BLG 252E - Object Oriented Programming Assignment #2

Due: April 28th 23:59

> Money, money, money
> Must be funny
> In the rich man's world
>
> *-Abba (1976)*

Introduction

You're a determined entrepreneur who's just set up a new car factory. Now, you've got 30 days to figure out how to invest smartly to make the most profit. You can buy machines and/or hire workers on any day. Each production unit has a daily cost and daily return value. But, be careful! Each unit has its own pros and cons to consider.

Given the description of the classes, implement them carefully and put them together to run the simulation for 30 days with your investment policy, and see how much you can earn.

For any issues regarding the assignment, you can ask your questions on Ninova Message Board. Before writing your question, please check whether your question has been asked. You can also contact T.A. Enes Erdoğan (**erdogane16@itu.edu.tr**).

1 Implementation Details

You are given a skeleton code that will help you to complete the project. Make sure you understand the structure of the code and the relations among the classes. Drawing a UML-like diagram might help. All necessary libraries are already included, so do not include any other library. Also, **do not modify any given header files** in any way!

For the assignment, you are expected to implement six classes: **Unit**, **Worker**, **Machine**, **Head Worker**, **Factory**, and **Simulation**. They are all connected somehow. You will discover their relations as you go through this document and the skeleton code.

The important details are explained here. For further details, read the given skeleton code carefully.

1.1 Unit

This is the base class of the Worker and Machine classes. So, it contains all the common features of these classes. Implement its constructor and related getter functions declared in the given header file.

```cpp
class Unit{
private:
    std::string m_name;
    float m_cost_per_day;
    float m_base_return_per_day;
    ...
}
```

1.2 Worker

A worker **is kind of** a unit. It has two additional attributes for experience and a counter for the number of head workers. Hiring a worker may not yield too much initially, but as a worker works, it gains experience. And if it reaches the 10 experience point, it will be promoted to head worker by the Factory.

When there are head workers around, regular workers work more efficiently. So, you also need to keep track of the number of head workers.

Implement declared functions in the header file. To implement the `getReturnPerDay()` function, use the following formula:

$$\text{daily return} = (\text{base return}) + (\text{experience}) * 2 + (\text{number of head workers}) * 3$$

Also, increase the experience, each time `getReturnPerDay()` called.

1.3 Machine

Machine also **is a kind of** a unit. It has the following declaration.

```cpp
class Machine: public Unit{
private:
    float m_failure_probability;
    int m_repair_time;
    float m_price;
    float m_repair_cost;

    int m_days_until_repair;
    ...
}
```

As can be seen, there are 5 additional attributes, which will be initialized based on the provided Marketplace Data.

Buying a machine has both one time paid price and a daily cost. There is a small probability that it will fail for a few days causing you to spend more money on repairing. However, it has quite a high return value.

Implement declared functions in the header file. Inside the `getReturnPerDay()` function, follow the these steps:

1. Check if the machine is currently broken. If so, return 0.

2. Machine will fail with probability `m_failure_probability`. Using `std::rand()` function, check if it is failed or not. If it fails, it will not be used for `m_repair_time` days. And return `-m_repair_cost`

3. If it does not fail, return the base return value.

1.4 Head Worker

Head worker **is a kind of** worker. It has a higher return than the regular worker. You can not directly hire the head worker. You need to wait for 10 days after hiring a worker then it will be promoted automatically by the Factory.

Implement declared functions in the header file. To implement the `getReturnPerDay()` function, use the following formula:

$$\text{daily return} = (\text{base return}) + (\text{experience}) * 5$$

1.5 Factory

A factory has three containers for workers, machines, and head workers. To implement the `passOneDay()` function follow these steps:

1. Calculate the overall daily costs and daily returns, and update the capital value

2. Promote the workers to head worker if they reach the 10 experience.

1.6 Simulation

The purpose of the simulation class is to encapsulate the user interface into a class. At the beginning, the simulation object will read the labor_market.txt and machines_market.txt files and put them into vectors. When you want to hire or buy a unit, you will randomly pick one from the corresponding vector.
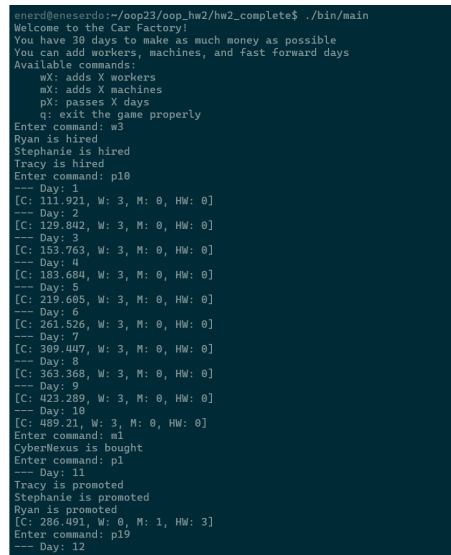
Then, the simulation object will handle the user inputs (e.g. hire N workers, buy N machines, or pass N days) and call the corresponding functions in the factory object.

2 Marketplace Data

In the project folder, there are two text files inside the data folder: labor_market.txt and machines_market.txt, they contain the all possible workers to hire and all available machines to buy. When you need to initialize a Worker or Machine object, randomly sample from these data. There are 100 workers and 100 machines available at max.

3 Gameplay

In this game, for each day you can either hire a worker, or buy a machine or fast forward days. Figure 1 shows an example gameplay.

Figure 1: An example game play (it did not fit to screenshot, but the idea is clear).

Be aware that if you can barely afford a machine, when machine fails you can go bankrupt due to its repair cost and daily cost. Such a scenario is given in the test file.

4 How to compile, run, and test your code

If you want to compile and run the provided code on a terminal, you can use these commands:

```
g++ -Wall -Werror src/*.cpp -Iinclude -o bin/main
```

```
./bin/main
```

Obviously, when there is randomness, the standardized calico test will not work as expected. To overcome this issue, we call std::srand(34) to set the seed to a pre-determined constant value and make the randomness reproducible. However, it also depends on the order of std::rand() calls. So, be aware that there is a slight chance that you may not pass the tests even if your implementation works perfectly fine.

```
python3 -m calico.cli hw2_tests.yaml
```

You should run Valgrind on a terminal to check your assignment with the command:

```
valgrind --tool=memcheck --leak-check=full --show-leak-kinds=all \
--track-origins=yes --log-file=valgrind_log.txt ./bin/main
```

5   Submission

Compress your whole project folder and submit it as a zip file on Ninova. Before submitting please make sure you are passing all cases in the Calico file. Be aware that your grades will be based on the quiz that is going to be held two weeks later. The exact date will be announced.

6   (Optional) Basics of Makefile

You might notice that even though there is not too much code, compiling takes a considerable amount of time (about 2 seconds, which is unacceptable). Even though you only change a single line from one file, the whole project is compiled from scratch.

To deal with this issue, makefiles offer a straightforward solution for organizing code compilation. They are mostly used to break down the compilation process into separate steps. So, only the required part of the code is compiled and combined with the rest.

A makefile essentially contains a set of rules that are executed by the GNU Make build system. A rule generally looks like the following:

```
target_file: dependency_files
     command
```

The target is the file we want to generate using the provided commands given the dependency files are available. Only if dependency files are changed, the target file will be created again. If dependency files are not available it will run the corresponding rule for the dependency.

You are provided a makefile to make the compilation process easy and efficient. Please try to understand the file. Available commands for the given makefile are the following:

- `make`: Compiles the code

- `make run`: Compiles and runs the code

- `make test`: Runs the calico test

- `make valgrind`: Runs the valgrind

- `make clean`: Deletes object files and the binary file

Note that the given makefile can be written in a more elegant and concise way. However, for the sake of simplicity, everything was written explicitly [1].

❖   **Notice:**   This section is just for students who want to invest more time into advancing their C++ knowledge. It is completely optional, and you will not be responsible for anything presented here.

---

[1]For the motivated students, here is a detailed tutorial