Command Undo Pattern

Estimated time for completion: 60 minutes

Overview:

The solution you are provided with implements a basic Sudoku game environment, allowing the user to attempt to solve a Sudoku by electing which value will go in a given square using the left mouse button to produce a drop down of values to go in the cell, or using the right mouse button to mark a value as a possible, and thus it is written in italic in the top left hand corner. When the user has successfully solved the puzzle a congratulations box appears. If the user gets frustrated they can either invoke the solver which will solve the entire grid (hopefully) or they could hit the Assist button to just help them move forward a bit. Your task in this lab is to provide Undo/Redo functionality as one of the most frustrating things with Sudoku is when you realize you have made a mistake you want to revert back to possibly have another go...

Goals:

- Develop and put to use the command undo pattern to implement Undo/Redo behaviour for the classic puzzle Sudoku
- Get better at Sudoku, ;-)

Lab Notes:

Part 1: Undo Invoker

In this part you will develop the Command pattern infrastructure so that in can be used with our Sudoku game. Whilst you could just build it and then simply use it in our game, it may be prudent to write some simple code to test it.

1. Open the WinSudoku.sln solution, add a new class library assembly to the project and call it CommandPattern. Then add a class called Command to this assembly, this will represent the basic abstract implementation of all commands that the invokers will be written against. This command class will have two methods Execute and Undo, Execute will be completely abstract whilst Undo, may be abstract or you could declare it virtual and make the implementation throw a NotImplementedException. This is useful if you were to write other invokers that don't need undo functionality.

```
public abstract class Command
{
   public abstract void Execute();

   public virtual void Undo()
   {
      throw new NotImplementedException("Undo not supported");
   }
}
```

© 2010 Digital Age Learning. All Rights Reserved

2. Now you need to build a Command Undo/Redo Invoker. The responsibility of this invoker is to remember each command once it has been successfully executed, such that if you want to undo the last operation that has been executed it will call the undo method on the command. Create a new class inside the CommandPattern assembly called UndoRedoInvoker
3. The Invoker will need at least three methods Execute, Undo, Redo. The execute method will take the command to be executed, and the Undo and Redo will take no parameters and simply perform the said operation. It would also be advisable to have query properties to determine if there are any commands to be undone or redone. Create stub methods for the necessary methods for the UndoRedoInvoker class and check that it all compiles

```
public class UndoRedoInvoker
{
   public void Execute(Command command){ }

   public bool HasCommandsToUndo
   {
      get { return false; }
   }

   public void UndoLastCommand() { }

   public bool HasCommandsToRedo
   {
      get { return false; }
   }

   public void RedoLastUndo() { }
}
```

4. Now implement the basic undo behavior for the invoker. It needs to remember the commands it has executed and if asked to undo, call the undo method for each command it has previously executed in LIFO order (Last In, First Out), so it would seem natural to use a Stack to hold the set of executed commands. Add a private field to your UndoRedoInvoker of type Stack<Command;>, and add the appropriate code to the Execute method to record the commands, and confirm that it compiles

```
public class UndoRedoInvoker
{
   private Stack<Command> commandsToUndo = new Stack<Command>();

   public void Execute(Command command)
   {
      command.Execute();
      commandsToUndo.Push(command);
   }

   public bool HasCommandsToUndo
   {
      get { return commandsToUndo.Count > 0; }
```

© 2010 Digital Age Learning. All Rights Reserved

```
    }

    public void UndoLastCommand()
    {
        Command cmd = commandsToUndo.Pop();

        cmd.Undo();
    }

// ...
}
```

5. Now look at the SimpleUndoRedo project inside this solution, this project simply creates a simple class to represent a bank account having a Credit and Debit operation. You will use this simple type as a test vehicle for your Undo/Redo logic. With the knowledge you gained in the previous Command lab create two command classes inside the SimpleUndoRedo project to represent the Credit and Debit operations, only implement the Execute method for now. Inside main create an instance of your UndoRedoInvoker and then using the command types you have created for Credit/Debit validate that you can apply credit and debit operations to the account via your invoker.
6. Now implement the Undo method in each one of the commands, the role of the Undo method is to perform the reverse operation of the execute method, so the CreditCommand will make a call to target.Debit(amount) in order to reverse the Credit operation. Test this undo logic by making a series of requests to the account, and then execute the Undo method on the Invoker this should then result in the previous command being rolled back.

```
public class CreditCommand : Command
{
    private BankAccount target;
    private decimal amount;

    public CreditCommand(BankAccount account , decimal amount)
    {
        target = account;
        this.amount = amount;
    }
    public override void Execute()
    {
        target.Credit(amount);
    }

    public override void Undo()
    {
        target.Debit(amount);
    }
}
static void Main(string[] args)
{
    UndoRedoInvoker invoker = new UndoRedoInvoker();
    BankAccount account = new BankAccount();

    invoker.Execute(new CreditCommand(account, 10));
```

© 2010 Digital Age Learning. All Rights Reserved

```
   invoker.Execute(new CreditCommand(account, 10));
   invoker.Execute(new CreditCommand(account, 10));

   Console.WriteLine("Balance is {0:C}" , account.Balance);

   while (invoker.HasCommandsToUndo)
   {
      invoker.UndoLastCommand();

      Console.WriteLine("Balance is {0:C}", account.Balance);
   }
}
```

7. Now time to implement the Redo logic, Redo is very similar to Undo in that after you have performed an Undo you need to remember the fact that you have undone the command so that if asked to redo you simply execute last command that was undone. This is very similar to Undo, in fact you will redo each undone command using LIFO too. So in addition to a stack to store executed commands for undo operations you also need one for Redo. Add a new private field to the invoker for redo commands, of type Stack<Command>, and then modify the UndoRedoInvoker class as follows

```
public class UndoRedoInvoker
{
   private Stack<Command> commandsToUndo = new Stack<Command>();
   private Stack<Command> commandsToRedo = new Stack<Command>();

   public void Execute(Command command)
   {
      commandsToRedo.Clear();

      command.Execute();
      commandsToUndo.Push(command);
   }

   public bool HasCommandsToUndo
   {
      get { return commandsToUndo.Count > 0; }
   }

   public void UndoLastCommand()
   {
      Command cmd = commandsToUndo.Pop();

      cmd.Undo();

      commandsToRedo.Push(cmd);
   }

   public bool HasCommandsToRedo
   {
      get { return commandsToRedo.Count > 0; }
   }
```

© 2010 Digital Age Learning. All Rights Reserved

```
    public void RedoLastUndo()
    {
        Command cmd = commandsToRedo.Pop();

        cmd.Execute();

        commandsToUndo.Push(cmd);
    }
}
```

8. Now modify your test code to validate that redo works too

```
static void Main(string[] args)
{
    UndoRedoInvoker invoker = new UndoRedoInvoker();
    BankAccount account = new BankAccount();

    // account.Credit(10);

    invoker.Execute(new CreditCommand(account, 10));
    invoker.Execute(new CreditCommand(account, 10));
    invoker.Execute(new CreditCommand(account, 10));

    Console.WriteLine("Balance is {0:C}" , account.Balance);

    while (invoker.HasCommandsToUndo)
    {
        invoker.UndoLastCommand();

        Console.WriteLine("Balance is {0:C}", account.Balance);
    }

    while (invoker.HasCommandsToRedo)
    {
        invoker.RedoLastUndo();

        Console.WriteLine("Balance is {0:C}", account.Balance);
    }

}
```

Part 2: Undo/Redo for Sudoku

In this part you will take the command pattern assembly you built in part one and use it to implement Undo/Redo behavior in the game. This will involve creating a new kind of Sudoku board, one that supports Undo and Redo.

9. Ensure that WinSudoku project is the startup project and Run the application, have a quick play and ensure you are happy with how the user interacts with the app, pressing the undo or redo buttons simply results in a not implemented popup. This is the new functionality you need to add.

© 2010 Digital Age Learning. All Rights Reserved

10. The application has the following basic structure, you will need to only perform minor changes in the UI layer, most of the changes will be done in the Sudoku assembly.
11. The BoardControl is core to the interaction between the user and the model, the BoardControl takes the user input and invokes the appropriate methods on the board. The board implements the observer pattern allowing the BoardControl to register for changes in the state of the board. Familiarize yourself with the structure and the code.
12. To implement undo/redo functionality you could modify the BoardControl itself to use commands and an invoker but that would mean modifying existing code and the undo/redo functionality would be bound into the implementation of the UI, which would not be great. You could modify the BasicBoard type to also support undo/redo but again that means modifying working code, a far better solution would be to have a new type of Board called an UndoBoard. The UndoBoard has all the same implementation as the BasicBoard but will need to override the actions of setting a cell and marking/unmarking possible values and use commands in order to implement the undo behavior. This perhaps suggests inheritance, but that would mean Undo/Redo functionality for a specific implementation of a Sudoku board, and looking at the code structure you can see trouble has been taken to allow the UI to work with any type of implementation.
13. The Sudoku Form is decoupled from the actual board being used by the fact that all its code is written against the abstract Board class. It is when the board is used in the context of the form that you wish to add the undo/redo behavior. The board has already been created so inheritance is really not an option you will have to use runtime style inheritance, via the Decorator pattern. Your first task is therefore to create a base decorator class for the Board type, and then create a specific decorator, called the UndoBoardDecorator, and add two stubbed out methods Undo and Redo. The structure you need to implement will look like this
14. It may not be too obvious how you build the decorator around event registration, you effectively have to implement the add and remove methods for the event, since the backing store for the event is in the wrapped object

```
public override event BoardCellUpdateHandler BoardCellUpdate
{
   add { wrappedBoard.BoardCellUpdate += value; }
   remove { wrappedBoard.BoardCellUpdate -= value; }
}

public override event EventHandler<EventArgs> Solved
{
   add { wrappedBoard.Solved += value; }
   remove { wrappedBoard.Solved -= value; }
}
```

15. Having built the BoardDecorator, and the UndoBoardDecorator confirm everything compiles
16. Before implementing the additional functionality provided by our decorator first checks that the decorator still supports the basic functionality. Modify the Constructor of SudokuForm to decorator the supplied board with the UndoBoardDecorator, and use a

© 2010 Digital Age Learning. All Rights Reserved

reference to the newly decorated object for the initialization of the BoardControl and solver. Compile and run the application it will continue to work as before

```
public partial class SudokuForm : Form
{
   private UndoRedoBoardDecorator board;
   private Solver solver;

   public SudokuForm(Board board)
   {
      this.board = new UndoRedoBoardDecorator(board);

      InitializeComponent();

      solver = new BasicSolver(this.board);

      boardView.Board = this.board;
   }
...
```

17. You now need to implement the correct functionality inside the UndoBoardDecorator. For that you need to

```
public class UndoRedoBoardDecorator : BoardDecorator
{
   private UndoRedoInvoker invoker;

   public UndoRedoBoardDecorator(Board board) : base(board)
   {
      invoker = new UndoRedoInvoker();
   }

   public override int this[int x, int y]
   {
      get
      {
            // no need to wrap this in a command
            // has it does not change state.
            return base[x, y];
      }
      set
      {
            SetCellCommand cmd = new SetCellCommand(wrappedBoard, x, y,
   value);

            invoker.Execute(cmd);
      }
   }

   public override void MarkPossibleValue(int x, int y, int val)
   {
      MarkPossibleValueCommand cmd = new
   MarkPossibleValueCommand(wrappedBoard, x, y, val);
```

© 2010 Digital Age Learning. All Rights Reserved

```
      invoker.Execute(cmd);
   }

   public override void UnMarkPossibleValue(int x, int y, int val)
   {
      UnMarkPossibleValueCommand cmd = new
   UnMarkPossibleValueCommand(wrappedBoard, x, y, val);

      invoker.Execute(cmd);
   }
}
```

18. Compile and rerun the application and verify you still have the original functionality.
19. Now for the next part you will need to implement the Undo functionality for each of the commands you have defined. In the case of the SetCell Command you simply need to have a field inside the command that is initialized to the current value of the cell prior to the command executing. The Undo logic then simply sets the cell to the previous value.

```
class SetCellCommand : Command
{
   private Board target;
   private int x;
   private int y;
   private int val;

   private int prevVal;

   public SetCellCommand(Board board , int x , int y , int val)
   {
      target = board;
      this.x = x;
      this.y = y;
      this.val = val;
   }

   public override void Execute()
   {
      prevVal = target[x, y];
      target[x, y] = val;
   }

   public override void Undo()
   {
      target[x, y] = prevVal;
   }
}
```

20. Do similar with the Mark and UnMark commands, note that you cannot assume that just because you have been asked to unmark does that mean that the value is currently marked. Users do the strangest things...

```
class MarkPossibleValueCommand : Command
{
```

© 2010 Digital Age Learning. All Rights Reserved

```
   private Board target;
   private int x;
   private int y;
   private int val;

   private bool? prevValue;

   public MarkPossibleValueCommand(Board board , int x , int y, int val)
   {
      target = board;
      this.x = x;
      this.y = y;
      this.val = val;
   }

   public override void Execute()
   {
      if (target.IsPossibleMarkedValue(x, y, val) == false)
      {
            prevValue = false;
            target.MarkPossibleValue(x, y, val);
      }
   }

   public override void Undo()
   {
      if (prevValue != null)
      {
            target.UnMarkPossibleValue(x, y, val);
      }
   }
}
```

21. Now add two additional methods to the UndoBoardDecorator, one called Undo and the other Redo. Make each of these methods call the appropriate Undo and Redo methods on the UndoInvoker instance. Finally inside the SudokuForm class change the Undo and Redo handlers to call the Undo and Redo methods on the board. Job Done..Well almost
22. You may have noticed that when you set a square to a value the possible marked values are lost, undoing the setting of the square simply reverts back to an empty cell and no marked possible values, that's because the implementation of the board clears the set of possible values when you decide on the actual value. To correctly support undo, you will need to extend the SetCell Command to record the set of marked possible values and then restore them on execution of the undo command.

```
class SetCellCommand : Command
{
   private Board target;
   private int x;
   private int y;
   private int val;

   private int prevVal;
```

© 2010 Digital Age Learning. All Rights Reserved

```
   private List<int> possibleMarkedValues;

   public SetCellCommand(Board board , int x , int y , int val)
   {
      target = board;
      this.x = x;
      this.y = y;
      this.val = val;
   }

   public override void Execute()
   {
      prevVal = target[x, y];

      if (prevVal == Board.EMPTY_CELL)
      {
            possibleMarkedValues = new
   List<int>(target.MarkedPossibleValues(x, y));
      }

      target[x, y] = val;
   }

   public override void Undo()
   {
      target[x, y] = prevVal;

      if (prevVal == 0)
      {
            foreach (int possibleValue in possibleMarkedValues)
            {
                  target.MarkPossibleValue(x, y, possibleValue);
            }
      }
   }
}
```

23. Now Job done...Through the use of the command pattern you have created a general purpose undo/redo solution for any scenario that can be broken down into a series of commands which have both a forward and reverse action. Through the use of the decorator pattern you have created Undo/Redo functionality that can be applied to any implementation of a Sudoku board.
24.

© 2010 Digital Age Learning. All Rights Reserved