**Name:**

Iterator

**Category:**

Behavioral Pattern

**Description:**

The Iterator Pattern is sort of a “middleman” client between an object accepting inputs and the inputs themselves. For example, if there was a large collection of objects in a collection, but we don’t want to show the implementation on how we use them or care about their types, we can simply iterate through them using this pattern. Another example is if we want to generate a **massive** list of numbers and calculate which numbers are prime and which aren’t. The iterator pattern allows us to see immediate results for the beginning numbers, rather than wait for the entire collection calculation to finish to see the results. Many programming languages have a native interpretation built in for this pattern. For C#, their built-in methods are the enumerator and yield.

**When to Use:**

Let’s say we wanted to iterate through the different results if we took a list and used the elements to transform a variable. We could iterate through the list and do a yield return to show what each element does to our variable, one at a time.

**Advantages:**

The iterator pattern has many advantages, such as the ability to allow for more abstract code. It is also flexible, in the sense that it can have different types of iterations as well. It is also very useful for lazy loading, which is useful in web applications.

**Disadvantages:**

If you wanted to know and show the underlying interpretation of your code, and have it more compact, the iterator pattern is the opposite of what you may need. This would also be suboptimal and unnecessary if we used simple collections. There is also added work and more code necessary to write for this pattern.

**General UML:**

A diagram of a client

Description automatically generated