Advanced Software Engineering (ASE)

Component 1

StudentID:**77261187**

Student Name: **Ayushma Tamrakar**

1. Introduction

This component consist of producing a simplified application for teaching simple programming concepts. The applications interface consist of command area for writing commands/ program, canvas for displaying output of program, a console where error message are displayed after checking syntax , an action command area where we write action command such as reset, run and clear. There are two menu items in menu strip file and help. File consist of sub menu with options save, load, and exit. All the commands written in command area can be saved as text file by clicking save option. Commands can also be loaded from file explorer by clicking load. Application can also be closed by clicking exit option. Position x and y shows current position of pen in canvas.

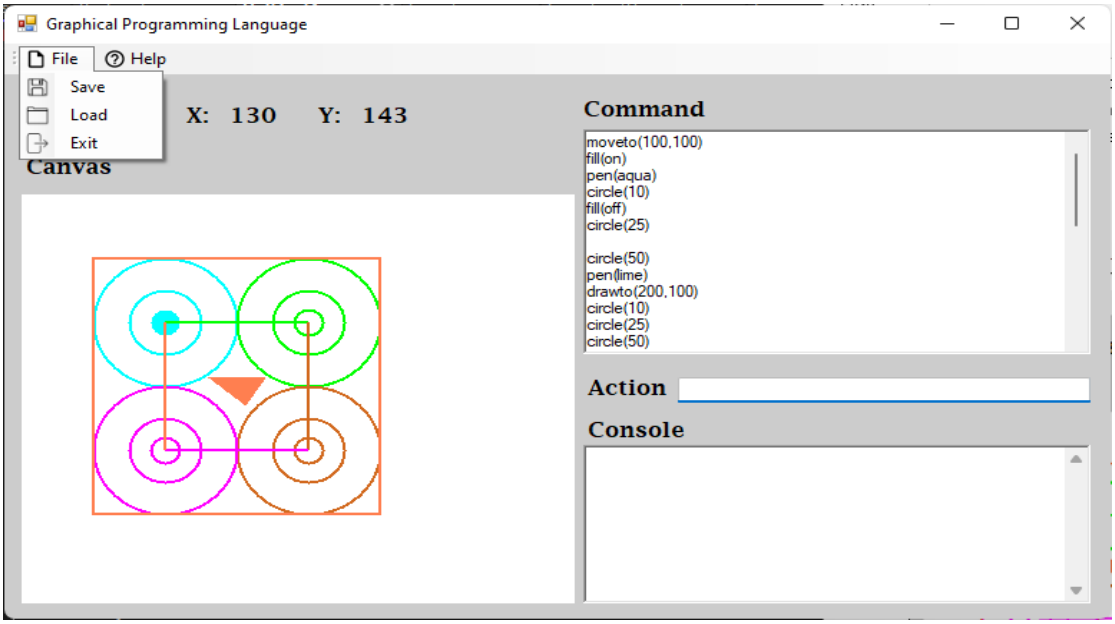

Fig 1. Interface of application.

2. Methodology

Agile methodology is a technique that encourages ongoing testing and development throughout the project's software development lifecycle. This project is developed using agile methodology. Firstly, requirements are gathered from module handbook. Then UML diagram is made to show work of new features in existing system. After defining requirements interface as well as other functionality are added iteratively to application. Functionalities are also tested through unit testing.

3. UML (Class Diagram)

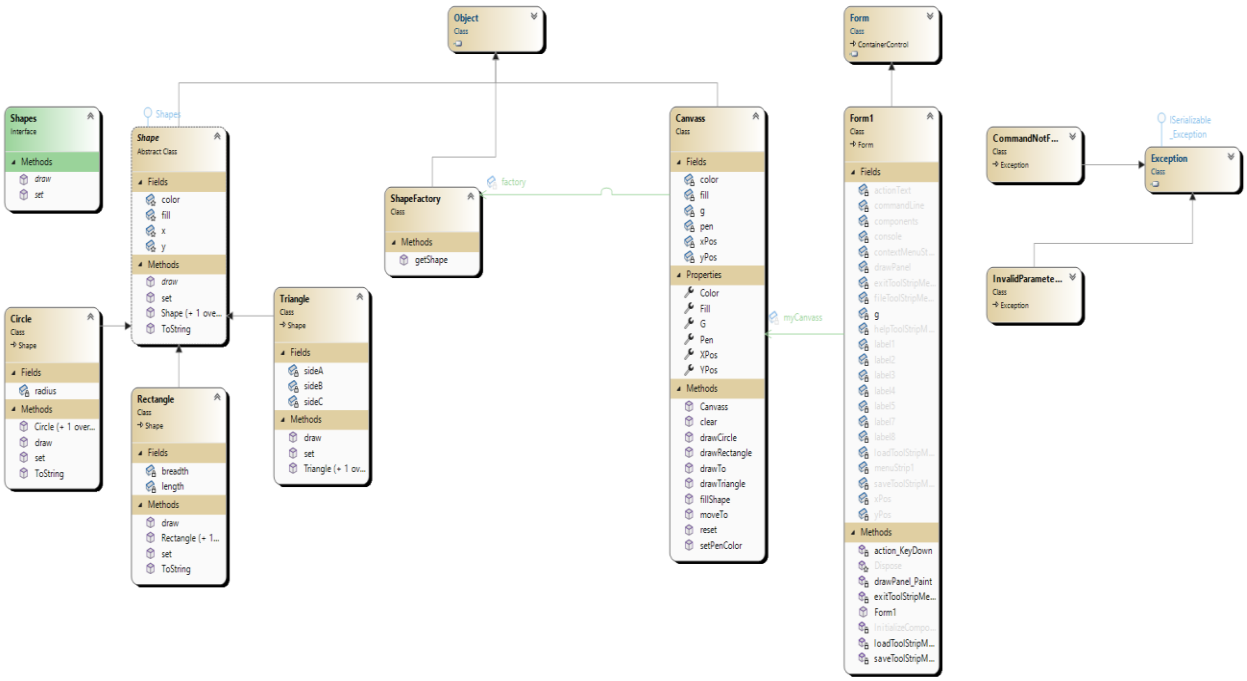

Fig 2. UML Class Diagram.

4. Version Control

Version Control done in bit bucket.

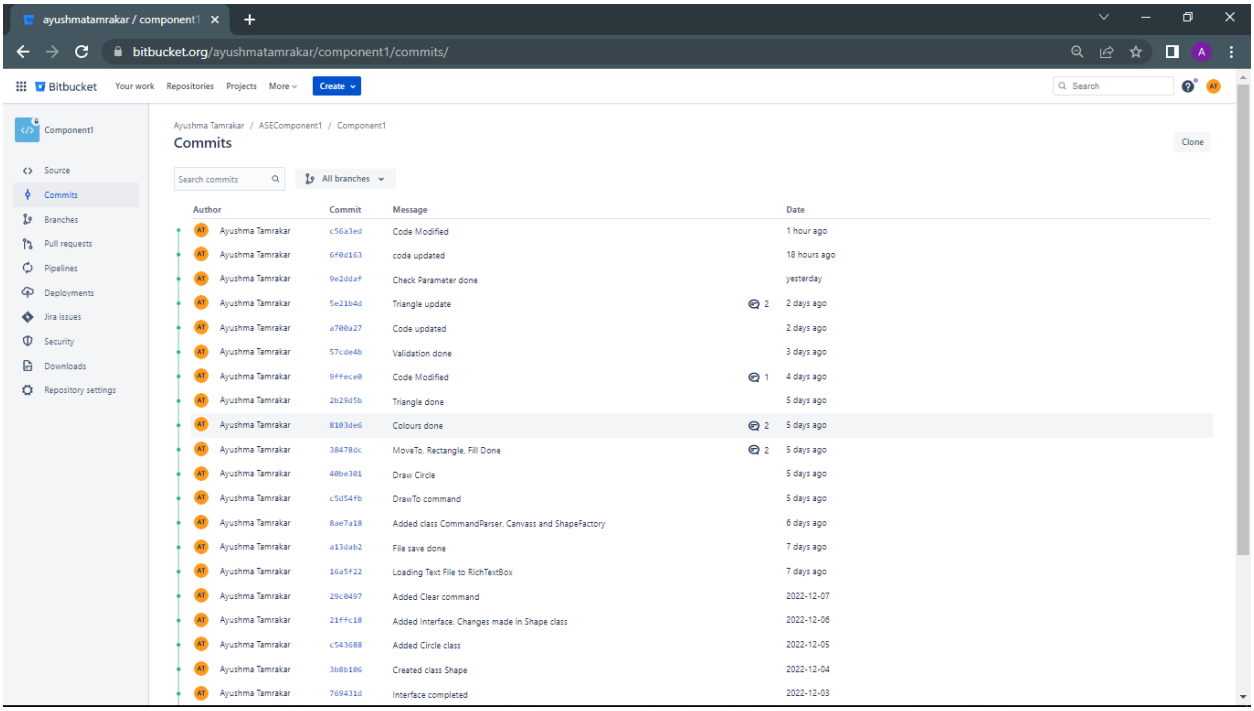

Fig 3. Version Control in BitBucket.

5. Commands Used in Project

This component uses seven drawing commands: circle, rectangle, drawto, moveto, triangle, fill, and pen as well as three action commands: run, reset and clear. All drawing commands and action commands are case-insensitive. Drawing commands requires one or more parameter enclosed in parentheses. Action commands are written in action text box and gets executed when pressing enter or return key.

A. Drawing Commands

a. Circle:

Circle command takes one parameter which is radius an integer.

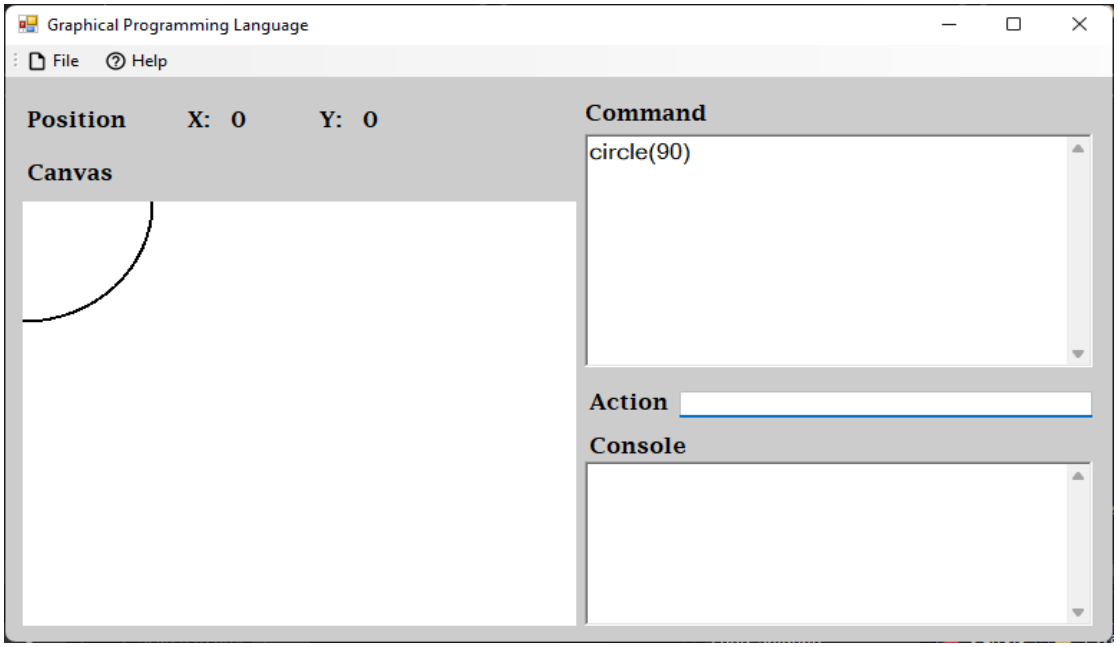

Fig 4. Circle with 90 radius drawn at (0, 0).

b. Rectangle:

Rectangle command takes two parameters: length and breadth.

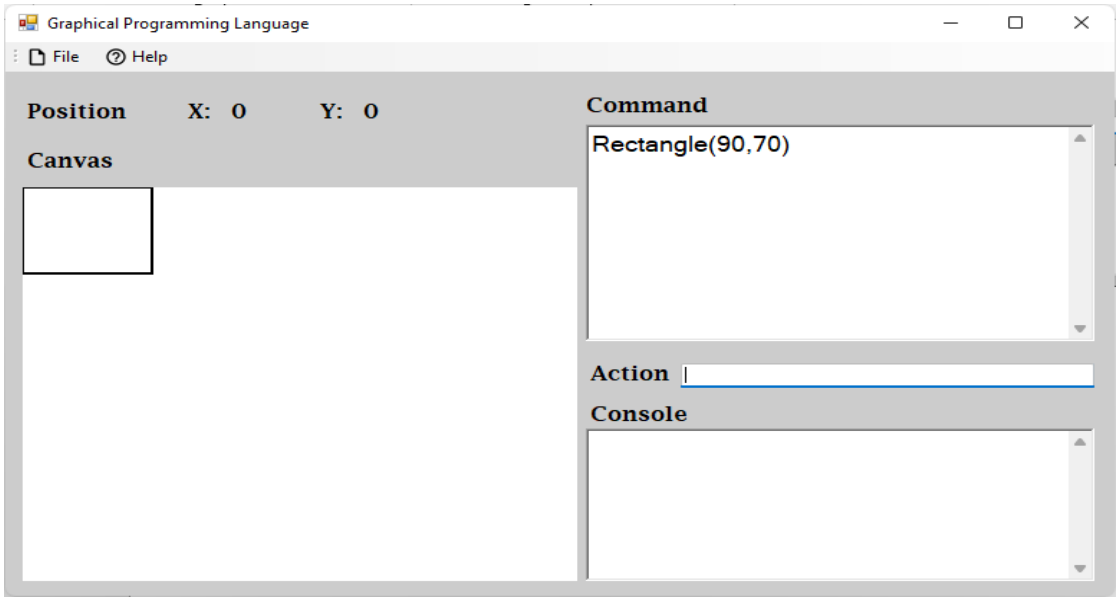

Fig 5. Rectangle with length 90 and breadth 70 drawn at (0, 0)

c. Triangle:

Triangle command takes three parameters: PointA, PointB and PointC.

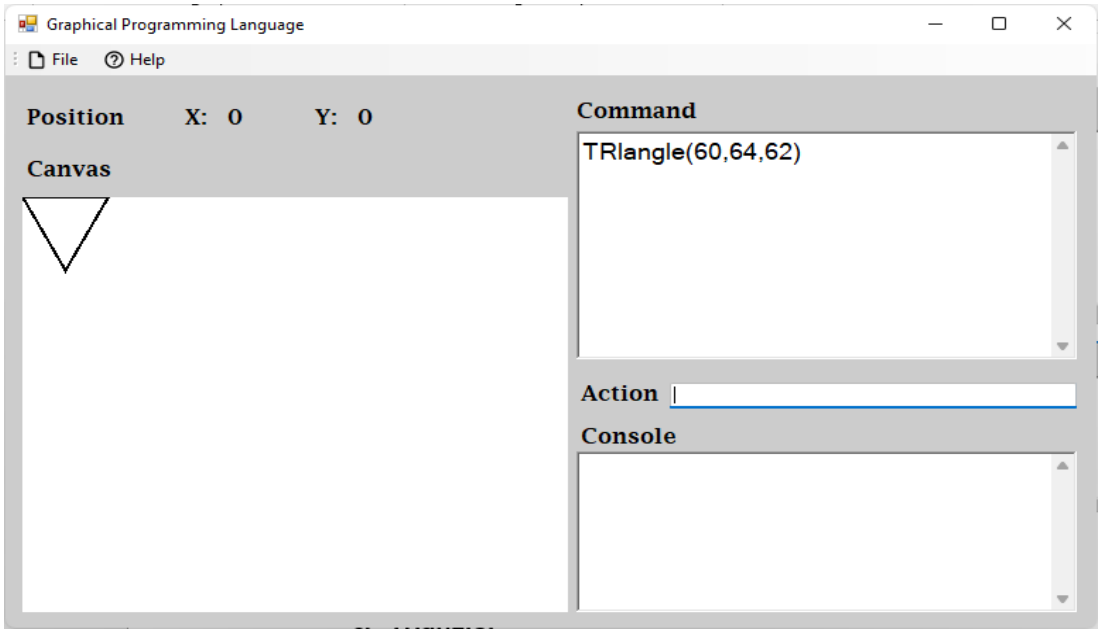

Fig 6. Triangle with PointsA, PointsB and PointsC drawn at (0, 0)

d. MoveTo:

MoveTo command takes two parameter: XPos and YPos. It allows to move points by setting value for x-axis and y-axis.

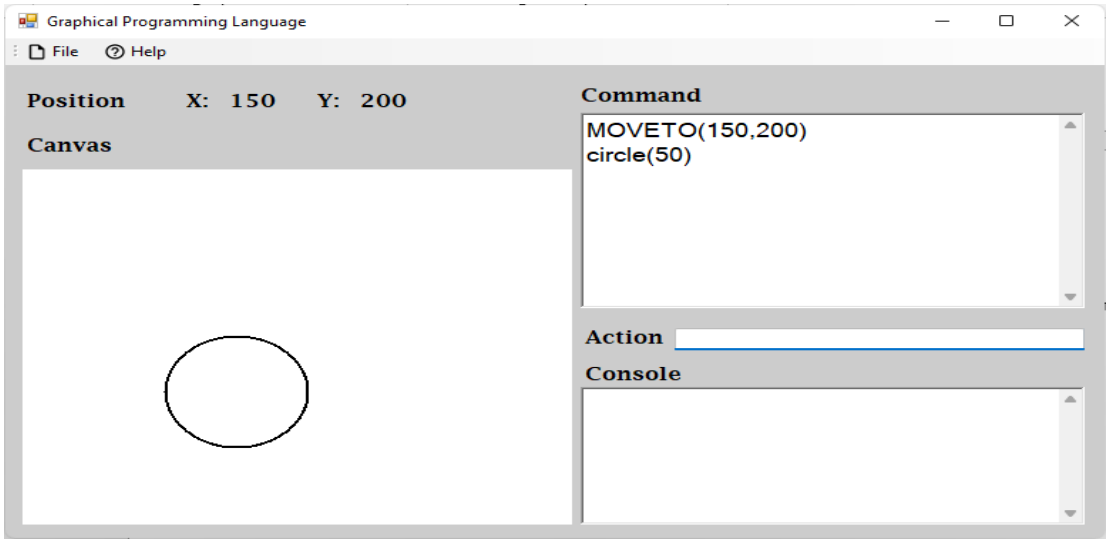

Fig 7. Circle drawn at point (150,200) using moveto command

e. DrawTo:

DrawTo command takes two parameter: PointA and PointB. This command is used to drawline.

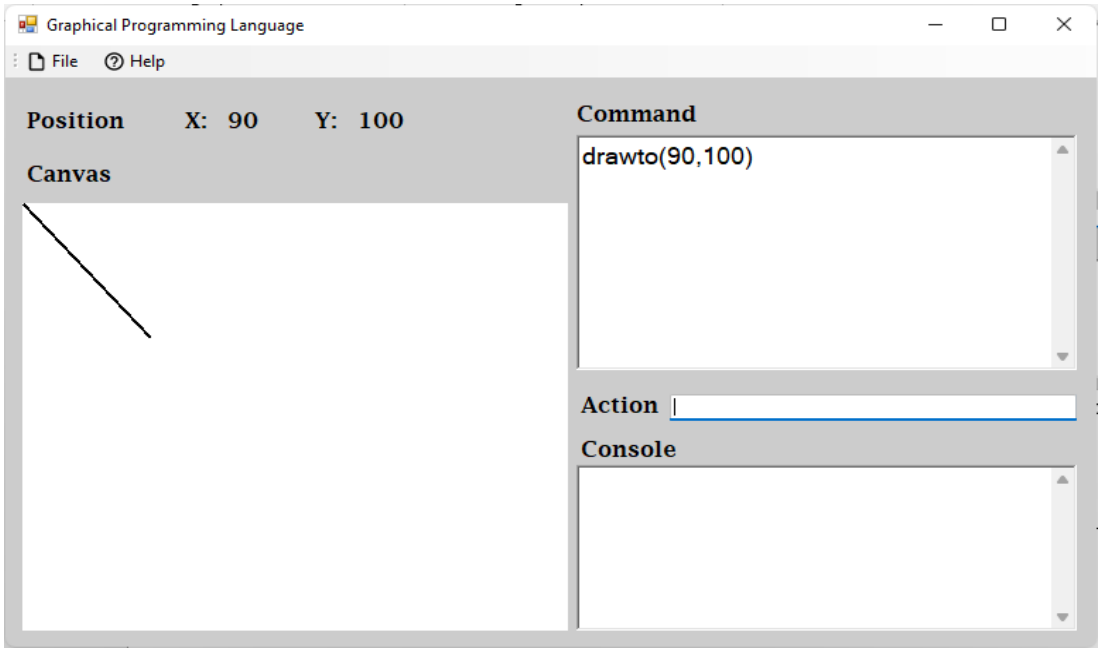

Fig 8. Line drawn from (0, 0) to (90, 100)

f. Fill:

Fill command takes one parameter: on or off. This command is used to fill color in shape.

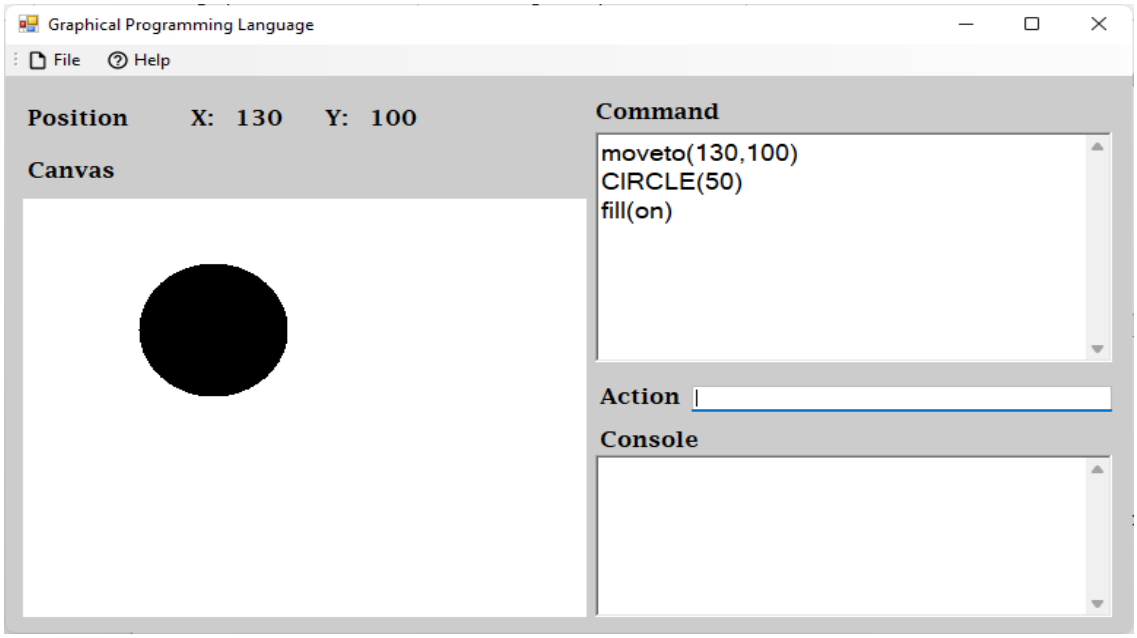

Fig 9. Circle with radius 50 drawn at (130, 100) with fill (on)

g. Pen:

Pen command takes one parameter: color. This command sets color of pen and brush.

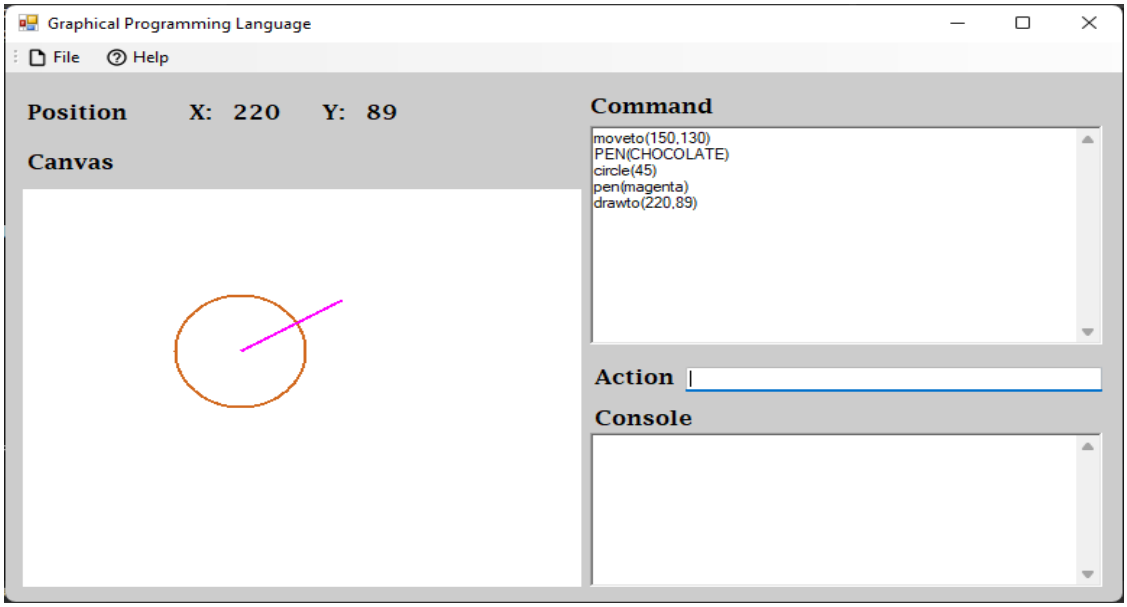

Fig 10.Circle with radius 45 with pen color “chocolate” drawn at (150,130) with line drawn to (220, 89)

B. Action Commands

a. Run:

Run command executes commands written in command text box.

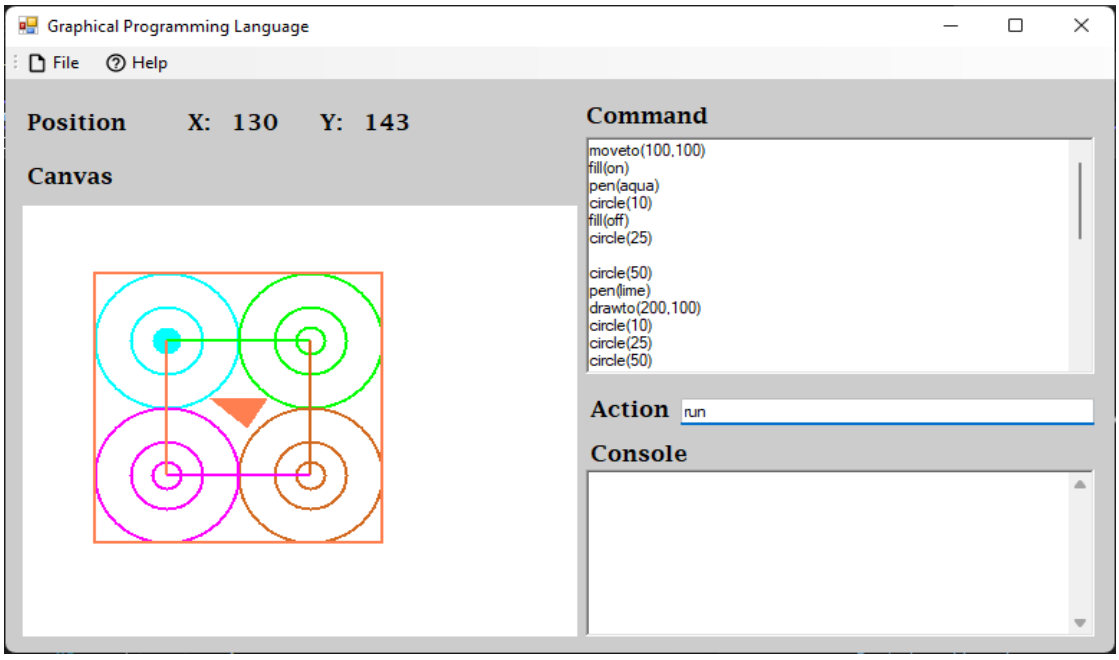

Fig 11. Run action command executing command in command box

b. Reset:

Reset command re-initialize position of x-axis and y-axis to (0, 0) and sets pen color to black.

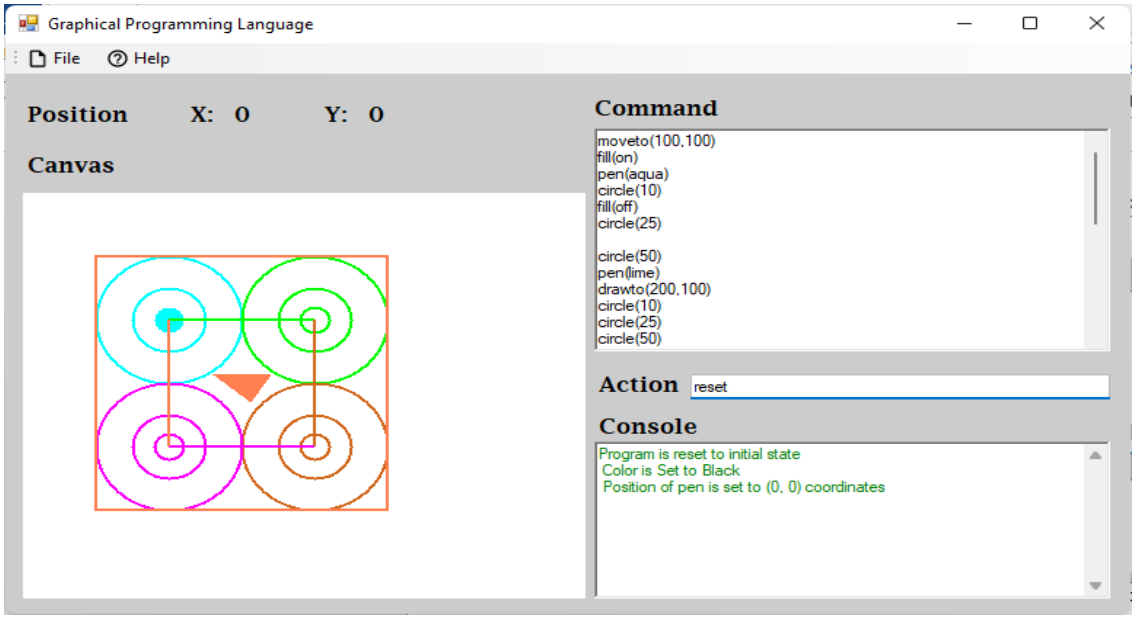

Fig 12. Reset action command re-initializing position and color

c. Clear:

Clear command clears drawings from canvas.

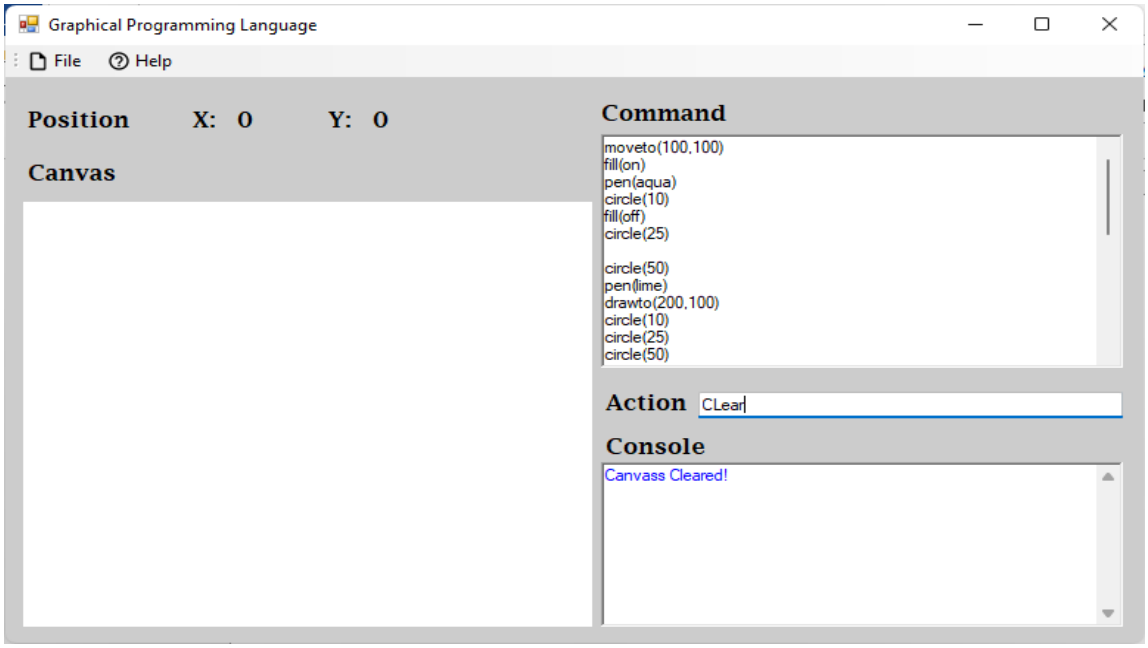

Fig 13. Clear action command clearing drawings from canvas

6. Error Message Shown In Application

Depending on the error that occurred with either drawing commands or action commands, different messages are displayed in the console.

A. Action Commands

a. Invalid Action command:

"Invalid command" displayed in console. Only three commands are valid in action commands: reset, clear and run.

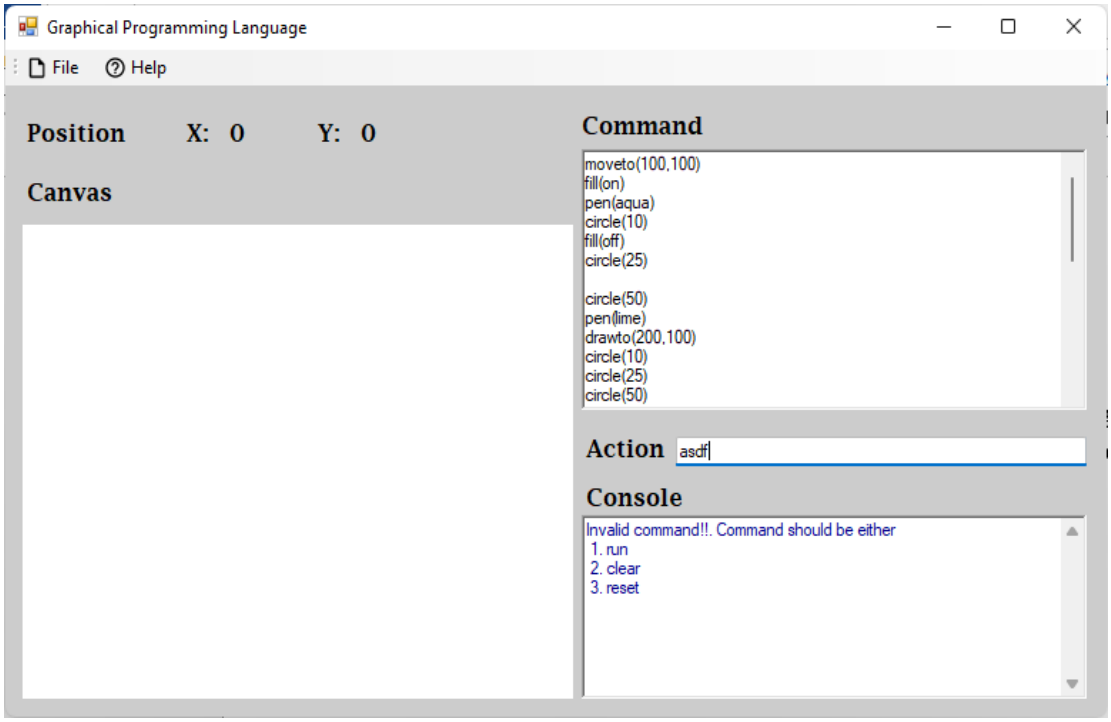

Fig 14. Invalid command message shown in console

B. Drawing Commands

a. No commands found

The console will display the message "No commands to run" if the command box is empty and there are no commands to execute.

Fig 15. No commands to run displayed in console

b. Parentheses Not Found

The console will display the message parentheses not found if commands do not have parentheses.

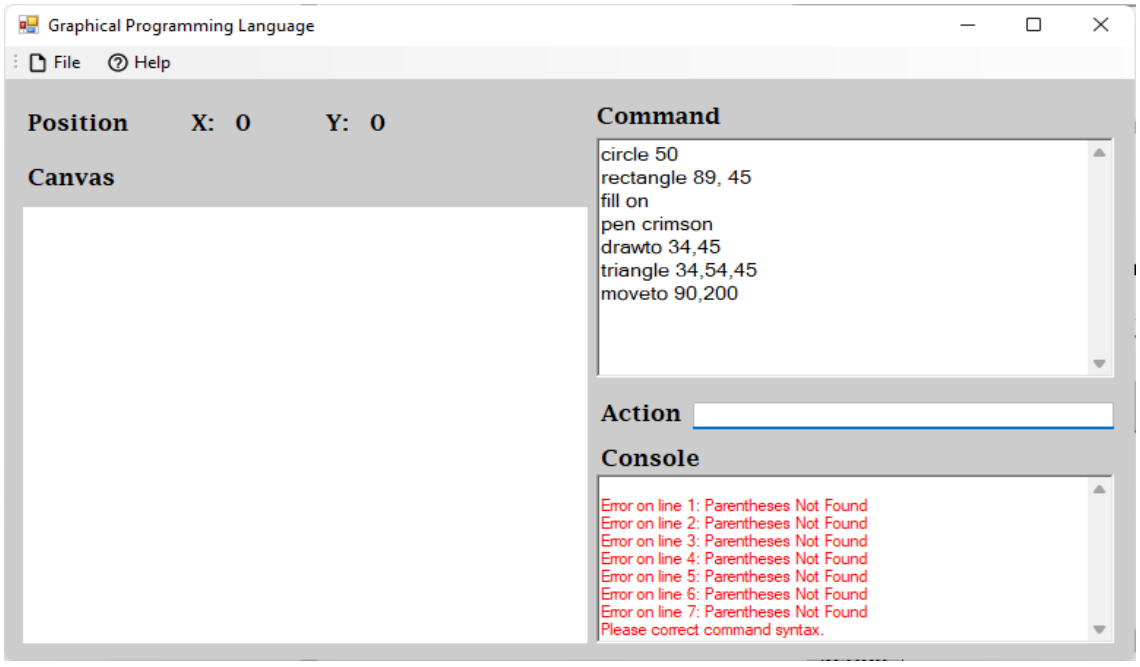

Fig 16. Parentheses not found displayed in console along with line number.

c. Left Parentheses Missing

Console will display “(Missing”, if the command does not have left parentheses.

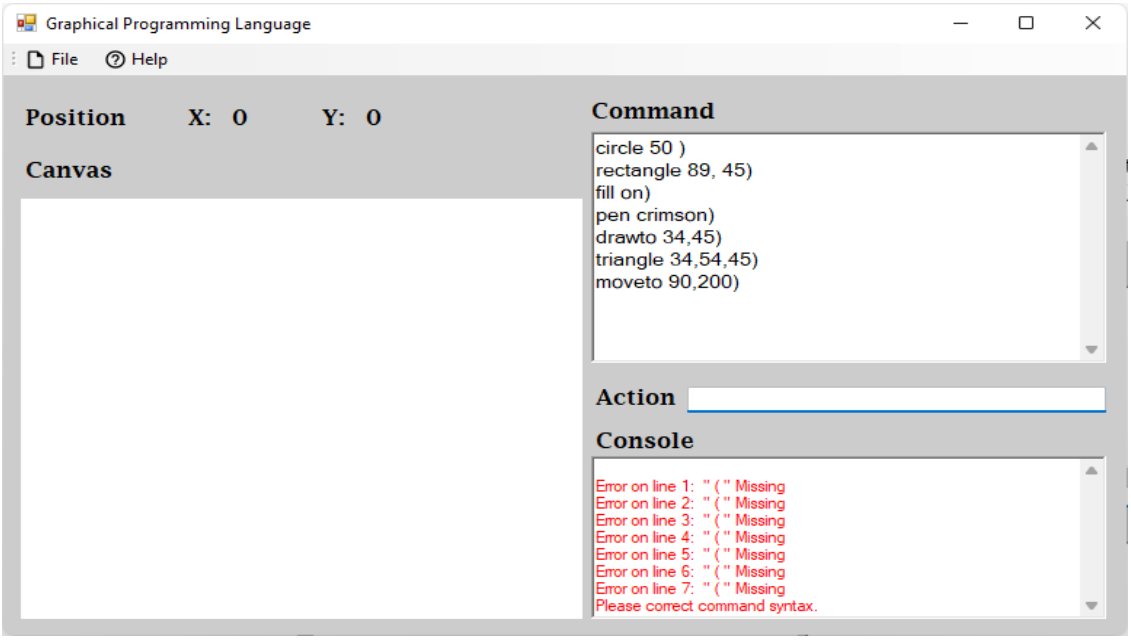

Fig 17. Left Parentheses missing displayed in console along with line number.

d. Right Parentheses Missing

Console will display “) Missing”, if the command does not have right parentheses.

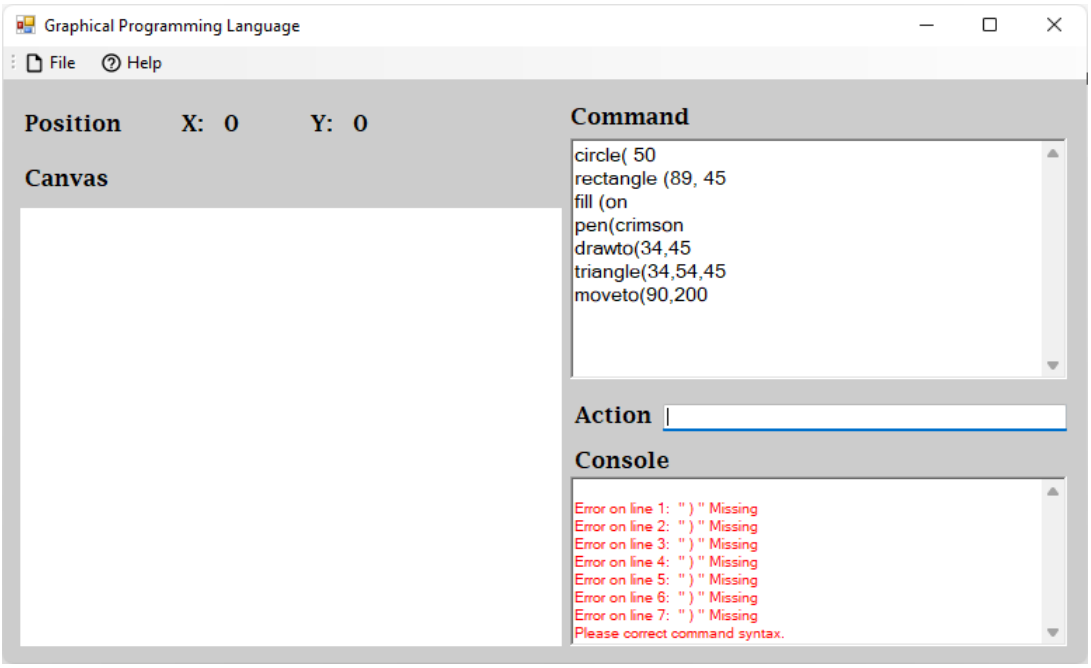

Fig 18. Right Parentheses missing displayed in console along with line number.

e. Invalid Command Name:

Console will display invalid command name if drawing commands are invalid.

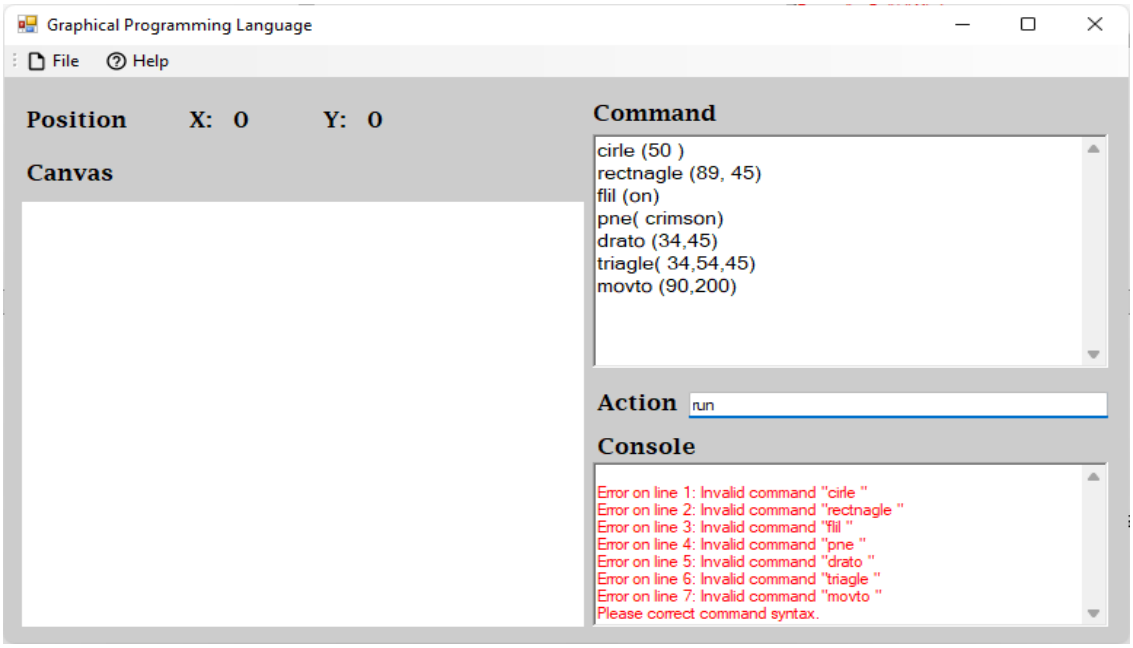

Fig 19. Invalid command name displayed in console along with line number.

f. Invalid Parameters:

i. Circle:

Program will check parameters for circle and display error message in console.

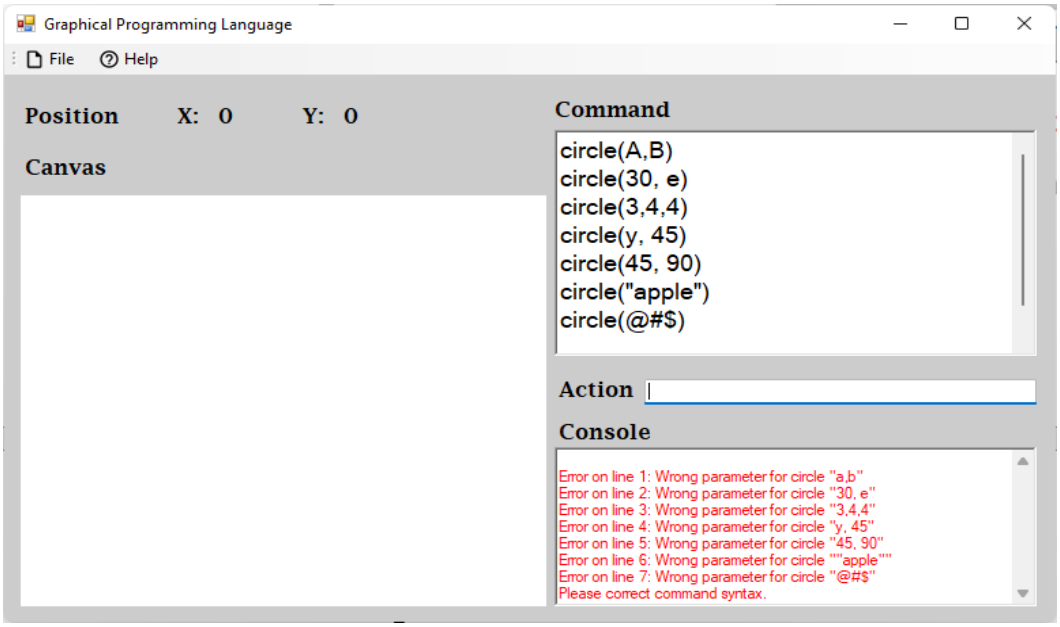

Fig 20. Invalid parameter of circle displayed in console along with line number.

ii. Rectangle:

Program will check parameters for rectangle and display error message in console.

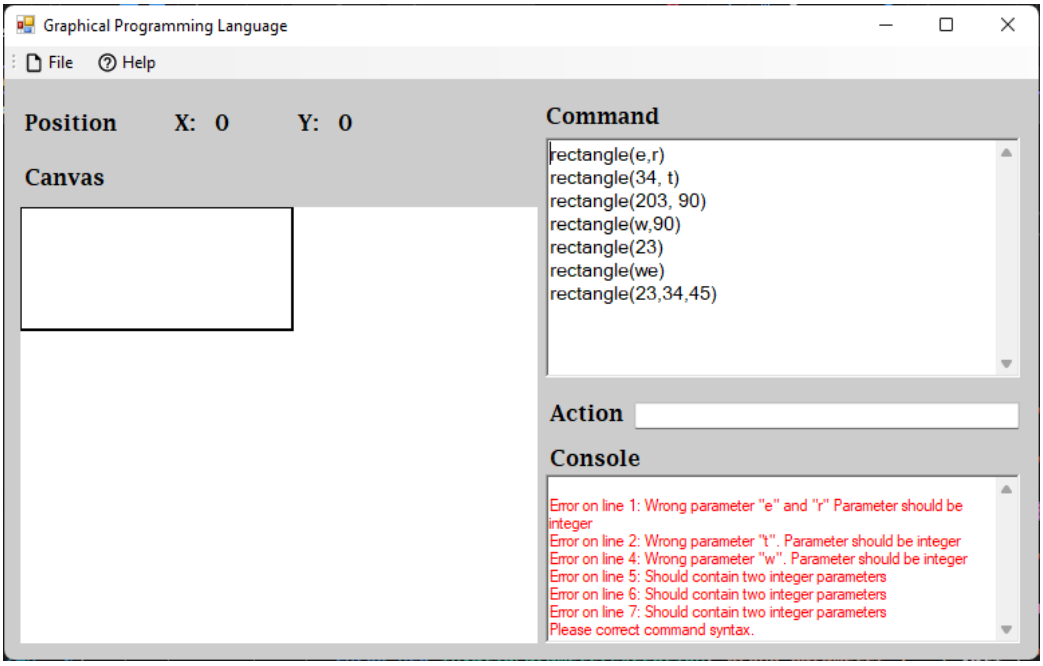

Fig 21. Invalid parameter of rectangle displayed in console along with line number. Line no 3 is valid command

iii. Triangle:

Program will check parameters for rectangle and display error message in console.

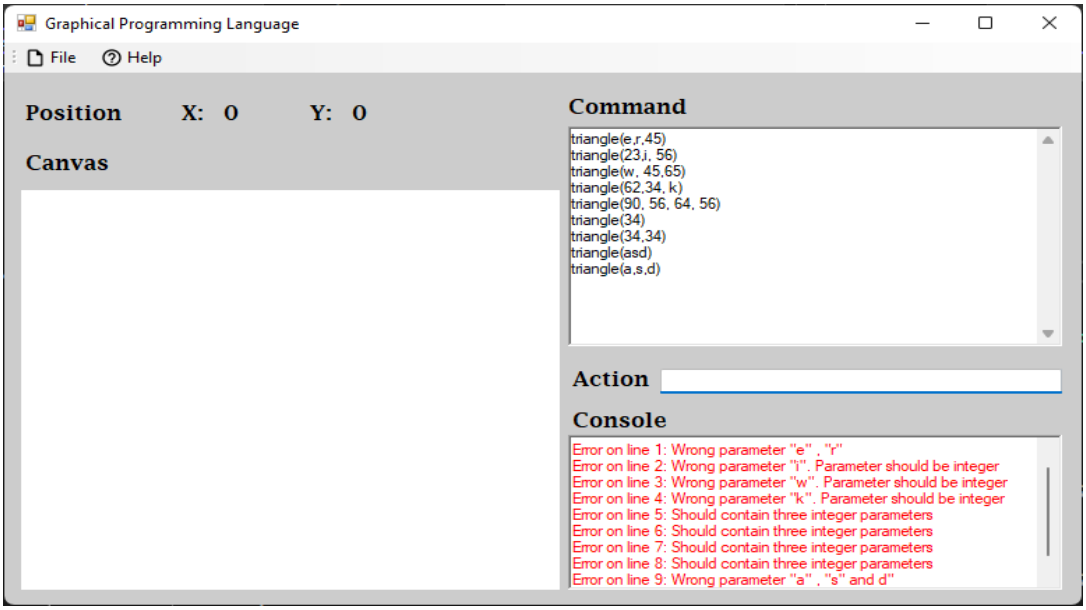

Fig 22. Invalid parameter of triangle displayed in console along with line number.

iv. DrawTo:

Program will check parameters for drawto and display error message in console.

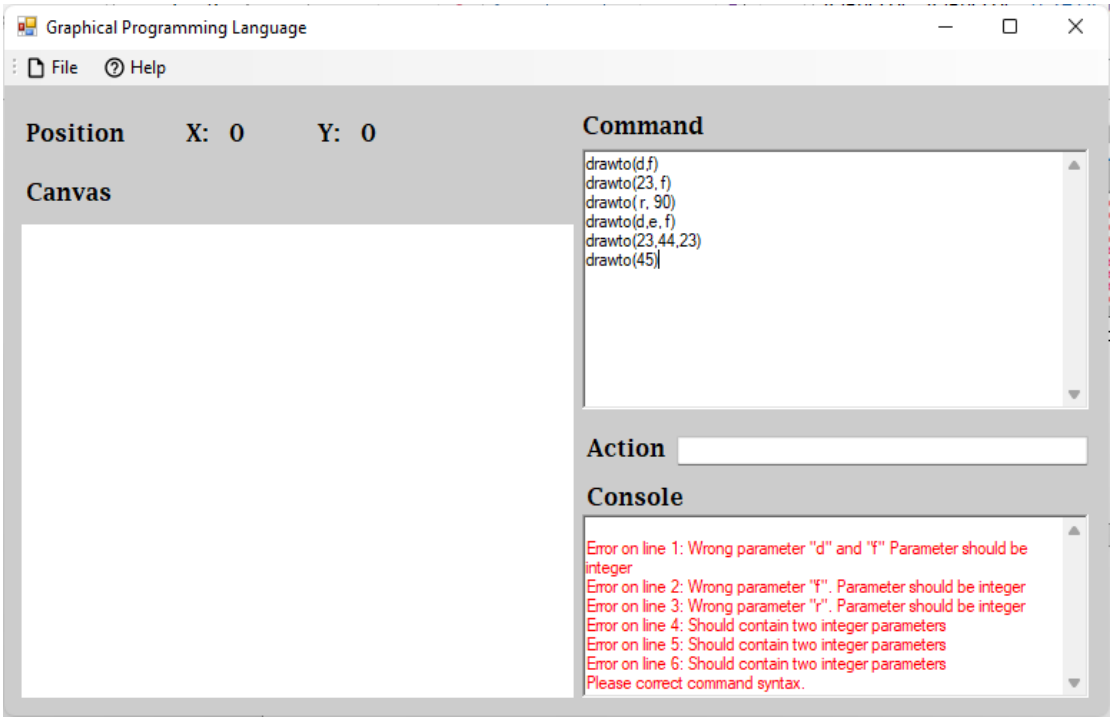

Fig 23. Invalid parameter of drawto displayed in console along with line number.

v. MoveTo:

Program will check parameters for moveTo and display error message in console.

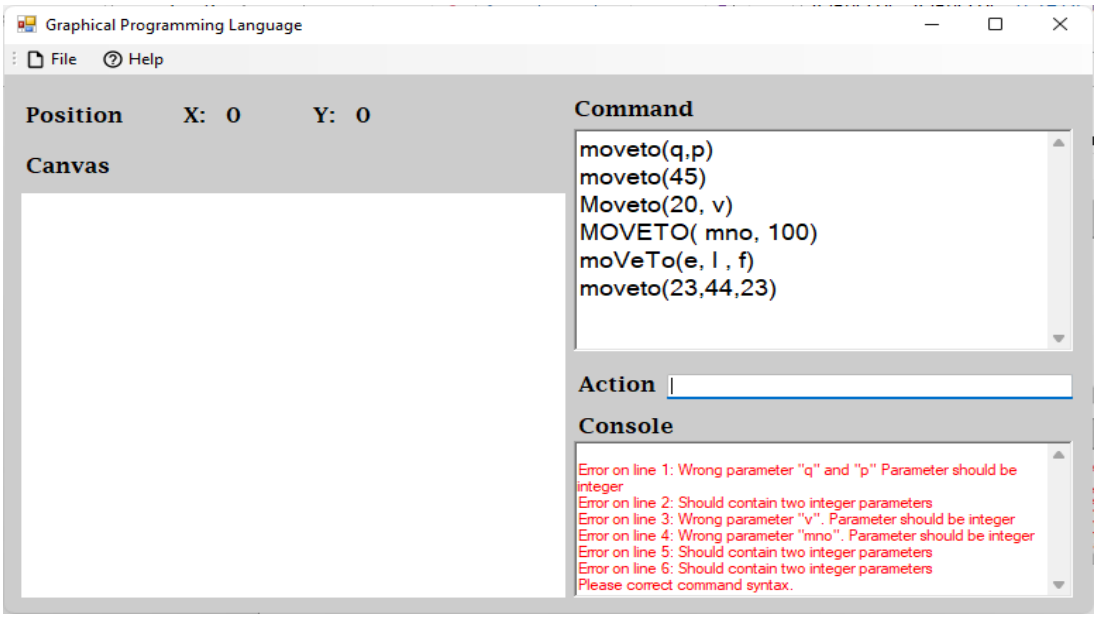

Fig 24. Invalid parameter of move to displayed in console along with line number.

vi. Fill:

Program will check parameters for fill command and display error message in console.

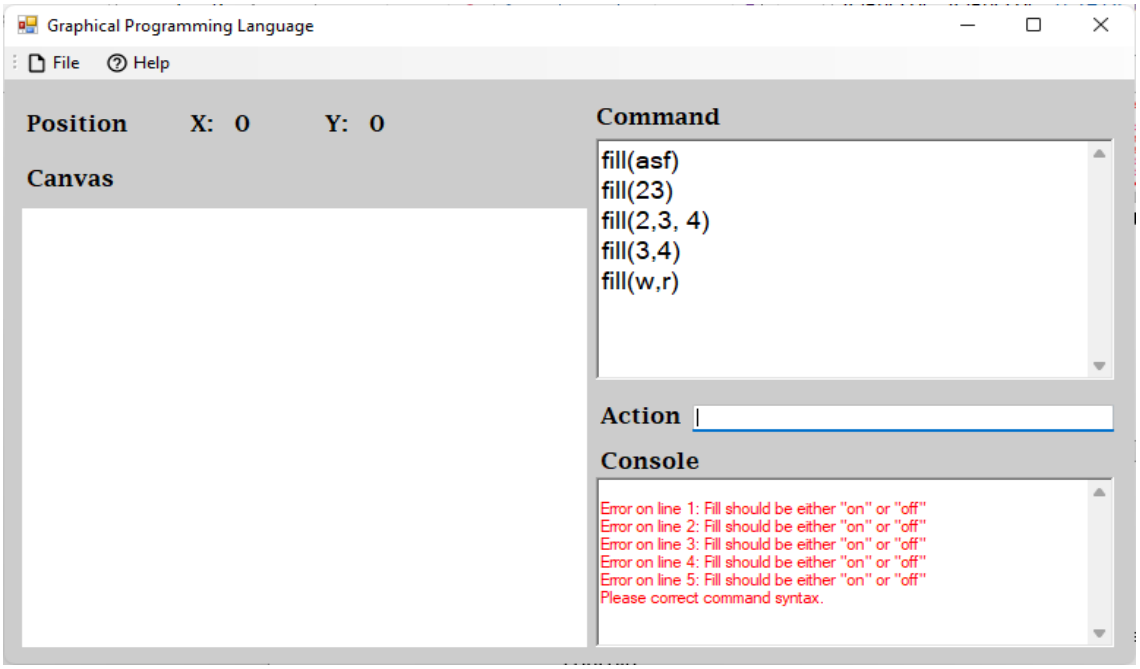

Fig 25. Invalid parameter of fill displayed in console along with line number.

vii. Pen:

Program will check parameters for pen command and display error message in console.

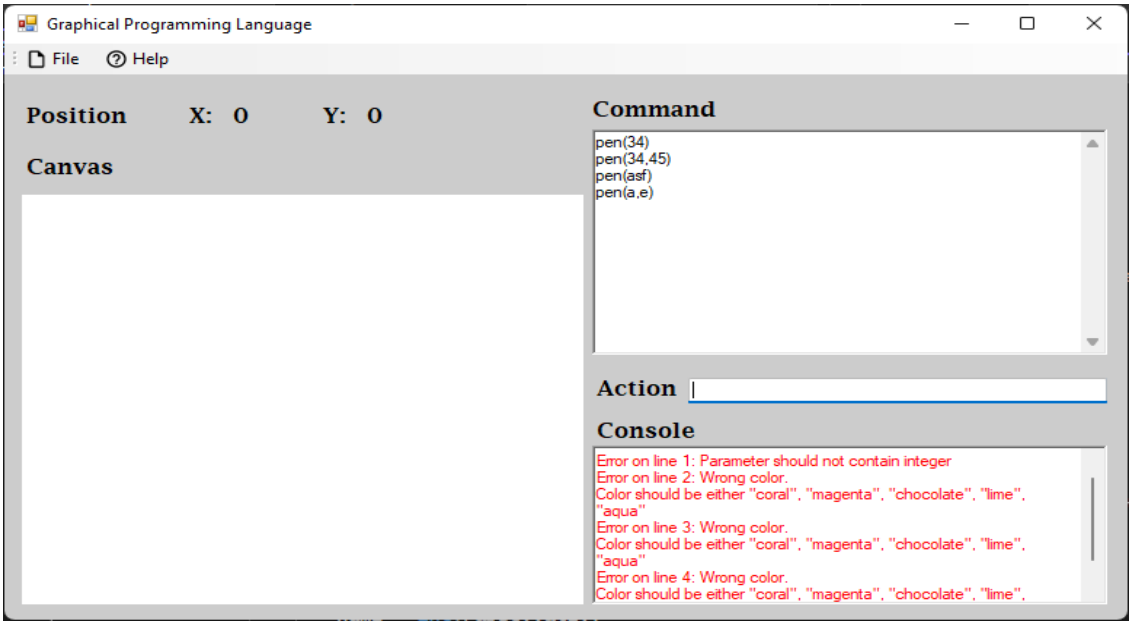

Fig 26. Invalid parameter of pen displayed in console along with line number.

7. Unit Test

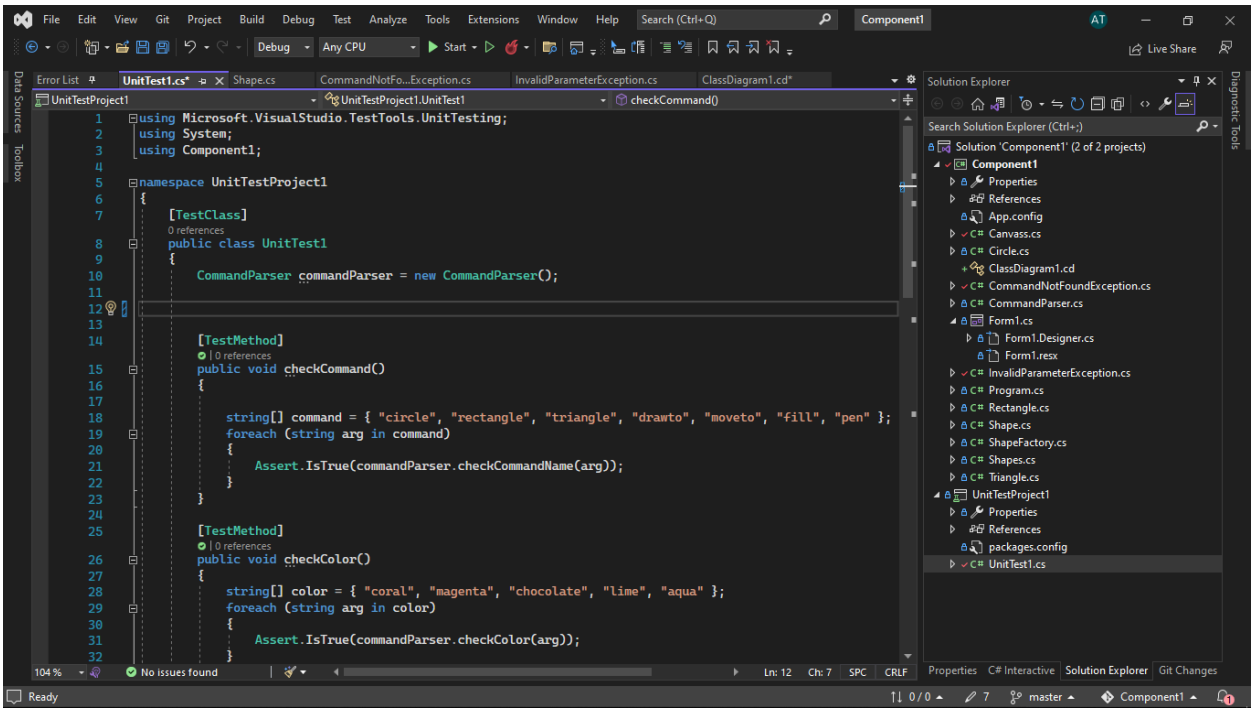

Fig 26. Unit Test Project added to project.

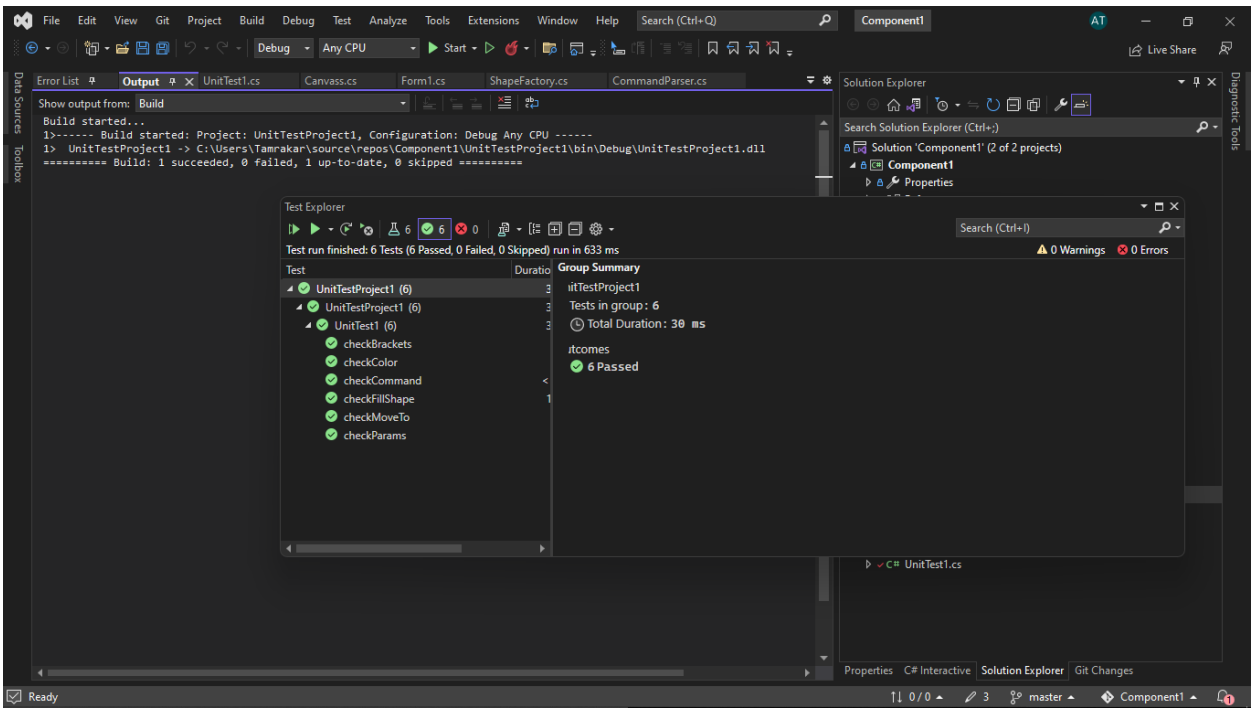

Fig 27. Showing all test result passed.

8. Unit Test Table

a. Test no 1: CheckCommand

SN.	Test Data
1.	circle
2.	rectangle
3.	Triangle
4.	drawto
5.	fill
6.	pen

Step	Step Details	Expected result	Actual Results	pass/Fail/Not Executed/ Suspended
1	String array was passed as parameter to checkCommandName method	It should return boolean value true	True value was returned	Pass

b. Test no 2: CheckColor

SN	Test Data
1.	coral
2.	magenta
3.	chocolate
4.	lime
5.	aqua

Step	Step Details	Expected result	Actual Results	pass/Fail/Not Executed/ Suspended
1	String array was passed as parameter to checkColor method	It should return boolean value true	True value was returned	Pass

c. Test no 3: CheckParameter

SN.	Test Data
1.	3, 4
2.	drawto

Step	Step Details	Expected result	Actual Results	pass/Fail/Not Executed/ Suspended
1	Two string was passed as parameter to checkParameter method	3	3	Pass
2	Two string was passed as parameter to checkParameter method	4	4	Pass

d. Test 4: CheckBrackets

SN.	Test Data
1.	Circle 90
2.	Circle(90)

Step	Step Details	Expected result	Actual Results	pass/Fail/Not Executed/ Suspended
1	string was passed as parameter to checkParentheses method	false	false	Pass
2	string was passed as parameter to checkParentheses method	true	true	Pass

e. Test 5: CheckMoveTo

SN.	Test Data
1.	2,3
2.	5,6

Step	Step Details	Expected result	Actual Results	pass/Fail/Not Executed/ Suspended
1	Integer parameter passed to moveto method are equal	2,3	2,3	Pass
2	Integer parameter passed to method are not equal	2, 3	5, 6	Pass

f. Test 6: Checkfill

SN.	Test Data
1.	true

Step	Step Details	Expected result	Actual Results	pass/Fail/Not Executed/ Suspended
1	Pass Boolean value to fillShape method	true	true	Pass

Codes

Shape Interface

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Component1
{
    internal interface Shapes
    {
        void set(Color color, bool colorFill, params int[] list);
        void draw(Graphics g);
    }
}
```

Abstract Shape

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Component1
{
internal abstract class Shape : Shapes
{
protected Color color;
protected int x, y;
protected bool fill;

public Shape()
{
//color = Color.Black;
x = y = 0;
fill = false;
}
public Shape(Color color, bool fill, int x, int y)
{
this.color = color;
this.x = x;
this.y = y;
this.fill = fill;
}
public abstract void draw(Graphics g);
public virtual void set(Color color, bool fill, params int[] list)
{
this.color = color;
this.fill = fill;
this.x = list[0];
this.y = list[1];

}
public override string ToString()
{
return base.ToString() + "    " + this.x + "," + this.y + " : ";
}
}
}
```

Circle

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Component1
{
    internal class Circle: Shape
    {
        int radius;
        public Circle() : base()
        {

        }

        public Circle(Color color, bool fill, int x, int y, int radius) : base(color,
fill, x, y)
        {
            this.radius = radius;
        }

        public override void draw(Graphics g)
        {
            Pen pen = new Pen(color, 2);
            SolidBrush brush = new SolidBrush(color);

            if (fill)
            {
                g.FillEllipse(brush, x, y, radius, radius);
            }
            else
            {
                g.DrawEllipse(pen, x, y, radius, radius);
            }

        }
        public override void set(Color colour,bool fill, params int[] list)
        {
            //list[0] is x, list[1] is y, list[2] is radius
            base.set(colour, fill, list[0], list[1]);
            this.radius = list[2];

        }
        public override string ToString() //all classes inherit from object and
ToString() is abstract in object
        {
            return base.ToString() + "  " + this.radius;
        }
    }


}
```

Canvass

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
```

```
namespace Component1
{
    public class Canvass
    {
        Graphics g;
        Pen pen;
        Color color;
        int xPos, yPos;
        bool fill = false;
        ShapeFactory factory = new ShapeFactory();



        public Canvass(Graphics g)
        {
            this.g = g;
            xPos = 0;
            yPos = 0;
            color = Color.Black;
            pen = new Pen(color, 2);
        }
        public Graphics G
        {
            get { return g; }
            set { g = value; }
        }
        public Pen Pen
        {
            get { return pen; }
            set { pen = value; }
        }
        public int XPos
        {
            get { return xPos; }
            set { xPos = value; }
        }
        public int YPos
        {
            set { yPos = value; }
            get { return yPos; }
        }
        public bool Fill
        {
            get { return fill; }
            set { fill = value; }
        }
        public Color Color
        {
            get { return color; }
            set { color = value; }
        }
        public void drawTo(int toX, int toY)
        {

            G.DrawLine(Pen, XPos, YPos, toX, toY);
            XPos = toX;
            YPos = toY;

        }


        public void drawCircle(int radius) //drawCircleMethod
        {
            shape = factory.getShape("circle");  //get shapes from Factory Class
            shape.set(Color,Fill, XPos-radius, YPos-radius, radius * 2, radius*2); //
sets value  in Set method of Shape class
            shape.draw(G);  // passing graphics to draw method
            shape.ToString();
        }
        public void drawRectangle(int length, int breadth)
        {
```

```
            shape = factory.getShape("rectangle");
            shape.set(Color, Fill, XPos, YPos, length, breadth);
            shape.draw(G);
            Console.WriteLine(shape.ToString());

        }
        public void drawTriangle(int sideA, int sideB, int sideC)
        {
            shape = factory.getShape("triangle");
           shape.set(Color, Fill, XPos, YPos, sideA, sideB, sideC);
            shape.draw(G);
            Console.WriteLine(shape.ToString());

        }
        public void fillShape(bool fill)
        {

            Fill = fill;
        }
        public void setPenColor(Color color)
        {
            pen = new Pen(color, 2);
            Color = color;

        }
        public void moveTo(int toX, int toY)
        {
            XPos = toX;
            YPos = toY;
        }

        public void reset()  // resets canvas
        {
            XPos = 0;
            YPos = 0;
            Color = Color.Black;
            Fill = false;

        }
        public void clear()  // clears canvas
        {
            G.Clear(Color.White);
        }
    }
}
```

CommandParser

```
using System;
using System.CodeDom;
using System.Collections;
using System.Drawing;
using System.Linq;
using System.Linq.Expressions;
using System.Security.AccessControl;
using System.Text.RegularExpressions;
using System.Windows.Forms;

namespace Component1
{
    public class CommandParser
    {


        String commandName;
        String parameter;
        int num1, num2, num3;
```

```
        string val1, val2, val3;
        bool validCommand;
        bool isFill, isColor;



        ArrayList colors = new ArrayList() { "coral", "magenta", "chocolate", "lime",
"aqua" };
        Color pen;


        ArrayList errors = new ArrayList();
        ArrayList error_lines = new ArrayList();
        int error = 0;
        int count_line;

        bool noCommand;

        public int Error
        {
            get { return error; }
            set { error = value; }
        }

        public ArrayList ErrorLines
        {
            get { return error_lines; }
            set { error_lines = value; }
        }
        public bool NoCommand
        {
            get { return noCommand; }
            set { noCommand = value; }
        }

        public int Num1 { get; set; }
        public int Num2 { get; set; }

        public CommandParser() { }

        public void parseCommand(string command, Canvass canvas)
        {

            if (String.IsNullOrEmpty(command) == true) // checks if richTextBox is empty
            {
                noCommand = true;

            }

            else
            {
                char[] delimeter = new[] { '\r', '\n' };
                String[] lines = command.Split(delimeter,
StringSplitOptions.RemoveEmptyEntries); //splits line

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

                    count_line++; // counts line
                    String line = lines[i];
                    try
                    {
                        //char[] parantheses = new[] { '(', ')' };

                        //// check if each line has parantheses
```

```
                        //if (line.Contains(parantheses[0]) == false &&
line.Contains(parantheses[1]) == false)
                        //{

                        //    throw new CommandNotFoundException("Parentheses Not Found
");

                        //}
                        //else if (line.Contains(parantheses[0]) == false)
                        //{
                        //    throw new CommandNotFoundException(" \" ( \" Missing   ");

                        //}
                        //else if (line.Contains(parantheses[1]) == false)
                        //{
                        //    throw new CommandNotFoundException(" \" ) \" Missing  ");

                        //}
                        //else
                        //{
                        checkParentheses(line);

                        commandName = line.Split('(')[0].Trim().ToLower(); // line split
to get commandName

                        checkCommandName(commandName); // checkCommandName method called
passing parameter commandName
                        try
                        {
                            if (validCommand == true) // if command is valid
                            {
                                parameter = line.Split('(', ')')[1].ToLower(); // line
split to get parameter between ()

                                if (parameter.Length != 0) // if parameter exists
                                {

                                    checkParameter(parameter, commandName); // splits and
check parameter

                                    drawCommand(commandName, canvas); // draw commands to
canvas
                                }
                                else
                                {           // error display if parameter not found
                                    error++;
                                    error_lines.Add(count_line);
                                    errors.Add("Parameter not found ");

                                }

                            }
                            else
                            {           // if commandName is invalid
                                throw new IndexOutOfRangeException("Invalid command \"" +
commandName + " \"");
                            }
                        }

                        catch (IndexOutOfRangeException e) // handles
IndexOutOfRangeExcepetion
                        {
                            error++;     // counts number of errorLines
                            error_lines.Add(count_line); // add count_line to error_lines
arraylist
                            errors.Add(e.Message);  // add to arrayList errors
```

```
                    }
                    //}

                }
                catch (CommandNotFoundException e)
                {
                    error++;
                    error_lines.Add(count_line);
                    errors.Add(e.Message);
                }
            }

        }

    }
    public bool checkParentheses(string line)
    {
        char[] parantheses = new[] { '(', ')' };

        // check if each line has parantheses
        if (line.Contains(parantheses[0]) == false && line.Contains(parantheses[1])
== false)
        {
            return false;
            throw new CommandNotFoundException("Parentheses Not Found ");


        }
        else if (line.Contains(parantheses[0]) == false)
        {
            return false;
            throw new CommandNotFoundException(" \" ( \" Missing   ");


        }
        else if (line.Contains(parantheses[1]) == false)
        {
            return false;
            throw new CommandNotFoundException(" \" ) \" Missing  ");

        }
        else
        {
            return true;
        }


        }
        public bool checkCommandName(string commandName)
        {

            if (String.IsNullOrEmpty(commandName) == false)
            {

                // string array of commands
                string[] commands = { "drawto", "moveto", "circle", "rectangle",
"triangle", "pen", "fill" };
                for (int i = 0; i < commands.Length; i++)
                {
                    if (commands[i] == commandName)  // checks commandName
                    {
                        validCommand = true;      // command is valid
                    }
                }
                return validCommand;    // returns bool value
            }
```

```
                else
                {
                    return validCommand = false;    // command is invalid , returns bool
value false

                }
            }

            public bool drawCommand(string commandName, Canvass canvass)  // draw command
to canvas
            {
                if (String.IsNullOrEmpty(commandName) == false)
                {
                    switch (commandName)   // checks commandName and draw
                    {
                        case "drawto":
                            canvass.drawTo(num1, num2);   // calls drawTo method from
Canvass class
                            break;
                        case "circle":
                            canvass.drawCircle(num1);
                            break;
                        case "moveto":
                            canvass.moveTo(num1, num2);
                            break;
                        case "rectangle":
                            canvass.drawRectangle(num1, num2);
                            break;
                        case "triangle":
                            canvass.drawTriangle(num1, num2, num3);
                            break;
                        case "pen":
                            canvass.setPenColor(pen);
                            break;
                        case "fill":
                            canvass.fillShape(isFill);
                            break;

                    }
                    return true;
                }
                else
                {
                    return false;
                }
            }
            public bool checkParameter(string parameter, string commmandName) //checks
parameter pass with commands
            {
                try
                {
                    if (commandName.Equals("drawto") || commandName.Equals("moveto") ||
commandName.Equals("rectangle"))
                    {

                        if (parameter.Split('\u002C').Length == 2)  // splits parameter
at ,
                        {
                            val1 = parameter.Split('\u002C')[0].Trim(); //unicode for
comma
                            val2 = parameter.Split('\u002C')[1].Trim();

                            if (Regex.IsMatch(val1, @"^\d+$") && Regex.IsMatch(val2,
@"^\d+$")) // if both values are digits
                            {
                                num1 = int.Parse(val1);
```

```
                            num2 = int.Parse(val2);
                        }

                        else if (!Regex.IsMatch(val1, @"^\d+$") &&
Regex.IsMatch(val2, @"^\d+$"))
                        {
                            throw new InvalidParameterException("Wrong parameter \""
+ val1 + "\". Parameter should be integer ");
                        }
                        else if (!Regex.IsMatch(val2, @"^\d+$") &&
Regex.IsMatch(val1, @"^\d+$"))
                        {
                            throw new InvalidParameterException("Wrong parameter \""
+ val2 + "\". Parameter should be integer ");
                        }
                        else if (!Regex.IsMatch(val1, @"^\d+$") &&
!Regex.IsMatch(val2, @"^\d+$"))  // if val1 and val2 is not [0-9]
                        {
                            throw new InvalidParameterException("Wrong parameter \""
+ val1 + "\" and \"" + val2 + "\" Parameter should be integer ");
                        }



                    }
                    else
                    {
                        throw new InvalidParameterException("Should contain two
integer parameters ");  //throw error
                    }


                }
                if (commandName.Equals("triangle"))
                {

                    if (parameter.Split('\u002C').Length == 3)
                    {
                        val1 = parameter.Split('\u002C')[0].Trim(); //unicode for
comma
                        val2 = parameter.Split('\u002C')[1].Trim();
                        val3 = parameter.Split('\u002C')[2].Trim();

                        if (Regex.IsMatch(val1, @"^\d+$") && Regex.IsMatch(val2,
@"^\d+$") && Regex.IsMatch(val3, @"^\d+$"))
                        {
                            num1 = int.Parse(val1);
                            num2 = int.Parse(val2);
                            num3 = int.Parse(val3);
                        }

                        else if (!Regex.IsMatch(val1, @"^\d+$") &&
Regex.IsMatch(val2, @"^\d+$") && Regex.IsMatch(val3, @"^\d+$"))
                        {
                            throw new InvalidParameterException("Wrong parameter \""
+ val1 + "\". Parameter should be integer ");
                        }
                        else if (!Regex.IsMatch(val2, @"^\d+$") &&
Regex.IsMatch(val1, @"^\d+$") && Regex.IsMatch(val3, @"^\d+$"))
                        {
                            throw new InvalidParameterException("Wrong parameter \""
+ val2 + "\". Parameter should be integer ");
                        }
                        else if (!Regex.IsMatch(val3, @"^\d+$") &&
Regex.IsMatch(val2, @"^\d+$") && Regex.IsMatch(val1, @"^\d+$"))
                        {
```

```
                                throw new InvalidParameterException("Wrong parameter \""
+ val3 + "\". Parameter should be integer ");
                            }

                            else if (!Regex.IsMatch(val1, @"^\d+$") &&
!Regex.IsMatch(val2, @"^\d+$") && Regex.IsMatch(val3, @"^\d+$"))
                            {
                                throw new InvalidParameterException("Wrong parameter \""
+ val1 + "\" , \"" + val2 + "\" ");
                            }
                            else if (!Regex.IsMatch(val2, @"^\d+$") &&
!Regex.IsMatch(val1, @"^\d+$") && Regex.IsMatch(val3, @"^\d+$"))
                            {
                                throw new InvalidParameterException("Wrong parameter  \""
+ val2 + "\" and  " + val3 + "\" ");
                            }
                            else if (!Regex.IsMatch(val3, @"^\d+$") &&
Regex.IsMatch(val2, @"^\d+$") && !Regex.IsMatch(val1, @"^\d+$"))
                            {
                                throw new InvalidParameterException("Wrong parameter \""
+ val1 + "\" , \" " + val3 + "\" ");
                            }
                            else if (!Regex.IsMatch(val1, @"^\d+$") &&
!Regex.IsMatch(val2, @"^\d+$") && !Regex.IsMatch(val3, @"^\d+$"))
                            {
                                throw new InvalidParameterException("Wrong parameter \""
+ val1 + "\" , \"" + val2 + "\" and " + val3 + "\" ");
                            }
                        }
                        else
                        {
                            throw new InvalidParameterException("Should contain three
integer parameters ");
                        }


                    }
                    if (commandName.Equals("fill"))
                    {

                        if (parameter.Equals("on") || parameter.Equals("off"))
                        {
                            switch (parameter)
                            {
                                case "on":
                                    isFill = true;
                                    break;

                                case "off":
                                    isFill = false;
                                    break;
                            }

                        }
                        else
                        {
                            throw new InvalidParameterException("Fill should be either
\"on\" or \"off\" ");
                        }

                    }
                    if (commandName.Equals("pen"))
                    {
                        if (Regex.IsMatch(parameter, @"^\d+$"))
                        {
```

```
                            throw new InvalidParameterException("Parameter should not
contain integer");
                        }
                        else if (colors.Contains(parameter) == true)
                        {

                            checkColor(parameter);

                        }
                        else
                        {
                            throw new InvalidParameterException("Wrong color. \nColor
should be either \"coral\", \"magenta\", \"chocolate\", \"lime\", \"aqua\" ");
                        }

                    }
                    if (commandName.Equals("circle"))
                    {

                        val1 = parameter;
                        if (!Regex.IsMatch(val1, @"^\d+$"))
                        {
                            throw new InvalidParameterException("Wrong parameter for
circle \"" + val1 + "\" ");
                        }

                        else
                        {
                            num1 = int.Parse(val1);
                        }

                    }
                    return true;
                }
                catch (InvalidParameterException e)
                {

                    error++;
                    error_lines.Add(count_line);
                    errors.Add(e.Message);
                    return false;
                }

            }

            public bool checkColor(string select)
            {
                if (String.IsNullOrEmpty(select) == false)
                {
                    switch (select)
                    {
                        case "coral":
                            pen = Color.Coral;
                            break;
                        case "magenta":
                            pen = Color.Magenta;
                            break;
                        case "chocolate":
                            pen = Color.Chocolate;
                            break;
                        case "lime":
                            pen = Color.Lime;
                            break;
                        case "aqua":
                            pen = Color.Aqua;
                            break;
```

```
                    default:
                        pen = Color.Black;
                        break;

                }
                isColor = true;
                return true;
            }
            return false;
        }

        public ArrayList error_list()
        {
            return errors;
        }

    }


}
```

Shape Factory

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Component1
{
    internal class ShapeFactory
    {
        public Shape getShape(String shapeType)
        {
            shapeType = shapeType.ToLower().Trim();

            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 we get here then what has been passed in is inkown so throw an
appropriate exception
                System.ArgumentException argEx = new System.ArgumentException("Factory
error: " + shapeType + " does not exist");
                throw argEx;
            }
        }
    }
}
```

Rectangle

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Component1
{
    internal class Rectangle: Shape
    {
        int length, breadth;

        public Rectangle() : base()
        {

        }
        public Rectangle(Color color, bool fill, int x, int y, int length, int breadth) :
base(color, fill, x, y)
        {
            this.breadth = breadth;
            this.length = length;
        }

        public override void draw(Graphics g)
        {
            Pen pen = new Pen(color, 2);
            SolidBrush brush = new SolidBrush(color);

            if (fill)
            {
                g.FillRectangle(brush, x, y, length, breadth);
            }
            else
            {
                g.DrawRectangle(pen, x, y, length, breadth);
            }

        }
        public override void set(Color colour, bool fill, params int[] list)
        {
            //list[0] is x, list[1] is y, list[2] is length, list[3] is breadth
            base.set(colour, fill, list[0], list[1]);
            this.length = list[2];
            this.breadth = list[3];

        }
        public override string ToString() //all classes inherit from object and
ToString() is abstract in object
        {
            return base.ToString() + "  " + this.length + " "+ this.breadth;
        }
    }
}
```

Triangle

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Component1
{
    internal class Triangle : Shape
    {
        int sideA, sideB, sideC;
```

```
        public Triangle() : base()
        {

        }
        public Triangle(Color color, bool fill, int x, int y, int sideA, int sideB, int
sideC) : base(color, fill, x, y)
        {
            this.sideA = sideA;
            this.sideB = sideB;
            this.sideC = sideC;
        }
        public override void draw(Graphics g)
        {
            Pen pen = new Pen(color, 2);
            SolidBrush brush = new SolidBrush(color);

            if (fill)
            {

                Point[] point = { new Point(x, y), new Point(x+sideA, y),
                    new Point((int)(x + sideB* Math.Cos(sideC * Math.PI / 180))  ,
(int)(y +sideB * Math.Sin(sideC * Math.PI/180))) };
                g.FillPolygon(brush, point);
            }
            else
            {
                Point[] point = { new Point(x, y), new Point(x+sideA, y),
                    new Point((int)(x + sideB* Math.Cos(sideC * Math.PI / 180))  ,
(int)(y +sideB * Math.Sin(sideC * Math.PI/180))) };
                g.DrawPolygon(pen, point);

            }
        }
        public override void set(Color colour, bool fill, params int[] list)
        {
            //list[0] is x, list[1] is y, list[2] is length, list[3] is breadth
            base.set(colour, fill, list[0], list[1]);
            this.sideA = list[2];
            this.sideB = list[3];
            this.sideC = list[4];

        }


    }
}
```

Form

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Component1
{
    public partial class Form1 : Form
    {
        private Canvass myCanvass;
        Graphics g;

        public Form1()
        {
            InitializeComponent();
```

```
            g = drawPanel.CreateGraphics();
            myCanvass = new Canvass(g);
            xPos.Text = myCanvass.XPos.ToString();
            yPos.Text = myCanvass.YPos.ToString();
        }

        private void exitToolStripMenuItem_Click(object sender, EventArgs e)
        {

            if (MessageBox.Show("Do you want to Exit?", "Exit", MessageBoxButtons.YesNo,
MessageBoxIcon.Information) == DialogResult.Yes)
            {
                Application.Exit();
            }
        }

        private void action_KeyDown(object sender, KeyEventArgs e)
        {
            if (e.KeyCode == Keys.Enter)
            {
                String commands = actionText.Text.Trim().ToLower(); //read commandLine
trim whitespaces and change to lowercase
                if (commands.Equals("clear") == true)
                {
                    myCanvass.clear();
                    console.ForeColor = Color.Blue;
                    console.Text = "Canvass Cleared!";

                    actionText.Text = "";
                }
                else if (commands.Equals("run") == true)
                {
                    int i = 0;
                    console.Text = String.Empty;
                    CommandParser parse = new CommandParser();
                    //parse.Error.Clear();
                    parse.parseCommand(commandLine.Text, myCanvass);
                    if (parse.NoCommand == true)
                    {
                        console.ForeColor = Color.Red;
                        console.Text = "No commands to run";
                    }


                    if (parse.Error != 0)
                    {

                        console.ForeColor = Color.Red;



                        foreach (string error_description in parse.error_list())
                        {
                            console.AppendText(Environment.NewLine + "Error on line " +
(int)parse.ErrorLines[i] + ": " + error_description);
                            i++;
                        }

                        console.AppendText(Environment.NewLine + "Please correct command
syntax.");
                    }

                    actionText.Text = "";
                    xPos.Text = myCanvass.XPos.ToString();
                    yPos.Text = myCanvass.YPos.ToString();
                }
                else if (commands.Equals("reset") == true)
                {
                    myCanvass.reset();
                    console.ForeColor = Color.Green;
```

```
                    console.Text = "Program is reset to initial state \n Color is Set to
Black\n Position of pen is set to (0, 0) coordinates";
                    actionText.Text = "";
                    xPos.Text = myCanvass.XPos.ToString();
                    yPos.Text = myCanvass.YPos.ToString();

                }
                else
                {
                    console.ForeColor = Color.DarkBlue;
                    console.Text = "Invalid command!!. Command should be either \n 1. run
\n 2. clear \n 3. reset";
                    actionText.Text = "";
                }

            }
        }


        private void drawPanel_Paint(object sender, PaintEventArgs e)
        {
            g = e.Graphics;
        }

        private void loadToolStripMenuItem_Click(object sender, EventArgs e)
        {
            OpenFileDialog openFile = new OpenFileDialog
            {
                DefaultExt = "txt",
                Filter = "txt files (*.txt)|*.txt|All files (*.*)|*.*"
            };
            if (openFile.ShowDialog() == DialogResult.OK)
            {
                string selectedFile = openFile.FileName;
                commandLine.Text = File.ReadAllText(selectedFile, Encoding.UTF8);
            }
        }

        private void saveToolStripMenuItem_Click(object sender, EventArgs e)
        {

            SaveFileDialog saveFile = new SaveFileDialog();
            saveFile.Filter = "txt files (*.txt)|*.txt|All files (*.*)|*.*";
            saveFile.FilterIndex = 1;

            if (saveFile.ShowDialog() == DialogResult.OK)
            {
                File.WriteAllText(saveFile.FileName, commandLine.Text);
            }
        }


    }
}
```

CommandNotFoundException

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Component1
{
     class CommandNotFoundException: Exception
    {
        public CommandNotFoundException()
        {
        }
```

```
        public CommandNotFoundException(string message) : base(message)
        {
        }
    }
}
```

InvalidParameterException

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Component1
{
     class InvalidParameterException: Exception
    {

        public InvalidParameterException()
        {
        }

        public InvalidParameterException(string message) : base(message)
        {
        }



    }
}
```

Unit test

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System;
using Component1;
using System.IO;

namespace UnitTestProject1
{
    [TestClass]
    public class UnitTest1
    {
        CommandParser commandParser = new CommandParser();


        /// <summary>
        /// Checks if command name is valid
        /// </summary>
        [TestMethod]
        public void checkCommand()
        {

            string[] command = { "circle", "rectangle", "triangle", "drawto", "moveto",
"fill", "pen" };
            foreach (string arg in command)
            {
                Assert.IsTrue(commandParser.checkCommandName(arg));
            }

        }

        /// <summary>
        /// checks validity of color
        /// </summary>
        [TestMethod]
        public void checkColor()
        {
```

```
            string[] color = { "coral", "magenta", "chocolate", "lime", "aqua" };
            foreach (string arg in color)
            {
                Assert.IsTrue(commandParser.checkColor(arg));
            }

        }

        /// <summary>
        /// checks validity of parameters
        /// </summary>
        [TestMethod]
        public void checkParams()
        {
            string parameter = "3,4";
            string commandName = "drawto";
            int expectedNum1 = 3;
            int expectedNum2 = 4;
            try
            {
                commandParser.checkParameter(parameter, commandName);
                Assert.AreEqual(expectedNum1, commandParser.Num1);
                Assert.AreEqual(expectedNum2, commandParser.Num2);
            }
            catch (Exception)
            {
                Assert.AreNotEqual(expectedNum1, commandParser.Num1);
            }
        }

        /// <summary>
        /// checks if there are parentheses
        /// </summary>
        [TestMethod]
        public void checkBrackets()
        {
            string string1 = "circle 90";
            Assert.IsFalse(commandParser.checkParentheses(string1));
            string string2 = "circle(90)";
            Assert.IsTrue(commandParser.checkParentheses(string2));

        }


        [TestMethod]
        public void checkMoveTo()
        {
            int expectedNum1 = 2;
            int expectedNum2 = 3;

            Canvass can = new Canvass();
            can.moveTo(2,3);
            Assert.AreEqual(expectedNum1, can.XPos);
            Assert.AreEqual(expectedNum2, can.YPos);

            can.moveTo(5, 6);
            Assert.AreNotEqual(expectedNum1, can.XPos);
            Assert.AreNotEqual(expectedNum2, can.YPos);
        }


        [TestMethod]
        public void checkFillShape()
        {
            bool fill = true;
            Canvass can = new Canvass();
            can.fillShape(false);
            Assert.AreNotEqual(fill, can.Fill);
        }
    }
}
```