QL-5) Working with Untestable Code.

Consider this small but innocuous piece of code

```
package com.celestial.files;

import java.time.LocalTime;

public class DataClerk
{
    public class FileLog
    {
        public  void    ClearTheLog()
        {
            // Simulated method that would do something to files in the
log
        }
    }

   private FileLog theFileLog;

   public  void    ProcessData()
   {
      LocalTime now = LocalTime.now();
      LocalTime stopTime = LocalTime.parse("20:00");

      if( now.isBefore(stopTime) )
      {
          System.out.println("Ready to process the data");
          FileLog fl = new FileLog();
          fl.ClearTheLog();
      }
   }

}
```

And here is the initial test

```
package com.celestial.files;

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class DataClerkTest
{
   @Test
   public void testProcessData()
   {
      // arrange
      DataClerk cut = new DataClerk();

      // act
      cut.ProcessData();

      // assert
      // There is no way of knowing if this ran as expected
   }
}
```

There are a number of things that should cause the alarm bells to ring

1. Line 22 constrains the testing of this method. We can only test all paths if the now variable holds a time that is less than 2000 hours
2. Line 25, the dependent class could cause an issue if it performs IO, Network, and DB calls.

Line 25 can be resolved by injecting the FileLog in through the constructor or ProcessData method. You would need to refactor the code and write a test to prove the CUT is not broken

Line 22 can be resolved by refactoring the code so that Lines 19-20 are in a method of their own, and calling that method in Line 22 if(getTime(). isBefore(stopTime))

Activities

1. If you run the tests before 8pm you should see the message `Ready to process the data`. If you change the time at line 20 to an hour before you are running the code, the message will not be displayed. This is not a suitable way to test if this code actually works as expected.
2. Refactoring the code and writing a new test to support the refactored code - read Activity 1 Observations
   a. Refactor the code so that the FileLog is injected into the DataClerk
   b. Decide how you can test it to verify that the code is still working.
   c. Give attention to whether you should be injecting the class or an interface
   d. Use the mockito doNothing().when() (or a library you are familiar with that has the same capabilities) feature to set up an expectation that can be used to verify when `DataClerk.clearLog()` is being called
3. Line 22 is an issue. The timestamp is hard coded. Not a problem you say. We can remove the magic number so it's declared as a constant field in the class. This doesn't really help us because we can only test alternative paths after 8pm. We can use a Spy mechanism to solve this. This can be done by refactoring the code so that the current time is returned from a private method in the DataClerk class.
   a. Turn line 19 into a private function
   b. Use the `package` level function to get the time
   c. In the test use mockito spy to override the returned time so it is greater than 2000 hours.

      d. `Mockito.doReturn(LocalTime.parse("21:00")).when(cutSpy).getTime();`

4. An alternative design for activity 2 would be to use a lambda expression in line 22
   a. Think about how this might work, and what would the test look like?

Solution can be found in java-why-spy.zip

Alternately git repo