TECHNICAL MANUAL

QUINE-MCCLUSKEY CALCULATOR

DEVELOPED BY:
ACOSTA, ARIANNE JAYNE D.
BONIFACIO, CHRISTIAN JESSE Q.

TABLE OF CONTENTS

I. INTRODUCTION

1.1 The QuineMcCluskeyCalculator 3

II. THE "TERM.JAVA" CLASS

2.1 Fields 5

2.2 Constructors 5

2.3 Methods 6

III. THE "QUINEMCCLUSKEYCALCULATOR.JAVA" CLASS

3.1 Fields 7

3.2 Constructors 8

3.3 Nested Class 8

3.4 Methods 9

IV. THE "MAINGUI.JAVA" & "QMCController" CLASS

4.1 Method in MainGUI.java 18

4.2 Methods in QMCControler.java 19

APPENDIX A: Glossary 23

APPENDIX B: JavaDoc 24

INTRODUCTION

Introducing the **Quine-McCluskey Calculator**, a sophisticated Java software crafted to aid in comprehending and utilizing the Quine-McCluskey method for reducing Boolean functions. Developed by Arianne Jayne Acosta and Christian Jesse Bonifacio, this program offers a robust solution for simplifying complex Boolean functions.

The Quine-McCluskey method is a crucial technique in digital design and Boolean algebra for minimizing logical functions. By iteratively combining terms and identifying essential prime implicants, the method produces a simplified Boolean expression that retains the same logic as the original but with fewer terms. This simulator aims to demystify the Quine-McCluskey method, making it accessible to students, educators, and professionals in the field of digital logic design.

The software undertakes a **dual-phase approach** to achieve Boolean function simplification. Initially, it utilizes the Quine-McCluskey method to detect prime implicants and iteratively enhance the expression. The subsequent stage entails additional refinement through techniques like row dominance, column dominance, and the utilization of **Petrick's method**.

This software accommodates a wide-ranging audience encompassing individuals engaged in digital logic design, computer science, and associated disciplines. Whether you're a student wrestling with Boolean algebra or a professional in search of a dependable tool for function simplification, this program is tailored to suit your requirements.

THE QUINE-MCCLUSKEY PACKAGE

The QMC Package, Quine-McCluskey Calculator Package, includes all essential files required for constructing the Quine-McCluskey Method Simulation program. It consists of three crucial classes: the Term class, the QuineMcCluskeyCalculator class, and the MainGUI class.

TERM.JAVA

This class includes the required fields and methods for representing terms within Boolean functions, facilitating the solving process in the QuineMcCluskeyCalculator class.

Q-M-C.JAVA

Serving as the primary solver, this class offers all necessary fields and methods for simplifying Boolean functions using the Quine-McCluskey Tabular Method.

MAINGUI.JAVA

This class houses the program's main method and provides a Graphical User Interface (GUI) with controls for user input and efficient output of solutions

This technical manual primarily focuses on these classes, detailing their contents within Java files and examining each element's significance. Additionally, it will elucidate methods utilized from other classes and discuss the implementation of Petrick’s method for simplification

THE "TERM.JAVA" CLASS

FIELDS

1. private String term

This field holds the present string representation of the Boolean function Term in binary format, utilizing '0' for zeros, '1' for ones, and '-' for dashes.

2. private int ones

This field contains the count of ones found in the binary representation of the Term.

3. private ArrayList<Integer> minterms

This field retains all the grouped integer values of minterms that depict the Term within an ArrayList of integers.

CONSTRUCTORS

1. public Term (Term term1, Term term2)

This constructor is utilized when a new Term is recognized through the merging of two properly grouped Terms.

CODE DESCRIPTION:

Both parameter Terms are examined by iterating through their respective characters in their binary string representations. A StringBuffer constructs the new term, copying each matching character while substituting non-matching characters with a dash (-). The resulting string is then assigned to the term field of the new Term object. The number of ones is counted and assigned to the object's ones field. Finally, the integer value minterms of both grouped Terms are added to the new Term object's nums ArrayList field.

CODE:

```
public Term(Term term1, Term term2) {
    // Create a new term by comparing and replacing non-matching characters
    StringBuffer temp = new StringBuffer();
    for (int i = 0; i < term1.getString().length(); i++) {
        if (term1.getString().charAt(i) != term2.getString().charAt(i))
            temp.append("-");
        else
            temp.append(term1.getString().charAt(i));
    }
    this.term = temp.toString();

    // Count the number of ones in the new term
    ones = 0;
    for (int i = 0; i < term.length(); i++) {
        if (this.term.charAt(i) == '1')
            ones++;
    }
```

THE "TERM.JAVA" CLASS

2. public Term (int value, int length)

This constructor is employed to initialize a new Term derived from an integer value representing the minterm and the maximum possible length of bits in a Boolean function.

CODE DESCRIPTION:

The value parameter undergoes conversion to a string via the Integer.toBinaryString() method. Subsequently, it accounts for the length parameter, padding zeroes to the binary string until it matches the maximum length of the Boolean function. As this constructor pertains to a single minterm value, it includes the integer value of the minterm in the nums ArrayList field of the Term object for subsequent groupings. Furthermore, it calculates the count of ones within the binary string and assigns this value to the ones field of the object.

CODE:

```
public Term(int value, int length) {
    // Convert minterm to binary string
    String binary = Integer.toBinaryString(value);

    // Left-pad zeroes if binary string length does not match desired length
    StringBuffer temp = new StringBuffer(binary);
    while (temp.length() != length) {
        temp.insert( offset: 0,  i: 0);
    }
    this.term = temp.toString();

    // Initialize array list to store minterm numbers
    mintermNumbers = new ArrayList<Integer>();
    mintermNumbers.add(value);

    // Count the number of ones in the binary representation
    ones = 0;
    for (int i = 0; i < term.length(); i++) {
        if(term.charAt(i) == '1')
            ones++;
    }
}
```

METHODS

1. String getString()

This method retrieves the binary representation of the Term in String format from its term field.

2. ArrayList<Integer> getMinterms()

This method provides an ArrayList of integers that represent the binary string of the Term, retrieved from the Term's nums field.

3. int getNumOnes()

This method retrieves the count of ones found in the binary string representation of the Term.

THE "QUINE-MCCLUSKEY CALCULATOR.JAVA" CLASS

FIELDS

1. private Terms[ ] terms

This field holds the array of Terms required for solving the problem.

2. private ArrayList<Integer> minterms

This field retains the ArrayList of integer values corresponding to the minterms input by the user.

3. private int maxLength

This field holds the maximum length allowed in the Boolean function for solving the problem.

4. private ArrayList<String>[ ] solution

This field stores an array of ArrayLists that accumulate solutions generated throughout the program's execution.

5. private ArrayList<String> primeImplicants

This field maintains an ArrayList that collects the prime implicants accumulated during the program's execution.

6. private ArrayList<Term> finalTerms

This field holds an ArrayList containing every term required for the second stage of the solution process.

7. private ArrayList<ArrayList<Term>> firstStep

This field contains an ArrayList of ArrayLists that store Terms collected from the first stage of the solution process.

8. private ArrayList<HashSet<String>> checkedFirstStep

This field holds an ArrayList of HashSets that store checked Terms collected from the first stage of the solution process.

9. private ArrayList<String> simplified

This field maintains an ArrayList that contains associated Terms eligible for simplification using Petrick's method.

THE "QUINE-MCCLUSKEY CALCULATOR.JAVA" CLASS

CONSTRUCTORS

1. public QuineMcCluskeyCalculator (String mintermsStr)

This constructor is utilized to initialize an object that will execute the Quine-McCluskey tabular method.

CODE DESCRIPTION:

The parameter String mintermsStr is initially transformed into an integer array. In the case where the array contains duplicates, an error message dialog is displayed to handle the issue. If not, the process continues and sorts the array. The maxLength is determined from the binary representation of the highest minterm in the array. Subsequently, the fields are initialized, with the minterms integer array stored in the object field, and each minterm is converted into Terms for the terms object field. The terms field is then sorted based on their number of ones for later solving purposes.

CODE:

```
public QuineMcCluskeyCalculator(String mintermsStr) {

    int[] minterms = convertString(mintermsStr);

    Arrays.sort(minterms);

    maxLength = Integer.toBinaryString(minterms[minterms.length - 1]).length();

    this.minterms = new ArrayList<>();
    primeImplicants = new ArrayList<String>();
    firstStep = new ArrayList<ArrayList<Term>[]>();
    checkedFirstStep = new ArrayList<HashSet<String>>();
    simplified = new ArrayList<String>();

    Term[] temp = new Term[minterms.length];
    int k = 0; // Index in temp array.
    for (int i = 0; i < minterms.length; i++) {
        temp[k++] = new Term(minterms[i], maxLength);
        this.minterms.add(minterms[i]);
    }

    terms = new Term[k];
    for (int i = 0; i < k; i++) {
        terms[i] = temp[i];
    }

    Arrays.sort(terms, new OnesComparator());
}
```

NESTED CLASS

1. private class OnesComparator implements Comparator<Term>

This constructor is used to set up an object that will carry out the Quine-McCluskey tabular method.

THE "QUINE-MCCLUSKEY CALCULATOR.JAVA" CLASS

1.1 public int compare (Term a, Term b)

This method simply compares two Terms based on their number of ones.

RETURN: Returns 0 if the number of ones is the same, a positive integer value if "a" is greater than "b", and a negative integer value if "a" is less than "b".

CODE:

```
private class OnesComparator implements Comparator<Term>{
    /**
     * Compares two terms based on their number of ones.
     * @param a First term to compare.
     * @param b Second term to compare.
     * @return 0 if same number of ones, positive int if a > b, negative int if a < b
     */
    Christian Jesse Bonifacio
    @Override
    public int compare (Term a, Term b) {
        return a.getNumOnes() - b.getNumOnes();
    }
}
```

METHODS

1. private int [ ] convertString (String s)

This method transforms the minterms String entered by the user into an integer array if it meets the criteria for validity.

CODE DESCRIPTION:

First, the commas within String s are replaced with spaces. If s is empty, the method returns an empty array. Otherwise, it proceeds by splitting the string into substrings delimited by spaces and storing them in an array. It attempts to parse the substrings into integers until reaching the end of the array length. If s contains characters other than digits, spaces, and commas, it triggers an error message dialog for invalid input. Furthermore, it checks the parsed integers array for duplicates using a HashSet structure. If duplicates are found, an error message dialog for encountered duplicates is displayed. If all elements in the parsed integer array are valid, it returns this array.

CODE:

```
private int[] convertString(String s) {
    s = s.replace( target: ", ", replacement: ",");
    s = s.replace( target: ",", replacement: " ");

    if (s.trim().equals("")) {
        return new int[] {};
    }

    String[] a = s.trim().split( regex: "\\s+");
    int[] t = new int[a.length]; // Array of minterms.

    for (int i = 0; i < t.length; i++) {
        try {
            int temp = Integer.parseInt(a[i]);
            t[i] = temp;
        } catch (Exception e) {
            if (!s.matches( regex: "[\\d,\\s]+"))
            {
                QMCController window = new QMCController();
                window.showAlert( msg: "Input must only contain integers & be comma or space delimited\ne.g. 1,2,3; 1 2 3; 1, 2, 3\nYour input: " + s,
                return new int[] {};
            }

        }
    }

    HashSet<Integer> dup = new HashSet<>();
    for (int i = 0; i < t.length; i++) {
        if (dup.contains(t[i])) {
            QMCController window = new QMCController();
            window.showAlert( msg: "List of minterms must not have duplicates.", title: "Invalid Input");
            break;
        }
        dup.add(t[i]);
    }

    return t;
}
```

THE "QUINE-MCCLUSKEY CALCULATOR.JAVA" CLASS

2. private ArrayList<Term> [ ] group (Term [ ] terms)

This method groups arrays of Terms in an ArrayList array based on their number of ones.

CODE DESCRIPTION:

The method begins by creating the groups array, which is an array consisting of ArrayLists categorized by the number of ones they contain, with the array's size corresponding to the maximum number of ones. Subsequently, each ArrayList in the array is initialized, followed by grouping the terms together in an ArrayList based on the number of ones they possess, while maintaining their respective positions in the groups array. Finally, the method returns the groups array.

CODE:

```
private ArrayList<Term>[] group(Term[] terms) {

    ArrayList<Term>[] groups = new ArrayList[terms[terms.length - 1].getNumOnes() + 1];

    for (int i = 0; i < groups.length; i++) {
        groups[i] = new ArrayList<>();
    }

    for (int i = 0; i < terms.length; i++) {
        int k = terms[i].getNumOnes();
        groups[k].add(terms[i]);
    }

    return groups;
}
```

3. boolean checkRepeats (int [ ] m)

This method verifies whether there are duplicate elements within the provided minterms integer array.

CODE DESCRIPTION:

The method employs a temporary HashSet to automatically filter out duplicates. It iterates through each element of the integer array m, using the add() method of HashSet. If the add() method encounters a duplicate element, it returns false. In such a case, the method returns false; otherwise, it returns true.

CODE:

```
HashSet<Integer> dup = new HashSet<>();
for (int i = 0; i < t.length; i++) {
    if (dup.contains(t[i])) {
        JOptionPane.showMessageDialog( parentComponent: null, message: "Duplicates encountered. Please try again.", title: "Error", JOptionPane.ERROR_MESSAGE);
    }
    dup.add(t[i]);
}

return t;
```

3. boolean checkValidity (Term term1, Term term2)

This method checks if two terms are valid for grouping.

CODE DESCRIPTION:

It examines both Terms initially to confirm if they have the same length in their binary string form; if not, it promptly returns false. Subsequently, it tallies the differing positions in the string; if a dash (-) is matched with a zero or one, it immediately returns false. However, if only one differing position is found, it returns true, otherwise false.

THE "QUINE-MCCLUSKEY CALCULATOR.JAVA" CLASS

CODE:

```
boolean checkValidity (Term term1, Term term2) {
    if (term1.getString().length() != term2.getString().length())
        return false;

    int k = 0;
    for (int i = 0; i < term1.getString().length(); i++) {
        if (term1.getString().charAt(i) == '-' && term2.getString().charAt(i) != '-')
            return false;
        else if (term1.getString().charAt(i) != '-' && term2.getString().charAt(i) == '-')
            return false;
        else if (term1.getString().charAt(i) != term2.getString().charAt(i))
            k++;
    }

    if (k != 1)
        return false;
    else
        return true;
}
```

4. boolean contains (Term term1, Term term2)

This method verifies whether all the numbers in two terms are present in another term.

CODE DESCRIPTION:

Initially, the method examines both Terms to determine if the number of minterms grouped in term1 is less than or equal to that of term2; if so, it promptly returns false. Otherwise, it gathers all numbers associated with term1 and term2 into corresponding ArrayLists, denoted as "a" and "b". If all numbers in "b" are found in "a", the method returns true; otherwise, it returns false.

CODE:

```
boolean contains(Term term1, Term term2) {
    if (term1.getMintermNumbers().size() <= term2.getMintermNumbers().size()) {
        return false;
    }

    ArrayList<Integer> a = term1.getMintermNumbers();
    ArrayList<Integer> b = term2.getMintermNumbers();

    if (a.containsAll(b))
        return true;
    else
        return false;
}
```

5. private boolean identifyPrimeImplicants()

This method is employed during the second stage of the solving process. It serves to recognize prime implicants and includes them in the primeimplicants ArrayList field. Additionally, it removes the identified prime implicants from the minterms and finalTerms ArrayLists.

CODE DESCRIPTION:

Initially, it initializes an array of ArrayLists named "columns" to store indices of final terms matching each minterm. The "columns" array is populated with indices of those in the "finalTerms" field that correspond to each minterm. Subsequently, it iteratively examines each minterm's matched final Terms. If a minterm has only one matching final Term, it is recognized as a prime implicant, and the numbers of the identified prime implicant are gathered. Associated minterms are then removed from the object's "minterms" field. The identified prime implicant is added to the object's "primeImplicants" field, and subsequently removed from the "finalTerms" field. The loop terminates upon identifying the first prime implicant, and returns true if at least one prime implicant is identified.

THE "QUINE-MCCLUSKEY CALCULATOR.JAVA" CLASS

CODE:

```
private boolean identifyPrimeImplicants(){
    ArrayList<Integer>[] columns = new ArrayList[minterms.size()];

    for (int i = 0; i < minterms.size(); i++) {
        columns[i] = new ArrayList();
        for (int j = 0; j < finalTerms.size(); j++) {
            if (finalTerms.get(j).getMintermNumbers().contains(minterms.get(i))) {
                columns[i].add(j);
            }
        }
    }
    boolean isPrimeImplicant = false;

    for (int i = 0; i < minterms.size(); i++) {
        if (columns[i].size() == 1) {
            isPrimeImplicant = true;

            ArrayList<Integer> del = finalTerms.get(columns[i].get(0)).getMintermNumbers();

            for (int i = 0; i < minterms.size(); i++) {
                if (del.contains(minterms.get(j))) {
                    minterms.remove(j);
                    i--;
                }
            }

            primeImplicants.add(finalTerms.get(columns[i].get(0)).getString());
            finalTerms.remove(columns[i].get(0).intValue());
            break;
```

5. private boolean rowDominance()

This method is utilized during the second stage of the solving process to identify dominating rows and eliminate their contents from the object's "minterms" and "finalTerms" fields.

CODE DESCRIPTION:

Initially, a boolean variable "flag" is declared and initialized as false. The method utilizes nested loops to attempt to identify and eliminate dominating rows from the "finalTerms" ArrayList. If the current row being read contains all elements of another row, it removes the other row. Conversely, if another row contains all elements of the current row, it removes the current row. If a row is removed, the flag is then set to true, and the inner nested loop breaks. Once the entire loop concludes, the method returns the "flag" variable.

CODE:

```
private boolean rowDominance(){
    boolean flag = false;

    for (int i = 0; i < finalTerms.size() - 1; i++) {
        for (int j = i + 1; j < finalTerms.size(); j++) {
            if (contains(finalTerms.get(i), finalTerms.get(j))) {
                finalTerms.remove(j);
                j--;
                flag = true;
            } else if (contains(finalTerms.get(j), finalTerms.get(i))) {
                finalTerms.remove(i);
                i--;
                flag = true;
                break;
            }
        }
    }
    return flag;
}
```

THE "QUINE-MCCLUSKEY CALCULATOR.JAVA" CLASS

5. private boolean columnDominance()

This method is employed during the second stage of the solving process to recognize dominating columns and erase their contents from the object's "minterms" and "finalTerms" fields.

CODE DESCRIPTION:

Initially, a boolean variable "flag" is declared and set to false. The method then constructs a table using a 2D ArrayList structure called "columns". It iterates through the "minterms" and "finalTerms" fields, populating the "columns" structure accordingly. Subsequently, it adds columns based on the indices of minterms associated with finalTerms. The method proceeds by utilizing nested loops to detect and eliminate dominating columns from the "columns" structure. If the current column being examined contains all elements of another column, it removes the other column. Conversely, if another column contains all elements of the current column, it removes the current column. If a column is removed, the flag is set to true, and the inner nested loop breaks. Once the entire loop concludes, the method returns the "flag" variable.

CODE:

```
private boolean columnDominance(){
    boolean flag = false;

    ArrayList<ArrayList<Integer>> columns = new ArrayList<>();

    for (int i = 0; i < minterms.size(); i++){
        columns.add(new ArrayList<Integer>());
        for (int j = 0; j < finalTerms.size(); j++){
            if (finalTerms.get(j).getMintermNumbers().contains(minterms.get(i)))
                columns.get(i).add(j);
        }
    }

    for (int i = 0; i < columns.size(); i++) {
        for (int j = i + 1; j < columns.size(); j++) {
            if (columns.get(j).containsAll(columns.get(i)) && columns.get(j).size() > columns.get(i).size()) {
                columns.remove(j);
                minterms.remove(j);
                i--;
                flag = true;
            } else if (columns.get(i).containsAll(columns.get(j)) && columns.get(i).size() > columns.get(j).size()) {
                columns.remove(i);
                minterms.remove(i);
                i--;
                flag = true;
                break;
            }
        }
    }
```

6. String mix (String str1, String str2)

This method combines terms and simplifies duplicates based on Boolean expression properties.

CODE DESCRIPTION:

The method employs a HashSet of characters to eliminate duplicates since in Boolean algebra, products and sums of the same variable result in the variable itself. It operates by adding the characters of the first String and the second String, leveraging the HashSet structure to disregard duplicates. An Iterator is subsequently utilized to build the resulting String by concatenating the characters in the HashSet "r".

THE "QUINE-MCCLUSKEY CALCULATOR.JAVA" CLASS

CODE:

```
String mix (String str1, String str2){
    HashSet<Character> r = new HashSet<>();

    for (int i = 0; i < str1.length(); i++)
        r.add(str1.charAt(i));

    for (int i = 0; i < str2.length(); i++)
        r.add(str2.charAt(i));

    StringBuilder result = new StringBuilder();
    for (Iterator<Character> i = r.iterator(); i.hasNext();)
        result.append(i.next());

    return result.toString();
}
```

6. HashSet<String> multiply (HashSet<String> [ ] p, int k)

The simplify() method utilizes this function to implement Petrick's method. It performs multiplication of elements from sets located at adjacent indices in the HashSet array and recursively calculates the Boolean product.

CODE DESCRIPTION:

The base case of this recursive method checks if the index "k" is greater than or equal to the length of the array "p" minus 1. If this condition holds true, it returns the set at the current index "k", effectively halting the recursion when the end of the array is reached. The multiplication process begins by initializing the resulting HashSet. It then utilizes a for loop with an Iterator to iterate through the elements of the "kth" element of the HashSet array "p". A nested loop is subsequently executed to scan for the "(k+1)th" element of "p". It employs the mix() method to combine the Strings found at "p[k]" and "p[k+1]". Once the for loop concludes, it updates the "(k+1)th" element in "p" with the resulting mixed String. The method then recursively calls itself with the updated "p" array and the next index "k+1", until it reaches the base case. Finally, it returns the HashSet array.

CODE:

```
HashSet<String> multiply(HashSet<String>[] p, int k){
    if (k >= p.length - 1)
        return p[k];

    HashSet<String> s = new HashSet<>();

    for (Iterator<String> t = p[k].iterator(); t.hasNext();) {
        String temp2 = t.next();
        for (Iterator<String> g = p[k + 1].iterator(); g.hasNext();) {
            String temp3 = g.next();
            s.add(mix(temp2, temp3));
        }
    }
    p[k + 1] = s;
    return multiply(p, k: k + 1);
}
```

7. void simplify ()

This method simplifies the Boolean function once the object's solution field is populated. Primarily part of the second stage of solving, it employs Petrick's method.

CODE DESCRIPTION:

The method commences by generating a temporary array of temporary HashSets and collects associated minterms from corresponding final Terms. Subsequently, it utilizes the multiply() method to compute the product of all the sets in the array and stores the result in a string HashSet. Then, it scrutinizes this HashSet, identifying the minimum Terms essential for the simplification of the Boolean function. Finally, it appends these minimum terms to the object's solution field.

CODE:

```
void simplify(){
    HashSet<String>[] temp = new HashSet[minterms.size()];

    //Construct temp array containing sets of associated characters for minterms in finalTerms
    for (int i = 0; i < minterms.size(); i++) {
        temp[i] = new HashSet<>();
        for (int j = 0; j < finalTerms.size(); j++) {
            if (finalTerms.get(j).getMintermNumbers().contains(minterms.get(i))) {
                char t = (char) ('a' + j);
                simplified.add(t + ": " + finalTerms.get(j).getString());
                temp[i].add(t + "");
            }
        }
    }

    HashSet<String> finalResult = multiply(temp, 0);

    int min = -1;
    int count = 0;
    for (Iterator<String> t = finalResult.iterator(); t.hasNext();) {
        String m = t.next();
        if (min == -1 || m.length() < min) {
            min = m.length();
            count = 1;
        } else if (min == m.length()) {
            count++;
        }
    }
```

```
solution = new ArrayList[count];
int k = 0;
for (Iterator<String> t = finalResult.iterator(); t.hasNext();) {
    String c = t.next();
    if (c.length() == min) {
        solution[k] = new ArrayList<>();
        for (int i = 0; i < c.length(); i++) {
            solution[k].add(finalTerms.get((int) c.charAt(i) - 'a').getString());
        }
        for (int i = 0; i < primeImplicants.size(); i++) {
            solution[k].add(primeImplicants.get(i));
        }
        k++;
    }
}
```

8. public void solve ()

This method constitutes the primary solving process of the Quine-McCluskey tabular method, essentially embodying the first stage of the method's solution.

CODE DESCRIPTION:

The method begins by initializing an ArrayList of unchecked terms, an ArrayList array to store the gathered first list of grouped Terms, and another ArrayList array to store resulting terms of each iteration later on. The current grouped Terms are then added to the firstStep field.

Following this, the method enters a do-while loop where it continues looping as long as the result array is not empty and its length is greater than 1. Within the loop, a HashSet of checked Terms is initialized, and the result ArrayList is reset. Nested for loops then check each element in adjacently indexed groups to determine if they can be validly grouped. If true, the terms are appended to the checked HashSet, and a new Term object is created from the grouped terms and added to the results array. If the results array is not empty, the unchecked terms list is updated, and the result and checked ArrayLists are added to the object's firstStep and checkedFirstStep fields.

Upon completion of the do-while loop, the method copies resulting minterms into a new ArrayList along with the unchecked terms to the finalTerms field. Finally, the method proceeds to the solveSecond() method for the second-stage of solving.

In summary, the method initializes and processes terms to identify valid groupings in a loop until no further simplification is possible. It then proceeds to the second stage of solving.

THE "QUINE-MCCLUSKEY CALCULATOR.JAVA" CLASS

CODE:

```
public void solve(){
    ArrayList<Term> unchecked = new ArrayList<>();

    ArrayList<Term>[] list = group(this.terms);

    ArrayList<Term>[] result;

    firstStep.add(list);

    boolean insert = true;

    do {
        HashSet<String> checked= new HashSet<>();

        result = new ArrayList[list.length - 1];

        ArrayList<String> temp;
        insert = false;

        for (int i = 0; i < list.length - 1; i++){
            result[i] = new ArrayList<>();
            temp = new ArrayList<>();

            for (int j = 0; j < list[i].size(); j++){
                for (int k = 0; k < list[i + 1].size(); k++){
                    if (checkValidity(list[i].get(j), list[i + 1].get(k))) {
                        checked.add(list[i].get(j).getString());
                        checked.add(list[i+1].get(k).getString());
                        Term n = new Term(list[i].get(j), list[i+1].get(k));

                        if (!temp.contains(n.getString())) {
                            result[i].add(n);
                            insert = true;
                        }
                        temp.add(n.getString());

                    }
                }
            }
        }

        if (insert) {
            for (int i = 0; i < list.length; i++) {
                for (int j = 0; j < list[i].size(); j++) {
                    if (!checked.contains(list[i].get(j).getString())) {
                        unchecked.add(list[i].get(j));
                    }
                }
            }
            list = result;

            firstStep.add(list);
            checkedFirstStep.add(checked);
        }
    } while (insert && list.length > 1);
```

8. public void solveSecond ()

This method continues the solving process initiated by solve(). It essentially executes the second stage of the solution using the designated method.

CODE DESCRIPTION:

The method initiates with nested conditional statements. Firstly, it checks if there are any identified prime implicants in the object's finalTerms field. If none are found, it proceeds to verify for row dominance among the final Terms. If no dominating rows are found, it then examines column dominance. If neither dominating rows nor columns are found, it proceeds to execute the simplify() method and concludes the process.

However, if any of these conditions are met, the method then verifies if there are still elements remaining in the object's minterms field. If so, it recursively calls this method until all elements are removed. Once there are no more elements in the minterms field, the found prime implicants are added to the object's solution field.

CODE:

```
public void solveSecond(){
    if (!identifyPrimeImplicants()) {
        if (!rowDominance()) {
            if (!columnDominance()) {
                simplify();
                return;
            }
        }
    }

    if (minterms.size() != 0)
        solveSecond();
    else {
        solution = new ArrayList[1];
        solution[0] = primeImplicants;
    }
}
```

THE "QUINE-MCCLUSKEY CALCULATOR.JAVA" CLASS

9. String toStandardForm (String s)

This method transforms the binary string representation of an element in the object's solution field into its standard form.

CODE DESCRIPTION:

This method utilizes a StringBuilder to construct the resulting String. It iterates through every character in the parameter String using a for loop, with the length of the String as the maximum length and the number of possible variables. The variables begin with 'A', and each increment in the for loop proceeds to the next letter. If the current character is a dash (-), it is ignored and nothing is appended. If the current character is '1', the current variable is appended. If the current character is '0', the current variable with a prime (') is appended. If the resulting string is empty, 1 is appended to represent a constant. Finally, the built String is returned.

CODE:

```
String toStandardForm(String s) {
    StringBuilder r = new StringBuilder();

    for (int i = 0; i < s.length(); i++) {

        if (s.charAt(i) == '-') {
            continue;
        }

        else if (s.charAt(i) == '1') {
            r.append((char) ('A' + i));
        }
        else {
            r.append((char) ('A' + i));
            r.append('\'');
        }
    }

    if (r.toString().length() == 0) {
        r.append("1");
    }
    return r.toString();
```

10. public String printResults (String[ ] variables)

This method transforms the binary string representation of an element in the object's solution field into its standard form.

CODE DESCRIPTION:

This method employs a StringBuilder to construct the resulting String. It iterates through every index in the solution array field using a for loop. If the length of the solution is greater than 1, multiple solution Strings are appended. Each element from the inner ArrayLists of the solution field is then converted to standard form and separated as a sum of products with '+'. Lastly, variables in the resulting String are replaced according to the input from the variables parameter, respecting their indices.

CODE:

```
public String printResults(String[] variables) {
    StringBuilder printedAnswer = new StringBuilder();
    for (int i = 0; i < solution.length; i++) {

        if (solution.length == 1)
            printedAnswer.append("Solution:").append("\n");
        else
            printedAnswer.append("Solution #").append(i+1).append(":").append("\n");

        StringBuilder finalAnswer = new StringBuilder();
        for (int j = 0; j < solution[i].size(); j++) {
            finalAnswer.append(toStandardForm(solution[i].get(j)));
            if (j != solution[i].size() - 1) {
                finalAnswer.append(" + ");
            }
        }
```

THE "MAINGUI.JAVA" CLASS

METHODS

1. public void start(Stage stage) throws Exception

This code initializes a JavaFX application by loading an FXML file to create the user interface. It sets up the scene with the loaded UI elements, assigns a title to the stage, sets an icon, configures the stage to be non-resizable, and displays the stage to start the application.

CODE DESCRIPTION:

This code is the start() method of a JavaFX application. It begins by obtaining the URL of the FXML file used to define the application's user interface. If the FXML file is not found, it throws an IOException. Next, it loads the FXML file into a Parent object, which represents the root node of the user interface. Then, it creates a Scene object using the root node and sets the title of the stage to "Quine-McCluskey Calculator". It also sets an icon for the stage using an image file. The stage is configured to be non-resizable, and finally, the stage is displayed to start the application. Overall, this code initializes and sets up the graphical user interface (GUI) of the Quine-McCluskey Calculator application using JavaFX.

CODE:

```
@Override
public void start(Stage stage) throws Exception {
    URL fxmlUrl = getClass().getResource( name: "/org/example/cs130mpfinal/MainGUI.fxml");
    System.out.println("URL" + fxmlUrl); // Validates that url is found and is not null

    if (fxmlUrl == null) {
        throw new IOException("FXML file not found");
    }

    Parent root = FXMLLoader.load(fxmlUrl);
    Scene scene = new Scene(root);

    Image icon = new Image(Objects.requireNonNull(getClass().getResourceAsStream( name: "logo.png")));
    stage.setTitle("Quine-McCluskey Calculator");
    stage.getIcons().add(icon);
    stage.setScene(scene);
    scene.getStylesheets().add(Objects.requireNonNull(getClass().getResource( name: "font.css")).toExternalForm());
    stage.setResizable(false);
    stage.show();
}
```

METHODS

1. public void initialize()

This code initializes a GUI component, sets default selections and text, and attaches event handlers to buttons for specific actions.

CODE DESCRIPTION:

This code initializes various components in a graphical user interface (GUI). It creates a ToggleGroup and assigns radio buttons "yes" and "no" to this group. It sets the "no" button as selected by default and hides a custom information element. Additionally, it sets default text in a text field. Furthermore, it attaches event handlers to buttons: "solve" and "clear" buttons trigger handleSolveButton() and handleClearButton() methods respectively. Moreover, when the "yes" button is clicked, it enables editing in a text field and clears its content, while clicking the "no" button disables editing and sets default text in the text field.

CODE:

```
public void initialize()
{
    ToggleGroup group = new ToggleGroup();
    yes.setToggleGroup(group);
    no.setToggleGroup(group);

    no.setSelected(true);
    customInfo.setVisible(false);
    varsInput.setText("A, B, C, D, E, F, G, H, I, J");

    solve.setOnAction(event -> handleSolveButton());
    clear.setOnAction(event -> handleClearButton());
    yes.setOnAction (event -> {
        varsInput.setEditable(true);
        varsInput.clear();
    });
    no.setOnAction (event -> {
        varsInput.setEditable(false);
        varsInput.setText("A, B, C, D, E, F, G, H, I, J");
    });
```

2. private void handleSolveButton()

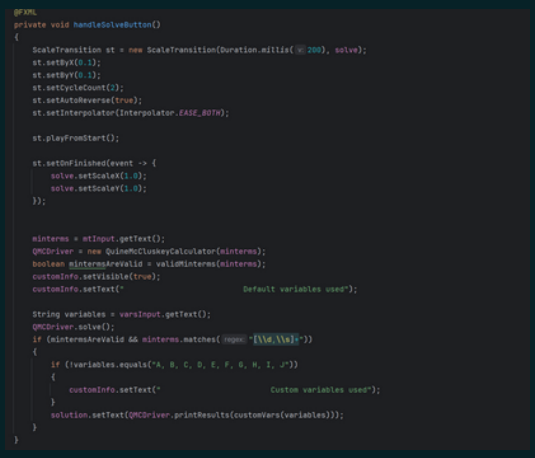
This method handles the "solve" button click event in the GUI. It retrieves user input for minterms and variables, creates a QuineMcCluskeyCalculator object, solves the problem, and displays the solution if the input is valid.

CODE DESCRIPTION:

This method handles the action triggered by clicking the "solve" button in the GUI. It retrieves input from text fields, creates a QuineMcCluskeyCalculator object with the provided minterms, checks if the minterms are valid, and displays custom information. Then, it retrieves variables input, solves the problem using the QuineMcCluskeyCalculator object, and displays the solution if the minterms are valid and properly formatted.

THE "OMCCONTROLLER" CLASS

CODE:



```
@FXML
private void handleSolveButton()
{
    ScaleTransition st = new ScaleTransition(Duration.millis( v: 200), solve);
    st.setByX(0.1);
    st.setByY(0.1);
    st.setCycleCount(2);
    st.setAutoReverse(true);
    st.setInterpolator(Interpolator.EASE_BOTH);

    st.playFromStart();

    st.setOnFinished(event -> {
        solve.setScaleX(1.0);
        solve.setScaleY(1.0);
    });


    minterms = mtInput.getText();
    QMCDriver = new QuineMcCluskeyCalculator(minterms);
    boolean mintermsAreValid = validMinterms(minterms);
    customInfo.setVisible(true);
    customInfo.setText("                        Default variables used");

    String variables = varsInput.getText();
    QMCDriver.solve();
    if (mintermsAreValid && minterms.matches( regex: "[\\d,\\s]+"))
    {
        if (!variables.equals("A, B, C, D, E, F, G, H, I, J"))
        {
            customInfo.setText("                        Custom variables used");
        }
        solution.setText(QMCDriver.printResults(customVars(variables)));
    }
}
```

3. private void handleClearButton()

This method handles the action triggered by clicking the "clear" button in the GUI. It clears the text input fields for minterms and solutions, sets default text for variables, resets radio button selection, hides custom information, and disables editing in the variables input field.

CODE DESCRIPTION:

The `handleClearButton()` method resets various components of the GUI when the user clicks the "clear" button. It clears the minterms input field, resets the variables input field to default text, and reverts the selection of the "no" radio button. Additionally, it hides any custom information, disables editing in the variables field, and clears the solution display area. This ensures a clean interface for the user to start fresh or enter new data.

CODE:

```
private void handleClearButton()
{
    mtInput.clear();
    varsInput.setText("A, B, C, D, E, F, G, H, I, J");
    no.setSelected(true);
    customInfo.setVisible(false);
    varsInput.setEditable(false);
    solution.clear();

}
```

4. private void handleHyperlink() throws URISyntaxException, IOException

This method handles a hyperlink action by opening a web browser and navigating to the specified URL ("https://www.desmos.com/scientific"). It uses the `Desktop.getDesktop().browse()` method to achieve this functionality.

CODE DESCRIPTION:

This method is triggered when a hyperlink is clicked in the GUI. It utilizes the `Desktop.getDesktop().browse()` method to open the default web browser and direct it to the specified URL, in this case, "https://www.desmos.com/scientific". This action allows users to access additional resources or external content related to the application's functionality, such as scientific calculators on the Desmos website.

CODE:

```
private void handleHyperlink() throws URISyntaxException, IOException {
    Desktop.getDesktop().browse(new URI("https://www.desmos.com/scientific"));
}
```

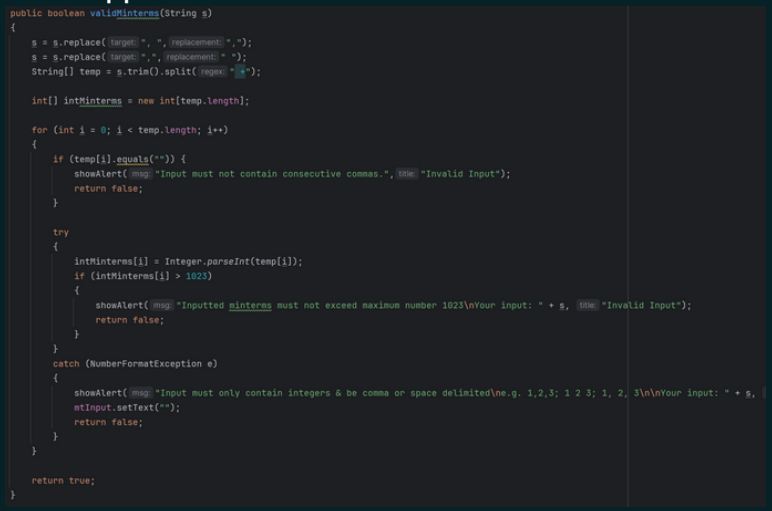
5. public boolean validMinterms(String s

This code validates the input string containing minterms. It checks for formatting errors, such as consecutive commas, and ensures that the minterm values are integers within the specified range (up to 1023). If any errors are found, it displays an alert message indicating the issue and returns false. Otherwise, it returns true to indicate that the input is valid.

CODE DESCRIPTION:

This method serves to validate a string input containing minterms before further processing. Initially, it ensures proper formatting by removing extra spaces around commas and replacing consecutive commas with a single space. Subsequently, the sanitized string is split into individual minterm values, which are parsed into integers. Throughout this process, the method checks for potential errors such as consecutive commas or non-integer values. If any formatting issues are detected, an error message is displayed, and the method returns false, indicating invalid input. Additionally, the method verifies that the minterm values fall within an acceptable range, specifically up to 1023. Should any value exceed this limit, another error message is shown, and the method again returns false. Conversely, if the input passes all validation checks, the method returns true, signaling that the input is valid and can be processed further. Through these steps, the method ensures the integrity of the minterm input, enhancing the reliability of subsequent computations in the application.

CODE:



```
public boolean validMinterms(String s)
{
    s = s.replace( target: ", ", replacement: ",");
    s = s.replace( target: ",", replacement: " ");
    String[] temp = s.trim().split( regex: " +");

    int[] intMinterms = new int[temp.length];

    for (int i = 0; i < temp.length; i++)
    {
        if (temp[i].equals("")) {
            showAlert( msg: "Input must not contain consecutive commas.", title: "Invalid Input");
            return false;
        }

        try
        {
            intMinterms[i] = Integer.parseInt(temp[i]);
            if (intMinterms[i] > 1023)
            {
                showAlert( msg: "Inputted minterms must not exceed maximum number 1023\nYour input: " + s, title: "Invalid Input");
                return false;
            }
        }
        catch (NumberFormatException e)
        {
            showAlert( msg: "Input must only contain integers & be comma or space delimited\ne.g. 1,2,3; 1 2 3; 1, 2, 3\n\nYour input: " + s,
            mtInput.setText("");
            return false;
        }
    }

    return true;
}
```

6. public void showAlert(String msg, String title)

This method creates and displays an error message dialog box with a given message and title.

CODE DESCRIPTION:

This method displays an error message dialog box to the user. It takes two parameters: a message `msg` to be shown in the dialog box and a `title` for the dialog box window. Inside the method, it creates an `Alert` object of type `ERROR`, sets the title, header text (which is null in this case), and the content text to the provided message. Finally, it displays the alert dialog box to the user and waits for the user to acknowledge it before proceeding further.

CODE:

```
public void showAlert(String msg, String title)
{
    Alert alert = new Alert(Alert.AlertType.ERROR);
    alert.setTitle(title);
    alert.setHeaderText(null);
    alert.setContentText(msg);
    alert.showAndWait();
}
```

7. public void showAlert(String msg, String title)

This method processes a string input of custom variables, returning an array of variables. It starts with default variables "A" to "J". If the input is empty, it returns defaults; otherwise, it splits and trims the input. It updates a text field for user notification. Finally, it returns the updated array.

CODE DESCRIPTION:

This method processes a string input containing custom variables and returns an array of variables. It first initializes an array of default variables from "A" to "J". If the input string is empty, it returns the default variables. Otherwise, it splits the input string into individual variables, trims excess spaces, and handles cases where fewer than 10 variables are provided. It also updates a text field to inform the user of any modifications made to the variables. Finally, it returns the updated array of variables.

CODE:

```
public String[] customVars(String s)
{
    String[] variables = {"A", "B", "C", "D", "E", "F", "G", "H", "I", "J"};
    if (s.trim().isEmpty())
    {
        return variables;
    }

    String[] inputVariables = s.replace(", ", ",").replace(",", " ").trim().split(" +");
    int inputLength = inputVariables.length;
    if (inputLength < 10)
    {
        customInfo.setText("Missing variables padded with default");
    }

    for (int i = 0; i < Math.min(inputLength, 10); i++)
    {
        variables[i] = inputVariables[i].length() > 1 ? "(" + inputVariables[i] + ")" : inputVariables[i];
    }
    if (inputLength > 10) {
        customInfo.setText("Using only first 10 inputted variables");
    }
    return variables;
}
```

APPENDIX B: GLOSSARY

ArrayList - In Java, an ArrayList is a resizable array data structure that implements the List interface. It provides methods like add(), get(), and remove() for managing collections of elements.

Binary Form - Binary form represents data using only two symbols, typically 0 and 1. In this program, binary form is used to represent Boolean terms as sequences of binary digits.

GUI (Graphical User Interface) - A GUI is a user interface that enables interaction with electronic devices through graphical elements like windows and buttons. The program uses a GUI to provide a visual interface for user interaction.

HashSet - HashSet is a data structure in Java that stores unique elements using a hash table. It offers constant-time performance for operations like add(), remove(), and contains().

Iterator - An Iterator in Java provides a way to traverse over a collection of elements sequentially without exposing the underlying structure of the collection.

Minterms - In Boolean algebra, minterms represent specific combinations of inputs for which the output is true (1). They are crucial for simplifying Boolean expressions.

Ones Comparator - A custom comparator used to compare terms based on the number of ones in their binary representation.

Petrick's Method - A technique for simplifying Boolean expressions to their minimum sum-of-products form.

Prime Implicants - Essential cubes obtained during the simplification of Boolean functions, representing the minimal terms necessary to cover all minterms.

Quine-McCluskey Method - a tabular method used for simplifying Boolean functions. It involves grouping minterms based on the number of ones in their binary representation and finding prime implicants.

Term - In the context of the Quine McCluskey Method Simulation program, a term refers to a combination of minterms in binary form used in Boolean algebra.

Term Length - refers to the number of bits or digits in the binary representation of a term. It is a measure of the complexity of the Boolean expression.

APPENDIX B: JAVADOC

This section presents a compilation of tables sourced from the generated JavaDoc of the entire package, offering a condensed overview of the package's essential content.

Q-M-C.JAVA

Field Summary

Fields

Modifier and Type	Field	Description
ArrayList<HashSet<String>>	checkedFirstStep	List of Hash sets storing checked terms gathered from the first step of solving.
ArrayList<ArrayList<Term>[]>	firstStep	List of lists storing terms gathered from the first step of solving.
ArrayList<String>	simplified	List storing simplified terms after using Petrick's method.

Constructor Summary

Constructors

Constructor	Description
QuineMcCluskeyCalculator(String mintermsStr)	Constructor for initializing the Quine-McCluskey calculator.

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method	Description
String	printResults(String[] variables)	Build a String for the final resulting solutions to be presented to the user.
void	solve()	First stage of solution using Quine-McCluskey method.
void	solveSecond()	Second stage of solution using Quine-McCluskey method.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Details

firstStep

```
public ArrayList<ArrayList<Term>[]> firstStep
```

List of lists storing terms gathered from the first step of solving.

checkedFirstStep

```
public ArrayList<HashSet<String>> checkedFirstStep
```

List of Hash sets storing checked terms gathered from the first step of solving.

simplified

```
public ArrayList<String> simplified
```

List storing simplified terms after using Petrick's method.

Constructor Details

QuineMcCluskeyCalculator

```
public QuineMcCluskeyCalculator(String mintermsStr)
```

Constructor for initializing the Quine-McCluskey calculator.

Parameters:

`mintermsStr` - A valid string containing the minterms to be solved.

Method Details

solve

```
public void solve()
```

First stage of solution using Quine-McCluskey method. Main solver method of the class to be called.

APPENDIX B: JAVADOC

This section presents a compilation of tables sourced from the generated JavaDoc of the entire package, offering a condensed overview of the package's essential content.

TERM.JAVA

Constructor Summary

Constructors

Constructor	Description
`Term(int value, int length)`	Constructs a Term object from an integer minterm value.
`Term(Term term1, Term term2)`	Constructs a Term object from two grouped terms.

Method Summary

Methods inherited from class java.lang.Object

`clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`, `wait`, `wait`

Constructor Details

Term

```
public Term(int value,
            int length)
```

Constructs a Term object from an integer minterm value.

Parameters:

`value` - The integer value of the minterm.

`length` - The length of the binary string, used for leading zero padding.

Term

```
public Term(Term term1,
            Term term2)
```

Constructs a Term object from two grouped terms.

Parameters:

`term1` - The first term to be grouped.

`term2` - The second term to be grouped.

MEET OUR TEAM

ARIANNE JAYNE ACOSTA

CHRISTIAN JESSE BONIFACIO