CUS1156 Programming Assignment 3

Loggers are software packages that automatically record messages about events, actions, data values, and errors that occur during the execution of a software program to provide an audit trail. Typically, messages are written to a logfile, but they could be displayed on a console or even inserted into a database system or fed to a debugging system. Developers, testers and support personal use the recorded information and data to identify software problems, to monitor live systems, for auditing purposes and post-deployment debugging.

Logging packages are typically programming-language dependent. For programs in C++, developers can use log4cxx or log4cplus. Logging for PHP programs can be done with log4php or Pear's Log package. And in Java, log4j, the Java Logging API, and SLF4J are all quite popular. However, for this project, you are going to build your own logger.

Loggers usually work with the concept of a log level. The log level determines what kinds of messages are actually logged. A current log level is set when the logger object is instantiated. Each log message has an associated log level, and is only written out to the log file if it is at the current log level or higher. Here are the log levels
SEVERE Severe errors that cause premature termination.
ERROR Other runtime errors or unexpected conditions.
WARNING System errors that do not cause the system to terminate, but could indicate a problem.
INFO Interesting runtime events (startup/shutdown).
DEBUG Detailed information on the flow through the system.

So, for example, if the current log level is set to WARNING, then messages at level FATAL, ERROR, or WARNING will be logged, but messages set to level INFO or DEBUG will not be. It is typical to only set the current log level to DEBUG while the software is being developed – once it is deployed, the log level is typically set to WARNING in order to keep the size of the logfile small.

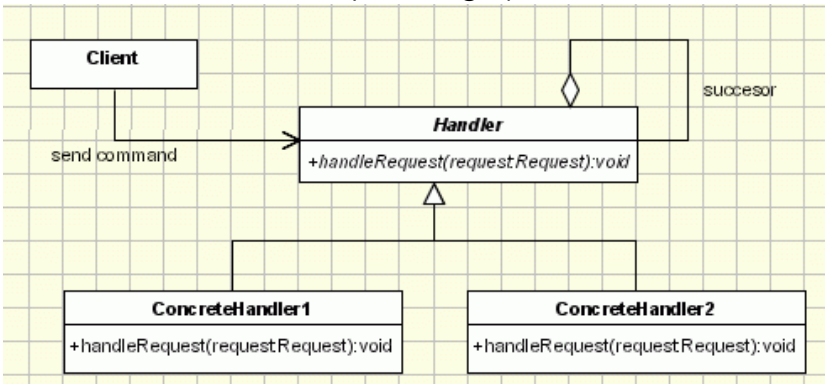
Here is an example of the code in a program that uses a logger

```
try {
        SimpleLogger logger = LoggerFactory.getLogger("File");
        logger.setFormatter(new XMLFormatter());
        logger.setLevel(Level.WARNING);

         logger.log(Level.INFO, "main", "system starting");

         double res1 = doDivision(2,3);
         double res2 = doDivision(3,0);
         logger.log(Level.INFO,  "main", "system ending");

 } catch (LoggerException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
```

And another example

```
if ( j == 0)
{
  logger.log(Level.SEVERE,"doDivision", "Attempt to divide by 0");

}
```

As stated earlier, logging may go to the console, to a file, to a software system, or to a database. Developers may want to easily swap logging methods, without having to rewrite a lot of code. Therefore, it makes sense to have an abstract class serve as the parent class, with specific implementations inheriting from that class. Then, developers who are using the logging system can use the same API whether logging to a file or to a database. This is actually a design pattern known as Chain of Responsibility.



It then also makes sense to have a Factory class that creates the specific type of logger needed. In the first code example, LoggerFactory performs that function. In this example, the getLogger method will create a FileLogger object and return it. The FileLogger class is a concrete implementation of the abstract class SimpleLogger. We could create a DatabaseLogger exactly the same way

```
SimpleLogger logger = LoggerFactory.getLogger("Database");
```

and then the rest of the code would be the same.

Loggers also usually have associated formatters. This is to make it easy to generate correctly formatted logs. For example, you might use a plain text formatter to generate log messages like this
2016-04-03T11:05:22.562:INFO:main:system ending
And an XML formatter to generate a message like this

```
<LogMsg>
        <Level>INFO</Level>
        <DateTime>2016-04-04T17:55:53.672</DateTime>
        <Component>doDivision</Component>
        <Msg>result is 0.0</Msg>
</LogMsg>
```

There is one more consideration. A logger should be a Singleton. The reason is because there should be only one logger in a software system, but it is needed throughout the system.

In this project you will work as a pair. One person will create FileLogger and the associated classes – the factory class, and a LoggerException class that wraps exceptions such as IOException. I will give you the abstract class SimpleLogger, which FileLogger will extend.

In addition, since this class is a Singleton, there should be a method
`public static FileLogger getInstance()`
as well as a private constructor. However, programs that use the logger will use the LoggerFactory to create the logger instance. That is because not all logger implementations are necessarily Singletons, and may not have a getInstance() method.

The other person will create PlainStringFormatter and XMLFormatter. These classes will implement the Formatter interface, which I am also providing. You will need to also implement an enumeration class for the levels, and LogMessage. This class should contain one log message: log level, date/time, component, and message. Look at https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html for an explanation of enumeration classes.

When a message is logged, the logfile should contain the following information:

- A timestamp
- The log level of the message
- The component that the program was in when the message is logged
- The message itself, or the stack trace of the Throwable (in the case of the second log method

The logfile should always be appended to, so that log messages from successive invocations of the logged software all appear in the file. This allows developers to compare log messages over time.

HOW TO TEST WHEN YOUR PARTNER'S CODE IS MISSING

To test the formatters on their own, write Junit tests.

To test the Logger without a formatter, simply create a string with just the date and message (for example) and print that out.

The team member doing the formatters should do PlainStringFormatter first since that is the simplest one, and should get it into GitHub ASAP.

Grading will be individual and will be based on the following:
Working 35
-compiles

-functionality

Design 35 (specific criteria depends on the classes you implemented)
- Singleton pattern correctly implemented
- Factory class correctly implemented
- Chain of Responsibility pattern is correctly implemented for both FileLogger and the formatters.
- classes have appropriate methods and maintain encapsulation
-Code is maintainable, no repeated code, constants are used

Collaboration 10
-maintained communication with your partner, made your code available on a timely basis
- both partners pushed a final version to GitHub.

Readability 10
-easy to understand class names, method names, and variable names
-Javadoc comments are informative
-code is formatted correctly

Work plan 10, due April 15
Give a detailed plan showing the order in which you will implement the classes and target completion dates for each class. Also explain your testing strategy. Both partners should submit this to Blackboard

This is due on April 29. Please submit to Blackboard AND push the final version to GitHub. Since there is no class that day, you do not need to hand in a paper copy.