Homework #3: From Cryptarithms to Algorithms

Due Thursday, February 9th at 11:59 p.m.

In this assignment, you will build a powerful puzzle solver using the expression evaluator library you wrote for Homework 2. The puzzle of interest is cryptarithm. 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 four part assignment. In Part 1, you will design and implement three versions of a general purpose permutation generator using different design patterns. In Part 2 you will use your expression library from Homework 2 to write a class representing a cryptarithm and write a solve method that generates and checks all possible solutions using your permutation generator from Part 1. In Part 3 your will put your general purpose permutation generator to test and solve a similar, but different puzzle. Last but not least, in Part 4 you will reflect on your design and discuss the different patterns from an API design perspective.

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 programming against existing interfaces and implementations, and designing API for reuse.

Part 1: A reusable permutation generator

To complete this part, you will need to implement the following:

- a permutation generator using the Template Method Pattern.
- a permutation generator using the Strategy Pattern.
- a permutation generator using the Iterator Pattern.

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.
- An API how a client gets access to the generated permutations or provides the computations to be performed on each permutation.
- 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.

To make your implementation reusable, there are many designs and many design patterns that could help you in your design. In this assignment, we want you to incorporate three design patterns in your solution: **template method**, **iterator**, and **strategy pattern**.[1]

Implement three different versions of the permutation generator using each design pattern. *Implement each permutation generator in a separate package.* Make sure that it is clear how to use the generator in each implementation (i.e., make sure the API is well designed and well documented). A good API should be easy to use and make the resulting client code easy to read. In your implementations avoid copy-pasting shared code, but rather find a form of reuse among your three implementations.

Testing your implementation. Test your solution using unit tests. Check the correctness of normal cases, edge cases, and error cases. We do not not have specific requirements for line coverage; use your judgement and strive for a reasonable test suite to gain confidence in the correctness of the code. We recommend that you start writing unit tests for your solution early, possibly even before implementing the solution; do not delay writing tests until after your implementation is complete. Remember that your permutation generators are independent components that demand own unit tests.

Part 2: Solving cryptarithms

To complete this part, you will need to do the following:

- generate a `.jar` file for your homework 2 solution, and add it to your dependencies
- write a cryptarithm representation that parses strings into expressions and can check whether two cryptarithms are identical
- using your general purpose permutation generator, write a program that solves 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`.)

[1] Some would argue that this makes sense more as a 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. However, this should not affect your implementation much as both strategy and command pattern have the same basic structure.

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 two sides of the cryptarithm's equation, and checking for equality. Remember that a possible solution is invalid if the first letter of any word in the cryptarithm represents zero. 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). Addtionally, make the class comparable, such that two objects representing the same cryptarithm equal each other.

Additionally, allow to compare two cryptarithms for equality. Two cryptarithms should be equal if they represent the same equation (but they may differ in white-space of their string representation).

After completing a representation write a program that takes a cryptarithms and generates and prints all solutions to a cryptarithm, by trying every possible permutation with a permutation generator from Part 1 (you may chose any of your three implementations for this). Your program should construct the cryptarithm from command-line parameters. 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)
```

Your program should run in a reasonable amount of time: As a rule of thumb, it should take less than five seconds to solve any of the above cryptarithms on a typical laptop computer.

Part 3: Re-purposing your permutation generator

Next, we will show that the permutation generator is truly general, by reusing it for another problem: Anagrams. An anagram is direct word switch or word play, the result of rearranging the letters of a word or phrase to produce a new word or phrase, using all the original letters exactly once; for example, the word anagram can be rearranged into nag-a-ram (fun fact: try searching anagram on Google).

Write a program that when given a dictionary file and a list of words, prints out valid anagrams for each word. A permutation is a valid anagram of a word, if we can find it

in the dictionary. Your solution must make use of your permutation generator (the same variant you used in Part 2). We provide a dictionary file in your `src/main/resources/` directory that contains one word per line.

You can define how your program behaves when multiple solutions are present, so long as you document the behavior. To make things concrete, your program should do the following:

```
$ java SolveAnagram dict.txt rowd nagaram abc
  rowd: word
  nagaram: anagram
  abc: no solution
```

Part 4: Design Discussion

You have implemented three variants of the permutation generator. In a short text file, discuss the tradeoffs among the solutions regarding design goals, such as easy reuse, information hiding, and so forth. For example, most industrial programmers will likely avoid the template method pattern implementation, but why is that?

Hand in your answer in a file named `design.pdf`, `design.txt`, or `design.md` in the HW3 directory. Your answer should not exceed 500 words.

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: 35 points
- Discussion of tradeoffs: 5 points

Hints and advice

- Gradle can compile your homework 2 code into a `.jar` file for later reuse. Run the command `gradle jar` to generate a jar file to your output folder. We have configured gradle for this assignment already to automatically include `.jar` files in the `lib/` directory. Depending on your IDE, you may also need to configure the `.jar` file as an additional dependency in your IDE.
- Feel free to make changes to your code of Homework 2. However, if you change code in the Homework 2 directory, make sure that it is obvious from the commit message that it is not intended as a change that we should consider for grading of Homework 2, such that we do not take off late days for it.
- 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.
- We have enabled Findbugs for your in gradle, but configured it to not break your build if it finds a warning. Please look for warnings in the build process to consider whether it is an issue worth fixing.