Overview

In this second CS1003 practical, I was asked to write a Java program to interact with the search API of DBLP. My program had to take different keywords as inputs: a specific query, together with a choice of author, publication, or venue, and a cache directory to store the previous requests. Finally, I had to run different tests, some mandatory through Stacscheck, others optional, to check that my program was fully functional.

I'm proud to say that I've achieved the previously established goals:

- The code can read the command line arguments
- The program can call the API successfully and retrieve the necessary information
- All tests are successful (see **Testing** for more details)

Design

The project is made up of two Java programs, CS1003P2.java and test.java. The first file contains the code to make successful API calls, and the second file contains additional tests to make sure my code in the first file works. I will now describe the CS1003P2.java file, the test.java file shall be explained in the **Testing** part of this report.

I used two variables throughout this project, that I created at the beginning of the class: a HashMap<String, String> to store the arguments' keywords and their corresponding values, and a String to store the URL. The URL's structure is identical for every API call, except for two elements: the search specification (represented by an "@") and the user query (represented by the string "test").

When the program is run, the first thing the code does is to check for valid command line arguments, using the checkValidArguments method (returns true if the command line arguments are valid, or false if they are not, followed by an error message). This method is made of 6 different checks.
The first step is to check for missing command line keywords: --search, --query, and --cache. To do so, I implemented a checkMissingArguments method that checks the command line arguments for those three keywords, and returns true if all three are in the command line arguments. If not, then the method prints out an error message, and returns false. If the checkMissingArguments method returns false, then the checkValidArguments method also returns false.
After checking for missing keywords, the method cycles through every command line argument. It then checks if this argument is the --search keyword. If that's the case, then it first checks if there is a value to the --search keyword. If that's not the case, then the algorithm prints out an error message and returns false. If there is a value, then the program checks the validity of this value, using the checkValidSearch method: the valid values for the --search keyword are "venue", "author" and "publication".
We then check for the cache validity, using the checkValidCache method: if the inputted cache does not exist, or is not a directory, then the method prints out an error message, and returns false. Note that this check is nested in the previous check: this is for the check priority. According to the Stacscheck tests, if both the --search and --cache keywords are invalid (regardless of their position in the command line), then the appropriate error message that should be returned is for the --search. Therefore we have to make sure that the --search keyword is valid before checking for the cache validity.

The fourth check is for the --query keyword. It first checks for a missing value for this keyword, and then checks for the value's validity, again using a user-created method called checkValidQuery. The only invalid queries are --search, --query and --cache, which are command line keywords.
The fifth test is to check for a missing value to the --cache keyword: if the user did not input a cache directory, then the program prints out an error message and returns false. This check is the final check within the looping through the command line arguments.
The sixth and final check is to check the number of command line arguments: if there are more than 6 command line arguments, then an appropriate error message is printed out, and the method returns false. If the command line arguments pass all 6 of those tests, then the checkValidArguments method returns true.

After checking for the validity of the command line arguments, I stored the keywords and their values in a HashMap called *arguments*, using the getArguments method. I also updated the URL, replacing the "@" with the search specification, and "test" with the user's query, thanks to the getURL method.

I then checked the value for the --search keyword, and ran the corresponding method (for example, if the value is "venue", then the program runs the getVenue method). The structure of these methods is quite similar, especially between the getVenue and getPublication. Only getAuthor is slightly different.
All three methods begin with creating a Document object called *doc* (*doc1* for the getAuthor method), using the DOM libraries (DocumentBuilderFactory → DocumentBuilder → Document). I also create a File object called *savedResponse* that contains a previously saved query using the getSavedResponse method. The getSavedResponse method searches the cache for a previous request, encrypted using the getEncodedURL method, and returns the file if it found it, or null if not. The next step is to check if *savedResponse* is null or not: if it is null, which means that the user did not make a previous API call identical to this one, then we update the *doc* variable to parse the URL, and we save this URL to the cache file using the saveResponse method. If *savedResponse* is not null, then *doc* parses the *savedResponse* file. By doing so, we do not repeat API calls.
We then search the xml file for a node called "info": in all three types of xml files (files for authors, venues or publications), all - or most - of the necessary information is in this node. We cycle through every element of this node, and print out the information we need:

- For the getVenue method, we print the content of the "venue" child node, which contains the name of the venue.
- For the getPublication method, we print the content of the "title" child node, which contains the title of the publication, and the number of "author" child nodes

The getAuthor method is a bit more complicated than the previous two. It firstly checks the number of nodes with the name "info". If there are none, then the author does not exist, or never wrote anything, and the method prints out the author's name, followed by 0 publications and 0 co-authors. If there are nodes with the name "info", then we print out the name of the author (found in the child node "author"). To get the number of publications and co-authors, we have to make another API call with the URL found in the "url" child node. Again, we must create a new Document object called *doc2* and check for a saved response to avoid making repeated API calls. From that second Document, we then print the number of nodes with names "r" (for the number of publications) and "co" (for the number of co-authors).

Testing

We can divide the testing of my program in two categories:

1. Mandatory Stacscheck tests
2. Optional tests that I implemented

To run the Stacscheck tests, I copied the cache directory from Studres into my P2 directory. I then ran the stackscheck command, and obtained the following output:

```
Testing CS1003 Practical 2 - DBLP
- Looking for submission in a directory called 'src': found in current directory
* BUILD TEST - public/build : pass
* TEST - public/_malformed/1/test : pass
* TEST - public/_malformed/2/test : pass
* TEST - public/_malformed/3/test : pass
* TEST - public/_malformed/4/test : pass
* TEST - public/_malformed/5/test : pass
* TEST - public/author/jackcole/test : pass
* TEST - public/author/johnrsmith/test : pass
* TEST - public/author/johnsmith/test : pass
* TEST - public/publication/database/test : pass
* TEST - public/publication/databaset/test : pass
* TEST - public/publication/mariadb/test : pass
* TEST - public/publication/semi-structured/test : pass
* TEST - public/venue/distributeddb/test : pass
* TEST - public/venue/logic/test : pass
* TEST - public/venue/math/test : pass
* TEST - public/venue/parallelmath/test : pass
17 out of 17 tests passed
```

To implement additional tests, I created a separate Java file called test.java. This file contains one main method. Within that method, I created an ArrayList of String arrays that contains the different tests. Every String array represents the command line arguments inputted by the user. To run these tests, I cycled through the entries of the ArrayList, and called the main method of the CS1003P2 class with the current entry as parameter. Note that the description of the tests that are ran here can be found in the comments in the test.java file. The output is as follows:

```
TESTING

Invalid value for --query: --cache
Malformed command line arguments.

Too many command line arguments

Jack Cole - 7 publications with 27 co-authors.
D. Jackson Coleman - 1 publications with 5 co-authors.
G. Cole Jackson - 1 publications with 4 co-authors.

Cache directory doesn't exist: file

antoine megarbane - 0 publications with 0 co-authors.
```

Missing keyword --query
Malformed command line arguments.

PS: Before running the test.java file, one has to navigate to the *src* directory, and create an empty cache directory called *emptyCache* and a new file called *file*. Otherwise, the third test will return Cache directory doesn't exist: emptyCache and not the expected output.

Evaluation

I am proud to say that my program works: it can call the search API of DBLP and retrieve some information regarding the user's query. The code also checks the cache directory before making any call to the API and can successfully retrieve a saved response to prevent the program of performing repeated API calls. Finally, the output is in accordance with the System's Specifications, as the CS1003P2.java file passes the 17 Stacscheck tests.

Conclusion

All in all, I have found this assignment a bit challenging at first, as I didn't know what to do. Despite the help offered during the tutorials, I was still struggling with the DOM libraries and reading XML files. But by re-watching the Lectures and re-doing the Tutorial questions and Exercises, I solidified my knowledge of DOM and XML files, and found a way to complete the second CS1003 practical.