Library Loan System Design Brief

Scope

Design and implement a prototype loan system for a local library, which they can operate manually. The library contains various items for loan: books, magazines, DVDs and CDs. These items have attributes: all have a title and a library ID number. Books and CDs have an author/artist, whereas magazines and DVDs instead have a publisher. All items are kinds of 'loanable item', which have two further attributes: a flag to indicate if the item is on loan, and a target return date.

The library needs to be able to see its collection printed out as a report. The report must show the library id, the item's type (i.e. if it is a book, magazine, DVD or CD), the item's title, if it is on loan and, if so, what its return date is. There should also be a summary line, showing the total number of items and how many are out on loan.

[Uni of Herts Assignment Briefing Sheet]

Goals

Functioning prototype with the ability to generate two reports:

1) Report one should be in the form of console output, using System.out.print(ln).
2) Report two should be saved to a text file in the root of the project.

Requirements

Parent class (common properties)
Four child classes (CD, DVD, Magazine, Book)
Tester Class (creation of objects and outputting of reports)

Not Required

Reading the specification there are no mentions of users requiring the ability to obtain information from individual objects. [1]

Requirements

Parent Class

All objects are to be a 'loanable item' – from the scope this defines at *least* two common properties: loan status and return date. Further to this, all properties will have an ID number and title.

Using this information, I will create the fields for the properties of each object and store them in this class. Each should have a private access modifier, as nothing should read or write directly to or from these fields. Get and Set methods will be created to provide the functionality to do this for all properties.

A way of enabling reporting information will be needed to produce the report output. I suspect this will be a public method in this class that will be called in the test class to produce a report.

Child Classes

Two of the child classes (DVD/Magazine) share the *publisher* property. The other two do not share this and will each have their own: CD has the *artist* property, whereas Book has *author*. Due to the prototype not being a complex programme these properties will be defined explicitly in their class. They could be defined in the parent class but then this is would qualify as over-exposure as they are not needed by all classes.

Tester/Report Class

The scope states the collection of objects is defined in the tester class, so iteration through the collection is going to be performed here. The tester/reports class needed a way of accessing the information in the other classes to output the reports.

Post Programming Analysis

Parent Class I

I explored the option of interfaces initially but decided against this for two reasons. The first is that I thought it might be slightly overkill for the complexity requirements of the assignment. The second is that I believe I can achieve a programme that runs as efficiently as one with interfaces; again, only due to the complexity of the application.

I started by declaring the member fields I knew for definite would be needed across all the child classes, ID, name, loanStatus, and returnDate.

I then started to create 'Getter' methods that would retrieve these fields, with the idea in mind that the report could call these methods without exposing the inner workings. These methods were public and simply returned the relevant field. (For example, GetName would return the name field)

After the 'Getter' methods had been created I started to look at ways I could create the objects and pass all the properties into them at run time. Constructors work for this exact purpose so with this in mind I created all of the 'Setters' I thought I would need, such as SetName and SetID. These were created public and void as I knew they would not need to return (GET) anything. Each had a single parameter, an 'in' version of what the setter was for. (public void SetName(String inName) as an example). This led me onto my child classes when I was finished.

Child Classes I

All child classes inherited from the parent. This was a clear requirement from the beginning.
Straight away, after creating my setters for the parent class, I knew that the constructors needed a way to parse this information and generate the properties for my objects. Firstly, I created the private member field for the parameters that were unique to the classes (artist, author, publisher). Secondly, in the constructor I started writing, I called all the setters I had created in the parent class and defined input parameters the constructor would parse to them. An example of one of my constructors is:

```
public DVD (int dvdID, String dvdName, String dvdPublisher, String dvdLoanStatus, String
dvdReturnDate) {
        SetID(dvdID);
        SetName(dvdName);
        publisher = dvdPublisher;
        SetLoanStatus(dvdLoanStatus);
        SetReturnDate(dvdReturnDate);
    }
```

I then set about creating the ‘Getters’ for the unique fields in each class, again to reduce the exposure to other classes.

Testing I

I was now in a position where I could start to create a small array of items (LoanableItem) and utilise a for loop and System.out.println() to test both the access modifiers, and whether the Setters/Getters were working as I had planned. Success!

Reports

After initial testing I was at the point where I could see the project starting to take shape. I created another class, Reports. I envisioned this being the class where the methods for printing and saving would happen so I figured that the object collection should reside here. In the constructor I generated a new type of array (again LoanableItem – the type of the parent class) and began to populate a static array of 20 items as required. Five of each type made sense.

After a lot of trial and error trying to create the entirety of the report in the reports class, I decided that it was better to create a single method in the parent class that can output the information I need.

Parent Class II

After all, the parent class is the type for the collection and natively has access to all the required properties needed. (Publisher, artist, and author are not required).

This, for me was the correct call. There may be a more efficient way but as soon as I started writing the code, I knew I could achieve what I wanted. Adapt and overcome.

Because I still had the plan for the method to be called in a for loop from the Reports class, I knew that whatever was in this method would be run every time, under the object that was being iterated at the time. After realising this I simple wrote the method to return the string that was going to be appended to a text file *and* output to the console. The string, which would run under the current iteration called upon all of the ‘getters’ I had previously created in the parent class.

```
String "Item-" + GetID() + ", " + GetObjectType() + ", "  + GetName() + ", " +
GetLoanStatus() + GetReturnDate();
```

Testing II

I tested the newly written method against each iteration using a for loop in the reports class. The output was not as desired. ReturnDate was being added every time even if one was not added, but the final ‘,’ was still being added. I added an if statement to compare if the loan status was equal to “No”, and if it was then the string returned would be absent of the GetReturnDate() call.

Initially this did not work and took me some time to figure out (the GetReturnDate method was still being called even though the loan status was equal to “No”. Eventually I realised the problem was because I had done if(x = y) instead of if(x == y). Stupid mistake resolved, time to move on.

Reports II
Now that the output was working as expected I just had to fine tune the methods that outputs to the console and then create the method to save to a file.

Both methods were simple enough to create and utilised System.out.println() for the console, and the PrintWriter method covered in the course material for the file saving.

The final step was, for both methods, to generate a summary to meet the requirements in the assignment. I explored multiple ways of doing this but failed, so I eventually decided to create another method with a field in it for counting items on loan. The for loop iterates through the collection and increments the 'itemsOnLoan' field by 1 if the GetLoanStatus method in the parent class is equal to "Yes". I then added a line into each report method (Print and Save) to append the summary to the end of the console or file, depending on the method.

Testing III
With the feeling that I had hit all the criteria and that everything was complete I performed another round of testing. Success again, much to my amazement.

Tidying Up and Optimization
After the third round of testing, the programme was complete to my initial design plan. I decided to go through all the code base and omit anything that was redundant. I noticed a few things I was not happy with.

Firstly, after reviewing the assignment scope again I realised that there was no requirement for users to be able to view the properties of each object individually. I decided to remove the 'getters' that were unique to each of the child classes.

Secondly, I was not happy with the access modifiers, as they were all public and needn't be. Again, as the specification required only a report, I realised that the setters could be protected instead of public. There is more harm than good can come from having methods exposed to more classes than needed. I knew that with the protected method, the child classes would still be able to call on the methods in the constructor to set the properties of the objects. I then extended this line of thought throughout the source code and set all methods to the tightest they could be in terms of security, without breaking the programme.

After a zoom conference with Peter to clarify a few things I had concerns about, and by going through the process of operations line by line, I realised that the programme in its current state would have to call all of the getters in the parent class every time the GenerateReport method was being run, as this was the way the report information collected the names, ID's, etc for each object. I removed the method calls and replaced them with the member field I knew would be used in each iteration. To cement this, I added the 'this' keyword to help distinguish the member fields of the object to the fields of the parent class.

With doing this I realised further that the majority of the 'getters' were now redundant. I removed them from the source code. I went through the source code again and replaced all method calls with fields where I thought the code base would remain functional. This was to increase efficiency.

Testing IV
The final round of testing brought everything together and the programme worked exactly how I wanted it to.

Finishing touches
I documented the source code using JavaDoc throughout and then packaged the programme in BlueJ.