Design Document

The idea behind using any of these three design principles was to not compute and store all possible permutations of each subset of numbers that was generated. This was taking too much time and memory and creating more problems. Thus, using the command, strategy or iterator pattern was required. My design goals focused efficiency, functionality and robustness.

In command pattern, requests can be handled at variant times or in variant orders. This is exactly what we want. By decoupling my code from the Permutation Generator, I would have gain re-usability and been able to implement my command interface to simply do *something* at the end of the function call to Permute a list. I chose not to use this because it would require less cohesion and class structuring that for me didn’t feel worthy. There were too many operations being made which was leading to more clutter.

I chose not use strategy pattern because I didn’t focus on scalability as much. I didn’t care to encapsulate Permutation generation in an object. This would create more encapsulation and hierarchy that I didn’t want to deal with when I simply just wanted a Permute method.

I decided design my permutation generator with an Iterator pattern, which I soon found out to be difficult. However, this made my client code in CryptarithmSolver much cleaner. I was able to think about only the .next() and .hasNext() which cleanly let me iterate through all the permutations because that implemented both the iterator and the iterable interfaces. Here is a snippet of my client code, and you can note how simple and easy to understand the code is because all that matters to the client is using the two methods.