Name: Jamie Penner Date: 12/1/19

Lab 8 – Command Pattern

Objectives

Part 1 – Setting up the Command Starter Code

Part 2 – Create a BatchInvoker

Background/Scenario

The command pattern is comprised of two parts: Command(s) and an Invoker. The invoker can wrap additional functionality around the invocation of the command. For this lab, you will use the CommandLogging demo as your starter code. From our discussion in class, we know that the CommandLogging demo uses the decorator, singleton, and command patterns to implement a "SuperSafeSpreadsheet" that saves/records the SetValue command to a binary file and can "replay" the commands later to recover lost work.

In this lab, we are going to create a new "batching" invoker that executes the SimpleSpreadsheet.SetValue() method only when five SetValue commands have been received. The new BatchInvoker will wait until it has received a batch of five SetValue commands. Once it has received a full batch of five SetValue commands, it will execute all five commands and then wait until another full batch has been received.

Required Resources

- Visual Studio 2017

Part 1: Setting up the Command Starter Code

Step 1: Download the CommandLoggingStarterCode.zip

Download the CommandLoggingStarterCode.zip from Canvas and extract it to your repository.

Step 2: Build and run the code.

Ensure everything builds successfully and familiarize yourself with the solution.

There are several projects many of which are libraries and one is the main project.

Part 2: Create a BatchInvoker

Step 1: Add a BatchInvoker.cs file to the Command project.

In your BatchInvoker class, instantiate a private List data member that contains ICommand types.

Add a public void Execute(ICommand command) method. This method should add the command parameter to the list and, when five commands have been added, it should foreach through the list, executing the commands.

After executing the commands, it should clear the list to make room for the next five commands. Add Thread.Sleep(500) after clearing commands from list.

Step 2: Add a SpreadsheetBatchCommandDecorator.cs file to the Decorator project.

This class will need to inherit from the SpreadsheetDecorator class. Add the requisite namespaces.

Create a private instance of the BatchInvoker—name it “invoker.”

Add the proper concrete decorator constructor.

Override the SetValue() method of the SpreadsheetDecorator. In the overridden SetValue() method, create an instance of a SetValue command. Next, call invoker.Execute(command); where “command” is the instance of the SetValue command just created.

Step 3: In Program.cs, add a new private static void BatchSpreadsheet() method.

Inside BatchSpreadsheet(), create an instance of a SimpleSpreadsheet.

Decorate the simplespreadsheet with an instance of the SpreadsheetBatchCommandDecorator.

Call the UpdateSpreadsheet() and ValidateSpreadsheet() helper functions, passing in the decorated spreadsheet, to prove your code still works as designed.

Step 4: Update Main

Comment out all code in Main() and add a call to BatchSpreadsheet().

Run the app—it should not throw any exceptions.

Reflection

1. Did you have any issues with the lab?

   I had an exception from not creating a new instance of the batchinvoker in the concrete decorator constructor. I was able to figure out the issue and get the program to work as intended.

2. Create a sequence diagram of the events that occur when the SpreadsheetBatchCommandDecorator’s SetValue() method is invoked by client code.

   Show the creation events. Show the event sequence for the first four SetValue() calls. Lastly, show the sequence of events on the fifth SetValue() call.

3. Update the UML diagram (handed out in class) by replacing classes that are no longer used with the new classes needed to batch commands.