TRAVEL AGENCY

DESCRIPTION OF THE PROJECT ("THE STORY")

General design

In this project I created a software in C++ representing the functionalism of a travel agency website. In this software there are three types of users which are clients, employees and superusers.
Users can login, logout and update their accounts.
Namely that, the SuperUser class inherit form Employee class. Employee and Client classes inherit from User class.

The super user is a default account. With this account it is possible to create or delete an employee account. So, the only way to manage an employee account is to do it with the super user account.

Employees users can fully interact with the Travel and hotel classes. They can create, update or delete a travel or a hotel. Hotels are contained in a travel. Employees can also publish or unpublish a travel but if they are following these conditions:

- A travel can be published if there is at least one hotel
- A travel cannot be unpublished if a client already booked it

Employees can likewise create and delete a client account (it can be related to the case where a client has some trouble to create/delete their account).
To finish with the Employee class: an employee cannot book a travel.

Clients users can fully interact with the Booking, Plane and Train classes. They can create, pay or delete a booking and create, update or delete planes and trains. Planes and trains are contained in a booking. Clients can also apply or unapplied a booking.
If a client wants to update a booking, which means modify a train or a plane, the booking should be unapplied.
If a client wants to pay a booking, the booking should be applied.
If a client modifies a booking already paid, and then pay again to actualize the booking, the client:

- can be refunded (if the new price is less than previous one)
- must give money (if the new price is more than the previous one | due = new price – previous price)
- we just see a message saying that all is ok (if the new price is equal to the previous price)

It means that for example if a client deletes a plane of 100€ paid by himself, the company will refund him about 100€.

The **TravelAgency** class is the main one. In this class there is some basics variables such as the travel agency name and there are some vectors in which all the data are stored:

- tab_superUser
- tab_employee
- tab_client
- tab_travel
- tab_booking
- tab_payment

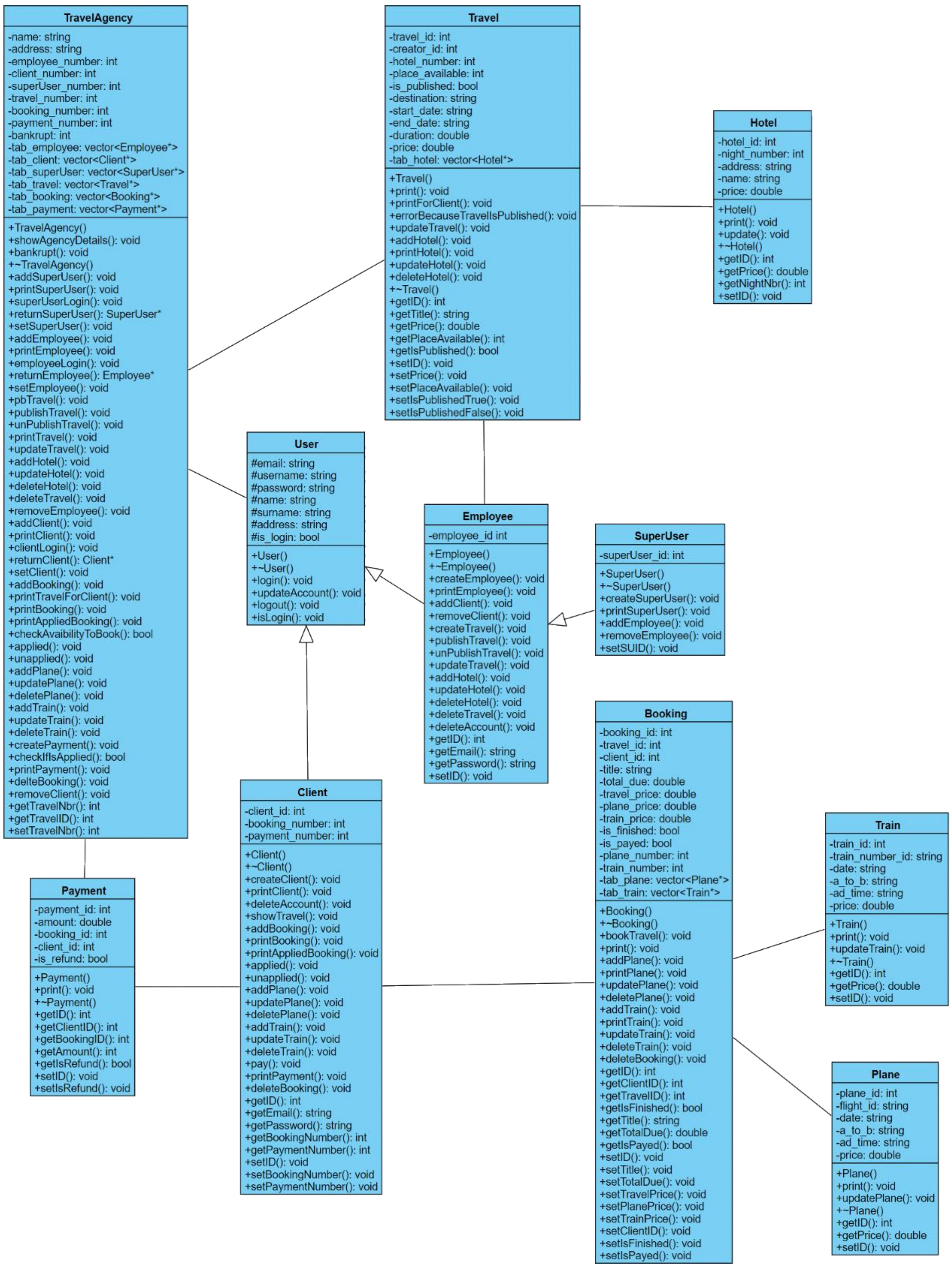
For all the other tab, ids are handled by the program. When a deletion occurs, the program put the ids equal to the line identification (example: tab_example[i].getID() will return i).

This is true for all the tab except for tab_booking where ids are handled according to the client. It means that for each client the booking_id of their first booking will be 0. When a booking is deleted, re-attribution of ids is done according to what was previously written but for each client.

Thanks to this, the program can find the correct client, employee, booking, travel or payment according to the launched function. All research are done with a "for" loop which finds the correct id.

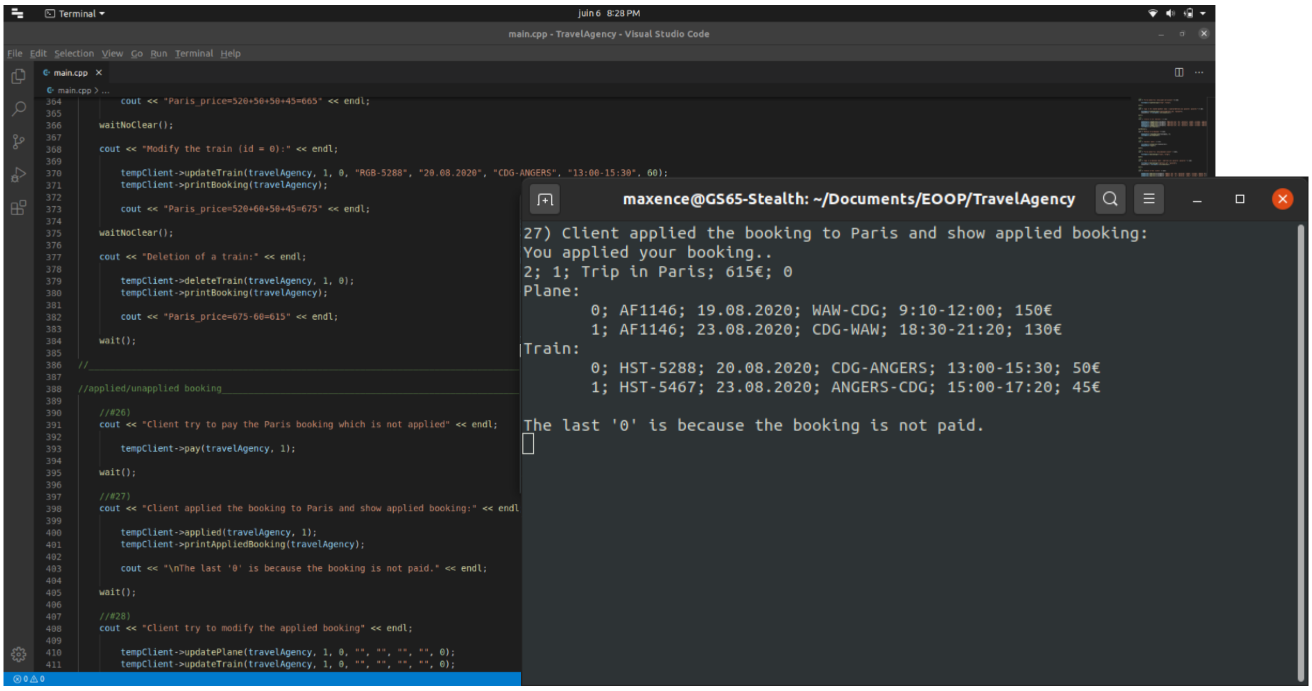
Travel and **Booking** classes, which are used respectively by employees and clients contain, as the **TravelAgency** class, some vectors. For the **Travel** class one of **Hotel** and for the **Booking** class, one of **Plane** and one of **Train**.
The three vectors have the same functionalism than the others in the **TravelAgency** class, which means that ids are handled by the program automatically and the research are done with some "for" loops.
Concerning the login and logout functions, I created one **tempUser** of each type in the main, so one super user, one employee and one client. Then when one user logs in, the user type is copied in the corresponding type **tempUser**. And when the login user logs out, the **tempUser** is copying to the corresponding user to save the modifications made.

CASE STUDY (MEMORY MAP)



TESTING

The program is made to show that the main functionalism is nicely done. So, basically there is no graphical interface except the output which displays the tests results. Finally, in the **main.cpp** all the functions are called to show and prove that they are all working properly. In the **main.cpp** I have delimited all the tests so that it is much easier to identify what is corresponding to the displayed result in the console in the **main.cpp**.


Terminal
juin 6 8:28 PM
main.cpp - TravelAgency - Visual Studio Code
File Edit Selection View Go Run Terminal Help
main.cpp
maxence@GS65-Stealth: ~/Documents/EOOP/TravelAgency

```cpp
364            cout << "Paris_price=520+50+50+45=665" << endl;
365
366        waitNoClear();
367
368        cout << "Modify the train (id = 0):" << endl;
369
370            tempClient->updateTrain(travelAgency, 1, 0, "RGB-5288", "20.08.2020", "CDG-ANGERS", "13:00-15:30", 60);
371            tempClient->printBooking(travelAgency);
372
373            cout << "Paris_price=520+60+50+45=675" << endl;
374
375        waitNoClear();
376
377        cout << "Deletion of a train:" << endl;
378
379            tempClient->deleteTrain(travelAgency, 1, 0);
380            tempClient->printBooking(travelAgency);
381
382            cout << "Paris_price=675-60=615" << endl;
383
384        wait();
385
386    //_______________________________________________
387
388    //applied/unapplied booking______________________
389
390        //#26)
391        cout << "Client try to pay the Paris booking which is not applied" << endl;
392
393            tempClient->pay(travelAgency, 1);
394
395        wait();
396
397        //#27)
398        cout << "Client applied the booking to Paris and show applied booking:" << endl
399
400            tempClient->applied(travelAgency, 1);
401            tempClient->printAppliedBooking(travelAgency);
402
403            cout << "\nThe last '0' is because the booking is not paid." << endl;
404
405        wait();
406
407        //#28)
408        cout << "Client try to modify the applied booking" << endl;
409
410            tempClient->updatePlane(travelAgency, 1, 0, "", "", "", "", 0);
411            tempClient->updateTrain(travelAgency, 1, 0, "", "", "", "", 0);
```

```
27) Client applied the booking to Paris and show applied booking:
You applied your booking..
2; 1; Trip in Paris; 615€; 0
Plane:
        0; AF1146; 19.08.2020; WAW-CDG; 9:10-12:00; 150€
        1; AF1146; 23.08.2020; CDG-WAW; 18:30-21:20; 130€
Train:
        0; HST-5288; 20.08.2020; CDG-ANGERS; 13:00-15:30; 50€
        1; HST-5467; 23.08.2020; ANGERS-CDG; 15:00-17:20; 45€

The last '0' is because the booking is not paid.
```

As you can see on the screenshot, there are two windows. The one in the background is the **main.cpp** and the one in the foreground is the output of the program.
Each test is presented as the one in the output console. The number of the first line of the output windows allows us the get a landmark in the **main.cpp** file. With this landmark it is much easier to find the code related to the output. Here the output indicates "**27**" which represent the line **397** in the **main.cpp**. Then the test is written to explain what will be tested, here it is "*Client applied the booking to Paris and show applied booking:*". Just after that the program "interesting" output is displayed. Sometime after the test there is some needed explanation to understand the functionalism much easily like here with the line "*That last '0' is because the booking is not paid*".

Moreover, as you can see with the lines **386** to **388**, there are separators all along the code to delimit all the blocs of tests which are all named by the tested functions.

In the main.cpp there is a total of **34** tests (*counter is going to 35 because the fist "page" displayed is a sum up of the program*).
The test **#2** is not in a bloc of test because it is just the test to create the travel agency.
Tests **#3** to **#17** is the bloc "**Login/Logout AddUser/UpdateUser/RemoveUser**".

- **#3** show that **superUserLogin** fail if the user does not have the correct combination of email/password.
- **#4** show that the **superUserLogin** works if the user had the right combination of email/password.

- **#5** shows that **addEmployee** works by creating four employees and shows that **removeEmployee** is working too.
- **#6** tries the **logout** function for the super user.
- **#7** tries the **employeeLogin** function with a wrong combination of email/password.
- **#8** tries the **employeeLogin** function with a correct combination of email/password.
- **#9** tries the **addClient** function by creating four clients and tries **removeClient** function by deleting one client.
- **#10** show that clients can also create their account.
- **#11** tries the **updateAccount** function for an employee.
- **#12** tries the **logout** function for an employee.
- **#13** tries the **clientLogin** function with a wrong combination of email/password.
- **#14** tries the **clientLogin** function with a correct combination of email/password.
- **#15** tries the **updateAccount** function for a client.
- **#16** tries the **logout** function for a client.
- **#17** tries the **deleteAccount** function for a client call by the client.

Test **#18** to **#22** is the bloc "**create/update/delete travel/hotel**".

- **#18** tries **createTravel** function by creating five travels, tries to modify one of the created travels and then delete one created travel with an employee account.
- **#19** tries **publishTravel** function without any hotel in the travel, return error message.
- **#20** tries **addHotel** function by adding four hotels in a travel and tries **deleteHotel** function.
- **#21** tries the **publishTravel** function with a travel containing some hotels.
- **#22** add one hotel in each created travel, publish each travel and display the prices. It shows that the calculations are performed correctly.

Tests **#23** to **#25** is the bloc "**create/update/delete booking add/update/delete plane/train**".

- **#23** tries **addBooking** function by booking four travels on a client account and then **deleteBooking** function is tried.
- **#24** tries **addPlane** function by adding three planes to a booking then **updatePlane** function and **deletePlane** function are tried.
- **#25** tries **addTrain** function by adding three trains to a booking then **updateTrain** function and **deletTrain** function are tried.

Test **#26** to **#33** is the bloc "**applied/unapplied booking**".

- **#26** tries **pay** function on an unapplied booking, return a message error.
- **#27** tries **applied** function and show applied booking. It shows that the booking is not paid.
- **#28** tries to modify an applied booking using **updatePlane** and **updateTrain** function. It returns an error message because the booking is applied and cannot be modified.
- **#29** tries **pay** function on an applied booking and tries **pay** function again. It shows that the booking is already paid. Then client unapplied the booking.
- **#30** tries **pay** function when the booking have been unapplied and reapplied without any modification. It shows that the client does not have anything to pay.

- **#31** tries **pay** function when the booking have been unapplied and reapplied with one plane less. It shows that the client will be refund about the plane price.
- **#32** tries **pay** function when the booking have been unapplied and reapplied with one plane modified (higher price). It shows that the client had to pay the differences between the new booking price and the old booking price.
- **#33** tries **deleteBooking** function.

Test **#34** is just to logout the client and the last test **#35** is calling the function bankrupt which delete the travel agency.

CONCLUSION

So, as you perhaps noticed, my **UML** diagram which was in the Preliminary Project have been modified a little bit. Classes Train and Plane are no longer link to Travel but to Booking which implies that the vectors **tab_plane** and **tab_train** are now in the **Booking** class. I made this change because all the customers are not leaving in the same city. It is impossible to only have booking of clients from the same start point. So, to simplify the code, I changed it so that clients should add their plane and train to the booking. If we imagine a part of the website which is linked to some partners company for transport, it makes sense. The client is reserving the plane or the train by the travel agency website.

I have also placed the vector **tab_booking** in the **TravelAgency** class. It helps me to use my functions much easier.

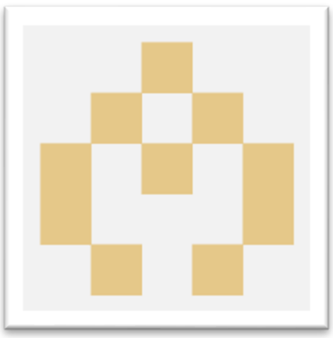
Some other differences with the preliminary project are about the number of functions. In fact, it is the first time that I am managing a project of this size. I learned how to code in **C++** during this project and I discovered that I needed a lot of getter and setter functions which are really useful function.

The creation of the **Makefile** takes me a lot of time because it was hard for me to understand the functionalism. But now I can create **Makefile** for my projects and it is convenient.

I am studying cyber security in my school in France (*ESAIP*). I am not about to work in programming it will just be a hobby. So, that is why I am proud of me about this project, even if all is not perfect.

Moreover, it was my first program in object-oriented programming. I understood my mistakes and how to create some better programs.

I always wanted to create some little games with *Unity,* but I was not able to do it. Now, thanks to this Erasmus, I am bringing back new knowledges that I can use to create some littles applications in **C#**, by using *Unity*, which is really close to **C++**. But before, for the challenge, I will try to do a basic graphical interface for my program to simulate the website properly. After our discussion I feel like if my project does not really finish. So, I will do it and modify all the things that you told me which are not that good.



You can find my code in my GitHub page by clicking the image above