**Project Report**

**IS1220\_Group 24: Thomas Cocher & Sarah Gross**

**Part 2**

*Introduction*

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. The used of this LMS would be a Library employee trying to update what happens in his library : arrival of a new item, member changing fidelity status, item borrowed, new room to the library…

This project was implemented in two parts. The first one only deals with the design and the implementation of the classes which will be needed for this library “the library core”.

The second part of this project, which you are currently consulting, consists in providing to the user a user-friendly interface in order to enable him to make easily operations on the library; for instance create new items, borrow them, add a new member, have the list of the items in a room of the library…

This user-friendly interface was created with two different tools :

* The first one is the CLUI : Command-Line User Interface. This interface is located in the package LMS and is implemented in the class Launcher. With this CLUI, the user interacts with a console which guides him through the creation and the management of a library.
* The second one is a GUI : Graphic User Interface. This interface is located in the package GUI and is implemented in the class GraphicalInterface. This GUI is like any executable you could download on the internet, it is a basic window. As the user make complicated modifications to the library, new windows pop open where he has to select or fill in some data.

As for the repartition of our work:

* Thomas made the CLUI, the UML diagram and the test asked in section 2.4
* Sarah made the GUI and this project report.
* The implementation of the methods in the LibraryFactory was designed by Sarah and implemented by both of us. The details of who made which methods are written in the section 3)b): Testing.

1. Analysis and design

We made a few changes about what we had decided in the first part.

We decided to unite all the commands the user would have access to in the CLUI in one single factory : the class LibraryFactory. Therefore, the factory ItemFactory created in the first part was completely deleted.

The LibraryFactory was thought thoroughly in order that it can easily be used for the CLUI, the Junit tests and the GUI.

* We could not use again the code we had implemented for the ItemFactory in the LibraryFactory because this code used Scanners. However, it is not possible to make Junit tests on methods using scanners. That is why we chose that all the arguments we needed must be filled before we call a method from the LibraryFactory class.

In order to get those arguments, Scanners are implemented in the CLUI with the class Launcher. In the GUI, those arguments are fetched in different manners, it can be text fields for the user to fill, or the user has to select a value in a list…

* We wanted our library to prevent the user from entering false information. That is why every data he enters is tested, our application checks if this data respects the right format (a positive integer if it is a number of shelves, a format dd/mm/yyyy if it is a date…) and sends an error message if the user does not enter a valid type of data.

These error messages are displayed in a different way for the GUI and the CLUI. The CLUI uses a System.out.println, whereas the GUI displays an error window. Therefore, we had to use a system in the LibraryFactory which would send error messages that could be perceived by both interfaces. Hence our use of exceptions. We created only one exception, named AlreadyExistsException, in case we tried to add an element which already existed. We used it in three cases: in case we try to add a member witch exactly add the same characteristics as another member, in case we try to add a room with a name already used in the library, in case we try to add a bookcase to a room where a bookcase with this name already exists. Since the instructions required us to be able to make a list of all the items in a room with its name, and the list of all the items in a bookcase of a room with this bookcase name, we deducted that a room is uniquely identified by its name, and the name for a bookcase within a room.

Since we had already created many classes, we decided for the other exceptions to use exceptions which already existed in JRE and which names corresponded well with their utilization. For instance, we throw an IllegalArgumentException if the user enters a data with a format which was not expected, for instance (dd/mm/yyyy) or (borrowing/online consultation) or (16 integers in a row for a credit card number); we throw an IllegalStateException if a member tries to borrow a book whereas he is suspended.

For the GUI, the package only uses 3 classes, each class corresponding to a frame. We thought about making more classes, which would extend Panels and not Frame in order to have a code less heavy in the class LibraryModifyer (which contains all the methods to make a modification on the library). It was actually not an easy manipulation to do (we cannot make popup error windows from a panel, only from a frame) so we kept it that way.

1. Implementation
2. Code structure

To see the structure of our code, please consult our UML diagram to see the interactions between the classes.

You can see in the UML a class which is only used by the User Interfaced, call Serialization.

This class was made to be able to save the work on the library. The method saveLibrary saves the library the user is currently working on in a directory called ”savedLibraries”, whose path is the same as the application path. Since we consider that a Library is uniquely identified by its name, if someone tries to save a library with the same name as an already saved library, it raises an AlreadyExistsException the application asks if you want to change your library name or if you want to save your library over the older one.

Then you can, right after having opened the application, chose to work again on your library and you enter the name of the library you want to fetch. The interface calls the method fetchLibrary(libraryName) from the Serialization class and you can again make some modifications on it.

Our storing strategies were quite not functioning well when the first part was delivered.

We inserted new arguments to the classes Shelf, Bookcase and Room called freeSpace.

* For a shelf, this space is the length of the shelf which is not already filled by items, in other words its remaining space.
* For a bookcase, this space is the sum of the freeSpaces of its shelfs.
* For a room, this space is the sum of the freeSpaces of its bookcases.

These new arguments were very useful insofar as they made our code lighter, and diminished a lot the number of loops made by our method and so the time for our method to run. We also deleted the method which gave all the potential locations of a item, because it added a useless loop to our storing strategies.

We also implemented the open/close principle for these algorithms by making them implement one same Interface called StoringStrategy.

Finally, we changed a bit the arguments of some classes. Namely we had forgotten the author attribute of LibraryItem, and we decided that it was not relevant to have the borrowing deadline as an attribute of a LibraryItem. We had not noticed that the number of months to lose the frequent membership was not the same as to obtain it, so we added the argument M’ to the class Library.

In order to respect the decisions detailed in the paragraph above “Analysis and Design”, we also decided to add arguments to the methods which would be used in the User Interface, regardless of what was written in the instructions sent by Mr. Paolini.

1. Testing

For the methods asked in the second part, the implementation of the JUnits was made along with the creation of the methods by using the TDD approach : we first tried to validate the JUnit test of a method before implementing the next method.

One person making a method therefore also coded the Junit test associated.

Here is our repartition of the work for the methods+tests:

Thomas : add\_room, add\_bookcase, add\_item, borrow\_item, check\_borrowed

Sarah: create\_library, store\_items, unstore\_items, list\_items, list\_room, list\_bookcase, find\_items, search\_title, add\_member.

The tests were the best way to notice the errors in our code in LibraryFactory.

* With the Junit, the error messages in the Junit window are often quite clear and point out what misuse was made;
* For the methods which return a String list of information, the code of these methods was first copy/pasted in the launcher to see if it really returned a list. Since some of them did not return results, asking to print a message every time the method entered in a loop was a good way to notice which loop was not functioning well. When the methods eventually printed the expected result, the Junit was implemented.
* Finally, testing directly with the CLUI or the GUI was also a good was to test errors. For instance, when entering a credit card number in the GUI, the application displayed an error window saying that we were not entering an integer number. That is how we realized that a credit card number with 16 numbers could not possibly be stored in a int since the biggest integer is around 2.000.000.000 ! Therefore we decided to store the credit card number as a string, nonetheless the application still checks that it is a number with 16 characters.

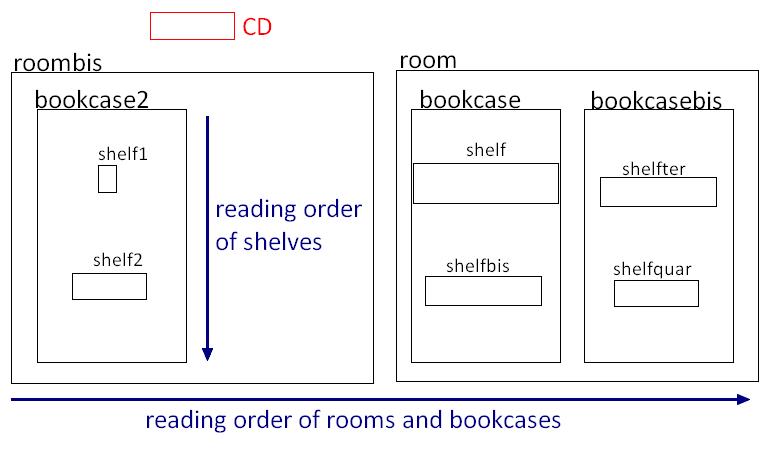
Description of every Junit test :

1. Storing Strategy tests:

Since we made corrections to our storing strategies, the Junit tests which did not work well in the first part now work fine.

For every of thoses tests, we first had to create a library with many elements in order to show that it was not random that our item ended in the right location, but because of our algorithm.

Here is the disposition of the room implemented in all of our storing strategy algorithms:



1. AnyFitTest:

The CD is too big to enter the shelf1 bus enters fine in shelf2. The biggest shelf is shelf, however with AnyFit we enter in any shelf big enough, so it is shelf2 which ends up hosting the CD: *assertTrue*(shelf2.getListItems().contains(CD));

1. BestRoomTest:

The room with the most available space is “room” so the CD ends up in any shelf of the room “room”:

*assertTrue*(shelf.getListItems().contains(CD)||shelfter.getListItems().contains(CD)

||shelfquar.getListItems().contains(CD)||shelfbis.getListItems().contains(CD));

1. BestBookcaseTest:

The bookcase with the most available space is “bookcase” so the CD ends up in any shelf of the bookcase “bookcase”:

*assertTrue*(shelf.getListItems().contains(CD)||shelfbis.getListItems().contains(CD));

1. BestShelfTest:

The shelf with the most available space of the Library is shelf so the CD ends up in shelf: *assertTrue*(shelf.getListItems().contains(CD));

1. LibraryFactoryTest:
2. Create\_factory

This test only uses the constructor and is very small. However it did not work at the beginning so we were really confused since it had only 3 lines !

We thought the problem might be because of the function equals. We tried the Junit test:

Library lib = new Library("coucou",2,1,5,6);

Library libTest = new Library("coucou",2,1,5,6);

assertTrue(lib.equals(libTest));

This test also triggered a false response from Junit. This confirmed that the problem was equals. Therefore, Sarah generated the HashCode() and equals() overwriting in the Classes Library, Book, CD, DVD, Bookcase, Room, Shelf, Cuboid, Location.

1. Add\_Room

There was also a failure of this test at the beginning. We added a room, created a new room like this one and checked if the list of rooms of the library contained this new room. Maybe it failed because the function contains() for an ArrayList uses the equals method and we had not overridden yet the equals method. To overcome this problem, Thomas used an “if” loop:

**try** {

Test.add\_room(library,"room",10,10,10);

} **catch** (AlreadyExistsException e) {

}

**boolean** bool = **false**;

**for** (Room room : library.getListRooms()){

**if**(room.getRoomName().equals("room")&&room.getHeight()==10&&room.getLength()==10&&room.getWidth()==10) {

**…**

1. Add\_bookcase

Same as add\_room. Since some exceptions might be raised, we do nothing if they occur.

1. Add\_item

We add an item to a newly created library, we check that the storage room is not empty, since it was at the beginning it means that it is our item which is inside.

1. Store\_items

We create a room in the same disposition as in 1.Storing Strategies and then add the items, however somehow this test fails without us being able to explain why.

1. Unstore\_items

Idem, code with very few lines, we store a few items in some shelfs, call unstore but the storage room does not seem to contain those items, it fails ..

1. List\_items

This test and the four following below are a real mystery. As explained at the beginning of b)Testing, these five methods have been first tested directly in the launcher to see what result it printed. It worked perfectly, so we implemented the JUnits. At the beginning they failed because somewhere there was a \n lacking, but after that the JUnits were successful. The day after, they did not work even though we did not change a thing…

One can directly see if they work by using the CLUI or the GUI : when one call these methods, the right result is displayed. The Junit must fail because of a missing space or something, we did not have time to implement then another time.

1. List\_room
2. List\_bookcase
3. Find\_items
4. Search\_title
5. Add\_member

Did not have time to implement it.

1. Borrow\_item

Did not have time to implement it.

1. Check\_borrowed

Did not have time to implement it.

1. Results

*Conclusion*