YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

CPSC 427: Object-Oriented Programming *Handout #9*
Professor M. J. Fischer *December 3, 2018*

Problem Set 7

Due before midnight on Wednesday, December 12, 2018.

1 Introduction

This problem set continues the development begun in Problem Set 4 of a simulator for a population of simple agents attempting to reach consensus on a choice value. The PS4 assignment handout describes two different such agent algorithms: *fickle* and *follow the crowd*. In PS4, you implemented a simulator for a collection of *fickle* agents.

In the PS5 assignment, you refactored the PS4 solution to add a number of new features. In particular, you made `Agent` into a pure abstract class, you split off a new `Population` class from the `Simulator` class, and you changed the method used for building the initial population from the command line parameters.

In the PS6 assignment, you implemented new classes `Block` and `Blockchain`, and you imported and perhaps modified demo classes `SPtr` and `Serial`.

The goal of this assignment is to simulate the blockchain consensus algorithm used in Bitcoin cryptocurrency, sometimes called *Nakamoto consensus*, to see how consensus gradually evolves.

2 Teaching Objectives

- Gain more experience in refactoring code to adapt it to new requirements.
- Learn how to factor out common code in a polymorphic class hierarchy.
- Find additional uses of delegation in order to keep functions close to the data members that they need.
- Learn how objects of class `Blockchain` can be passed around freely in the simulator without concern about storage management issues.
- Take a step closer to simulating a realistic blockchain consensus algorithm.

3 Problem

Integrate and extend your PS5 and PS6 solutions to result in a simulator for agents that are attempting to agree on a blockchain rather than on a single bit. In particular, you will need to do the following:

1. Find everywhere in your code from PS5 involved with reaching consensus on an a 0-1 value of type `int`. Change the value type instead to `Blockchain`. In particular, the abstract virtual function `choice()` in `Agent` should now return a value of type `Blockchain` rather than of type `int`.

2. Add a new abstract virtual function `extend()` to class `Agent()` that causes the agent to extend its current blockchain and to make the extended blockchain its new choice. This will apply to all agent classes, including `Fickle` and `Crowd`.[1]

3. Add another agent class, `Nakamoto`, to model the Nakamoto algorithm's rule of ignoring a new blockchain received from another agent unless the new chain is longer than the current one, in which case the longer blockchain replaces the shorter one.

4. Add a new class `AgentBase` that is publicly derived from `Agent` and from which the actual agents `Fickle`, `Crowd`, and `Nakamoto` will be publicly derived. The purpose of `AgentBase` is to give a place for the data members and functions that are common to all agents such as the agent's current choice and the public functions `choice()` and `extend()`. The reason for not putting these things directly in `Agent` is that `Agent` is a pure abstract class, so it cannot be instantiatied and therefore also cannot have data members.

5. Change `Simulator` to randomly decide at each step whether to perform an update() operation or an extend() operation. It does this by calling dRandom() and comparing the result to a new command line argument `probExtend`. In case it decides to simulate an extend, it chooses an agent at random to perform the extend. Similarly, if it decides to simulate the sending of a message, it works as in PS4 and PS5 by choosing a random sender and a distinct random receiver for the simulated sending of a message.

6. Instead of running until consensus is reached, the new simulator will run for `maxRounds`, which is a new command line argument that is passed to `Simulator::run()` as a parameter. Thus, there is no longer any need for the functions and data members that were formerly involved in trying to determine whether or not consensus had been reached, and if so, what the consensus value was. Rather, at the end of the simulation, we'll simply print out a list of agents with each agent's current choice.

7. The code in class `Population` that creates the population should now make a 3-way choice between `Fickle`, `Crowd`, and `Nakamoto`. New command line parameters provide the desired probabilities for each kind of agent. All agents, regardless of type, now all start with *copies* of the same initial genesis `Blockchain` for their initial choice. The genesis `Blockchain` contains a smart pointer that points to the genesis `Block`. The genesis `Block` is unique in that its `SPtr` data member has both `target` and `count` set to `nullptr`.

8. `Population` should also have functions `extend(int receiver)` and `sendMessage(int senter, int receiver)` to translate between the simulator's use of integers to identify agents and the agents themselves, which actually carry out those operations. The agents in turn delegate some of the work to `Blockchain` functions that were defined in PS6.

9. Modify `main.cpp` to accept the following command line arguments,
   `numAgents maxRounds probNak probFickle probExtend [seed]`
   where the arguments have the following meanings:

[1]This is intended to model what happens when a Bitcoin miner successfully solves the current proof-of-work puzzle and is therefore allowed to add a set of transactions to the current blockchain.

`numAgents` The total number of agents (as before).

`maxRounds` The total number of simulation rounds to perform.

`probNak` The probability of selecting a `Nakamoto` agent when building the population.

`probFickle` The probability of selecting a `Fickle` agent when building the population.

`probExtend` The probability that the simulator chooses to simulate an `extend()` operation rather than a `sendMessage()` operation.

`[seed]` Optional seed for the random number generator (as before).

The probability of selecting a `Crowd` agent is $1.0 -$ `probNak` $-$ `probFickle`. It is an error if the result does not lie in the closed interval $[0.0, 1.0]$.

10. Modify `Population` to have two print functions, one which prints the statistics as in PS5 (but I'm now calling it `printStats()`), and one that prints out each agent's choice of blockchain, one per line (which is what I'm now calling `print()`). Naturally, `print()` delegates the printing of an agent's current choice to `Blockchain::print()`.

A sample call on your simulator is given on the Zoo in `/c/cs427/code/ps7/sample.sh`, along with the output from a run on my machine. To give a better idea of what the simulator is doing, I have added print statements to the simulator to show what operation is being performed at each step of the simulation. I've also added a print statement to `Population::extend()` to print each new blockchain when it is first produced.

4 Programming Notes

1. There should be no public data members and no use of `friend` classes or functions.
2. Dynamic memory (allocated by `new`) should only be used in the following places:
   (a) Within the furnished classes `SPtr` and `Serial`;
   (b) For creating objects of type `Block`;
   (c) For creating objects of type `Agent` and for creating the array of agents.

   All `Blockchain` objects should be stack allocated.
3. The only delete statements outside of class `SPtr` should be to delete objects created in case 2c above. You should not explicitly delete any blocks. They should be deleted automatically by the `SPtr` objects that manage them.

5 Grading Rubric

Your assignment will be graded according to the scale given in Figure 1 (see below).

#	Pts.	Item
1.	3	All relevant standards from previous problem sets are followed regarding submission, identification of authorship on all files, and so forth. A well-formed `Makefile` or `makefile` is submitted that specifies compiler options `-O1 -g -Wall -std=c++17`. Running `make` successfully compiles and links the project and results in an executable file `blockchain`. Each function definition is preceded by a comment that describes clearly what it does.
2.	2	Sample input and output files are submitted that show good coverage of the parameter space, e.g., small inputs, large inputs, edge cases for the probabilities (e.g., 0.0 and 1.0) as well as reasonable intermediate cases. This is in addition to the furnished sample file.
3.	3	The program shows good style. All functions are clean and concise. Inline initializations, inline functions, and `const` are used where appropriate. Variable names are appropriate to the context. Programs are consistently indented according to the course indenting style. Each class has a separate `.hpp` file and, if needed, a separate `.cpp` file. However, it is acceptable to group the three polymorphic agent classes together in the same `.hpp` and `.cpp` files.
4.	2	The restrictions in section 4 are all obeyed.
5.	10	All of the functionality in section 3 is correctly implemented.
	20	Total points.

Figure 1: Grading rubric.