Refactoring Exercise

You are provided with the following four classes:

IMatrix is an interface that represents a square matrix. The size of a matrix is represented by the property `size` — the number of rows (and hence also columns). We also have a read indexer for the matrix and a typed comparer.

DiagonalMatrix is a concrete class that represents a diagonal square matrix — a matrix where all elements but the main diagonal have the value 0. The representation used is just an array that represents the values of the main diagonal. Two constructors are provided: a default constructor which initializes the matrix to the zero matrix and a copy constructor.

FullMatrix is a concrete class that represents a general matrix. The representation is an array that holds all elements in row-major format. Here also a default and copy constructor are provided.

MatrixArray is a concrete class that represents an array of matrices. The representation is an array of `IMatrix` objects and the class implements the `IEnumerable<IMatrix>` interface. One constructor is provided which accepts the size as an argument. An `addAll` function is provided which implements a pointwise addition of matrices into a newly allocated `MatrixArray`.

Your task:
Refactor the `addAll` method of class `MatrixArray`. The refactored version of the method has to satisfy the refactoring criteria which are provided below. The refactored version must pass the unit tests provided.

The motivation for the refactoring is the following: The marketing department came to the conclusion that in the future, your system might have to deal with new kinds of matrices (e.g., sparse, symmetric or identity matrices). An adaptation of the refactored version of the method in response to an introduction of new kinds of matrices should be as simple as possible.

The refactored version of the method has to satisfy the following criteria:

- If A or B is a diagonal matrix of size `size`, the current implementation of the method uses only `size`, but not `size`^{2}, additions to compute $A+B$. Since efficiency is crucial for the implementation, it is expected that the refactored version of the method preserves this property.
- The refactored code cannot use dynamic type casts or type checks (`as` or `is` or `(Type)e`).
- The refactored code should not have duplication of code or functionality.
- The current implementation has some assertions — your implementation must preserve these assertions and add additional ones if relevant.
- The research department came to the conclusion that, for any pair of matrices A and B, $A + B$ is equal to $B + A$. You can use this discovery by replacing source code which performs an $A+B$ computation by source code which performs a $B+A$ computation.

```
public MatrixArray addAll(MatrixArray ma)
{
    Contract.Requires(size == ma.size);

    var result = new MatrixArray(size);

    for (var index = 0; index < size; index++)
    {
        var m1 = this[index];
        var m2 = ma[index];
        Contract.Assert(m1.size == m2.size);

        IMatrix mi;

        var m1f = m1 as FullMatrix;
        if (m1f != null)
        {
            var m2f = m2 as FullMatrix;
            if (m2f != null)
            {
                var t = new FullMatrix(m1f);
                for (var i = 0; i < m1f.size; i++)
                    for (var j = 0; j < m1f.size; j++)
                        t[i,j] =m2f[i, j];
                mi = t;
            }
            else
            {
                var m2d = m2 as DiagonalMatrix;
                Contract.Assert(m2d != null);
                var t = new FullMatrix(m1f);
                for (var i = 0; i < m1f.size; i++)
                    t[i, i] += m2d[i, i];
                mi = t;
            }
        } //m1 is a FullMatrix
        else
        {
            var m1d = m1 as DiagonalMatrix;
            Contract.Assert(m1d != null);
            var m2f = m2 as FullMatrix;
            if (m2f != null)
            {
                Contract.Assert(m1d.size == m2f.size);
                var t = new FullMatrix(m2f);
                for (var i = 0; i < m1d.size; i++)
                    t[i, i]+=m1d[i, i];
                mi = t;
            }
            else
            {
                var m2d = m2 as DiagonalMatrix;
                Contract.Assert(m2d != null);
                var t = new DiagonalMatrix(m1d);
                for (var i = 0; i < m1d.size; i++)
                    t[i, i] += m2d[i, i];
                mi = t;
            }
        }
        result[index] = mi;
    }

    return result;
}
```