Calculation Engines

Xingwu Zheng

September 17, 2018

1 GOAL

1.1 Program function

Develop a program that performs calculations on a series of integers. The program should supply two engines: multiplier and divider.

1.2 A user command line interface

User can chose which engine to use by pass the parameters in command line.

1.3 Extendable module design

These modules should have the standard interface so that a user can write his own engine.

1.4 Input and output files

- Inputs: .txt files or lines of integers.
- Outputs: display on *stdout*.

2 Set up the Calculation Engines

2.1 Program Installing

The first task is to setup the program. As the program is written in c++, a standard C++ compiler is required to be installed. Users have two way to install the program, through the *git* command or unpack the source code file.

2.1.1 Download the code through git

User can use the command line
git clone https://github.com/xingwuzheng/calculation_engines.git

2.1.2 Unzip the source code

The code is stored in a tar.gz archive. User need to extract the program tar.gz archive by using the following command:

```
tar -zxvf calculation.tar.gz
```

Then the user will see the folder *calculation*. Go into the folder. Users need to clean the program first by using the command:

```
make clean
```

Then build the program by using the command:

```
make
```

Then the program "calc" is complied and ready to use. The user can also compile the program themselves by using the command:

```
g++ -o calc src/*.cpp -std=c++11
```

The user can also add their own engines then compile the program to use. The following subsection will provide details information on how to run the program.

2.2 The Command Line Keys

The program will run from the command line. It has two versions:

```
calc <engine_name><file_list>
calc <engine_name><list of integers>
```

That is, to use the calculation engines, the following command line examples are valid. Noted that the *1.txt* and *2.txt* are already inside the folder, feel free to write your own files for the inputs.

```
./calc Divider input 1.txt
25
./calc Divider input 1.txt 2.txt
0.25
./calc Divider 50
50
./calc Divider 50 2
25
./calc Muiltiplier input 1.txt
100
./calc Muiltiplier input 1.txt 2.txt
10000
./calc Muiltiplier 50
50
./calc Muiltiplier 50 2
100
```

The following command line examples are invalid. The error message will show in terminal. And the following command lines are also tested in the unit test.

```
./calc Add 50 2
Invalid Engine!
Illegal Inputs!
./calc
Not enough arguments!
```

```
Illegal Inputs!
./calc Divider
Not enough arguments!
Illegal Inputs!
./calc Divider input 50
Illegal files input!
Illegal Inputs!
./calc Divider input a.txt
Illegal Inputs!
./calc Divider 50 0
Inputs contain ZERO dividor!
./calc Divider 50 a
Illegal integers input!
Illegal Inputs!
```

2.3 Unit test

We use a simple shell unit testing tool called "roundup". The tool is very easy to use and already packed in the folder *unit_test*. To run the unit test, user can use the following command:

```
bash run_unit_test.sh
```

This command will generate the unit test tool "roundup" framework for the user and automatically run the unit test file.

2.3.1 Unit test details

The unit test file is in the folder *unit_test* and called *test.sh*. User can define their own unit test for the Divider engine, or the Multiplier engine. We only test the Divider engine here by requirement. We use 16 unit test cases to test our program. The first 9 tests is correct. We compare the correct results with the outputs.

```
./calc Divider 50 2
./calc Divider 2 50
./calc Divider 50
./calc Divider 2
./calc Divider input 1.txt
./calc Divider input 2.txt
./calc Divider input 1.txt 2.txt
./calc Divider input 3.txt
./calc Divider input 1.txt 2.txt 3.txt
```

The next 7 tests is incorrect. We compare the incorrect messages with the outputs.

```
./calc Add 50 2
./calc
./calc Divider
./calc Divider input 50
./calc Divider input a.txt
./calc Divider 50 0
./calc Divider 50 a
```

All 16 test cases are passed.

3 Design Document

This section we discuss our design document.

3.1 Brief Models Chart

```
Main
├── Parsing
└── Factory
    ├── Multiplier Engine
    ├── Divider Engine
    └── More Engines to be added
```

3.2 Models

3.2.1 main.cpp

The *main.cpp* is the Main function. It take the input and pass the input files or integers to the Parsing model. If the inputs are valid, the main function pass the engine name and data to the factory model and run the calculation engine. The main function also display the results or error messages.

3.2.2 parsing.h

The *parsing.h* is the Parsing model. It take the input files or integers and parse them into a input vector.

3.2.3 factory.h&factory_entry.h

The *factory.h&factory_entry.h* is the Factory model. It get the engine name and data from the main function. Pass the income parameter to selected calculation module and return the module to main function.

3.2.4 multiplier.h&multiplier.cpp

The *multiplier.h&multiplier.cpp* is the Multiplier engine model. It takes the input vector from the Factory model and returns a multiplication.

3.2.5 divider.h&divider.cpp

The *divider.h&divider.cpp* is the Divider engine model. It takes the input vector from the Factory model and returns a division.

3.3 User Define Engines

This program is designed to be an extendable one. The user can easily define their own calculation engines. User could write the special calculation engines as the subclass of the *factor* class. Then they can write their own *compute* function to fit their request.