Homework #3: From Cryptarithms to Algorithms

Due Sunday, September 25th at 11:59 p.m.

In this assignment, you will build a powerful cryptarithm solver using the expression evaluator library you wrote for Homework 2. A cryptarithm (or alphametic) is a puzzle where you are given an equation with letters instead of digits. For example, one famous cryptarithm published by Henry Dudeney in 1924 is:

```
  SEND
+ MORE
------
 MONEY
```

To solve a cryptarithm you must figure out which digit each letter represents. Cryptarithms typically follow standard rules: The first letter of each word (in the above example, S and M) cannot represent zero, and each letter represents a different digit. Good cryptarithms have exactly one solution.

You can use logic to solve a cryptarithm by hand, but on a computer, brute force works fine due to the small size of the search space. Because each letter represents a different digit there can be at most ten distinct letters, so there are only 10! (or 3,628,800) possible assignments of digits to letters. Because modern computers execute billions of instructions per second, checking 3.6 million possible solutions is easy.

This is a three part assignment. In Part 1, you will design and implement a general purpose permutation generator. In Part 2 you will use your expression library from Homework 2 to write a class representing a cryptarithm. In Part 3 you will write a solve method that generates and checks all possible solutions using your permutation generator from Part 1.

Your learning goals for this assignment are to:

- Demonstrate mastery of earlier learning goals, especially the concepts of information hiding and polymorphism, software design based on informal specifications, and Java coding and testing practices and style.
- Use inheritance, delegation, and design patterns effectively to achieve design flexibility and code reuse.
- Discuss the relative advantages and disadvantages of alternative design choices.
- Gain experience working with a medium-sized application, including programming against existing interfaces and implementations.

Part 1: A reusable permutation generator

A *permutation* of a set of items is an arrangement of the items into an ordered sequence. One way to generate all possible assignments of k digits to letters (to solve a cryptarithm) is to generate all permutations of all size-k subsets of digits, assigning letters by simply matching the letters (in some fixed order) to the digits in each permutation.

To solve cryptarithms for this assignment, we require that you design, implement, and use a permutation generator. It must be a reusable component that is not specific to—and does not depend on—cryptarithms. To build a general permutation generator you must make many design choices, including (but not limited to):

- The representation of a permutation.
- The representation of sets to be permuted, such as arrays, collections, `String`s, or other aggregate data types.
- Architecture-level design for the permutation generator and for how the generator provides access to the resulting permutations.
- The extent to which permutation generation is coupled (or decoupled) to subset generation for input sets.

These choices are complicated by the fact that a permutation generator will typically be used as a component in a brute-force algorithm (such as the cryptarithm solver), and thus the permutation generator needs to be a high-quality, high-performance component. Performance requirements may constrain your architectural decisions, but should not be the only non-functional requirements of your design. You may use any reasonable algorithm to generate permutations, but we recommend a standard technique such as Heap's Algorithm.

Start by designing three versions of a permutation generator, one each using the template method, iterator, and command pattern. (The command pattern is structurally similar to the strategy pattern, with an interface having multiple implementations, each representing some action to be done. For a permutation generator using this pattern, the command can encapsulate the work to be done for each permutation.) For each version, concisely describe the permutation generator's API and write client code that demonstrates how that API would be used to do some work for each permutation of a set. A good API should be easy to use and the resulting client code easy to read. Describe your APIs and write your client code in a short design document, `design.md` or `design.pdf` in your `homework/3` directory.

Implement one of your permutation generator designs, then describe your design rationale in at most 500 words in your design document. Explicitly discuss the designs that you considered, and highlight the design goals and design principles that guided your decisions.

Part 2: Representing cryptarithms

Design and implement a class whose instances represent cryptarithms. Your class should have a constructor that takes a single `String` array representing the cryptarithm. For example, the `String` array `{"SEND", "+", "MORE", "=", "MONEY"}` represents the cryptarithm above. (Command-line arguments will be similarly passed to `main` if you run your program as `java <class name> SEND + MORE = MONEY`.)

Note that mathematical equations can be represented as a pair of expressions, one for each side of the equation. You should use your `Expression` library from Homework 2 to help you represent and evaluate cryptarithms. Notice that each letter in a cryptarithm is essentially a variable, and that you can build an expression that represents (in terms of the variables) each side of a cryptarithm. This allows you to evaluate a potential solution by assigning a value to each variable, evaluating the two expressions that form the cryptarithm's equation, and checking for equality. As a practical matter, it's OK to compare two `double` values for exact equality for this homework, but don't make a habit of it.

For this assignment, a cryptarithm may use addition, subtraction, and/or multiplication. Each side of the cryptarithm may use an arbitrary number of operands and operators. Assume that all operands have equal precedence; this assumption greatly simplifies parsing, so you can parse a cryptarithm easily from left-to-right. If you are familiar with regular expressions or grammars, the following grammar might help you understand the cryptarithms that your program must parse:

```
cryptarithm ::= <expr> "=" <expr>
       expr ::= <word> [<operator> <word>]*
       word ::= <alphabetic-character>+
   operator ::= "+" | "-" | "*"
```

In plain English, this grammar says:

- A cryptarithm consists of two expressions separated by an equals sign (`=`).
- Each expression is a word optionally followed by one or more operator-word pairs.
- A word is a sequence of one or more alphabetic characters.
- An operator is either `+`, `-`, or `*`.

Your cryptarithm constructor must throw an appropriate exception if the given sequence of `String`s does not form a syntactically valid cryptarithm, or if the cryptarithm uses more than ten letters. (Recall that each letter in a cryptarithm must represent a distinct digit).

Part 3: Solving cryptarithms

Write a program that generates and prints all solutions to a cryptarithm. Your program must take a single cryptarithm on the command line, with tokens separated by one or more spaces. Your program might look like this when you run it:

```
$ java SolveCryptarithm SEND + MORE = MONEY
  1 solution(s):
    {S=9, E=5, N=6, D=7, M=1, O=0, R=8, Y=2}

$ java SolveCryptarithm WINTER + IS + WINDIER + SUMMER + IS = SUNNIER
  1 solution(s):
    {W=7, I=6, N=0, T=2, E=8, R=1, S=9, D=4, U=3, M=5}

$ java SolveCryptarithm NORTH * WEST = SOUTH * EAST
  1 solution(s):
    {N=5, O=1, R=3, T=0, H=4, W=8, E=7, S=6, U=9, A=2}

$ java SolveCryptarithm JEDER + LIEBT = BERLIN
  2 solution(s):
    {J=6, E=3, D=4, R=8, L=7, I=5, B=1, T=2, N=0}
    {J=4, E=3, D=6, R=8, L=9, I=5, B=1, T=2, N=0}

$ java SolveCryptarithm I + CANT + GET = NO + SATISFACTION
  0 solution(s)
```

You must use the permutation generator you wrote in Part 1 to generate all possible assignments of digits to letters, and use your `Expression` library from Homework 2 to check if each possible solution is valid. Remember that a possible solution is invalid if the first letter of any word in the cryptarithm represents zero. Your program should run in a reasonable amount of time: it should take less than ten seconds to solve any of the above cryptarithms on a typical laptop computer.

Testing your implementation

Test your solution using JUnit tests. Check the correctness of normal cases, edge cases, and error cases. We recommend that you start writing unit tests for your solution early; do not delay writing tests until after your implementation is complete. Remember that your permutation generator is an independent component that demands its own unit tests.

Evaluation

This homework is worth 100 points. We'll evaluate your solution approximately as follows:

- Mastery of earlier learning goals, especially the concepts of information hiding and polymorphism, software design based on informal specifications, and Java coding, specification, and testing practices and style: 60 points
- Effective use of inheritance, delegation, and design patterns to achieve design flexibility and code reuse: 30 points
- Discussion of relative advantages and disadvantages of design alternatives: 10 points

Hints and advice

- When parsing words, use `Character.isAlphabetic` to check if a character is legal.
- Here is a trick that may help you generate all subsets of size k of a set of size n. Let each of the low order n bits of an int represent the presence or absence of an element:

```
for (int bitVec = 0; bitVec < 1 << n; bitVec++) { // 1 << n is 2^n
    if (Integer.bitCount(bitVec) == k) {
         // The positions of all of the one bits in bitVec represent
         // one combination of k int values chosen from 0 to n-1.
    }
}
```

  This technique is not the most general or efficient, but it's good enough for this assignment. If you use it, be sure you understand how it works.
- There exist many good example cryptarithms on the web. We recommend Truman Collins's page and Torsten Sillke's page.
- You may (but aren't required to) improve the performance of your cryptarithm solver by extending the expression library from Homework 2 with an `Expression` implementation optimized for cryptarithm words. If you elect to optimize the performance of your solver, please use `System.nanotime` to measure its performance before and after each attempted optimization, and discuss any optimizations in your design rationale.
- If your cryptarithm solver is fast enough, you can use it to *generate* cryptarithms, by attempting to solve many potential cryptarithms and printing the ones that have a single solution. One possible source of potential cryptarithms is consecutive words in an input file. If you discover any interesting cryptarithms, please include them in your design document.