NEA Write-Up

Analysis

1. Have a project that the user can interact with.
    - Give the user multiple emails/letters that are either a large effect or a small effect of a random attribute. They can decide whether to increase the value for the given attribute or decrease it.
    - Give the user time to decide when they want to start looking at the emails/letters so if they want, they can continue to look at the town map.
2. Show the user what their decisions do to the town in real time.
    - Have a map of the town that both grows over time and changes as the user makes decisions on the town. If there is a higher value of one attribute present, the user can see more of that attribute's buildings show up on the map.
3. Make the program's objects interact with each other, over time.
    - When the user responds to an email/letter, the attribute that will be changed will also cause other linked attributes to undergo smaller changes. This will be done in specialized subs for each attribute.
4. Investigate which strategies allow the town to last longer.
    - Focus on certain attributes as I test and see how long the town survives for.
5. Make my code more presentable by reducing the number of strings that are visible in the code.
    - Have another project that encodes the email/letter text by RLE and creates a dictionary which will help decode the encoded text. The main project will take the dictionary and encoded text to decode it and add the lines into a new letter class for accessibility.
6. Have all the emails/letters, that the user must respond to show up immediately after the user states that they want to see them.
    - Use a Queue to line up the attributes needed so the choosing letters phase and the showing them to the user phase, are separated. This allows all letters to be picked and decoded before the user states that they want to answer them.

Design

There are two solutions that used in this project. One is the **Main Program**, which is what the user interfaces with, and the other is the **Compressor** that is used once to help convert long lines of text to compressed RLE whilst also making a dictionary that will help decode the text back to its original form. This was made so that I could fulfill requirement **6**.

Compressor:

- Add text into the Letter class.
    - Read LetterFile which is a text file stored in the program's files.
    - Save each line of text into the *Text* variable in an instance of the LetterLayout class.
- Create a dictionary.
    - Separate the text, in each LetterLayout instance, into words, punctuation and spaces.
    - Add new pairs into the dictionary is they do not exist in it already. Each pair consists of the number the word/char will be encoded to and the original plaintext value.
    - Save the completed dictionary into the Dictionary.txt file.
- Encode the text.
    - Separate the text, in each LetterLayout instance, into words, punctuation and spaces.
    - Encode each word/char using the dictionary made earlier.
    - Save the new line into the object's *Text* variable.

    - Save the encoded text into the CompressedLetterFile.txt file.
- Decode the RLE.
    - Numbers in the RLE are separated by space so read up to a space.
    - Use the dictionary to convert the numbers, before the space, into text.
    - Save the new text line into the object's *Text* variable.

The last step (Decode the RLE) is copied and pasted into the Main Program as it is not needed in this program. The text within the new, updated, Dictionary.txt and CompressedLetterFile.txt is copied and pasted into files of the same name into the Main Program, so it can used there.

Main Program:

- Start the program and open the Main Menu.
    - Close the program if the user clicks on the 'Exit' button.
    - Open an instance of LeaderSandbox – the main game – and hide the menu.
    - Hide itself and start a ReShowMenutmr, that checks whether the game has ended or not so that the menu can be reshown.
- Start a new game.
    - Load Game Structures
        - Instantiate Letters from the Letterlayout class using the txt file.
        - Instantiate attributes and values from the AttributeBase class using the txt file.
        - Create a dictionary with values form the txt file.
    - Load Game Features
        - Load all pictureBoxes into a large collection – making sure they are in order by looking at their names.
        - Load all labels that should display the attributes' status into a collection.
        - Read the lines in the TownLayout.txt file and use the Substring function to save the values into the TownMap 2D array.
            - Use the TownMap array so that the pictureboxes, on the form, show the user what the town looks like in the TownMap array.
        - Hide all buttons and labels that should not be visible at the start.
- Start Day (Part 1 of the loop).
    - Fill the Letter Queue
        - Find the first attribute to add to the queue.
            - Look through the attributes choose the attribute with the lowest value to the queue.
            - Reset the ValueChange values for each attribute as it is a new day.
        - Find the second attribute to add to the queue.
            - Pass in the first attribute chosen and then look through all the attributes to find the second lowest attribute to add to the queue.
        - Find the third attribute to add to the queue.
            - Use the attributes in the queue already, to find a random attribute to add.
        - Add the letter markers to the queue.
            - Loop so that every attribute name is replace with a letter marker. Loop 3 times.
                - Check the chosen attribute's name against the attributes collection for a name match and then collect the marker that signifies which 10 letters are for which attribute.
                - Go through the 10 letters that are available for each attribute. If one is unread then add it to the queue.
    - Update the labels so that the value changes are no longer visible.

- Wait for user response to press the 'See Emails' button to start part 2 of the day loop.
    - Loop for each marker in the queue.
        - Use the dictionary to decode the text within the letter object.
        - Save the text into the public Lettertext variable.
        - Create an instance of the next form, GenericEmailApp, where the user will respond to the letter.
            - Output the text, from the LetterText variable, for the user to see.
            - Take in the input given by the user, save that, the letter marker, and the input status in public variable for the previous form to access.
        - Start the CheckResponsetmr for responses from the user.
    - The user decides a response for the letter and causes the InputGiven variable to become positive, this is seen by the CheckResponsetmr.
        - The response and letter marker are recorded into local variables, and the InputGiven variable is changed back to false.
        - Increment a variable that shows how many letters have been answered.
        - Produce the letter effects on the attributes.
            - Find the attribute the letter belonged to by doing marker % 10.
            - Find whether the letter had a large effect or not (even = high, odd = low)
            - Use these values to find and alter the attributes affected by the letter.
                - Each attribute has a specialised function that is called when their values change. It also affects a select number of attributes linked to it.
                - The Percentage attributes act differently as the rest of them must be changed evenly to accommodate the change in the original percentage attribute.
        - Update the labels to show the new values and any ValueChange values.
        - Add or Remove buildings
            - Do for every percentage attribute.
                - Look at the attribute's value and use that with the MapArea to find out how many, of each building, should be shown on the map.
                - Count how many of the attribute's buildings are already on the map and use it with the number needed to find out if that number needs to be altered.
                - Relative to the BuildingDifference, add or remove buildings randomly by using LandLength to determine what parts of the map are available to the user.
                - Load the town again to show the changes done to the town map.
        - Check the variable that shows how many letters have been answered for the day. If less than 3, loop. If it is 3 then reset the variable and show the 'Next Day' button.
- Wait for the user to move onto the next day by pressing the button.
    - Use the attribute's Advance functions for each one to represent natural changes over time.
    - Increment the dayspassed variable.
    - Increase the LandLength variable every two 'days' as long as it is below 10.
    - Check for the End of the game
        - Check if dayspassed has become greater than 15.
        - Check if population is less than 500.
        - Check if air-cleanliness has gone below 0.
- Act depending on whether the game has ended or not.
    - If game has not ended, then go back to Start Day (Part 1)
    - If game has ended, then show the ending label and button.
- Wait for the user to click on the 'Return to main menu' button.
    - Hide the form and set the public EndGame variable as true.

- Bool change is seen by the ReShowMenutmr and the Main Menu is reshown.

Implementation

Compressor:

```
using System;
using System.Collections.Generic;
using System.IO;

namespace Creating_a_Dictionary_for_Leader
{

    class Program
    {
        //creating a collection to hold the instances of LetterLayout class
        static LetterLayout[] Letters = new LetterLayout[80];
        //create a new dictionary int is for the key
        //and string is for the word/char
        private static Dictionary<int, string> letterDictionary;

        static void Main(string[] args)
        {
            //read the LetterFile into the LetterLayout objects
            ReadLetterFile();

            //instantiate a new dictionary
            letterDictionary = new Dictionary<int, string>();
            //add pairs to the dictionary
            CreateDictionary();
            //save the dictionary into the Dictionray.txt file
            WriteDictionary();

            //Encode the text in the LetterLayout class objects
            EncodeText();
            //save the decoded text into the
            //CompressedLetterFile.txt file
            WriteText();

            //decode the RLE back to the original text
            DecodeText();
       }

        public static void ReadLetterFile()
        {
            //use a streamreader to make every line
            int Lettercounter = 0;
            StreamReader sr = new StreamReader(@"TextFiles\\LetterFile.txt");
            string line = "";

            while ((line = sr.ReadLine()) != null)
            {
                //instantiate new LetterLayout objects, use lettercounter as an id.
                //save the line as the object's text.
                Letters[Lettercounter] = new LetterLayout(Lettercounter, line);
                //increment the lettercounter for the next object.
                Lettercounter++;
            }
        }
```

```
public static void CreateDictionary()
{
    //go through all the text within the letter objects, separate the words
    //check whether they are in dictionary or not
    //if so then increment
    //if not then add a new word to the dictionary

    //this takes note of how many words are in the dictionary
    int wordcounter = 0;
    //go through all the LetterLayout objects in the collecction
    foreach (LetterLayout thisletter in Letters)
    {   //take the text stored in the object
        string line = thisletter.GetText();
        int charcounter = 0;
        int wordstart = 0;
        int wordlength = 0;
        string word = "";
        //until it gets to the end of the line
        while (charcounter <= line.Length - 1)
        {   //go through the line by looking at each char
            char character = Convert.ToChar(line.Substring(charcounter, 1));
            //if its another char then:
            if (Char.IsLetter(character))
            {   //increase the length of the word
                wordlength++;
                //check for the end of the line
                if (charcounter == line.Length - 1)
                {
                    //the end of the line is reached, word is sent off:
                    //the whole word is picked up
                    word = line.Substring(wordstart, wordlength);
                    //the wordlength returns to 0 for the next word
                    wordlength = 0;
                    //the word is then added to the dictionary.
                    wordcounter = AddToDictionary(word, wordcounter);
                }
            }
            else if (Char.IsPunctuation(character))
            {   //character may be punctuation
                //so, send off the word before it.
                word = line.Substring(wordstart, wordlength);
                wordcounter = AddToDictionary(word, wordcounter);
                //the punctuation is also added
                wordcounter = AddToDictionary(Convert.ToString(character), wordcounter);
                wordlength = 0;
            }
            else
            {
                //character may be a space
                word = line.Substring(wordstart, wordlength);
                if (!Char.IsPunctuation(Convert.ToChar(line.Substring(charcounter - 1, 1))))
                {   //if the last character wasn't punctuation,
                    //then add the word before the space
                    wordcounter = AddToDictionary(word, wordcounter);
                }
                //add the space to the dictionary.
                wordcounter = AddToDictionary(Convert.ToString(character), wordcounter);
                wordlength = 0;
                //wordstart is used with wordlength to show where
                //a word starts so that it can added to the dictionary.
                //move the wordstart up for the next word.
                wordstart = charcounter + 1;
```

```
            }
            //move up to the next char
            charcounter++;
        }
    }
}

public static int AddToDictionary(string value, int counter)
{
    //if the value isn't in the dictionary, add it using the word counter to choose the key
    if (!letterDictionary.ContainsValue(value))
    {   //add a new pair
        letterDictionary.Add(counter, value);
        //counter increments to show the next empty space for a new word
        counter++;
    }
    return counter;

}

public static void EncodeText()
{
    //The text in each letter object will be edited and changed by looking at the dictionary
    //changes will be held in newline until the end of the loop.
    //code below is almost identical to the code in ReadLetterFile().
    foreach (LetterLayout thisletter in Letters)
    {   //each line is taken from the object
        string line = thisletter.GetText();
        //this will hold all the compressed text until it is to save into the object
        string newline = "";

        int charcounter = 0;
        int wordstart = 0;
        int wordlength = 0;
        string word = "";

        while (charcounter <= line.Length - 1)
        {
            char character = Convert.ToChar(line.Substring(charcounter, 1));

            if (Char.IsLetter(character))
            {
                wordlength++;
                if (charcounter == line.Length - 1)
                {
                    //the end of the line is reached, word is sent off
                    word = line.Substring(wordstart, wordlength);
                    wordlength = 0;
                    //word is compressed and added to the newline.
                    newline = CompressText(word, newline);
                }
            }
            else if (Char.IsPunctuation(character))
            {   //character may be punctuation
                //so the word before it is added to the dictionary
                word = line.Substring(wordstart, wordlength);
                newline = CompressText(word, newline);
                //the character is also compressed
                newline = CompressText(Convert.ToString(character), newline);
                wordlength = 0;
            }
            else
```

```
                {
                    //character may be a space
                    word = line.Substring(wordstart, wordlength);
                    if (!Char.IsPunctuation(Convert.ToChar(line.Substring(charcounter - 1, 1))))
                    {   //if the last character wasnt punctuation the add the word before the space
                        newline = CompressText(word, newline);
                    }
                    //compress the space as well
                    newline = CompressText(Convert.ToString(character), newline);
                    wordlength = 0;
                    wordstart = charcounter + 1;
                }

                charcounter++;

            }
            //set th enew line into the object's text variable.
            thisletter.SetText(newline);
        }
    }

    public static string CompressText(string word, string newline)
    {
        //look for corresponding word in the dictionary
        //add this value to the variable newline
        foreach (KeyValuePair<int, string> item in letterDictionary)
        {
            if (item.Value == word)
            {   //separate the keys by spaces to make decoding easier.
                newline = newline + item.Key + " "; ;
            }
        }
        //return the changed line
        return newline;
    }

    public static void DecodeText()
    {
        //go through the text until a space is reached, convert using dictionary
        //and then save to the letter object
        //go through every object in the collection
        foreach (LetterLayout thisletter in Letters)
        {   //save the object's text into a string variable.
            string line = thisletter.GetText();
            int counter = 0;
            string key = "";
            //the var that will hold the decoded words
            string text = "";

            //go through the entire line
            while (counter <= line.Length - 1)
            {   //if a char isn't a space then:
                if (!Char.IsNumber(Convert.ToChar(line.Substring(counter, 1))))
                {   //convert the key into a word
                    //and save the result into the text variable
                    text = text + DecompressText(Convert.ToInt32(key));

                    key = "";
                }
                key = key + line.Substring(counter, 1);
                //move on through the line
                counter++;
```

```
                }
                //save the final text into the text variable in each LetterLayout object
                thisletter.SetText(text);
            }

        }

        public static string DecompressText(int key)
        {
            string output = "";
            //find the value the key corresponds to and return the value
            foreach (KeyValuePair<int, string> item in letterDictionary)
            {
                if (item.Key == key)
                {
                    output = item.Value;
                }
            }
            //return the decoded word
            return output;
        }

        public static void WriteText()
        {   //use a streamwriter to write the RLE into the file
            StreamWriter sw = new StreamWriter(@"TextFiles\\CompressedLetterFile.txt");
            foreach (LetterLayout thisletter in Letters)
            {   //go through every LetterLayout object's text
                sw.WriteLine(thisletter.GetText());
            }
            sw.Close();
        }

        public static void WriteDictionary()
        {   //create a streamreader to add to the file
            StreamWriter sw = new StreamWriter(@"TextFiles\\Dictionary.txt");
            foreach (KeyValuePair<int, string> item in letterDictionary)
            {   //every pair in the dictionary is added to the file.
                sw.WriteLine(item.Key);
                sw.WriteLine(item.Value);
            }
            sw.Close();
        }

        public class LetterLayout
        {
            private int ID;
            private string Text;
            //every time a new letter is made, the id and text will be recorded
            //the letter will always be unread at first.
            public LetterLayout(int id, string text)
            {
                this.ID = id;
                this.Text = text;
            }
            //setter and getter functions
            public void SetText(string text) { this.Text = text; }
            public int GetID() { return ID; }
            public string GetText() { return Text; }
        }
    }
}
```

This is the dictionary made and an extract of the CompressedLetterFile made as the program is run.

This is an extract of the text in the LetterFile file.

```
LargeEffect - Population
SmallEffect - Population
LargeEffect - Population
SmallEffect - Population
LargeEffect - Population
SmallEffect - Population
LargeEffect - Population
SmallEffect - Population
LargeEffect - Population
SmallEffect - Population
LargeEffect - Road Network
SmallEffect - Road Network
LargeEffect - Road Network
SmallEffect - Road Network
LargeEffect - Road Network
SmallEffect - Road Network
LargeEffect - Road Network
SmallEffect - Road Network
LargeEffect - Road Network
SmallEffect - Road Network
LargeEffect - Air Cleanliness
SmallEffect - Air Cleanliness
LargeEffect - Air Cleanliness
```

Dictionary - Notepad

File Edit Format View

```
0
LargeEffect
1

2

3
-
4
Population
5
SmallEffect
6
Road
7
Network
8
Air
9
Cleanliness
10
Recreation
11
Housing
12
Jobs
13
Education
14
Wildlife
```

CompressedLetterFile - Notepad

File Edit Format View Help

```
0 1 2 3 1 4
5 1 2 3 1 4
0 1 2 3 1 4
5 1 2 3 1 4
0 1 2 3 1 4
5 1 2 3 1 4
0 1 2 3 1 4
5 1 2 3 1 4
0 1 2 3 1 4
5 1 2 3 1 4
0 1 2 3 1 6 1 7
5 1 2 3 1 6 1 7
0 1 2 3 1 6 1 7
5 1 2 3 1 6 1 7
0 1 2 3 1 6 1 7
5 1 2 3 1 6 1 7
0 1 2 3 1 6 1 7
5 1 2 3 1 6 1 7
0 1 2 3 1 6 1 7
5 1 2 3 1 6 1 7
0 1 2 3 1 8 1 9
5 1 2 3 1 8 1 9
0 1 2 3 1 8 1 9
5 1 2 3 1 8 1 9
0 1 2 3 1 8 1 9
```

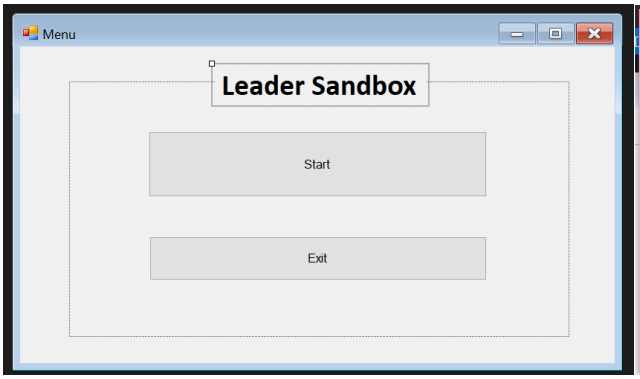
Main Program:

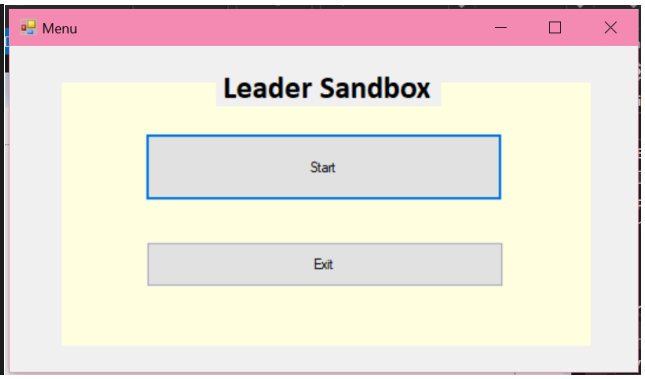
There are 3 parts/ forms to the main program: the main menu, the game UI, and the letter display.

Main Menu (Form 1):

```
using System;
using System.Drawing;
using System.Windows.Forms;

namespace Trying_To_Put_Leader_Together
{
    public partial class MainMenu : Form
    {
        public MainMenu()
        {
            InitializeComponent();
        }
        private void Menu_Load(object sender, EventArgs e)
        {   //title image
            pictureBox1.BackColor = Color.LightYellow;
        }
        private void button2_Click_1(object sender, EventArgs e)
        {   //closes the program
            Close();
        }
        private void button1_Click_1(object sender, EventArgs e)
        {
            //show all file buttons
            //starts game
            Leader_Sandbox game = new Leader_Sandbox();
            game.Show();
            //hides the menu
```

```
            Hide();
        }
        private void ReShowMenutmr_Tick(object sender, EventArgs e)
        {   //always on to check the status of the game
            if (Leader_Sandbox.EndGame == true)
            { //reshow the form so the user can choose to play again or end the game
                Show();
            }
        }
    }
}
```





Whilst running ←

Game (Form 2):

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Drawing;
using System.IO;
using System.Linq;
using System.Windows.Forms;


namespace Trying_To_Put_Leader_Together
{
    public partial class Leader_Sandbox : Form
    {
        public Leader_Sandbox()
        {
            InitializeComponent();
        }

        ///many of the subs require the below collections and variables to work
        ///However, these subs depend on user inputs and so cannot be passed as
        ///parameters between subs. Therefore they must be gloabl.

        //a group for all the attributes
        static AttributeBase[] Attributes = new AttributeBase[9];
        //a group for all the letters
        static LetterLayout[] Letters = new LetterLayout[80];
        //dictionary for decompressing CompressedLetterFile.txt
        static Dictionary<int, string> letterDictionary;
        //a collection to hold all the pictureboxes
        static PictureBox[] Pictureboxes = new PictureBox[100];
        //used by buttons so it must be static
        static Queue LetterQueue = new Queue(3);
        //a collection to hold all the labels
```

```
        static Label[] Labels = new Label[16];
        //a 2D array to hold the status of the map as the game runs
        char[,] TownMap = new char[10, 10];


        //global variables that are shared with the third form
        //when emails are being answered.
        public static string LetterText;
        public static int LetterElement;
        public static bool InputGiven = false;

        //variables used by timers in this cs file so they must be global so they can access them.

        //holds how many letters has been responded
        //to so that the game can progress to the next day
        static int LettersRespondedTo = 0;
        //also shared with the menu - checks whether game has ended or not.
        public static bool EndGame = false;

        //variables that are incremented over time so can only be created and given a value once

        //holds how many days have passed
        static int dayspassed = 1;
        //holds how much of the map will be available to the user.
        static int LandLength = 4;

        private void Leader_Sandbox_Load(object sender, EventArgs e)
        {
            //load up everything
            //first load the non-visible parts and then the form's (visible) parts.
            letterDictionary = new Dictionary<int, string>();
            ConfigureGame();
            ConfigureForm();
            //start the game
            StartDay();
        }
public static int Random(int largevalue)
        {   //returns a random number from a specified range
            Random random = new Random();
            int num = random.Next(0, largevalue);
            return num;
        }

        public void ConfigureGame()
        {   //this sub will load all value stored in the text files into the dictionary, classes etc.
            //Reads and creates instances of attributes from TownBaseVariables.txt
            LoadBase();
            //creates a dictionary with data from the Dictionary.txt
            LoadDictionary();
            //creates instances of LetterLayout with data from CompressedLetterFile.txt
            LoadLetterFile();
        }

        public void ConfigureForm()
        {   //place all the pictureboxes into an ordered collection
            LoadBoxes();
            //place all the labels into a collection
            LoadLabels();
            //fill the 2D array with the data in TownLayout.txt
            FillTownArray();

            //hide controls that should not be visible to the user right now
```

```
            endlabel.Hide();
            menubutton.Hide();
            Daybt.Hide();
        }

        public void LoadBase()
        {

            //go through the text file and call the setter variables every new line
            StreamReader sr = new StreamReader(@"TextFiles\\TownBaseVariables.txt");
            string line;

            //configures every attribute and adds it to the Attributes group.
            for (int i = 0; i < Attributes.Length; i++)
            {
                Attributes[i] = new AttributeBase();
                //every 4 lines in the file belong to one attribute
                line = sr.ReadLine();
                Attributes[i].SetName(line);
                line = sr.ReadLine();
                Attributes[i].SetValue(Convert.ToInt32(line));
                Attributes[i].SetStartValue(Convert.ToInt32(line));
                line = sr.ReadLine();
                Attributes[i].SetMultiplier(Convert.ToSingle(line));
                line = sr.ReadLine();
                Attributes[i].SetType(Convert.ToChar(line));
                Attributes[i].SetMarker(i);

            }
        }

        public void LoadDictionary()
        {
            StreamReader sr = new StreamReader(@"TextFiles\\Dictionary.txt");
            string line;
            //every two lines in the file make up a dictionary entry
            //if there is a key, there wil be a value after it.
            while ((line = sr.ReadLine()) != null)
            {
                int key = Convert.ToInt32(line);
                string value = sr.ReadLine();
                //key and value are found so add to the dictionary
                letterDictionary.Add(key, value);
            }
        }

        public void LoadLetterFile()
        {   //use a streamreader to read from the txt file
            int Lettercounter = 0;
            StreamReader sr = new StreamReader(@"TextFiles\\CompressedLetterFile.txt");
            string line = "";

            while ((line = sr.ReadLine()) != null)
            {   //a new instance of LetterLayout is made and holds the encoded text.
                Letters[Lettercounter] = new LetterLayout(Lettercounter, line);
                Lettercounter++;
            }
        }

        public void LoadLabels()
        {   //go through all the labels on the form
            int counter = 0;
```

```
            foreach (Label thislabel in this.Controls.OfType<Label>())
            {   //ignore the two labels that do not relate to attributes
                if (thislabel.Name != "endlabel" && thislabel.Name != "daylabel")
                {
                    Labels[counter] = thislabel;
                    counter++;
                }
            }
        }

        public void UpdateLabels()
        {
            for (int i = 0; i < Labels.Length; i++)
            {
                //first 8 labels are for the attributes and rest are for showing changes in the
attribute's values.
                if (i < 8)
                {
                    Labels[i].Text = " " + Attributes[i].GetName() + " = " + Attributes[i].GetValue();
                }
                else
                {
                    int change = Attributes[i - 8].GetValueChange();
                    if (change != 0)
                    {
                        //changing the colours of the 'changed' labels. Green is given when the value
increased,
                        //red is given when it has decreased.
                        if (change > 0) { Labels[i].ForeColor = System.Drawing.Color.Green; }
                        else { Labels[i].ForeColor = System.Drawing.Color.Red; }
                        Labels[i].Text = Convert.ToString(change);
                    }
                    else
                    {
                        Labels[i].Text = "";
                    }
                }

            }
        }

        public void LoadBoxes()
        {
            //goes through each of the 100 pictureboxes and adds then to the Pictureboxes collection
            foreach (PictureBox control in this.Controls.OfType<PictureBox>())
            {
                if (control.Name.StartsWith("pictureBox"))
                {
                    //converting 001 to int would save as 1
                    //so take the last 3 digits in the boxes' names and use it
                    //to add the box to the collection (in the correct order)
                    int number = Convert.ToInt32(control.Name.Substring(10, 3));
                    Pictureboxes[number - 1] = control;

                }
            }
        }

public void FillTownArray()
        {   //use a streamreader to go through the file
            StreamReader sr = new StreamReader(@"TextFiles\\TownLayout.txt");
            string line;
```

```
            for (int i = 0; i < 10; i++)
            {   //the txt has 10x10 chars
                //so each one is mapped to its place in the array
                line = sr.ReadLine();
                for (int j = 0; j < 10; j++)
                {
                    TownMap[i, j] = Convert.ToChar(line.Substring(j, 1));
                }
            }
            //use the new, updated array to add images to the form
            LoadTown();

        }

        public void LoadTown()
        {
            int basedigit = 0;
            //use the already made map array to set the picture boxes' images
            for (int i = 0; i < 10; i++) //go down the column
            {
                for (int j = 0; j < 10; j++) //go across the row.
                {   //set every picturebox to Zoom so that images don't look off
                    Pictureboxes[basedigit + j].SizeMode = PictureBoxSizeMode.Zoom;
                    char Char = TownMap[i, j];
                    switch (Char)
                    {   //use a switch-case to set the image
                        case 'H'://housing
                            Pictureboxes[basedigit + j].Image =
Image.FromFile(@"ImageFiles\\house.png");
                            break;

                        case 'E'://education
                            Pictureboxes[basedigit + j].Image =
Image.FromFile(@"ImageFiles\\education.png");
                            break;

                        case 'J'://jobs
                            Pictureboxes[basedigit + j].Image =
Image.FromFile(@"ImageFiles\\workplace.png");
                            break;

                        case 'R'://recreation
                            Pictureboxes[basedigit + j].Image =
Image.FromFile(@"ImageFiles\\recreation.png");
                            break;

                        case 'W'://wildlife
                            Pictureboxes[basedigit + j].Image =
Image.FromFile(@"ImageFiles\\wildlife.png");
                            break;
                        default://empty space
                            Pictureboxes[basedigit + j].BackColor = System.Drawing.Color.DarkGreen;
                            break;
                    }
                }   //used to set which picturebox is being changed
                basedigit = basedigit + 10;
            }
        }

        public void StartDay()
        {   //clean up controls and timers from the last day
```

```
        CheckResponsetmr.Stop();
        InputGiven = false;
        Daybt.Hide();
        //shows the user what day it is
        daylabel.Text = "Day: " + Convert.ToString(dayspassed);

        //fills the queue
        FillLetterQueue();
        //updates the labels with the current attribute values
        UpdateLabels();
        //gives the user the option to continue
        emailbt.Show();
        //the user then decides when to answer the emails
        //game continues when the button is pressed.
    }
    //the program then waits for the user to decide whether they want to keep looking
    //at the town of it they want to answer emails instead
    private void emailbt_Click(object sender, EventArgs e)
    {
        //by clicking on the emails button, it starts the 2nd part of the day cycle
        emailbt.Hide();
        //the user can now answer the letters chosen previously
        GetLetterResponse();
    }
    //this will start a new day and increment the dayspassed var.
    private void Daybt_Click(object sender, EventArgs e)
    {   //calls the sub that will cause time changes
        bool Ended = AdvanceTime();
        //if the game hasn't met the end critria let it continue
        if (!Ended) { StartDay(); }
    }

    public bool AdvanceTime()
    {
        //the attributes will be affected by their multipliers
        foreach (AttributeBase thisAttribute in Attributes)
        {   //the advance sub will be called for each attribute
            thisAttribute.Advance();
        }//increment day counter
        dayspassed++;
        if (dayspassed % 2 == 0 && LandLength <10)
        {   //increase lang length every 2 days
            LandLength++;

        }//check for an end and return the result
        bool Ended = CheckEnd();
        return Ended;
    }

    public bool CheckEnd()
    {
        //you can end the game if dayspassed >20,
        //or if population goes below 500 / if air cleanliness <0
        bool Ended = false;
        if (dayspassed > 20)
        {   //game ends
            Ended = true;
            ShowEnd();
        }
        foreach (AttributeBase attribute in Attributes)
        {
```

```
                if (attribute.GetName() == "Population")
                {   //if less than 500 people are in the town
                    if (attribute.GetValue() < 500)
                    {   //game ends
                        Ended = true;
                        ShowEnd();
                    }
                }
                if (attribute.GetName() == "Air Cleanliness")
                {   //if air cleanliness gets below 0
                    if (attribute.GetValue() <= 0)
                    {   //game ends
                        Ended = true;
                        ShowEnd();
                    }
                }
            }
            return Ended;
        }

        public void ShowEnd()
        {
            //hides and shows the relevant controls
            //including the ending message
            menubutton.Show();
            endlabel.Show();
            //other two buttons are hidden to prevent the user from continuing
            Daybt.Hide();
            emailbt.Hide();
        }

  private void menubutton_Click(object sender, EventArgs e)
        {
            //new form links back to main menu; it will recognise the bool change
            Hide();
            EndGame = true;
        }

        public void FillLetterQueue()
        {
            //first fill the queue with the names of the attributes of which's letters will be used
            //the first two will be dependent on FindLowest whilst the last one will be random.

            //there is no first attribute at first so pass in a null value
            LetterQueue.Enqueue(FindLowestAtt(""));
            //the second value must not be the same as the first so use peek to pass the first's name
in.
            LetterQueue.Enqueue(FindLowestAtt(Convert.ToString(LetterQueue.Peek())));
            //third one is random but must also not be equal to the other two.
            LetterQueue.Enqueue(FindRandomAtt());

            //then use the names as parameters for ChooseLetter and hold the letter IDs in the queue.
            for (int i = 0; i < 3; i++)
            {   //every attribute is removed and then a letter is added in its place
                //this is done 3 times to return the queue back to its original order
                LetterQueue.Enqueue(ChooseLetter(Convert.ToString(LetterQueue.Dequeue())));
            }
        }

        public string FindLowestAtt(string FirstAttribute)
        {   //smallest ratio is used to find the lowest attribute
            float smallestratio = 1;
```

```
            //holds the name of the smallest attribute
            string smallestname = "";

            foreach (AttributeBase thisobject in Attributes)
            {
                //every 'day' the valuechange var in the class is reset. This is the first
                //place all attributes are accessed so this is put here.
                thisobject.SetValueChange(0);
                //if the object hasn't been completely read and isnt equal to the first attribute or a
free space attribute
                if (!thisobject.GetAllRead() && thisobject.GetName() != FirstAttribute &&
thisobject.GetName() != "Free Space")
                {   //Make the current value a percentage of the original value
                    float newratio = thisobject.GetValue() / thisobject.GetStartValue();
                    if (newratio < smallestratio)   //each attribute is looked at so the samllest % is
chosen
                    {   //smallest ratio and smallest name is set
                        smallestratio = newratio;
                        smallestname = thisobject.GetName();
                    }
                }

            }
            //at the begining all the ratios will equal 1 so there won't be a smaller attribute
            //so choose a random attribute:
            if (smallestname == "")
            {
                bool found = false;
                do
                {   //choose a random attribute
                    int element = Random(8);
                    //as long as it isn't all read or equal to the first attribute, you can use it
                    //don't check if its 'Free space' as it is the 9th attribute and the random
function was only given 8.
                    if (Attributes[element].GetAllRead() == false && Attributes[element].GetName() !=
FirstAttribute)
                    {
                        smallestname = Attributes[element].GetName();
                        found = true;
                    }

                } while (!found);
            }
            //return the name found
            return smallestname;
        }

        public string FindRandomAtt()
        { //choose a random attribute, use the queue to make sure attributes haven't been repeated.
            bool found = false;
            string first = Convert.ToString(LetterQueue.Dequeue());
            string second = Convert.ToString(LetterQueue.Dequeue());
            int element;
            do
            {
                element = Random(8);
                //check if all the values have been read or not
                if (Attributes[element].GetAllRead() == false && Attributes[element].GetName() != "Free
Space")
                {
                    //check if it is already in the queue
```

```
                if (Attributes[element].GetName() != first && Attributes[element].GetName() !=
second)
                {
                    found = true;
                }
            }
        } while (!found);

        //add back to the queue
        LetterQueue.Enqueue(first);
        LetterQueue.Enqueue(second);
        //retrun the found attribute
        return Attributes[element].GetName();

    }

    public int ChooseLetter(string attributename)
    {
        bool found = false;
        int counter = 0;
        int marker = 0;
        //first find the letter range for the attribute
        while (!found)
        {
            if (Attributes[counter].GetName() == attributename)
            {   //for example, road network would have a marker of 1
                //1 * 10 = 10 so the letters for road network start at 10
                marker = (Attributes[counter].GetMarker() * 10);
                found = true;
            }
            else { counter++; }
        }

        //now choose a letter from the range - if it hasn't been read before
        //as you go down the letters, marker will increment
        while (Letters[marker].GetRead() && marker < 80)
        {
            marker++;
        }
        if (marker % 10 == 9)//if it reaches the end of the range, set the attribute as AllRead
        {
            Attributes[counter].MakeAllRead();
        }
        //what if all the letters for that type has already been read?
        //then the check in FindLowest handles that
        return marker;
    }

    public void GetLetterResponse()
    {
        ///this will take a letter and present it in the third form as an email.
        ///there the user will give a response to the question, form 3 will close
        ///the changes that will occur are shown in form 2 (here)
        for (int i = 0; i < 3; i++)
        {   //get the letter marker/element from the queue
            LetterElement = Convert.ToInt32(LetterQueue.Dequeue());
            //decode it and make read so that it can be put in public variables
            Letters[LetterElement].DecodeText(letterDictionary);
            Letters[LetterElement].MakeRead();
            LetterText = Letters[LetterElement].GetText();
            //open the third form
            Generic_Email_App email = new Generic_Email_App();
```

```
                email.Show();
                //the third form can see the public variables and use the data there
                //in its own form.
                //this timer is used to check whether the user has awnsered an email or not
                CheckResponsetmr.Start();
            }

        }

        //a timer that waits for a user's input
        private void CheckResponsetmr_Tick(object sender, EventArgs e)
        {
            if (InputGiven)
            {
                //use the global variable from the third form
                bool response = Generic_Email_App.Response;
                //repoonse bool is used to tell the next function what the user chose
                //it carries the value that is held in Generic_Email_App
                //make input given false for the next emails
                InputGiven = false;
                LettersRespondedTo++;

                //causes change in the attributes
                ProduceLetterEffects(LetterElement, response);
                //shows the changes to attributes
                UpdateLabels();
                //changes and shows the changes in the town map
                EditBuildings();

                //once all the letters have been responded to, the next day button is shown.
                if (LettersRespondedTo == 3)
                {   //set back to 0 for the next day
                    LettersRespondedTo = 0;
                    //show the button that lets the user move on to the next day.
                    Daybt.Show();
                }
            }

        }
//Produce letter effects
        public void ProduceLetterEffects(int element, bool response)
        {
            bool LargeEffect = false;
            //look at the letter's ID to find which attribute it belongs to
            //as well as by how much the response will affect it
            //every attribute has 10 letters
            //even ones will have a large effect and odd ones will have a small effect

            //find its effect:
            if (element % 2 == 0) { LargeEffect = true; }
            //find which attribute it belongs to:
            int group = element / 10;   //this should truncate the units
            switch (group)
            {
                case 0:
                    //population change function called and so on
                    PopulationChange(response, LargeEffect);
                    break;
                case 1:
                    RoadNetworkChange(response, LargeEffect);
                    break;
                case 2:
```

```
                    AirCleanlinessChange(response, LargeEffect);
                    break;
                case 3:
                    RecreationChange(response, LargeEffect);
                    break;
                case 4:
                    HousingChange(response, LargeEffect);
                    break;
                case 5:
                    JobsChange(response, LargeEffect);
                    break;
                case 6:
                    EducationChange(response, LargeEffect);
                    break;
                case 7:
                    WildlifeChange(response, LargeEffect);
                    break;
                default:
                    break;
            }
        }

        public double DetermineFactors(bool response, bool LargeEffect)
        {
            ///the two variables used in the change functions are determined here.
            ///they are determined by the two bools passed in.
            double factor;
            //if response is true then the value will increase (>1)
            //if response is false then the value will decrease (<1)
            if (response)
            {
                if (LargeEffect) { factor = 1.3; }
                else { factor = 1.1; }
            }
            else
            {
                if (LargeEffect) { factor = 0.7; }
                else { factor = 0.9; }
            }
            return factor;
        }
public void EvenOutPercentages(string name, bool response, bool LargeEffect)
        {
            //changes to type P attributes will happen in proportion to the others.
            //also dependent on the two bools passed in.
            int PChange;
            if (LargeEffect) { PChange = 8; }
            else { PChange = 4; }
            //in this case, response determines whether the PChange should be positive or not.
            if (!response) { PChange = PChange * -1; }

            int otherChange = PChange / 4;

            foreach (AttributeBase thisobject in Attributes)
            {
                string objectname = thisobject.GetName();

                if (thisobject.GetTheType() == 'P' && objectname != name && objectname != "Free Space")
                {
                    //opposite of change is put onto the others (so it equals 100)
                    thisobject.AddToValue(-otherChange);
                }
```

```
            if (objectname == name)
            {   //change is made to the original attribute
                thisobject.AddToValue(PChange);
            }
        }

    }

    public void PopulationChange(bool response, bool LargeEffect)
    {
        //first decide how much the attributes will be affected
        //some attributes will increase in value and others will decrease in value
        //this is also decided by the response bool made by the user.
        //use on the main attribute
        double change;
        //use on linked attributes
        double oppositechange;
        //change + oppositechange = 2:
        change = DetermineFactors(response, LargeEffect);
        oppositechange = 2 - change;

        //percentage attributes act differently so different variables are used.

        //this also change other attributes
        //like housing and air cleanliness
        //so look for them and change them
        foreach (AttributeBase thisobject in Attributes)
        {
            switch (thisobject.GetName())
            {
                case "Population":
                    thisobject.ChangeValue(change);
                    break;
                //if population increases, this goes down and vice versa
                case "Air Cleanliness":
                    thisobject.ChangeValue(oppositechange);
                    break;

                case "Housing":
                    //a change in a percentage affect all of them
                    //if an initial negative change, add minus sign to response.
                    //false is given here, where normally the LargeEffect bool
                    //is put, this is because this is a secondary 'same way'
                    //change. So the change should be smaller.
                    EvenOutPercentages("Housing", response, false);
                    break;

                default:
                    break;
            }
        }
    }
    //all the _Change subs do similar things
    //but they are specific to the attribute so they can't be put into one sub
    public void RoadNetworkChange(bool response, bool LargeEffect)
    {
        double change;
        double oppositechange;
        change = DetermineFactors(response, LargeEffect);
        oppositechange = 2 - change;

        foreach (AttributeBase thisobject in Attributes)
```

```
            {
                switch (thisobject.GetName())
                {
                    case "Road Network":
                        thisobject.ChangeValue(change);
                        break;

                    case "Air Cleanliness":
                        thisobject.ChangeValue(oppositechange);
                        break;

                    case "Wildlife":
                        EvenOutPercentages("Wildlife", !response, LargeEffect);
                        break;

                    default:
                        break;
                }

            }

        }
public void AirCleanlinessChange(bool response, bool LargeEffect)
        {
            double change;
            double oppositechange;
            change = DetermineFactors(response, LargeEffect);
            oppositechange = 2 - change;

            foreach (AttributeBase thisobject in Attributes)
            {
                switch (thisobject.GetName())
                {
                    case "Air Cleanliness":
                        thisobject.ChangeValue(change);
                        break;

                    case "Wildlife":
                        //false is added so a small change is made every time
                        //as it is a secondary same change
                        EvenOutPercentages("Wildlife", response, false);
                        break;
                    default:
                        break;
                }
            }
        }

        public void RecreationChange(bool response, bool LargeEffect)
        {
            double change;
            double oppositechange;
            change = DetermineFactors(response, LargeEffect);
            oppositechange = 2 - change;

            foreach (AttributeBase thisobject in Attributes)
            {
                switch (thisobject.GetName())
                {
                    case "Recreation":
                        EvenOutPercentages("Recreation", response, LargeEffect);
                        break;
```

```
                    case "Population":
                        thisobject.ChangeValue(DetermineFactors(response, false)); //secondary same
change
                        break;

                    case "Air Cleanliness":
                        thisobject.ChangeValue(oppositechange);
                        break;
                    default:
                        break;
                }
            }
        }

        public void HousingChange(bool response, bool LargeEffect)
        {
            double change;
            double oppositechange;
            change = DetermineFactors(response, LargeEffect);
            oppositechange = 2 - change;

            foreach (AttributeBase thisobject in Attributes)
            {
                switch (thisobject.GetName())
                {
                    case "Housing":
                        EvenOutPercentages("Housing", response, LargeEffect);
                        break;

                    case "Population":
                        thisobject.ChangeValue(DetermineFactors(response, false));
                        break;

                    case "AirCleanliness":
                        thisobject.ChangeValue(oppositechange);
                        break;
                    default:
                        break;
                }
            }
        }

        public void JobsChange(bool response, bool LargeEffect)
        {
            double change;
            double oppositechange;
            change = DetermineFactors(response, LargeEffect);
            oppositechange = 2 - change;

            foreach (AttributeBase thisobject in Attributes)
            {
                switch (thisobject.GetName())
                {
                    case "Jobs":
                        EvenOutPercentages("Jobs", response, LargeEffect);
                        break;

                    case "Air Cleanliness":
                        thisobject.ChangeValue(oppositechange);
                        break;
                    default:
```

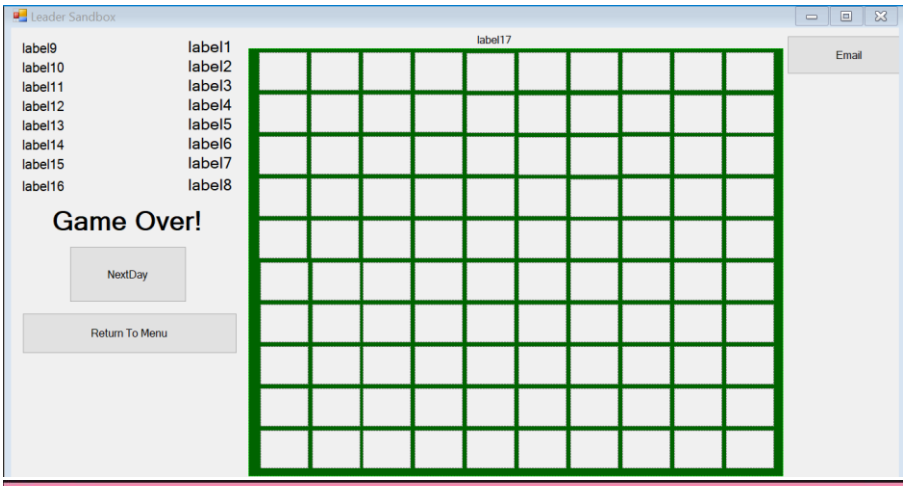
```
                        break;
                }
            }
        }
public void EducationChange(bool response, bool LargeEffect)
        {
            double change;
            double oppositechange;
            change = DetermineFactors(response, LargeEffect);
            oppositechange = 2 - change;

            foreach (AttributeBase thisobject in Attributes)
            {
                switch (thisobject.GetName())
                {
                    case "Education":
                        EvenOutPercentages("Education", response, LargeEffect);
                        break;

                    case "AirCleanliness":
                        thisobject.ChangeValue(DetermineFactors(response, false));
                        break;
                    default:
                        break;
                }
            }
        }

        public void WildlifeChange(bool response, bool LargeEffect)
        {
            double change;
            double oppositechange;
            change = DetermineFactors(response, LargeEffect);
            oppositechange = 2 - change;

            foreach (AttributeBase thisobject in Attributes)
            {
                switch (thisobject.GetName())
                {
                    case "Wildlife":
                        EvenOutPercentages("Wildlife", response, LargeEffect);
                        break;

                    case "Air Cleanliness":
                        thisobject.ChangeValue(DetermineFactors(response, false));
                        break;
                    default:
                        break;
                }
            }
        }
        //adding new buildings
         public void EditBuildings()
        {
            ///every 2 turn, the land length will increment, this means that the map
            ///should also change. The percentages can be get-ten from the attribute class' getter
method
            ///first find how many of a certain building should be on the map then count how many ARE
on the map
            ///then add or remove as needed.
            //find the area
            int MapArea = LandLength * LandLength;
```

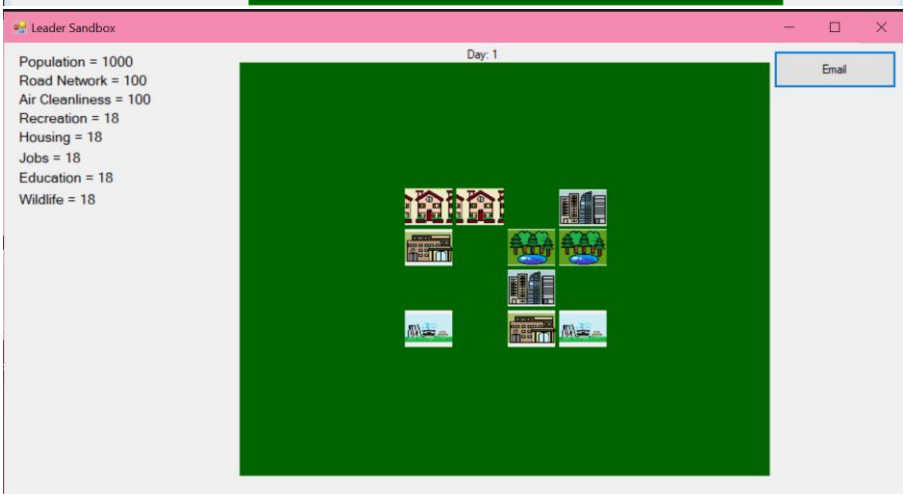
```
            int TotalPercentage = FindTotalPercentage();
            foreach (AttributeBase thisattribute in Attributes)
            {   //only the percentage attributes show on the map
                if (thisattribute.GetTheType() == 'P')
                {
                    //find how much is needed
                    int NeededNoOnMap = Convert.ToInt32(Math.Truncate((MapArea) *
(thisattribute.GetValue()/TotalPercentage)));
                    //negative values will be set as zero
                    if (NeededNoOnMap < 0) { NeededNoOnMap = 0; }
                    //find out how much is on the map
                    char BuildingChar = FindBuildingChar(thisattribute.GetName());
                    int NoOnMap = FindNoOnMap(BuildingChar);
                    //this will give you how many you need to add/remove
                    int BuildingDifference = NeededNoOnMap - NoOnMap;
                    while (BuildingDifference != 0)
                    {   //add and remove buildings until no more changes are needed
                        if (BuildingDifference > 0) { BuildingDifference = AddBuildings(BuildingChar,
BuildingDifference); }
                        else if (BuildingDifference < 0) { BuildingDifference =
RemoveBuildings(BuildingChar, BuildingDifference); }
                    }
                }
            }
            //show the changes on the form
            LoadTown();

        }

        public int FindTotalPercentage()
        {   //as the values of the percentage attributes change,
            //the total may change from 100. Find the total here.
            int total = 0;
            foreach (AttributeBase thisattribute in Attributes)
            {
                if (thisattribute.GetTheType() == 'P' && thisattribute.GetValue() > 0)
                {
                    total += Convert.ToInt32(thisattribute.GetValue());
                }
            }
            return total;
        }
        public char FindBuildingChar(string name)
        {   //switch and case that hold the chars for the attributes
            char Char = ' ';
            switch (name)
            {
                case "Recreation":
                    Char = 'R';
                        break;
                case "Housing":
                    Char = 'H';
                    break;
                case "Jobs":
                    Char = 'J';
                    break;
                case "Education":
                    Char = 'E';
                    break;
                case "Wildlife":
                    Char = 'W';
                    break;
```

```
                case "Free Space":
                    Char = 'F';
                    break;
                default:
                    break;
            }
            return Char;
        }

        public int FindNoOnMap(char Char)
        {   //choose the available spaces by using land length
            int LeftUpSpace = (10 - LandLength) / 2;
            int RightDownSpace = LeftUpSpace;
            //if it's an odd value then add a 1 to the other value
            if (LandLength % 2 == 1) { RightDownSpace += 1; }
            int BuildingCount = 0;

            for (int i = LeftUpSpace; i < 10 - RightDownSpace; i++)
            {
                for (int j = LeftUpSpace; j < 10 - RightDownSpace; j++) //go across the row.
                {   //if the right char is read, increment the counter
                    if (Char == TownMap[i,j]) { BuildingCount++; }
                }
            }//return number of buildings on the map
            return BuildingCount;
        }

        public int AddBuildings(char BuildingChar, int NoToAdd)
        {
            int LeftUpSpace = (10 - LandLength) / 2;
            int RightDownSpace = LeftUpSpace;
            //if length is an odd number, the greater number of spaces will be at the right and below
            if (LandLength % 2 == 1) { RightDownSpace += 1; }

            ///say leftup = 2 and rightdown = 3
            ///up down difference is 2 < i < 7, so 5 with is the sum of the variables
            /// left right difference is 2 < j < 7, same as above

            for (int i = LeftUpSpace; i < 10 - RightDownSpace; i++)
            {
                for (int j = LeftUpSpace; j < 10 - RightDownSpace; j++)
                {
                    //20% chance a building will be put there
                    if ((TownMap[i, j] == '-' || TownMap[i, j] == 'F') && NoToAdd >0 && (Random(5) ==
1))
                    { TownMap[i, j] = BuildingChar; NoToAdd--;}
                }
            }
            return NoToAdd;
        }

        public int RemoveBuildings(char BuildingChar, int NoToRemove)
        {   //same code as AddBuildings
            int LeftUpSpace = (10 - LandLength) / 2;
            int RightDownSpace = LeftUpSpace;
            if (LandLength % 2 == 1) { RightDownSpace += 1; }

            for (int i = LeftUpSpace; i < 10 - RightDownSpace; i++)
            {
                for (int j = LeftUpSpace; j < 10 - RightDownSpace; j++)
                {
                    //20% chance a building will be removed from there
```

```
                    if (TownMap[i, j] == BuildingChar && NoToRemove < 0 && (Random(5) == 1))
                    { TownMap[i, j] = '-'; NoToRemove++; }
                }
            }
            return NoToRemove;
        }
//classes below
        public class AttributeBase
        {   //variables that used to hold info about an attribute
            public string name;
            public int value;
            public int startvalue;
            public float multiplier;
            public char type;
            public int lettermarker;
            public bool allllettersread;
            public int valuechange;

            public AttributeBase() { allllettersread = false; }
            //setters
            public void SetName(string name) { this.name = name; }
            public void SetValue(int num) { this.value = num; }
            public void SetStartValue(int num) { this.startvalue = num; }
            public void SetMultiplier(float num) { this.multiplier = num; }
            public void SetType(char Char) { this.type = Char; }
            public void SetMarker(int num) { this.lettermarker = num; }
            public void SetValueChange(int num) { this.valuechange = num; }
            //actions
            //used when the day is advanced
            public void Advance() { value = Convert.ToInt32(value * multiplier); }
            public void ChangeValue(double factor) //when a change is made due to a letter
            {
                int pastvalue = value;
                if (value == 1 && factor <= 1.0) { value = 0; }
                else { value = Convert.ToInt32(value * factor); }
                valuechange = valuechange + (value - pastvalue);//value change is used to show the
change
            }
            public void AddToValue(int factor)
            { //used when a percentage attribute is changed due to a letter
                value += factor;
                valuechange += factor;
            }
            public void MakeAllRead() { allllettersread = true; }
            //getters
            public string GetName() { return name; }
            public float GetValue() { return value; }
            public float GetStartValue() { return startvalue; }
            public float GetMultiplier() { return multiplier; }
            public char GetTheType() { return type; }
            public int GetMarker() { return lettermarker; }
            public bool GetAllRead() { return allllettersread; }
            public int GetValueChange() { return valuechange; }

        }

        public class LetterLayout
        {
            private int ID;
            private string Text;
            private bool Read;
            //every time a new letter is made, the id and text will be recorded
```

```
        //the letter will always be unread at first.
        public LetterLayout(int id, string text)
        {
            this.ID = id;
            this.Text = text;
            this.Read = false;
        }
        //setters and getters
        public void SetText(string text) { this.Text = text; }
        public void MakeRead() { this.Read = true; }
        public int GetID() { return ID; }
        public string GetText() { return Text; }
        public bool GetRead() { return Read; }

        //same code as in the Compressor file:
        public void DecodeText(Dictionary<int, string> letterDictionary)
        {//decodes the text held.
            int counter = 0;
            string key = "";
            string newtext = "";

            while (counter <= Text.Length - 1)
            {
                if (!Char.IsNumber(Convert.ToChar(Text.Substring(counter, 1))))
                {
                    newtext = newtext + GiveValue(Convert.ToInt32(key), letterDictionary);
                    key = "";
                }
                key = key + Text.Substring(counter, 1);
                counter++;
            }
            this.Text = newtext;
        }

        public string GiveValue(int key, Dictionary<int, string> letterDictionary)
        {
            string output = "";
            //find the value the key corresponds to and return the value
            foreach (KeyValuePair<int, string> item in letterDictionary)
            {
                if (item.Key == key)
                {
                    output = item.Value;
                }
            }
            return output;
        }
    }
```





When the form is first loaded, this is what appears.

←

Txt File Contents:

CompressedLetterFile.txt

```
0 1 2 3 1 4
5 1 2 3 1 4
0 1 2 3 1 4
5 1 2 3 1 4
0 1 2 3 1 4
5 1 2 3 1 4
0 1 2 3 1 4
5 1 2 3 1 4
0 1 2 3 1 4
5 1 2 3 1 4
0 1 2 3 1 6 1 7
5 1 2 3 1 6 1 7
0 1 2 3 1 6 1 7
5 1 2 3 1 6 1 7
0 1 2 3 1 6 1 7
5 1 2 3 1 6 1 7
0 1 2 3 1 6 1 7
5 1 2 3 1 6 1 7
0 1 2 3 1 6 1 7
5 1 2 3 1 6 1 7
0 1 2 3 1 8 1 9
5 1 2 3 1 8 1 9
0 1 2 3 1 8 1 9
5 1 2 3 1 8 1 9
0 1 2 3 1 8 1 9
5 1 2 3 1 8 1 9
0 1 2 3 1 8 1 9
5 1 2 3 1 8 1 9
```

Dictionary.txt

```
0
LargeEffect
1

2

3
-
4
Population
5
SmallEffect
6
Road
7
Network
8
Air
9
Cleanliness
10
Recreation
11
Housing
12
Jobs
13
Education
14
Wildlife
```

TownBaseVariables.txt

```
Population
1000
1.2
I
Road Network
100
0.95
D
Air Cleanliness
100
0.80
D
Recreation
18
1.0
P
Housing
18
1.0
P
Jobs
18
1.0
P
Education
18
1.0
P
Wildlife
18
1.0
P
Free Space
10
1.0
P
```

TownLayout.txt

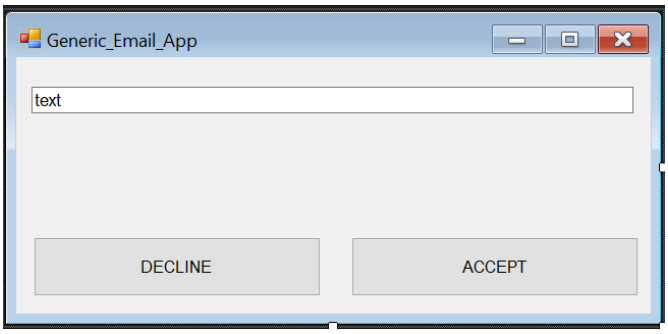
```
----------
----------
----------
---HH-J---
---E-WW---
----FJ----
---R-ER---
----------
----------
----------
```

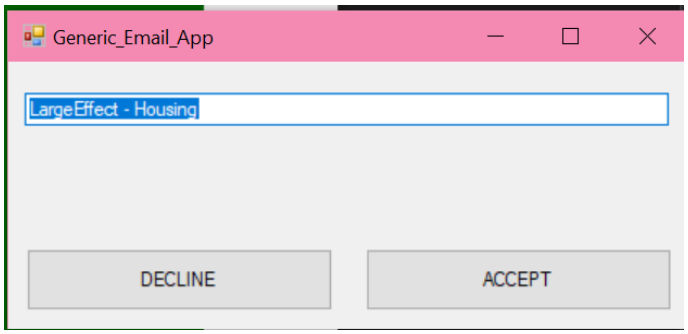
Building Images:

Letter Display (Form 3):

```
using System;
using System.Windows.Forms;

namespace Trying_To_Put_Leader_Together
{
    public partial class Generic_Email_App : Form
    {
        public Generic_Email_App()
        {
            InitializeComponent();
        }

        public static bool Response;

        private void Generic_Email_App_Load(object sender, EventArgs e)
        {
            //displays letter text for the user
            textBox1.Text = Leader_Sandbox.LetterText;
            //as the value of LetterElement changes quickly, everytime an instance
            //of this form is loaded, it takes the value in LetterElement
            //and saves it to a Label that is away from the sight of the user
            label1.Text = Convert.ToString(Leader_Sandbox.LetterElement);
        }

        private void Acceptbt_Click(object sender, EventArgs e)
        {
            ///when a button is pressed, it tells the CheckResponsetmr
            ///in the 2nd form that a response has been given via InputGiven.
            ///the text in the hidden label is used to tell the timer what letter was responded to.
            Response = true;
            Leader_Sandbox.InputGiven = true;
            Leader_Sandbox.LetterElement = Convert.ToInt32(label1.Text);
            this.Close();
        }

        private void Declinebt_Click(object sender, EventArgs e)
        {   //the same as the above button but response is false
            Response = false;
            Leader_Sandbox.InputGiven = true;
            Leader_Sandbox.LetterElement = Convert.ToInt32(label1.Text);
```

```
            this.Close();
        }
    }
}
```





Testing

Here I will see how far the program will let the town run if I prioritise certain attributes over other ones, by ignoring all letters referring to the tested attribute. The criteria for the simulation ending are: **Days Passed is greater than 20 OR Population is less than 500 OR Air Cleanliness is less than 0.** And if any other these criteria are met, the game will show a Game Over message and tell the user to return to the menu.

The program chooses the letters given to the user by looking at which attributes have the lowest values in comparison to its original value. For example, if the value for **Housing** was low, the program would give me more **Housing Letters** to try and make the user increase it again. This means that it is easier to decrease an attribute's value the lower it is as letters for it will appear more with lower values, making the testing speed up as the simulation runs. Attributes are also linked to others so that when a decision is made to increase/decrease an attribute's value, other linked attributes will also experience changes. For example, **Housing** is a **Percentage** attribute so when it is altered, the other **Percentage** attributes will also change to accommodate it; **Population** will also experience a small change according to the response given and **Air Cleanliness** will experience an opposite change to the response given, as seen in the code. This may lengthen the testing process.

To test each attribute, I will only be **declining** the **Letters** relating to the attribute being tested and **accept all others**. This means that the value of the tested attribute may increase as I go, lengthening the test, as they may be linked to other attributes' **Letters** that I would have accepted.

Population Test:

Reason for Game Over: **Day limit reached**.

As **Population** is multiplied by **1.2** every day, every change made to decrease **Population** was quickly overwritten. This resulted in less **Letters** being given for the attribute as the program determined that its level was too high for **Letters** to be given. However, when **Population Letters** were being declined, its large value resulted in larger reductions, seen in red below:

The increase of **Population's** value also accelerated due to it being linked to multiple other attributes like **Housing, Jobs, Education** and **Wildlife**, when they increased in value, so would **Population**.

When **Population Letters** are declined, **Housing** decreases as well, which explains why there are only **2 Housing** spaces on the map.

Therefore, it is difficult for the population to decrease if only **Population Letters** are being accepted.

Road Network Test:

Reason for Game Over: **Day limit reached**.

Road Network is a decreasing attribute, so its value is multiplied by **0.95** every day. This means that initially, the attribute's value decreased quickly as more **Letters** were being given to decrease its value. However, once the value reached **5**, the day changes would not affect it anymore, this occurred due to a maths error. As **5 / 0.95 = 4.75,** it is **rounded up** by to **5 again**, giving no overall change. As there was not any change in value, the program assumes that the value is not low and does not give the user anymore **Letters**, even though the value is low.

Road Network's value does not affect the ending criteria and no visual indicator on the map. However, when its **Letters** are declined, it allows **Air Cleanliness** and **Wildlife** to increase in value. Which explains why the map is overwhelmed with **Wildlife** spaces.

Air Cleanliness Test:

Reason for Game Over: **Air Cleanliness is less than 0**.

Air Cleanliness is a decreasing attribute, so its value is multiplied by **0.80** every day. This causes the value decrease very quickly, **Air Cleanliness** is also affected negatively when the values of **Road Network, Housing** and **Population** for example, increase. This causes the attribute to be the greatest problem to users and they must constantly keep the value up to let the town last for longer. As the value of **Air Cleanliness** got lower, I was given more **Letters** to decrease the values further. In the end, the town lasted for **9 days**, this was very quick as the full map had not been unlocked yet, as seen below.

As Air Cleanliness' Letters we declined, Wildlife also decreased, as they are linked. Which explains why there are very few Wildlife spaces on the map in comparison to housing and education.

Recreation Test:

Reason for Game Over: **Day limit reached**.

Recreation is one of the 5 (not including **Free Space** as it does not change) **Percentage** attributes, so it will affect and be affected by changes in the other changes in the **Percentage** attributes. Declining **Recreation Letters** would also cause **Population** to decrease and **Air Cleanliness** to increase. By looking at the screenshot below, we can see that the simulation was about to end due to **Air Cleanliness** almost reaching 0, however, we can say this is due to other factors as **Recreation** causes the value to that attribute to increase.

In the screenshot, we can see that there are no **Recreation** spaces on the map instead, it is filled with other buildings instead.

Housing Test:

Reason for Game Over: **Air Cleanliness is less than 0**.

The simulation lasted for 16 days and behaved similarly to the **Recreation** test, even though there are different values held in the linked attributes. For example, as **Housing** decreases, **Population** will also decrease, and **Air Cleanliness** would increase. However, **Air Cleanliness** still reached 0 as the other **Letters**, that were accepted as the test went on, caused the value to decrease.

On the map, there are no **Housing** spaces on the town map as the value is negative.

Jobs Test:

Reason for Game Over: **Day limit reached**.

The test also behaved the same as the rest of the percentage attributes, however in this case, **Air Cleanliness** does not have a value equal to or less than **1** at the end. Jobs is only linked to **Air Cleanliness** and when a **Jobs Letter** is rejected, it will increase, which is why there is a higher value than usual.

As usual, the map has no **Jobs** spaces and is instead filled with the other spaces, most notably **Housing**.

Education Test:

Reason for Game Over: **Air Cleanliness is less than 0**.

The simulation lasted for 13 days. **Air Cleanliness** decreased quickly as it is linked to the tested attribute, **Education**. When an **Education Letter** is declined, **Air Cleanliness** also decreases, this caused the value of it to drop heavily, causing the town to not last very long. There are no **Education** spaces on the board as its value is negative.

Wildlife Test:

Reason for Game Over: **Air Cleanliness is less than 0**.

The simulation lasted for 13 days. This is because **Wildlife** is linked to **Air Cleanliness** so when **Letters** for **Wildlife** were declined, the value of **Air Cleanliness** also decreased, which then caused the simulation to end quickly. As seen below, there are no **Wildlife** spaces on the town map, as its value is negative.

Conclusion to Testing:

By looking at the test above, we can see that, to make a town that lasts for longer, the user must prioritise: **Air Cleanliness, Wildlife, Education** and **Housing**. This is because when the **Letters** for those attributes were ignored, the simulation was not able to reach 20 days. The simulation survived for a minimum of 9 days, which can be seen in the **Air Cleanliness** test. Out of the 8 tests done, only half of them ended due to **Air Cleanliness** reaching 0, and the other half ended due to the **Day limit** being reached. Therefore, it is very unlikely that population would reach <500 and end the game.

Evaluation

Objective 1, **"Have a project that the user can interact with",** was successfully implemented, as the user is able to choose when to do certain functions, like check emails and move on to the next day. The user is also able to use the **Letters** to alter the attribute values and hence change the look of the town.

Objective 2, **"Show the user what their decisions do to the town in real time",** was successfully implemented as there is a map of 100 **pictureboxes** on the form. Each box represents a space on the town map and, as the player makes decisions, the types of images held in each **picturebox** changes. Every decision the user makes updates the map so the user can see what their choices are doing in real time. Due to the random nature of the **Add/Remove Buildings** functions, each simulation will give a different pattern of buildings in the map.

Objective 3, **"Make the program's objects interact with each other, over time",** was successfully implemented to an extent, if each attribute is referred to as objects. When a change is made to one attribute, its specialised **__Change** function will also alter the values of attributes like the original one. Hence, over time, the changes done to an attribute will affect others.

Objective 4, **"Investigate which strategies allow the town to last longer",** was done earlier where it was found that there was a 50/50 chance of the simulation ending due to **Air Cleanliness** reaching 0 or the **Day limit** being reached. **Air Cleanliness, Wildlife, Education** and **Housing** must be prioritised to allow the town to last for longer.

Objective 5, **"Make my code more presentable by reducing the number of strings that are visible in the code",** was successfully implemented to an extent. All strings for the **Letters** we compressed using the **Compressor** project and then decoded within the **Main Program,** so there are no long lines of strings left in the code. However, originally, I wanted to make each **Letter** more unique and have sentences describing the things being done to the town, to change the values. This would have made the **Compressor** code more useful. As I ran out of time, I had to leave it in the **SmallChange/LargeChange - (attribute name)** form seen in the screenshots.

Objective 6, **"Have all the emails/letters, that the user must respond to show up immediately after the user states that they want to see them",** was successfully implemented. When the user presses the 'Email' button, the three emails are immediately given to the user and stay up until the user answers them.

Therefore, I have been able to accomplish most of my initial goals that I had originally made for the project. However due to time I had to cut multiple ideas that I had for the program. Originally, I wanted to have a file system where the user was able to save their progress to one of three files and choose a file to return to once they re-opened the program. However, as I saved my text files within the project solution, and not externally, there was no way for me to use a **Streamwriter** on the text files and have the results be remembered next time the program is run. As I tested, I also found that the town could survive for longer, so if I had more time to work on the project, I would remove the 20-day limit and fix problems with the **Letter Read** system to allow the simulation to run for longer.