Refactoring Exercise (Solution)

The approach for refactoring is to apply a variant of the visitor pattern in order to use dynamic dispatch instead of down-casts. The refactoring process can be described as the following sequence of steps:

- Essentially we refactor the loop body of the method `MatrixArray.addAll`. It can be represented schematically in the following way:

  ```
  if(M1 is FullMatrix)
    if(M2 is FullMatrix)
      AddFullFull(M1, M2);
    else
      AddFullDiagonal(M1, M2);
  else
    if(M2 is FullMatrix)
      AddDiagonalFull(M1, M2);
    else
      AddDiagonalDiagonal(M1, M2);
  ```

  Here M1 and M2 are elements of arrays of matrices which are currently being processed by the loop body. $AddXY$, where $X, Y \in \{Full, Diagonal\}$, are the different specializations of the addition operation.

- To get rid of the outer `if` statement, we add the method `add(IMatrix)` to the interface `IMatrix` and the classes `DiagonalMatrix` and `FullMatrix`:

```
interface IMatrix{
  IMatrix add(IMatrix m);
}

class FullMatrix{
  public IMatrix add(IMatrix m){
    if(m is FullMatrix)
      AddFullFull(this, m);
    else
      AddFullDiagonal(this, m);
  }
}

class DiagonalMatrix{
  public IMatrix add(IMatrix m){
    if(m is FullMatrix)
      AddDiagonalFull(this, m);
    else
      AddDiagonalDiagonal(this, m);
  }
}
```

  The first step, of extracting four static add methods, can be done using Resharper.

  After the addition of these methods we can rewrite the loop body as `M1.add(M2)`.

- Now we would like to get rid of the remaining `if` statement in both the `add` methods of `DiagonalMatrix` and `FullMatrix`. To do that we add `add(DiagonalMatrix)` and `add(FullMatrix)` (using overloading) to the interface `IMatrix` and the classes `DiagonalMatrix` and `FullMatrix`. We use these methods to determine the dynamic type of `m`. In order to achieve this we use the equivalence $A + B = B + A$ to have one implementation for `AddFullDiagonal` and `AddDiagonalFull`. The source code after these transformations is as follows:

```
interface IMatrix{
  IMatrix add(IMatrix m);
  IMatrix add(DiagonalMatrix m);
  IMatrix add(FullMatrix m);
}

class FullMatrix{
  public IMatrix add(IMatrix m){
    m.add(this);
  }

  public IMatrix add(DiagonalMatrix m){
    addFullDiagonal(this, m);
  }

  public IMatrix add(FullMatrix m){
    addFullFull(this, m);
  }
}

class DiagonalMatrix{
  public IMatrix add(IMatrix m){
    m.add(this);
  }

  public IMatrix add(DiagonalMatrix m){
    addDiagonalDiagonal(this, m);
  }

  public IMatrix add(FullMatrix m){
    addDiagonalFull(this, m);
  }
}
```

- We can see that `AddFullDiagonal` and `AddDiagonalFull` are semantically equivalent. We would like to eliminate this redundancy. To do it we change the body of method `DiagonalMatrix.add(FullMatrix)` of class `DiagonalMatrix` to use the implementation from `FullMatrix` with transposed arguments.

```
class DiagonalMatrix{
  public IMatrix add(FullMatrix m){
    m.add(this);
  }
}
```

The refactored version of the source code is as follows:

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

              var result = new MatrixArray(size);

              for (var index = 0; index < size; index++)
              {
                     Contract.Assert(this[index]!=null);
                     Contract.Assert(ma[index] != null);
                     Contract.Assert(this[index].size == ma[index].size);
                     result[index] = this[index].add(ma[index]);
              }

              return result;
       }
}
```

```
public class FullMatrix{
       public IMatrix add(IMatrix m)
       {
              return m.add(this);
       }

       public IMatrix add(FullMatrix m)
       {
              var result = new FullMatrix(this);
              for (var i = 0; i < size; i++)
                     for (var j = 0; j < size; j++)
                            result[i, j] += m[i, j];
              return result;
       }

       public IMatrix add(DiagonalMatrix m)
       {
              var result = new FullMatrix(this);
              for (var i = 0; i < size; i++)
                     result[i, i] += m[i, i];
              return result;
       }
}
```

```
public class DiagonalMatrix{
       public IMatrix add(IMatrix m)
       {
           return m.add(this);
       }

       public IMatrix add(FullMatrix m)
       {
           return m.add(this);
       }

       public IMatrix add(DiagonalMatrix m)
       {
           var result = new DiagonalMatrix(this);
           for (var i = 0; i < size; i++)
              result[i, i] += m[i, i];
           return result;
       }
}
```