Hanoi University
of Science and Technology

Object Oriented Programming - IT3100E

Logic Expression Normalizer

Group 23

Instructor: Dr. Hung Tran The

Members:

Bui Dang Quy - 20225995

Lo Duc Tai - 20225999

Pham Quang Anh - 20220071

Contents

1 Assignment of members

The following table show the tasks and the involvement of each members in each task:

Name	Diagrams	Source Code	Demo Video	Slide	Report
Lo Duc Tai	10%	50%	100%	0%	30%
Pham Quang Anh	0%	50%	0%	10%	35%
Bui Dang Quy	90%	0%	0%	90%	35%

Table 1: Involvement of members in each task

2 Description

2.1 Project Requirements

The *Logic Expression Normalizer* is designed to simplify and standardize logical expressions, primarily those expressed in Boolean algebra. The core functionality involves transforming various forms of logic expressions into a standard normal form, such as Disjunctive Normal Form - Sum of Product (SoP) or Conjunctive Normal Form - Product of Sum (PoS). To achieve this, the software must handle complex expressions comprising of multiple variables. A critical requirement is the development of a user-friendly interface that allows users to input their logical expressions effortlessly. The software should then process these inputs, validate their syntax, and convert them into the desired normal form, ensuring that the output is both accurate and simplified. Additionally, robust error-checking mechanisms are necessary to manage invalid inputs and syntax errors, providing feedback to users to correct any mistakes. The overall goal is to create an intuitive tool that assists users in understanding and manipulating logical expressions efficiently.

2.2 Use-case Diagram

The use case diagram provided outlines the interaction between the user and the *Logic Expression Normalizer* software.

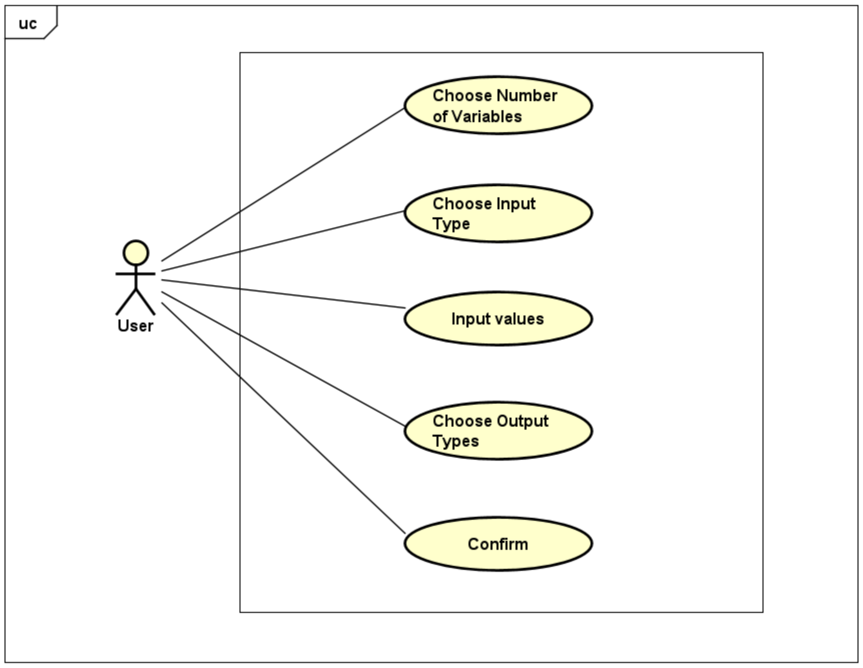


Figure 1: UseCase Diagram demonstrates how the application works

- **Choose Number of Variables:**
  - **Objective:** Determine the scope of the logical expression by specifying the number of variables involved.
  - **Details:**
    * **User Interaction:** The user is presented with a menu to select the number of variables (3 or 4 in this case).
    * **System Response:** Once a selection is made, the system prepares the interface for the next steps, ensuring that subsequent input fields can accommodate the chosen number of variables.
- **Choose Input Type:**
  - **Objective:** Specify the format of the logical expression input.
  - **Details:**
    * **User Interaction:** The user is given several options for the input type, such as:
      · **Truth Table:** A table where users can fill in true/false values for all combinations of variables.
      · **Logical Formula:** An input box where users can type a logical expression using variables and operators.
      · **Others:** Possibly other forms like Karnaugh maps or standard forms.

        - **System Response:** Based on the chosen input type, the system dynamically adjusts the input interface:
            - For a truth table, a matrix of input fields is generated corresponding to the number of variables.
            - For a logical formula, a single text input box is provided.

- **Input Values:**
    - **Objective:** Receive the specific values or logical expressions.
    - **Details:**
        - **User Interaction:**
            - **Truth Table:** The user fills in the table with true (1) or false (0) values for each combination of variable states. Since the user already input number of variables, the users selects
        - **System Response:** The system provides real-time feedback:
            - For truth tables, it checks that all required fields are filled.
            - For logical formulas, it checks for proper syntax and highlights any errors.

- **Choose Output Types:**
    - **Objective:** Select the desired format for the normalized output.
    - **Details:**
        - **User Interaction:** The user chooses from available output formats, such as:
            - **Sum of Products (SoP):** A standard form where the expression is a disjunction (Sum) of conjunctive clauses (Product).
            - **Product of Sum (PoS):** A standard form where the expression is a conjunction of disjunctive clauses.
        - **System Response:** The system notes the selected output type and prepares to process the input accordingly.

- **Present the result:**
    - **Objective:** Review and finalize the input and choices, then proceed with normalization.
    - **Details:**
        - **User Interaction:**
            - Users can scroll through the panel if the result display is larger than the given display area.
            - Users can show the circuit diagram for logic expression and save the diagram image.
        - **System Response:** Upon confirmation, the system processes the input to generate the normalized output:
            - If user wants to save the diagram image, the system pops up a window of File Explorer to designate the image location
        - **Validation:**

3 Design

3.1 General class diagram

Our project is divided into 2 main sections: the algorithm part handles all the logical expression processing, the UI part displays the required UI. All logical values will be stored as instances of class Minterm. For each time the user uses the simplication funcionality, a new ColumnTable is created to find the prime implicants, then it will be sent to a PITable to form the final expression.

All of the necessary details will be displayed in the OutputScreen.

3.2 Algorithm class diagram

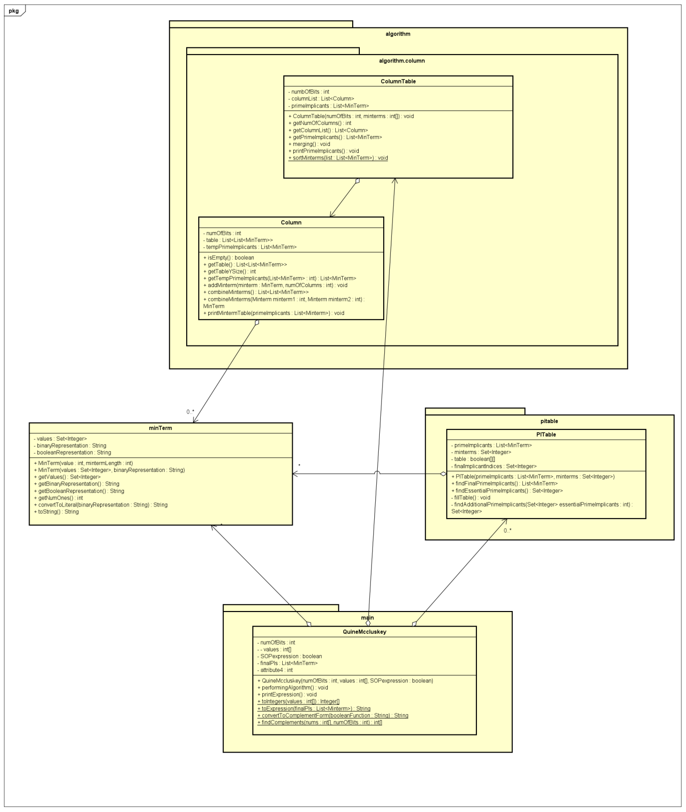

pkg
algorithm
algorithm.column
ColumnTable
- numbOfBits : int
- columnList : List<Column>
- primeImplicants : List<MinTerm>
+ ColumnTable(numOfBits : int, minterms : int[]) : void
+ getNumOfColumns() : int
+ getColumnList() : List<Column>
+ getPrimeImplicants() : List<MinTerm>
+ merging() : void
+ printPrimeImplicants() : void
+ sortMinterms(list : List<MinTerm>) : void
Column
- numOfBits : int
- table : List<List<MinTerm>>
- tempPrimeImplicants : List<MinTerm>
+ isEmpty() : boolean
+ getTable() : List<List<MinTerm>>
+ getTableYSize() : int
+ getTempPrimeImplicants(List<MinTerm> : int) : List<MinTerm>
+ addMinterm(minterm : MinTerm, numOfColumns : int) : void
+ combineMinterms() : List<List<MinTerm>>
+ combineMinterms(Minterm minterm1 : int, Minterm minterm2 : int) : MinTerm
+ printMintermTable(primeImplicants : List<Minterm>) : void
0..*
minTerm
- values : Set<Integer>
- binaryRepresentation : String
- booleanRepresentation : String
+ MinTerm(value : int, mintermLength : int)
+ MinTerm(values : Set<Integer>, binaryRepresentation : String)
+ getValues() : Set<Integer>
+ getBinaryRepresentation() : String
+ getBooleanRepresentation() : String
+ getNumOnes() : int
+ convertToLiteral(binaryRepresentation : String) : String
+ toString() : String
*
pitable
PITable
- primeImplicants : List<MinTerm>
- minterms : Set<Integer>
- table : boolean[][]
- finalImplicantIndices : Set<Integer>
+ PITable(primeImplicants : List<MinTerm>, minterms : Set<Integer>)
+ findFinalPrimeImplicants() : List<MinTerm>
+ findEssentialPrimeImplicants() : Set<Integer>
- fillTable() : void
- findAdditionalPrimeImplicants(Set<Integer> essentialPrimeImplicants : int) : Set<Integer>
0..*
main
QuineMccluskey
- numOfBits : int
- - values : int[]
- SOPexpression : boolean
- finalPIs : List<MinTerm>
- attribute4 : int
+ QuineMccluskey(numOfBits : int, values : int[], SOPexpression : boolean)
+ performingAlgorithm() : void
+ printExpression() : void
+ toIntegers(values : int[]) : Integer[]
+ toExpression(finalPIs : List<Minterm>) : String
+ convertToComplementForm(booleanFunction : String) : String
+ findComplements(nums : int[], numOfBits : int) : int[]

Figure 2: Overview of Algorithm Diagram

After the user have selected the minterms from the InputScreen and the output format (SoP or PoS), a new QuineMccluskey instance is created to simplify the expression.

First, a ColumnTable is created. It tries to merge the implicants until no more new ones can be created. Each iteration creates a new instance of Column to store the data for displaying purpose.

When the merging stops, it stores a list of Prime Implicants. This list will be put in the PITable, to select the necessary implicants for the final simplified expression.

Attributes and methods of each class

1. **Minterm**

   (For better explanation, we'll give an example of a new minterm composes of integer values 6 and 7)

   Attributes:

   - values: a set of integers corresponding to that implicants ((6,7) with binary values 0110 and 0111)
   - binaryRepresentation: the implicant's binary represtation (011-)
   - booleanRepresentation: the implicants's Boolean representation. (This representation uses the capitalized English alphabet letters, starting from the left, to store the literals of expression. In this case, it is stored as 'A'BC '- the apostrophe represents NOT operation)

   Methods:

   - Minterm(int value, int mintermLength): constructor with the input of a single value, used by the first column when initiate the data.
   - Minterm(Set ⟨Integer⟩ values, String binaryRepresentation: overloading constructor for multiple-value implicants. The binaryRepresentation needs to be calculated beforehand.
   - getValues(), getBinaryRepresentation(), getBooleanRepresentation(): getters of corresponding attributes.
   - getNumOnes(): returns number of 1s of implicants. This method is important for the process of merging.
   - convertToLiteralString(String binaryRepresentation): returns the BooleanRepresentation for the constructors.
   - toString(): (for testing purposes only) override the string representation to print the values.
   - getPositionOfBinaries(): returns the positions of don't care values, use for generating circuit diagram.

2. **Column**

   Attributes:

   - numOfBits: number of bits
   - table: a 2D dynamic array that contains all the implicants for that column. Rows organized by number of 1s in the implicants.
   - tempPrimeImplicants: store the prime implicants of that column. This value will be collected by the ColumnTable later.

   Methods:

- Column(int numOfBits) and Column(int numOfBits, List⟨List⟨Minterm⟩⟩ table): constructor of class Column. The first one is used for the first column, the later one overrides to construct Column for next columns.
- isEmpty(): check if the table is empty, for display purposes
- getTable(), getTableYSize(), getTempPrimeImplicants: getters of attributes. getTableYSize() is used for displaying purposes
- addMinterm(): add an implicant to the corresponding row
- combineMinterms(. . .): there are 2 versions of this method, one to return the merged value, and one to create an entire new table contain the merged values
- printMintermTable(): (for testing purposes only) print the table to the Terminal

3. **ColumnTable**

   Attributes:

   - numOfBits: number of bits
   - columnList: the List of Column(s) for the merging process
   - primeImplicants: the List of prime implicants.

   Methods:

   - ColumnTable(int: numOfBits, int[] minterms): constructor for ColumnTable, initate by an array of minterms. The merging process is also done during the construction.
   - getNumOfColumns(), getColumnList(), getPrimeImplicants(): getters of attributes
   - merging(): loops the merging process until the new possible table is empty (no more merging can be done)
   - sortMinterms(): sorts the primeImplicants in order. This serves as preparation for the next step, implicants that have fewer values are prioritized.

     This is based on the real world application of logical expression simplifier, since fewer values means less logic gates are used. A 4-input AND gate uses 3 2-input AND gates

4. **PITable**

   Attributes:

   - primeImplicants: list of prime implicants
   - minterm: set of all minterms used by the primeImplicants
   - table: table of prime implicants selection
   - finalImplicantsIndices: to get the final implicant positions

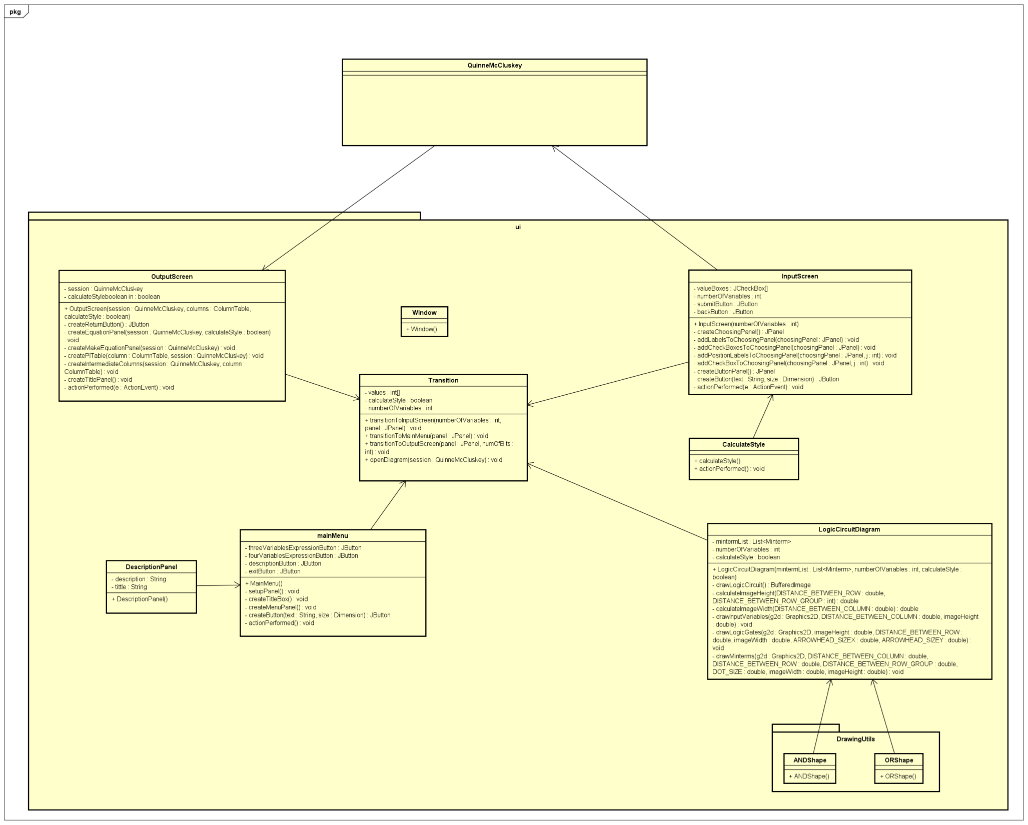
   Methods:

   - PITable(List ⟨Minterm⟩ primeImplicants, Set ⟨Integer⟩ primeImplicants): constructor for class PITable, it receives a list of Prime Implicants (PI), as well as a list of minterms covered by all the PIs

- findFinalPrimeImplicants(): returns the PI used for the final expression, takes input from the indices attribute
- findFinalPrimeImplicantsIndex(): this method wraps the following 2 methods/steps to find the required PI indices
- Set⟨Integer⟩ findEssentialPrimeImplicants(): finds the essential PIs. It searches for minterms that can only be covered by 1 implicant. That implicant is chosen as 'essential'
- Set⟨Integer⟩ findAdditionalPrimeImplicants(): finds the other PIs to cover the rest of the minterms. This is done by brute-force search.
- fillTable(): fills the PI chart for processing

5. **QuinneMccluskey**: this class wraps all the functionality of the algorithm package for

3.3 UI class diagram


pkg
QuinneMcCluskey
ui
OutputScreen
- session : QuinneMcCluskey
- calculateStyleboolean in : boolean
+ OutputScreen(session : QuinneMcCluskey, columns : ColumnTable, calculateStyle : boolean)
- createReturnButton() : JButton
- createEquationPanel(session : QuinneMcCluskey, calculateStyle : boolean) : void
- createMakeEquationPanel(session : QuinneMcCluskey) : void
- createPITable(column : ColumnTable, session : QuinneMcCluskey) : void
- createIntermediateColumns(session : QuinneMcCluskey, column : ColumnTable) : void
- createTitlePanel() : void
- actionPerformed(e : ActionEvent) : void
Window
+ Window()
InputScreen
- valueBoxes : JCheckBox[]
- numberOfVariables : int
- submitButton : JButton
- backButton : JButton
+ InputScreen(numberOfVariables : int)
- createChoosingPanel() : JPanel
- addLabelsToChoosingPanel(choosingPanel : JPanel) : void
- addCheckBoxesToChoosingPanel(choosingPanel : JPanel) : void
- addPositionLabelsToChoosingPanel(choosingPanel : JPanel, j : int) : void
- addCheckBoxToChoosingPanel(choosingPanel : JPanel, j : int) : void
- createButtonPanel() : JPanel
- createButton(text : String, size : Dimension) : JButton
- actionPerformed(e : ActionEvent) : void
Transition
- values : int[]
- calculateStyle : boolean
- numberOfVariables : int
+ transitionToInputScreen(numberOfVariables : int, panel : JPanel) : void
+ transitionToMainMenu(panel : JPanel) : void
+ transitionToOutputScreen(panel : JPanel, numOfBits : int) : void
+ openDiagram(session : QuinneMcCluskey) : void
CalculateStyle
+ calculateStyle()
+ actionPerformed() : void
DescriptionPanel
- description : String
- tittle : String
+ DescriptionPanel()
mainMenu
- threeVariablesExpressionButton : JButton
- fourVariablesExpressionButton : JButton
- descriptionButton : JButton
- exitButton : JButton
+ MainMenu()
- setupPanel() : void
- createTitleBox() : void
- createMenuPanel() : void
- createButton(text : String, size : Dimension) : JButton
- actionPerformed() : void
LogicCircuitDiagram
- mintermList : List<Minterm>
- numberOfVariables : int
- calculateStyle : boolean
+ LogicCircuitDiagram(mintermList : List<Minterm>, numberOfVariables : int, calculateStyle : boolean)
- drawLogicCircuit() : BufferedImage
- calculateImageHeight(DISTANCE_BETWEEN_ROW : double, DISTANCE_BETWEEN_ROW_GROUP : int) : double
- calculateImageWidth(DISTANCE_BETWEEN_COLUMN : double) : double
- drawInputVariables(g2d : Graphics2D, DISTANCE_BETWEEN_COLUMN : double, imageHeight : double) : void
- drawLogicGates(g2d : Graphics2D, imageHeight : double, DISTANCE_BETWEEN_ROW : double, imageWidth : double, ARROWHEAD_SIZEX : double, ARROWHEAD_SIZEY : double) : void
- drawMinterms(g2d : Graphics2D, DISTANCE_BETWEEN_COLUMN : double, DISTANCE_BETWEEN_ROW : double, DISTANCE_BETWEEN_ROW_GROUP : double, DOT_SIZE : double, imageWidth : double, imageHeight : double) : void
DrawingUtils
ANDShape
+ ANDShape()
ORShape
+ ORShape()
Figure 3: Overview of the UI package

For the UI of our project, we used the Swing package to develop it. Our UI consists of 3 main part:

- The Window class, which inherits the JFrame class, acts as a container that contains our other components
- The Screen classes, which inherit the JPanel class and will be added into our main Frame during the runtime of our program
- The Utility classes, including the transition logic class, the drawing utilities and the calculate style choice.

1. **Window**
   The Window class, which inherits the JFrame class, acts as a container to display the screens of our program.

   Methods:

   - Window(JPanel panel): The constructor of Window class. It takes a JPanel as parameter and create a JFrame to contain this panel. The Constructor also set the attribute of the Frame (dimension, visibility,...).

2. **Transition**

   The Transition class is an utility class that contains the methods required to switch between screens when invoked. This class also keep track of the numberOfVariables and calculateStyle variables, which are used by many different classes.

   Attributes:

   - values: List of the values taken from the input, this is necessary since we need this to calculate the output
   - calculateStyle: A boolean variable that instruct our program which type of calculation to do. False is Sum Of Products and True is Product Of Sums
   - numberOfVariables: the number of variables that our program can process, it varies between 3 and 4 variables

   Methods: We swap between screens by using these methods. They all operate very similarly by remove the current panel and add new panel afterward.

   - transitionToInputScreen: Transition to input screen. Accessible from the main menu
   - transitionToMainMenu: Transition to the main menu. Accessible from the input and output screens
   - transitionToOutputScreen: Transition to Output screen. Accessible from the input screen. Aside from the transition, this method also invoke an instance of QuinneMcCluskey object, which enable us to get the output.
   - openDiagram: Used to open the circuit diagram. Accessible from the output screen.

3. **MainMenu**

   The MainMenu class acts as a hub for user to access to the functions of our program. It allow user to choose between 3 and 4 variables calculation, it also has a description button that show a brief description of the program. The exit button closes the program.

   Methods:

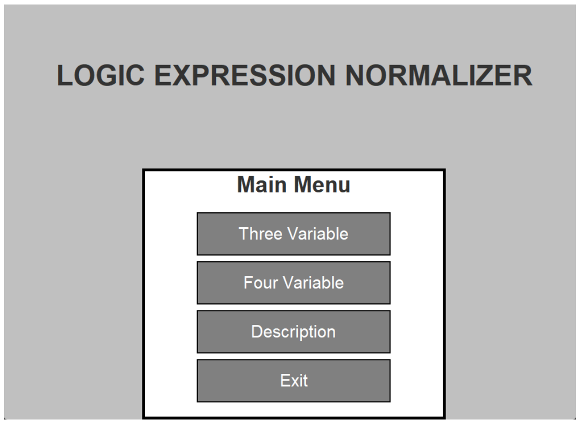
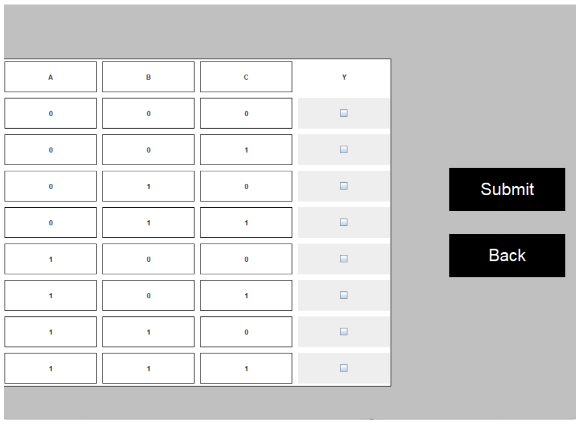


Figure 4: Illustration of the Main Menu

- MainMenu: Constructor of the class.
- setupPanel: Assign the attributes (color, size,...) of the panel
- createTitleBox: Create the title.
- createMenuPanel: Create the subpanel that contains the buttons.
- createButton: Create buttons

4. **InputScreen**



A	B	C	Y
0	0	0	☐
0	0	1	☐
0	1	0	☐
0	1	1	☐
1	0	0	☐
1	0	1	☐
1	1	0	☐
1	1	1	☐

Figure 5: Three Variables input screen

The InputScreen provides an interface allows user to choose the values in the form of a truth table. Its appearance changes based on the number of variables. Consists of a subpanel that allow users to choose the input, a Submit button and a Back button.

Methods:

- InputScreen: Constructor
- createChoosingPanel: Create the choosing panel.
- createButtonPanel: Create the panel which hold the buttons
- createButton: Create buttons

5. **CalculateStyle**

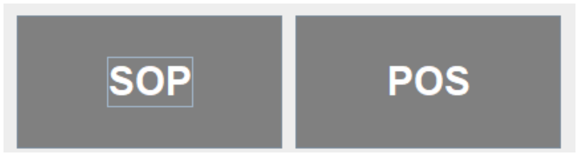
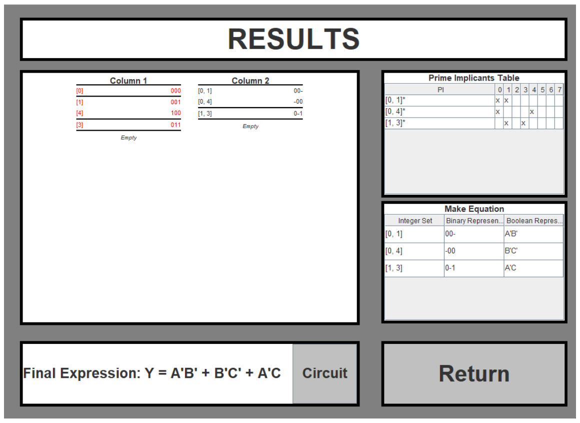


Figure 6: Selection of Output format

This class inherits JDialog class. Contains two button SOP (for Sum of Products) and POS (for Product Of Sums). User can choose the calculate style they desire here.

6. **OutputScreen**



Column 1

[0]	000
[1]	001
[4]	100
[3]	011

Empty

Column 2

[0, 1]	00-
[0, 4]	-00
[1, 3]	0-1

Empty

Prime Implicants Table

PI	0	1	2	3	4	5	6	7
[0, 1]*	x	x						
[0, 4]*	x				x			
[1, 3]*		x		x				

Make Equation

Integer Set	Binary Represen...	Boolean Repres...
[0, 1]	00-	A'B'
[0, 4]	-00	B'C'
[1, 3]	0-1	A'C

Final Expression: Y = A'B' + B'C' + A'C

Figure 7: Output screen corresponding to the input 0,1,3,4

Our output screen consists of the Intermediate Columns (The upper left panel), the prime implicants table and the make equation table. The lower left panel has the final, normalized expression as well as a button that allows user to see the circuit blog diagram. The Return button return to menu.

7. **LogicCircuitDiagram**

The logic circuit screen. It shows the circuit implemented using AND and OR gates. It also has a Save button , which allows user to save the circuit as a png image file. The Return button close this window.

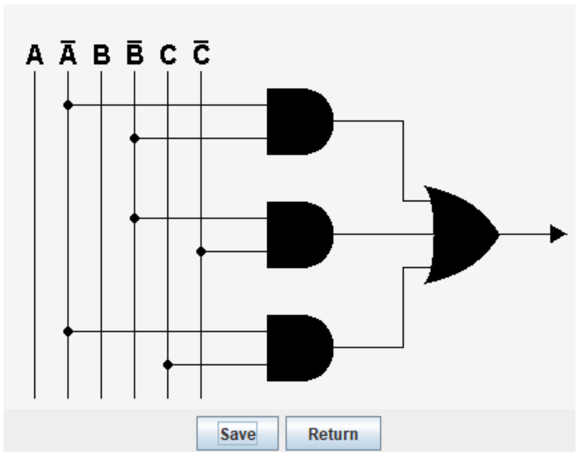


Figure 8: Circuit corresponding to the input 0,1,3,4

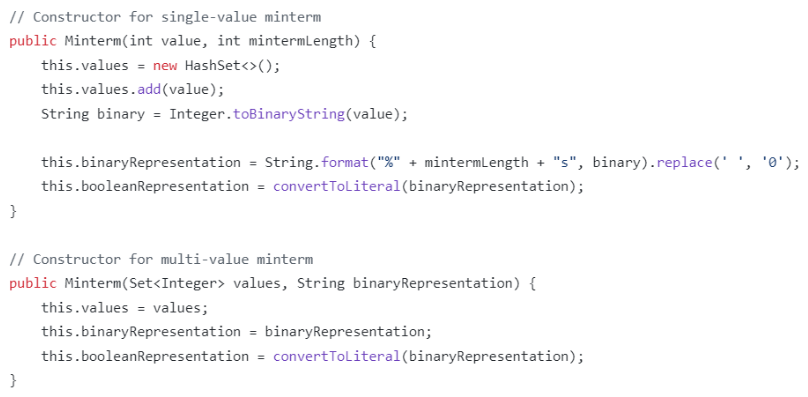
4 Explanation of design

4.1 Inheritance

Inheritance is used for the creation of UI, where we use classes of Java Swing like JDialog, JFrame, JPanel to create different screens.

4.2 Overloading

Constructor of class Minterm is overrided for easier construction. One constructor takes integer as input, while one takes a set of integer and its binary representation (which is pre-calculated).



```
// Constructor for single-value minterm
public Minterm(int value, int mintermLength) {
    this.values = new HashSet<>();
    this.values.add(value);
    String binary = Integer.toBinaryString(value);

    this.binaryRepresentation = String.format("%" + mintermLength + "s", binary).replace(' ', '0');
    this.booleanRepresentation = convertToLiteral(binaryRepresentation);
}

// Constructor for multi-value minterm
public Minterm(Set<Integer> values, String binaryRepresentation) {
    this.values = values;
    this.binaryRepresentation = binaryRepresentation;
    this.booleanRepresentation = convertToLiteral(binaryRepresentation);
}
```

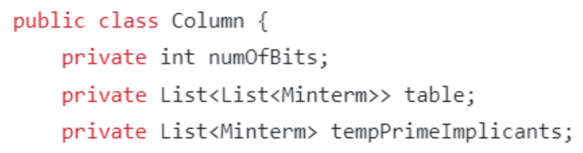
Figure 9: Aggregation relation between Column and Minterm

4.3 Encapsulation

Most of classes' attributes can only be accessed by getter methods, and only the necessary ones can be acquired through methods.

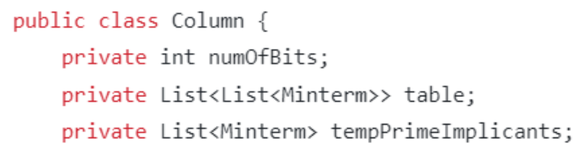
4.4 Aggregation

The ColumnTable contains a list of Column, while Column contains a 2D array of Minterms.



```
public class Column {
    private int numOfBits;
    private List<List<Minterm>> table;
    private List<Minterm> tempPrimeImplicants;
```

Figure 10: Aggregation relation between Column and Minterm



```
public class Column {
    private int numOfBits;
    private List<List<Minterm>> table;
    private List<Minterm> tempPrimeImplicants;
```

Figure 11: Aggregation relation between ColumnTable and Column

5 Conclusion

The Quine-McCluskey method is a powerful and precise tool for minimizing Boolean functions, particularly advantageous when dealing with a large number of variables where traditional methods like Karnaugh maps fall short. Despite its computational demands, its systematic approach ensures thorough minimization, making it a valuable technique in digital logic design and computer engineering