Design Document

Zhilin Huang
Zhilinh

To solve the Cryptarithm problem, my design principle is using an Iterator pattern to generate permutations by brute-force to match the situation and requirement that the problem offered.

The design methods of my Iterator pattern are following:

First, create an array list to receive any type of input.
Second, implement Heap's Algorithm to generate permutations.
Third, output each permutation one time to the client code.

Here is the design rationale. For the type of input, because the Heap's Algorithm only considers the index (or position) of elements in the list, the permutation generator will have no limits on the input type. It permits the generality. In this problem, I will pass a list of combination numbers in. For the Heap's Algorithm, I used an iterative method and avoided a recursive method in order to generate every permutation one time and output it to the client code. The reason why I generate permutation in this way is that the data size of permutation could be considerably large (n!), and what we do about each permutation is only one-time test. Therefore, we don't need to store a huge size of data then analyze. So we could use the Iterator pattern to generate all possible solutions fast.

Template Method Client Code:

```
public class Parsing {

    public Parsing(String[] args) { … }

    /**
    * Method to separate every character to determine whether they are
    * legal or not and assign them to variables.
    * @return a variable list of all characters.
    */
    public List<Variable> doParsing() { return List<Variable> vList; }
}

public class ExpressionConstructor {

    public ExpressionConstructor(String[] args, List<Variable> vList) { … }

    /**
    * Method to analyze the composition of an equation.
    * Transform string and operation into Expression to calculate.
```

```
        * @return a list of two Expression result besides the "=".
        */
        public List<Expression> Constructor() {  return List<Expression>; }
    }

    public class Solution {

        public Solution(List<List<Integer>> getcList, List<Variable> vList, List<Expression> result)
{ … }

        /**
         * Method to generate the values of two Expression and determine
         * whether they are equal to output the solutions.
         * @return a list of solutions to output.
         */
        public List<E> Comparison() {
            for List<E> i : ConcreteClass() {
                … //Assign the values of elements of i to variables and expressions.
                if (the equation is true for Cryptarithm) {
                    List<E>.add(i);
                }
            }
            return List<E>;
        }
    }
```

Template Method API: AbstractClass will have a constructor to pass the input, handle its type and create instances needed for combination and permutation. And it will have a method for combination generator and a method for permutation generator. Also it might include the method Parsing, ExpressionConstructor and Solution.

Command Pattern Client API: The interface will describe how the client will use the permutation and what is the type of input from command and which method to implement it. For Cryptarithm, the interface will let concrete strategy method to generate permutation and calculate the value.