```cpp
#include "Add.h"

int Add::calculate(unsigned a, unsigned b) const {
    return a + b;
}

```

```cpp
//
// Created by André on 10.03.2022.
//

#ifndef POA_L1_ADD_H
#define POA_L1_ADD_H


#include "Operation.h"

/**
 * Add operation.
 *
 * @author Nelson Jeanrenaud
 * @author André Marques Nora
 * @version 1.0
 */
class Add : public Operation{
public:


   /**
    * Add two numbers together
    *
    * @param a The first number to add.
    * @param b The second operand.
    * @return The result of the addition.
    */
   int calculate(unsigned a, unsigned b) const override;
};


#endif //POA_L1_ADD_H

```

File - C:\Heig-vd\POA\POA_L1\src\main.cpp

```cpp
#include <iostream>
#include <sstream>
#include "Matrix.h"
using std::cout;
using std::endl;

/**
 * Test the matrix class
 *
 * @param n1 number of rows of the first matrix
 * @param m1 the number of columns of the first matrix
 * @param n2 number of rows of the second matrix
 * @param m2 the number of columns of the second matrix
 * @param n the modulo of the matrix
 */
void testMatrix(unsigned n1, unsigned m1, unsigned n2, unsigned m2, unsigned n){
    cout << "Test initialisation" << endl;

    Matrix* matrix1 = new Matrix(n1, m1, n);

    Matrix* matrix2 = new Matrix(n2, m2, n);

    Matrix* matrix3 = new Matrix(n1+1, m1+1, n);

    Matrix* matrix4 = new Matrix(n1, m1, n+1);

    cout << "matrix1 : " << endl << matrix1 << endl;
    cout << "matrix2 : " << endl << matrix2 << endl;
    cout << "matrix3 : " << endl << matrix3 << endl;
    cout << "matrix4 : " << endl << matrix4 << endl;

    cout << "test constructor copy : " << endl;
    Matrix c(*matrix1);
    cout << "c : " << endl << c << endl;

    cout << "test operator = : " << endl;
    Matrix d = *matrix2;
    cout << "d : " << endl << d << endl;

    cout << "test operator : " << endl;

    cout << "ADD with same modulus and size : " << endl;
    cout << "Inline : " << endl;

    matrix1->add(*matrix2);
    cout << "matrix1 : " << endl << matrix1 << endl;
    matrix1 = &c;
    cout << "by value : " << endl;
    d = matrix1->addByValue(*matrix2);
    cout << "d : " << endl << d << endl;
    cout << "by pointer : " << endl;
    c = matrix1->addByPtr(*matrix2);
```

```
 53     cout << "d : " << endl << d << endl;
 54
 55     cout << "ADD with same modulus and different size : " << endl;
 56     cout << "Inline : " << endl;
 57     matrix1->add(*matrix3);
 58     cout << "matrix1 : " << endl << matrix1 << endl;
 59     matrix1 = &c;
 60     cout << "by value : " << endl;
 61     d = matrix1->addByValue(*matrix3);
 62     cout << "d : " << endl << d << endl;
 63     cout << "by pointer : " << endl;
 64     d = matrix1->addByPtr(*matrix3);
 65     cout << "d : " << endl << d << endl;
 66
 67     /*cout << "ADD with different modulus : " << endl;
 68     cout << "Inline : " << endl;
 69     matrix1->add(*matrix4);
 70     cout << "matrix1 : " << endl << matrix1 << endl;
 71     matrix1 = &c;
 72     cout << "by value : " << endl;
 73     d = matrix1->addByValue(*matrix4);
 74     cout << "d : " << endl << d << endl;
 75     cout << "by pointer : " << endl;
 76     d = matrix1->addByPtr(*matrix4);
 77     cout << "d : " << endl << d << endl;*/
 78
 79     cout << "SUB with same modulus and size : " << endl;
 80     cout << "Inline : " << endl;
 81     matrix1->sub(*matrix2);
 82     cout << "matrix1 : " << endl << matrix1 << endl;
 83     matrix1 = &c;
 84     cout << "by value : " << endl;
 85     d = matrix1->subByValue(*matrix2);
 86     cout << "d : " << endl << d << endl;
 87     cout << "by pointer : " << endl;
 88     c = matrix1->subByPtr(*matrix2);
 89     cout << "d : " << endl << d << endl;
 90
 91     cout << "SUB with same modulus and different size : " << endl;
 92     cout << "Inline : " << endl;
 93     matrix1->sub(*matrix3);
 94     cout << "matrix1 : " << endl << matrix1 << endl;
 95     matrix1 = &c;
 96     cout << "by value : " << endl;
 97     d = matrix1->subByValue(*matrix3);
 98     cout << "d : " << endl << d << endl;
 99     cout << "by pointer : " << endl;
100     d = matrix1->subByPtr(*matrix3);
101     cout << "d : " << endl << d << endl;
102
103     /*cout << "SUB with different modulus : " << endl;
104     cout << "Inline : " << endl;
105     matrix1->sub(*matrix4);
```

```
106     cout << "matrix1 : " << endl << matrix1 << endl;
107     matrix1 = &c;
108     cout << "by value : " << endl;
109     d = matrix1->subByValue(*matrix4);
110     cout << "d : " << endl << d << endl;
111     cout << "by pointer : " << endl;
112     d = matrix1->subByPtr(*matrix4);
113     cout << "d : " << endl << d << endl;*/
114
115     cout << "MULTIPLY with same modulus and size : " << endl;
116     cout << "Inline : " << endl;
117     matrix1->mult(*matrix2);
118     cout << "matrix1 : " << endl << matrix1 << endl;
119     matrix1 = &c;
120     cout << "by value : " << endl;
121     d = matrix1->multByValue(*matrix2);
122     cout << "d : " << endl << d << endl;
123     cout << "by pointer : " << endl;
124     c = matrix1->multByPtr(*matrix2);
125     cout << "d : " << endl << d << endl;
126
127     cout << "MULTIPLY with same modulus and different size : " <<
    endl;
128     cout << "Inline : " << endl;
129     matrix1->mult(*matrix3);
130     cout << "matrix1 : " << endl << matrix1 << endl;
131     matrix1 = &c;
132     cout << "by value : " << endl;
133     d = matrix1->multByValue(*matrix3);
134     cout << "d : " << endl << d << endl;
135     cout << "by pointer : " << endl;
136     d = matrix1->multByPtr(*matrix3);
137     cout << "d : " << endl << d << endl;
138
139     /*cout << "MULTIPLY with different modulus : " << endl;
140     cout << "Inline : " << endl;
141     matrix1->mult(*matrix4);
142     cout << "matrix1 : " << endl << matrix1 << endl;
143     matrix1 = &c;
144     cout << "by value : " << endl;
145     d = matrix1->multByValue(*matrix4);
146     cout << "d : " << endl << d << endl;
147     cout << "by pointer : " << endl;
148     c = matrix1->multByPtr(*matrix4);
149     cout << "d : " << endl << d << endl;*/
150
151     delete matrix1;
152     delete matrix2;
153     delete matrix3;
154     delete matrix4;
155 }
156
157 int main(int argc, char* argv[]) {
```

```
158
159     if(argc != 6){
160         std::cerr << "Usage: " << argv[0] << " N1 M1 N2 M2 N" << std
    ::endl;
161         return 1;
162     }
163
164     std::istringstream issn1(argv[1]), issm1(argv[2]), issn2(argv[3
    ]), issm2(argv[4]), issn(argv[5]);
165     unsigned n1, m1, n2, m2, n;
166
167     if(issn1 >> n1 and issm1 >> m1 and issn2 >> n2 and issm2 >> m2
    and issn >> n) {
168         testMatrix(n1, m1, n2, m2, n);
169         return 0;
170     }
171     std::cerr << "Error: " << argv[0] << " Arguments are not integers
    " << std::endl;
172     return 1;
173 }
```

```cpp
#include "Matrix.h"
#include "iostream"
#include "Add.h"
#include "Sub.h"
#include "Multiply.h"
#include "Random.h"
#include <cmath>


static const Add ADD = Add();
static const Sub SUB = Sub();
static const Multiply MUL = Multiply();

Matrix::Matrix() {
    this->n = 0;
    this->m = 0;
    this->modulo = 0;
    this->data = nullptr;
}


Matrix::Matrix(unsigned int n, unsigned int m, unsigned int modulo, bool initRandom) {
    if(n < MIN_N)
        throw std::runtime_error("number of rows is out of bounds");
    if(m < MIN_M)
        throw std::runtime_error("number of columns is out of bounds");
    if(modulo < MIN_MODULO)
        throw std::runtime_error("modulus is out of bounds");

    Random* rand = Random::getInstance();
    this->n = n;
    this->m = m;
    this->modulo = modulo;
    this->data = new unsigned* [m];

    for (int i = 0; i < m; ++i) {
        this->data[i] = new unsigned [n];
    }
    if(initRandom) {
        for (int i = 0; i < this->n; ++i) {
            for (int j = 0; j < this->m; ++j) {
                this->data[i][j] = mod(rand->getRandom(this->modulo), this->modulo);
            }
        }
    }
}

Matrix::Matrix(unsigned int n, unsigned int m, unsigned int modulo) : Matrix(n, m, modulo, true) {}

```

```
 50 Matrix::Matrix(const Matrix &other) : Matrix(other.n, other.m, other.
    modulo, false){
 51     std::cout << "copy Matrix" << std::endl;
 52     for (unsigned i = 0; i < this->m; ++i) {
 53         for (unsigned j = 0; j < this->n; ++j) {
 54             this->data[i][j] = other.data[i][j];
 55         }
 56     }
 57 }
 58 
 59 /**
 60  * Deletes the dynamically allocated memory for the data member
 61  */
 62 Matrix::~Matrix() {
 63     deleteValues();
 64 }
 65 
 66 void Matrix::deleteValues(){
 67     for (int i = 0; i < m; ++i) {
 68         delete[] this->data[i];
 69     }
 70     delete[] this->data;
 71 }
 72 
 73 std::ostream &operator<<(std::ostream &os, const Matrix &dt) {
 74     for (int i = 0; i < dt.m; ++i) {
 75         for (int j = 0; j < dt.n; ++j) {
 76             os << dt.data[i][j];
 77             if(j + 1 < dt.n)
 78                 os << " ";
 79         }
 80         os << std::endl;
 81     }
 82     return os;
 83 }
 84 
 85 std::ostream &operator<<(std::ostream &os, Matrix* dt) {
 86     os << *dt;
 87     return os;
 88 }
 89 
 90 Matrix &Matrix::operator=(const Matrix &other){
 91     return operator=(&other);
 92 }
 93 
 94 Matrix & Matrix::operator=(const Matrix *other) {
 95 
 96     if(other != this){
 97         // We use a temporary variable to not leave the object in a
    broken state
 98         // in case the allocation throws an exception.
 99         unsigned** tmpData = new unsigned* [other->n];
100         for (int i = 0; i < other->n; ++i) {
```

```
101             tmpData[i] = new unsigned [other->m];
102         }
103 
104         deleteValues();
105 
106         this->n = other->n;
107         this->m = other->m;
108         this->modulo = other->modulo;
109 
110         this->data = tmpData;
111         for (int i = 0; i < this->n; ++i) {
112             for (int j = 0; j < this->m; ++j) {
113                 this->data[i][j] = other->data[i][j];
114             }
115         }
116     }
117 
118     return *this;
119 }
120 
121 unsigned int Matrix::getValueOrZero(unsigned i, unsigned j) const {
122     return i < this->n && j < this->m ? this->data[i][j] : 0;
123 }
124 
125 Matrix& Matrix::operation(const Matrix &other, const Operation &op) {
126     if(other.modulo != this->modulo)
127         throw std::invalid_argument("Error : Not the same modulus");
128 
129     unsigned newN = std::max(this->n, other.n);
130     unsigned newM = std::max(this->m, other.m);
131 
132     unsigned** tmpData = new unsigned* [newN];
133     for (int i = 0; i < newN; ++i) {
134         tmpData[i] = new unsigned [newM];
135     }
136 
137     for (unsigned i = 0; i < newM; ++i) {
138         for (unsigned j = 0; j < newN; ++j) {
139             tmpData[i][j] = mod(op.calculate(this->getValueOrZero(i,j
    ), other.getValueOrZero(i,j)),modulo);
140         }
141     }
142 
143     this->n = newN;
144     this->m = newM;
145 
146     deleteValues();
147     this->data = tmpData;
148     return *this;
149 }
150 
151 Matrix *Matrix::operationByPtr(const Matrix &other, const Operation &
    op) const {
```

```
152     if(other.modulo != this->modulo)
153         throw std::invalid_argument("Error : Not the same modulus");
154 
155     Matrix* res = new Matrix(std::max(this->n, other.n), std::max(
    this->m, other.m), this->modulo, false);
156 
157     for (unsigned i = 0; i < n; ++i) {
158         for (unsigned j = 0; j < m; ++j) {
159             res->setValue(i, j, op.calculate(this->getValueOrZero(i,j
    ), other.getValueOrZero(i,j)));
160         }
161     }
162 
163     return res;
164 }
165 
166 Matrix Matrix::operationByValue(const Matrix &other, const Operation
     &op) const {
167     if(other.modulo != this->modulo)
168         throw std::invalid_argument("Error : Not the same modulus");
169 
170     Matrix res = Matrix(std::max(this->n, other.n), std::max(this->m
    , other.m), this->modulo, false);
171 
172     for (unsigned i = 0; i < n; ++i) {
173         for (unsigned j = 0; j < m; ++j) {
174             res.setValue(i, j, op.calculate(this->getValueOrZero(i,j
    ), other.getValueOrZero(i,j)));
175         }
176     }
177 
178     return res;
179 }
180 
181 void Matrix::setValue(unsigned int i, unsigned int j, unsigned int
    value) {
182     if(i >= n || j >= m)
183         throw std::runtime_error("Error out of bounds");
184 
185     data[i][j] = mod(value, modulo);
186 }
187 
188 Matrix& Matrix::add(const Matrix &other) {
189     return this->operation(other, ADD);
190 }
191 
192 Matrix Matrix::addByValue(const Matrix &other) const {
193     return this->operationByValue(other, ADD);
194 }
195 
196 Matrix *Matrix::addByPtr(const Matrix &other) const {
197     return this->operationByPtr(other, ADD);
198 }
```

```
199
200 Matrix& Matrix::sub(const Matrix &other) {
201     return this->operation(other, SUB);
202 }
203
204 Matrix Matrix::subByValue(const Matrix &other) const {
205     return this->operationByValue(other, SUB);
206 }
207
208 Matrix *Matrix::subByPtr(const Matrix &other) const {
209     return this->operationByPtr(other, SUB);
210 }
211
212 Matrix& Matrix::mult(const Matrix &other) {
213     return this->operation(other, SUB);
214 }
215
216 Matrix Matrix::multByValue(const Matrix &other) const {
217     return this->operationByValue(other, MUL);
218 }
219
220 Matrix *Matrix::multByPtr(const Matrix &other) const {
221     return this->operationByPtr(other, MUL);
222 }
223
224 unsigned int Matrix::mod(int a, int b){
225    return (unsigned int) std::abs(a - ((floor((a / b))) * b));
226 }
227
```

```
 1 #ifndef POA_L1_MATRIX_H
 2 #define POA_L1_MATRIX_H
 3
 4 #include "iostream"
 5 #include "Operation.h"
 6
 7 /**
 8 * Matrix of unsigned int.
 9 *
10 * @author Nelson Jeanrenaud
11 * @author André Marques Nora
12 * @version 1.0
13 */
14 class Matrix {
15
16    /**
17     * Minimal value for n
18     */
19    static const int MIN_N = 1;
20
21    /**
22     * Minimal value for m
23     */
24    static const int MIN_M = 1;
25
26    /**
27     * Minimal value for modulo
28     */
29    static const int MIN_MODULO = 1;
30
31    /**
32    * the number of rows in the matrix
33    */
34    unsigned int n;
35
36    /**
37    * the number of columns in the matrix
38    */
39    unsigned int m;
40
41    /**
42    * Modulo for the numbers contained in the matrix
43    */
44    unsigned int modulo;
45
46    /**
47    * Array containing the numbers of the matrix
48    */
49    unsigned int** data;
50
51    Matrix();
52
53    /**
```

```
 54    * Create a matrix with n rows and m columns
 55    *
 56    * @param n the number of rows in the matrix
 57    * @param m the number of rows in the matrix
 58    * @param modulo the modulo for the numbers contained in the matrix
 59    * @param initRandom if true, the matrix will be initialized with
    random values.
 60    */
 61    Matrix(unsigned int n, unsigned int m, unsigned int modulo, bool
    initRandom);
 62
 63    /**
 64    * If the indices are valid, return the value at the specified
    indices. Otherwise, return 0
 65    *
 66    * @param i The row of the matrix.
 67    * @param j The column index.
 68    * @return The value of the element at position (i, j) or zero if
    the position is out of bounds.
 69    */
 70    unsigned int getValueOrZero(unsigned i, unsigned j) const;
 71
 72    /**
 73    * Destroy the value array in memory
 74    */
 75    void deleteValues();
 76
 77    /**
 78    * Methode to calculate the modulus of two given numbers
 79    * @param a The first operand
 80    * @param b The second operand
 81    * @return Unsigned int representing the result of the modulus'
    calcul
 82    */
 83    static unsigned int mod(int a, int b);
 84
 85 public:
 86
 87    /**
 88    * Operator for the stream output
 89    * @param os The output stream
 90    * @param dt A matrix
 91    * @return A stream output corresponding to a matrix
 92    */
 93    friend std::ostream& operator<<(std::ostream& os, const Matrix& dt
    );
 94
 95    /**
 96    * Operator for the stream output
 97    * @param os The output stream
 98    * @param dt A pointer to a matrix
 99    * @return A stream output corresponding to a matrix
100    */
```

```
101    friend std::ostream& operator<<(std::ostream& os, Matrix* dt);
102
103    /**
104    * Operator = for a matrix
105    * @param other A matrix
106    * @return A Matrix
107    */
108    Matrix& operator= (const Matrix& other);
109
110    /**
111    * Operator = for a matrix
112    * @param other A pointer to a matrix
113    * @return A Matrix
114    */
115    Matrix& operator= (const Matrix *other);
116
117    /**
118     * Create a matrix with n rows and m columns
119     *
120     * @param n the number of rows in the matrix
121     * @param m the number of rows in the matrix
122     * @param modulo the modulo for the numbers contained in the
    matrix
123     */
124    Matrix(unsigned n, unsigned m, unsigned modulo);
125
126    /**
127    * The function creates a new matrix with the same dimensions as
    the other matrix and copies the values
128    * from the other matrix into the new matrix
129    *
130    * @param other the matrix to copy
131    */
132    Matrix(const Matrix& other);
133
134    /**
135    * Matrix destructor
136    */
137    virtual ~Matrix();
138
139    /**
140    * This function adds the values of the two matrices and returns
    the result
141    *
142    * @param other The matrix to add to this one.
143    * @return a reference to this object
144    */
145    Matrix& add(const Matrix& other);
146
147    /**
148    * Add two matrices together by value
149    *
150    * @param other The matrix to add to the current matrix.
```

```
151     * @return The result of the addition operation.
152     */
153
154     Matrix addByValue(const Matrix& other) const;
155     /**
156     * This function adds two matrices together and returns a pointer
    to the resulting matrix
157     * The resulting matrix is created dynamically
158     *
159     * @param other the other matrix to be added to this matrix.
160     * @return A new Matrix object.
161     */
162     Matrix* addByPtr(const Matrix& other) const;
163
164     /**
165     * Subtracts the other matrix from this matrix
166     *
167     * @param other The matrix to subtract from this matrix.
168     * @return a reference to this object.
169     */
170     Matrix& sub(const Matrix& other);
171
172     /**
173     * Subtracts the values of the other matrix from the values of this
     matrix
174     *
175     * @param other The matrix to subtract from this matrix.
176     * @return The result of the subtraction operation.
177     */
178     Matrix subByValue(const Matrix& other) const;
179
180     /**
181     * This function subtracts the values of the other matrix from the
    values of this matrix and returns a
182     * new dynamically allocated matrix
183     *
184     * @param other the matrix to subtract from this matrix
185     * @return A new Matrix object.
186     */
187     Matrix* subByPtr(const Matrix& other) const;
188
189     /**
190     * Multiply the matrix by another matrix
191     *
192     * @param other The matrix to subtract from this matrix.
193     * @return a reference to this object.
194     */
195     Matrix& mult(const Matrix& other);
196
197     /**
198      * Multiply the matrix by a another matrix
199      *
200      * @param other The matrix to multiply by.
```

```
201     * @return The result of the multiplication.
202     */
203    Matrix multByValue(const Matrix& other) const;
204
205    /**
206     * Multiply the matrix by another matrix
207     *
208     * @param other the matrix to multiply by
209     * @return A pointer to a new Matrix object.
210     */
211    Matrix* multByPtr(const Matrix& other) const;
212
213    /**
214    * Given a matrices, this function returns the result of the
    operation between the matrix and this
215    *
216    * @param other the matrix to be added
217    * @param op the operation to perform
218    * @return A reference to this object
219    */
220    Matrix& operation(const Matrix& other, const Operation& op);
221
222    /**
223    * Given a matrices, this function returns the result of the
    operation between the matrix and this
224    *
225    * @param other the matrix to be added
226    * @param op the operation to perform
227    * @return A pointer to a new Matrix object.
228    */
229    Matrix* operationByPtr(const Matrix& other, const Operation& op)
    const;
230
231    /**
232    * Given a matrices, this function returns the result of the
    operation between the matrix and this
233    *
234    * @param other the matrix to be added
235    * @param op the operation to perform
236    * @return A new Matrix object.
237    */
238    Matrix operationByValue(const Matrix& other, const Operation& op)
    const;
239
240    /**
241    * Set the value of the element at row i and column j to value
242    *
243    * @param i The row of the matrix.
244    * @param j The column index of the value to be set.
245    * @param value the value to be set
246    */
247    void setValue(unsigned i, unsigned j, unsigned value);
248 };
```

```
249
250
251 #endif //POA_L1_MATRIX_H
252
```

File - C:\Heig-vd\POA\POA_L1\src\Multiply.cpp

```cpp
#include "Multiply.h"

int Multiply::calculate(unsigned a, unsigned b) const {
    return a * b;
}

```

```cpp
#ifndef POA_L1_MULTIPLY_H
#define POA_L1_MULTIPLY_H

#include "Operation.h"

/**
 * Multiplication operation.
 *
 * @author Nelson Jeanrenaud
 * @author André Marques Nora
 * @version 1.0
 */
class Multiply : public Operation{
public:

   /**
    * Multiply the two unsigned integers
    *
    * @param a The first number to multiply.
    * @param b The number of times to multiply a by b.
    * @return The result of the multiplication.
    */
   int calculate(unsigned a, unsigned b) const override;
};


#endif //POA_L1_MULTIPLY_H

```

```cpp
#ifndef POA_L1_OPERATION_H
#define POA_L1_OPERATION_H

/**
 * Operation abstract class.
 * Extended to calculate an operation between two unsigned int.
 *
 * @author Nelson Jeanrenaud
 * @author André Marques Nora
 * @version 1.0
 */
class Operation {
public:

    /**
     * Calculate the operation between two unsigned int.
     * @param a first operand.
     * @param b second operand..
     * @return result of the operation
     */
   virtual int calculate(unsigned a, unsigned b) const = 0;
};


#endif //POA_L1_OPERATION_H

```

```cpp
#include "Random.h"
#include <ctime>

Random* Random::instance = nullptr;
/**
 * The constructor initializes the random number generator with the current time
 */
Random::Random() {
    srand(time(nullptr));
}

Random* Random::getInstance() {
    if(!instance)
        instance = new Random;
    return instance;
}

unsigned Random::getRandom(unsigned n) {
    return (unsigned) (1 + rand() / (RAND_MAX + 1.0) * n);
}

```

```cpp
#ifndef POA_L1_RANDOM_H
#define POA_L1_RANDOM_H

#include <cstdlib>

/**
 * Singleton Random unsigned int generator.
 *
 * @author Nelson Jeanrenaud
 * @author André Marques Nora
 * @version 1.0
 */
class Random {

    /**
     * Single instance of the random class
     */
    static Random* instance;
    /**
     * Default constructor
     */
    Random();

public:

    /**
     * If there is no instance of Random, create one
     *
     * @return The Random class is a singleton.  The getInstance() method returns the single instance of
     * the class.
     */
    static Random* getInstance();

    /**
      * Generate a random number between 1 and n
      *
      * @param n The upper bound of the number generated.
      * @return A random number between 1 and n.
      */
    unsigned int getRandom(unsigned n);

    /**
     * Singleton class can't be copied
     */
    Random(Random &other) = delete;
    /**
     * Singleton class can't be assigned
     */
    void operator=(const Random &) = delete;
};


```

```
53 #endif //POA_L1_RANDOM_H
54
```

File - C:\Heig-vd\POA\POA_L1\src\Sub.cpp

```cpp
#include "Sub.h"

int Sub::calculate(unsigned a, unsigned b) const {
    return a - b;
}

```

```h
//
// Created by André on 10.03.2022.
//

#ifndef POA_L1_SUB_H
#define POA_L1_SUB_H


#include "Operation.h"

/**
 * Substraction operation.
 *
 * @author Nelson Jeanrenaud
 * @author André Marques Nora
 * @version 1.0
 */
class Sub : public Operation{
public:

   /**
    * "Subtract b from a and return the result."
    *
    * @param a The first number to be subtracted.
    * @param b The number of times to run the test.
    * @return The result of the substraction
    */
   int calculate(unsigned a, unsigned b) const override;
};


#endif //POA_L1_SUB_H

```