|  |
| --- |
|  |
| A-Level Computer Science Project |
| Employee management system |

|  |
| --- |
| Ryan Krage |

**Introduction**

My client owns a small company, with several employees on a zero-hour contract. He is currently tracking each employee’s salary in a spreadsheet, but this is cumbersome and unprofessional. He would prefer to have a bespoke application to more easily manage each employee’s salary.  
For this project I will analyse the clients problem, design a solution, implement it and then test it.

### Disclaimer This is a Computer Science A-level project, and is not commercial software. It is provided to the client with no warranty or guarantee it will fulfil the client’s requests, stated for implied.

**Analysis**

**Interview**I asked a series of questions to establish the scope of the problem and to find out what my client wanted the application to do. The following is a summarization of the conversation we had over IM (it has been slightly edited for clarity);

Me: How many employees do you currently have?  
Client: Currently we have six employees.  
M: Will the program need to be able to scale up to accommodate more employees in the future?  
C: Yes, it will need to scale up in the future.  
M: How exactly does the zero-hour contract work?  
C: Employees are paid according to how much work they have completed, not how many hours they have worked.  
M: Who will use the application? Just yourself, or employees too?  
C: Just myself.  
M: Should the data be exportable? If so, in what format?  
C: Yes, to a spreadsheet.  
M: What are the things the application must be able to do?  
C: Have an input for the number of tasks completed, the pay per task, employee name, total amount paid and how much is due.  
M: What features would be nice to have?  
C: Save, export and import.

Based on this, I came up with a set of functions and features for the application, which I also used to make the design diagrams.

**Functions**These are things that the application must do to provide a basic level of functionality.  
The application must record set tasks for each employee. The tasks will need some additional information, such as;

* task name/description
* pay (how much it is worth)
* deadline
* status (complete or not)

The application must also calculate the pay each employee is due, based on how many of the set tasks they have completed.

**Features**These are additional functionality, and are things the application should do to make it easier to use.

**Export and import data**The application should export the data as a format that can be used by a spreadsheet program such as Excel. The format for importing was never specified, and as there will only be one instance of this application, importing seems unnecessary and so will not be in the application.

**Employee information**Another feature that I added which the client did not request is the option to add information about the employees, such as the date they were employed, contact information and their role within the company.

**Employee image**I also decided to add the option to include an image of the employee, which would make it easier to see which employee’s profile was open. I decided to add this as it will be simple to implement and be a useful feature.

**Basic Design Choices**I made the decision to use a GUI rather than a command line interface for several reasons. Firstly, most users are more comfortable using a GUI than a CLI, so it will the program easier to use. As most users are familiar with a GUI, less training would be required than if the program used a CLI.  
Secondly, Visual Studio comes with many features that make it very easy to create a Windows Form, so I will not have to program everything from scratch. Lastly, by using a Windows Form, the code will be clearer to other programmers than if I created a GUI from scratch, as it will use standard WinForm elements.  
Another advantage of using a GUI rather than CLI is that program used to create one will use an event-driven model. Event-driven programs are generally simpler to code as the programmer only has to account for events they intend for the user to use, as it will be impossible for the program to do anything else.

**Scope**While it is possible to create enterprise-level software to do what the client has requested (such as one with networking that utilises database servers), however, this is not what the client needs or has requested and as such it is outside the scope of this project. Therefore, I will be limiting the scope of the application to something more manageable. The program will be stand-alone, not requiring any external libraries or dependencies to run. It will not have networking, as the client has stated that only he will use it on one local machine. It will not use a database management system, as a fully-featured database would well exceed the requirements. Features such as multiple tables, relationships and querying large volumes of data will not be needed. Instead, the necessary data management can be done using text files, as updating and deleting if records won’t be required. Appending data can be easily implemented with text files, especially concerning the small number of records likely to be involved. Not using a DBMS will also reduce disk and memory usage, increasing performance.  
The program will also not have an installer - instead, it will run ‘out of the box’ (i.e, be a standalone .exe that can run without being installed). (See Appendix 1.2 for more on relative filepaths)

**Input/Output**The program will take input from the user, through the windows form. The form will have buttons and text boxes for entering information such as employee and task details. The output will also be presented via the form, largely through labels. Another form of I/O that is not presented to the user is file reading & writing. The programs data must be saved to a file (output), and then read when the program is opened again (input).

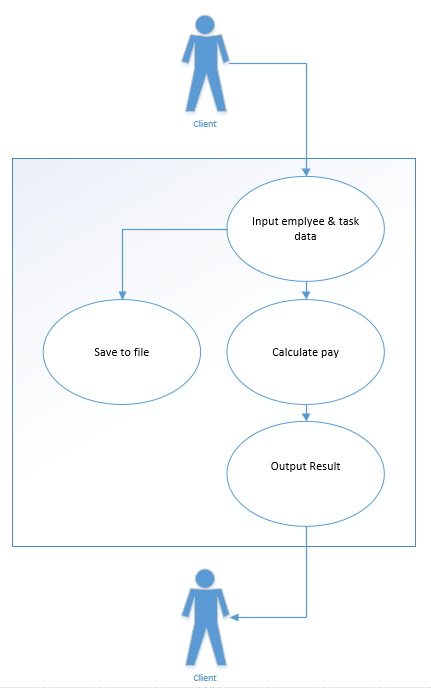
**Sufficient Complexity**While the basic premise of this software is quite simple, the client has requested some features that make it more complex. While the actual calculation of pay is very simple, the complexity comes from the data management aspect of the problem. Firstly, the data needs to presented in an easily human-readable format. This will require selecting data and formatting the output accordingly. This leads to the next non-basic function - saving data. For a project of this size, plain text is a good storage medium as it is easy to write and read data from using built-in features of the language. This project will likely require reading data from a file non-sequentially, and manipulating the strings read from the file, for example by splitting them into arrays.  
Thirdly, this will be an event-driven program, so the user can input data in any order. Therefore, it must validate this data before passing it to other functions, to ensure that the user cannot break or crash the program by accident.

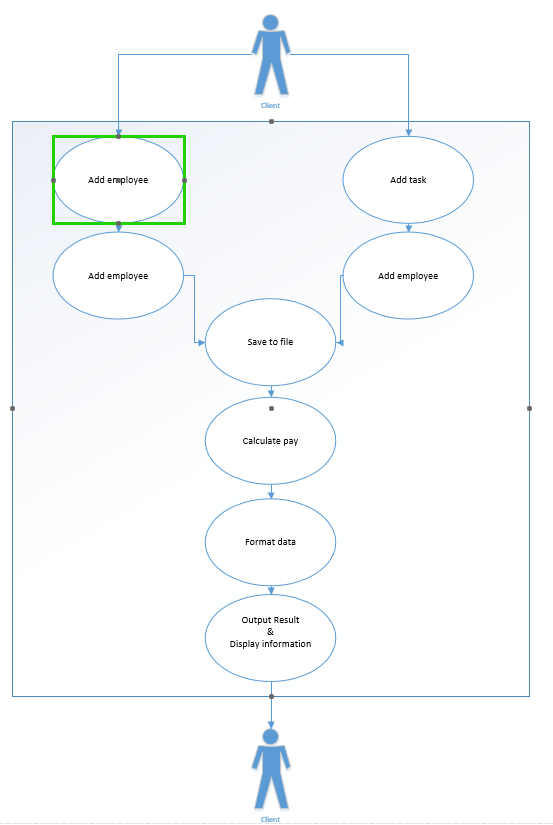
**Priorities**In order to complete this project within the deadline, it is important to prioritise which aspects of the software I will focus on first. For the implementation of the software I will focus first on the functions I outlined above. Then, I will implement the features my client has requested. Lastly, I will worry about the feasibility of the features. This seems counter-intuitive, as logic dictates that a problem cannot be solved unless it is possible to solve. However, as almost any feature a client will request is possible to implement, it will save time to only worry about the feasibility of a function or feature if and when a problem with implementing it arises during the design stage of the software life cycle. This way I will not spend too long worrying about how to implement a feature before I need to.  
This would enable me to use a prototyping development model. The advantages of this include more flexibility with implementing features. For example, if I discover a function doesn’t work as intended, I can spot this early on and an alternative approach.  
While prototyping is suitable for small projects such as this one, it does not scale well and would not be suitable for larger, more complex projects involving more developers. Prototyping also often results in projects exceeding deadlines and running over budget. Another problem with prototyping is ‘project creep’ - when the client sees the incomplete product and is inspired to request additional functions and features which were not part of the original specification. This is the main cause of the previous problem.

**IOPS**An Input Output Processing Storage diagram will help to understand the flow of data within the program and how the client interacts with the program. This will help in subsequent stages of the project, such as design.

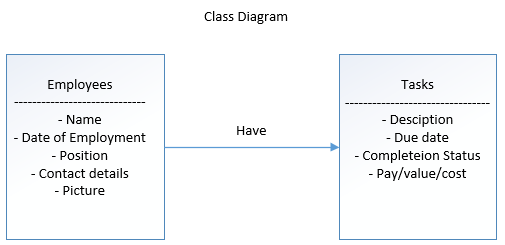
| **Input** | **Processing** | **Output** | **Storage** |
| --- | --- | --- | --- |
| Add employee | Create an employee object | Display the employee info | Save the employee details to file |
| Assign task to employee | Create a task object | format the task details and display them | Save the task details to file |
| edit employee details | update the details | refresh the output to reflect the changes | save the changes |
| edit task details | update the details | refresh the output to reflect the changes | save the changes |
| Add an image | add the filepath to the employee details | display the image | save the filepath in the employee details |

**Use-case Diagram**To help me better understand how the client will use the program, I made two use-case diagrams. One shows the client current workflow (using a spreadsheet), and the other shows what the improved workflow with the new software will look like.





**Class Diagram**I have created a class diagram to show the relationship between employees and tasks. While very simple, this could be considered a relational database (Employees have a one-to-many relationship with tasks), but the data set is small enough that a full-blown DBMS is not required (see Scope, above). It also makes it clear which attributes belong to which class.



**S.M.A.R.T objectives**In order to help with evaluating the success of this project, I have used the above information to create a set of Specific, Measurable, Achievable, Relevant, Time-Bound objectives to complete. The goals are things the program should do in order to fulfil the clients requests. The objectives must be S.M.A.R.T so that can be used as a measure of progress and success. While the objectives themselves are not necessarily time-bound, I will follow the schedule in appendix 3.1, so all of these goals do have a deadline.

| **Ensure that:** | **Specific** | **Measurable** | **Achievable** | **Relevant** | **Time-bound** |
| --- | --- | --- | --- | --- | --- |
| User can add employees | Y | Y | Y | Y | Y |
| User can edit employees | Y | Y | Y | Y | Y |
| Program correctly adds pay according to complete tasks | Y | Y | Y | Y | Y |
| Program saves data | Y | Y | Y | Y | Y |
| Program retrieves data correctly | Y | Y | Y | Y | Y |

**Design**

### Setting & completing tasks The primary purpose of the application is to calculate pay for each employee, so first there must be a way to represent and store information about employees. Using OOP (object-orientated programming), each employee could be an instance of a class (created when the user adds a new employee), with tasks as properties of the employee class. This solution would require the user to re-enter all of the employees information and tasks each time, so it would be even less useful than the spreadsheet my client is currently using. A better solution would be to create a file to store information about each employee. This could be a simple text file, with the name of each task (and whether or not it is completed) on each line. This solution also allows the program to save the data, satisfying two requirements in one. Assuming that different tasks have different pay, this will also be saved in the file. Each line could have the format of taskName;pay;completed.

### Calculating pay The calculation is very simple, number of tasks completed \* pay per task. There will need to be a setting for the pay per task. As different tasks may be worth different amounts, the pay will be set when the task is created.

### Saving data I have already outlined how data for tasks will be saved (in a text file), however it may be necessary to save other data about employees, such as contact details. This could be incorporated into the employees file. A set number of lines can be dedicated to each section (e.g; first 5 lines for employee information, then line 6 onwards for tasks).

### Exporting data A huge part of exporting has been solved already - the information will be saved in separate text files. This means exporting is a relatively simple matter of formatting the data for different purposes. It would be best to export to .XML, as this format can be used a variety of spreadsheet programs such as Excel, Libre Office Calc and Google Sheets. Depending on the complexity of this format, more research is needed to determine if this is within the scope of this project. If it is not, then the data could be formatted for easy reading in .txt format.

### Aesthetics While I have more experience with making console applications, it would be much easier for the user if I created a windows form application instead. My first draft of the design is below;

### On the left, there is a scrollable list of employees. The user can add, remove or edit employees through the ‘Edit’ menu at the top. In the middle is a section to show information about the employees. There will also be an option to add a picture, to make it easier to identify which employee is currently selected at a glance. On the right is a list of tasks for the currently selected employee. Here, the user can add and remove tasks, as well as editing details such as the description, when it should be completed by, whether or not it has been completed, and how much it is worth.

#### **Making a GUI** By looking at some tutorials for making Windows Forms (which I determined was the easiest way to make a GUI with C#), I have outlined some ideas for how to make GUI;

##### The employee list This could be a ListView, a winform element that can display a list of items. These items can have sub-items, which could be used to store information about the employee, and tasks.

##### The employee details These could be shown using labels, which are basically just text that can be programmatically updated. The image could be shown using an imageBox, a winform element for displaying images. It can automatically resize them, so limiting the user to a particular resolution isn’t required. Limiting filetypes for this image might be.

##### The task list This will probably be another ListView, but it’s contents will be taken from the subitems of the first. It can be programmatically updated whenever an employee is selected.

##### Titles, menus, etc There is a standard menu element for winforms, and titles can simply be labels.

### Help section/manual To ensure the client can use the software without assistance, I will include a manual that can be opened from within the application, probably in the form of a web page.

### Install/portability The program will not require installation, but as it requires resources (such as employee data and the help section), I will need to use relative filepaths to locate these resources. See Appendix 1.2 for more on Relative filepaths.

### Validation As this software will be event-driven, it is important to ensure the user’s input is correctly validated to avoid exceptions and other errors. The easiest way to do this is remove the need for validation - this can be done to any inputs that are not actually processed, such as employee details. As the data is not used by the program, the user can enter anything they like. Next, is to limit what the user is able to enter. This can be achieved using masks, which will stop invalid input from appearing in the input box. For example, if a text box requires a number, the mask will stop anything other than integers appearing in the input box. Lastly, is checking the input to see if it meets the criteria. One place this may be need is for editing employees and their tasks - an employee cannot be edited if none exist, and a task cannot be created if there is no employee to assign it to. This means the program will need check there is actually something to edit when the user clicks the edit button.

### Flowchart This is a flow chart I created to show how the program will work. As the program is event-driven, it does not translate well to flowchart format, but will still help in designing the program.

### 

**Technical solution**

The following is the code used to create the form, and process input and output. There were other files automatically generated by Visual studio, but as I did not code them myself, I have declined to include them here.  
The code for the help feature can be found in Appendix 1.1.  
The code is annotated through comments I wrote throughout the prototyping process. Should they not be clear enough, refer to the flowchart on the previous page.

**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;

**using** System.IO;

**using** System.Reflection;

**namespace** compSciProject

{

**public** partial **class** Form1 : Form

{

**public** Form1()

{

InitializeComponent();

}

//this is where global variables go

**public** **static** **class** Globals

{

//selectedItem isused to store the index of the last selected employee so it can be programatically re-selected when needed.

**public** **static** **int** selectedItem = 0;

//path is the directory the program is running in, so all filepaths are relative that.

//this makes the application portable. Just copy the release folder to another PC or USB stick.

**public** **static** **string** path = Path.GetDirectoryName(Assembly.GetEntryAssembly().Location);

}

**private** **void** navigatewToolStripMenuItem\_Click(**object** sender, EventArgs e)

{

}

//SELECT EMPLOYEE

**private** **void** listView1\_SelectedIndexChanged(**object** sender, EventArgs e)

{

//check that only one item is selected

**if** (listView1.SelectedItems.Count == 1)

{

//assign the selected item to a var so if deselected it can be programatically re-selected later.

Globals.selectedItem = listView1.SelectedIndices[0];

//update the labels to match their subItem text

label8.Text = listView1.SelectedItems[0].SubItems[1].Text;

label9.Text = listView1.SelectedItems[0].SubItems[2].Text;

label10.Text = listView1.SelectedItems[0].SubItems[3].Text;

label11.Text = listView1.SelectedItems[0].SubItems[4].Text;

//update the image from the filepath in the 5th SubItem

pictureBox1.Image = Image.FromFile(listView1.SelectedItems[0].SubItems[5].Text);

//Now to re-populate ListView2 with tasks specific to the employee, from the employees file.

//each line of [file].txt becomes an element of the array lines

//first, clear the listView

listView2.Clear();

//then, we generate the filepath for the selected employee;

**int** filename = Globals.selectedItem;

**string** name = filename.ToString();

**string** path = Globals.path + "\\data\\" + name + ".txt";

//then we re-populate the listview from that file

**string**[] lines = System.IO.File.ReadAllLines(path);

//each element is dealt with one at a time

**foreach** (**string** line **in** lines)

{

//create an array 'info'. Each element in info is one peice of information from the line in [file].txt

**string**[] info = line.Split(',');

ListViewItem item = listView2.Items.Add(info[0]);

item.SubItems.Add(info[1]);

item.SubItems.Add(info[2]);

item.SubItems.Add(info[3]);

item.SubItems.Add(info[4]);

}

//check if ListView is populated, and if so, select the first item & then calculate the employees due pay

**if** (listView2.Items.Count != 0)

{

listView2.Items[0].Selected = **true**;

//calculate the employee's pay

**float** pay = 0;

**foreach** (ListViewItem item **in** listView2.Items)

{

**if** (item.SubItems[4].Text == "Yes")

{

pay += **float**.Parse(item.SubItems[3].Text);

}

}

//update label showing pay

label23.Text = String.Format("{0:0,0.00}", pay);

}

**else**

{

label23.Text = "00.00";

}

}

}

**private** **void** label1\_Click(**object** sender, EventArgs e)

{

}

//LOAD APPLICATION. POPULATE EMPLOYEE LISTVIEW.

**private** **void** Form1\_Load(**object** sender, EventArgs e)

{

//read employees.txt and populate the ListView with employees & info

//each line of employees.txt becomes an element of lines

**string**[] lines = System.IO.File.ReadAllLines(Globals.path + "\\data\\" + "employees.txt");

//each element is dealt with one at a time

**foreach** (**string** line **in** lines)

{

//create an array info. Each element in info is one peice of info from the line in employees.txt

**string**[] info = line.Split(',');

ListViewItem item = listView1.Items.Add(info[0]);

item.SubItems.Add(info[1]);

item.SubItems.Add(info[2]);

item.SubItems.Add(info[3]);

item.SubItems.Add(info[4]);

item.SubItems.Add(info[5]);

}

//check if ListView is populated, and if so, select the first item

**if**(listView1.Items.Count != 0)

{

listView1.Items[0].Selected = **true**;

}

}

**private** **void** label8\_Click(**object** sender, EventArgs e)

{

}

**private** **void** openFileDialog1\_FileOk(**object** sender, CancelEventArgs e)

{

}

//EDIT PICTURE

**private** **void** pictureBox1\_Click(**object** sender, EventArgs e)

{

//open a file dialog to chose an image and assign to the 5th subitem of the selected item

**if**(DialogResult.OK == openFileDialog1.ShowDialog())

{

//assign the image filepath to the 5th SubItem

listView1.SelectedItems[0].SubItems[5].Text = openFileDialog1.FileName;

//update the picture

pictureBox1.Image = Image.FromFile(listView1.SelectedItems[0].SubItems[5].Text);

}

}

**private** **void** listView1\_ItemActivate(**object** sender, EventArgs e)

{

}

**private** **void** listView1\_ItemActivate\_1(**object** sender, EventArgs e)

{

}

//ADD EMPLOYEE

**private** **void** addEmployeeToolStripMenuItem\_Click(**object** sender, EventArgs e)

{

//create a new item and its subitems

ListViewItem item = listView1.Items.Add("Employee");

item.SubItems.Add("Name");

item.SubItems.Add("Position");

item.SubItems.Add("Contact");

item.SubItems.Add("Employee Since");

//add a default image.

item.SubItems.Add(Globals.path +"\\resources\\"+ "gaming masters custom size.png");

//create a text file to store the tasks

//first, get the position of the item just created. This will be used to name the file so it can be associated with the employee later

**int** index = item.Index;

//convert that index to a string so it can be put in the filename

**string** fileName = index.ToString();

//finally, create the filepath & then create & then close the file.

**string** path = Globals.path + "\\data\\" + fileName + ".txt";

**var** myFile = File.Create(path);

myFile.Close();

}

//EDIT EMPLOYEE

**private** **void** button1\_Click(**object** sender, EventArgs e)

{

//check there is something to edit

**if** (listView1.Items.Count != 0)

{

//show the textboxes and make the other button visible

textBox1.Visible = !textBox1.Visible;

textBox2.Visible = !textBox2.Visible;

textBox3.Visible = !textBox3.Visible;

textBox4.Visible = !textBox4.Visible;

button1.Visible = !button1.Visible;

button2.Visible = !button2.Visible;

//check if an item is not selected, and if so, select the first item

**if** (listView1.SelectedItems.Count == 0)

{

listView1.Items[0].Selected = **true**;

}

}

//otherwise prompt the user to add an employee.

**else**

{

MessageBox.Show("You need to add an employee before you can edit one! Try Edit>Add Employee in the menu.");

}

}

//FINSIH EDITING EMPLOYEE (CLICK DONE)

**private** **void** button2\_Click(**object** sender, EventArgs e)

{

//find the selected item and update each of the subItems to the contents of the textboxes

**foreach** (ListViewItem item **in** listView1.SelectedItems)

{

item.SubItems[1].Text = textBox1.Text;

item.SubItems[2].Text = textBox2.Text;

item.SubItems[3].Text = textBox3.Text;

item.SubItems[4].Text = textBox4.Text;

}

//Update the labels

label8.Text = listView1.SelectedItems[0].SubItems[1].Text;

label9.Text = listView1.SelectedItems[0].SubItems[2].Text;

label10.Text = listView1.SelectedItems[0].SubItems[3].Text;

label11.Text = listView1.SelectedItems[0].SubItems[4].Text;

//hide the textboxes and make the other button visible

textBox1.Visible = !textBox1.Visible;

textBox2.Visible = !textBox2.Visible;

textBox3.Visible = !textBox3.Visible;

textBox4.Visible = !textBox4.Visible;

button1.Visible = !button1.Visible;

button2.Visible = !button2.Visible;

}

//REMOVE EMPLOYEE

**private** **void** removeEmployeeToolStripMenuItem\_Click(**object** sender, EventArgs e)

{

//find the selected employee

**foreach** (ListViewItem item **in** listView1.SelectedItems)

{

//delete their task file & re-order the other ones

**int** index = item.Index;

//convert that index to a string so it can be put in the filename

**string** fileName = index.ToString();

//finally, create the filepath & then create & then close the file.

**string** path = Globals.path + "\\data\\" + fileName + ".txt";

File.Delete(path);

**int** fileCount = (Directory.GetFiles(Globals.path + "\\data").Length) - 1;

**while**(index < fileCount)

{

//rename file (index+1).txt as (index).txt (move every file down by 1)

File.Move(Globals.path + "\\data\\" + (index+1) + ".txt", Globals.path + "\\data\\" + (index) + ".txt");

index += 1;

}

//remove the employee

listView1.Items.Remove(item);

}

}

//SAVE.

//Writes each item in the Listview to a new line, with the subitems also on that line.

**private** **void** homeToolStripMenuItem\_Click(**object** sender, EventArgs e)

{

StringBuilder sb;

//set apendExistingFile to false so the new save overwrites the old one.

**bool** appendExistingFile = **false**;

//tell it where to save to.

**using** (System.IO.StreamWriter sw = **new** System.IO.StreamWriter(Globals.path + "\\data\\" + "employees.txt", appendExistingFile))

//check that there is something actually in the listview.

**if** (listView1.Items.Count > 0)

{

//for each Item in the Listview, write it to file and then write each of its subitems, seperated by commas

**foreach** (ListViewItem lvi **in** listView1.Items)

{

sb = **new** StringBuilder();

**foreach** (ListViewItem.ListViewSubItem listViewSubItem **in** lvi.SubItems)

{

sb.Append(**string**.Format("{0},", listViewSubItem.Text));

}

sw.WriteLine(sb.ToString());

}

}

}

//ADD TASK

**private** **void** button3\_Click(**object** sender, EventArgs e)

{

//create a new task and its subitems

ListViewItem item = listView2.Items.Add("Task");

item.SubItems.Add("Set");

item.SubItems.Add("Due");

item.SubItems.Add("Value");

item.SubItems.Add("Status");

}

//UPDATE INFO WHEN A TASK IS SELECTED

**private** **void** listView2\_SelectedIndexChanged(**object** sender, EventArgs e)

{

//check that only one item is selected

**if** (listView2.SelectedItems.Count == 1)

{

//update the labels to match their subItem text

label12.Text = listView2.SelectedItems[0].SubItems[1].Text;

label13.Text = listView2.SelectedItems[0].SubItems[2].Text;

label14.Text = listView2.SelectedItems[0].SubItems[3].Text;

label15.Text = listView2.SelectedItems[0].SubItems[4].Text;

}

}

//EDIT TASK

**private** **void** button4\_Click(**object** sender, EventArgs e)

{

**if** (listView1.Items.Count != 0 && listView2.Items.Count != 0)

{

//when the 'Edit' button for tasks is clicked, show the textboxes and make the 'Done' button visible

maskedTextBox2.Visible = !maskedTextBox2.Visible;

maskedTextBox3.Visible = !maskedTextBox3.Visible;

maskedTextBox1.Visible = !maskedTextBox1.Visible;

comboBox1.Visible = !comboBox1.Visible;

button4.Visible = !button4.Visible;

button5.Visible = !button5.Visible;

//check if an item is not selected, and if so, select the first item

**if** (listView2.SelectedItems.Count == 0)

{

listView2.Items[0].Selected = **true**;

}

}

**else**

{

MessageBox.Show("You need to add a task before you can edit one! Try clicking 'Add Task'.");

}

}

//FINSIH EDITING TASK

**private** **void** button5\_Click(**object** sender, EventArgs e)

{

{

//find the selected item and update each of the subItems to the contents of the textboxes

**foreach** (ListViewItem item **in** listView2.SelectedItems)

{

item.SubItems[1].Text = maskedTextBox2.Text;

item.SubItems[2].Text = maskedTextBox3.Text;

//if a task has been marked as completed without a value supply a default one.

//this prevents the pay calculator from complaining about incorrectly formatted strings when converting to a float.

**if** (comboBox1.Text == "Yes" && maskedTextBox1.Text == " .")

{

item.SubItems[3].Text = "00.00";

}

**else**

{

item.SubItems[3].Text = maskedTextBox1.Text;

}

item.SubItems[4].Text = comboBox1.Text;

}

//Update the labels

label12.Text = listView2.SelectedItems[0].SubItems[1].Text;

label13.Text = listView2.SelectedItems[0].SubItems[2].Text;

label14.Text = listView2.SelectedItems[0].SubItems[3].Text;

label15.Text = listView2.SelectedItems[0].SubItems[4].Text;

//hide the textboxes and make the other button visible

maskedTextBox2.Visible = !maskedTextBox2.Visible;

maskedTextBox3.Visible = !maskedTextBox3.Visible;

maskedTextBox1.Visible = !maskedTextBox1.Visible;

comboBox1.Visible = !comboBox1.Visible;

button4.Visible = !button4.Visible;

button5.Visible = !button5.Visible;

//now to save the tasks so they aren't lost when another employee is selected.

//first, select the last selected item in ListView1

listView1.Items[Globals.selectedItem].Selected = **true**;

listView1.Select();

//now to write all the tasks in listView2 to the selectedItem'th file in the data folder

//that is, write the tasks to the file associated with the employee

{

StringBuilder sb;

//set apendExistingFile to false so the new save overwrites the old one.

**bool** appendExistingFile = **false**;

//tell it where to save to.

**int** filename = listView1.SelectedIndices[0];

**string** name = filename.ToString();

**string** filepath = Globals.path + "\\data\\" + name + ".txt";

**using** (System.IO.StreamWriter sw = **new** System.IO.StreamWriter(filepath, appendExistingFile))

//check that there is something actually in the listview.

**if** (listView2.Items.Count > 0)

{

//for each Item in the Lsitview, write it to file and then write each of its subitems, seperated by commas

**foreach** (ListViewItem lvi **in** listView2.Items)

{

sb = **new** StringBuilder();

**foreach** (ListViewItem.ListViewSubItem listViewSubItem **in** lvi.SubItems)

{

sb.Append(**string**.Format("{0},", listViewSubItem.Text));

}

sw.WriteLine(sb.ToString());

}

}

}

}

}

//REMOVE TASK

**private** **void** button6\_Click(**object** sender, EventArgs e)

{

//find the selected task

**foreach** (ListViewItem Item **in** listView2.SelectedItems)

{

//remove it

listView2.Items.Remove(Item);

}

}

//OPEN HELP

**private** **void** helpToolStripMenuItem\_Click(**object** sender, EventArgs e)

{

//open a HTML file with the default browser to give help

System.Diagnostics.Process.Start(Globals.path + "\\resources\\" + "help.HTML");

}

}

**Testing**

This sample of tests is primarily focused on user interaction rather than the underlying processes and algorithms that respond to this input, as it is the user giving unexpected input that is the most common source of errors and crashes in a program.

|  |  |  |  |
| --- | --- | --- | --- |
| What’s being tested | Input/steps taken | Expected output | Actual output |
| An employee can be added | Click ‘Add employee’ in the Edit menu’ | An employee will appear in the list with the default details |  |
| An employee can be removed | Select an employee and click ‘Remove employee’ | The employee will disappear from the list |  |
| If there are no employees, clicking ‘remove’ will not crash the program | Remove all employees, and then click ‘remove employee again’ | None | None |
| If an employee is not selected, clicking ‘remove’ won’t crash the program. | Deselect an employee and click ‘remove employee’ | None | None |
| An employee’s details can be edited | Select an employee and click ‘Edit’, add some details and then click ‘done’. | The Edit button will be hidden, four text boxes will appear over the details, and the ‘done button will appear. |  |
| If the user edits an employee when none are selected, the first one will be selected by default. | Deselect an employee and click ‘Edit’. | The edit will be applied to the first employee in the list | The changes applied to the first employee. |
| When a different employee is selected, the details and tasks update | Select an employee. | The details will update to the corresponding employee | Selecting a different employee causes the details to update |
| If the user attempts to edit an employee when none exist, they will be notified | Click ‘Edit’ without adding an employee first. | A message box will pop up notifying the user they must add an employee first. |  |
| A task can be assigned to an employee | Select an employee and click ‘Add task’ | A task with default details will appear in the task list. |  |
| If the user attempts to assign a task to an employee when none exist, they will be notified | Click ‘Add Task’ without adding any employees. | A message box will pop up notifying the user they must add an employee first. | The program allowed the task to be added. |
| **(This test is a result of unexpected behaviour in last one)**  Editing a task not assigned to an employee (i.e a task added when there are no employees) will result in a popup asking the user to add an employee. | Add and edit a task without adding any employees first. | A message box will pop up informing the user they need to add a task. | The message is generic, but the code does check if there are any employees added before allowing the user to edit a task. |
| If the user attempts to edit a task when none exist, they will be notified | Click ‘Edit’ without adding a task | A message box will pop up notifying the user they must add a task first. |  |
| Tasks can be edited | Select a task and click ‘Edit’. | The Edit button will be hidden, three text boxes and a dropdown menu will appear over the details, and the ‘done’ button will appear. |  |
| When a different task is selected the details will be updated. | Select a task. | The details will update to the corresponding task |  |
| Tasks can be removed. | Select a task and click ‘remove task’ | The task will disappear from the task list |  |
| When no tasks exist, clicking ‘remove task will not crash the program. | Click ‘remove task’ without adding one first. | none | None |
| When a user is removed, their file is deleted and the remaining files are correctly renamed so tasks stay assigned to the correct employee. | Remove one of multiple employees. | The tasks will correspond to the correct employees when another employee is selected. | These screenshots show that the first employee has one task and the second has two. Below, when the second one is removed, the new second employee (formerly the third) has three and does not inherit the deleted employees tasks. |
| The program calculates an employee’s due pay correctly | Add three tasks. One is worth 25.50 and marked as completed, one is worth 12.45 and is marked as not completed, one is worth 15.00 and is marked as completed | A total of 40.50 under the due pay section of the employee details |  |

**Evaluation**

My application fulfils all of its specified functions, which were to keep a record of employees and their tasks and calculate their pay.

It does not have one of the features the client requested – import and export – although it does have a save feature.

I deemed the save feature to be a function rather than a feature, as one of the goals of the application was to be an improvement over a spreadsheet, and without a save function this wouldn’t be possible, regardless of how many other features it had.

I thought that the import feature was unnecessary, as this is a bespoke application to be used by one person. This means the feature would only ever be used once to import a very small data set, and therefore would not be worth the time it would take to implement. It would be quicker and easier for the client to simply enter the data manually, which he did without any trouble.

I also did not add an export feature, due to time constraints and the complexity of implementing it. However, data can be shared between instances of the application by manually copying the data folder and putting it in the folder of another instance of the application, a happy side-effect of the way the data is stored.  
I made the decision to store each of the employees on separate lines in a text file, with values comma-separated, and have a different file for each employee’s tasks. This was partially due the fact it is easier to access text files than XML or a database, and as the dataset is very small, having a more rigid storage solution was not necessary.  
This means that it is relatively easy to transfer all the saved data between different versions of the program, as my client tried a few different updated versions while I was testing it.

Another unrequested feature I added was the option to assign a picture to an employee, to make it clear which one is selected at a glance. The picture can be changed by clicking on it, which my client liked because it was very easy to use.

The UI is very plain, but the layout is primarily for making it easy to see information, which it does well.  
The controls are clumsy, with employees being added via the menu, but editing and adding tasks being done by buttons – also the employee/task must be selected before editing/assigning a task, which may confuse the average user.

However, as this a bespoke application just for my client, I simply told him to use the ‘help’ feature I included, which opens a HTML document in the default browser, fully documenting how to use the program, and after reading the document my client said the program was simple to use.

I made quite a few changes to the program and its features throughout the prototyping and coding process, the developer equivalent of ‘project creep’. Some of this was to optimise the development process (i.e remove features that would cost more to implement than they would benefit the user), and some was simply that the client and myself agreed that a particular feature was not needed. However, at each time I did this I contacted my client and informed of what I intended to do, and he agreed that my changes were acceptable.

Once the program was complete, I sent it to the user to try out, and he spotted some typos in user prompt dialogues, which I corrected immediately. Other than that, the client confirmed that the program met all of his requirements and he was satisfied with it.

The code is not particularly tidy, efficient or well written, as I was primarily concerned with making it functional first. Rather than making procedures for near-identical functions I copied and pasted code between different parts of the program. I also re-used variable names between similar algorithms, such as the ones for generating file paths to the task data. This will make the code difficult to maintain or edit, as it is not clear they are the same variables. Some variables also have very similar names, and it is not always clear exactly what they are for, particularly those in loops and the names of WinForm elements. I have commented the code extensively to make it clear to myself and anyone else who may use it to make it clear what each part does, which somewhat negates the problem of the untidiness and bad variable naming.

### Did I meet my S.M.A.R.T objectives?

This table shows the S.M.A.R.T objectives I laid out in the analysis, and whether I have completed them.

| **Goal** | **Met?** |
| --- | --- |
| User can add employees | Y |
| User can edit employees | Y |
| Program correctly adds pay according to complete tasks | Y |
| Program saves data | Y |
| Program retrieves data correctly | Y |

### Future development To improve and expand on this program, I would first tidy up the code. This would include making procedures for things such as generating file paths and updating labels, and giving the variables clearer names, particularly the WinForm elements. This will make it much easier for myself or another developer to improve the program at a later date I would also implement an export feature, and perhaps an import feature if my client still wanted it, or if the datasets became much larger.

**Appendix**

### 1.1: Help feature This is the help feature the user can reference for help with utilising the program;

### 

This is the code that generates the above page;

    <!DOCTYPE html>

    <html>

      <head>

        <title>Page Title</title>

      </head>

      <body>

        <h1>Help</h1>

        <h2>Contents</h2>

        <p><a href="#1">Adding and removing employees</a></p>

        <p><a href="#2">Editing employee information</a></p>

        <p><a href="#3">Adding & removing tasks</a></p>

        <p><a href="#4">Editing tasks</a></p>

        <p><a href="#5">Due pay</a></p>

        <h3 id = 1>Adding & removing employees</h3>

        <p>To add an employee, click on the Edit menu at the top of the application, and then click 'Add Employee'.</p>

        <img src="..\resources\addEmployee.JPG" alt="Add Employee">

        <p>To remove an employee, select the employee you would like to remove, and click 'Remove Employee' in the Edit menu.</p>

        <h3 id = 2>Editing employees</h3>

        <p>To edit an employees information, select an employee, and then click on it again to edit the name.<br>To edit the information, select an employee and then click the edit button undernaeth their details. Enter the details into the text boxes, and then click 'done'.<p>

        <img src="..\resources\editEmployee.JPG" alt="Add Employee">

        <p>The employee being edited can be changed at anytime by selecting a different employee, and only the selected employee will have the details updated when you click 'Done'.<br>To add a picture, click on the picture to choose a new. .JPG, .PNG and .GIF files can be used. Images will automatically be resized to 100\*100 pixels.<br>To save the changes, click on the 'File' menu and click 'Save'</p>

        <h3 id = 3>Adding & removing tasks</h3>

        <p>Employees can be assigned tasks on the right-hand side of the application.<br>First, select the employee to have the task assigned to. Then, Click 'Add Task'.<br>To remove a task, select the task and then click 'Remove Task'.</p>

        <h3 id = 4>Editing tasks</h3>

        <p>To edit a task, select a task, and then click it again to enter the details.<br>To edit the other details, select a task, and then click 'Edit' underneath the task details. Enter the information into the textboxes and click 'Done' to save the changes.</p>

        <img src="..\resources\editTask.JPG" alt="Add Employee">

        <p>The task being edited can be changed at any time by selecting a different task before clicking 'Done'.</p>

        <h3 id = 5>Due Pay</h3>

        <p>Due pay is calculated by adding the value of all tasks which have been completed.<br>If the due pay does not update after marking a task as complete, try selecting a different employee and the re-selecting the first one.</p>

      </body>

    </html>

### 1.2: Relative file paths A relative file path is used when there is no access to higher directories, only lower ones. Most file paths start at the root directory, e.g, C:/ on windows or / on Linux and Unix. As my program is designed to be run on any windows computer in any location, it cannot use the install location to get the file paths of the program resources, such as the help document.