Practice 3: Influencer Representation Agency

Programming 2

Academic Year 2023-2024

This practice involves managing an influencer representation agency and its impact on different social networks, and the corresponding monetization for the agency of said impact, all following the object-oriented programming paradigm. The necessary concepts to develop this practice are worked on in all the theoretical units, although especially in *Unit 5*.

Submission conditions

- The submission deadline for this practice is **Friday, May 24**, until **23:59**
- The practice consists of several files: `SNData.cc`, `SNData.h`, `SNFollowers.cc`, `SNFollowers.h`, `Influencer.cc`, `Influencer.h`, `Agency.cc`, `Agency.h`, `Util.cc`, and `Util.h`. All of these should be compressed into a single file called `prog2p3.tgz` that will be submitted through the practice server in the usual way. To create the compressed file you should do it in the following way (in a single line):

Terminal
```
$ tar cvfz prog2p3.tgz SNData.cc SNData.h SNFollowers.cc SNFollowers.h
            Influencer.cc Influencer.h Agency.cc Agency.h Util.cc Util.h
```

Honor Code

If plagiarism (total or partial) is detected in your practice, you will get a **0** for the submission and it will be reported to the management of the Higher Technical School so they can take disciplinary measures

It is okay to discuss with your peers possible solutions to the practices
It is okay to join an academy if it helps you to commit to study and do the practices

It is not okay to copy code from other classmates to solve your problems
It is not okay to join an academy so that they do the practices for you

If you need help, go to your teacher
Do not copy

General rules

- You must submit the practice solely through the practice server of the Department of Languages and Computer Systems (DLSI). There are two ways to access it:

  - DLSI main page (https://www.dlsi.ua.es), option "HOMEWORK UPLOAD"
  - Directly at the address https://pracdlsi.dlsi.ua.es

- Considerations you should have in mind when delivering:
  - The username and password for delivering practices are the same as those used in UACloud
  - You can submit the practice multiple times, but only the last submission will be graded
  - Deliveries via other means, such as email or UACloud, will not be accepted
  - Deliveries after the deadline will not be accepted

- Your practice must be able to compile without errors using the C++ compiler available in the Linux distribution of the practice labs

- If your practice cannot be compiled, its grade will be 0

- The correction of the practice will be done automatically (**there will be no manual correction by the professor**), so it is essential to strictly respect the texts and output formats indicated in this statement

- At the beginning of all delivered source files, you must include a comment with your NIF (or equivalent) and your name. For example:

SNData.h
```
// NIF 12345678X GARCIA GARCIA, JUAN MANUEL
...
```

- The calculation of the grade for the practice and its relevance to the final grade of the course are detailed in the presentation slides of the course (*uNIT 0*)

1 Description of the practice

In this practice, the management of an influencer representation agency will be implemented, which on one hand will sign influencers (agreeing on a commission that the agency will take), and on the other hand will organize events that will have an impact on various influencers and on different social networks. Depending on the success of the event, usually there will be an increase in followers for each influencer on the social networks where the agency advertised the event, and logically the increase in followers will be monetized. Periodically, or whenever the agency director wants, the money generated by the events will be collected, applying the agreed commission with each influencer.

2 Implementation details

In the *Moodle* of the course, various files will be published that you will need for the correct completion of the practice:

- `Util.h` and `Util.cc`. The `Util.h` file contains some types necessary for the practice, and the definition of the `Util` class with auxiliary methods; the `Util.cc` file contains the implementation of these methods

- `prac3.cc`. File that contains the `main` of the practice. It is responsible for creating objects of the classes involved in the problem, and simulate an event, and collect the money generated by the event. This file should not be included in the final delivery, it is only a help to test the practice, and it can be modified or used as a base for other tests

- `makefile`. File that allows compiling optimally all the source files of the practice and generating a single executable

- `autocorrector-prac3.tgz`. Contains the files of the auto-corrector to test the practice with several unit tests to test the methods separately. The automatic correction of the practice will be performed with a similar corrector, with these tests and others more defined by the faculty of the course

In this practice, each of the classes will be implemented in a separate module, so that we will have two files for each of them: `SNData.h` and `SNData.cc` for social network data, `SNFollowers.h` and `SNFollowers.cc` to manage the followers of an influencer on a social network, `Influencer.h` and `Influencer.cc` for influencers, `Agency.h` and `Agency.cc` for the agency, and `Util.h` and `Util.cc` for auxiliary methods. These files, together with `prac3.cc`, should be compiled together to generate a single executable. One way to do this is as follows:

Terminal
```
$ g++ SNData.cc SNFollowers.cc Influencer.cc Agency.cc Util.cc prac3.cc -o prac3
```

This solution is not optimal, as it compiles all the source code again when it may be that only some of the files have been modified. A more efficient way to perform the compilation of code distributed in multiple source files is through the `make` tool. You should copy the `makefile` file provided in Moodle into the directory where the source files are located and enter the following command:

Terminal
```
$ make
```

- You can consult slides 60 onwards of *Unit 5* if you need more information on the functioning of `make`

2.1 Exceptions

Some of the methods you will create in this practice must throw exceptions. To do this, you should use `throw` followed by the type of exception, which in C++ can be any value (an integer, a string, etc.), but in this practice it must be one of the values defined in the enumerated type `Exception` that is in `Util.h`, or the standard exception `invalid_argument` (as explained later):

Terminal
```
enum Exception {
  EXCEPTION_INFL_NOT_FOUND,
  EXCEPTION_WRONG_COMMISSION,
  EXCEPTION_UNKNOWN_SN
};
```

Exceptions can be caught using `try/catch` wherever a method that can throw an exception is invoked, and they can also be propagated (in C++ it is enough not to put a `try/catch` for it to propagate). In those methods that throw more than one exception, the exceptions should be thrown in the order in which they are mentioned in the method description. In this practice, only exceptions of type `Exception` should be caught; the `invalid_argument` should not be caught.

Below is an example of how an exception would be thrown from a method and how it would be caught elsewhere in the code:

```
// Method where the exception occurs
void Influencer::setCommission(double commission)
{
  ...
  if(...){  // comission is not in the valid range of values
    throw EXCEPTION_WRONG_COMMISSION;
  }
  ...
}
...
// Function or method where the exception is caught
  ...
  try{
    ...
    influencer.setCommission(0.9); // might throw EXCEPTION_WRONG_COMMISSION

    // if it is correct, the code continues
    ...
  }
  // If the exception occurs, we display the corresponding error
  catch(Exception e){
    if (e == EXCEPTION_WRONG_COMMISSION)
        Util::error(ERR_WRONG_COMMISSION);
    else
        Util::debug(e);   // if it's another exception, there's a programming error
  }
}
```

To throw the exception `invalid_argument` you would do:

```
  if (...)
   throw invalid_argument(message)
```

where `message` is a string (`string` or vector of `char`); if the invalid argument is a number, it can be converted to string with the function `to_string`:

```
  if (...)
   throw invalid_argument(to_string(number))
```

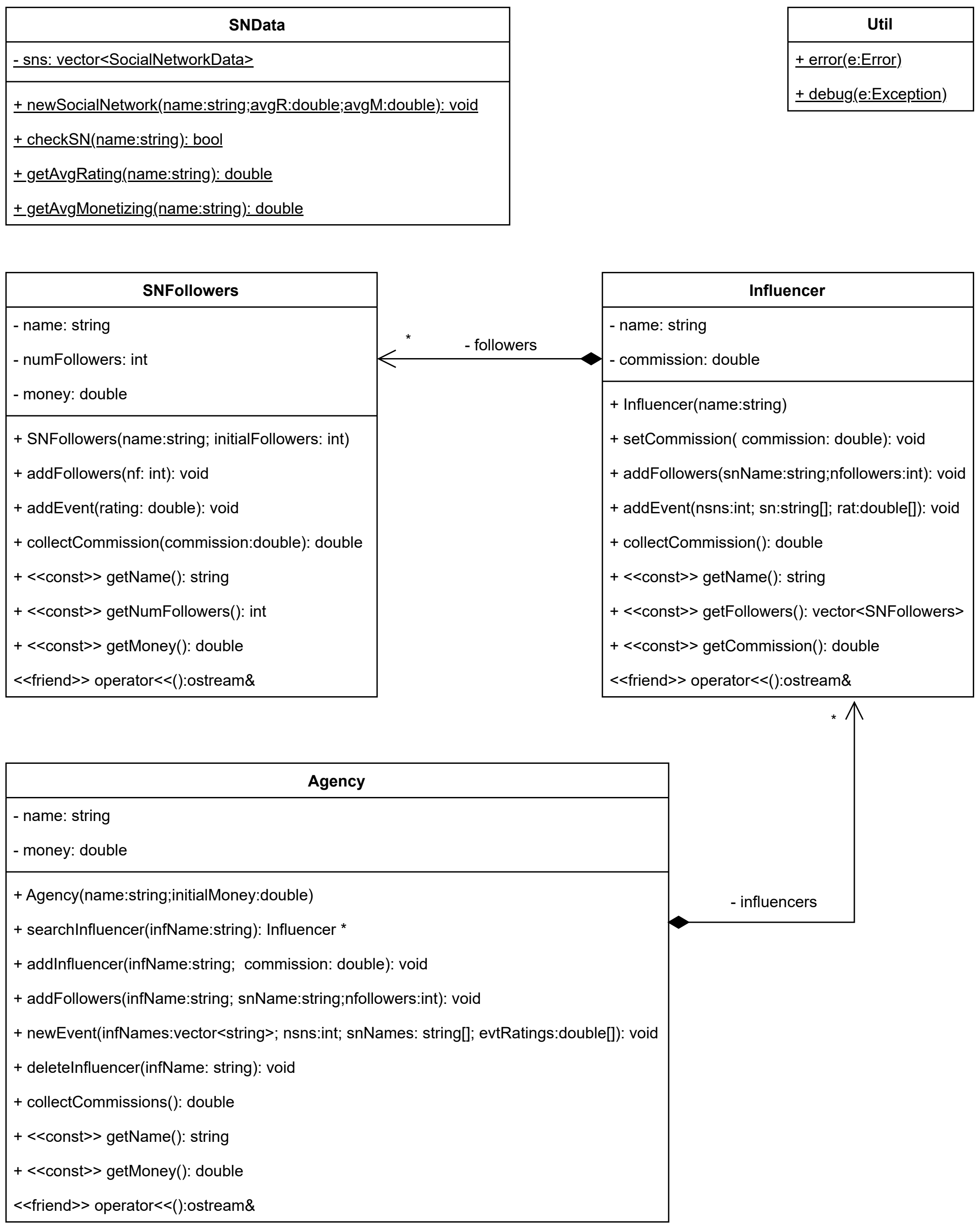
To use the `invalid_argument` you must include the header `<stdexcept>`.

3 Classes and methods

The figure on the next page shows a UML diagram with the classes to be implemented, along with the attributes, methods, and relationships that take place in the scenario of this practice.

In this practice, it is not allowed to add any public attributes or methods to the classes defined in the diagram, nor to add or change the arguments of the methods. If you need to incorporate more methods and attributes to the classes described in the diagram, you can do so, but always including them in the private part of the classes. Remember also that *aggregation* and *composition* relationships give rise to new attributes when translated from the UML diagram to code. Consult slides 58 and 59 of *Unit 5* if you have doubts about how to translate *aggregation* and *composition* relationships to code.

Below are the methods of each class. Some of these methods may not be necessary to use in practice, but they will be used in unit tests during the correction. It is recommended to implement the classes in the order they appear in this statement.


SNData
- sns: vector<SocialNetworkData>
+ newSocialNetwork(name:string;avgR:double;avgM:double): void
+ checkSN(name:string): bool
+ getAvgRating(name:string): double
+ getAvgMonetizing(name:string): double
Util
+ error(e:Error)
+ debug(e:Exception)
SNFollowers
- name: string
- numFollowers: int
- money: double
+ SNFollowers(name:string; initialFollowers: int)
+ addFollowers(nf: int): void
+ addEvent(rating: double): void
+ collectCommission(commission:double): double
+ <<const>> getName(): string
+ <<const>> getNumFollowers(): int
+ <<const>> getMoney(): double
<<friend>> operator<<():ostream&
*
- followers
Influencer
- name: string
- commission: double
+ Influencer(name:string)
+ setCommission(commission: double): void
+ addFollowers(snName:string;nfollowers:int): void
+ addEvent(nsns:int; sn:string[]; rat:double[]): void
+ collectCommission(): double
+ <<const>> getName(): string
+ <<const>> getFollowers(): vector<SNFollowers>
+ <<const>> getCommission(): double
<<friend>> operator<<():ostream&
*
- influencers
Agency
- name: string
- money: double
+ Agency(name:string;initialMoney:double)
+ searchInfluencer(infName:string): Influencer *
+ addInfluencer(infName:string; commission: double): void
+ addFollowers(infName:string; snName:string;nfollowers:int): void
+ newEvent(infNames:vector<string>; nsns:int; snNames: string[]; evtRatings:double[]): void
+ deleteInfluencer(infName: string): void
+ collectCommissions(): double
+ <<const>> getName(): string
+ <<const>> getMoney(): double
<<friend>> operator<<():ostream&

3.1 Util

This class will be provided in the Moodle of the subject, which will include:

- the enumerated type `Error` with all possible errors that can occur in the program, in addition to the `error` method to issue the corresponding errors on the screen. In this practice, errors are only issued in the file `prac3.cc`
- the enumerated type `Exception` with all possible exceptions that may occur.
- a `debug` method as help in debugging exceptions (not necessary to use, but can be useful for finding faults related to exceptions)

- Keep in mind that `error` and `debug` are class methods (`static`) and therefore should be invoked using this syntax: class name, followed by `::` and finally the method name. See slides 45 and 46 of *Unit 5* if you have doubts about this respect. For example, to display the error message `ERR_NAME`, you must invoke the `error` method passing the corresponding parameter in the following way:

```
Util::error(ERR_NAME);
```

3.2 SNData

This class is used to store information about social networks, and for this purpose, it uses a static `vector<>` (`sns`) that contains records like the following:[1]

Terminal

```
struct SocialNetworkData {
   string name;
   double averageRating;
   double averageMonetizing;
};
```

For each social network, the name (`name`), the average rating of its events (`averageRating`), and the average monetization obtained (`averageMonetizing`); these values are indicated in the constructor and do not change during the program execution. The methods of this class are all static, and are as follows:

- `void newSocialNetwork(string name,double avgR,double avgM)`. Adds a new record at the end of the `sns` vector with the data of the new social network. If a network with that name already exists, the standard `invalid_argument` exception is thrown with the network name as the parameter. The `avgR` and `avgM` values must be greater than 0 and less than 1, otherwise the `invalid_argument` exception should be thrown with the incorrect value converted to string using `to_string`, for example with `to_string(avgM)`.
- `bool checkSN(string name)`. Searches in the `sns` vector for a social network called `name`, and returns `true` if found, and false otherwise.
- `double getAvgRating(string name)`. Returns the average rating of the social network called `name`, or `0.0` if not found.
- `double getAvgMonetizing(string name)`. Returns the average monetization of the social network called `name`, or `0.0` if not found.
- It is allowed to add new private static methods.

[1]You will need to copy the record in the file `SNData.h`, before the class.

3.3 SNFollowers

This class stores the social network's name (`name`), followers (`numFollowers`), and the money earned (`money`) by an influencer on a social network, and recalculates the number of followers and the money when an event occurs. The methods of this class are:

- `SNFollowers(string name,int initialFollowers)`. Constructor that creates a new object with the data passed by parameters, and with an initial money value (`money`) of 0. If the social network does not exist, the `EXCEPTION_UNKNOWN_SN` exception will be thrown; if the initial number of followers is less than 0, the standard `invalid_argument` exception will be thrown passing that number as a parameter (converted to string with the function `to_string`, since the parameter of that exception must be a string).

- `void addFollowers(int nFollowers)`. Adds to the number of followers the value passed by parameter (which could be negative if the influencer loses followers). After the addition, if the value of the `numFollowers` attribute is negative it should be set to 0 (the number of followers cannot be negative, it must be at least 0).

- `void addEvent(double rating)`. Method that recalculates the number of followers (and the monetization obtained) based on the event's rating (`rating`), which is passed as a parameter, and the average rating of the social network. If `rating` is negative, the `invalid_argument` exception should be thrown with `rating` (converted to string with `to_string`). The calculation is done as follows:

  - If the ratio between the event's rating and the network's is greater than 0.8, the number of followers will increase by an amount that is the value of that ratio multiplied by the current number of followers (taking the integer part of that operation). For example, if the event's rating is 0.65 and the social network's is 0.8, the ratio will be $0.65/0.8 = 0.8125$; if the number of followers was 1000, 812 followers will be added to the number of followers, and 812 multiplied by the average monetization of the network will be added to the amount of money earned.
  - If the ratio is less than or equal to 0.8, followers will be lost and no money will be earned. The number of followers lost will be the difference of 0.9 minus the ratio (e.g., if the ratio is 0.7, the subtraction would be 0.2) multiplied by the number of followers (taking the integer part). For example, if the ratio is 0.7 and there were 1000 followers, 200 followers will be lost (remember that the number of followers cannot be negative).

- `double collectCommission(double commission)`. Calculates the monetization corresponding to the agency, multiplying the commission passed as a parameter by the amount of money obtained up to the moment, and returns it. Before finishing, it sets the amount of money to 0 because the agency is going to charge this commission immediately, and it must reset this amount to 0 so that the next time money is collected, commission is not charged on the previous money for which it has already been charged (the influencers would notice). The value of `commission` must be greater than 0 and less than 1, otherwise the exception `invalid_argument` will be thrown (converting to string that value) without doing anything else, as in other methods.

- `string getName() const`, `int getNumFollowers() const`, `double getMoney() const`. *Getters* that return the values of the attributes.

- `ostream& operator<<(ostream &os, const SNFollowers &snf)`. Output operator that displays the data of the followers. For example, if the social network is "`TokTik`", has 1005 followers, and has made an amount of money of 157.5, it would show the following:

  ```
  [TokTik|1005|157.5]
  ```

  After the "`]`" no newline should be written (`\n` or `endl`).

3.4 Influencer

This class stores the basic data of the influencer (name and commission agreed with the agency), and a vector with the data of social network followers (`SNFollowers`).

The methods are:

- `Influencer(string name)`. Constructor that creates a new influencer with the name passed as a parameter, and with an initial commission of 0.1
- `void setCommission(double commission)`. Modifies the commission of the influencer, as long as it is greater than 0 and less than 0.8; if the value passed as a parameter is not correct, the exception `EXCEPTION_WRONG_COMMISSION` will be thrown
- `void addFollowers(string snName,int nFollowers)`: Adds followers to the influencer's social network called `snName`; if there was not a social network with that name, it tries to create a new one (an object of the class `SNFollowers`) and adds it to the end of the vector `followers` (if it is possible to create the object, of course). If the social network turns out to be unknown (the exception `EXCEPTION_UNKNOWN_SN` that is generated in the constructor of `SNFollowers`) must be caught, the error `ERR_UNKNOWN_SN` will be emitted (and logically the social network will not be added).
- `void addEvent(int nsns,string sn[],double rat[])`. Updates the data of the influencer's social networks with the data of an event, which are a vector of social network names (`sn`) and a vector of ratings (`rat`), along with the size of both vectors (`nsns`). The name vector may contain names of networks in which the influencer does not have presence, in which case nothing will be done; only the data of the social networks in which they are present will be updated.
- `double collectCommission()`. Calculates the agency's commission by summing the commissions of all the influencer's social networks (calling the homonymous method of `SNFollowers`), and returns it.
- `string getName() const`, `vector<Followers> getFollowers() const`, `double getCommission() const`. *Getters* that return the values of the attributes.
- `ostream& operator<<(ostream &os,const Influencer &inf)`. Output operator that displays the data of the influencer. For example, if the influencer's name is "`Lorelay`" and they have a commission of 0.24, and are present on social networks "`TokTik`" and "`YZ`", the displayed information would be:

  ```
  Influencer: Lorelay (0.24)
  [TokTik|1005|157.5][YZ|400|78.75]
  ```

  Finally, a newline should be written (`\n` or `endl`) even if there are no social networks.

3.5 Agency

This class stores the agency's data (name, money, and influencers) and allows managing them and reflecting the influence of events. The methods of this class are:

- `Agency(string name,double initialMoney)`. Constructor that initializes the name with the one passed as an argument, and the initial amount of money. It should be assumed that the arguments are correct; no verification is necessary.
- `Influencer *searchInfluencer(string infName)`. Searches in the vector for an influencer whose name matches the one passed by parameter, and returns a pointer to that influencer if found; if not found, it throws the exception `EXCEPTION_INFL_NOT_FOUND`
- `void addInfluencer(string infName,double commission)`. Adds a new influencer to the vector, if they did not already exist. If an influencer with that name already exists (detected by calling `searchInfluencer`), the error `ERR_DUPLICATED` should be issued and nothing else done. If the commission is incorrect, it will be detected in the `setCommission` method of `Influencer` and an exception will be thrown, which should be captured in this method and issue the error `ERR_WRONG_COMMISSION`. Obviously, if an error occurs the new influencer will not be added to the vector.
- `void addFollowers(string infName,string snName,int nFollowers)`: Adds followers to the influencer whose name matches `infName`; if not existing, it will be detected in `searchInfluencer` by throwing an exception, which should be captured in this method and issue the error `ERR_NOT_FOUND`

- `void newEvent(vector<string> infNames, int nsns, string snNames[],double evRats[])`. Updates the data of the agency's influencers who attended the event; if an influencer who does not belong to the agency appears (should use `searchInfluencer`), nothing should be done, just continue with the next influencer. The last three arguments are used to call the method `addEvent` of the influencers.
- `void deleteInfluencer(string infName)`. Deletes the influencer whose name is passed as a parameter, first obtaining the commission for the agency and adding it to the agency's money. If there is no influencer with that name, the error `ERR_NOT_FOUND` should be issued
- `double collectCommissions()`. Collects the commissions from all influencers, and adds them to the agency's money. Returns the collected amount.
- `string getName() const`, `double getMoney() const`. *Getters* that return the values of the attributes
- `ostream& operator<<(ostream &os,const Agency &ag)`. Output operator that shows the name and money of the agency, and the data of its influencers, as in this example:

```
Agency: InfluRepAgency [1225.87]
Influencer: Lorelay (0.24)
[TokTik|1005|157.5][YZ|400|78.75]
Influencer: Bdian (0.7)
[TokTik|105|88.55][Pupagram|9717|289.20]
```

4 Main program

The main program is in the file `prac3.cc` published on the Moodle page of the course. This file contains code to create several social networks and an agency, and to add influencers to the agency and followers to the influencers, and finally to manage an event and calculate the money generated by the event for the agency.

It is simply an example of how the classes you have created might be used in a program. You can modify it as you wish to test your classes; it should not be delivered with the rest of the files.deliver it, it's okay, it will be ignored).