1/16/2022

**saugat thapa**

**77227277**

**GRAPHICAL PROGRAMMING LANGUAGE**

COMPONENT 1 AND 2

Contents

[COMPONENT 1 2](#_Toc93083957)

[The Interface 2](#_Toc93083958)

[MOVETO 2](#_Toc93083959)

[DRAWTO 3](#_Toc93083960)

[RESET 3](#_Toc93083961)

[FILE MENU 4](#_Toc93083962)

[RECTANGLE 6](#_Toc93083963)

[CIRCLE 6](#_Toc93083964)

[TRIANGLE 7](#_Toc93083965)

[PEN COLOR AND FILL 7](#_Toc93083966)

[COMPONENT 2 8](#_Toc93083967)

[COMMITS 8](#_Toc93083968)

[VARIABLES 9](#_Toc93083969)

[LOOP 10](#_Toc93083970)

[IF STATEMENT with ENDIF block 10](#_Toc93083971)

[Syntax Checking Before the program is run 11](#_Toc93083972)

[EXECUTE button reenabled once all errors are fixed 11](#_Toc93083973)

[METHODS 12](#_Toc93083974)

[FLASHING COLOR 12](#_Toc93083975)

[Use of FACTORY DESIGN PATTERN 13](#_Toc93083976)

[Use of EXCEPTION handling 14](#_Toc93083977)

[Use of user generated exceptions 14](#_Toc93083978)

[ADDITIONAL FUNCTIONALITY 15](#_Toc93083979)

[Complex Shape (POLYGON) 15](#_Toc93083980)

[Testing 16](#_Toc93083981)

[ADDITIONAL DESIGN PATTERNS 17](#_Toc93083982)

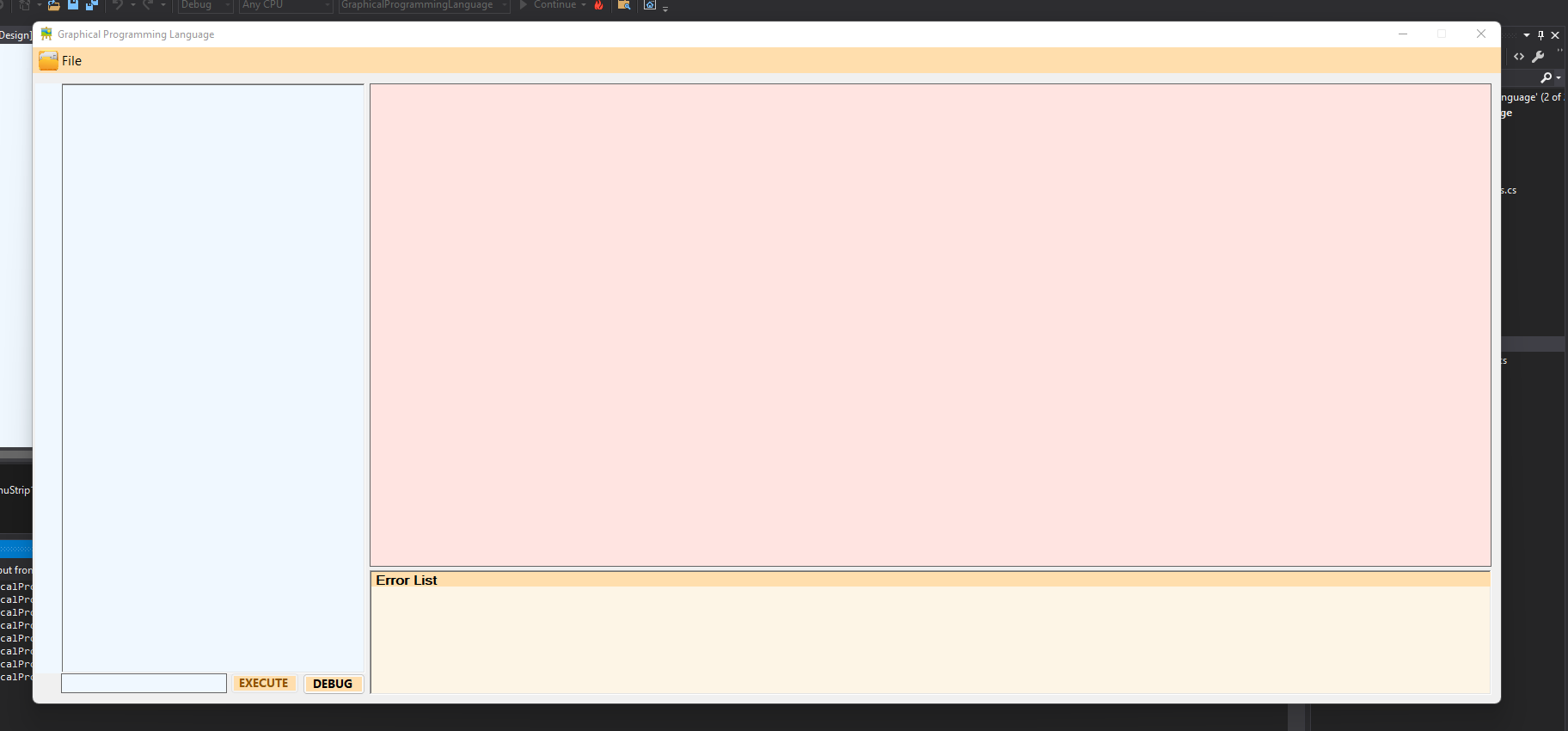
[Façade Design Pattern 17](#_Toc93083983)

[Command Design Pattern 19](#_Toc93083984)

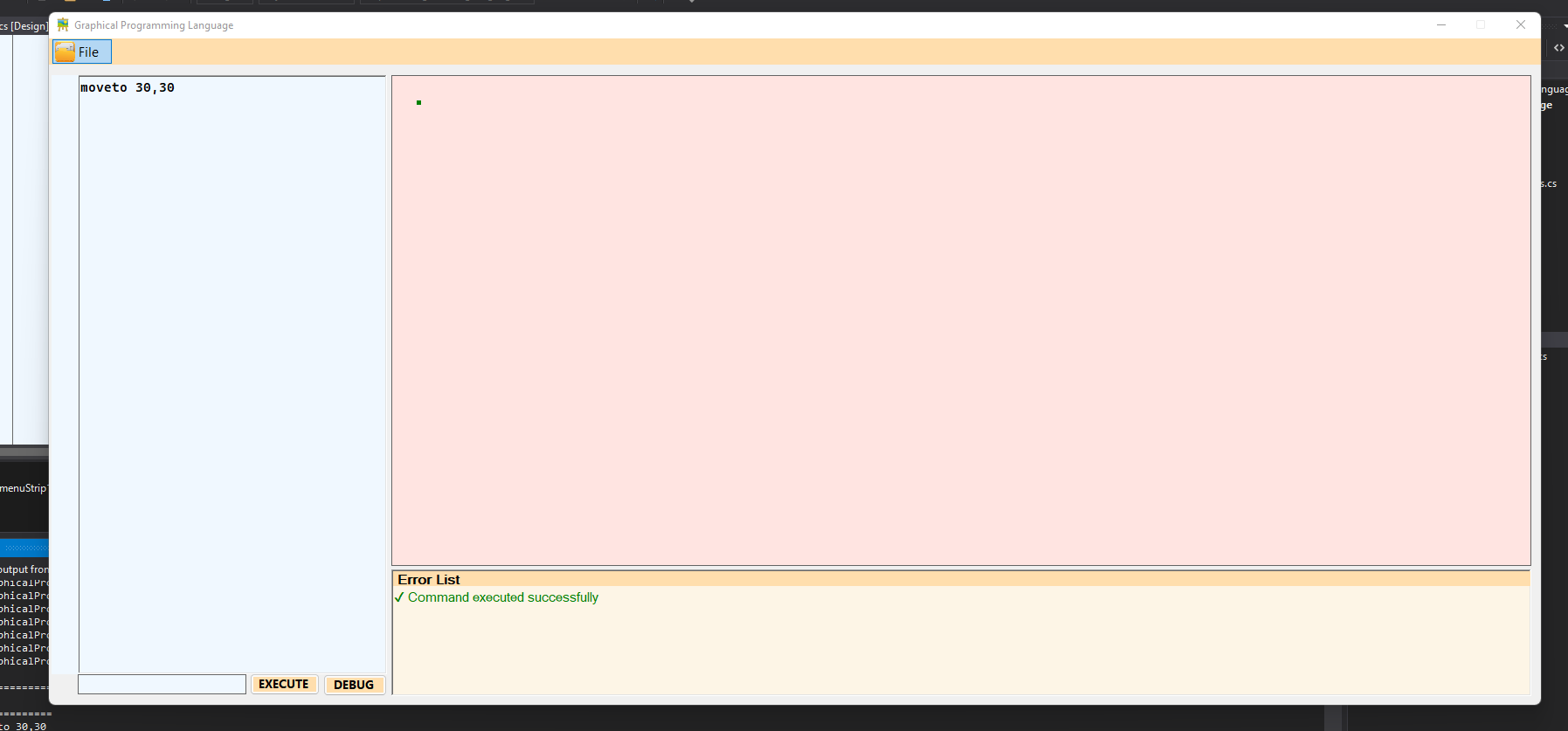
[APPENDIX (source code according to class) 22](#_Toc93083985)

# COMPONENT 1

## The Interface

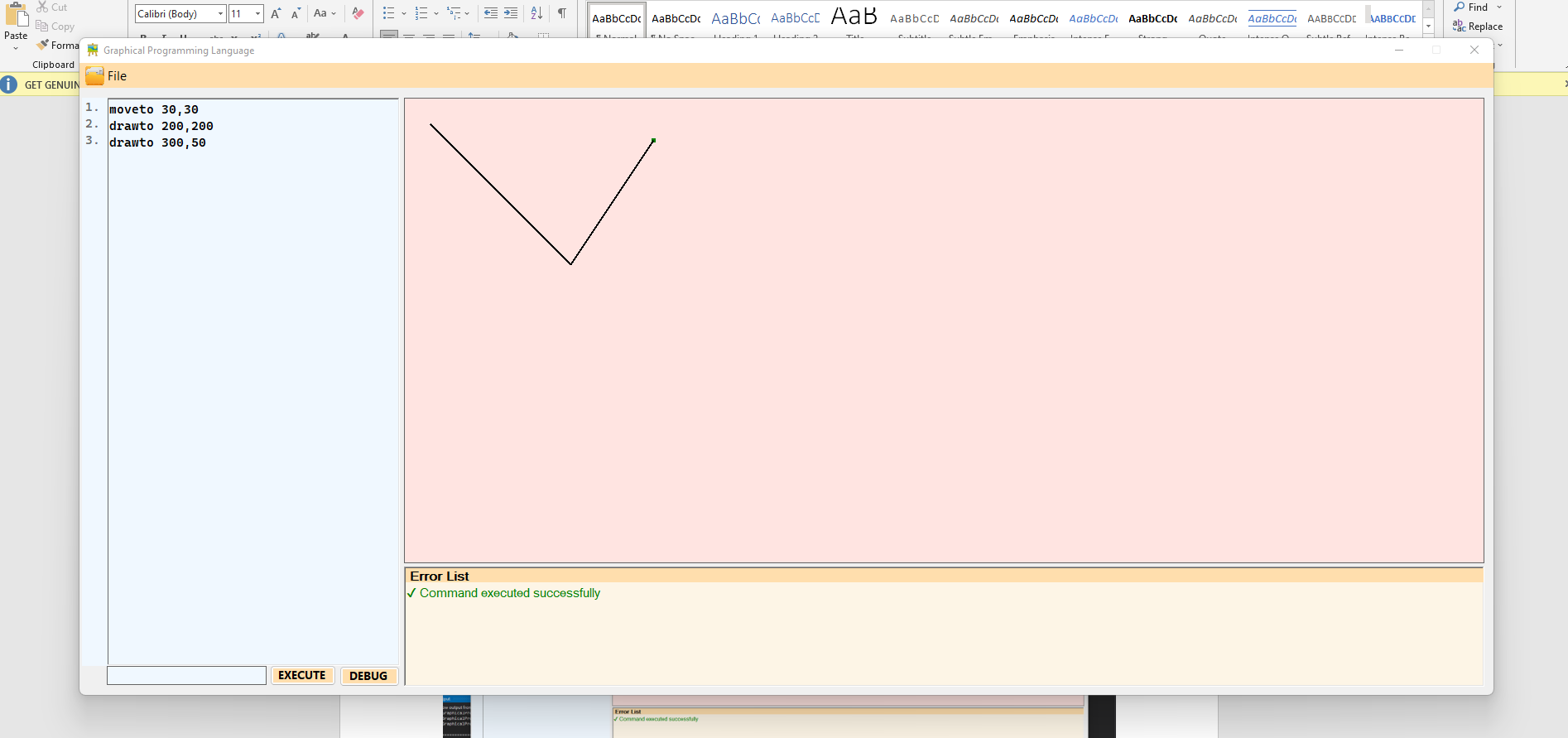


## MOVETO



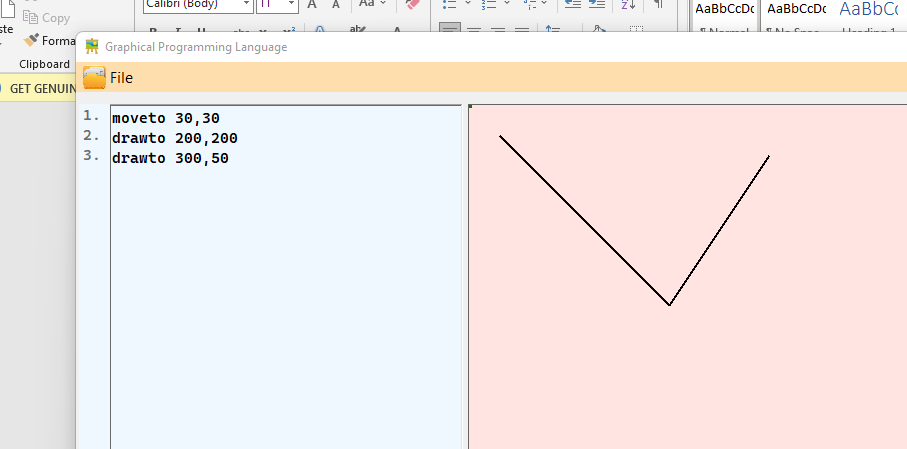
## DRAWTO

Draws line from previous MOVETO or DRAWTO position



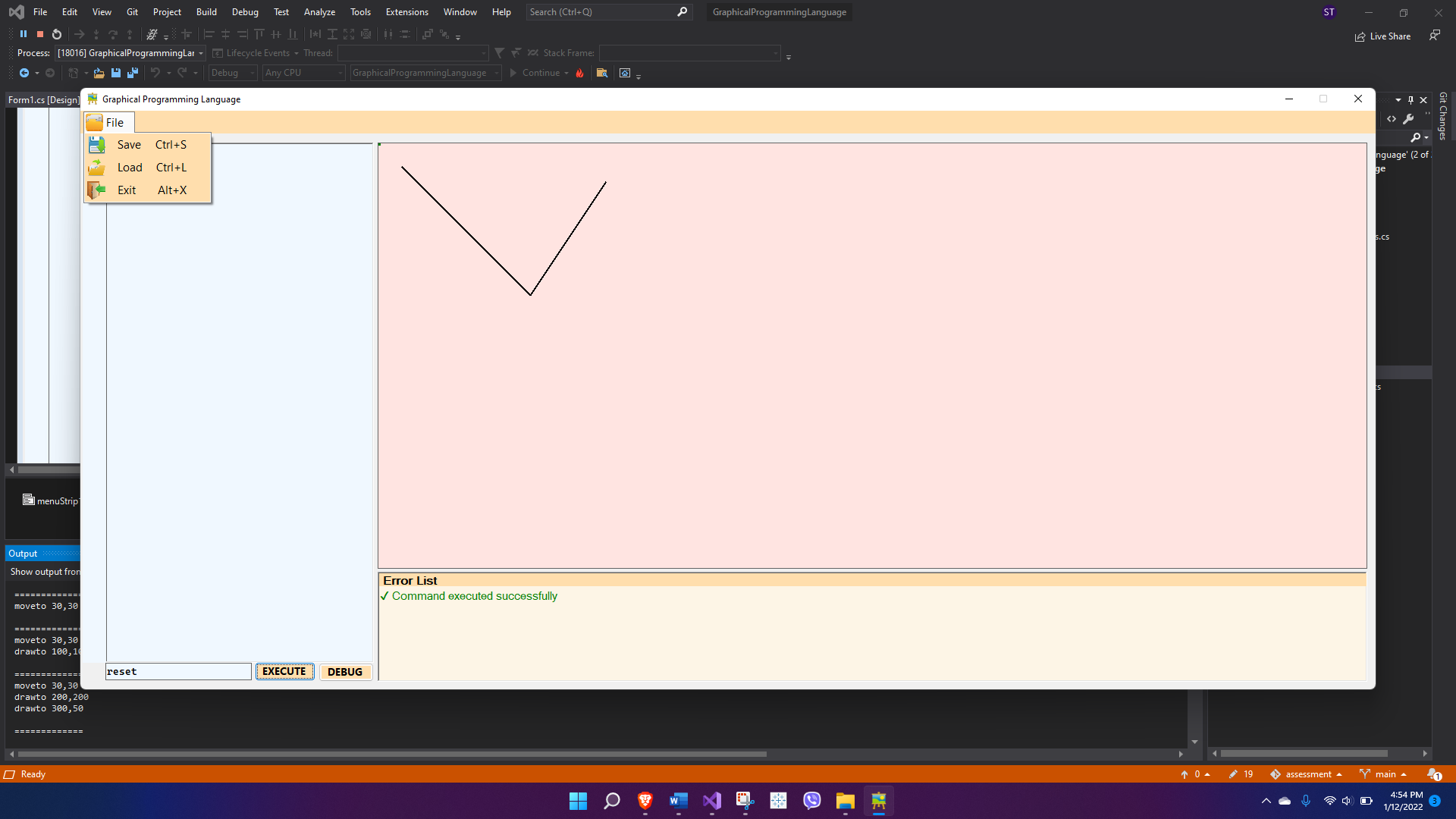
## RESET

Moves the MOVETO object to top left corner of the screen

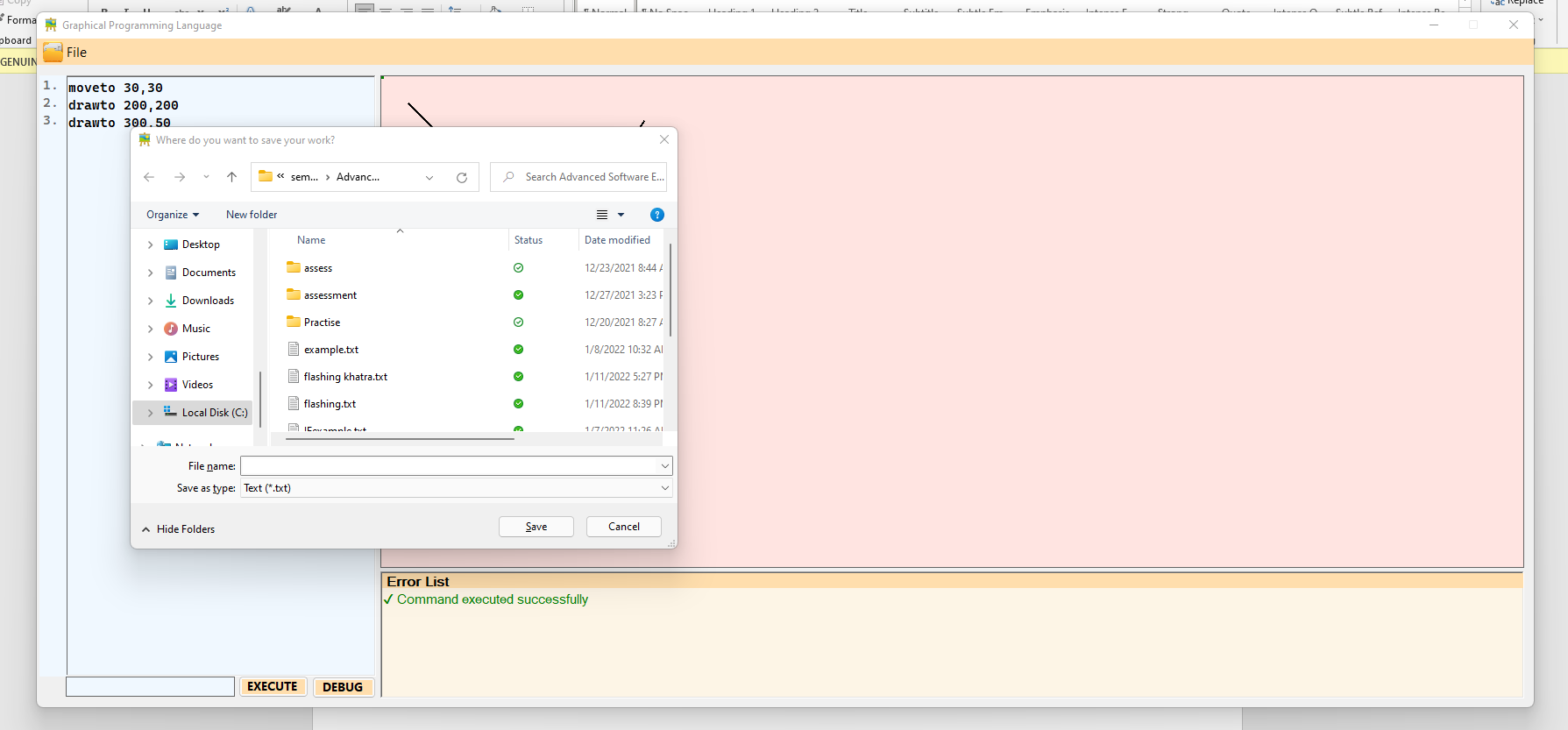


## FILE MENU

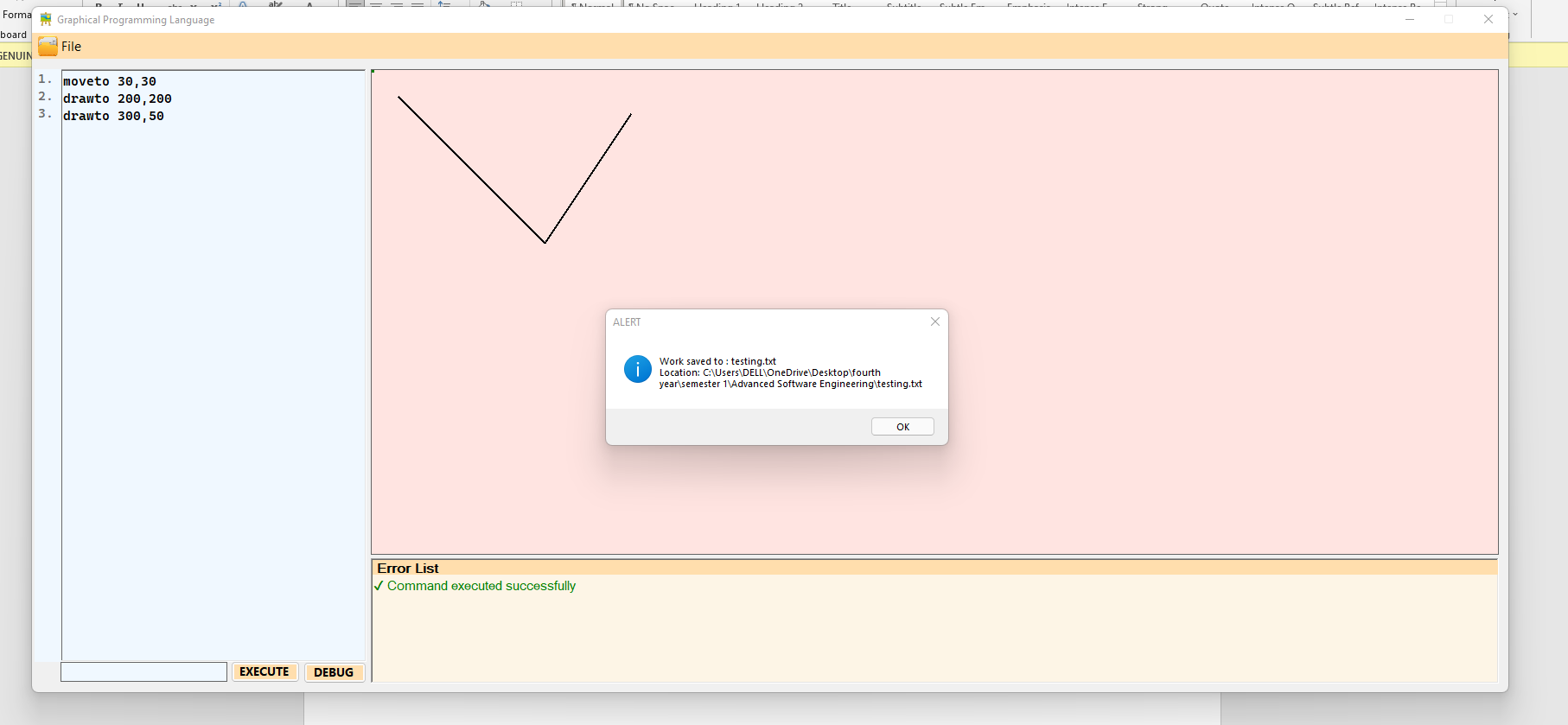
To SAVE, LOAD and also EXIT the application. Also includes shortcuts for all items. If a file is already loaded on the app, we can save it directly to the same file name in the same location just by Clicking the SAVE menu item. But if no file is loaded , on clicking save it will give u the option to save the file as a new file.

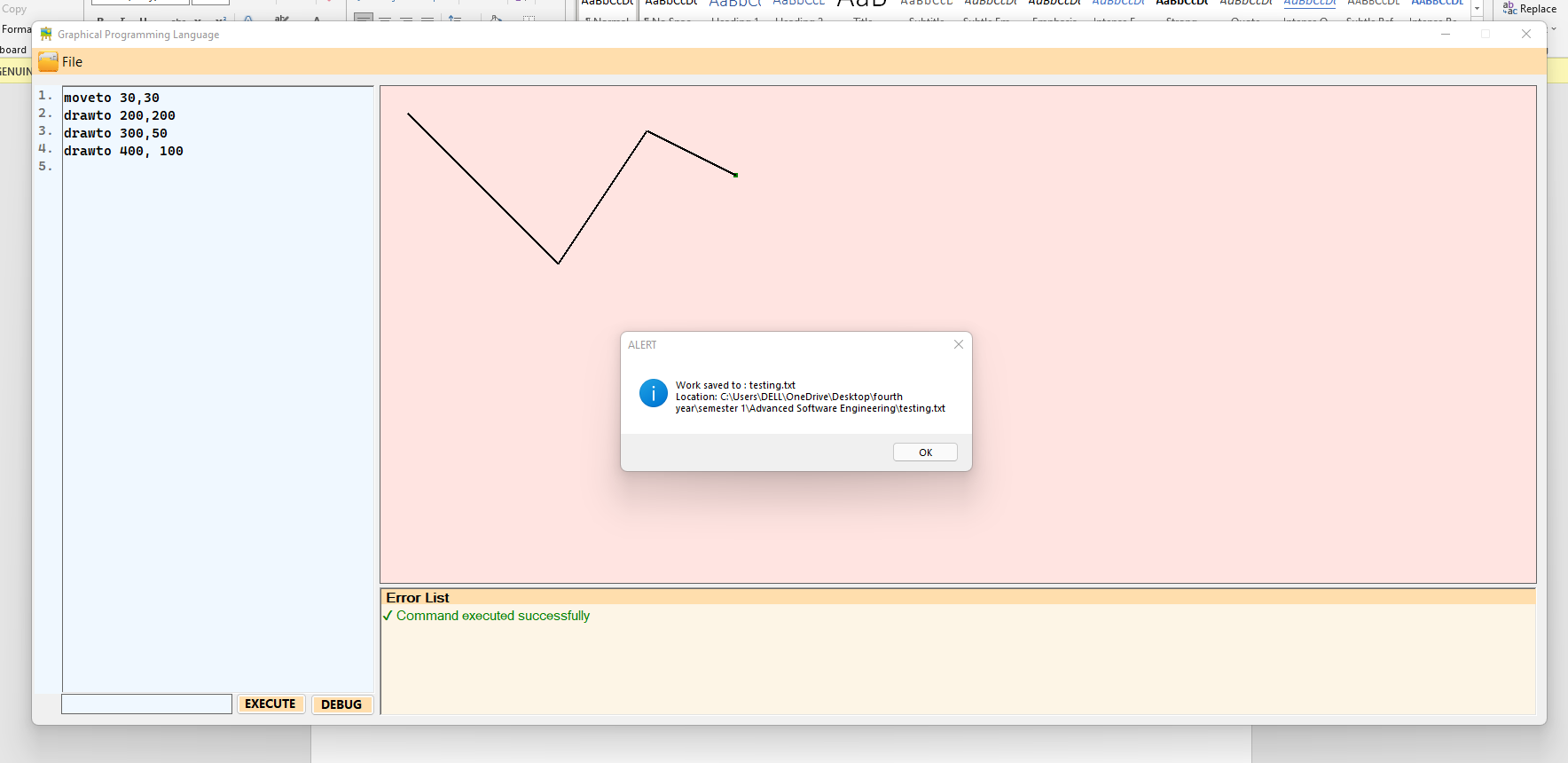


Saving a new file:

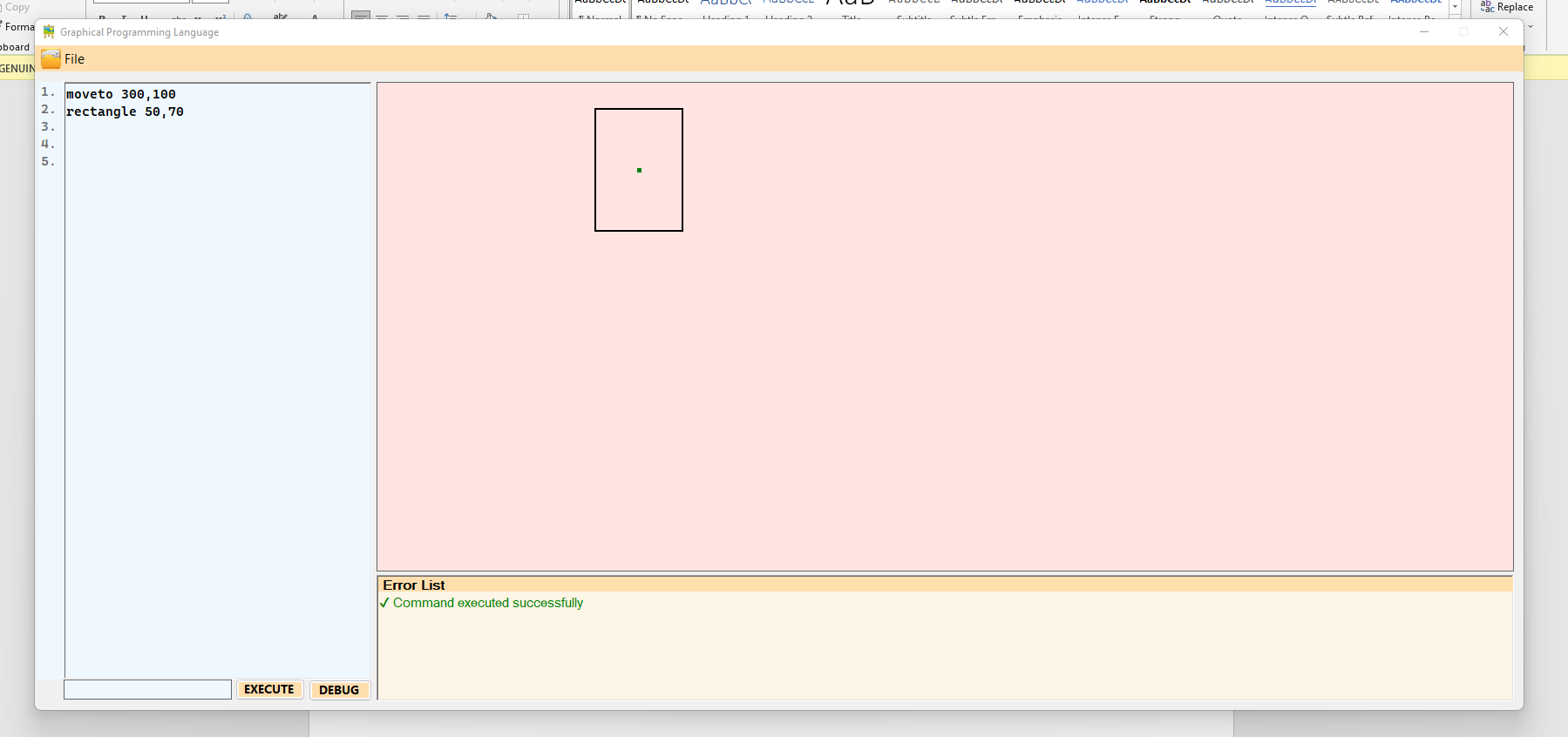


File saved message along with the location

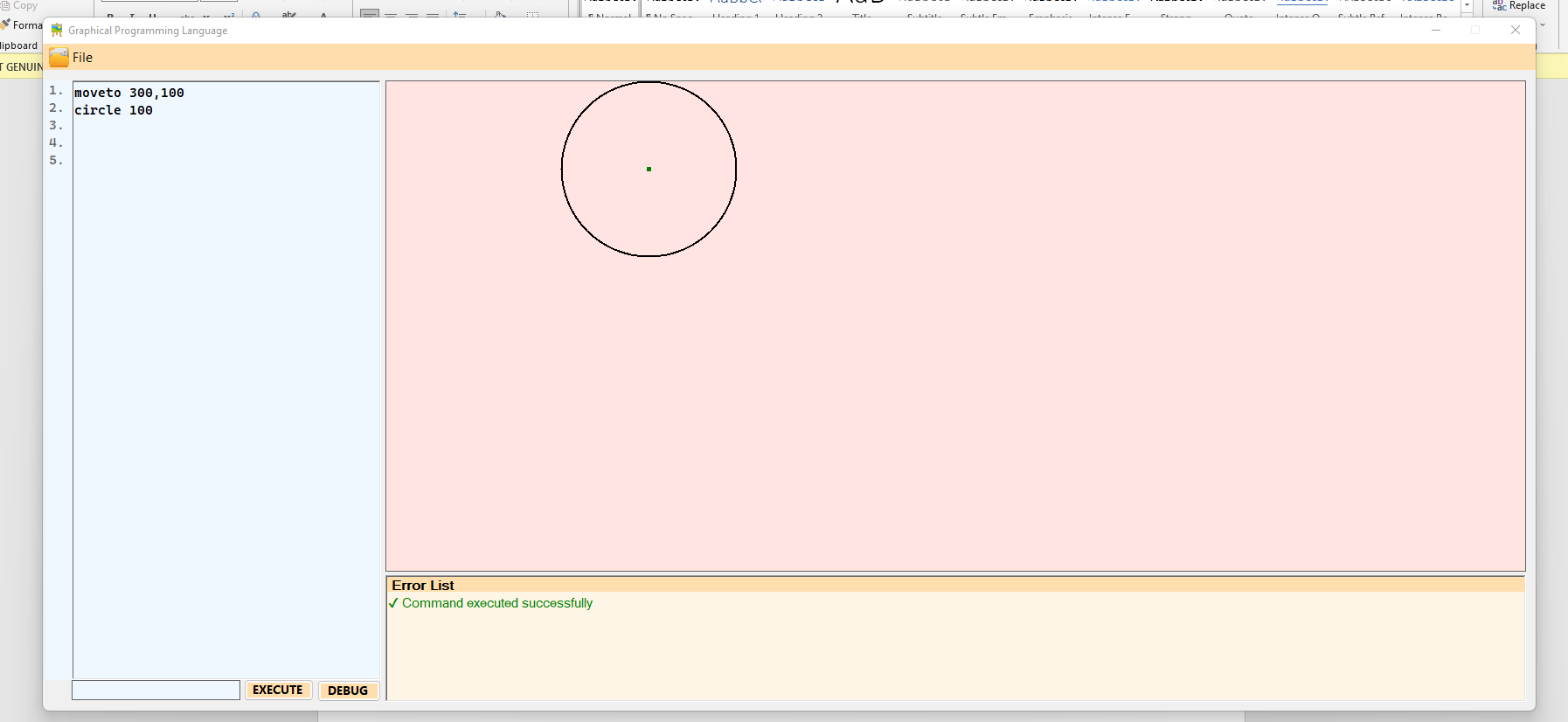


Saving something new on a preloaded file

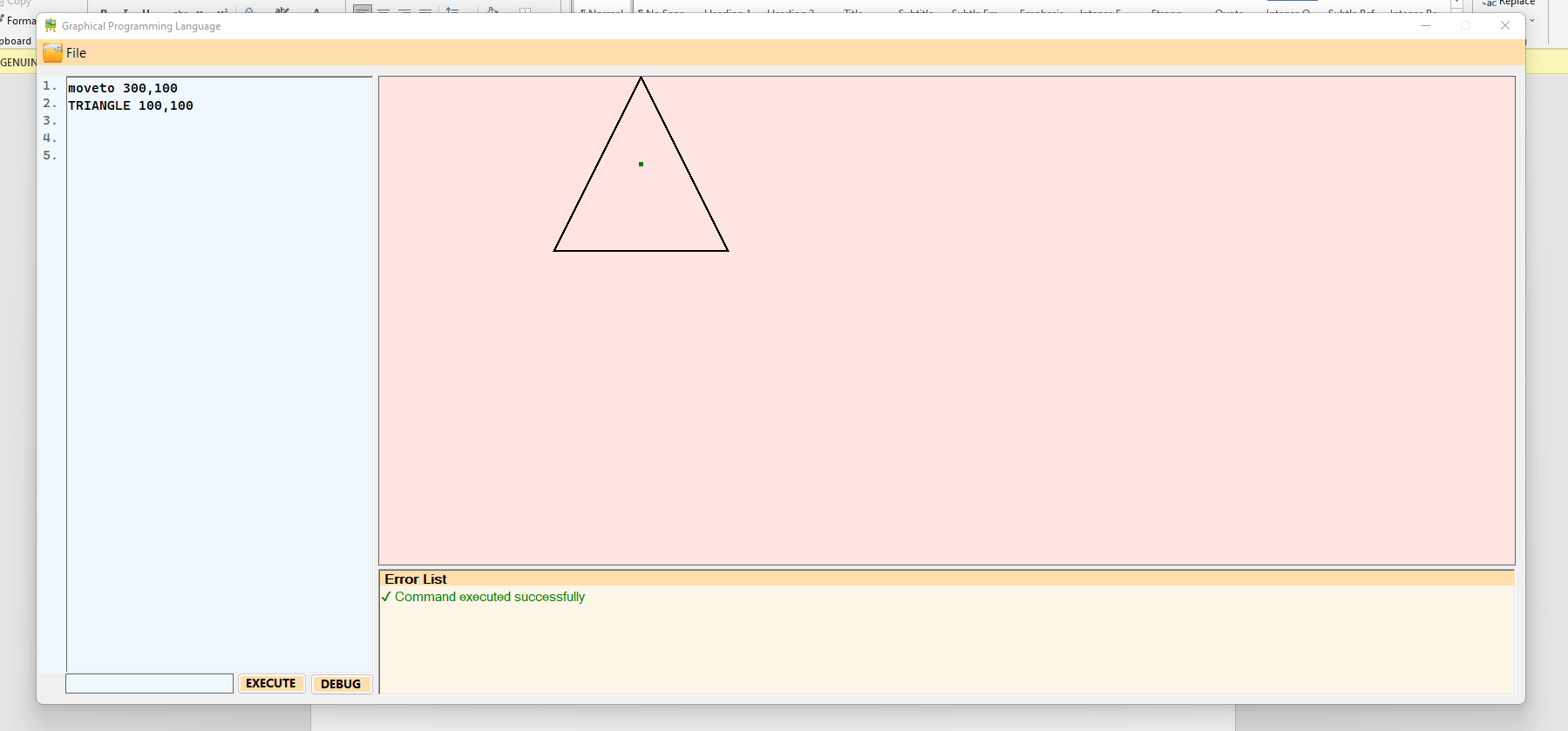
## RECTANGLE



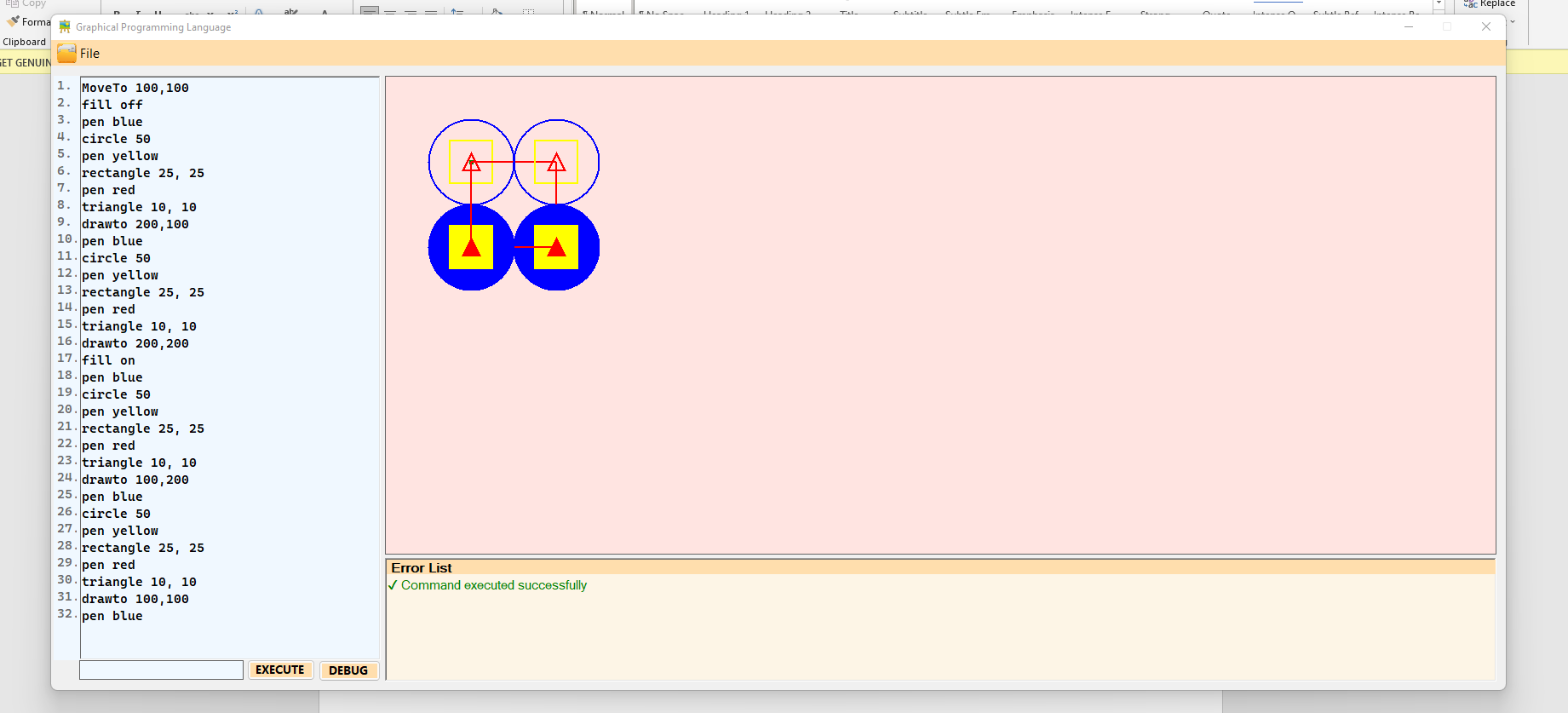
## CIRCLE



## TRIANGLE

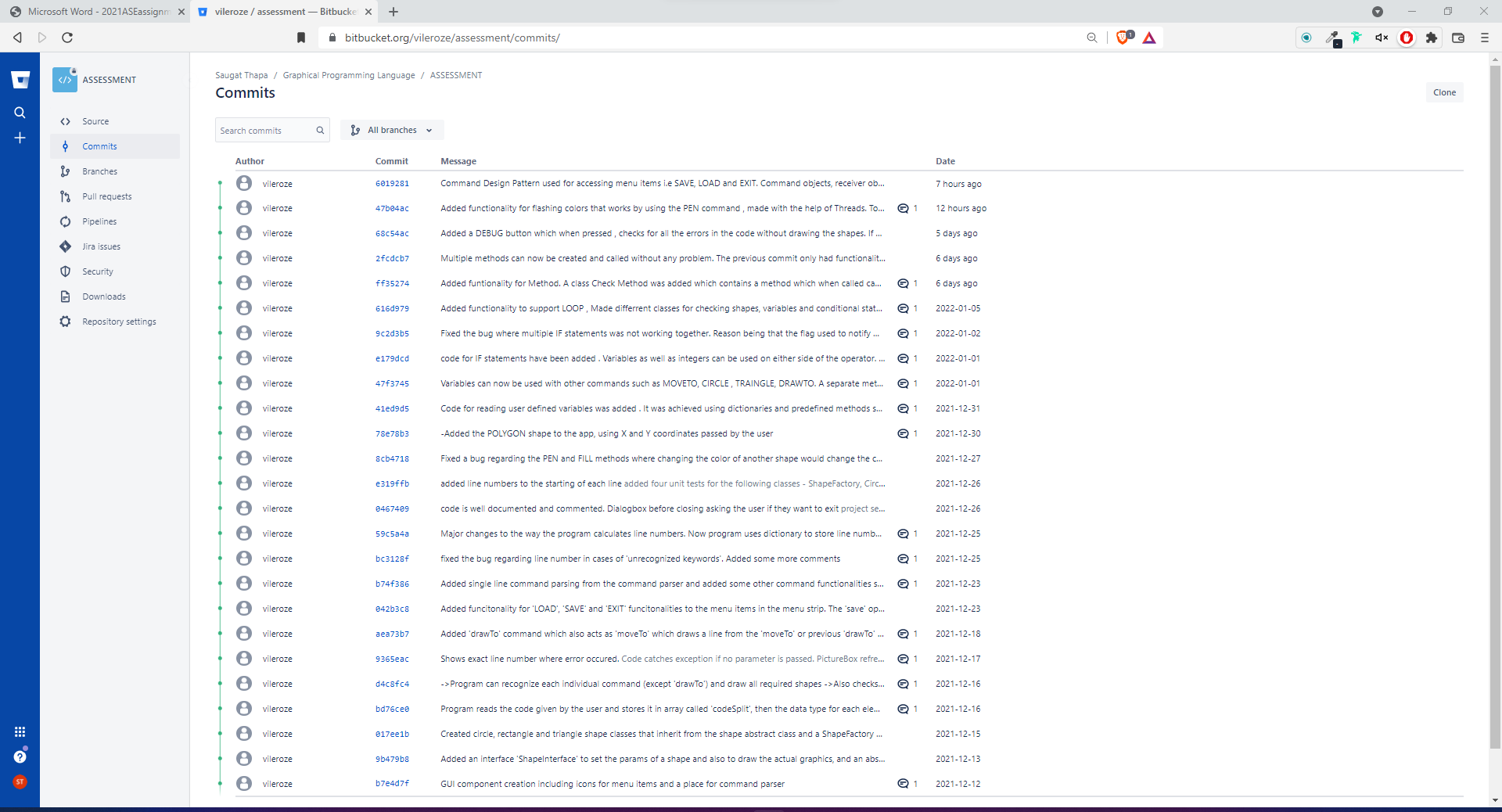


## PEN COLOR AND FILL

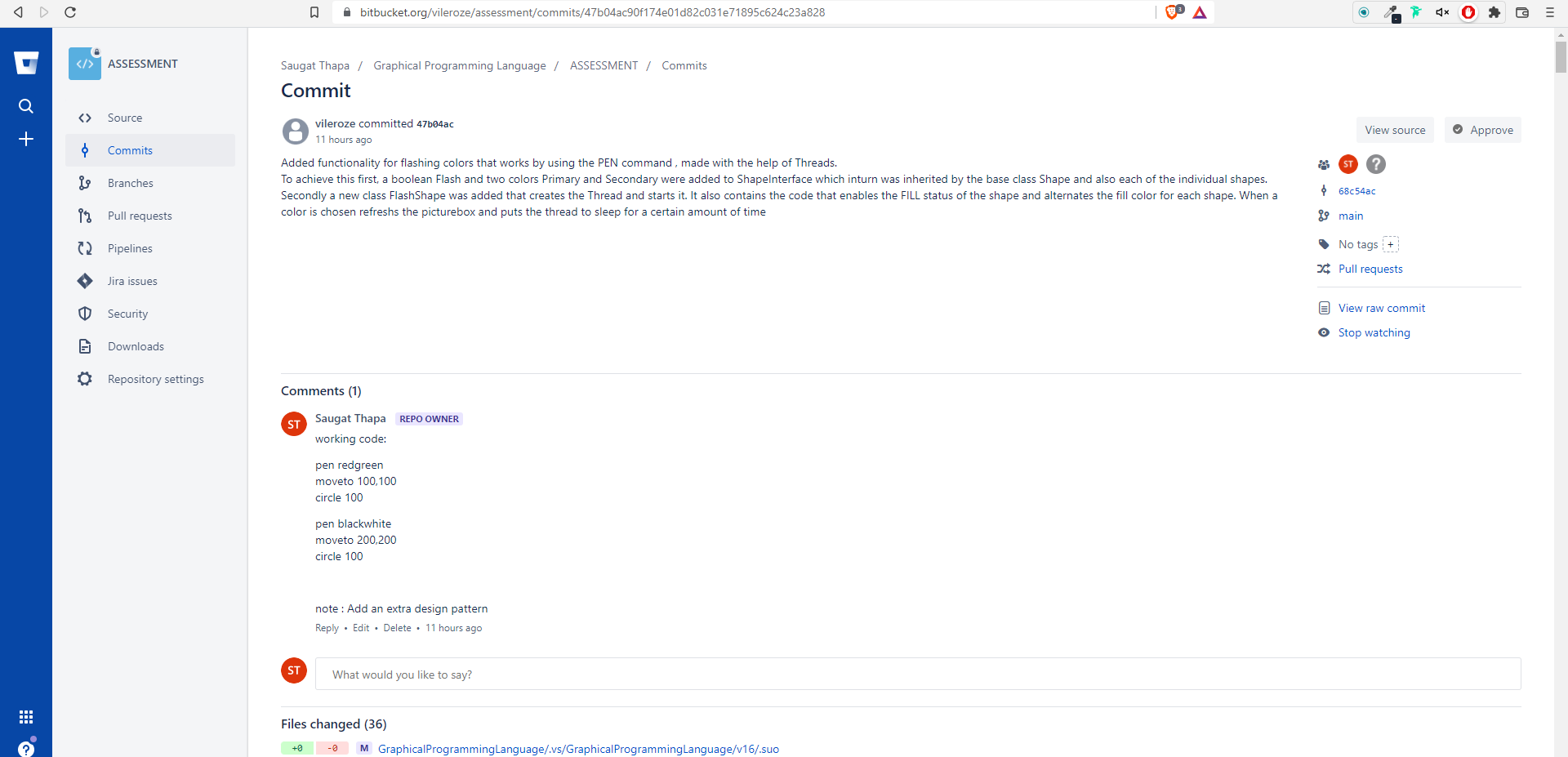


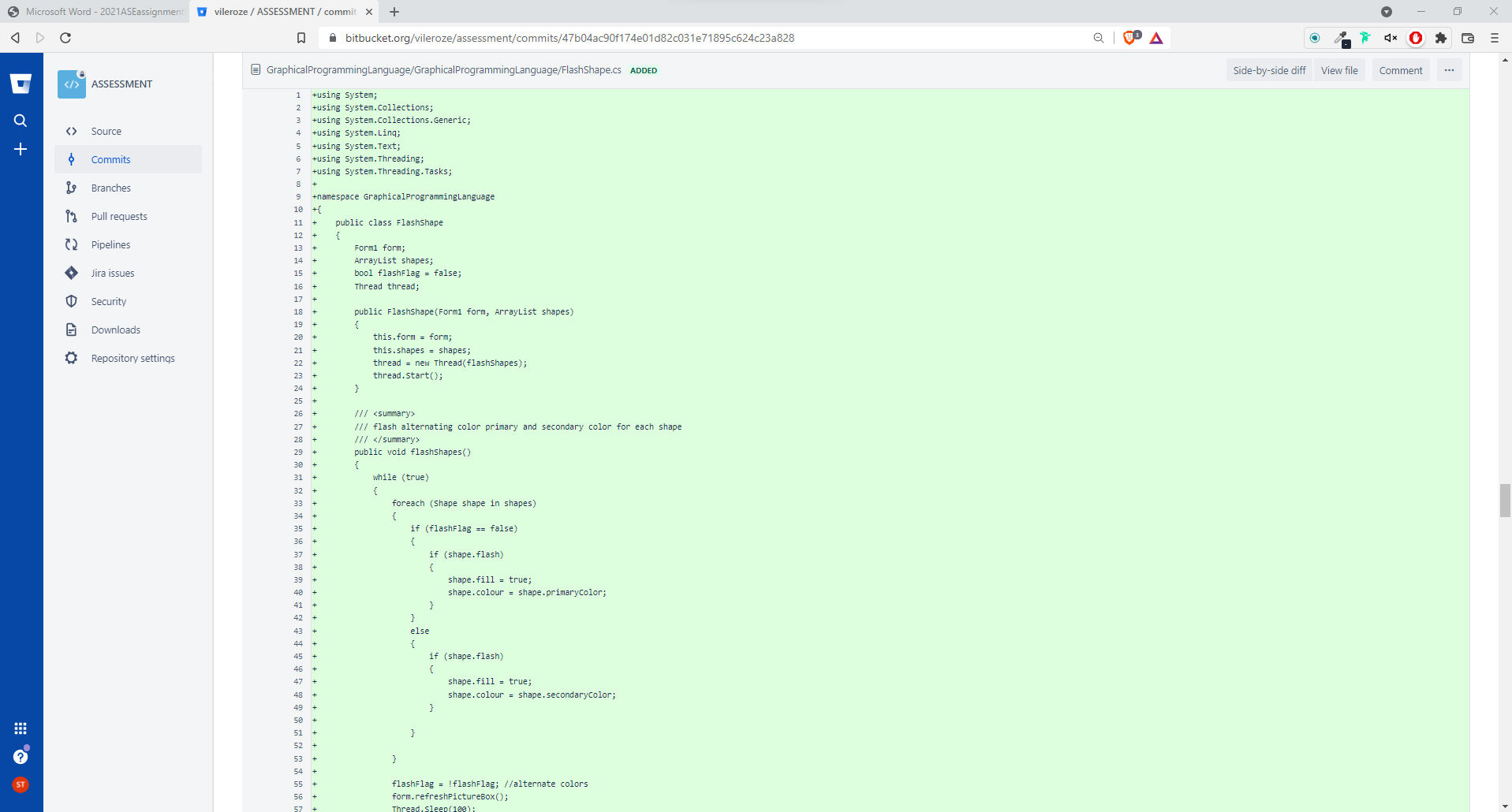
# COMPONENT 2

## COMMITS

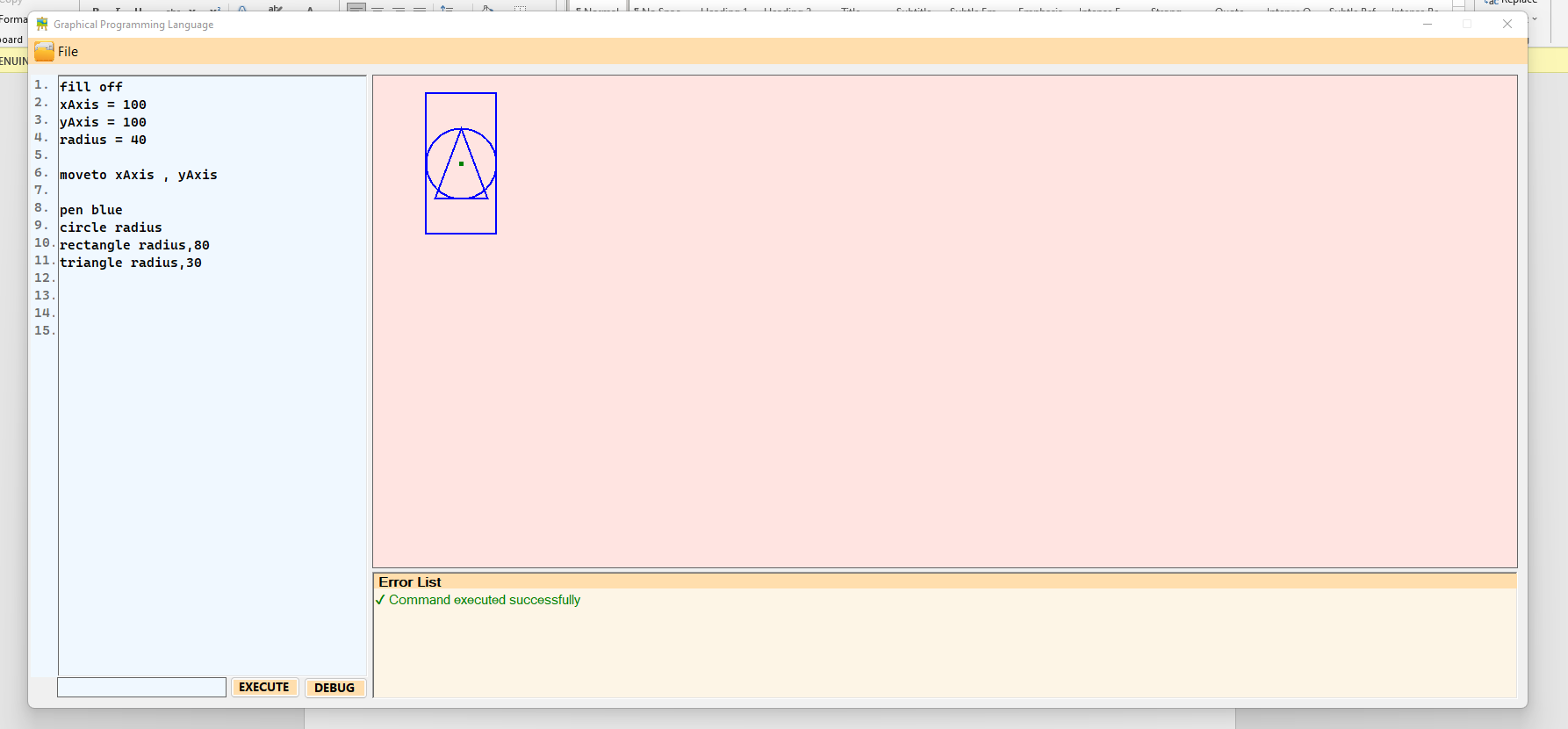


Individual Commit:

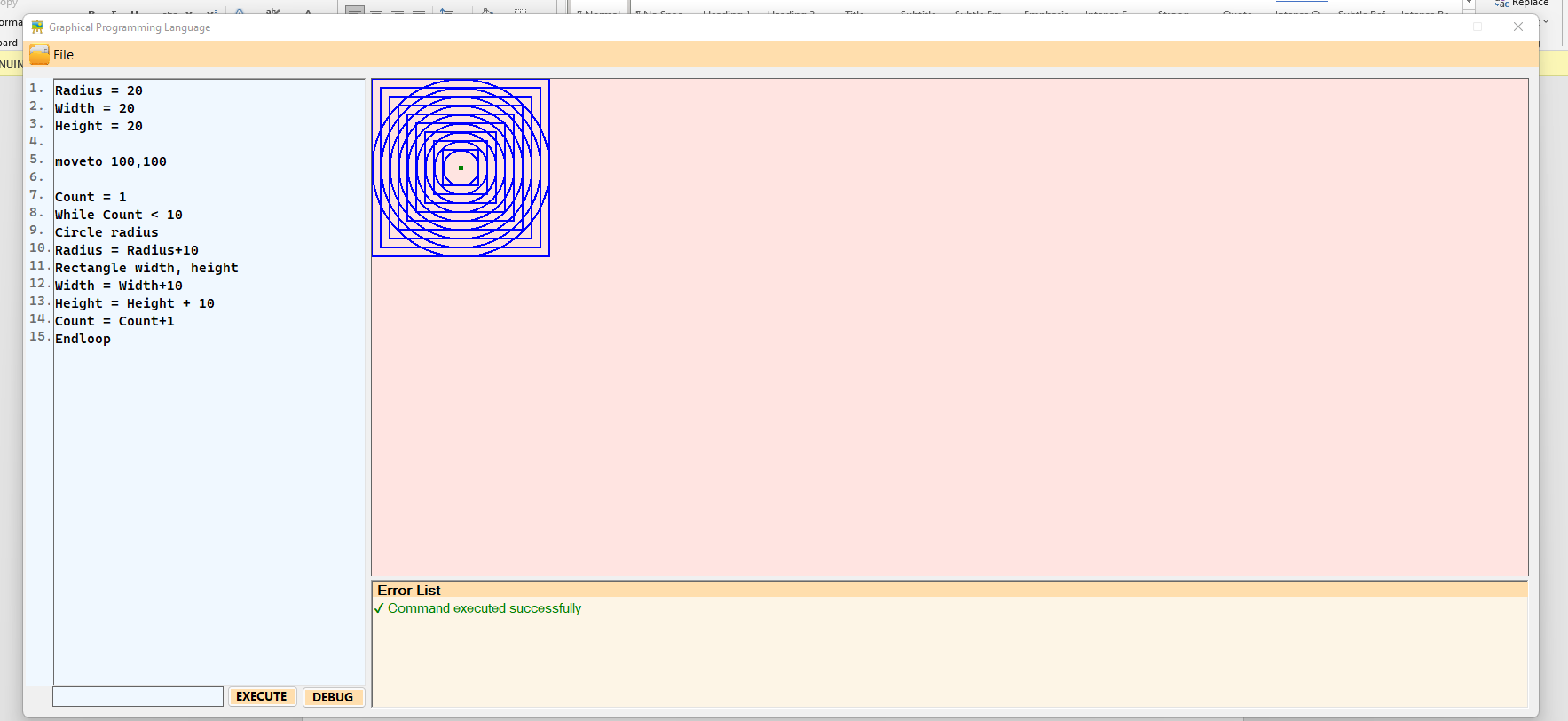




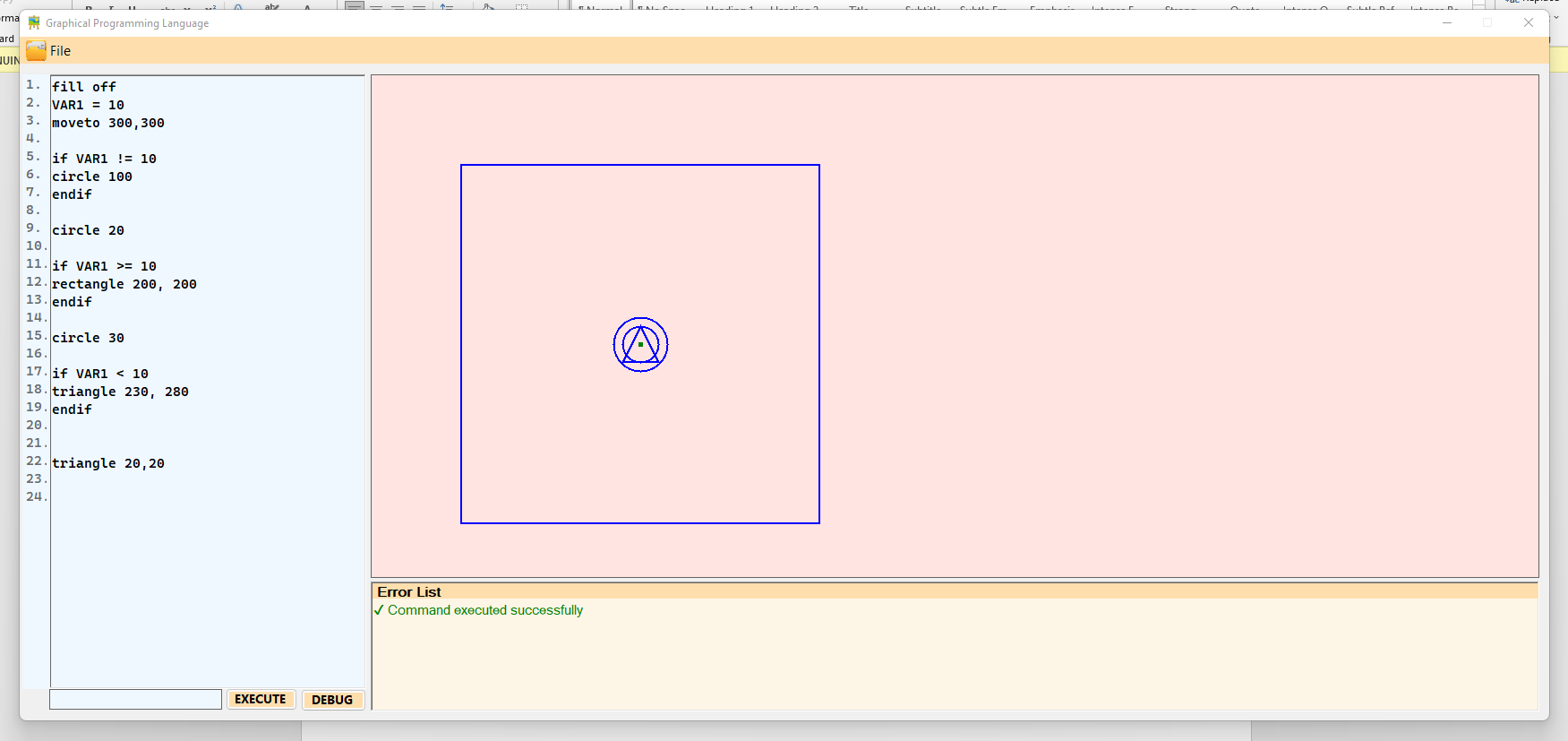
## VARIABLES



## LOOP

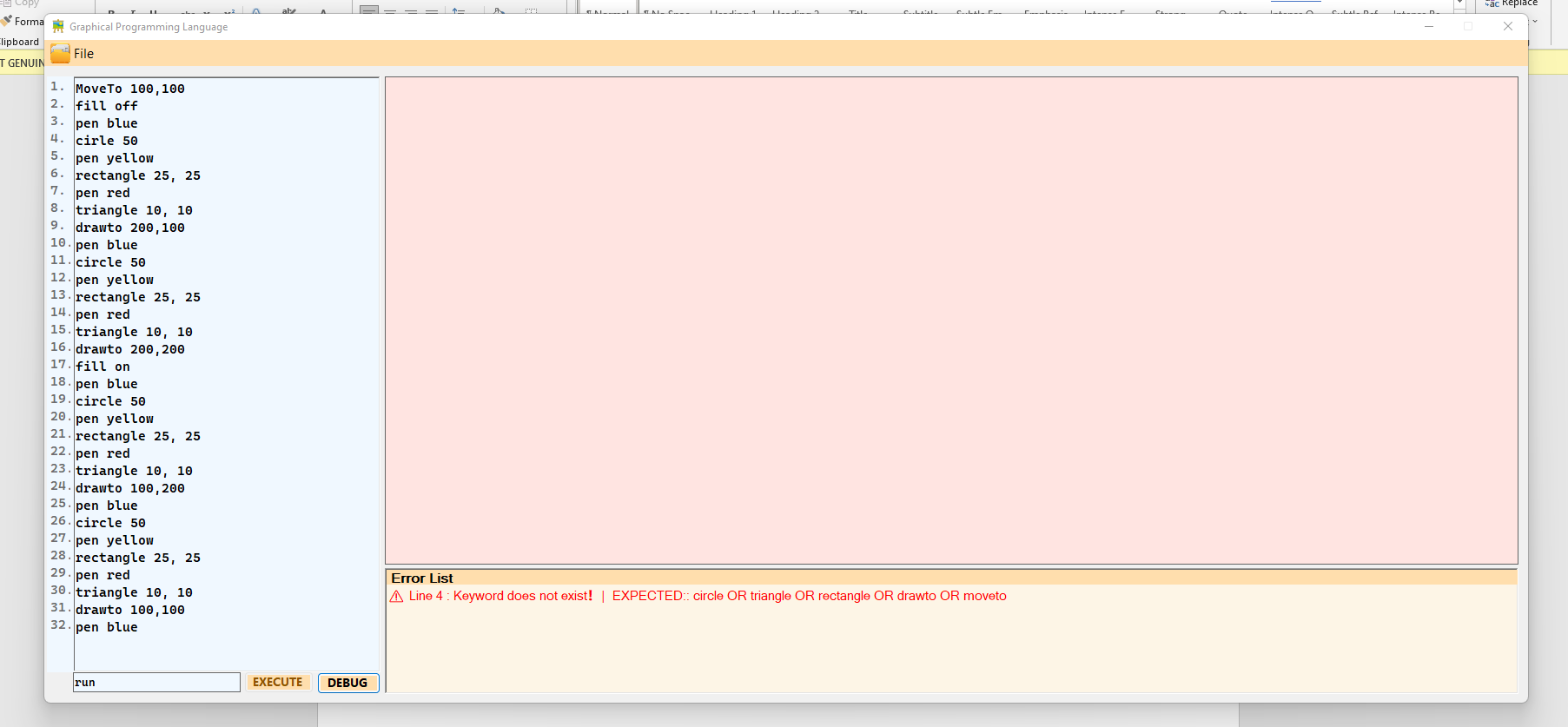


## IF STATEMENT with ENDIF block

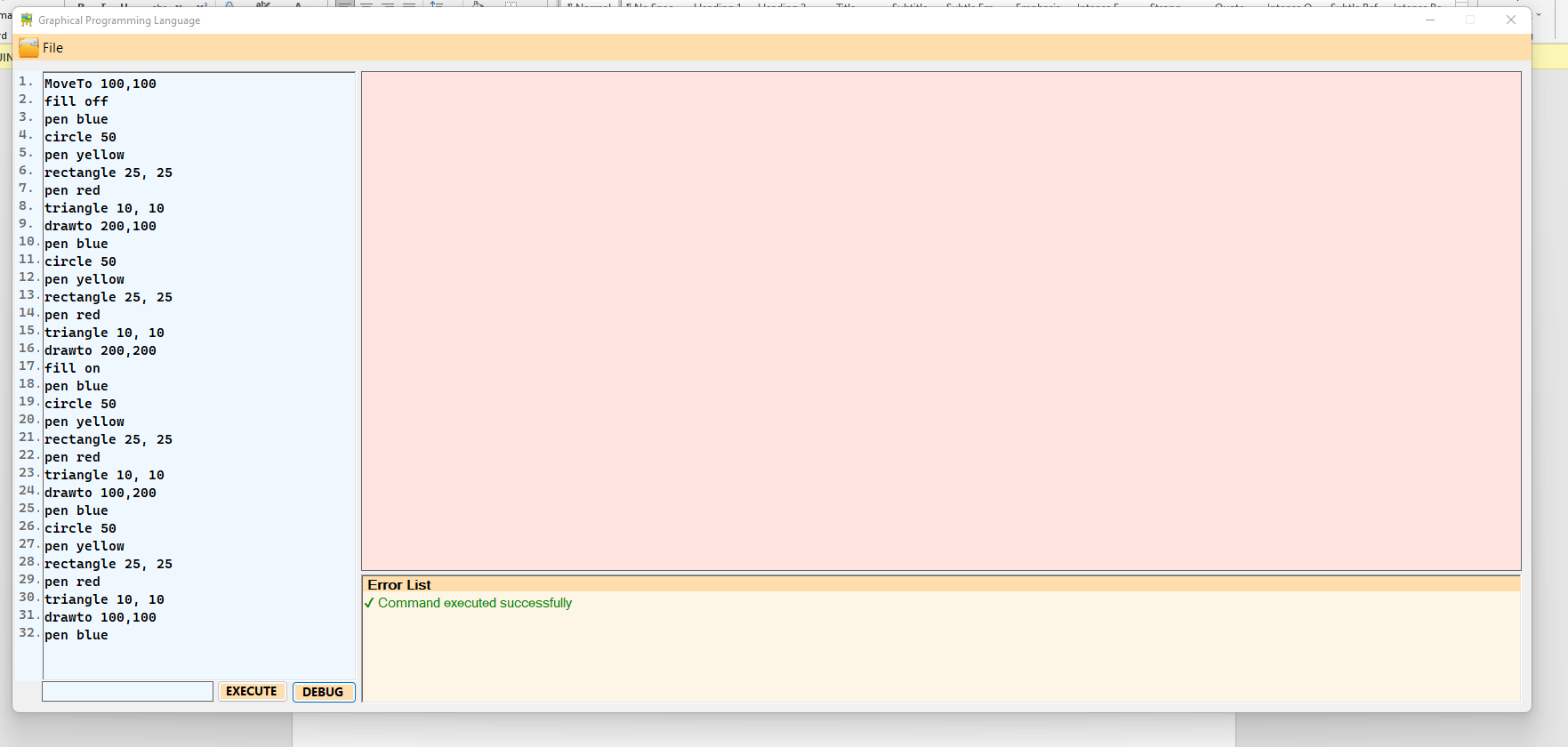


## Syntax Checking Before the program is run

As part of the requirement the EXECUTE button remains disabled until all the errors are solved, which can be viewed by pressing the DEBUG button. If an error is found later then the EXECUTE button is again disabled

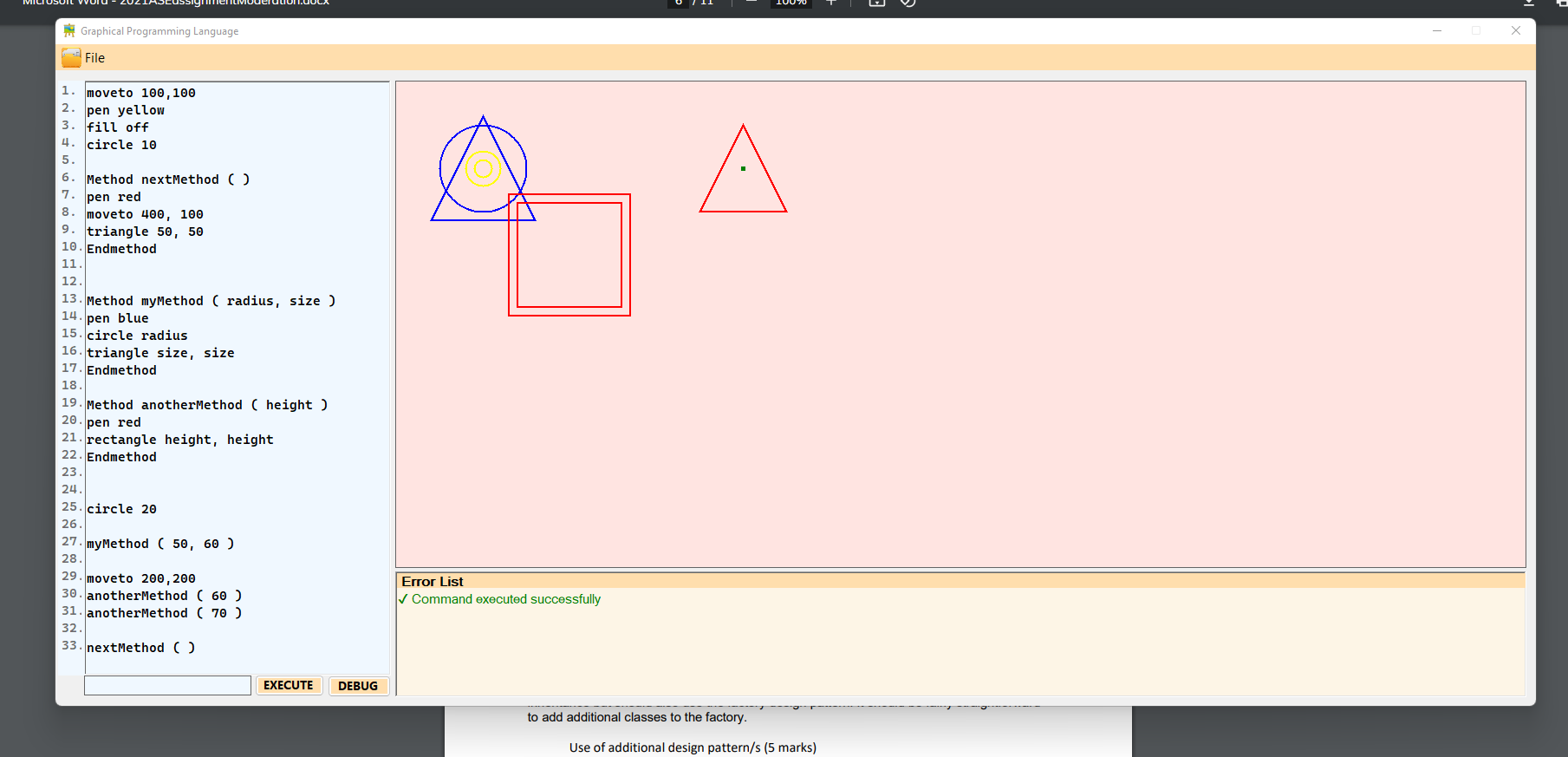


## EXECUTE button reenabled once all errors are fixed

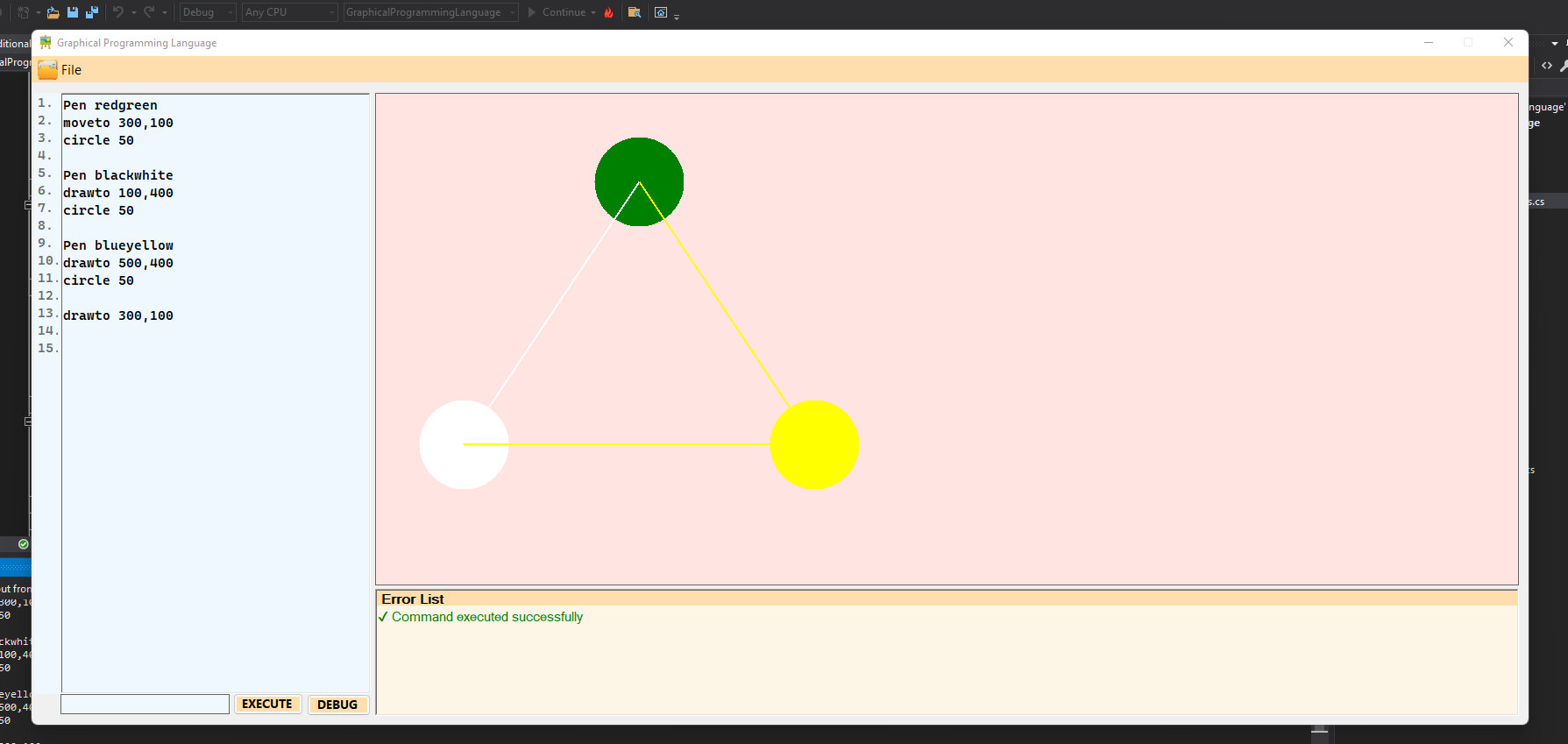


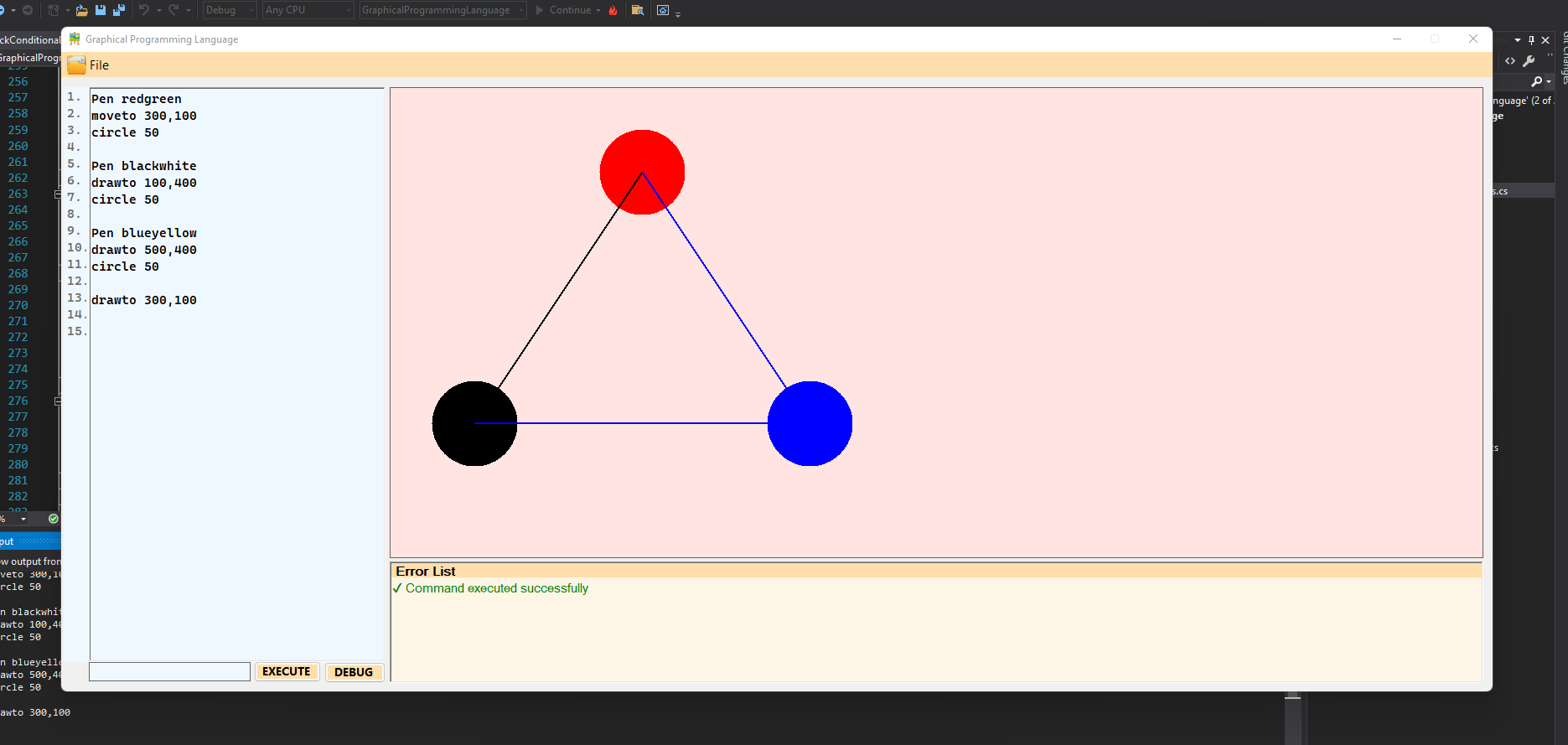
## METHODS

Works with parameters and without. Multiple calls can be made to the same method. Also works with multiple parameters.

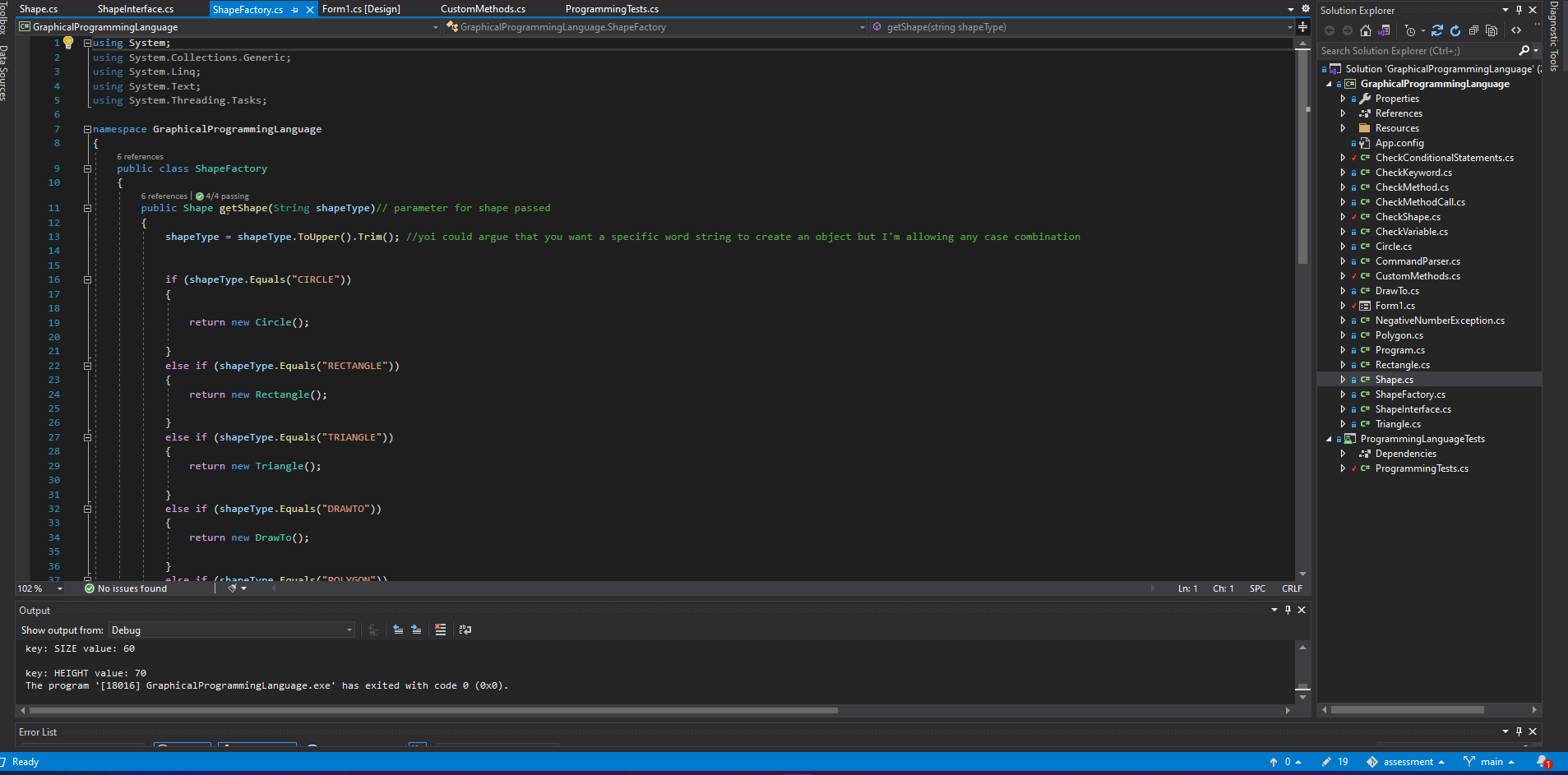


## FLASHING COLOR

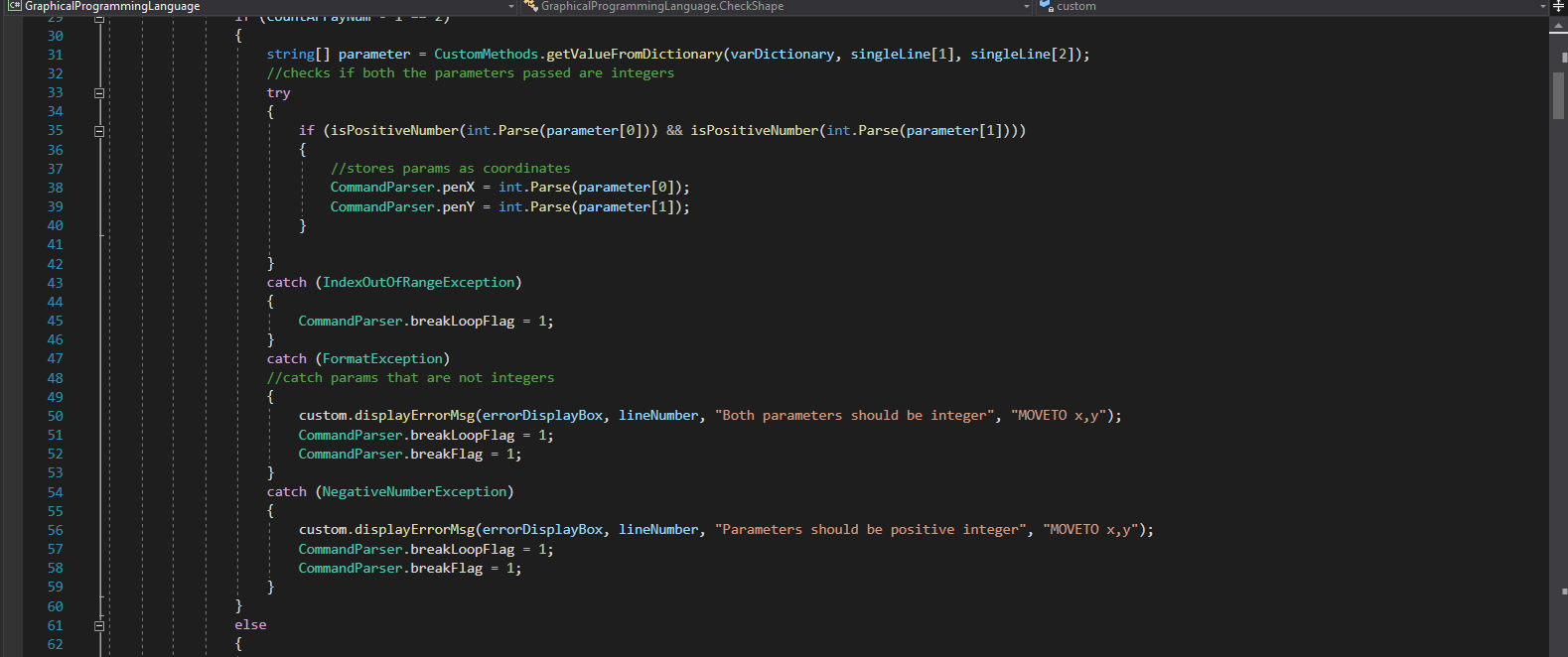




## Use of FACTORY DESIGN PATTERN

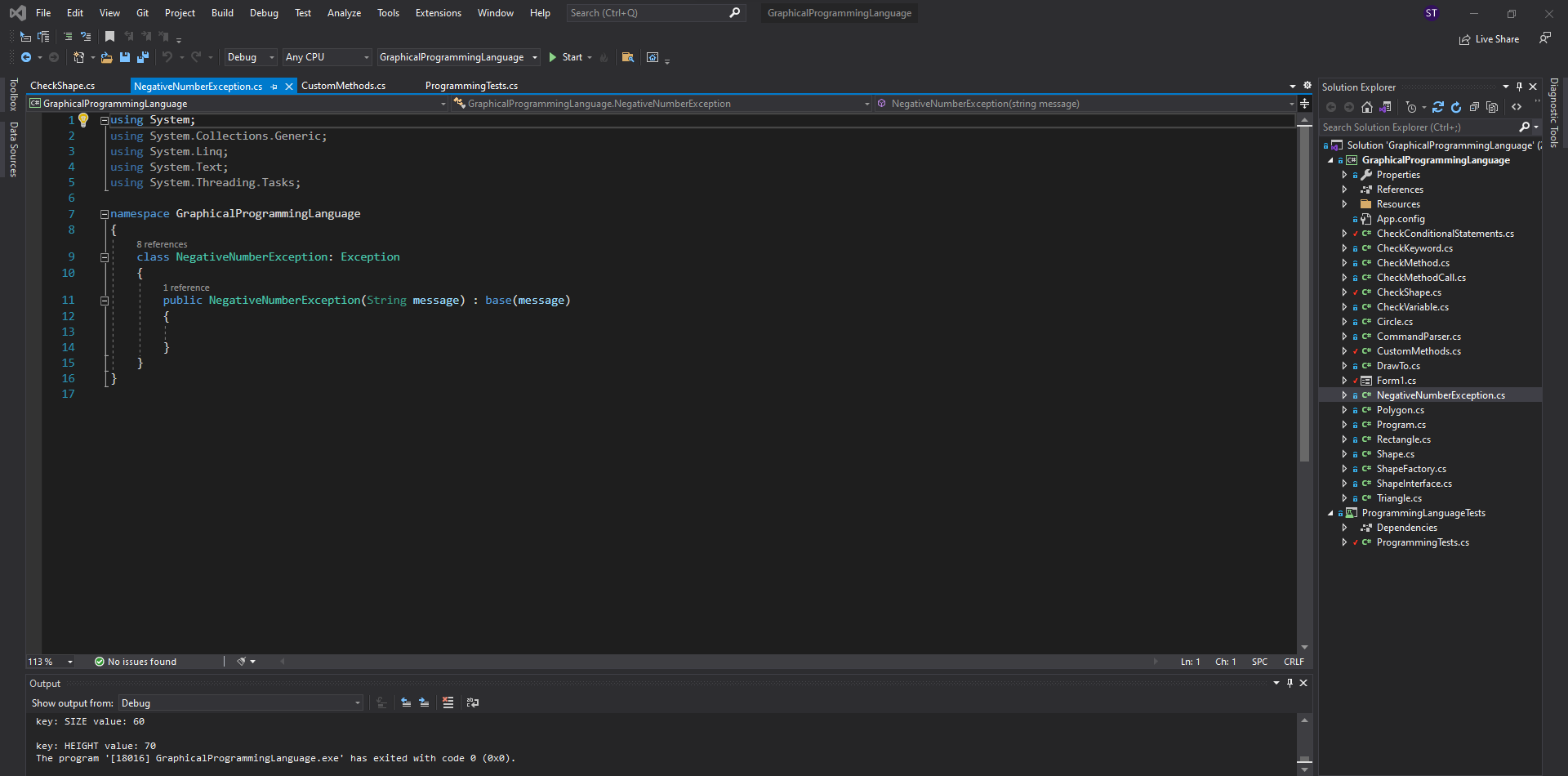


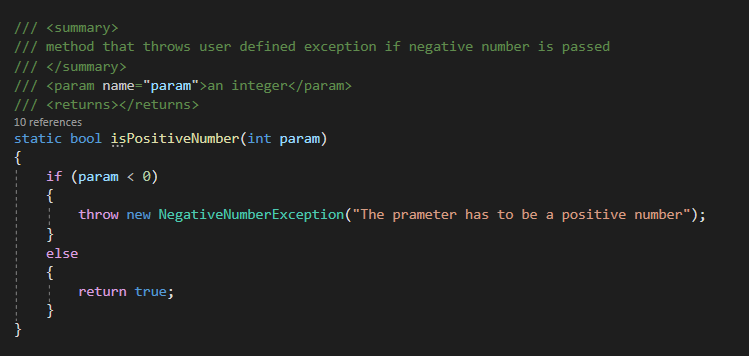
## Use of EXCEPTION handling



## Use of user generated exceptions

Throws exceptions for negative numbers

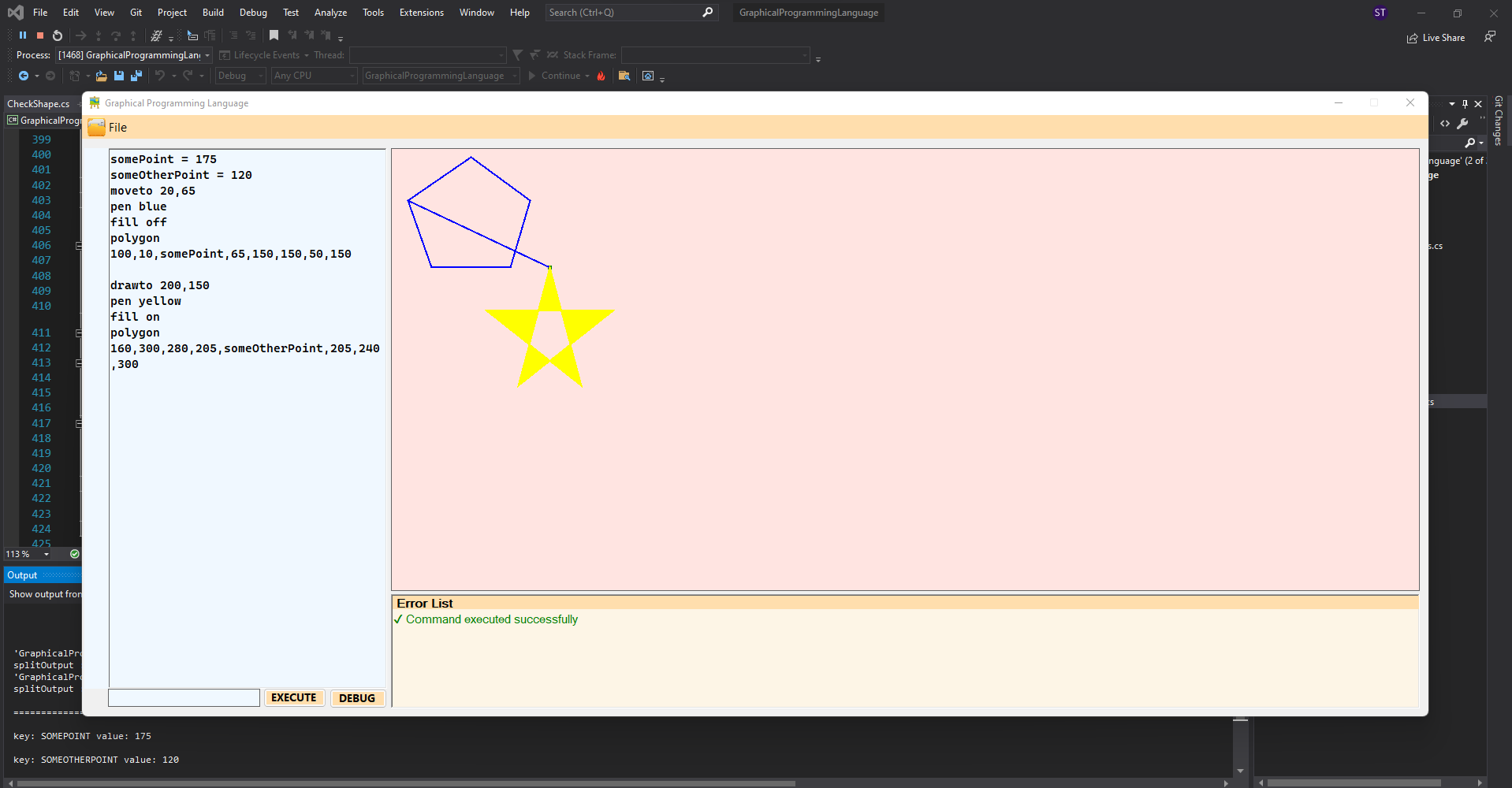




## ADDITIONAL FUNCTIONALITY

### Complex Shape (POLYGON)

Takes pairs of X and Y coordinates to draw the shapes



## Testing

**For Component 1**

Test for ShapeFactory\_Return\_Shape()

|  |  |  |  |
| --- | --- | --- | --- |
| Input | Expected | Actual | Result |
| Circe | ArgumentException | ArgumentException | Pass |
| Circle | No Exception is thrown | ArgumentException | Fail |

Test for Check\_if\_Possible\_Command ()

|  |  |  |  |
| --- | --- | --- | --- |
| Input | Expected | Actual | Result |
| moveto | possibleCommands contains input | possibleCommands contains input | Pass |
| forloop | possibleCommands contains input | possibleCommands does not contains input | Fail |

Test for Set\_Rectangle\_Width\_Height()

|  |  |  |  |
| --- | --- | --- | --- |
| Input | Expected | Actual | Result |
| 50, 60 | rect.width = 50  rect.height = 60 | rect.width = 50  rect.height = 60 | Pass |
| 50 | rect.width = 50  rect.height = | Throws System.IndexOutOfRangeException | Fail |

Test for Return\_Circle ()

|  |  |  |  |
| --- | --- | --- | --- |
| Input | Expected | Actual | Result |
| circle | IsTrue | IsTrue | Pass |
| triangle | IsTrue | IsFalse | Fail |

Test for Return\_Triangle ()

|  |  |  |  |
| --- | --- | --- | --- |
| Input | Expected | Actual | Result |
| triangle | IsTrue | IsTrue | Pass |
| rectangle | IsTrue | IsFalse | Fail |

**For Component 2**

Test for Check\_if\_Valid\_Variable ()

|  |  |  |  |
| --- | --- | --- | --- |
| Input | Expected | Actual | Result |
| VAR2 | 30 | 30 | Pass |
| VAR3 | 30 | 40 | Fail |

Test for Compare\_Operands ()

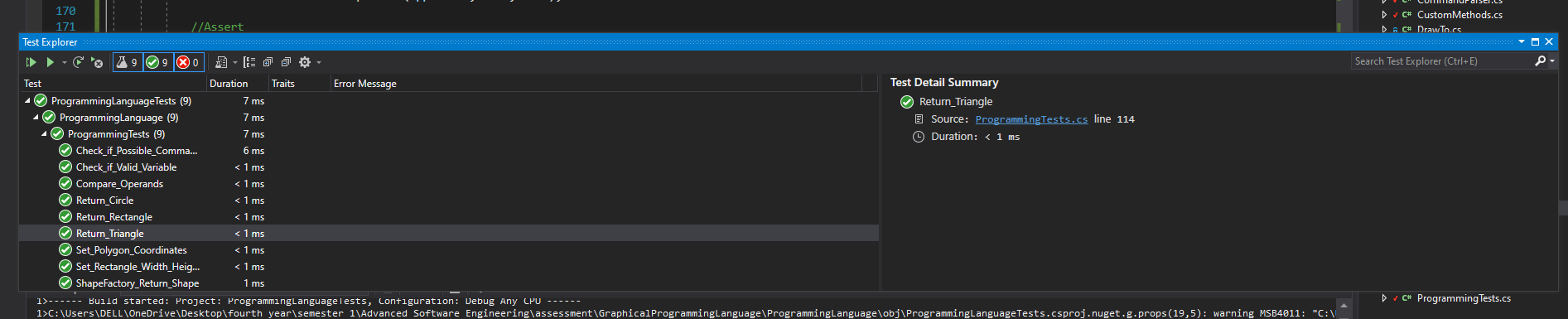
|  |  |  |  |
| --- | --- | --- | --- |
| Input | Expected | Actual | Result |
| num1 = 20;  opperator = "<";  num2 = 100; | IsTrue | IsTrue | Pass |
| num1 = 100;  opperator = "==";  num2 = 100; | IsTrue | IsFalse | Fail |

Test for Set\_Polygon\_Coordinates()

|  |  |  |  |
| --- | --- | --- | --- |
| Input | Expected | Actual | Result |
| int[] polyArray = new int[] { 20, 65, 100, 10 } | poly.polyArray = Shape.polyArray | poly.polyArray = Shape.polyArray | Pass |

Test for Test\_User\_Generated\_Exception()

|  |  |  |  |
| --- | --- | --- | --- |
| Input | Expected | Actual | Result |
| -20 | NegativeNumberException | NegativeNumberException | Pass |
| 20 | NegativeNumberException | No exception thrown | Fail |

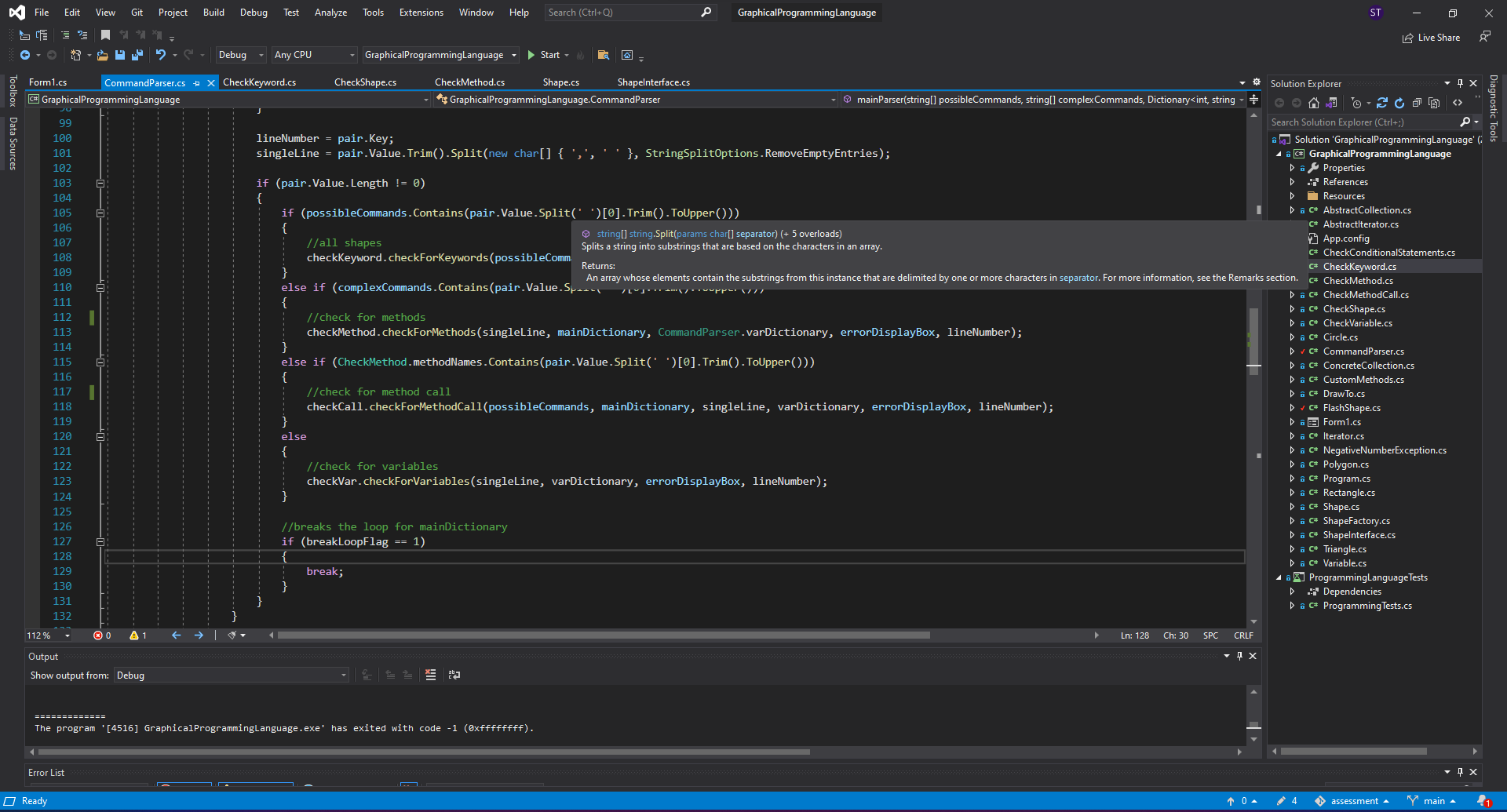


## ADDITIONAL DESIGN PATTERNS

### Façade Design Pattern

The class Command Parser acts as a façade class as the method mainParser() inside the class, when called in Form1, checks for all the commands the user may have entered like METHODS, METHODCALLS, VARIABLES, SHAPES and so on as show below.

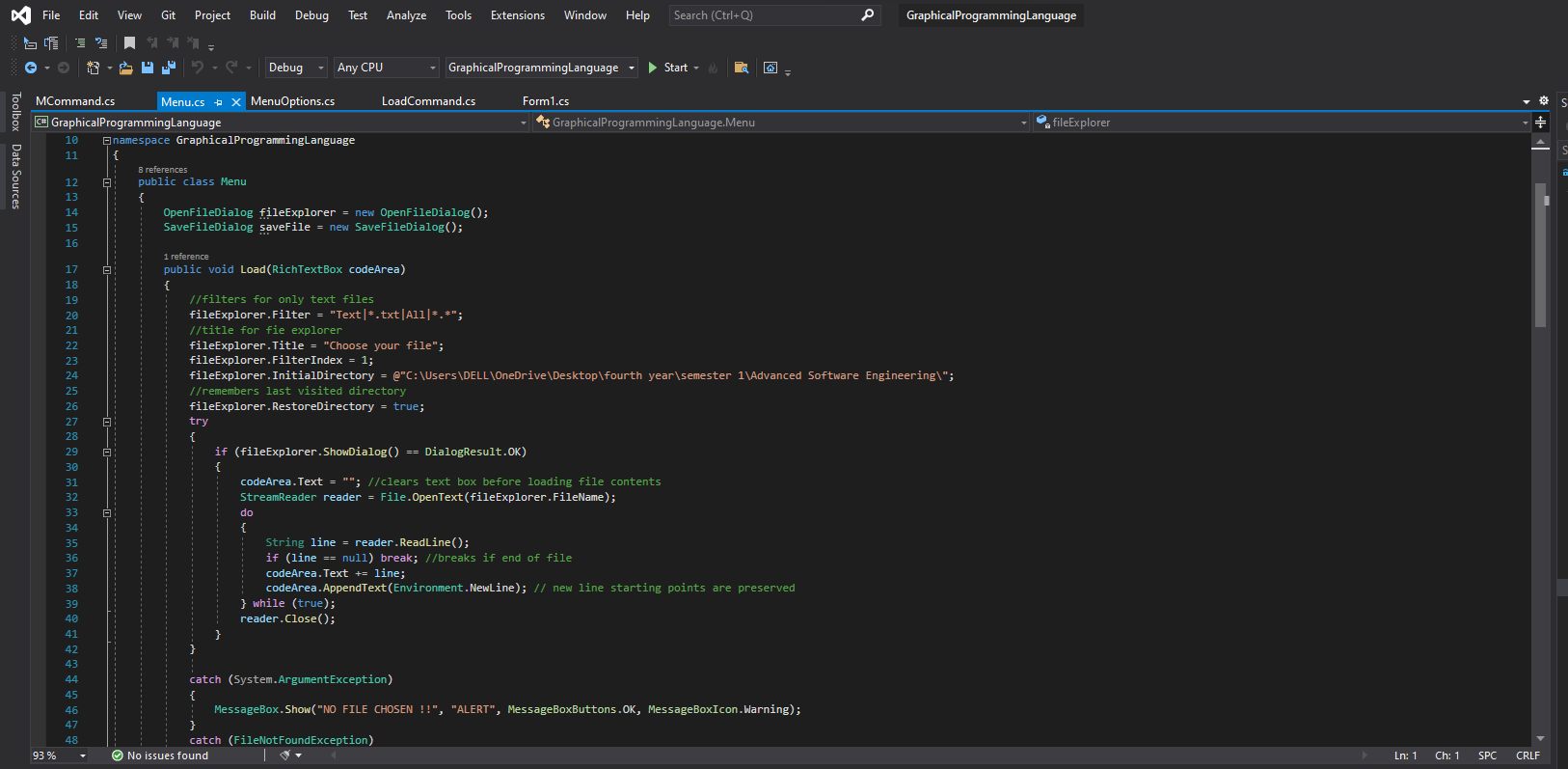
Since a single method i.e. mainParser is making all these calls but making it seem like only one method has been called, we can say that Façade design pattern has been implemented.



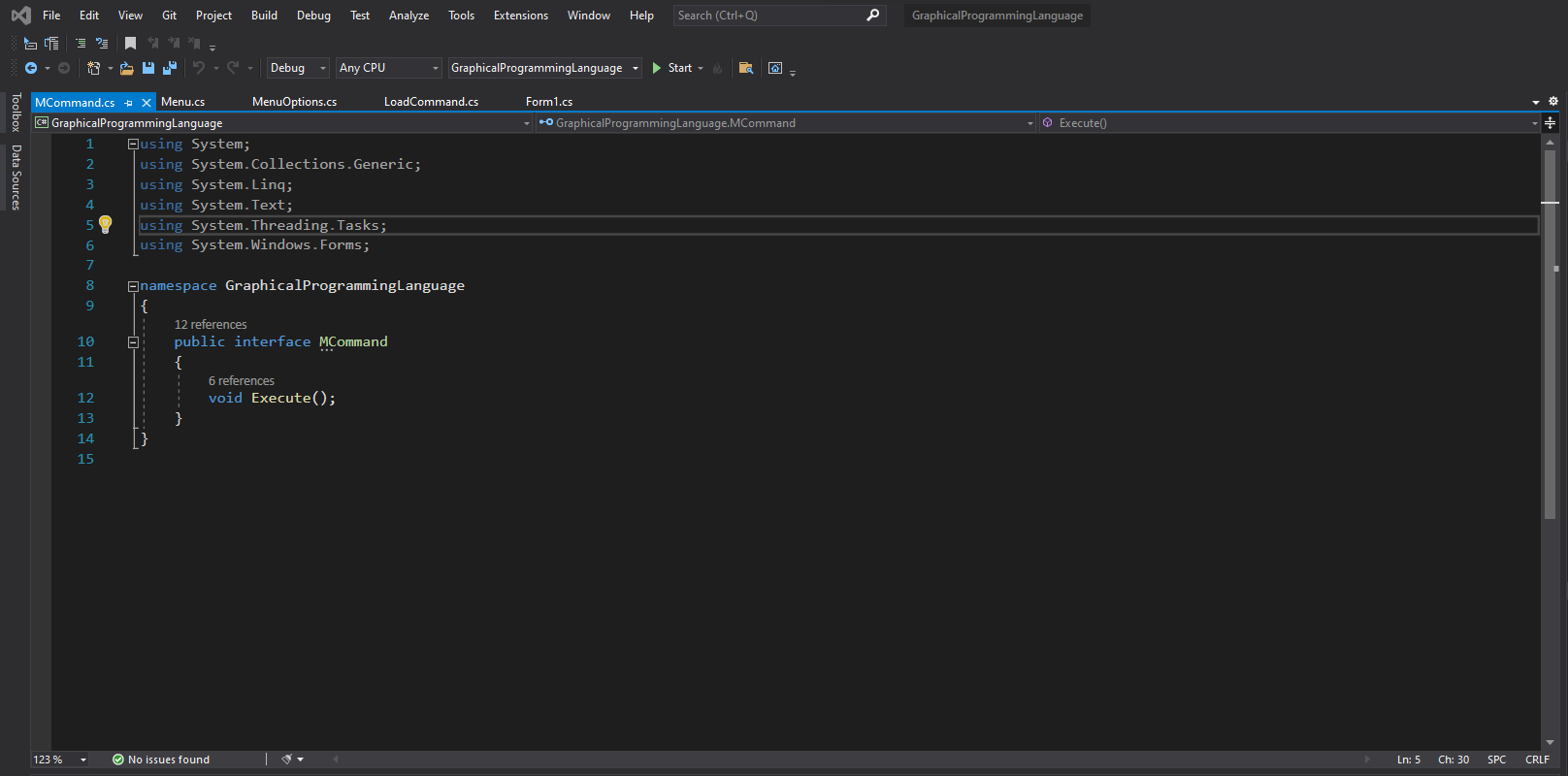
### Command Design Pattern

The Command Design Pattern was use to make execution commands for the menu options which includes Save, Close and Load commands for a file.

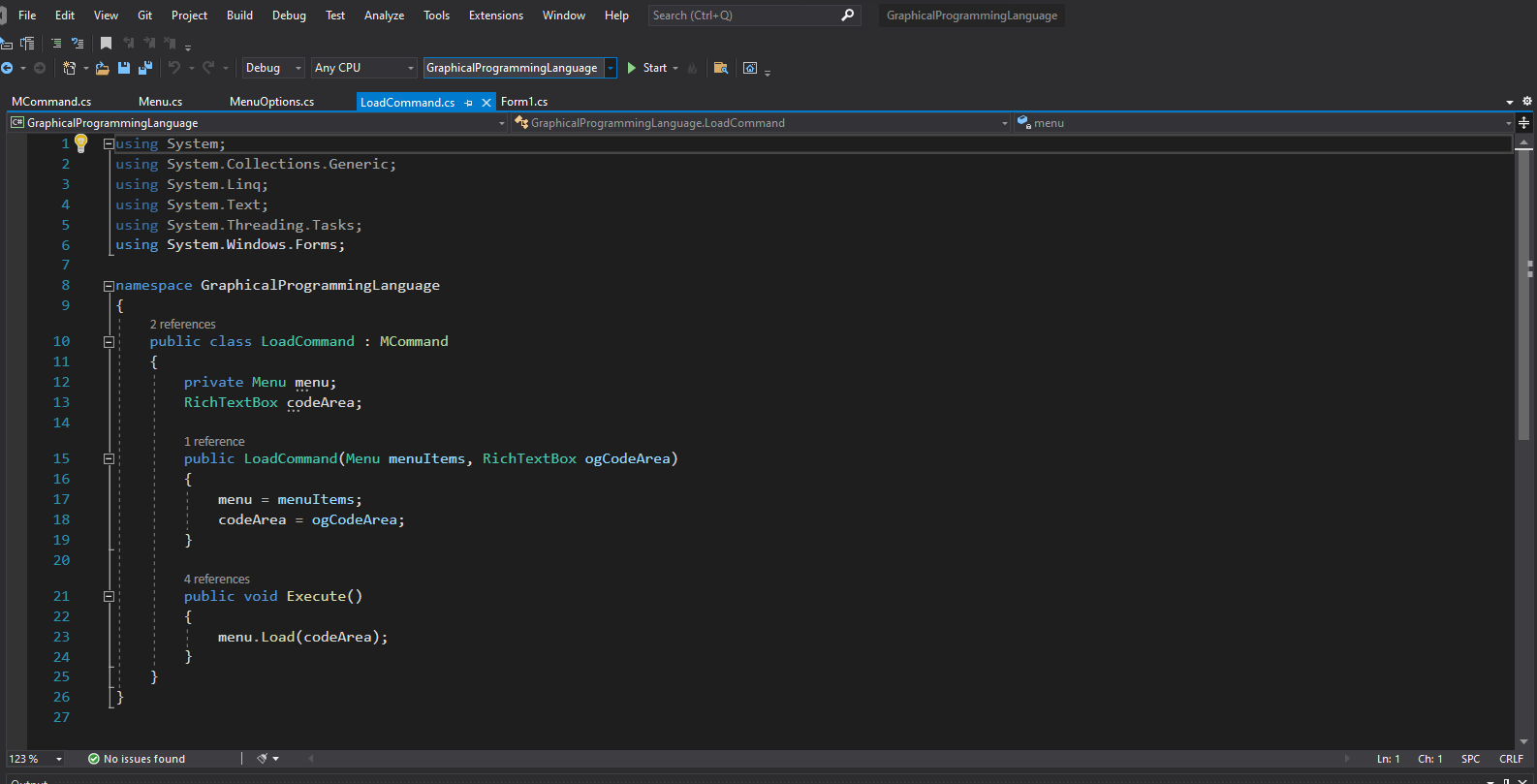
**Creating the receiver object**



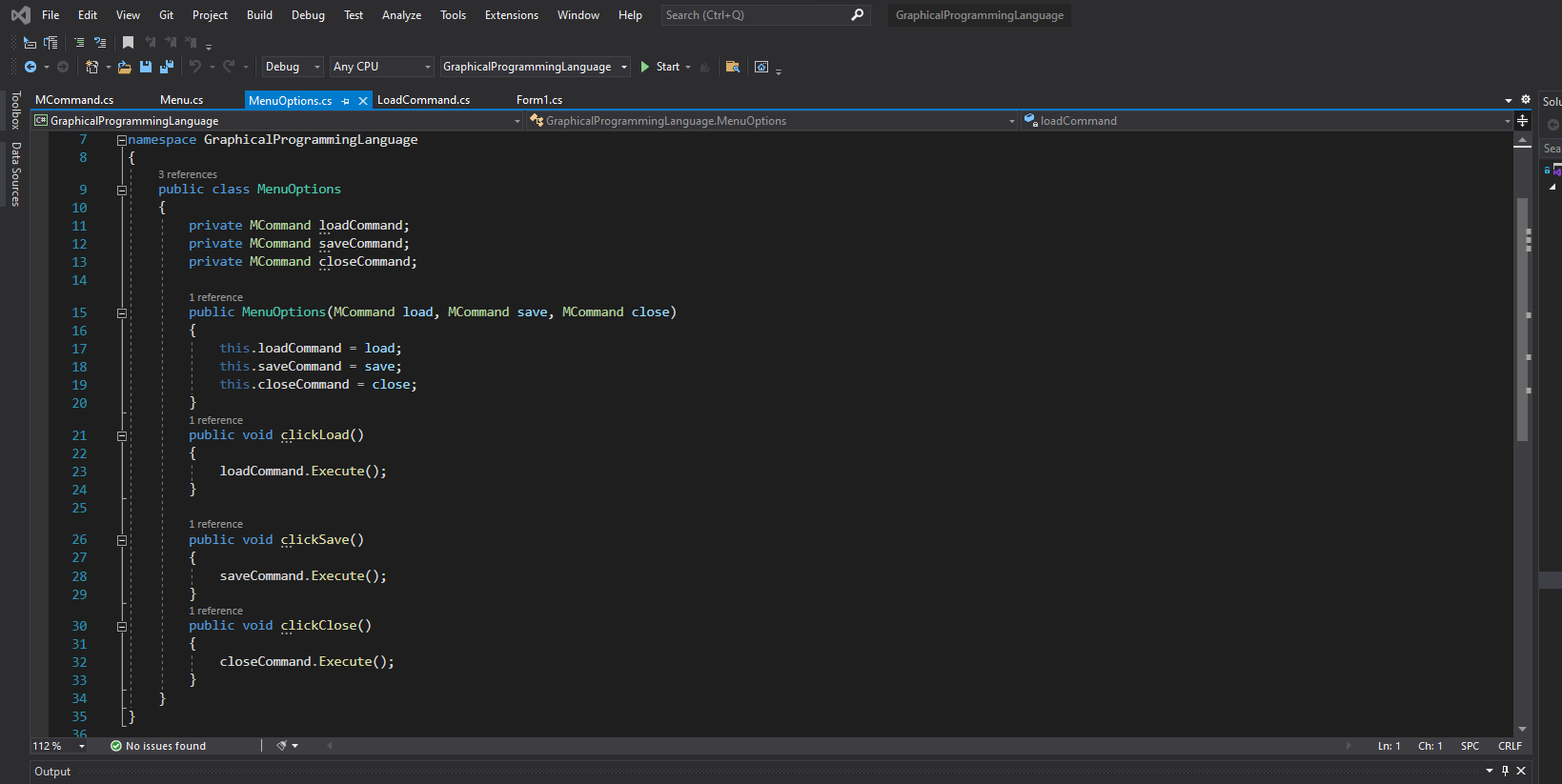
**Creating an interface** which is used to execute a command



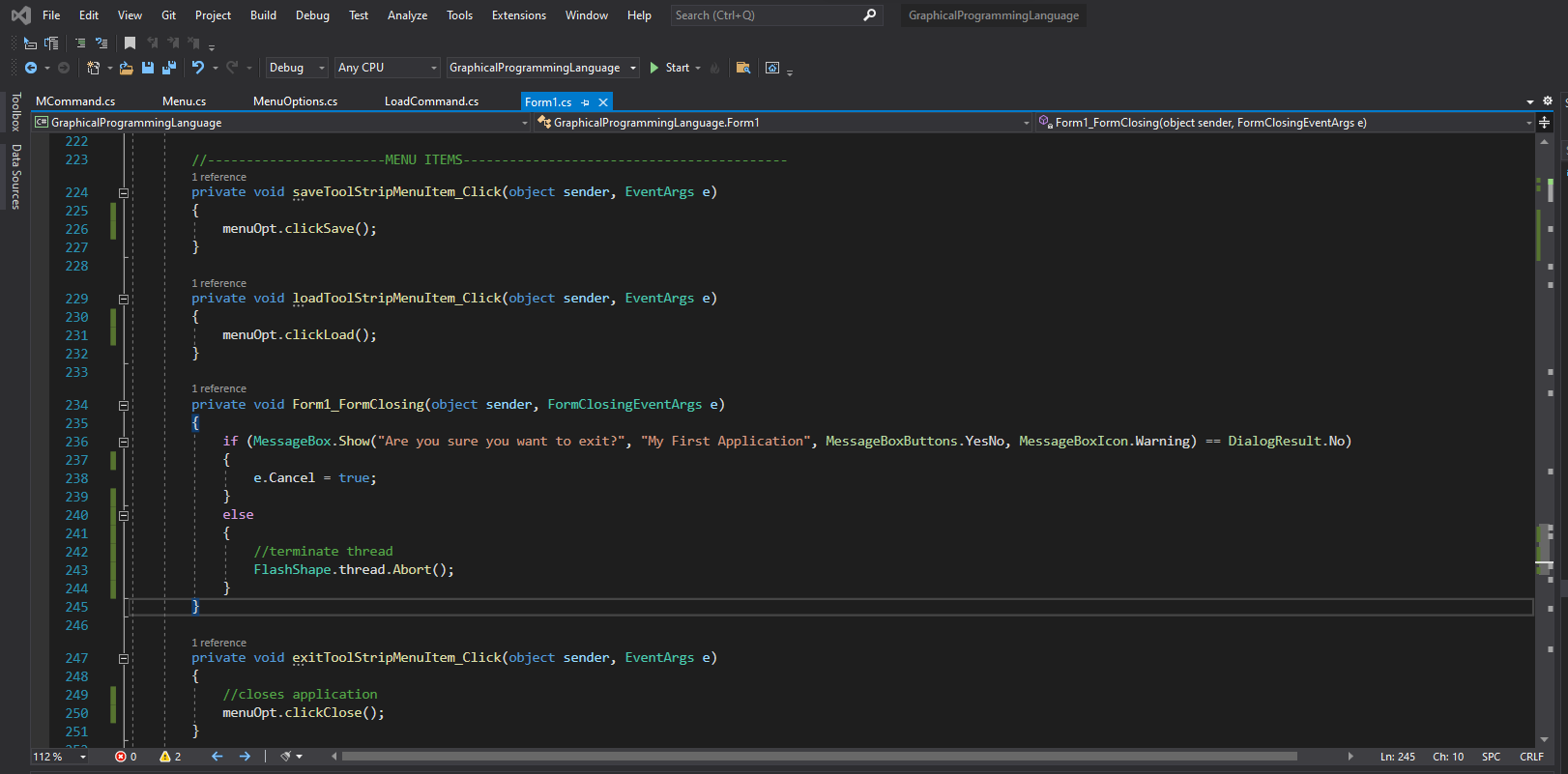
**Creating command classes** that implement the above interface



**Creating the invoker**. The invoker object does not depend on the concrete command or receiver classes. It passes the request to a receiver indirectly by executing a command.



**Usage**



# APPENDIX (source code according to class)

**CheckConditionalStatements.cs**

using System;

using System.Collections;

using System.Collections.Generic;

using System.Data;

using System.Diagnostics;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace GraphicalProgrammingLanguage

{

class CheckConditionalStatements

{

CustomMethods custom = new CustomMethods();

public static int checkLoops = 0;

/// <summary>

/// is called if the user has enterd any WHILE or IF conditions

/// </summary>

/// <param name="singleLine">single line from codeArea</param>

/// <param name="mainDictionary">dictionary that contains all lines the user inputs</param>

/// <param name="varDictionary">dictionary that contains all variables</param>

/// <param name="errorDisplayBox">RichTextBox where the errors are to be displayed</param>

/// <param name="lineNumber"></param>

public void checkForConditionalStatements( string[] singleLine, Dictionary<int, string> mainDictionary, Dictionary<string, int> varDictionary, RichTextBox errorDisplayBox, int lineNumber)

{

//--------------------IF STATEMENT--------------------------------

if ((string)singleLine[0].ToUpper() == "IF")

{

int countArrayNum = singleLine.Length;

int varToCompare = 0;

string compOperator = "";

int lastElement = 0;

CommandParser.ifLineNumber = lineNumber;

int tempLineNumber = lineNumber;

//check if line has 3 components beside the actual keyword

if (countArrayNum - 1 == 3)

{

//get the line number of ENDIF

int tempEndif = 0;

foreach (var row in mainDictionary)

{

if (row.Key > lineNumber)

{

if (row.Value.ToUpper() == "ENDIF")

{

CommandParser.endIfLineNumber = row.Key;

tempEndif = row.Key;

break;

}

try

{

if (mainDictionary[tempLineNumber + 1].Split(' ')[0].Trim().ToUpper() == "IF")

{

break;

}

}

catch (KeyNotFoundException)

{

}

tempLineNumber++;

}

}

CommandParser.ifConditionStatus = 0;

//runs when only one IF statement

if (tempEndif == 0)

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "IF statement was never ended", "IF <some integer value> == <some integer value> ....... ENDIF");

CommandParser.breakFlag = 1;

CommandParser.breakLoopFlag = 1;

}

//get the value of the variable

string[] parameter = CustomMethods.getValueFromDictionary(varDictionary, singleLine[1]);

try

{

//check if the value of variable is positive

if (int.Parse(parameter[0]) >= 0)

{

varToCompare = int.Parse(parameter[0]);

//check if statement has operator

if (singleLine[2].ToUpper() == "==" || singleLine[2].ToUpper() == ">" || singleLine[2].ToUpper() == "<" || singleLine[2].ToUpper() == ">=" || singleLine[2].ToUpper() == "<=" || singleLine[2].ToUpper() == "!=")

{

compOperator = singleLine[2];

//check if RHS element is an integer

try

{

string[] rhsParam = CustomMethods.getValueFromDictionary(varDictionary, singleLine[3]);

lastElement = int.Parse(rhsParam[0]);

bool checkCondition = false;

//check if condition not true accordint to the operators used

checkCondition = custom.checkForOperator(singleLine[2], varToCompare, lastElement);

if (checkCondition == false)

{

CommandParser.ifConditionStatus = 1;

}

}

catch (FormatException)

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Right hand side of operator should be number", "IF <variable> == <some integer value>");

CommandParser.breakLoopFlag = 1;

}

}

else

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Comparison operator not recognized", "== OR <= OR >= OR != OR > OR <");

CommandParser.breakLoopFlag = 1;

}

}

}

catch (FormatException)

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Left hand side of operator should be number", "IF <variable> == <some integer value>");

CommandParser.breakLoopFlag = 1;

}

}

else

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Wrong syntax for IF statement", "IF <variable> == <some integer value>");

CommandParser.breakLoopFlag = 1;

}

}

//---------------------WHILE STATEMENT-------------------------------

if ((string)singleLine[0].ToUpper() == "WHILE")

{

int countArrayNum = singleLine.Length;

int varToCompare = 0;

string compOperator = "";

int lastElement = 0;

CommandParser.whileLineNumber = lineNumber;

int tempLineNumber = lineNumber;

//check if line has 3 components beside the actual keyword

if (countArrayNum - 1 == 3)

{

//get the line number of ENDLOOP

int tempEndLoop = 0;

foreach (var row in mainDictionary)

{

//ensures ENDLOOP comes after WHILE and breaks

if (row.Key > CommandParser.whileLineNumber)

{

if (row.Value.ToUpper() == "ENDLOOP")

{

CommandParser.endLoopLineNumber = row.Key;

tempEndLoop = row.Key;

break;

}

try

{

if (mainDictionary[tempLineNumber + 1].Split(' ')[0].Trim().ToUpper() == "WHILE")

{

break;

}

}

catch (KeyNotFoundException)

{

}

tempLineNumber++;

}

}

CommandParser.whileConditionStatus = 0;

//displays error if ENDLOOP is not found

if (tempEndLoop == 0)

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "WHILE statement was never ended", "WHILE <some integer value> > <some integer value> ....... ENDIF");

CommandParser.breakLoopFlag = 1;

CommandParser.breakFlag = 1;

}

//check while condition

string[] parameter = CustomMethods.getValueFromDictionary(varDictionary, singleLine[1]);

try

{

//check if the value of variable is positive

if (int.Parse(parameter[0]) >= 0)

{

varToCompare = int.Parse(parameter[0]);

//check if statement has operator

if (singleLine[2].ToUpper() == "==" || singleLine[2].ToUpper() == ">" || singleLine[2].ToUpper() == "<" || singleLine[2].ToUpper() == ">=" || singleLine[2].ToUpper() == "<=" || singleLine[2].ToUpper() == "!=")

{

compOperator = singleLine[2];

//check if RHS element is an integer

try

{

//get the value of the variable

string[] rhsParam = CustomMethods.getValueFromDictionary(varDictionary, singleLine[3]);

lastElement = int.Parse(rhsParam[0]);

bool whilecheckCondition = false;

//check if condition true according to the operators used

whilecheckCondition = custom.checkForOperator(singleLine[2], varToCompare, lastElement);

if (whilecheckCondition == false)

{

CommandParser.whileConditionStatus = 1;

}

else

{

checkLoops = 1;

}

//else

//{

// checkLoops = 1;

// //while (checkLoops == 1)

// //{

// foreach (var row in mainDictionary)

// {

// if (row.Key > CommandParser.whileLineNumber && row.Key < CommandParser.endLoopLineNumber)

// {

// whileTuple.Add(new Tuple<int, string>(row.Key, row.Value));

// }

// }

// foreach (var tuple in whileTuple)

// {

// Console.WriteLine("while bhitra kokura {0} - {1} - {2}", tuple.Item1, tuple.Item2);

// }

// //foreach (var tuple in whileTuple)

// //{

// // //grab only those tuples have the name of the praticualr method

// // //if (tuple.Item1.Trim().ToUpper() == methodName)

// // //{

// // //split line once more beofre parsing

// // string[] whileCallLine = tuple.Item2.Trim().Split(new char[] { ',', ' ' }, StringSplitOptions.RemoveEmptyEntries);

// // checkKeyword.checkForKeywords( mainDictionary, errorDisplayBox, tuple.Item1, whileCallLine);

// // //}

// //}

// //}

//}

}

catch (FormatException)

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Right hand side of operator should be number", "WHILE <variable> > <some integer value>");

CommandParser.breakLoopFlag = 1;

}

}

else

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Comparison operator not recognized", "== OR <= OR >= OR != OR > OR <");

CommandParser.breakLoopFlag = 1;

}

}

}

catch (FormatException)

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Left hand side of operator should be number", "WHILE <variable> > <some integer value>");

CommandParser.breakLoopFlag = 1;

}

}

else

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Wrong syntax for WHILE statement", "WHILE <variable> > <some integer value>");

CommandParser.breakLoopFlag = 1;

}

}

}

}

}

**CheckKeyword.cs**

using System;

using System.Collections;

using System.Collections.Generic;

using System.Data;

using System.Diagnostics;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace GraphicalProgrammingLanguage

{

class CheckKeyword

{

CheckShape checkShape = new CheckShape();

CheckMethod checkMethod = new CheckMethod();

CheckConditionalStatements checkStatement = new CheckConditionalStatements();

CustomMethods custom = new CustomMethods();

ShapeFactory factory = new ShapeFactory(); //to return the shapes

public static ArrayList shapes = new ArrayList(); //stores shapes

public Shape shape; //shape of type Shape

public int[] polyArray; //store all points of polygon

/// <summary>

/// extract lineNUmber and all commands from data dictionary, catches and displays errors, displays shapes and reads commandLine

/// </summary>

/// <param name="possibleCommands">array of all possible commands</param>

/// <param name="mainDictionary">dictionary that holds each line to be executed</param>

/// <param name="errorDisplayBox">textBox to display all errors</param>

/// <param name="lineNumber">line number of exch command</param>

/// <param name="singleLine">on single line from the dictionary</param>

public void checkForKeywords(string[] possibleCommands, Dictionary<int, string> mainDictionary, RichTextBox errorDisplayBox, int lineNumber, string[] singleLine)

{

foreach (string element in singleLine)

{

//checks if singleLine[0] is one of the possible commands

if (custom.isPossibleCommand(possibleCommands, singleLine[0]) == true)

{

// check for conditional statements

checkStatement.checkForConditionalStatements(singleLine, mainDictionary, CommandParser.varDictionary, errorDisplayBox, lineNumber);

//check for keywords

checkShape.checkForShape(singleLine, CommandParser.varDictionary, factory, shapes, errorDisplayBox, lineNumber, shape, polyArray);

}

//break foreach loop

if (CommandParser.breakFlag == 1)

{

break;

}

}

}

}

}

**CheckMethod.cs**

using System;

using System.Collections;

using System.Collections.Generic;

using System.Data;

using System.Diagnostics;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace GraphicalProgrammingLanguage

{

class CheckMethod

{

CustomMethods custom = new CustomMethods();

public static List<string> parameters = new List<string>(); // strores all params

public static Dictionary<string, int> methodAndNumberOfParams = new Dictionary<string, int>(); //

public static List<string> methodNames = new List<string>(); // stores all method name

public static List<Tuple<string, int, string>> methodTuple = new List<Tuple<string, int, string>>(); // store methodname, lineNumber, everything inside method

/// <summary>

/// Check for the lines that start with the Keyword METHOD and check and display errors. If no error found then store every line in method inside a Tuple

/// </summary>

/// <param name="singleLine">single line from the mainDictionary</param>

/// <param name="mainDictionary">holds info about the input</param>

/// <param name="varDictionary">dictionary that sotres all variables</param>

/// <param name="errorDisplayBox">to display errors</param>

/// <param name="lineNumber">current line number</param>

public void checkForMethods(string[] singleLine, Dictionary<int, string> mainDictionary, Dictionary<string, int> varDictionary, RichTextBox errorDisplayBox, int lineNumber)

{

if ((string)singleLine[0].ToUpper() == "METHOD")

{

//List<string> parameters = new List<string>();

parameters.Clear();

int countArrayNum = singleLine.Length;

string methodName = "";

int tempEndMethod = 0;

CommandParser.methodLineNumber = lineNumber;

int tempLineNumber = lineNumber;

//check if line has minimu required syntax

if (singleLine.Length >= 4)

{

if (singleLine[2] == "(" && singleLine[singleLine.Length-1] == ")")

{

//check if method is ended

foreach (var row in mainDictionary)

{

if (row.Key > CommandParser.methodLineNumber)

{

if (row.Value.Trim().ToUpper() == "ENDMETHOD")

{

CommandParser.endMethodLineNumber = row.Key;

tempEndMethod = row.Key;

break;

}

try

{

if (mainDictionary[tempLineNumber + 1].Split(' ')[0].Trim().ToUpper() == "METHOD")

{

break;

}

}

catch (KeyNotFoundException)

{

}

tempLineNumber++;

}

}

//return error if not ended

if (tempEndMethod == 0)

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "METHOD was never ended", "Method <method name> ( parameter list ) ....... ENDMETHOD");

CommandParser.breakFlag = 1;

CommandParser.breakLoopFlag = 1;

}

//check if method name is string

bool allowedName = int.TryParse(singleLine[1], out int mName);

if (allowedName == false)

{

methodName = singleLine[1].Trim().ToUpper();

methodNames.Add(methodName.ToUpper());

//check for parameters

for (int i = 3; i < singleLine.Length - 1; i++)

{

bool paramIsInt = int.TryParse(singleLine[i], out int ssd);

if (paramIsInt == false)

{

//add it to the list of parameters and also the varDictionary

parameters.Add(singleLine[i].Trim().ToUpper());

varDictionary[singleLine[i].Trim().ToUpper()] = 0;

}

else

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Parameter list cannot be integers", "Method <method name> ( parameter list ) ....... ENDMETHOD");

CommandParser.breakLoopFlag = 1;

CommandParser.breakFlag = 1;

break;

}

}

//set the name of the method and the number of parameters it contains

methodAndNumberOfParams[methodName] = parameters.Count();

//get methodname, lineNumber and every line inside method besides endmethod

foreach (var row in mainDictionary)

{

if (row.Key >= CommandParser.methodLineNumber && row.Key < CommandParser.endMethodLineNumber)

{

methodTuple.Add(new Tuple<string, int, string>(methodName, row.Key, row.Value));

}

}

}

else

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Method names cannot be a number", "Method <method name> ( parameter list ) ....... ENDMETHOD");

CommandParser.breakLoopFlag = 1;

CommandParser.breakFlag = 1;

}

}

else

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Bracket is missing around the parameters", "Method <method name> ( parameter list ) ....... ENDMETHOD");

CommandParser.breakLoopFlag = 1;

CommandParser.breakFlag = 1;

}

}

else

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "lll Wrong syntax for METHOD", "Method <method name> ( parameter list ) ....... ENDMETHOD");

CommandParser.breakLoopFlag = 1;

CommandParser.breakFlag = 1;

}

//make it 1 becasude we want to skip the method unless called

CommandParser.methodConditionStatus = 1;

}

}

}

}

**CheckMethodCall.cs**

using System;

using System.Collections;

using System.Collections.Generic;

using System.Data;

using System.Diagnostics;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace GraphicalProgrammingLanguage

{

class CheckMethodCall

{

CustomMethods custom = new CustomMethods();

CheckKeyword checkKeyword = new CheckKeyword();

static int everythingOK = 0; //flag set if thera are no errors

/// <summary>

/// Checks if the previously declared methods are called again on a later line. If called, checks for syntax and displays error, if none found sends every line inside

/// the tuple to be parsed again including the line that declares the method

/// </summary>

/// <param name="possibleCommands">array of all the possibel commands</param>

/// <param name="mainDictionary">holds info about the input</param>

/// <param name="singleLine">single line from the mainDictionary</param>

/// <param name="varDictionary">dictionary that sotres all variables</param>

/// <param name="errorDisplayBox">to display errors</param>

/// <param name="lineNumber">current line number</param>

public void checkForMethodCall(string[] possibleCommands, Dictionary<int, string> mainDictionary, string[] singleLine, Dictionary<string, int> varDictionary, RichTextBox errorDisplayBox, int lineNumber)

{

everythingOK = 0;

//check for method call

if (CheckMethod.methodNames.Contains(singleLine[0].ToUpper()))

{

string methodName = singleLine[0].Trim().ToUpper();

//check for syntax errors

if (singleLine.Length >= 3)

{

//check for brackets

if (singleLine[1] == "(" && singleLine[singleLine.Length - 1] == ")")

{

int numOfParamValue = 0;

//count the number of parameters passed

for (int i = 2; i < singleLine.Length - 1; i++)

{

numOfParamValue++;

}

//stores all params

List<int> allParams = new List<int>();

//check if number of given parameters are equal to the parameters of method

if (numOfParamValue == CheckMethod.methodAndNumberOfParams[methodName])

{

if (CheckMethod.methodAndNumberOfParams[methodName] == 0)

{

//////pass all lines between method to command parser be parsed again

foreach (var tuple in CheckMethod.methodTuple)

{

/////////////////////////

if (tuple.Item1.Trim().ToUpper() == methodName)

{

string[] methodCallLine = tuple.Item3.Trim().Split(new char[] { ',', ' ' }, StringSplitOptions.RemoveEmptyEntries);

checkKeyword.checkForKeywords(possibleCommands, mainDictionary, errorDisplayBox, tuple.Item2, methodCallLine);

}

}

}

//checks if all the prams passed is an integer

for (int i = 2; i < singleLine.Length - 1; i++)

{

bool paramIsInteger = int.TryParse(singleLine[i].Trim(), out int ssd);

if (paramIsInteger)

{

allParams.Add(int.Parse(singleLine[i]));

everythingOK = 1;

}

else

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Parameter list must be only integers", "<method name> ( parameters )");

CommandParser.breakLoopFlag = 1;

CommandParser.breakFlag = 1;

break;

}

}

//runs if no errors

if (everythingOK == 1)

{

string firstLine = "";

int flag = 0;

//get only the first line from the tuple (i.e something like (MYMETHOD - 5 - Method myMethod ( radius, size )))

foreach (Tuple<string, int, string> tuple in CheckMethod.methodTuple)

{

if (flag == 0 && tuple.Item1.Trim().ToUpper() == methodName)

{

firstLine = tuple.Item3;

flag++;

}

}

//split the line

string[] splitTuple = firstLine.Split(new char[] { ',',' ' }, StringSplitOptions.RemoveEmptyEntries);

List<string> storeParams = new List<string>();

//grab only the parameters and store it (radius AND size)

for (int i = 3; i < splitTuple.Length - 1; i++)

{

storeParams.Add(splitTuple[i]);

}

//add those parameters to the vardictionary

for (int ii = 0; ii < storeParams.Count(); ii++)

{

varDictionary[storeParams[ii].ToUpper()] = allParams[ii];

}

//pass all lines between method to command parser be parsed again

foreach (var tuple in CheckMethod.methodTuple)

{

//grab only those tuples have the name of the praticualr method

if (tuple.Item1.Trim().ToUpper() == methodName)

{

//split line once more beofre parsing

string[] methodCallLine = tuple.Item3.Trim().Split(new char[] { ',', ' ' }, StringSplitOptions.RemoveEmptyEntries);

checkKeyword.checkForKeywords(possibleCommands, mainDictionary, errorDisplayBox, tuple.Item2, methodCallLine);

}

}

}

}

else

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Number of prameters are not equal", "<method name> ( parameters )");

CommandParser.breakLoopFlag = 1;

}

}

else

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Wrong syntax for METHOD calling", "<method name> ( parameters )");

CommandParser.breakLoopFlag = 1;

}

}

else

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Wrong syntax for METHOD calling", "<method name> ( parameters )");

CommandParser.breakLoopFlag = 1;

}

}

else

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Keyword does not exist", "circle OR triangle OR rectangle OR drawto OR moveto");

CommandParser.breakLoopFlag = 1;

}

}

}

}

**CheckVariable.cs**

using System;

using System.Collections;

using System.Collections.Generic;

using System.Data;

using System.Diagnostics;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace GraphicalProgrammingLanguage

{

class CheckVariable

{

CustomMethods custom = new CustomMethods();

/// <summary>

/// checks for variables and if present adds it to varDictionary along with its value. If a variable with the same name exists it updates it value. If variable holds any

/// arithmatic expression it caclculates it. Throws error if a purely numerical name is given as variable names (Alha numeric is allowed). If variables are assigned or

/// calculations are performed using any other variable names then it is checked for in the dicitonary and appropiate value is used in the calculation

/// </summary>

/// <param name="singleLine">the line from the codeArea that may contain syntax to declare or nodify a variable variable</param>

/// <param name="varDictionary">dictionary that contains all variables and their respective values</param>

/// <param name="errorDisplayBox"></param>

/// <param name="lineNumber">tline number of the currrent line</param>

public void checkForVariables(string[] singleLine, Dictionary<string, int> varDictionary, RichTextBox errorDisplayBox, int lineNumber)

{

try

{

//checks for variabless

if (singleLine.Contains("=") && Array.IndexOf(singleLine, "=") == 1)

{

if (singleLine[1] == "=")

{

int indexOfEqualsSign = Array.IndexOf(singleLine, "="); //var = 10+20+30

string varName = "";

string output = ""; //to store all the things that need to be calculated

//create string to calculate value to put in the dictionary

string[] varValueArray = new string[10];

Array.Copy(singleLine, indexOfEqualsSign + 1, varValueArray, 0, singleLine.Length - 2);

varValueArray = varValueArray.Where(c => c != null).ToArray();

foreach (string input in varValueArray)

{

//stores something like : 10+20+30 OR height+20

output += input;

}

//replace variable with its value

string[] splitOutput = output.Split(new char[] { '+', '-', '/', '\*' });

Debug.WriteLine("splitOutput : " + splitOutput.Length);

foreach (string operand in splitOutput)

{

string opp = operand.Trim().ToUpper();

if (varDictionary.ContainsKey(opp))

{

int valueOfOperand = varDictionary[opp];

output = output.Replace(operand, valueOfOperand.ToString());

}

}

//compute variable name and value for dictionary

try

{

var result = new DataTable().Compute(output, null);

//check if variable name is a string

bool isVarString = int.TryParse(singleLine[indexOfEqualsSign - 1], out int varrName);

if (isVarString == false)

{

varName = singleLine[indexOfEqualsSign - 1];

}

else

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "variable names cannot be a number", "<variable name> = <some integer>");

CommandParser.breakLoopFlag = 1;

//break;

}

try

{

//check if result returns a positive integer

if (Convert.ToInt32(result) >= 0)

{

//store the result

int varValue = Convert.ToInt32(result);

//check if variable already exists

if (varDictionary.ContainsKey(varName.Trim().ToUpper()))

{

//update value

varDictionary[varName.Trim().ToUpper()] = varValue;

}

else

{

//add value

varDictionary.Add(varName.Trim().ToUpper(), varValue);

}

}

}

catch (InvalidCastException)

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Variable value cannot be empty", "<variable name> = <some integer>");

CommandParser.breakLoopFlag = 1;

}

}

catch (FormatException)

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "variable names cannot be a number", "<variable name> = <some integer>");

CommandParser.breakLoopFlag = 1;

}

// asdf = asdf

catch (EvaluateException)

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "varaibles cannot store strings", "<variable name> = <some integer>");

CommandParser.breakLoopFlag = 1;

//break;

}

catch (SyntaxErrorException)

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "varaibles cannot store strings", "<variable name> = <some integer>");

CommandParser.breakLoopFlag = 1;

//break;

}

}

}

else

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Keyword does not exist", "circle OR triangle OR rectangle OR drawto OR moveto");

CommandParser.breakLoopFlag = 1;

//break;

}

}

catch (IndexOutOfRangeException)

{

errorDisplayBox.Text += "\naaaaaaaaaaaaaaaaaaaaa";

}

}

}

}

**Circle.cs**

using System;

using System.Collections.Generic;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace GraphicalProgrammingLanguage

{

public class Circle : Shape

{

int radius;

public Circle() : base()

{

}

public Circle(Color colour, int x, int y, int radius) : base(colour, x, y)

{

this.radius = radius; //the only thing that is different from shape

}

public override void set(Color colour, Boolean fill, bool flash, Color primaryColor, Color secondaryColor, params int[] list)

{

base.colour = colour;

//list[0] is x, list[1] is y, list[2] is radius

base.set(colour, fill, flash, primaryColor, secondaryColor, list[0], list[1]);

this.radius = list[2];

}

public override void draw(Graphics g, Boolean fill)

{

SolidBrush brush = new SolidBrush(Color.Transparent);

Pen pen = new Pen(base.colour, 2);

if (base.fill == true)

{

brush = new SolidBrush(base.colour);

}

else

{

brush = new SolidBrush(Color.Transparent);

}

g.FillEllipse(brush, x - radius, y - radius, radius \* 2, radius \* 2);

g.DrawEllipse(pen, x - radius, y - radius, radius \* 2, radius \* 2);

}

public override string ToString() //all classes inherit from object and ToString() is abstract in object

{

return base.ToString() + " " + this.radius;

}

}

}

**CloseCommand.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace GraphicalProgrammingLanguage

{

public class CloseCommand : MCommand

{

private Menu menu;

public CloseCommand(Menu menuItems)

{

menu = menuItems;

}

public void Execute()

{

menu.Close();

}

}

}

**CommandParser.cs**

**CustomMethods.cs**

using System;

using System.Collections;

using System.Collections.Generic;

using System.Data;

using System.Diagnostics;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace GraphicalProgrammingLanguage

{

public class CustomMethods

{

public Shape shape;//shape of type Shape

/// <summary>

/// checks if command passed is within possibleCommand array

/// </summary>

/// <param name="possibleCommands">all possible commands</param>

/// <param name="command">the command to be checked</param>

/// <returns>returns true if the array passed contains the command passed and vice versa</returns>

public bool isPossibleCommand(string[] possibleCommands, string command)

{

if (possibleCommands.Contains((string)command.ToUpper()))

{

return true;

}

return false;

}

/// <summary>

/// displays the error message along with line number and the correct format in the specified textBox with some extra formatting

/// </summary>

/// <param name="errorDisplayBox">richTextBox to display errors</param>

/// <param name="lineNumber">line number where the error is located</param>

/// <param name="error">the name of the error</param>

/// <param name="correctFormat">the correct format for the command</param>

public void displayErrorMsg(RichTextBox errorDisplayBox, int lineNumber, String error, String correctFormat)

{

errorDisplayBox.Text += "\n⚠️ Line " + lineNumber + " : " + error + "❗ |" + " EXPECTED:: " + correctFormat;

}

/// <summary>

/// draws the current position of the MOVETO co-ordinates

/// </summary>

/// <param name="e">paint event argument to cal the Graphics object</param>

/// <param name="penX">x-coordinate</param>

/// <param name="penY">y-coordinate</param>

public void drawCurrMoveToPos(PaintEventArgs e, int penX, int penY)

{

Graphics movePos = e.Graphics;

//color of pen

Pen p = new Pen(Color.Green, 1);

movePos.DrawRectangle(p, penX - (4 / 2), penY - (4 / 2), 4, 4);

movePos.FillRectangle(new SolidBrush(Color.Green), penX - (4 / 2), penY - (4 / 2), 4, 4);

}

/// <summary>

/// gets the shapes from the factory class and stores it in a variable called shape of type Shape. Then sets the color and parameter of the shapes and finally adds it to the arraylist

/// </summary>

/// <param name="color">color of the shape</param>

/// <param name="fill">specifies the fill state of the shape</param>

/// <param name="flash">flash state of shape</param>

/// <param name="primaryColor">primary color for flash</param>

/// <param name="secondaryColor">secondary color for flash</param>

/// <param name="factory"> returns the actual shape</param>

/// <param name="keyword">the shape to be returned</param>

/// <param name="shapes">arraylis to store the shapes</param>

/// <param name="list">list of parameters for the given shape (e.g. height, width, radius etc)</param>

public void getAndAddShape(Color color, Boolean fill, bool flash, Color primaryColor, Color secondaryColor, ShapeFactory factory, String keyword, ArrayList shapes, params int[] list)

{

//get the specific shape from factory class

shape = factory.getShape((String)keyword);

//set the color and parameter of the shape

shape.set(color, fill, flash, primaryColor, secondaryColor, list);

//add shape to the array

shapes.Add(shape);

}

/// <summary>

/// takes eah shape from arraylist and casts it to the type of 'Shape' and uses graphics to draw the actual shape

/// </summary>

/// <param name="shapes">array that stores the shapes</param>

/// <param name="shape">the actual shape</param>

/// <param name="draw">Graphics instance to draw the shape</param>

/// <param name="fill">fill state of shape</param>

public void drawShapes(ArrayList shapes, Shape shape, Graphics draw, Boolean fill)

{

// draw all shapes stored in the 'shapes' arralist

for (int i = 0; i < shapes.Count; i++)

{

//cast all shape as type "Shape"

shape = (Shape)shapes[i];

//checks until the end

if (shape != null)

{

shape.draw(draw, fill); //draw the actual shape

}

}

}

/// <summary>

/// checks if the dictionary contains the list passed, if it does returns its numerical value , if not returns the list as is

/// </summary>

/// <param name="varDictionary">dictionary that contains all variables and its value</param>

/// <param name="list">all variables that we need the value of</param>

/// <returns></returns>

public static string[] getValueFromDictionary(Dictionary<string, int> varDictionary, params string[] list)

{

string[] ParamNumList = new string[100];

for (int i = 0; i < list.Length; i++)

{

string tempVar = list[i].Trim().ToUpper();

if (varDictionary.ContainsKey(tempVar))

{

int valueOfOperand = varDictionary[tempVar];

ParamNumList[i] = tempVar.Replace(tempVar, valueOfOperand.ToString());

}

else

{

ParamNumList[i] = tempVar;

}

}

return ParamNumList;

}

/// <summary>

/// checks if the dictionary passed contains a WHILE statement

/// </summary>

/// <param name="dictionary">dictionary to be checked</param>

/// <returns></returns>

public Boolean hasWhile(Dictionary<int, string> dictionary)

{

Boolean whileExists = true;

foreach (KeyValuePair<int, string> res in dictionary)

{

string allLines = "";

allLines += res.Value;

string[] alllinesArray = allLines.Trim().ToUpper().Split(new char[] { ' ', '+', '=', '+', '-', '/', '\*', '<', '>', '!', ',' },

StringSplitOptions.RemoveEmptyEntries);

if (alllinesArray.Contains("WHILE")) { whileExists = true; break; }

else { whileExists = false; }

}

return whileExists;

}

/// <summary>

/// returns true if any of the condition mathces with the predefined IF conditions else returns false

/// </summary>

/// <param name="opp">the opertor to be used in the comparison</param>

/// <param name="varToCompare">one of the operands (LHS)</param>

/// <param name="lastElement">other operand (RHS)</param>

/// <returns></returns>

public Boolean checkForOperator(string opp, int varToCompare , int lastElement)

{

Boolean chk = false;

if (opp.ToUpper() == "==")

{

if (varToCompare == lastElement)

{

chk = true;

}

}

else if (opp.ToUpper() == ">")

{

if (varToCompare > lastElement)

{

chk = true;

}

}

else if (opp.ToUpper() == "<")

{

if (varToCompare < lastElement)

{

chk = true;

}

}

else if (opp.ToUpper() == ">=")

{

if (varToCompare >= lastElement)

{

chk = true;

}

}

else if (opp.ToUpper() == "<=")

{

if (varToCompare <= lastElement)

{

chk = true;

}

}

else if (opp.ToUpper() == "!=")

{

if (varToCompare != lastElement)

{

chk = true;

}

}

else

{

chk = false;

}

return chk;

}

}

}

**Drawto.cs**

using System;

using System.Collections.Generic;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace GraphicalProgrammingLanguage

{

class DrawTo : Shape

{

int xCor, yCor;

public DrawTo() : base()

{

}

public DrawTo(Color colour, int x, int y, int xCor, int yCor) : base(colour, x, y)

{

this.xCor = xCor; //the only thing that is different from shape

this.yCor = yCor;

}

public override void set(Color colour, Boolean fill, bool flash, Color primaryColor, Color secondaryColor, params int[] list)

{

base.set(colour, fill, flash, primaryColor, secondaryColor, list[0], list[1]);

this.xCor = list[2];

this.yCor = list[3];

}

public override void draw(Graphics g, Boolean fill)

{

Pen pen = new Pen(base.colour, 2);

g.DrawLine(pen, x, y, xCor, yCor);

}

public override string ToString() //all classes inherit from object and ToString() is abstract in object

{

return base.ToString() + " " + this.xCor + " " + this.yCor;

}

}

}

**FlashShape.cs**

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

namespace GraphicalProgrammingLanguage

{

public class FlashShape

{

Form1 form;

ArrayList shapes;

bool flashFlag = false;

public static Thread thread;

public FlashShape(Form1 form, ArrayList shapes)

{

this.form = form;

this.shapes = shapes;

thread = new Thread(flashShapes);

thread.Start();

}

/// <summary>

/// flash alternating color primary and secondary color for each shape

/// </summary>

public void flashShapes()

{

while (true)

{

foreach (Shape shape in shapes)

{

if (flashFlag == false)

{

if (shape.flash)

{

shape.fill = true;

shape.colour = shape.primaryColor;

}

}

else

{

if (shape.flash)

{

shape.fill = true;

shape.colour = shape.secondaryColor;

}

}

}

flashFlag = !flashFlag; //alternate colors

form.refreshPictureBox();

Thread.Sleep(500);

}

}

}

}

**Form1.cs**

using System;

using System.Collections;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Diagnostics;

using System.Drawing;

using System.IO;

using System.Linq;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace GraphicalProgrammingLanguage

{

public partial class Form1 : Form

{

//public static Thread colorFlash;

bool flashFlag = false, running = false;

FlashShape flash;

//flag to stop flashing

public static int abortFlag = 0;

CheckShape chShape = new CheckShape();

CommandParser parser = new CommandParser();

CustomMethods custom = new CustomMethods();

CheckKeyword chkKeyword = new CheckKeyword();

public static Dictionary<int, string> mainDictionary = new Dictionary<int, string>();

ArrayList singleLineCommand = new ArrayList();

//flag set to determine if command is from commandLine or codeArea (i.e 1 for codeArea and 2 for commandLine)

static int flag = 0;

static int debugFlag = 0;

// stores all possible commands

public string[] possibleCommands = { "DRAWTO", "MOVETO", "CIRCLE", "RECTANGLE", "TRIANGLE", "PEN", "FILL", "POLYGON", "IF", "ENDIF", "WHILE", "ENDLOOP"};

public string[] possibleComplexCommands = {"METHOD", "ENDMETHOD"};

//client side object to invoke invoker objects for file menu items

public static Menu menu = new Menu();

public static MCommand loadCommand;

public static MCommand saveCommand;

public static MCommand closeCommand;

MenuOptions menuOpt;

public Form1()

{

//supresses cross thread call errros

PictureBox.CheckForIllegalCrossThreadCalls = false;

InitializeComponent();

flash = new FlashShape(this, CheckKeyword.shapes);

loadCommand = new LoadCommand(menu, codeArea);

saveCommand = new SaveCommand(menu, codeArea);

closeCommand = new CloseCommand(menu);

menuOpt = new MenuOptions(loadCommand, saveCommand, closeCommand);

}

/// <summary>

/// made this so the PictureBox (drawingArea) 'refresh' method can be called on another class

/// </summary>

public void refreshPictureBox()

{

drawingArea.Refresh();

}

/// <summary>

/// takes input from either the commandLine or the actual code area depending on the commands passed in the commandLine

/// </summary>

/// <param name="input">the command to be parsed</param>

public void startExecution( String input)

{

//for multi line codes i.e codeArea

CommandParser.breakLoopFlag = 0;

CheckConditionalStatements.checkLoops = 0;

if (flag == 1)

{

errorDisplayBox.Text = "";

string code = input;

mainDictionary.Clear();

// split lines

string[] splitLine = code.Split(new char[] { '\n' });

int lineNumber = 1;

foreach (string line in splitLine)

{

// add the entire line as value and the lineNumber as key

mainDictionary.Add(lineNumber, line);

Debug.WriteLine(line);

lineNumber++;

}

}

//for single line codes i.e commandLine

if (flag == 2)

{

if (CheckKeyword.shapes.Count > 0) { singleLineCommand.Clear(); }

errorDisplayBox.Text = "";

//take input

string singleCommand = input;

//assign lineNumber

int lineNumber = 1;

//add the command to arraylist

singleLineCommand.Add(singleCommand);

// for every line add the lineNumber as key and entire line as value

foreach (string line in singleLineCommand)

{

mainDictionary.Add(lineNumber, line);

lineNumber++;

}

}

// main execution part of the program (see xml file for full specifications)

parser.mainParser(possibleCommands, possibleComplexCommands, mainDictionary, errorDisplayBox, drawingArea, commandLine);

}

private void runCode\_Click(object sender, EventArgs e)

{

if (CommandParser.breakLoopFlag == 0)

{

debugFlag = 1;

forExecution();

}

else

{

runCode.Enabled = false;

}

}

public void forExecution()

{

String commandLineInput = commandLine.Text; //reads the command in the 'commandLine'

if (commandLineInput.Equals("run", StringComparison.InvariantCultureIgnoreCase))

{

//stop falshing and make the primary and secondary colors transparent by default

CheckShape.flash = false;

CheckShape.primaryColor = Color.Transparent;

CheckShape.secondaryColor = Color.Transparent;

CheckMethod.methodTuple.Clear();

CheckMethod.parameters.Clear();

mainDictionary.Clear();

String codeAreaInput = codeArea.Text;

flag = 1;

abortFlag = 0;

startExecution(codeAreaInput);

}

else if (commandLineInput.Equals("clear", StringComparison.InvariantCultureIgnoreCase))

{

CommandParser.varDictionary.Clear();

errorDisplayBox.Clear();

//clears all the sahpes in the array then refreshes the pictureBox so everything dissapears

CheckKeyword.shapes.Clear();

CommandParser.fill = false;

drawingArea.Refresh();

}

else if (commandLineInput.Equals("reset", StringComparison.InvariantCultureIgnoreCase))

{

//resets moveto position to (0,0)

CommandParser.penX = 0;

CommandParser.penY = 0;

//refresh to implement above changes

drawingArea.Refresh();

}

else if ((string.IsNullOrWhiteSpace(commandLineInput) && commandLine.Text.Length > 0) || commandLine.Text == "")

{

errorDisplayBox.Text = "\n⚠️ No command given on the command parser (try: run, clear, reset or any of the other possible commands)";

}

else

{

flag = 2;

//runs the command like a multiline command

startExecution(commandLineInput);

}

}

private void commandLine\_KeyDown(object sender, KeyEventArgs e)

{

if (e.KeyCode == Keys.Enter)

{

//peforms click operation of runCode button

runCode.PerformClick();

//refresh line number

displayLineNumber.Refresh();

// these last two lines will stop the beep sound

e.SuppressKeyPress = true;

e.Handled = true;

}

}

private void DrawingArea\_Paint(object sender, PaintEventArgs e)

{

if(debugFlag == 1)

{

//draws a 3 by 3 rectangle to help visualize the current position of the 'Moveto' object

custom.drawCurrMoveToPos(e, CommandParser.penX, CommandParser.penY);

CommandParser.draw = e.Graphics;

//draw all shapes stored in the 'shapes' arralist

custom.drawShapes(CheckKeyword.shapes, chkKeyword.shape, CommandParser.draw, CommandParser.fill);

}

}

//-----------------------MENU ITEMS------------------------------------------

private void saveToolStripMenuItem\_Click(object sender, EventArgs e)

{

menuOpt.clickSave();

}

private void loadToolStripMenuItem\_Click(object sender, EventArgs e)

{

menuOpt.clickLoad();

}

private void Form1\_FormClosing(object sender, FormClosingEventArgs e)

{

if (MessageBox.Show("Are you sure you want to exit?", "My First Application", MessageBoxButtons.YesNo, MessageBoxIcon.Warning) == DialogResult.No)

{

e.Cancel = true;

}

else

{

//terminate thread

FlashShape.thread.Abort();

}

}

private void exitToolStripMenuItem\_Click(object sender, EventArgs e)

{

//closes application

menuOpt.clickClose();

}

int maxLineCount = 1; //maxLineCount - should be public

private void codeArea\_KeyUp(object sender, KeyEventArgs e)

{

//gets the current number of lines using charIndex

int linecount = codeArea.GetLineFromCharIndex(codeArea.TextLength) + 1;

if (linecount != maxLineCount)

{

displayLineNumber.Clear();

//loops for every line in codeArea

for (int i = 1; i < linecount + 1; i++)

{

displayLineNumber.AppendText(Convert.ToString(i) + ".\r\n"); //displays the formatted line number to the textBox

}

maxLineCount = linecount;

}

}

private void codeArea\_VScroll(object sender, EventArgs e)

//realign line numbers when codeArea is scrolled

{

displayLineNumber.Text = "";

//gets the current number of lines using charIndex

int linecount = codeArea.GetLineFromCharIndex(codeArea.TextLength) + 1;

if (linecount != maxLineCount)

{

displayLineNumber.Clear();

//loops for every line in codeArea

for (int i = 1; i < linecount + 1; i++)

{

displayLineNumber.AppendText(Convert.ToString(i) + "\r\n");

}

maxLineCount = linecount;

}

//refreshes line numbers after scroll

displayLineNumber.Refresh();

}

private void debugButton\_Click(object sender, EventArgs e)

{

debugFlag = 0;

forExecution();

if (CommandParser.breakLoopFlag == 0)

{

runCode.Enabled = true;

}

else

{

runCode.Enabled = false;

}

}

}

}

**LoadCommand.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace GraphicalProgrammingLanguage

{

public class LoadCommand : MCommand

{

private Menu menu;

RichTextBox codeArea;

public LoadCommand(Menu menuItems, RichTextBox ogCodeArea)

{

menu = menuItems;

codeArea = ogCodeArea;

}

public void Execute()

{

menu.Load(codeArea);

}

}

}

**MCommand.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace GraphicalProgrammingLanguage

{

public interface MCommand

{

void Execute();

}

}

**Menu.cs**

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.IO;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace GraphicalProgrammingLanguage

{

public class Menu

{

OpenFileDialog fileExplorer = new OpenFileDialog();

SaveFileDialog saveFile = new SaveFileDialog();

/// <summary>

/// load the contents of the selected text file to the RichTextBox that is passed

/// </summary>

/// <param name="codeArea">RichTextBox to display the contents of the file that is loaded</param>

public void Load(RichTextBox codeArea)

{

//filters for only text files

fileExplorer.Filter = "Text|\*.txt|All|\*.\*";

//title for fie explorer

fileExplorer.Title = "Choose your file";

fileExplorer.FilterIndex = 1;

fileExplorer.InitialDirectory = @"C:\Users\DELL\OneDrive\Desktop\fourth year\semester 1\Advanced Software Engineering\";

//remembers last visited directory

fileExplorer.RestoreDirectory = true;

try

{

if (fileExplorer.ShowDialog() == DialogResult.OK)

{

codeArea.Text = ""; //clears text box before loading file contents

StreamReader reader = File.OpenText(fileExplorer.FileName);

do

{

String line = reader.ReadLine();

if (line == null) break; //breaks if end of file

codeArea.Text += line;

codeArea.AppendText(Environment.NewLine); // new line starting points are preserved

} while (true);

reader.Close();

}

}

catch (System.ArgumentException)

{

MessageBox.Show("NO FILE CHOSEN !!", "ALERT", MessageBoxButtons.OK, MessageBoxIcon.Warning);

}

catch (FileNotFoundException)

{

MessageBox.Show("FILE NOT FOUND !!", "ERROR", MessageBoxButtons.OK, MessageBoxIcon.Exclamation);

}

catch (IOException)

{

MessageBox.Show("Something went wrong, try again !!", "ERROR", MessageBoxButtons.OK, MessageBoxIcon.Error);

}

}

/// <summary>

/// save the contents of the RichTextBox to a text file in a location specified by the user

/// </summary>

/// <param name="codeArea">contains the contents that is to be read ans saved</param>

public void Save(RichTextBox codeArea)

{

try

{

//checks if a file is open and if its is, saves it to the same file

using (StreamWriter outputFile = File.CreateText(fileExplorer.FileName))

{

// Write the info to the file.

outputFile.WriteLine(codeArea.Text);

outputFile.Close();

String filename = Path.GetFileName(fileExplorer.FileName);

String message = "Work saved to : " + filename;

message += "\nLocation: " + fileExplorer.FileName;

MessageBox.Show(message, "ALERT", MessageBoxButtons.OK, MessageBoxIcon.Information);

}

}

catch (System.ArgumentException)

{

//prompts user to save file, if no file is open

saveFile.RestoreDirectory = true;

saveFile.Title = "Where do you want to save your work?";

saveFile.InitialDirectory = @"C:\Users\DELL\OneDrive\Desktop\fourth year\semester 1\Advanced Software Engineering\";

//saves file filter type

saveFile.Filter = "Text|\*.txt|All|\*.\*";

try

{

//displays dialog box and checks if user has selected a file

if (saveFile.ShowDialog() == DialogResult.OK)

{

StreamWriter fWriter = File.CreateText(saveFile.FileName);

fWriter.WriteLine(codeArea.Text);

fWriter.Close();

String filename = Path.GetFileName(saveFile.FileName);

String message = "Work saved to : " + filename;

//displays the location where the file was saved

message += "\nLocation: " + saveFile.FileName;

MessageBox.Show(message, "ALERT", MessageBoxButtons.OK, MessageBoxIcon.Information);

}

}

catch (IOException)

{

MessageBox.Show("Error", "IO exception");

}

}

}

/// <summary>

/// exit applicaiton

/// </summary>

public void Close()

{

Application.Exit();

}

}

}

**MenuOptions.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace GraphicalProgrammingLanguage

{

public class MenuOptions

{

private MCommand loadCommand;

private MCommand saveCommand;

private MCommand closeCommand;

public MenuOptions(MCommand load, MCommand save, MCommand close)

{

this.loadCommand = load;

this.saveCommand = save;

this.closeCommand = close;

}

public void clickLoad()

{

loadCommand.Execute();

}

public void clickSave()

{

saveCommand.Execute();

}

public void clickClose()

{

closeCommand.Execute();

}

}

}

**NegativeNumberException.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace GraphicalProgrammingLanguage

{

class NegativeNumberException: Exception

{

public NegativeNumberException(String message) : base(message)

{

}

}

}

**Polygon.cs**

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace GraphicalProgrammingLanguage

{

public class Polygon : Shape

{

public Polygon() : base()

{

}

public Polygon(Color colour, int x, int y, int[] polyArray) : base(colour, x, y)

{

this.polyArray = polyArray;

}

public override void setPoly(Color colour, Boolean fill, bool flash, Color primaryColor, Color secondaryColor, int xAxis, int yAxis, int[] polyArray)

{

base.colour = colour;

base.set(colour, fill, flash, primaryColor, secondaryColor, xAxis, yAxis);

base.polyArray = polyArray;

}

public override void draw(Graphics graphics, bool fill)

{

//created two separate arrays to store X and Y coordinates

int[] polyArrayX = new int[30];

int[] polyArrayY = new int[30];

//to store the total elements in polyArray (from base)

int totalCoor = 0;

//to store new coordinates

List<Point> pointList = new List<Point>();

SolidBrush brush = new SolidBrush(Color.Transparent);

Pen pen = new Pen(base.colour, 2);

if (base.fill == true)

{

brush = new SolidBrush(base.colour);

}

else

{

brush = new SolidBrush(Color.Transparent);

}

//separating X and Y coordinates using index (i.e. even index for X and odd for Y)

for (int index = 0; index < polyArray.Length; index++)

{

if (index % 2 == 0)

{

polyArrayX[index] = polyArray[index];

}

else

{

polyArrayY[index] = polyArray[index];

}

totalCoor++;

}

//added on of the points to be the current posi of moveTo

pointList.Add(new Point(x, y));

for (int index = 0; index < totalCoor; index++)

{

try

{

//take only evenly placed elements

if (index % 2 == 0)

{

//doesn't work if 1 is not added to polyArrayY[index] because of the way points are stored in the list

pointList.Add(new Point(polyArrayX[index], polyArrayY[index + 1]));

}

}

catch (IndexOutOfRangeException)

{

Debug.WriteLine("pointList.Add garda IndexOutOfRange");

}

}

//convert the list to an array as DrawPolygon only accepts array

Point[] pointArray = pointList.ToArray();

graphics.DrawPolygon(pen, pointArray);

graphics.FillPolygon(brush, pointArray);

}

}

}

**Rectangle.cs**

using System;

using System.Collections.Generic;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace GraphicalProgrammingLanguage

{

class Rectangle : Shape

{

int width, height;

public Rectangle()

{

}

public Rectangle(Color colour, int x, int y, int width, int height) : base(colour, x, y)

{

this.width = width; //the only thing that is different from shape

this.height = height;

}

public override void set(Color colour, Boolean fill, bool flash, Color primaryColor, Color secondaryColor, params int[] list)

{

base.colour = colour;

//list[0] is x, list[1] is y, list[2] is width, list[3] is height

base.set(colour, fill, flash, primaryColor, secondaryColor, list[0], list[1]);

this.width = list[2];

this.height = list[3];

}

public override void draw(Graphics graphics, Boolean fill)

{

SolidBrush brush = new SolidBrush(Color.Transparent);

Pen pen = new Pen(base.colour, 2);

if (base.fill == true)

{

brush = new SolidBrush(base.colour);

}

else

{

brush = new SolidBrush(Color.Transparent);

}

graphics.FillRectangle(brush, x - width, y - height, width \* 2, height \* 2);

graphics.DrawRectangle(pen, x - width, y - height, width \* 2, height \* 2);

}

public override string ToString() //all classes inherit from object and ToString() is abstract in object

{

return base.ToString() + " " + this.height + " " + this.width;

}

}

}

**SaveCommand.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace GraphicalProgrammingLanguage

{

public class SaveCommand : MCommand

{

private Menu menu;

RichTextBox codeArea;

public SaveCommand(Menu menuItems, RichTextBox ogCodeArea)

{

menu = menuItems;

codeArea = ogCodeArea;

}

public void Execute()

{

menu.Save(codeArea);

}

}

}

**Shape.cs**

using System;

using System.Collections.Generic;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace GraphicalProgrammingLanguage

{

public abstract class Shape : ShapeInterface

{

public Color colour = Color.Black;

public Boolean fill;

protected int x, y;

protected int[] polyArray = new int[40];

public Color primaryColor;

public Color secondaryColor;

public bool flash;

public Shape()

{

}

public Shape(Color colour, int x, int y)

{

this.colour = colour; //shape's colour

this.x = x; //its x pos

this.y = y; //its y pos

}

public abstract void draw(Graphics graphics, Boolean fill);

public void draw(Graphics graphics, Color color)

{

throw new NotImplementedException();

}

public virtual void set(Color colour, Boolean fill, bool flash, Color primaryColor, Color secondaryColor, params int[] list)

{

this.fill = fill;

this.colour = colour;

this.x = list[0];

this.y = list[1];

this.primaryColor = primaryColor;

this.secondaryColor = secondaryColor;

this.flash = flash;

}

public virtual void setPoly(Color colour, Boolean fill, bool flash, Color primaryColor, Color secondaryColor, int x, int y, int[] polyArray)

{

this.fill = fill;

this.colour = colour;

this.x = x;

this.y = y;

this.polyArray = polyArray;

}

public override string ToString()

{

return base.ToString() + " " + this.x + "," + this.y + " : ";

}

}

}

**ShapeFactory.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace GraphicalProgrammingLanguage

{

public class ShapeFactory

{

public Shape getShape(String shapeType)// parameter for shape passed

{

shapeType = shapeType.ToUpper().Trim(); //yoi could argue that you want a specific word string to create an object but I'm allowing any case combination

if (shapeType.Equals("CIRCLE"))

{

return new Circle();

}

else if (shapeType.Equals("RECTANGLE"))

{

return new Rectangle();

}

else if (shapeType.Equals("TRIANGLE"))

{

return new Triangle();

}

else if (shapeType.Equals("DRAWTO"))

{

return new DrawTo();

}

else if (shapeType.Equals("POLYGON"))

{

return new Polygon();

}

else

{

throw new System.ArgumentException("Factory error: " + shapeType + " does not exist");

}

}

}

}

**ShapeInterface.cs**

using System;

using System.Collections.Generic;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace GraphicalProgrammingLanguage

{

interface ShapeInterface

{

void set(Color c, Boolean fill, bool flash, Color primaryColor, Color secondaryColor, params int[] list); // no need to declare the size of array if "params" is used

void draw(Graphics g, Color color);

}

}

**Triangle.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Drawing;

using System.Text;

using System.Threading.Tasks;

namespace GraphicalProgrammingLanguage

{

public class Triangle : Shape

{

int hypotenuse, perpendicular;

public Triangle() : base()

{

}

public Triangle(Color colour, int x, int y, int hypotenuse, int perpendicular) : base(colour, x, y)

{

this.hypotenuse = hypotenuse;

this.perpendicular = perpendicular;

}

public override void set(Color colour, Boolean fill, bool flash, Color primaryColor, Color secondaryColor, params int[] list)

{

base.colour = colour;

//list[0] is x, list[1] is y, list[2] is hypotenuse, list[3] is perpendicular

base.set(colour, fill, flash, primaryColor, secondaryColor, list[0], list[1]);

this.hypotenuse = list[2];

this.perpendicular = list[3];

}

public override void draw(Graphics graphics, Boolean fill)

{

Point[] point = new Point[3];

SolidBrush brush = new SolidBrush(Color.Transparent);

Pen pen = new Pen(base.colour, 2);

if (base.fill == true)

{

brush = new SolidBrush(base.colour);

}

else

{

brush = new SolidBrush(Color.Transparent);

}

//right point

point[0].X = x + perpendicular;

point[0].Y = y + hypotenuse;

//top point

point[1].X = x;

point[1].Y = y - (hypotenuse);

//left point

point[2].X = x - perpendicular;

point[2].Y = y + hypotenuse;

graphics.DrawPolygon(pen, point);

graphics.FillPolygon(brush, point);

}

}

}

**CheckShape.cs**

using System;

using System.Collections;

using System.Collections.Generic;

using System.Data;

using System.Diagnostics;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace GraphicalProgrammingLanguage

{

class CheckShape

{

CustomMethods custom = new CustomMethods();

public static bool flash = false;

public static Color primaryColor;

public static Color secondaryColor;

/// <summary>

/// checks for basic shapes and commands like moveto, drawto, pen and fill

/// </summary>

/// <param name="singleLine">a single line from the mainDictionary</param>

/// <param name="varDictionary">dictionary that contains all variables and their values</param>

/// <param name="factory">object of the factory class that returns the actual shape</param>

/// <param name="shapes">ArrayList that contains all the shapes</param>

/// <param name="errorDisplayBox">to display errors</param>

/// <param name="lineNumber">current line number</param>

/// <param name="shape">object of type Shape</param>

/// <param name="polyArray">array that contains all the coordinates to draw the polygon</param>

public void checkForShape(string[] singleLine, Dictionary<string, int> varDictionary, ShapeFactory factory, ArrayList shapes, RichTextBox errorDisplayBox, int lineNumber, Shape shape, int[] polyArray)

{

//checks for moveto

if ((string)singleLine[0].ToUpper() == "MOVETO")

{

//gets the current length of line

int countArrayNum = singleLine.Length;

// checks if the required number of parameters are met (i.e. array.length - 1 = number of parameters)

if (countArrayNum - 1 == 2)

{

string[] parameter = CustomMethods.getValueFromDictionary(varDictionary, singleLine[1], singleLine[2]);

//checks if both the parameters passed are integers

try

{

if (isPositiveNumber(int.Parse(parameter[0])) && isPositiveNumber(int.Parse(parameter[1])))

{

//stores params as coordinates

CommandParser.penX = int.Parse(parameter[0]);

CommandParser.penY = int.Parse(parameter[1]);

}

}

catch (IndexOutOfRangeException)

{

CommandParser.breakLoopFlag = 1;

}

catch (FormatException)

//catch params that are not integers

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Both parameters should be integer", "MOVETO x,y");

CommandParser.breakLoopFlag = 1;

CommandParser.breakFlag = 1;

}

catch (NegativeNumberException)

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Parameters should be positive integer", "MOVETO x,y");

CommandParser.breakLoopFlag = 1;

CommandParser.breakFlag = 1;

}

}

else

{

//if wrong number of parameters are passed

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Wrong number of parameters for keyword MOVETO", "MOVETO x,y");

CommandParser.breakLoopFlag = 1;

CommandParser.breakFlag = 1;

}

}

if ((string)singleLine[0].ToUpper() == "CIRCLE")

{

//gets the current length of line

int countArrayNum = singleLine.Length;

// checks if the required number of parameters are met (i.e. array.length - 1 = number of parameters)

if (countArrayNum - 1 == 1)

{

string[] parameter = CustomMethods.getValueFromDictionary(varDictionary, singleLine[1]);

try

{

if (isPositiveNumber(int.Parse(parameter[0])))

{

int radius = int.Parse(parameter[0]); // stores radius

custom.getAndAddShape(CommandParser.color, CommandParser.fill, flash, primaryColor, secondaryColor, factory, (string)singleLine[0].ToUpper(), shapes, CommandParser.penX, CommandParser.penY, radius);//creates and adds the shape

}

}

catch (IndexOutOfRangeException)

{

CommandParser.breakLoopFlag = 1;

}

catch (FormatException)

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Parameter should be of type integer", "CIRCLE radius");

CommandParser.breakLoopFlag = 1;

CommandParser.breakFlag = 1;

}

catch (NegativeNumberException)

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Parameters should be positive integer", "CIRCLE radius");

CommandParser.breakLoopFlag = 1;

CommandParser.breakFlag = 1;

}

}

//check if required number of parameters are passed

else

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Wrong number of parameters for keyword CIRCLE", "CIRCLE radius");

CommandParser.breakLoopFlag = 1;

CommandParser.breakFlag = 1;

}

}

//checks for RECTANGLE

if ((string)singleLine[0].ToUpper() == "RECTANGLE")

{

//gets the current length of line

int countArrayNum = singleLine.Length;

// checks if the required number of parameters are met (i.e. array.length - 1 = number of parameters)

if (countArrayNum - 1 == 2)

{

string[] parameter = CustomMethods.getValueFromDictionary(varDictionary, singleLine[1], singleLine[2]);

try

{

//isPositiveNumber(int.Parse(parameter[0]));

//if (int.Parse(parameter[0]) >= 0 && int.Parse(parameter[1]) >= 0)

if (isPositiveNumber(int.Parse(parameter[0])) && isPositiveNumber(int.Parse(parameter[1])))

{

int height = int.Parse(parameter[0]);

int width = int.Parse(parameter[1]);

custom.getAndAddShape(CommandParser.color, CommandParser.fill, flash, primaryColor, secondaryColor, factory, (string)singleLine[0].ToUpper(), shapes, CommandParser.penX, CommandParser.penY, height, width);//creates and adds the shape

}

}

catch (IndexOutOfRangeException)

{

CommandParser.breakLoopFlag = 1;

}

catch (FormatException)

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Both parameters should be of type integer", "RECTANGLE height, width");

CommandParser.breakLoopFlag = 1;

CommandParser.breakFlag = 1;

}

catch (NegativeNumberException)

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Parameters should be positive integer", "RECTANGLE height, width");

CommandParser.breakLoopFlag = 1;

CommandParser.breakFlag = 1;

}

}

else

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Wrong number of parameters for keyword RECTANGLE", "RECTANGLE height, width");

CommandParser.breakLoopFlag = 1;

CommandParser.breakFlag = 1;

}

}

//checks for TRIANGLE

if ((string)singleLine[0].ToUpper() == "TRIANGLE")

{

int countArrayNum = singleLine.Length;

if (countArrayNum - 1 == 2)

{

string[] parameter = CustomMethods.getValueFromDictionary(varDictionary, singleLine[1], singleLine[2]);

try

{

if (isPositiveNumber(int.Parse(parameter[0])) && isPositiveNumber(int.Parse(parameter[1])))

{

int bases = int.Parse(parameter[0]);

int height = int.Parse(parameter[1]);

custom.getAndAddShape(CommandParser.color, CommandParser.fill, flash, primaryColor, secondaryColor, factory, (string)singleLine[0].ToUpper(), shapes, CommandParser.penX, CommandParser.penY, bases, height);//creates and adds the shape

}

}

catch (IndexOutOfRangeException)

{

CommandParser.breakLoopFlag = 1;

}

catch (FormatException)

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Both parameters should be of type integer", "TRIANGLE base, height");

CommandParser.breakLoopFlag = 1;

CommandParser.breakFlag = 1;

}

catch (NegativeNumberException)

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Parameters should be positive integer", "TRIANGLE base, height");

CommandParser.breakLoopFlag = 1;

CommandParser.breakFlag = 1;

}

}

else

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Wrong number of parameters for keyword TRIANGLE", "TRIANGLE base, height");

CommandParser.breakLoopFlag = 1;

CommandParser.breakFlag = 1;

}

}

if ((string)singleLine[0].ToUpper() == "POLYGON")

{

int countArrayNum = singleLine.Length;

shape = factory.getShape((string)singleLine[0].ToUpper());

//must have atleast 4 coordinates

if ((((countArrayNum - 1) % 2) == 0) && (countArrayNum - 1 >= 4))

{

polyArray = new int[countArrayNum - 1];

string[] parameter = new string[singleLine.Length - 1];

//errorDisplayBox.Text += "\nparameter array size : " + parameter.Length;

for (int index = 0; index < singleLine.Length - 1; index++)

{

string tempVar = singleLine[index + 1].Trim().ToUpper();

if (varDictionary.ContainsKey(tempVar))

{

int valueOfOperand = varDictionary[tempVar];

parameter[index] = tempVar.Replace(tempVar, valueOfOperand.ToString());

}

else

{

parameter[index] = tempVar;

}

}

for (int index = 0; index < parameter.Length; index++)

{

try

{

if (isPositiveNumber(int.Parse(parameter[0])))

{

polyArray[index] = int.Parse(parameter[index]);

}

}

catch (ArgumentNullException)

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "ARGUMENT NULL EXCEPTION", "");

CommandParser.breakLoopFlag = 1;

CommandParser.breakFlag = 1;

}

catch (FormatException)

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Parameter '" + parameter[index] + "' was never declared", "");

CommandParser.breakLoopFlag = 1;

CommandParser.breakFlag = 1;

}

catch (NegativeNumberException)

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Parameters should be positive integer", "Polygon x,y,x,y");

CommandParser.breakLoopFlag = 1;

CommandParser.breakFlag = 1;

}

}

shape.setPoly(CommandParser.color, CommandParser.fill, flash, primaryColor, secondaryColor, CommandParser.penX, CommandParser.penY, polyArray);

shapes.Add(shape);

}

else

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Wrong number of parameters for keyword POLYGON", "POLYGON 23,2,32,5 (must be in pairs (i.e divisible by two))");

CommandParser.breakLoopFlag = 1;

CommandParser.breakFlag = 1;

}

}

//checks for DRAWTO

if ((string)singleLine[0].ToUpper() == "DRAWTO")

{

int countArrayNum = singleLine.Length;

if (countArrayNum - 1 == 2)

{

string[] parameter = CustomMethods.getValueFromDictionary(varDictionary, singleLine[1], singleLine[2]);

try

{

//isPositiveNumber(int.Parse(parameter[0]))

if (isPositiveNumber(int.Parse(singleLine[1])) && isPositiveNumber(int.Parse(singleLine[2])))

{

//if 'moveTo' is written before 'drawTo'

int drawFromX = CommandParser.penX;

int drawFromY = CommandParser.penY;

//store params

int drawToX = int.Parse(parameter[0]);

int drawToY = int.Parse(parameter[1]);

//the new co-ordinates for MOVETO params

CommandParser.penX = drawToX;

CommandParser.penY = drawToY;

//creates and adds the shape

custom.getAndAddShape(CommandParser.color, CommandParser.fill, flash, primaryColor, secondaryColor, factory, (string)singleLine[0].ToUpper(), shapes, drawFromX, drawFromY, drawToX, drawToY);

//makes the ending point of the previous drawTo the starting point of the next drawTo

drawFromX = drawToX;

drawFromY = drawToY;

}

}

catch (IndexOutOfRangeException)

{

CommandParser.breakLoopFlag = 1;

}

catch (FormatException)

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Both parameters should be of type integer", "DRAWTO x,y");

CommandParser.breakLoopFlag = 1;

CommandParser.breakFlag = 1;

}

catch (NegativeNumberException)

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Parameters should be positive integer", "DRAWTO x,y");

CommandParser.breakLoopFlag = 1;

CommandParser.breakFlag = 1;

}

}

else

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Wrong number of parameters for keyword DRAWTO", "DRAWTO x,y");

CommandParser.breakLoopFlag = 1;

CommandParser.breakFlag = 1;

}

}

if ((string)singleLine[0].ToUpper() == "PEN")

{

int countArrayNum = singleLine.Length;

if (countArrayNum - 1 == 1)

{

if (singleLine[1].ToUpper() == "RED" || singleLine[1].ToUpper() == "YELLOW" || singleLine[1].ToUpper() == "BLUE" || singleLine[1].ToUpper() == "REDGREEN" || singleLine[1].ToUpper() == "BLACKWHITE" || singleLine[1].ToUpper() == "BLUEYELLOW")

{

if (singleLine[1].ToUpper() == "RED")

{

CommandParser.color = Color.Red;

}

if (singleLine[1].ToUpper() == "YELLOW")

{

CommandParser.color = Color.Yellow;

}

if (singleLine[1].ToUpper() == "BLUE")

{

CommandParser.color = Color.Blue;

}

if (singleLine[1].ToUpper() == "REDGREEN")

{

primaryColor = Color.Red;

secondaryColor = Color.Green;

flash = true;

}

if (singleLine[1].ToUpper() == "BLACKWHITE")

{

primaryColor = Color.Black;

secondaryColor = Color.White;

flash = true;

}

if (singleLine[1].ToUpper() == "BLUEYELLOW")

{

primaryColor = Color.Blue;

secondaryColor = Color.Yellow;

flash = true;

}

}

else

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Such color doesnt exist ", "red OR green OR blue");

CommandParser.breakLoopFlag = 1;

CommandParser.breakFlag = 1;

}

}

else

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Wrong number of parameters for keyword DRAWTO", "PEN red");

CommandParser.breakLoopFlag = 1;

CommandParser.breakFlag = 1;

}

}

if ((string)singleLine[0].ToUpper() == "FILL")

{

int countArrayNum = singleLine.Length;

if (countArrayNum - 1 == 1)

{

if (singleLine[1].ToUpper() == "ON" || singleLine[1].ToUpper() == "OFF")

{

if (singleLine[1].ToUpper() == "ON")

{

CommandParser.fill = true;

}

if (singleLine[1].ToUpper() == "OFF")

{

CommandParser.fill = false;

}

}

else

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Wrong parameter for keyword FILL", "FILL on");

CommandParser.breakLoopFlag = 1;

CommandParser.breakFlag = 1;

}

}

else

{

custom.displayErrorMsg(errorDisplayBox, lineNumber, "Wrong number of parameters for keyword FILL", "FILL on");

CommandParser.breakLoopFlag = 1;

CommandParser.breakFlag = 1;

}

}

}

/// <summary>

/// method that throws user defined exception if negative number is passed

/// </summary>

/// <param name="param">any integer</param>

/// <returns></returns>

static bool isPositiveNumber(int param)

{

if (param < 0)

{

throw new NegativeNumberException("The prameter has to be a positive number");

}

else

{

return true;

}

}

}

}