Version1 — Template Method Pattern

```
public abstract class AbstractGenerator<E> {
  /**
    * Converts the generated permutations into different formats based on client requirements.
    */
   abstract public void convertFormat();
}
class TMPPermutationGenerator<E> extends AbstractGenerator {
   /**
    * Subclass constructor.
    *
    * @param set the input set of items
    */
   public TMPPermutationGenerator(List<E> set) {
      super(set);
   }

   @Override
   public void convertFormat() {

      // Call super.generatePermutations();
      // Convert the resulting permutations to the required format and store them in the field
'result'.
   }

   /**
    * Returns the resulting generated permutations in the required format of the client.
    *
    * @return the resulting permutations in the required format of the client
    */
   public Set<List<E>> returnResult() {
      return result;
   }

   /**
    * Sample client code: suppose the client wants all the permutations of a set of alphabets
{a,b,c}
    * and wants the resulting permutations to be stored in a set of lists.
    *
    * @param args
    */
   public static void main(String[] args) {
      List<Character> set = new ArrayList<>();
      set.add('a');
      set.add('b');
      set.add('c');
      TMPPermutationGenerator<Character> client = new TMPPermutationGenerator<>(set);
      client.convertFormat();
      Set<List<Character>> clientResult = client.returnResult();
```

```
        System.out.println(clientResult);
    }
}
```

Version2 — Iterator Pattern

```
public class Permutations<E> implements Iterable<List<E>> {
     /**
     * Constructor method: creates a 'permutation container'.
     *
     * @param set the given set of items to be permuted
     */
    public Permutations(List<E> set) {
        this.generatePermutations(set);
    }
     @Override
    public Iterator<List<E>> iterator() {
        return new PermutationIterator();
    }
    /**
     * Private nested class PermutationIterator: provides iterator objects for the Permutations.
     */
    private class PermutationIterator implements Iterator<List<E>> {
        @Override
        public boolean hasNext() {
            // check if this.result has next element to be iterated.
            return false;
        }
        @Override
        public List<E> next() {
            // return the next element in this.result to be iterated
            return null;
        }
    }
    /**
     * Sample client code: suppose the client wants all the permutations of a set of alphabets
{a,b,c}
     * and wants to access each permutation of the generated permutations separately
     *
     * @param args
     */
    public static void main(String[] args) {
        List<Character> set = new ArrayList<>();
        set.add('a');
        set.add('b');
        set.add('c');
        Permutations<Character> client = new Permutations<>(set);
        Iterator<List<Character>> itr = client.iterator();
        while (itr.hasNext()) {
            System.out.println(itr.next());
        }
    }
}
```

Version3 — Command Pattern

```
public interface CommandInterface<E> {
    /**
     * The 'generate()' command.
     *
     * @return a list of all the results that are generated
     */
    List<List<E>> generate();
}
public class PermutationCommand<E> implements CommandInterface<E> {
       /**
     * Constructor method.
     *
     * @param set the set of items to be permuted.
     */
    public PermutationCommand(List<E> set) {
        this.set = set;
    }

    /**
     * Generates all permutations of a given set.
     *
     * @return a list of all the permutations that are generated;
     * Return null if the given list is null or empty.
     */
    @Override
    public List<List<E>> generate() {
        if (set == null || set.size() == 0) return null;
        List<List<E>> result = new ArrayList<>();
        this.helper(this.set, set.size(), result);
        return result;
    }
    /**
     * Sample client code: suppose the client wants all the permutations of a set of alphabets
{a,b,c}
     * and wants to print out all the permutations
     *
     * @param args
     */
    public static void main(String[] args) {
        List<Character> set = new ArrayList<>();
        set.add('a');
        set.add('b');
        set.add('c');
        CommandInterface<Character> client = new PermutationCommand<>(set);
        List<List<Character>> clientResult = client.generate();
        System.out.println(clientResult);
    }
}
```

Design Rationale

For the representation of sets to be permuted and the representation of a permutation, I chose List<E> (Collections.List) over other data types such as Strings and arrays, because: (1)List can store generic types and objects; (2)Its size is mutable; (3) It's easy to use based on many useful APIs for different general purposes.

For the representation of all the resulting permutations generated, I chose List<List<E>>, based on not only the reasons listed above, but also the 'iterable' property of List.

I chose to implement in generic type(E) instead of primitives or specific objects(Integer, Character, etc.) to enhance reusability of this permutation generator.

I think the three design patterns all have their own advantages and weaknesses.
For example, the iterator pattern is unique in that it provides us a convenient API to access each permutation of the given set separately. And the template method pattern is useful for code reuse and extensibility. The command pattern uses a separate object each time; it is much more flexible, and we can enable more functionality/commands by swapping out the command object at runtime.

For this homework, I designed three versions of general-purpose permutation generator for further reuse and extensions. Also, I implemented the command pattern for the specific task of our cryptarithm, because: (1)I don't have very specific requirements for the representation of the resulting permutations and the representation of the sets to be permuted, since List interface is already very user friendly. (2)I choose interface over abstract class since interface is better for information hiding. (3)The command pattern implementation is relatively considered clean and simple, with only one exposed public method(API). (4) The sample client code is very simple and easy to read, which suggests good usability given that the client doesn't have very specific requirements for the format of the input sets and the output permutations. (5)If we want another functionality later, we just need to create another class implementing the interface and swap the command object to newly-created one.