1. Introduction

This document describes the functionalities of CalculationEngine, the design description, how to run and how to execute unit test. In addition, API specification is provided by doxygen html document.

CalcuationEngine supports two engines.

- Multiplier
- Divider

Multiplier is a calculator of the multiplication. It takes only integer as an input data. You can check a calculated result from stdout.
Divider calculates the division. As an input data, integers and files are allowed. The calculated results are displayed in stdout.

1.1 Architectural Overview

First of all, CalculationEngine is an abstraction class suppling a factory method to generate the specific engines.
CalculationEngins defines three virtual methods which should be implemented in the sub class.

- read_input_data
- calculate
- display_result

Multiplier and Divider inherits CalcuationEngine so that each class provides three methods.

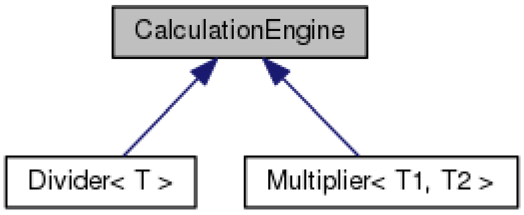


Fig 1. Class Diagram

Basically, CalculatoinEngine works from the sequential execution of methods including read_input_data(), calculate() and display_result() in order. However, even though you don't comply with this order, no problem happens because every method handles exception case.

1.2 How to run

The program, *calc* provides command line execution for CalculationEngine. This chapter describes how to run this program. Basic command is mentioned below.

$ calc <engine_name> <list of integer> or <file_list>

All file names have an implicit ".txt" on the end. So please enter only file name except ".txt"

NOTE : *calc* executable is located in *bin* directory.

1.2.1 Multiplier

Multiplier allows only file name for the input data. For instance, you can execute Multiplier like mentioned below.

```
bin$ ./calc Multiplier input1
*2*4*6*4*3*10*100*20*50
576000000
```

In addition, you can enter one more files like mentioned below.

```
bin$ ./calc Multiplier input1 input2
*2*4*6*4*3*10*100*20*50*88*4*60*5*5*6
1824768000000000
```

First line shows all integers will be calculated. And second line is the calculated result.

NOTE : As an limitation, the file name is only supported. If you enter integer in the command line, the Multiplier produces error.

```
bin$ ./calc Multiplier 1 2
[Multiplier][Error] read_input_data : 2
```

1.2.2 Divider

Divider allows file name and integer value. For instance, you can execute Divider with mixed input data like mentioned below. In this case, Divider takes input1.txt and 4 as the input data.

```
bin$ ./calc Divider input1 4
2/4=0.5
6/4=1.5
3/10=0.3
100/20=5
50/4=12.5
```

The file, input1.txt contains the number listed below.

```
2 4 6 4
3
10
100 20 50
```

Number 4 is provided by the command line since the number of integer in input1.txt is odd.
As a result, Divider divides all the integers, one after another in order.

Of course, you can enter the list of file names and the list of integer.

```
bin$ ./calc Divider input2 input3
88/4=22
60/5=12
5/6=0.833333
10/-3=-3.33333
68/0=0

/bin$ ./calc Divider 10 5 4 5
10/5=2
4/5=0.8
```

2. Design Description

2.1 The content of input file

In order to supply the integers for calculation, the file should be provided.

NOTE : The files should be located in the same directory of *calc* program. (e.g. *bin* directory)

2.1.1 Separated by space and tab

The files contains one or more lines of integers. In the same line, integers can be separated by space or tab.

```
bin$ cat input2.txt
88 4 60 5
5
6
bin$ cat input2_tab.txt
88	4	60	5
5
6
```

2.1.2 Termination by alphabet and symbol

In case that the file contains non-integer value such as alphabet and symbol(e.g. #), Multiplier and Divider immediately terminates to read input data.
For instance, the files contains the data like listed below.

```
10 20 a 5
```

Both calculators saves only 10 and 20 for input data.

2.2 Input/Output storage

2.2.1 Multiplier

Multiplier has two member variables including an input storage and an output storage.

```
	std::vector<T1>* m_input_storage;
	T2* m_result;
```

The input storage, m_input_storage is std::vector. So Multiplier saves the input data with push_back() method.

```
m_input_storage->push_back(in_data);
```

Lastly, Multiplier save the calculated result to m_result

```
T2* m_result;
```

2.2.2 Divider

In the similar way, Divider has two storage as well.

```
	std::vector<T>* m_input_storage;
	std::vector<float>* m_result;
```

For the output storage, Divider uses std::vector because it can have one more result.

NOTE : If you want more precise resolution, you can change the type of m_result to *double*.

2.3 Calculation on a series of integers

CalculationEngine can take a series of integers by read_input_data method. *calc* program also hands over a serial of integers to CalculationEngine in the command line.

2.3.1 Multiplier

Firstly, using isNumber() function, Multiplier checks if the input data is a number or string in read_input_data method.

```
inline bool isNumber(string s)
{
      // first can be +, - and digit
      if(s.empty() || ((!isdigit(s[0])) && (s[0] != '-') && (s[0] != '+')))
            return false;

    for (unsigned long i = 1; i < s.length(); i++) {
            if (isdigit(s[i]) == false)
                  return false;
    }

    return true;
}
```

If the input data is a number, Multiplier return error because it support only file as the input.

```
      if (true == isNumber(param))
            return RES_ERR_NOT_SUPPORT;
```

If the input data is a file, Multiplier considers it as the file name. Then it extract integers from that file and store the integer to the input storage.

```
      while(data_file >> in_data) {
            m_input_storage->push_back(in_data);
      }
```

Lastly, Multiplier multiplies all the integers in the input storage and saves the calculated result to the output storage, m_result.

```
      for (unsigned int i = 0; i < m_input_storage->size(); i++) {
            tmp = tmp * m_input_storage->at(i);
            // check max/min value of T2
            if (tmp > numeric_limits<T2>::max() ||
                tmp < numeric_limits<T2>::min())
                  return RES_ERR_OUT_OF_RANGE;
      }

      *m_result = tmp;
```

2.3.2 Divider

Firstly, in read_input_data method, Divider checks if the input data is a number or string using isNumber() function.
In case of the number, Divider save it to the input storage.

```
      if (true == isNumber(param)) {
            try {
                  in_data = stoi(param);
            } catch (const out_of_range& oor) {
                  return RES_ERR_OUT_OF_RANGE;
            }
            m_input_storage->push_back(in_data);
      }
```

In case of the file name, Divider saves all the integer in that file by the same way of Multiplier.

Next, Divider checks the size of the input storage because it cannot calculate if the size is zero or odd.

```
      if (m_input_storage->empty() || ((m_input_storage->size() % 2) != 0))
            return RES_ERR_SIZE;
```

Lastly, Divider divides all the integers in the input storage, one after another. Then Divider saves the calculated result to the output storage in order.

```
      for (unsigned int i = 0; i < m_input_storage->size(); i=i+2) {
            if (m_input_storage->at(i+1) == 0) {
                  // In case of divide by zero, make the result zero
                  m_result->push_back(0);
                  continue;
            }
            else {
                  res = m_input_storage->at(i) /
                        static_cast<float>(m_input_storage->at(i+1));
                  m_result->push_back(res);
            }
      }
```

2.4 Factory method

CalculationEngine provides an abstraction class, *CalculationEngine*. A user who wants to implement the additional engine(e.g. subtraction) can inherit *CalculationEngine* class in order to implement three virtual methods.

```
class CalculationEngine
{
public:

      virtual ~CalculationEngine() {}

      static CalculationEngine *generateCalculator(Calculation cal_t,
                                                   InputType input_t);

      virtual CResult read_input_data(std::string &param) = 0;
      virtual CResult calculate() = 0;
      virtual void display_result() = 0;
};
```

In addition, *CalculationEngine* supplies a factory method, generateCalculator() so that the user is able to generate Multiplier and Divider engines easily.

When the factory method is called, the calculation and the allowable input type should be given. From these parameters given, the factory method is able to call the corresponding constructor defining the specific data type.

```
      if (cal_t == MULTIPLICATION) {
            if (input_type == INTEGER)
                  return new Multiplier<int,long long>;
      }
      else if (cal_t == DIVISION)
            if (input_type == INTEGER)
                  return new Divider<int>;
```

NOTE : In this version of CalculationEngine, only integer can be supported.

2.5 Template class

The class of Multiplier and Divider is the template class.

```
template <class T1, class T2>
class Multiplier : public CalculationEngine
{
[snip]
};

template <class T>
class Divider : public CalculationEngine
{
[snip]
};
```

For Multiplier, T1 is for the data type of the input storage and T2 is for the data type of the output storage. For Divider, T is for the data type of the input storage.

The types of template is defined in the factory method, CalculationEngine::generateCalculator().

2.6 Error handling

CalculationEngine defines several error case by CResult.

```
typedef enum {
      RES_OK = 0,               /**< No Error **/
      RES_ERR,                  /**< General Error **/
      RES_ERR_NOT_SUPPORT,      /**< Not support functionalities
                                     (e.g. wrong data type) **/
      RES_ERR_FILE_NOT_EXIST,   /**< There are no files to open **/
      RES_ERR_OUT_OF_RANGE,     /**< The value exceed its range **/
      RES_ERR_SIZE,             /**< Size Error (e.g not enough to calculate) **/
      RES_ERR_INVALID_PARAM,    /**< Invalid parameter was given**/

      RES_MAX
} CResult;
```

2.6.1 RES_ERR_NOT_SUPPORT

This error can happen if the input data is a number in Multiplier::read_input_data method. Please refer to 2.3.1 Multiplier for more details.

2.6.2 RES_ERR_FILE_NOT_EXIST

Multiplier and Divider handle the files in order to read the integers from that files. For this, both calculators open the file. In this step, read_input_data method produces this error if the file is not open.

```
            fstream data_file(file_name.c_str(), std::ios_base::in);
            if (!data_file.is_open())
                  return RES_ERR_FILE_NOT_EXIST;
```

2.6.3 RES_ERR_OUT_OF_RANGE

For Multiplier, this error happens when the calculated result exceeds the scope of the data type of the output storage. Multiplier checks the max/min value of T2 by std::numeric_limits.

```
	for (unsigned int i = 0; i < m_input_storage->size(); i++) {
		tmp = tmp * m_input_storage->at(i);
		// check max/min value of T2
		if (tmp > numeric_limits<T2>::max() ||
		    tmp < numeric_limits<T2>::min())
			return RES_ERR_OUT_OF_RANGE;
	}
```

For Divider, there are two cases producing this error.
First one is when the Divider converts the integer from the input data to the real integer using std::stoi. Std::stoi is for conversion from string to integer. So if the input parameter is out of scope, std::stoi throws the out_of_range exception. Then, Divider::read_input_data method catches this exception and return RES_ERR_OUT_OF_RANGE error.

```
		try {
			in_data = stoi(param);
		} catch (const out_of_range& oor) {
			return RES_ERR_OUT_OF_RANGE;
		}
```

Second one is when the input data from the file exceeds the scope of Integer.
In case that the file contains the number exceeds the scope of Integer, Divider is not able to handle it since the current version of Divider supports only Integer as the input data.

```
		while(data_file >> in_data) {
			if((in_data > numeric_limits<int>::max()) ||
			   (in_data < numeric_limits<int>::min()))
				return RES_ERR_OUT_OF_RANGE;

			m_input_storage->push_back(in_data);
		}
```

2.6.4 RES_ERR_SIZE

In Divider::calculate method, Divider checks the size of the input storage before it calculates. If the size of the input storage is less than 2 or odd, Divider produces this error because it is not possible to calculate.

```
	if (m_input_storage->empty() || ((m_input_storage->size() % 2) != 0))
		return RES_ERR_SIZE;
```

2.6.5 RES_ERR_INVALID_PARAM

In Divider::read_input_data method, Divider checks if the file name is blank or not. If the file name is blank then this error happens.

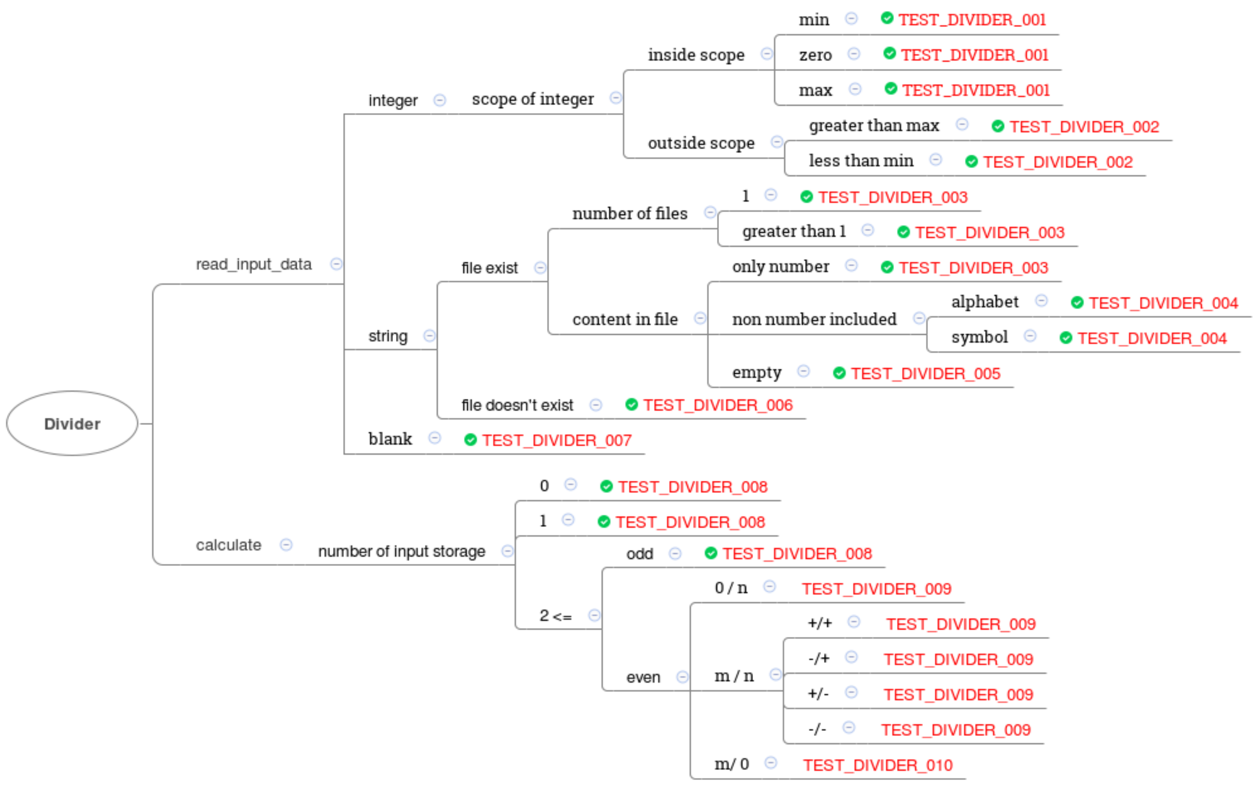
```
		if (param.size() == 0)
			return RES_ERR_INVALID_PARAM;
```

3. API specification

Please refer to the doxygen html document. Please open doc/html/index.html

4. Unit Test

For the unit test, *UnitTest* class is provided. There are 10 test cases derived from the mindmap mentioned below. (doc/mm4ut.png)
And please refer to the doxygen html document for the test description. (doc/html/class_unit_test.html)



4.1 How to run

You can execute the unit test in the command line like mentioned below. You can find the procedure of each unit test and the final result from stdout.

```
bin$ ./calc UnitTest
#########################################
###        TEST_DIVIDER_001         ###
#########################################
[Test Step 001] Input a value : -2147483648
[Pass Criteria 001-1] Check the size of input storage
---> PASS
[Pass Criteria 001-2] Check the out value if it is equal to min value
---> PASS

[snip]
```