Project Report

Is1220_Group 24: Thomas Cocher & Sarah Gross

Introduction

The two major difficulties we had making this project are technical issues and difficulties in choosing the design of the LMS.

The technical issues were long to solve but quite easy (they are detailed in “4) Results”) whereas the choosing difficulties where a bit harder and took us some time since we changed the way we would design some parts of our Library a few times, for instance to store the location of an object we first chose an Array, then we decided to store it in a Class Location which forced us to change our code in many other classes.

1) Presentation of the project

This project is about developing a Library Management System (LMS). We have to implement with OOP a fully-operational library, where 3 kinds of items are stored: books, CDs and DVDs. One can register to the library and then be able to consult onsite or borrow those items, with rules which depend on the status of their membership.
The first part of this project, which you are currently consulting, only deals with the design and the implementation of the classes which will be needed for this library “the library core”; the methods which enable the user to use the library will come in the second part.

2) Analysis and design

The user of the LMS ought to be able to use it without understanding how it works. Therefore, we decided to implement a Factory pattern for the manipulation of library items.

We also wondered about how to manage the update of the status of the loans and the fidelity cards. Since we needed to continuously make updates, we decided to use an observer pattern for the reservation and the WARNING systems. Updating the cards every time the program was launched seemed a better solution. Thus, we created a Launcher class which will implement all the actions we need when the Library is “opened” a new time
- Creating a new Factory so that the user can create new items whenever he wants
- Fetching from the database all the information about the Library
- Updating the status of a member (suspended/unsuspended); the fidelity cards and the loans.

We also realized that we needed a kind of database in order to keep all the information about our Library. In TPs, the examples were just to see if our code worked, we did not need

our examples to remain once we had closed the program. However here it is different we have one single Library on which the users will work during several months so it is compulsory for the information to be accessible all this time and we have to be able to close the program without losing it!
The information needs therefore to be stored somewhere. Since we cannot have a kind of MySQL-like database with Java, we decided to use the InputOutput streams to store the composition of the Library and its members in text files. We decided to store the information about each member, and the information about the Library. The problem was then how to translate the idea of an object into a text file. That's where we decided to use serialization, which is exactly the function made for this. First we thought about serializing the Library and the Members, but after coding the serialization of the Library we realised that since the information about the members were all stored in the library.listMembers it was not necessary to serialize the Members alone.

The reservation method needs the borrowing method in order to be implemented, therefore we will deal those two methods in the second part.

3) Implementation
   a) Code structure
      In order to see clearly what to do, we first drew on paper the UML diagram of the LMS we stated which classes we needed, what attributes that had, of what type. We chose to use some type which were not seen in class, for instance the ENUM type for the attributes which had only a couple of values possible (items bookCDDVD , fidelity card standard, frequent, gold) because this type was created for this kind of use.
      Thinking about the classes we needed made us reflecting about how they would be used in the project and thus raised the questions above.
      Then we could start to implement the project.

      We decided not to draw the papyrus UML diagram yet because if we did if would be incomplete since it would lack the methods we will implement in the second part. Therefore we will draw it for the second part.

      We were not sure if using a Factory was the right solution because the information to give at the creation depends on the item, for instance a book needs an ISVN number. Which means that the user has to know beforehand which information are needed for each item, which is in contradiction with the idea that the user does not know the details of the factory…
      That is why we decided to create a dialogue box using Scanner Objects where the user would declare to type of item he desires. Then the box asks him to fill in all the attributes needed one by one. If he enters the wrong type of data (for instance a String where he is asked to enter a number) he is asked to fill again as long as he enters the wrong data. This situation is managed by Exceptions of type InputMismatchException. While the user fills in these attributes, they are stored in a String ArrayList. Once he has filled all the

attributes, an Object of the Class of the item requested is created from the information stored in the ArrayList, then he is placed in the library using the placement algorithms (any fit, best shelf, …).

Concerning the updates of the suspension of a member, of the fidelity cards and the loans, we first implemented three functions for each kind of update, but since for each update we needed to make a loop on the list of the members, for speed considerations we decided to put all those updates in one same "check" method inside the Library class, hence reducing the loops needed to one.

b) Testing

We hesitated about first making the tests we needed, then programming the classes which would make those tests work, as in the Test Driven Development implementation seen in class. The advantage is that we can see how much work is left very easily that way. Nevertheless since the methods were supposed to be programmed on the second part we could not use this strategy.

We still made the tests for the few methods which were created for the first part. Namely, we made Junit tests for the placement algorithms (any fit, best shelf, best bookcase, best room).

Since we had implemented the creation of the objects with the factory, it is possible to create an object by executing the main function in the Launcher class. For this, you only have to uncomment the code in this class. We tried to manage the case where the user presses enter without entering data, but our researches we unsuccessful.
Each part of the dialogue box (filling in the Title, then filling in the Producer, …) was carefully tested in the launcher class independently. Then we tested the whole creation of an object and got errors, detailed in the next part "Results".

The serialization-unserialization of a library was tested as well in the Laucher class and worked successfully. A Junit test will be implemented in the second part.

We did not test the check method (updates of the loans and the status which will be executed in the Launcher) since it needed things which will be implemented in the second part.

4) Results

Concerning the placement algorithms, the "any fit" and "best shelf any bookcase fit" work successfully, the "best bookcase fit" and "best room fit" produce errors. That was to be expected since the second two algorithms are way more complicated (you can see many comments in our code in order to make them not too hard to understand). We did not have time to fix these errors; they will be fixed in the second part.

Concerning the createItem method in the ItemFactory class, these errors were all due to Scanners. First, if we call successively two times a Scanner with for instance scanner.nextLine(), the second time we do not have the same result as the first time. To fix this, the result of the scanner.nextLine() must be stored in a local variable which will be used two times. Second, if we close a Scanner, the next time we call new Scanner(System.in) it does not work ! That is why none of our Scanners were closed in our method createItem inside our ItemFactory class.

Conclusion

We have enjoyed working on the LMS but lately realized during this first part that we may have spent too much time working on elements of the second part.

The serialization, check and item factory systems should be done during the second part of the project and we have done it in the first one which left us less time to work on the first part.

That is why the last two algorithms of storing are imcomplete.