Command Pattern

Author: Brian Crites

You <u>must</u> work in a group of two for this lab

The command pattern is used to encapsulate a request or a change in state. It is used in the Lexi example from the slides to allow for users to change the document, and allow us to redo and undo these changes. It accomplishes that by having any user request to change the system issue a command object that encapsulates the information necessary to fully describe the change.

For example, if we were going to change a segment of code from font A to font B, we would need to keep the start and end points of the text we are going to change as well as font A as the starting font and font B as the ending font. This would allow us to make the change from font A to font B, as well as change that same segment of the document back if necessary. We can go back and forward into our commands history by keeping a collection of commands, as well as a reference to the current command we have run (which will change if we undo/redo commands).

The Assignment

In this lab we will build upon the composite class that you created in the composite and strategy lab to build a (very) simple calculator application. This calculator will allow the user to do running total calculations (similar to a simple phone calculator) with the additional function of allowing the user to undo and redo commands that they have input. We will accomplish this by leveraging the command pattern to create a history that we can write to, traverse, and overwrite as necessary, and which will save the state of our calculation every time a new operation is requested.

Since this lab relies on a completed composite pattern, you have been provided with an implementation of the composite pattern that you were to design. You may also use your own composite classes if you would prefer.

You have also been provided with a main function in `calculator.cpp` that will deal with user input and output, and interface that user I/O with the menu class. This menu has the following commands:

- Exit the Program
    - Typing “exit” as a command will exit the program
- Undo a Command

    - Typing “undo” will go back to the last command, and can be called until there are no more commands to undo
- Redo a Command
    - Typing “redo” will go forward to the next command, and can be called until there are no more commands to redo
- Do a Calculation
    - Do a running calculation. This calculation must start with a single number (ex: 3), and after a single number has been input should take an operator followed by a number (ex: + 10). Note that a space is required between the operator and the number.

The Menu Class

The menu class is used as an interface between user input and the commands that are created for keeping track of the calculation. It contains two internal members `vector<Command*> history`, which is used to hold all the commands that have been run, and `int history_index`, which is used to keep track of our position in history.

The menu class also has the following functions, which have been declared but their functionality has not yet been written.

- `Menu()`
    - Constructor, which properly sets all necessary data members
- `void execute()`
    - Executes the command that we are currently referring to (if any commands exist)
- `bool initialized()`
    - Checks if the history has an initial command, this is necessary because the first command must be a number with no operation
- `void add_command(Command* cmd)`
    - Adds or replaces a command object to the next position after the history command we are currently referring to, and advances the history_index
- `Command* get_command()`
    - Returns the current command we are referring to, so it can be used as the base for the next command
- `void undo()`
    - Undo a command, going back to the statement before the command we are currently at (being careful to check our history’s size)
- `void redo()`
    - Redo a command, going forward to the statement after the command we are currently at (being careful to check our history’s size)

The Command Class

You will also have to create command classes, which are capable of encapsulating the current state of a calculation. Since we are doing a running calculation, you do not need to worry about accounting for precedence. Instead you can guarantee that the composite tree from the last command will be one child of the new command, and the new value input will be the other child (with the parent of both being determined by the operator that is input by the user).

This means that you will be required to create 5 concrete command classes (one for each composite node type) as well as a command base class.

```
BaseCommand
OpCommand
AddCommand
SubCommand
MultCommand
SqrCommand
```

The base class has been predefined in the `command.h` file. It has two functions `execute()`, which evaluates the composite tree that that command holds, and `get_root()`, which returns the commands tree so that it can be used as a subtree in another commands tree. You will be in charge of creating the following classes, each of which will have a single constructor that is in charge of adding the proper new operator, which will have the last commands tree as one child and the newly input value as the other child.

There are two exceptions; the `OpCommand` class, which will only take in a value and will create a single Op composite node as the commands data, and the `SqrCommand` class, which will only take in a subtree (which is necessary for creating a Sqr composite node).