UNIVERSITY OF NEW BRUNSWICK

SWE4403

Software Architecture and Design Patterns

Name: Cooper Dickson

Student ID: 3684104

Assignment Title: Lab 3 Report

Date Due: February 14, 2023

Date Submitted: February 8, 2023

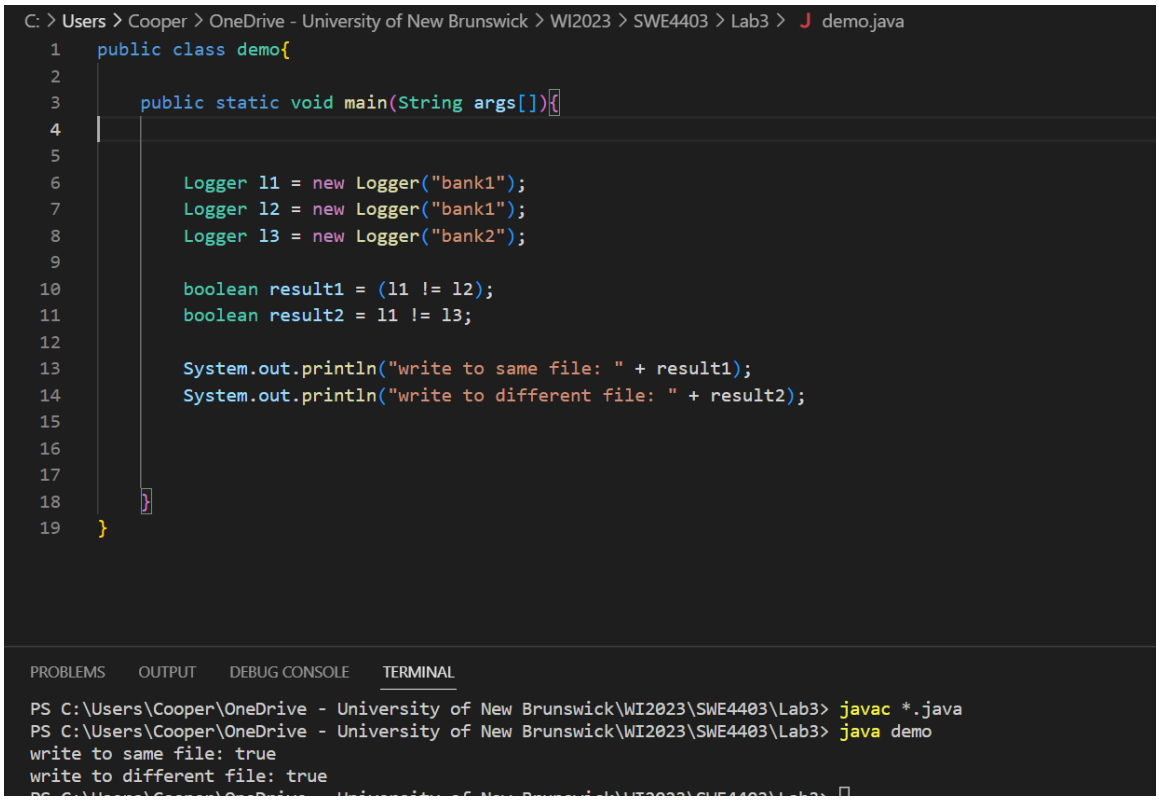
I warrant that this is my own individual work, except for portions that are clearly cited as

Signature: Cooper Dickson

1.a) Write a Demo class that implements the main() method and test if:

i. You can create multiple loggers writing to the same log file concurrently.

ii. You can create multiple loggers writing to different log files.



b.) Use the singleton pattern to ensure only one single logger is instantiated for a given file.

i. What type of initialization (early or lazy) you used? Why?

The type of instantiation in my solution is the lazy instantiation. This is because the instance of the Logger will only be created at the request time and checks if it has already been created then creates a new Logger, instead of creating a new instance when the class is loaded. This way the Logger object will not be created until it is actually needed by the main program so no extra memory is used in the case the program does not use a Logger.

C: > Users > Cooper > OneDrive - University of New Brunswick > WI2023 > SWE4403 > Lab3 > Logger.java

```java
import java.util.*;

public class Logger {

    private String fileName;
    private static Logger instance;
    private static Map<String, Logger> logMap = new HashMap<String, Logger>();

    public Logger(String fileName) {
        this.fileName = fileName;
    }

    public static synchronized Logger getInstance(String fileName){

        if(!(logMap.containsKey(fileName))){

            instance = new Logger(fileName);
            logMap.put(fileName, instance);
        }

        return logMap.get(fileName);
    }

    public void write(String message) {
        System.out.println("User made a " + message + " from " + fileName);
    }


}
```

ii. Upload your solution to D2L or provide a link to a git repo containing it.

C: > Users > Cooper > OneDrive - University of New Brunswick > WI2023 > SWE4403 > Lab3 > demo.java

```java
public class demo{

    public static void main(String args[]){

        //PART 2
        Logger l1 = Logger.getInstance("bank");
        Logger l2 = Logger.getInstance("bank");
        Logger l3 = Logger.getInstance("bank1");

        boolean result1 = (l1 == l3);
        System.out.println("Write to diffrent files: " + result1);

        boolean result2 = (l1 == l2);
        System.out.println("write to same file: " + result2);



    }
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\Users\Cooper\OneDrive - University of New Brunswick\WI2023\SWE4403\Lab3> java demo
Write to diffrent files: false
write to same file: true
PS C:\Users\Cooper\OneDrive - University of New Brunswick\WI2023\SWE4403\Lab3>
```

c. Write a Demo class to show how the logger can be used in a bank application to register all withdraws, deposits and transfers in a single file.

C: > Users > Cooper > OneDrive - University of New Brunswick > WI2023 > SWE4403 > Lab3 > J demo.java

```java
public class demo{

    public static void main(String args[]){

        Logger log1 = Logger.getInstance("bank");
        Logger log2 = Logger.getInstance("bank");

        log1.write("Withdrawal");
        log1.write("Deposit");
        log2.write("Transfer");
        log2.write("Deposit");

        // Thread t1 = new Thread(new RunnableProcess("bank", "deposit"));
        // Thread t2 = new Thread(new RunnableProcess("vault", "withdraw"));

        // t1.start();
        // t2.start();


    }
}
```

```
PS C:\Users\Cooper\OneDrive - University of New Brunswick\WI2023\SWE4403\Lab3> java demo
User made a Withdrawal from bank
User made a Deposit from bank
User made a Transfer from bank
User made a Deposit from bank
PS C:\Users\Cooper\OneDrive - University of New Brunswick\WI2023\SWE4403\Lab3>
```

d. Is your singleton implementation thread-safe? Why?

This implementation is not thread safe because multiple threads can access the instance at the same time. When the variable is not initialized, multiple threads can possibly enter the if loop that checks if the instance is null and create multiple instances of that object. This can lead to multiple instances of Loggers being created when the goal of the singleton pattern is to ensure only a single object exists so the program only can reference the one object.

e. Create a multithreaded application where two threads access the same logger.

i. Screenshot an output of your program showing that two threads using a logger.

```
    public void run() {
        try {
            Logger log = Logger.getInstance(fileName);
            Thread.sleep(this.time);
            log.write(type);
        }

        catch(Exception ex) {
            ex.printStackTrace();
        }
    }
```

C: > Users > Cooper > OneDrive - University of New Brunswick > WI2023 > SWE4403 > Lab3 > J demo.java

```
public class demo{

    public static void main(String args[]){

        Thread t1 = new Thread(new RunnableProcess("bank", "deposit"));
        Thread t2 = new Thread(new RunnableProcess("vault", "withdraw"));

        t1.start();
        t2.start();


    }
}
```

```
PS C:\Users\Cooper\OneDrive - University of New Brunswick\WI2023\SWE4403\Lab3> java demo
User made a deposit from bank
User made a withdraw from vault
PS C:\Users\Cooper\OneDrive - University of New Brunswick\WI2023\SWE4403\Lab3> java demo
User made a withdraw from vault
User made a deposit from bank
PS C:\Users\Cooper\OneDrive - University of New Brunswick\WI2023\SWE4403\Lab3> java demo
java.lang.NullPointerException: Cannot invoke "Logger.write(String)" because "<local1>" is null
        at RunnableProcess.run(RunnableProcess.java:18)
        at java.base/java.lang.Thread.run(Thread.java:833)
User made a withdraw from vault
PS C:\Users\Cooper\OneDrive - University of New Brunswick\WI2023\SWE4403\Lab3>
```

f. Modify your Logger class to make if thread safe.

i. Briefly explain why your solution is thread safe.

This implementation is thread safe because of the key word *synchronized*, meaning that the method of the object can only be used one at a time, therefore any object that can reference Logger is guaranteed to see the correct value. The instance being returned will be created before any other thread can access this method, so no new instances of a Logger will be created.

```java
public static synchronized Logger getInstance(String fileName){

    if(!(logMap.containsKey(fileName))){

        instance = new Logger(fileName);
        logMap.put(fileName, instance);
    }

    return logMap.get(fileName);
}
```

```
PS C:\Users\Cooper\OneDrive - University of New Brunswick\WI2023\SWE4403\Lab3> java demo
User made a withdraw from vault
User made a deposit from bank
PS C:\Users\Cooper\OneDrive - University of New Brunswick\WI2023\SWE4403\Lab3> java demo
User made a deposit from bank
User made a withdraw from vault
```

g. Run the code as in 2.e with the thread safe singleton implementation and see discuss the reasons for similarities or dissimilarities you encounter in the outputs of 2.e and 2.g.

The similarities between the code is that the outputs order can vary depending on the runnable process sleep time, which is why it displays a different Logger object first in the output. The differences is that with the thread safe option there is never an error in the output. This is because the synchronize in the getInstance method requires the method to finish before another thread can access it, so there are no discrepancies.

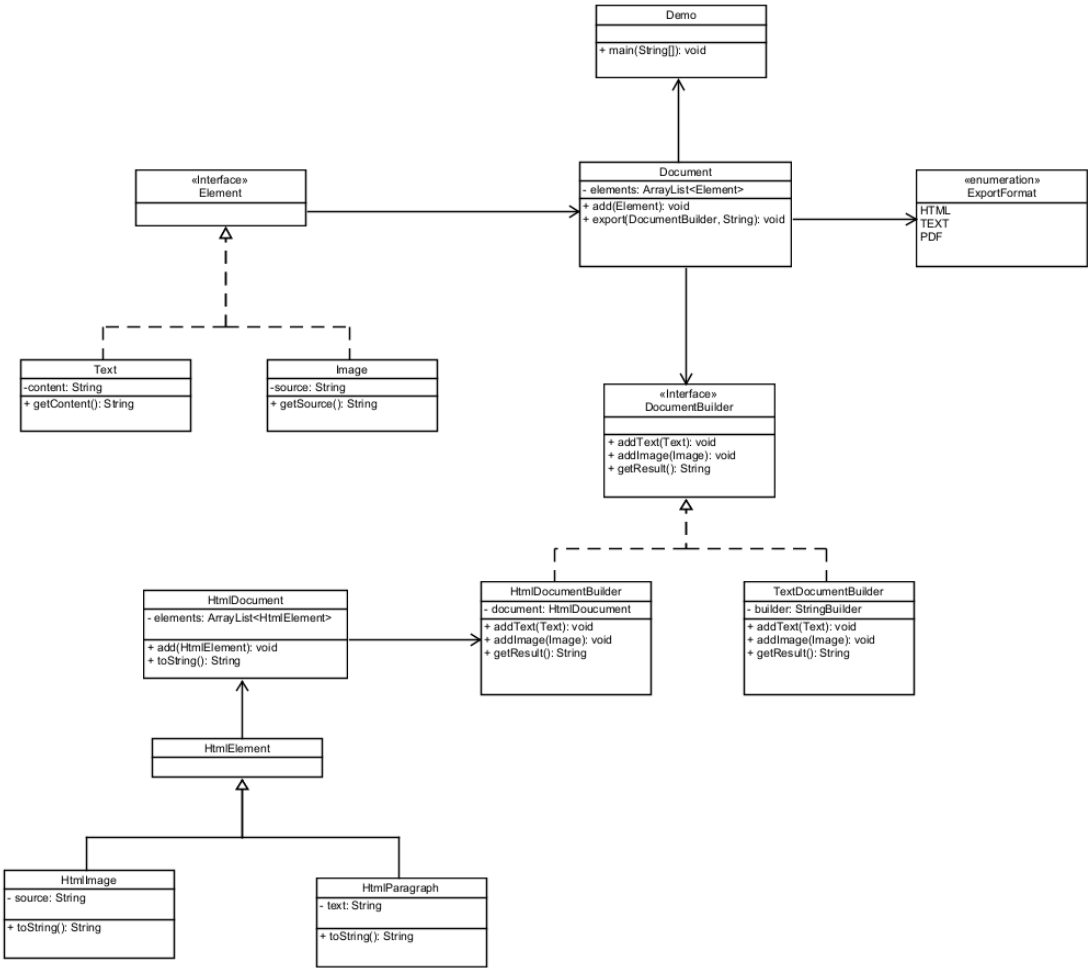
2. You are designing an editor application that generates documents in a specific binary format. As part of the user interface, you want to let the user export the file to other, more generalized, formats. For this purpose, you consider a menu option like the one presented in Figure 3.

a. Given the codes in files Lab03Exercise01option1.zip and Lab03Exercise01option2.zip:
i. Select a solution that you would prefer to implement for the app. Provide at least two reasons for your choice.

I chose solution two because it followed the gang of four principals more closely, specifically single responsibility rule and the open close principal. This option has follows the single responsibility rule because the complex objects are divided into more

concise classes making it simpler and easier to understand. Option two also follows the open close principle because the code is more flexible. This allows for the extension of different types of objects and in addition allows for the client to specify the components that are required.

ii. Draw the UML diagram of the selected option.



b. Upload your UML diagram to D2L