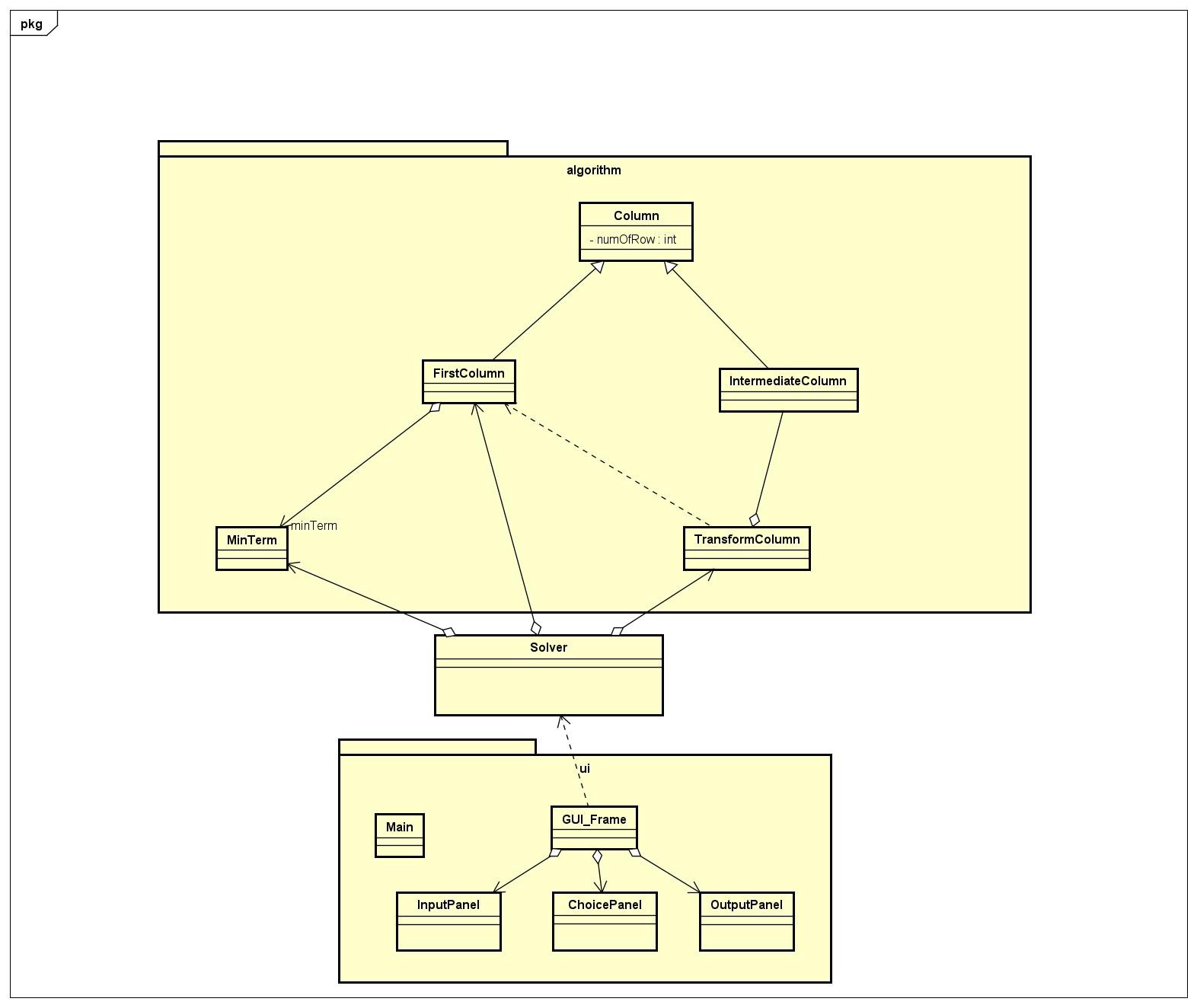
1. Use case Diagram

In our project, we want to do a GUI that allows people to choose the number of variables, type of input( Truth table or Kmap), choose which values are contained and type of output(SOP or POS). By clicking confirm, user can get the logical expression, prime implicants table and circuit drawing.



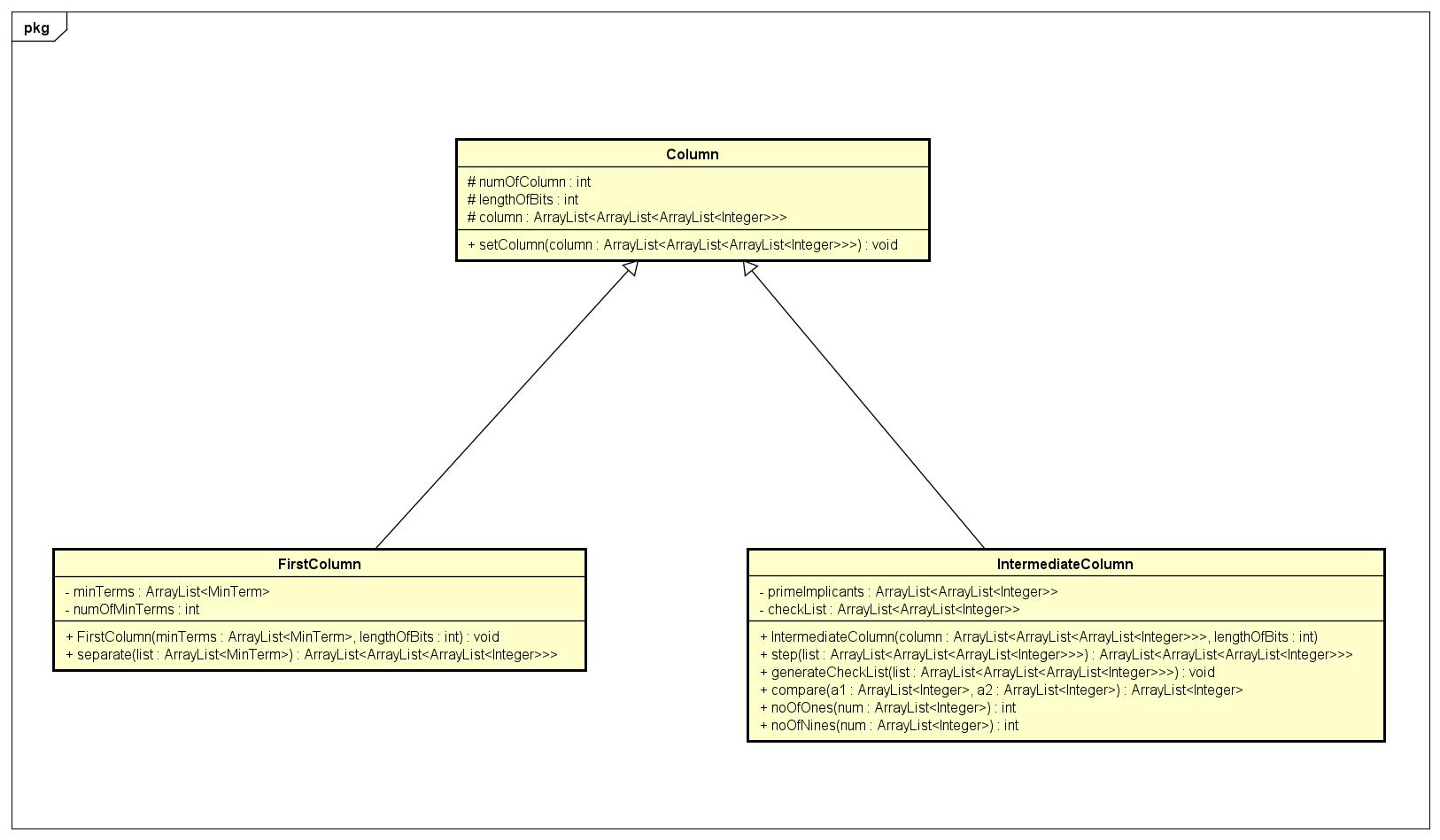
1. *Design*
2. General Class Diagram

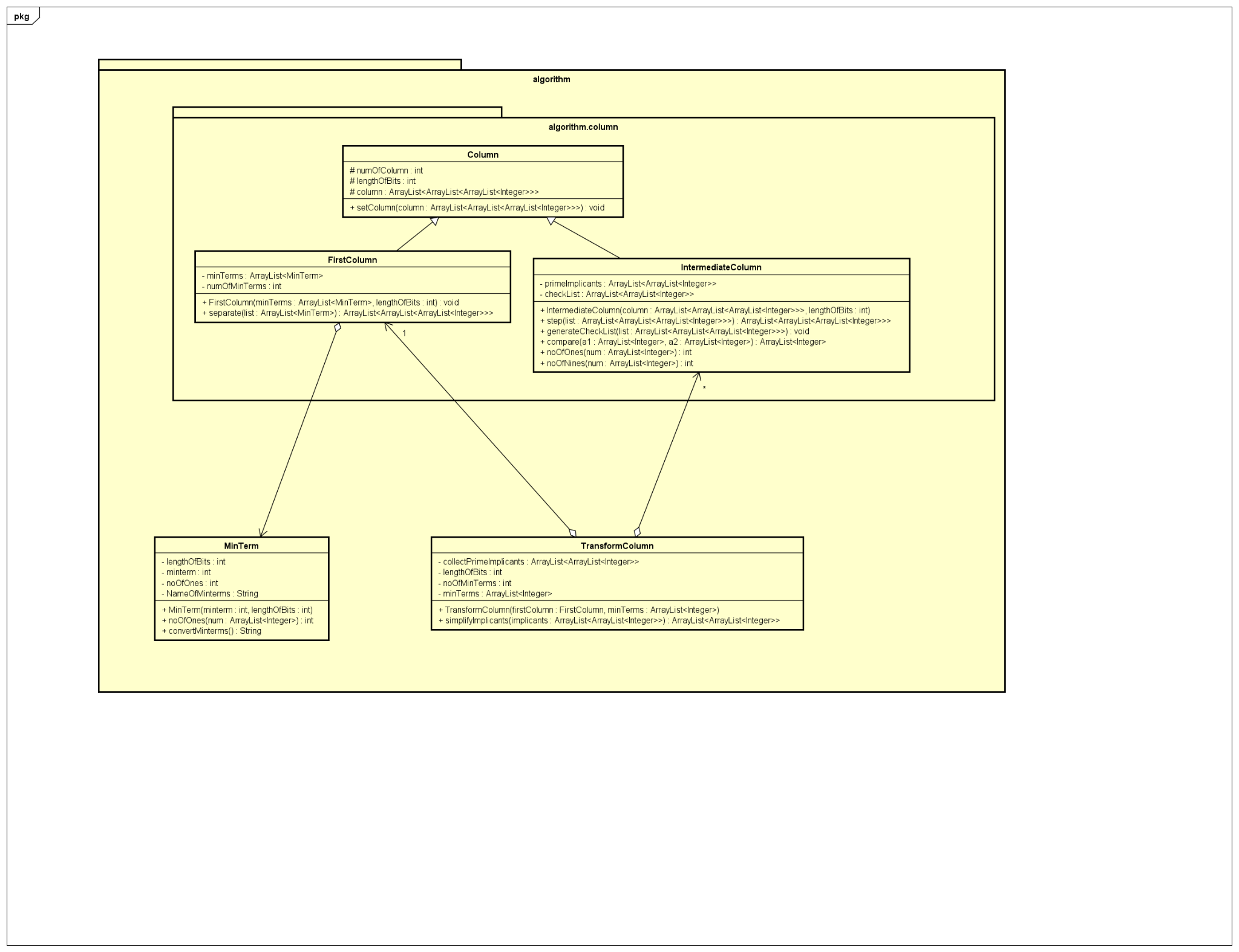
Below is the general class diagram we create for the method:



This is the general idea for our project. From the Input Panel, users can choose input type and input value and the values will be recognized as minterm. In the MinTerm class, it will be processed and represented in binary string before grouped and shown in FirstColumn. The TransformColumn will take FirstColumn as an initial value, create many IntermediateColumn, collecting prime implicants by making group of IntermediateColumn. Then from the TransformColumn we will take the prime implicants to show to OutputPanel.

1. Algorithm Class Diagram





For Algorithm part, first, we will create MinTerms by using the minterm that we have from the InputPanel.

The MinTerms then will be collected to an ArrayList<MinTerm> to construct a FirstColumn class.

The TransformColumn class will be created with the initial FirstColumn, then create the IntermediateColumn by using method step. After each step, TransformColumn will collect prime implicants and add to attribute ArrayList<ArrayList<Integer>> collectPrimeImplicants.

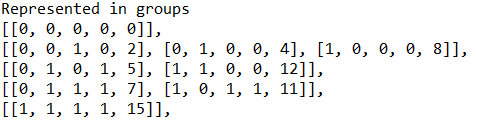
***Attributes and methods of each class***

**1. MinTerm:**

* *Attributes*
* lengthOfBits: length of binary representation, equals to number of variables
* minterm: the minterm as an integer
* noOfOnes: number of ones appear in the binary representation
* NameOfMinterms: the name of minterm as a string
* minTerms: min term as arraylist of binary representation
* *Methods*
* Minterm(String nameOfMinterms,int lengthOfBits): constructor with the input argument as string nameOfMinterms and int lenghtOfBits
* Minterm(int minterm, int lengthOfBits): overloading constructor that input is minterm as an integer and lengthOfBits.
* convertMinterms(): a method to change from NameOfMinterms into binary representation and the return type is a String.

**2. Column:**

* Attributes:
* numOfColumn: number of groups in column
* lengthOfBits: number of variables, taken from the InputPanel
* column: ArrayList<ArrayList<ArrayList<Integer>>> to store group of terms as groups ,each member of group has the same number of 1, or be grouped by method in TransformColumn



ArrayList<Integer> stored will be presented as [0, 1, 9, 1, 5, 7], where n first integers will be the binary representation (n = lengthOfBits), and the latter integers will be the terms that can be grouped to have the binary representation. The binary representation here will have 1 is true, denoted as A for example, 0 is false, denoted as Ā for example, and 9 is don’t care value, which will not be represented.

1. **FirstColumn:**

* **Attributes:**
* minTerms: ArrayList<MinTerm> to store object MinTerm
* numOfMinTerms: number of minterms that the arraylist minTerms store
* **Methods:**
* FirstColumn(ArrayList<MinTerm> minTerms, int lengthOfBits): constructor that arguments contain minTerms - an arraylist have all the object type MinTerm and lengthOfBits as an integer.
* separate(ArrayList<MinTerm> list): divide the list of Min Term into group which are sorted base on the number of 1 in binary representation. The method will return a ArrayList<ArrayList<ArrayList<Integer>>> with each index of the outside most arraylist is each group corresponding to ‘index’ number of 1.

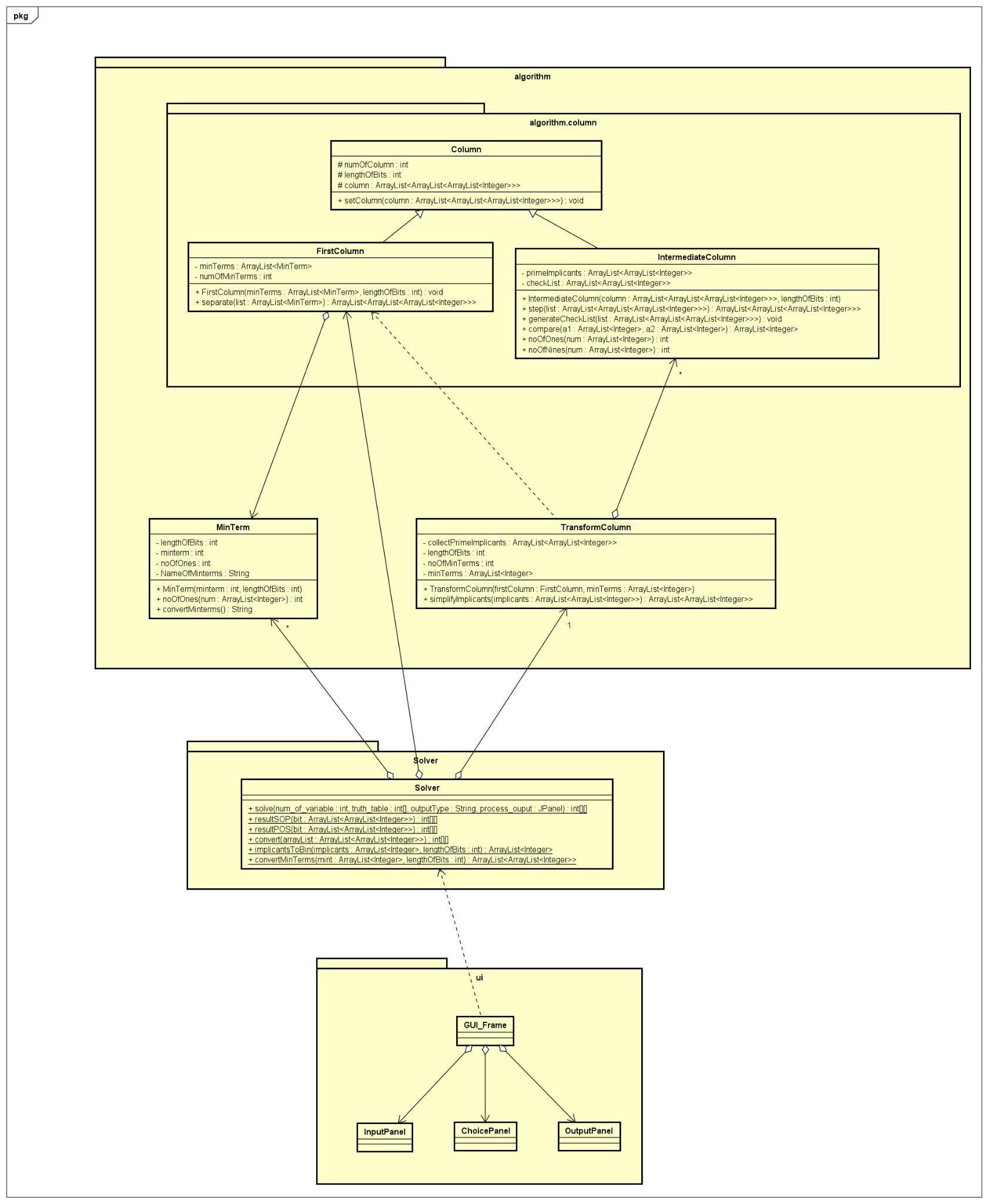
1. **IntermediateColumn:**

* **Attributes:**
* FirstColumn(ArrayList<MinTerm> minTerms, int lengthOfBits): constructor that arguments contain minTerms - an arraylist have all the object type MinTerm and lengthOfBits as an integer.
* separate(ArrayList<MinTerm> list): divide the list of Min Term into group which are sorted base on the number of 1 in binary representation. The method will return a ArrayList<ArrayList<ArrayList<Integer>>> with each index of the outside most arraylist is each group corresponding to ‘index’ number of 1.
* **Methods:**
* generateCheckList: void method to create the checkList, with all values initialized 0.
* compare(ArrayList<Integer> a1, ArrayList<Integer> a2): check 2 binary representations a1 and a2, if only one position is different (0 != 1), then compare will return a new binary representation of the group of minterms, by adding previous minterms.
* step(ArrayList<ArrayList<ArrayList<Integer>>>): take the previous column, return a ArrayList<ArrayList<ArrayList<Integer>>> as the new column, storing intermediate group of terms, then compare each 2 adjacent groups, by using method compare(ArrayList<Integer> a1, ArrayList<Integer> a2) for all the pairs of 2 adjacent groups. If can use compare, both the positions of 2 groups of terms will be noted by changing the checkList as that position to 1.
* IntermediateColumn(ArrayList<ArrayList<Integer>> Column, int lengthOfBits): constructor to take the previous column, then processing the step method. After all, we will use the checkList to find the group of terms that haven’t been used with compare(), it will be added to the primeImplicants.
* noOfOnes(ArrayList<Integer>): check a binary representation to know how many ones appear in that
* noOfNines(ArrayList<Integer>): a number 9 in the representation will represent the don't care expression, which means it is a combination of 1 and 0, whatever it is, we will not need to know. For example, a binary representation [1,9,1,1] will denote ACD, the B position represented by 9 will not be listed in.

1. **TransformColumn:**

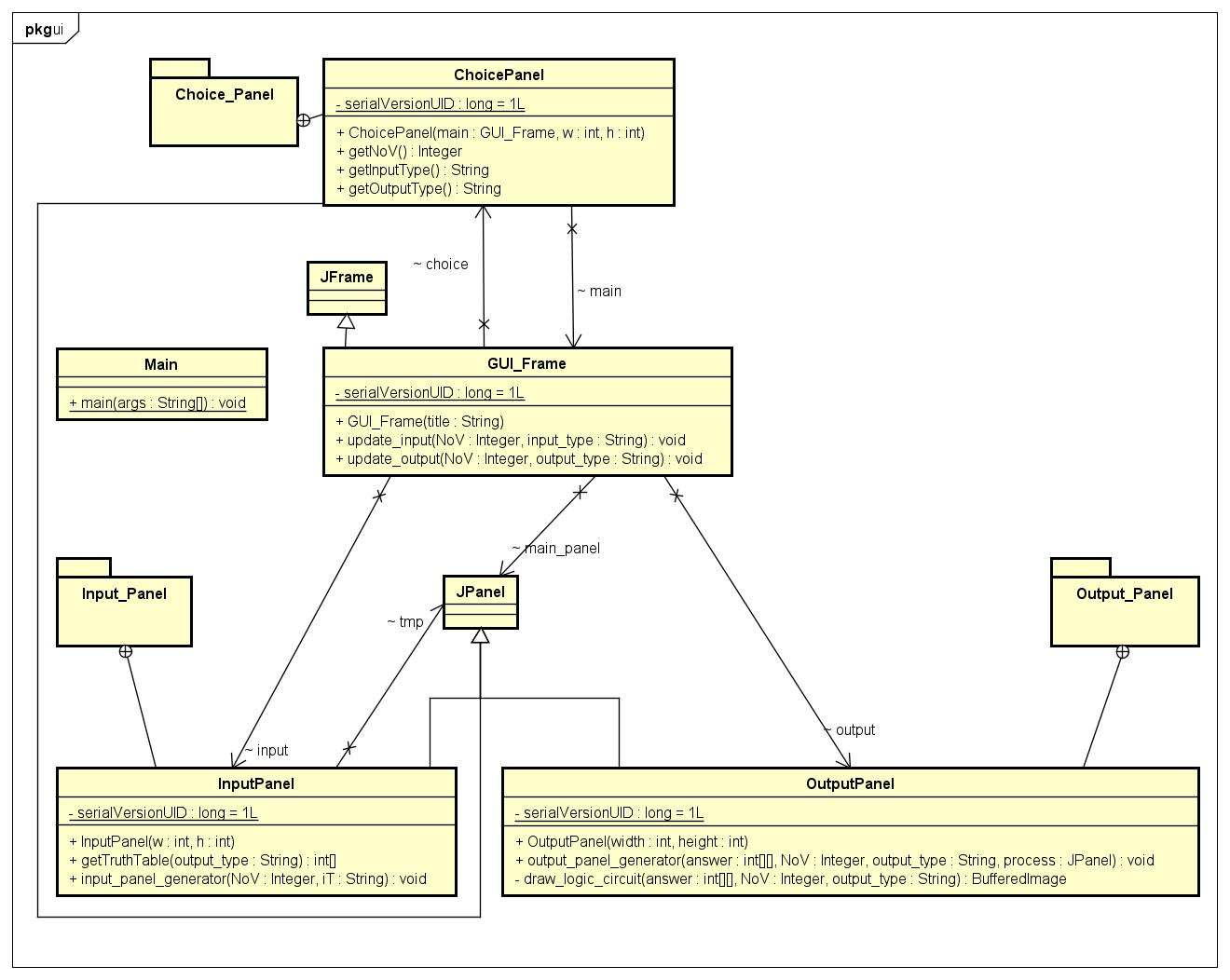
* **Attributes:**
* collectPrimeImplicants: an ArrayList<ArrayList<Integer>> to store all the group of terms as prime implicants to show.
* lengthOfBits: length of binary representation, equals to number of variables.
* minTerms: an ArrayList<Integer> to store all the minterm collected at the InputPanel, as integer.
* numOfMinTerms: number of minterms that the arraylist minTerms store.
* **Methods**
* TransformColumn(FirstColumn firstColumn, ArrayList<Integer> minTerms): constructor with initial value firstColumn and minTerms. IntermediateColumn will be created based on FirstColumn, and the latter IntermediateColumn will be created based on the former IntermediateColumn. When an IntermediateColumn is created, we will collect its attribute primeImplicants to add to the attributes collectPrimeImplicants. We will created IntermediateColumn sequentially until all prime implicants we need is collected. The attributes collectPrimeImplicants then will be optimized by the method simplifyImplicants to remove prime implicants that are unnecessary.
* simplifyImplicants(ArrayList<ArrayList<Integer>> implicants): take the group of prime implicants, we will optimize by checking each prime implicant, if it can be represented by other prime implicants, we will remove it from the collectedPrimeImplicants. Also, we sort each group of terms as prime implicant with ascending order, check again to avoid duplication.
* getCollectPrimeImplicants(): return CollectPrimeImplicants

1. Solver Class Diagram



**Solver:**

* solve(Integer num\_of\_variable, int[] truth\_table, String ouput\_type, JPanel process\_ouput): take num\_of\_variable, truth\_table, output\_type and process output. First we will create ArrayList<MinTerms> minTerms with minterms taken from truth\_table, create FirstColumn(minTerms, num\_of\_variable). Then we will create the TransformColumn(firstColumn, num\_of\_variable), we will have the list of prime implicants in transformColumn.getCollectPrimeImplicants(). Then for return\_type as “SOP” or “POS”, we will have the equivalent return with returnSOP() and returnPOS().
* returnSOP(ArrayList<ArrayList<Integer): return int[][] the table of prime implicants in SOP type.
* returnPOS(ArrayList<ArrayList<Integer): return int[][] the table of prime implicants in POS type. In POS return type, we will take as opponent to the SOP return type. So later in the OutputPanel with return type is POS, we only need to change multiply to add and change add to multiply.
* convert(ArrayList<ArrayList<Integer>> arrayList): return an int[][] which represents the same variables and positions as the ArrayList<ArrayList<Integer>>. This is to fit with the GUI\_Frame.
* implicantsToBin(ArrayList<Integer> implicants, int lengthOfBits): return a group of terms, for example [0,2] to binary representation, [0,0,9,0].
* convertMinTerms(ArrayList<Integer> mint, int lengthOfBits): transform minterm, for example 1 to binary represenation, [0,0,0,1].

1. User Interface Class Diagram

The user interface is placed inside a GUI\_Frame which is the main JFrame. It includes 3 JPanel: a ChoicePanel that takes user input, output setting; an InputPanel that takes user input; an OutputPanel that shows group implicants, normalized expression, and output diagram.

**GUI\_Frame:**

* main\_panel: the content pane.
* choice, input, output: 3 parts of the design.
* update\_input(NoV, input\_type): function when input setting is changed, prompt ‘input’ to update itself.
* update\_output(NoV, input\_type): function when user confirm input, call solver and provide result for ‘output’ to print.

**ChoicePanel:**

* num\_variable, input\_type, output\_type: combo box for respective setting.
* main: the GUIFrame that include this panel, used to handle events.
* OptionChanged(itemEvent), ConfirmClicked(actionEvent): event handler that take needed information to pass to ‘main’.
* getNoV(), getInputType(), getOutputType(): respectively getter function.

**InputPanel:**

* tmp: panel to include user input.
* input\_panel\_generator(NoV, input\_type): function to draw input panel.
* getTruthTable(output\_type): getter function for user input.

**OutputPanel:**

* output\_panel\_generator(answer, NoV, output\_type, process\_pane): function to draw output panel.
* draw\_logic\_circuit(answer, NoV, output\_type): function to draw logic circuit, AND\_Shape and OR\_Shape is custom Shape used in this function.

1. Explanation of Design

***Inheritance***

Since FirstColumn, IntermediateColumn have the same attribute *numOfColumn*, *lengthOfBits* and *column* as well as the methods like *getNumOfColumn, getLengthOfBits, getColumn,....* So that we create the column class for *FirstColumn* and *IntermediateColumn* to inherit*.*

***Aggregation***

As we can see in the Solver Class Diagram, the Solver contains many MinTerm, added to an ArrayList<MinTerm>, a FirstColumn and a TransfromColumn. The FirstColumn also contains an ArrayList<MinTerm>. The TransformColumn contains many IntermediateColumn.

In GUI\_Frame, we contain 3 panels: ChoicePanel, InputPanel, OutputPanel. Also each part can process its own property, while communicating with each other using only necessary information.

***Encapsulation***

Inside InputPanel, the data is hidden from outer interference, the JPanel ‘tmp’ that contains user input can only be accessed or modified by method: input\_panel\_generator and getTruthTable.

***Dependency***

As we can see in Solver Class Diagram, the GUI\_Frame will create a variable which takes value as the return of a method from Solver.Also, the TransformColumn takes FirstColumn to construct.

1. *Result*
2. Result

From the screen, user can choose a number of variables. In the following image, we choose the number of variables 3 and it displays only (A,B,C). Changing input type, values and types of output and we will get the following result.

