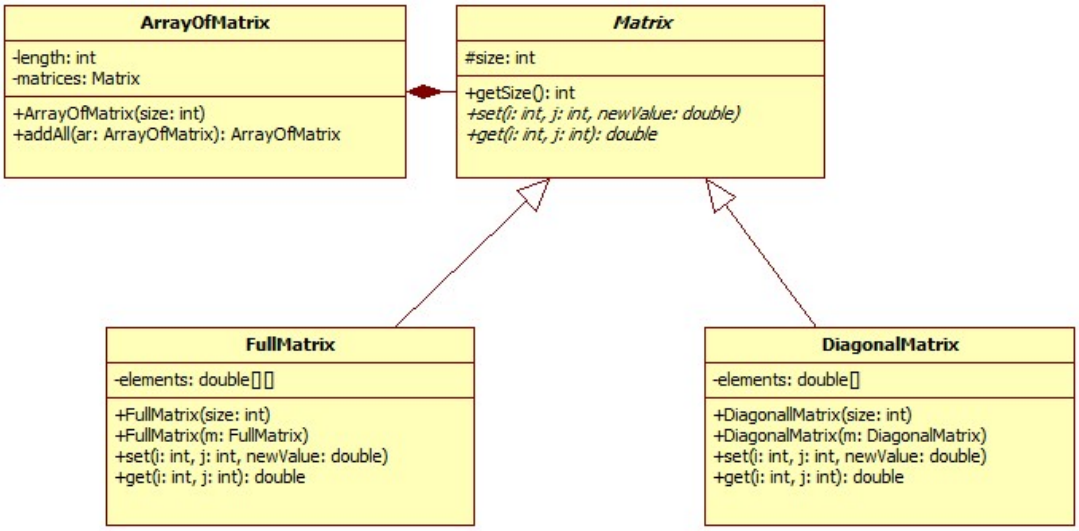
# Exercise 11
# – Refactoring –

**Your task:**

1. Write **JUnit** tests for the provided code.
2. Use **EasyPmd NetBeans** plugin to detect bugs and code style violations in the provided code.
   (a) For all detected bugs write a **JUnit** test which exposes the bug.
   (b) Fix all detected bugs and code style violations.
   (c) Check that the updated version of the source code successfully passes the new **JUnit** tests.
3. Use the **NetBeans** code coverage plugin to ensure that **JUnit** tests cover 100% of the provided code.
   (a) If coverage is not 100%, add **JUnit** test which cover missed parts of the code.
4. Refactor *addAll* method of class **ArrayOfMatrix**. The refactored version of the method *addAll* has to satisfy the refactoring criteria which are provided below.
5. Run **JUnit** test on the refactored code to ensure that its behavior is preserved.
   (a) If a JUnit test fails, find the reason of the failure and fix the corresponding bug.
6. Use the **NetBeans** code coverage plugin to identify which parts of the refactored code are not covered by the **JUnit** tests.
7. Write **JUnit** tests which covers the missed part of the refactored code.
8. Use the **EasyPmd NetBeans** plugin to detect bugs and code style violations in the refactored code.

You are provided with the following four classes:




**Matrix** is an abstract class that represents a square matrix. The size of a matrix is represented by a protected field *size*. Here by size of a matrix we mean the number of rows and columns in the matrix. The value of the field *size* can be extracted via the getter *getSize*. **Matrix** has two abstract methods; the getter of matrix elements *get* and the setter of matrix elements *set*.

**DiagonalMatrix** is a concrete class that represents a diagonal square matrix. A diagonal matrix is a matrix with all elements outside of the diagonal (elements with index $\langle i, j \rangle$ were $i \neq j$) equal to zero. The class has a private field *elements*, which contains elements of the matrix diagonal. **DiagonalMatrix** has two constructors. The first one initializes a new matrix of a given size. The second one is a copy constructor, which initializes a new matrix with size and elements equal to an input parameter matrix. The field *elements* has length equal to the size of the matrix. Also **DiagonalMatrix** provides implementations for *get* and *set*. Setter *set* throws an exception if $i \neq j$. Otherwise it saves *newValue* in $elements[i]$. Getter *get* returns value of $elements[i]$ if $i = j$, and zero otherwise.

**FullMatrix** is a concrete class that represents a full square matrix. A full matrix is a matrix with possibly all elements different from zero. In other words, it is an ordinary matrix without additional constraints. The class has a private field *elements*, which contains elements of the matrix. Similarly to **DiagonalMatrix**, it has two constructors; one creates a new matrix of a given size and another one is a copy constructor. The field *elements* has the first and second dimension lengths equal to the size of the matrix. Also **FullMatrix** implements *get* which reads values of *elements* and *set* which writes values of *elements*.

**ArrayOfMatrix** is a concrete class that represents arrays of matrices. It has two private fields; *length* represents the length of the array and *matrices* represents elements

of the array. **ArrayOfMatrix** has one constructor which creates a new array of matrices of a given length. Elements of the freshly-allocated array are equal to *null*.

**ArrayOfMatrix** has a method *addAll*. The method gets as input parameter an array of matrices and returns a freshly-allocated array of matrices which contains the elementwise sum of the receiver and the input array of matrix, the source code of method *addAll* is presented in figure 1.

**The motivation for the refactoring is the following:**

- The marketing department came to the conclusion that in the future, your system has to deal with new kinds of matrices (e.g., symmetric or identity matrices). An adaptation of the refactored version of the method in response to an introduction of new kinds of matrices must 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 can not use type casts.
- The refactored code can not have duplication of code or functionality
- The current implementation uses assertions to check preconditions (properties of input parameters). The refactored version have to check exactly the same properties.

**During the refactoring process you can use, among other, the following transformation:**

- The research department came to the conclusion that, for any matrices $A$ and $B$, $A + B$ is equal to $B + A$. You can use this discovery by replacing a source code which performs an $A + B$ computation by a source code which performs an $B + A$ computation.

```
public ArrayOfMatrix addAll(ArrayOfMatrix ar){
  // Check that arrays have the same length
  assert(length == ar.length);
  ArrayOfMatrix result = new ArrayOfMatrix(size);
  // Iterates over elements of the arrays
  for(int ind=0; ind<length; ind++){
    Matrix resultTemp;
    if (matrices[ind] instanceof FullMatrix){
      FullMatrix m1 = (FullMatrix) matrices[ind];
      if (ar.matrices[ind] instanceof FullMatrix){
        // Add two full matrices
        FullMatrix m2 = (FullMatrix) ar.matrices[ind];
        assert(m1.getSize() == m2.getSize());
        FullMatrix temp = new FullMatrix(m1);
        for(int i=0; i<m1.getSize();i++)
          for(int j=0; j<m1.getSize();j++)
            temp.set(i, j, temp.get(i, j) + m2.get(i, j));
        resultTemp = temp;
      } else {
        // Add a full and a diagonal matrices
        DiagonalMatrix m2 = (DiagonalMatrix) ar.matrices[ind];
        assert(m1.getSize() == m2.getSize());
        FullMatrix temp = new FullMatrix(m1);
        for(int i=0; i<m1.getSize();i++)
          temp.set(i, i, temp.get(i, i) + m2.get(i, i));
        resultTemp = temp;
      }
    } else {
      DiagonalMatrix m1 = (DiagonalMatrix) matrices[ind];
      if (ar.matrices[ind] instanceof FullMatrix){
        // Add a diagonal and a full matrices
        FullMatrix m2 = (FullMatrix) ar.matrices[ind];
        assert(m1.getSize() == m2.getSize());
        FullMatrix temp = new FullMatrix(m2);
        for(int i=0; i<m1.getSize();i++)
          temp.set(i, i, temp.get(i, i) + m1.get(i, i));
        resultTemp = temp;
      } else {
        // Add two diagonal matrices
        DiagonalMatrix m2 = (DiagonalMatrix) ar.matrices[ind];
        assert(m1.getSize() == m2.getSize());
        DiagonalMatrix temp = new DiagonalMatrix(m1);
        for(int i=0; i<m1.getSize();i++)
          temp.set(i, i, temp.get(i, i) + m2.get(i, i));
        resultTemp = temp;
      }
    }
    // Save the result of the addition into the output array
    result.matrices[ind] = resultTemp;
  }
  return result;
}
```

Figure 1: The source code of method *addAll* of class **ArrayOfMatrix**.