Jonathon Roscoe
CS 4910-06
HW 2

1. **Given the following driver code, refactor to improve readability and increase decomposition (*Omitting full original driver code due to length. Both original and modified files uploaded with assignment)***

   Initially, the first thing that stood out to me when examining the code was the large amount of print statements within main(). I decided to extract method into a DisplayWelcomePrompt() method. I included the word Prompt within the name because it prompts the user to press a key to continue.

```
public static void Main() {
    Console.WriteLine("Welcome … press 'enter' to
      continue");
    Console.ReadLine();
    Console.WriteLine("This program has 2 steps …
      completely \n" +
      "and the second step is …");
    Console.WriteLine("the purpose … \n" +
      "until…");
```

```
private void DisplayWelcomePrompt()
{
    Console.WriteLine(
        "Welcome, this program has 2 steps …
            completely \n" +
        "and the second step is … \n" +
        "the purpose … \n" +
        "until… \n" +
        "press 'enter' to continue"
        );
    Console.ReadLine();
}

public static void Main()
{
    DisplayWelcomePrompt();
```

   While it changes the functionality of the program slightly, I felt it was much more appropriate to move the ReadLine() down to bottom of the method. This allows the whole introduction to be printed and read by the user before continuing execution. Merging the WriteLine() calls into one single call also improves readability and makes it easier to determine the exact format of the output.

   For consistency, I next extracted the single Goodbye message at the end of the driver into its own method. While the method is only 1 line, this facilitates future modifications.

```
public static void Main()
{
    …
    Console.WriteLine("\n goodBye");
}
```

```
private void DisplayGoodbyePrompt()
{
    Console.WriteLine("\n goodBye");
}

public static void Main()
{
    …
    DisplayGoodbyePrompt();
}
```

Jonathon Roscoe
CS 4910-06
HW 2

Next, I began to examine more closely what the driver itself is doing. Based on my understanding, it initializes Factor objects to a positive integer supplied by the user and performs both an automatic and manual (requires user input) test on several Factor methods.

I decided to extract the while loop that takes a positive integer from the user, as it is used twice within the driver.

Original:

```
public static void Main()
{
   …
   int input = 0;
    while (input <= 0)
    {
        Console.WriteLine("please enter … ");
        input = Convert.ToInt32(Console.ReadLine());
    }

    factor = new Factor(input);

    Console.WriteLine("you've inserted " + input + " program …\n" +
        "until … for testing purposes");
```

Refactored:

```
private int PromptUserForPositiveInteger()
{
    int input = 0;

    while (input <= 0)
    {
        Console.WriteLine("Please enter a positive integer ...");
        int.TryParse(Console.ReadLine(), out input);
    }

    Console.WriteLine(
        "you've inserted " + input + " program …\n" +
        "until … for testing purposes"
        );

    return input;
}
```

Jonathon Roscoe
CS 4910-06
HW 2

```
public static void Main()
{
    …
    int input = PromptUserForPositiveInteger();

    Factor factor = new Factor(input);
```

Here I made both input and factor local to Main(), reducing clutter of global variables. The assumption made here is that these two variables are not used in any other methods, so reducing their scope will not impact any other parts of the program. TryParse() is used instead of Convert to eliminate unhandled exceptions if a non-numeric string is entered.

Next, I extracted the while loop that runs the first test. Instead of having the numbers 3 and 4 hard coded, I replaced them with constants in main and pass them as parameters to the extracted method. This allows extensibility in testing, as it is easy to now change the range of the generated numbers. Assuming that the random variable is not used anywhere else in main, I decided to initialize it in the extracted method instead. The if statement inside the while loop can be moved underneath, assuming calling query() repeatedly does not affect the factor object's state.

Original:

```
public static void Main()
{
    …
    random = new Random(DateTime.Now.Millisecond);

    while (factor.getActive() && factor.query() != 3)
    {
        int divTest = random.Next(input, input * 4);
        Console.WriteLine("testing " + divTest);
        factor.div(divTest);
        if (factor.query() == 3)
            Console.WriteLine(" 3 multiples found ");
    }

    Console.WriteLine("\n automated test finished \n");
```

Jonathon Roscoe
CS 4910-06
HW 2

Refactored:

```
private void AutomatedDivTest(Factor factor, int input, int queryLimit, int
randScalar)
{
    Random random = new Random(DateTime.Now.Millisecond);

    while (factor.getActive() && factor.query() != queryLimit)
    {
        int divTest = random.Next(input, input * randScalar);
        Console.WriteLine("testing " + divTest);
        factor.div(divTest);
    }

    if (factor.query() == queryLimit)
        Console.WriteLine(" 3 multiples found ");

    Console.WriteLine("\n automated test finished \n");
}

public static void Main()
{
    const int AUTOMATED_TEST_SCALAR = 4;
    const int AUTOMATED_TEST_LIMIT = 3;
    …
    AutomatedDivTest(factor, input, AUTOMATED_TEST_LIMIT,
      AUTOMATED_TEST_SCALAR);
```

Finally, I extracted the second “test” that requires user input into its own method. I don’t agree with test cases that require user input to run, but I wanted to keep the functionality of the code similar to the original, so for the most part I left it alone. I did change the Convert again to TryParse() to avoid errors on bad input.

Original:

```
public static void Main()
{
    …
    while (factor.getActive())
    {
        input = Convert.ToInt32(Console.ReadLine());
        factor.div(input);
    }
```

Jonathon Roscoe
CS 4910-06
HW 2

Refactored:

```
private void ManualDivTest(Factor factor)
{
    Console.WriteLine("Now insert your desired ...\n");

    int input = 0;

    while (factor.getActive())
    {
        int.TryParse(Console.ReadLine(), out input);
        factor.div(input);
    }
}

public static void Main()
{
    …
    ManualDivTest(factor);
```

To test these changes, I created a simple Factor class that implements the methods tested. (*See Factor.cs)* I ran both main functions using, passing preinitialized factor objects and omitting any code that required user input. Due to the random number generation, the outputs were not the exact same, but the execution flow through both drivers was maintained. The original driver was written quite poorly, I feel like modifying it to effectively test against an array of factor objects would require a complete rewrite. *See Q1_TestDriver.cs for test driver*

2. Given the following code, extracted from a class (type) definition
   a. Refactor the following code to improve readability and decrease layering
   b. Summarize the potential impact of your refactoring

Original:

```
class Question2Class
{
    private void SwitchOff()
    {
        activeState = false;
    }
```

Jonathon Roscoe
CS 4910-06
HW 2

```
    public bool IsActive()
    {
        return activeState;
    }

    private void CheckLastVal(int check)
    {
        if (lastValue == check)
            SwitchOff();
    }

    public bool Div(int testVal)
    {
        if (testVal == 0)
        {
            return false;
        }
        CheckLastVal(testVal);

        if (IsActive())
        {
            lastValue = testVal;
            if (testVal % factorValue == 0)
            {
                countMultiples++;
                return true;
            }
            return false;
        }
        return false;
    }
}
```

The first thing that catches my attention is the IsActive() function. While somewhat semantic, I think it is syntactically much prettier to use get and set instead.

```
public bool Active {get; protected set;}
```

This change requires all calls within the class to IsActive to be changed to reference Active instead and will require any client code that calls this method to be changed as well. Granted this might require a lot of work for a small syntactically change, in this case I am assuming that these additional changes will be minimal.

Jonathon Roscoe
CS 4910-06
HW 2

Next, the SwitchOff() function is private and only does one thing.

```
private void SwitchOff()
{
    activeState = false;
}
```

This can method can be in-lined without much additional work as it can only be reference from within the class because it is private. The only place it is called is in CheckLastVal()

Original:

```
private void CheckLastVal(int check)
{
    if (lastValue == check)
        SwitchOff();
}
```

Refactored:

```
private void CheckLastVal(int check)
{
    if (lastValue == check)
        Active = false;
}
```

This method itself can also be in-lined as it is another private one liner. The only place this function is referenced is within Div(). Turning my attention to the Div() method, I also flatten out the nested if statements without having to worry about any hidden side effects as I am not calling any unknown methods. The outer if statement can be inverted and turned into a guard clause

Original:

```
public bool Div(int testVal)
{
    if (testVal == 0)
    {
        return false;
    }
    CheckLastVal(testVal);

    if (IsActive())
```

Jonathon Roscoe
CS 4910-06
HW 2

```
        {
            lastValue = testVal;
            if (testVal % factorValue == 0)
            {
                countMultiples++;
                return true;
            }
            return false;
        }
        return false;
    }
```

Refactored:

```
    public bool Div(int testVal)
    {
        if (testVal == 0)
            return false;

        if (lastValue == testVal)
            Active = false;

        if (!Active)
            return false;

        lastValue = testVal;
        if (testVal % factorValue == 0)
        {
            countMultiples++;
            return true;
        }

        return false;
    }
```

Jonathon Roscoe
CS 4910-06
HW 2

Having eliminated the two private methods, I added a constructor for testing as well as a private factorVal. The partial class refactored looks like this:

```
class Question2Class
{
    public Question2Class(int factorValue, bool active)
    {
        this.factorValue = factorValue;
        Active = active;
        lastValue = -1;
        countMultiples = 0;
    }

    public bool Active {get; protected set;}

    public bool Div(int testVal)
    {
        if (testVal == 0)
            return false;

        if (lastValue == testVal)
            Active = false;

        if (!Active)
            return false;

        lastValue = testVal;
        if (testVal % factorValue == 0)
        {
            countMultiples++;
            return true;
        }

        return false;
    }

    private int factorValue;
    private int lastValue;
    private int countMultiples;
}
```

Jonathon Roscoe
CS 4910-06
HW 2

Using a similar testing method to the last assignment, I wrote a driver that compared the return calls for both the old and new classes public methods. *See hw2p2.cs for full test driver.*

3. Software design and refactoring overlap. Consider the following posts:

Read the blog posts and their associated comments, all of which address the task of clarifying control flow. Consider all the restructurings suggested and then complete the following:

Categorize all suggested restructurings by general effect: none, good, ill-advised.

Justify your categorization of each restructuring. Assess the effect of each restructuring on testing

Technique	**Analysis**
Put shorter clause first	Good, as long as the conditional statements don't have any hidden dependencies on the order they are called in. Putting the shorter condition first makes it much less intimidating to read. Testing wise you would need to make sure the state remains the same throughout all possible conditional flows.
Removing the Else statement when you return in the if	Good, reduces code clutter and in my opinion makes it easier to trace flow.
Hoisting return statement into if-else clause	None, can only be done when the conditional statement is immediately followed by a return statement and prevents additional code from being added following the exit of the conditional block.
Manifesting implicit flow control	Good, as long there are no elseif statements that could potentially modify state. Testing should ensure that any statements skipped by continue do not modify state of the program.
Removing redundant if	GOOD, I once used to code like this and now it is one of my pet peeves.
Adding else	None, undecided about this one, as long as you remove the else afterwards it is simply inverting the conditional which should not have any effect on testing. Otherwise, adding an else statement

Jonathon Roscoe
CS 4910-06
HW 2

	where one is implicitly implied is up for debate.
https://softwareengineering.stackexchange.com/a/122488 (the second example they give)	Ill-advised, having a function that returns null is not very versatile and requires you to check the return value every time you call get_contents().
https://softwareengineering.stackexchange.com/a/122510 double condition if	Good, I approve of this double conditional. It might be a little obscure to assign contents at the same time it is being passed to SomeTest(), but once you understand what is happening it is very pretty. Testing this should not have any adverse effects given that the language supports short circuiting.
https://softwareengineering.stackexchange.com/a/122508	Ill-advised. Given this solution was designed to be bad, using goto statements obscures flow. Using negative if conditionals is also a smell. This comment by herby on this one is even uglier, wrapping the whole code in a for loop.