HANGMAN

PROJECT DEVELOPMENT REPORT

The 21st century is an advancing era, and mainstream games which were once played on paper have begun to require digitisation in order to remain relevant to the screens and capabilities of today. The Hangman Project Development Report endeavours to document the gamification process of the age-old ‘hangman’, imperative to improving spelling and literacy skills in young audiences, continuing its educational benefits in modern times. The project follows the structured approach of software design and development to support the proposed application solution while utilising a distinct ‘dark’ modern aesthetic. To effectively implement the structured approach, appropriate design tools, maintenance, evaluation and programming constructs have been employed throughout the project.

AUTHOR/TEAM MEMBERS: Jade Harris 11SDD

MENTOR/TEACHER: Adam Leserve

CONTENTS

REQUIREMENTS REPORT

AIMS AND OBJECTIVES | REQUIREMENTS REPORT

Purpose

This project endeavours to demonstrate competency and understanding of the C# programming language through the documentation and development of a working graphical user-interface game of Hangman. Through the utilisation of coding constructs, graphical interface design, documentation and the structured programming approach, a Windows Form application solution will be produced.

Functionality

The basic functionality of the solution uses only the mouse to emulate the gameplay of traditional Hangman, continuing to improve spelling and literacy skills in its users. Much like the classic version, an educational experience is delivered by allowing the player to attempt to guess letters in order to reveal a randomly selected word. If a guess is contained in the hidden word, the letter will appear in its correct position. If the guess is incorrect, a body part of the hangman is drawn. The player has 11 guesses and if they successfully reveal the word before the hangman has been completed, the user has won and should be congratulated. However, if the hangman is complete before the word is guessed, then the player has run out of attempts and lost, a condolence message offered. At any time, the player should be able to start a new game, however, this feature must be protected from accidental use with an appropriate mechanism.

Interface Usability

As a graphical user-interface solution, the program provides an engaging and intuitive user experience which supports the functionality necessary for its gameplay. The proposed software solution is a Windows Form application which requires only the mouse as input hardware, alongside a computer with a Windows-based operating system and a monitor. The interface begins with a title screen that has a start button, establishing a positive first impression. Once the start button has been pressed, a brief loading screen captures the player's attention and creates a seamless flow into the gameplay. The gameplay interface then allows the user to select a letter from an alphabetical on-screen keyboard made from the class Button. Their chosen letter is entered into an instance of TextBox which contains the current letter that will be guessed. However, the player must click on a separate 'submit' instance of Button class to actually guess whatever letter is in the TextBox. A red delete object of class Button is also featured in the on-screen keyboard to allow the user to remove any guess. If they wish to choose a different letter to what is currently in the textbox (as whatever is contained only becomes a guess once the submit button is clicked), the user can simply click on a different letter on the keyboard and it will replace their current guess. These submit and delete buttons prevent a letter from accidentally being selected, and declutter the interface, greatly enhancing user-experience. Once a guess is made, another 'loading screen' appears, **to give a lifelike property to the AI through an illusion that the computer is 'thinking'**. This increases suspense and provides a seamless transition between interfaces. If the guessed letter is not in the randomly selected word, an animated drawing of the hangman body part appears in the interface to indicate that their guess was incorrect. This is an engaging way to display the hangman and a 'next' button appears with the animation to allow the player to make their next guess (when they have sufficiently watched the .gif). When the user returns to the interface, a static image of the hangman with the body part just drawn is featured. The player has 11 guesses until the hangman has been completed. Alternatively, if the player makes a correct guess, then the letter appears in its correct position in the hidden word. Whether the guess was correct or incorrect, the interface then disables the button for that letter so that the user is aware of and can no longer select letters which have previously been guessed. This design choice reduces the interface clutter of a separate section to display already-guessed letters. To further enhance usability, a button which allows a new game to be launched at any time is featured in the on-screen keyboard but protected from accidental use by a confirmation message. The interface also features a distinct form icon, so that the application can be identified in the toolbar when the game is minimised. The simplistic, intuitive aesthetic promotes ease of use and enhances user experience.

Security

Overall, the Hangman application contains minimal security issues as it does not involve the storing of personal data or expose any vulnerabilities of a computer system. In particular, the information stored in the provided external file is not sensitive and is publicly available, therefore there are no security risks involved. Additionally, by using the File.ReadAllLines method, the stream is automatically opened and closed. However, appropriate caution should still be taken regarding the text file - say if the user chose to alter the word list file to include their personal passwords. Additionally, if the capabilities of the application were extended, for instance implementing a scoreboard which enabled users to enter their name, further security issues could arise. If the word list file or another file was to store such personally identifiable or sensitive information, the sequential access method should be replaced with a random access alternative such as a database. Further, cryptology could increase security and reduce likelihood of interception by unauthorized users.

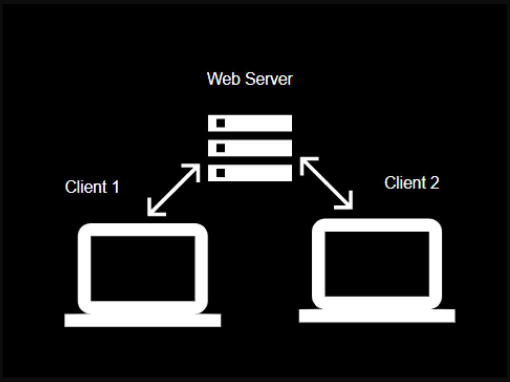
Portability

Windows Form applications have limited portability, functioning solely on Windows desktops and laptops as they require a Windows-based operating system. However, the versatility of traditional Hangman gameplay lends itself to being extended in the future to work in browser or mobile. In terms of the implementation itself, as Windows Form applications are not compatible with Android, MacOS and iOS, and a web browser doesn't directly interpret C#, broadening platform compatibility would either require complete recoding or use of a different framework. This could involve extending the existing codebase with a C# web framework such as ASP.NET, allowing use in a web browser which broadens access scope to Mac, Linux and Windows. For mobile and Android platforms, the .NET framework Xamarin could be leveraged to great effect. As the C# codebase already exists, effort can be significantly minimised by utilising a framework which already implements C#. Overall, this application has rich potential for portability.

Networkability

The developed Hangman solution does not contain any network operability as it is an application designed to execute on the user's independent device and its current gameplay is single-player only. However, given the mechanics of traditional hangman, the application's functionality has potential to be adjusted for networkability by introducing multiplayer. Perhaps it could emulate the traditional gameplay of Hangman where one player selects a word which the other guesses, or rather a more contemporary style as a race to guess a random word. This would require the use of a web framework as discussed in the previous 'portability' section such as ASP.Net. The use of a web server which communicates with the players' client machines would enable the application's output to display on each monitor while obfuscating the code which could allow cheating.

This client-server topology can be viewed to the right:



OVERVIEW OF SYSTEM | REQUIREMENTS REPORT

Data and Information

The nature of the data used and produced by Hangman is centered around the functionality of its gameplay, involving user input for the letters and program output for the monitor display. The application is designed to receive input from the user's mouse-click to trigger click-events on buttons. These events then output information on the monitor, displaying the letter which the user has selected in the textbox. Once the program has received a mouse input on the submit button, if the letter was contained in the word, then the system will display the letter in its correct position. If the guess was incorrect, the application will produce information for the graphical picture box. This picture box is influenced by the data of how many guesses have been made. The computer also stores data of the player's state through the use of global variables – the number of guesses remaining and whether the user has won or lost. Through continued mouse inputs, if the player has revealed the entire word then the program will display a congratulatory system MessageBox. Whereas if the player has failed to guess the correct word after 11 letters are selected, a condolence message will be displayed. A message box is also produced if a click-event is triggered on the reset button.

Software Structure

The structure of the proposed software solution is a Windows Form application which utilises an external text file in conjunction with the use of programming constructs, data structures and graphical interface instances. The program uses classes to build the interface, an instance of class

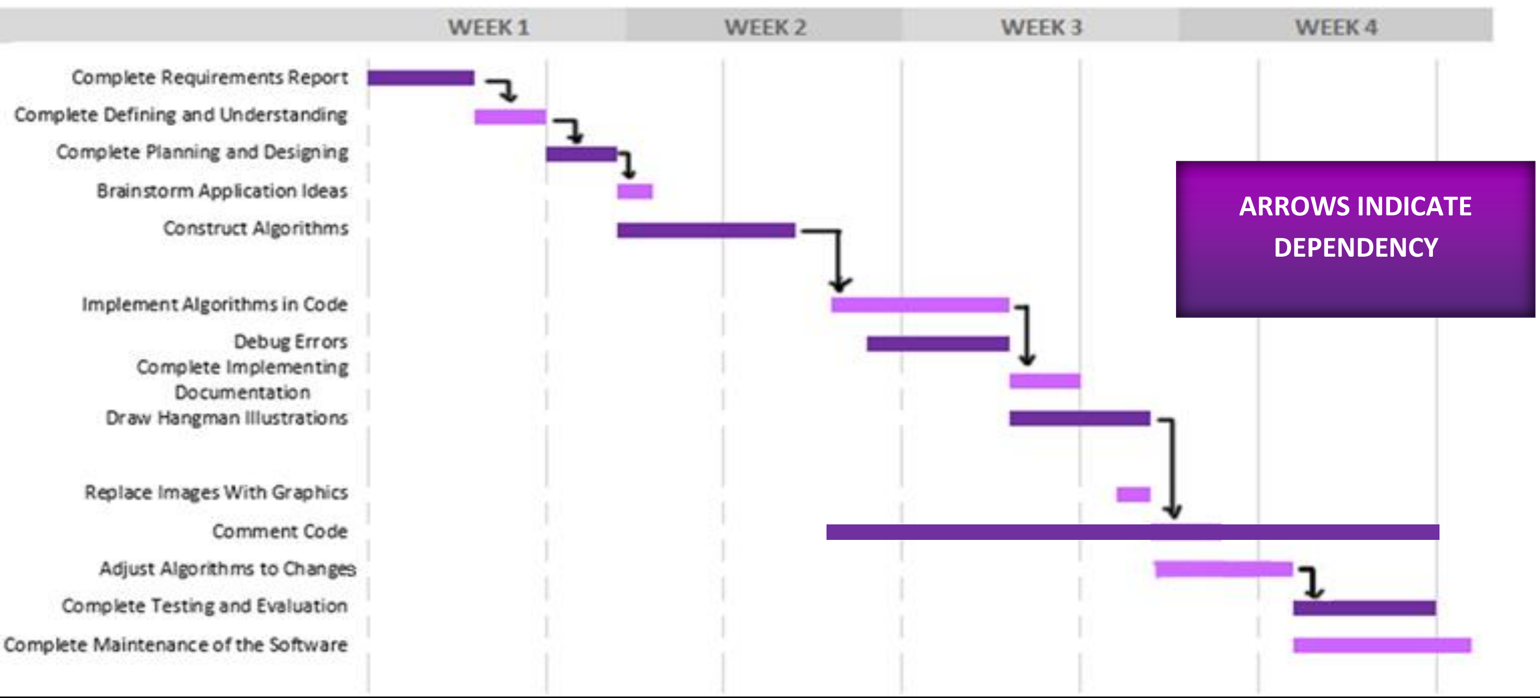
Button for each letter on the keyboard, TextBox to display the hidden word and the user's current guess, and PictureBox to display the hangman. These are combined with the use of an external text file and the System.IO namespace to store the list of possible words. Numerous of these list and array data structures are included throughout. Additionally, the System.Threading namespace creates the pause in the 'computer thinking' effect with its Thread.Sleep() method. This software structure creates an aesthetic and intuitive working Hangman application with an engaging and distinct user experience.

THE BODY OF PROJECT DEVELOPMENT REPORT (PDR)

DEFINING AND UNDERSTANDING THE PROBLEM | PDR BODY

Timing and Sequencing of Project

The below Gantt chart illustrates the time frame of the project and clearly define application milestones.



Identification of the problem

Identifying the problem involves articulating the problem that the application must solve. Typically, the problem would be identified through interviews with the stakeholders. However, a thorough description of the problem has been provided which **can be succinctly articulated into the following points:**

→ Produce a working graphical user-interface game of Hangman, or alternatively Snowman, designed to run in a Windows form application
→ Randomly select a word from a list of words residing in an external text file which the player must guess. This should be hidden.
→ If the user guesses an incorrect letter, each body part is to appear or disappear
→ If the user makes a correct guess, the letter/s must appear in their location in the chosen word
→ Already used letters must be displayed on the screen
→ If a player successfully solves the word, the game will provide a congratulatory message
→ If a player runs out of guesses and the final body part appears or disappears (if Snowman was selected) a condolence message will appear
→ A button should be accessible that allows a new game to be played at any time but must be protected from accidental use
→ The project must follow the structured approach and its appropriate documentation in a PDR

Ideas generation

While there were obviously numerous solutions to elements of the entire problem, for instance how to choose a random word, these were categorised into **three primary solutions which could have been pursued:**

SNOWMAN

This solution entailed the choice of the less confronting game of 'Snowman'.

❖ To create a random word, rather than read from a populated sequential file, this option would extract the HTML data from a webpage and read the lines as separate items in an array. This would provide a larger word pool.
❖ A randomly generated number would become the index for a word in the array and then that word would become the hidden word.
❖ This word would be concealed under numerous underscores as placeholders for each letter.
❖ The player could make a guess by typing any letter on their hardware keyboard and providing the letter hadn't been guessed, it would automatically be submitted.
❖ An interesting idea for the Snowman melting was to use multiple picture boxes with animated .gif's. A switch case would allow each to become visible and play when the appropriate part disappeared.
❖ The congratulatory message would be a large gif with a puddle (snowman melted) and the words 'YOU LOSE'. If the player won, a .gif with 'YOU WIN' would be displayed.
❖ A possible solution to the restart button was a button where the first click would activate a timer and the buttons opacity would gradually decrease. If the player double clicked the button before 30 seconds, then that would be interpreted as a confirmation and the program would reset.

HANGMAN 'ORGANIC'

This solution was the most plausible and organic solution to the problem, balancing traditional gameplay with Windows Form capabilities.

❖ A sequential file would be provided with the application, filled with 200 words separated by a comma. The line is read into the program and split at each comma into an array.
❖ The random class would select a random integer that would solve the problem of randomly selecting a word.
❖ A for loop would produce a string which replaces the letters with '_' for the placeholders. For aesthetics, a separate function would then add a space between each placeholder underscore before the hidden word is displayed.
❖ An on-screen keyboard. The player can make a guess by selecting a letter and to prevent accidental guesses, they can select a different letter which will replace their guess. This letter can be changed but once they click the submit button, the current letter in the textbox will become their guess and checked against the hidden word.
❖ A solution to display the used letters was to disable the button for that letter.
❖ If the letter matches a letter in the word, the placeholder underscore at that point will be replaced by the letter the user guessed by splitting the target word into a character array and replacing the item in the index.
❖ To display a body part if an incorrect guess is made, a switch case will make a separate picture box visible with the new body part added.
❖ A possible solution for protecting the reset button was an extremely user-friendly confirmation message box.

HANGMAN 'POSTMODERN'

This solution closely resembles the 'organic' hangman version, however, features arguably more distinct solutions to the problem.

❖ Using labels instead of underscores to represent the placeholders
❖ Instead of making separate picture boxes visible depending on the number of guesses left, the program could simply change the contents of a single image box accordingly using Properties Resources
❖ Headset and audio involvement to create an engaging and immersive experience. For instance, when a letter is selected the program makes a beeping sound. Additionally, C# provides the ability for an audio file to be included in the directly which can be played on command. This could be used to emphasize the players loss or congratulate the player's victory.
❖ A possible solution for protecting the reset button was a unique idea where the player could type the word reset with the on-screen keyboard and the feature would trigger.
❖ To replace the placeholder with the correctly guessed letter, a more readable .Insert() and .Remove() method could be used instead.

In the end, no one solution was selected, instead a combination of all three categories contributed to the 'balanced' final product. The cohesive benefits of each category were extracted and implemented in the final application. **The evaluation of the advantages and disadvantages of each solution which formed the final product can be seen below:**

SNOWMAN

ADVANTAGES (USED IN SOLUTION)

SNOWMAN

+ Selecting the random word by generating a random number and using it as an index was a clean and effective solution

+ Underscores as placeholders resemble the traditional hangman and are a universal indicator of the game

DISADVANTAGES

- Extracting HTML from dictionary website was a unique idea, however, poses many security and legal issues

- Animated gif idea was definitely interesting and unique. However, the graphic of a snowman melting was overly ambitious.

- Considering the project's scope, using timers and manipulating opacity in Windows Form applications to protect the restart button is ambiguous and convoluted for the user

'ORGANIC' HANGMAN

ADVANTAGES (USED IN SOLUTION)

SNOWMAN

+ Reading from a sequential file was the initial requirement provided, and although the client stated either method could work (reading site HTML), for legal reasons an external text file is safer. Also, 200 words is plenty enough for the program.

+ A for loop specifically can be used to expand on the previous idea of generating underscores as the placeholders in 'Snowman'

+ An on-screen keyboard provides substance to the application interface, creating a more immersive user-experience. It also solves the other problem of allowing the user to see which letters they have already chosen. The keyboard button could simply be disabled.

+ Switch case to determine which picture the picture box is changed to is a clean and efficient ideas

+ Reset button protection simply being a pop-up dialogue box is extremely user-friendly

+ Function which would add spaces between each placeholder underscore

DISADVANTAGES

- As a method to reveal the letter, breaking the placeholder into an array of chars and then changing the character at the index where the two characters match is overly complicated – even to explain.

SNOWMAN

ADVANTAGES (USED IN SOLUTION)

SNOWMAN

+ Use of the .Insert() and .Remove() methods are an extremely simple and clean alternative to reveal the correct letters

+ By leveraging the Properties.Resources method to change the contents of a single image box, the application designer can remain a lot neater (rather than having layered image boxes). This allows managing errors to be a lot simpler. Additionally, changing which images is in a textbox is a lot less tedious than setting the visibility of multiple.

DISADVANTAGES

- Using labels seemed an obvious and intuitive solution to the hidden word problem. However, the limited style options of labels weren't entirely suited to an aesthetic and consistent user-experience. This includes the restricted font size of the text and the inability to centre the placeholders.

- While a unique idea to make the game more immersive was to include audio with Beeps, this reduces the inclusivity of the application. The program's full experience is limited for low-income households who may not be able to afford a headset. It also could cause ear pains to player's with sensitive hearing.

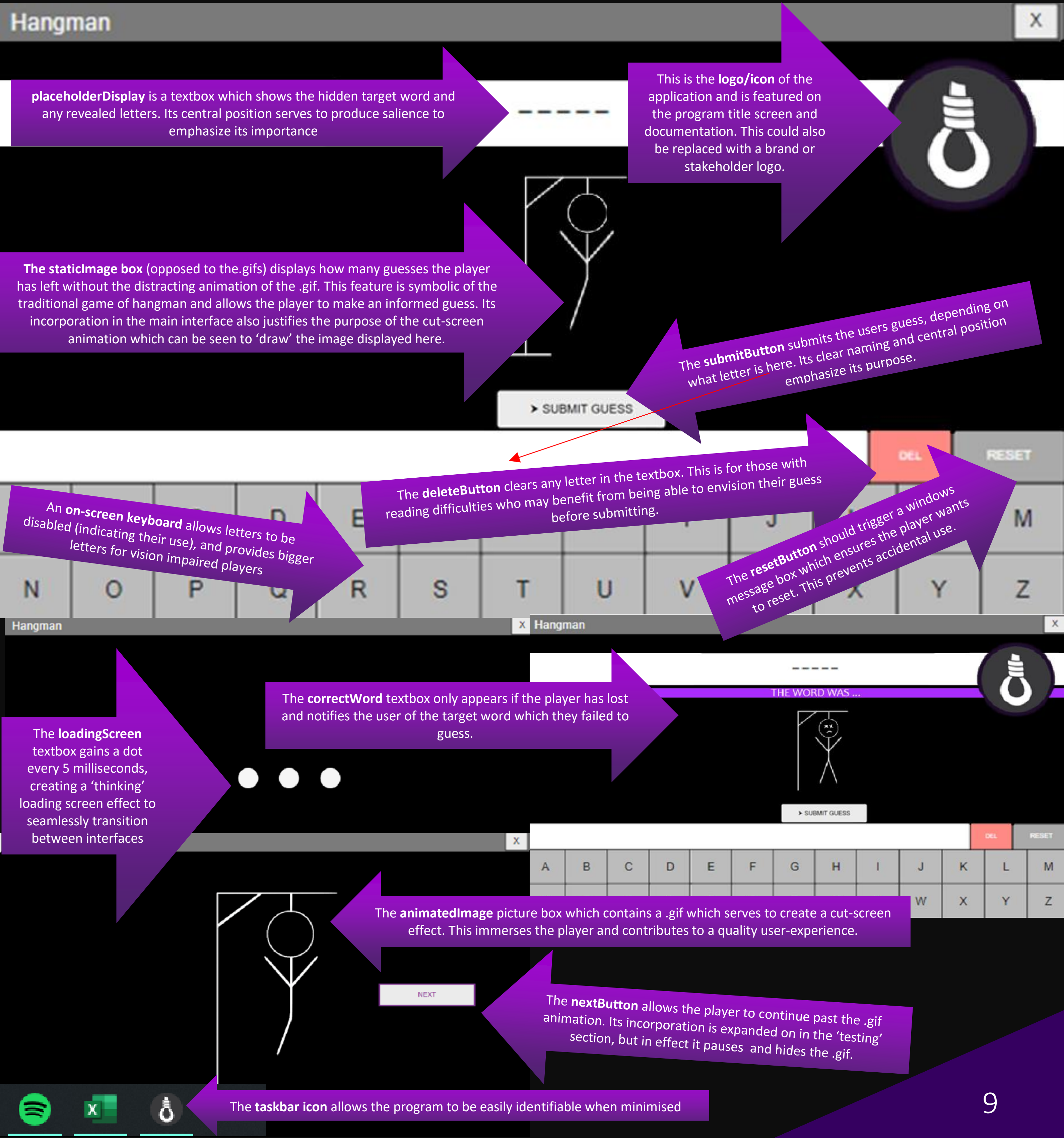
- The idea of implementing the restart command into the textbox is undoubtably unique, but impractical especially if RESTART is the target word.

The final idea is comprised of the advantages from these three main ideas. While originally a single category was going to be chosen from the three, this decision to incorporate the advantages of each was preferred as it enabled the program to have the benefits of three different perspectives.

Interface design

This section presents ideas for the chosen user-interface design to improve understanding of the user-experience. While the application only uses one form, **the Hangman application is comprised of the central 'gameplay' interface, a 'loading screen' interface, a 'cut-screen' interface and a 'loss' interface:**

The illusion of these separate interfaces is achieved by hiding specific elements - as detailed in the 'implementing' section - and creates a distinct and immersive user experience.



placeholderDisplay is a textbox which shows the hidden target word and any revealed letters. Its central position serves to produce salience to emphasize its importance

This is the **logo/icon** of the application and is featured on the program title screen and documentation. This could also be replaced with a brand or stakeholder logo.

The staticImage box (opposed to the.gifs) displays how many guesses the player has left without the distracting animation of the .gif. This feature is symbolic of the traditional game of hangman and allows the player to make an informed guess. Its incorporation in the main interface also justifies the purpose of the cut-screen animation which can be seen to 'draw' the image displayed here.

The **submitButton** submits the users guess, depending on what letter is here. Its clear naming and central position emphasize its purpose.

The **deleteButton** clears any letter in the textbox. This is for those with reading difficulties who may benefit from being able to envision their guess before submitting.

An **on-screen keyboard** allows letters to be disabled (indicating their use), and provides bigger letters for vision impaired players

The **resetButton** should trigger a windows message box which ensures the player wants to reset. This prevents accidental use.

The **loadingScreen** textbox gains a dot every 5 milliseconds, creating a 'thinking' loading screen effect to seamlessly transition between interfaces

The **correctWord** textbox only appears if the player has lost and notifies the user of the target word which they failed to guess.

The **animatedImage** picture box which contains a .gif which serves to create a cut-screen effect. This immerses the player and contributes to a quality user-experience.

The **nextButton** allows the player to continue past the .gif animation. Its incorporation is expanded on in the 'testing' section, but in effect it pauses and hides the .gif.

The **taskbar icon** allows the program to be easily identifiable when minimised

Communication with others involved in the proposed system

While application development should undeniably utilise regular stakeholder feedback and external communication, due to the conditions of this project as an assignment and the thorough description of the problem provided, completion was primarily independent. However, allocated sessions of work in class enabled the project mentor/client to constantly remain in the process and provide valuable suggestions. Any design queries and functionality questions were immediately raised to ensure the product was as close to the envisioned program as possible, and adjustments were made accordingly. By aligning application development and testing with these crucial feedback sessions, the client remained involved throughout the process and their feedback could shape the algorithms and software solution.

Although the project was completed solo, occasional collaboration was also conducted with surrounding classmate developers to gain a different perspective from their solutions to the problem. It also enabled a more effective and evaluated program to be reached. Further, a brief questionnaire was conducted with my family to select the most aesthetic and intuitive user-interface and colours – where black and purple were seen as the most striking. As well, some of the errors faced during development (detailed in the ‘implementing errors’ section) required communication by reading forums by the broader coding community to find an appropriate solution. Specifically, this was leveraged in the .gif implementation.

Overall, as the client will communicate with the developer in the classroom beyond this assignment, development and refinement of the application can continue.

Consideration of social and ethical issues

Overall, the proposed application has reasonably minimal social and ethical impact in relation to intellectual property, ethical consideration and inclusivity. Each Hangman illustration and the application interface was hand-drawn in Adobe Photoshop. This avoids any copyright breaches as Adobe products have licencing for commercial use. However, the .ico file for the application icon, while sourced from a site which enabled its commercial use, appeared to be featured on another site as well which contained the same icon except under a personal use licence. This poses a unique ethical and legal consideration. However, I believe the source which enabled commercial use was the original source.

The social impact of the application involves more considerations. Design choices which considered inclusivity was crucial. The on-screen keyboard enabled the letters of the keys to be bigger. Such feature could be extended by providing an option to decrease and increase font size. This allows physically disadvantaged players to select the letters more easily as well as visually impaired users to see the letters easier. Additionally, the gameplay mechanism to submit the guess independently to choosing a letter broadens usability for users who may struggle with reading, by enabling them to envision the letter. As mentioned, it also reduces pressure for players who may struggle with dexterity.

However, considering the scope of the project, adjustments for all backgrounds were not viable to be implemented. The project does undoubtably have potential for these improvements though. For a visually impaired user, the game could integrate audio files of the letter chosen and potentially dictate the placeholders if a letter is revealed or which body part was drawn if the guess was incorrect. The code and computer system could be adjusted for different inputs if a braille keyboard was used. In terms of cultural diversity, in the application's current state the word file could be adjusted to other languages. However, changing the external text file is not obvious, and the current interface with its reset and delete buttons would continue to remain in English. These do have potential to be translated. By integrating an option to choose a language at the beginning of the Hangman, cultural diversity could easily be promoted. The application does contain minor instances of secondary notation specific to Western cultures, such as red for delete, which is less inclusive to differing cultural backgrounds. However, in ratio comparison the application has already intentionally minimised use of symbolic colours and symbols to reduce exclusivity and further improvements are viable and reasonably minor.

Another social and ethical impact is the premise of hangman, potentially considered inappropriate for younger users. The graphics may be seen to promote violence by viewing a figure being hung and dying, particularly in a country where capital punishment is illegal. If the project had a broader scope, this could be overcome by changing the graphical animations to a Snowman melting and providing a parental option to unlock 'hangman mode'. Additionally, for younger uses a difficulty setting could be implemented which could change the length of words chosen and increase the number of guesses.

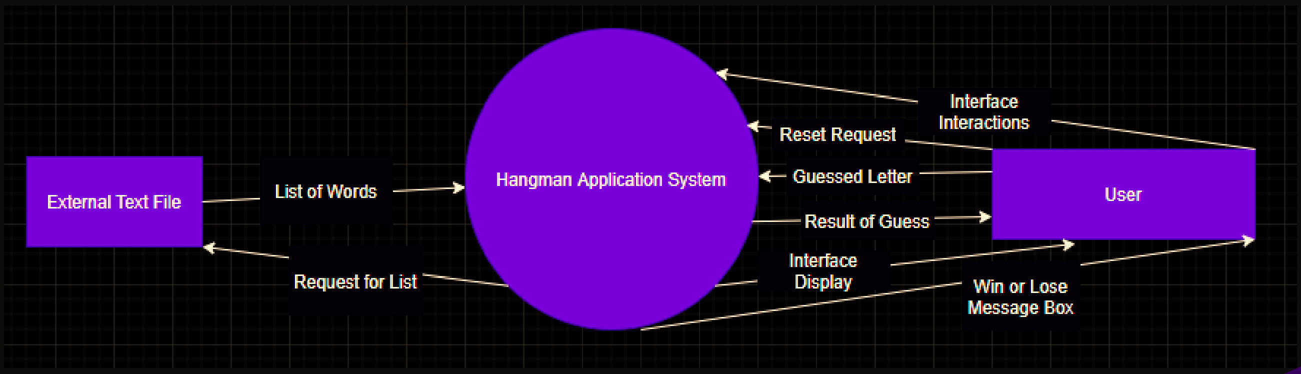
While social, legal and ethical implications were considered explored throughout the application's design, the solution has clear potential to further promote inclusivity which is greatly advantageous to the celebration of diversity in education and in software.

Modelling your chosen Solution

To assist in developing a deeper understanding of the chosen solution, a range of modelling tools have been developed. This begins with a level 0 context diagram, then a level 1 data flow diagram and finally the input process output (IPO) charts necessary for each module.

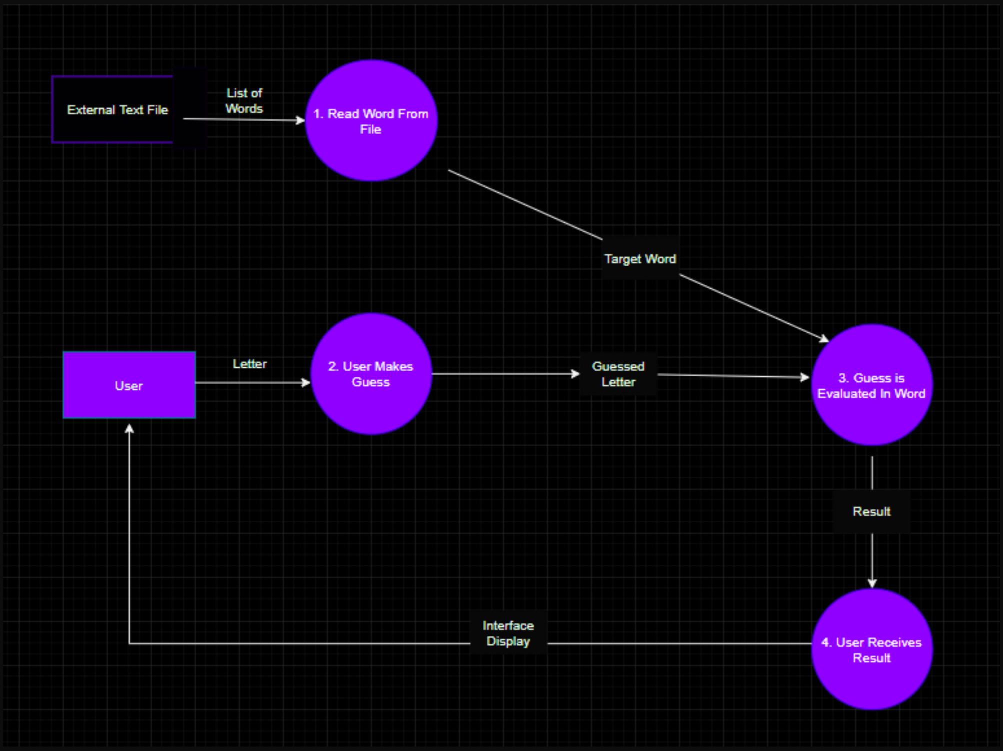
LEVEL 0 DIAGRAM | Context Diagram

The context diagram is the highest level in a data flow diagram and establishes the context and boundaries of the system to be modelled. By identifying the flow and interaction of information between the system and external entities, it provides an improved perspective of the scope under investigation.



LEVEL 1 DIAGRAM | Dataflow Diagram

The dataflow diagram illustrates the interaction between processes within the system and the flow of data between these processes. However, as the Hangman application processes typically read from public variables and rely on differing events to be used, **the broader diagram below represents the sequence of processes in the most effective and accurate way:**



MODELLING | Input Processing Output (IPO) Chart

An input, process, output chart is used to describe the data elements which will enter the module, the process necessary to produce the output, and the output itself that will leave the function. **A chart has been created for each module:**

Form1()

INPUT	PROCESS	OUTPUT
Windows Form application is launched	**1.** Hide all of the other elements so only the title screen and start button is visible	Title screen and start button is displayed

MakeInvisible()

INPUT	PROCESS	OUTPUT
All gameplay interface elements are visible	**1.** Hide all of the elements except the titleScreen, startButton, nextButton, loadingScreen and correctWord	All gameplay elements are hidden

StartButton_Click()

INPUT	PROCESS	OUTPUT
Mouse input clicking on startButton	**1.** Hide start button and title screen (as now game has started) **2.** Call Start()	A hidden start button Begin gameplay

Start()

INPUT	PROCESS	OUTPUT
GuessesLeft No elements displayed on the interface No guesses or gameplay active	**1.** Show 'loading screen' effect by calling LoadingScreen() **2.** Set guesses to 11 **3.** Hide staticImage picture box **4.** Show icon picture box **5.** Disable animatedImage (prevent **6.** Enable submit button **7.** Enable delete button **8.** Set letterGuessed textbox to empty	Seamless 'loading screen' transition Set/reset guesses Prevent gif from looping in background Allow use of submit and delete button Prepare for gameplay interface

LoadingScreen()

INPUT	PROCESS	OUTPUT
Current program interface	**1.** Make gameplay elements invisible **2.** Display a textbox which adds a "." at a specified time interval to creating a loading screen effect **3.** Hide textbox once the 'loading screen' is complete **4.** Display gameplay interface	Seamless loading screen transition Gameplay interface displayed for use

MakeVisible()

INPUT	PROCESS	OUTPUT
All gameplay interface elements are invisible	**1.** Show all the elements except the titleScreen, startButton, nextButton, loadingScreen and correctWord	All gameplay elements are displayed

SelectTargetWord()

INPUT	PROCESS	OUTPUT
External text file full of words	**1.** Open, sequentially read and close text file **2.** For each word in the text file read into a list **3.** Generate a random number **4.** Use random number as index for the list (to randomly select a word) **5.** Create a separate string which replaces each letter in target word with an underscore **6.** Add a space between each underscore **7.** Set textbox to display concealed word	Randomly selected target word Target word concealed under underscore placeholders Textbox displays concealed word (with spaces)

AddSpacesForDisplay()

INPUT	PROCESS	OUTPUT
Placeholders string	**1.** Split placeholder into an array of its characters **2.** Set/Reset display string to empty **3.** For the number of characters in the placeholder array, add that character and a space to the display variable **4.** Set the text of placeholderDisplay to the display string	Textbox displays concealed target word with spaces in between each placeholder

ActivateButtons()

INPUT	PROCESS	OUTPUT
All buttons are deactivated	**1.** Set all buttons on the keyboard to enabled = true	All buttons on the on-screen keyboard are activated

Letter_Click()

INPUT	PROCESS	OUTPUT
Button of letter played guessed	**1.** Set textbox to the text of the button the player guessed	Text box displays the letter the player has chosen

DeleteButton_Click()

INPUT	PROCESS	OUTPUT
Delete button clicked	**1.** Set the text box of current guess to “”	Remove any letter chosen/no guess is currently active

SubmitButton_Click()

INPUT	PROCESS	OUTPUT
Submit button clicked	**1.** Add a layer of protection so that an empty guess isn’t made **2.** Call Start() to begin	Guess is made and processed.

SubmitGuess()

INPUT	PROCESS	OUTPUT
Letter player guessed Target word	**1.** For each letter in target word, compare it with the guessed letter **2.** If they match, add position to a record of other positions which match **3.** For every position that matches, replace the underscore with the correct letter **4.** Update the text box with the concealed target word with the letter that has been revealed **5.** Add a space between each letter before display **6.** If no positions match, run FailedGuess() **7.** Call LoadingScreen()	If the guess was correct, reveal the letter in the placeholders string If the guess was incorrect, a hangman body part should appear and a guess left subtract Disable submit and delete button (as a guess was just submitted so no guess made yet) Loading face interface (to increase suspense)

FailedGuess()

INPUT	PROCESS	OUTPUT
Number of guesses left Loading screen interface	**1.** Subtract one guess from guesses left **2.** Call PlayAnimation() **3.** Call CheckForLoss()	One less guess left Indicate a life was lost If the player has lost, display loss interface

PlayAnimation()

INPUT	PROCESS	OUTPUT
Loading screen interface	**1.** Set a ‘next’ button to visible **2.** Call MakeInvisible() **3.** Show and enable animated image box **4.** Use switch case to choose which .gif image is displayed	Animated ‘drawing’ cut-screen of the hangman body part according to number of guesses left ‘Next’ button for when user is ready to make their next guess

NextButton_Click()

INPUT	PROCESS	OUTPUT
Next button clicked	**1.** Hide next button **2.** Call MakeVisible() to return to gameplay interface **3.** Hide and disenable animated image box **4.** Use switch case to choose which static image is displayed **5.** If case is 0 (no guesses left) then display textbox which contains word	Interface returns to gameplay interface Static image is included on the interface which changes according to number of guesses left Display loss interface (specifically once the next has been clicked)

CheckForLoss()

INPUT	PROCESS	OUTPUT
Number of guesses left	**1.** If number of guesses left is less than 1, set letter guessed text to "YOU LOSE" **2.** Disable buttons on the on-screen keyboard	Provide condolence message Disable on-screen keyboard (as loss has occurred)

CheckForWin()

INPUT	PROCESS	OUTPUT
Placeholders string	**1.** If there are no placeholders in the placeholders string (no underscores), set letter guessed text to "YOU WIN!" **2.** Disable buttons on the on-screen keyboard	Provide congratulatory message Disable on-screen keyboard (as win has occurred)

DeactivateButtons()

INPUT	PROCESS	OUTPUT
All buttons are activated	**1.** Set all buttons on the keyboard to enabled = false	All buttons on the on-screen keyboard are deactivated

FormClose()

INPUT	PROCESS	OUTPUT
Form closing	**1.** Trigger a message box to pop up which asks "Are you sure you want to exit?" **2.** If player selects no, cancel the form closing event	Protects form from being accidentally closed

ResetButton_Click()

INPUT	PROCESS	OUTPUT
Reset button clicked	**1.** Trigger a message box to pop up which asks "Are you sure you want to reset?" **2.** If player selects yes, call Start()	Protects gameplay from being accidentally reset Restarts game

PLANNING AND DESIGNING | PDR BODY

Software Development Approach

As the most suited software development approach to larger projects with a long history of successful use, the structured development approach lends itself to the specific nature of the Hangman solution. While team discussion and decision of this approach was only conducted by the developer as the hangman project was completed solo, extensive deliberation and justification still occurred. The minor variation in development requirements from the comprehensive problem definition and clear gameplay instructions of the project lends itself to this decision. While a significant disadvantage of the structured approach is that each stage in the process usually cannot be started until the previous stage is complete, the definite objectives of Hangman are well-suited to the thorough planning required as a result. Further, each deadline estimation can be reasonably accurate as the features won't expand. Thus, the resulting efficient time management can enhance the quality of the final application and is extremely appropriate to the long nature of the 5-week project. The structured approach also allows back-tracking to the previous stage in case a problem is discovered. This lends itself to software development, particularly as a learning developer, and its structure discourages procrastination. This report follows the 5 stages of the approach: defining and understanding the problem, planning and designing a solution, implementing the solution, testing and evaluating the solution and maintaining the solution. By employing the thorough process of the structured approach, a well-considered and justified product with effective documentation has been produced.

Algorithm Creation

```
BEGIN Form1
        MakeInvisible
END Form1

SET guessedLetter as public
SET guessedLetterChar as public
SET targetWord as public
SET placeholders as public
SET guessesLeft as public

BEGIN MakeInvisible
        Hide all elements of the gameplay interface
END MakeInvisible

BEGIN StartButton_Click
        Hide startButton
        Hide titleScreen
        Start
END StartButton_Click

BEGIN Start
        LoadingScreen
        SET Guesses to 11
        SET staticImage to Static0
        SET iconImage to Icon0
        Enable animatedImage
        Disable submitButton
        Disable deleteButton
```

This part of the algorithm is not included in any function as they are deliberately defined outside any module to be used as public variables. This is a result of the event-based subroutines of Windows Applications Forms.

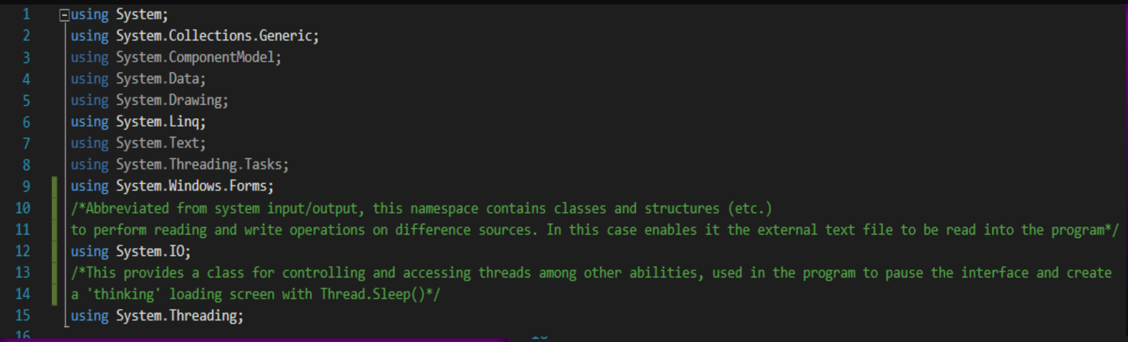
```
        SET textbox letterGuessed to “”
        SelectTargetWord
        ActivateButtons
END Start

BEGIN LoadingScreen
        MakeInvisible
        Show loadingScreen textbox
        FOR i = 0 TO 4 STEP 1
                Update interface
                Add “.” to loadingScreen textbox
                Wait 500 milliseconds
        NEXT i
        Set loadingScreen textbox to “”
        Wait 500 milliseconds
        Hide loadingScreen textbox
        MakeVisible
END LoadingScreen

START MakeVisible
        Show all elements of the gameplay interface
END MakeVisible

START SelectTargetWord
        SET path to file ”words.txt”
        Create new list wordPool
        Open words.txt file for reading
        Read all lines from words.txt into an array of lines
        Close words.txt file
        numberOfWordsInPool = 0
        FOREACH line IN the array of lines STEP 1
                wordsInFile = Split each line into words at the comma
                numberOfWordsInPoolAdjustedForIndex = length of wordsInFile - 1
                FOR i = 0 TO numberOfWordsInPoolAdjustedForIndex STEP 1
                        Add wordInFile (i) to wordPool
                NEXT i
        NEXT line
        randomIndex = Get a random number
        SET targetWord to wordpool (randomIndex)
        SET placeholders to "" to remove text
        FOR i = 0 TO length of target word STEP 1
                Add "_" to placeholders
        NEXT i
        AddSpacesForDisplay
END SelectTargetWord

START AddSpacesForDisplay
        GET placeholders
        splitPlaceholders = placeholders string split into separate characters
SET display textbox to ""
        FOR i = 0 TO length of placeholders STEP 1
                Add splitPlaceholders (i) to display
        STEP i
        SET the textbox placeholderDisplay to display
END AddSpacesForDisplay

```

```
START ActivateButtons
        Enable all buttons on the on-screen keyboard
END ActivateButtons

START Letter_Click
        GET letter
        SET the textbox letterGuessed to letter
        SET guessedLetterChar to letter
Enable submit button
        Enable delete button
END Letter_Click

START DeleteButton_Click
        SET letterGuessed textbox to  “”
        Disable submit button
        Disable delete button
END DeleteButton_Click

START SubmitButton_Click
        IF guessedletter <> “” THEN
                SubmitGuess
        ENDIF
END SubmitButton_Click

START SubmitGuess
        splitTargetWord = split target word into characters
        SET positionsOfGuessedLetter to empty list
        SET letterPosition to 0
        LoadingScreen
        FOREACH letter IN splitTargetWord
            If guessedLetterChar = letter THEN
                ADD letterPosition to positionsOfGuessedLetter
                SET letterGuessed textbox to “Correct!”
            ENDIF
            letterPosition = letterPosition + 1
        NEXT letter
        IF length of positionsOfGuessedLetter = 0 THEN
            FailedGuess
        ENDIF
        FOREACH position IN positionsOfGuessedLetter
            Placeholders = remove placeholder at position and insert with guessedLetter
        NEXT position
        Clear positionsOfGuessedLetter
        AddSpacesForDisplay
        CheckForWin
        Disable SubmitButton
        Disable DeleteButton
    END SubmitGuess

    START FailedGuess
        guessesLeft = guessesLeft - 1
        PlayAnimation
        CheckForLoss
    END FailedGuess

    START PlayAnimation
```

```
        Show nextButton
        MakeInvisible
        Enable animatedImage
        Show animatedImage
        CASEWHERE guesses is
            10 : SET animatedImage to Frame1
            9 : SET animatedImage to Frame2
            8 : SET animatedImage to Frame3
            7 : SET animatedImage to Frame4
            6 : SET animatedImage to Frame5
            5 : SET animatedImage to Frame6
            4 : SET animatedImage to Frame7
            3 : SET animatedImage to Frame8
            2 : SET animatedImage to Frame9
            1 : SET animatedImage to Frame10
            0 : SET animatedImage to AnimatedLose
        ENDCASE
    END PlayAnimation

    START NextButton_Click
        Hide nextButton
        MakeVisible
        Hide animatedImage
        Disable animatedImage
        Show staticImage
        CASEWHERE guesses is
            10 : SET staticImage to Static1
            9 : SET staticImage to Static2
```

```
                8 : SET staticImage to Static3
                7 : SET staticImage to Static4
                6 : SET staticImage to Static5
                5 : SET staticImage to Static6
                4 : SET staticImage to Static7
                3 : SET staticImage to Static8
                2 : SET staticImage to Static9
                1 : SET staticImage to Static10
                0 : SET staticImage to StaticLose
                    Show correctWord
                    Display "THE WORD WAS " + targetWord
                    SET image of Icon to Icon2
        ENDCASE
END NextButton_Click

START CheckForLoss
        IF guesses < 1 THEN
                Display "YOU LOSE"
                DeactivateButtons
        ENDIF
END CheckForLoss

START DeactiveButtons
        Disable all buttons on the keyboard
END DeactiveButtons

START FormClose
        End = Display a message box "Are you sure you want to exit?"
        IF End = no THEN
                Cancel the form from closing
        ENDIF
END FormClose

START ResetButton_Click
        Restart = Display a message box “Are you sure you want to restart?”
        IF Restart = yes THEN
                Hide correctWord
                Start
        ENDIF
END  RestartButton_Click
```

Development of Code



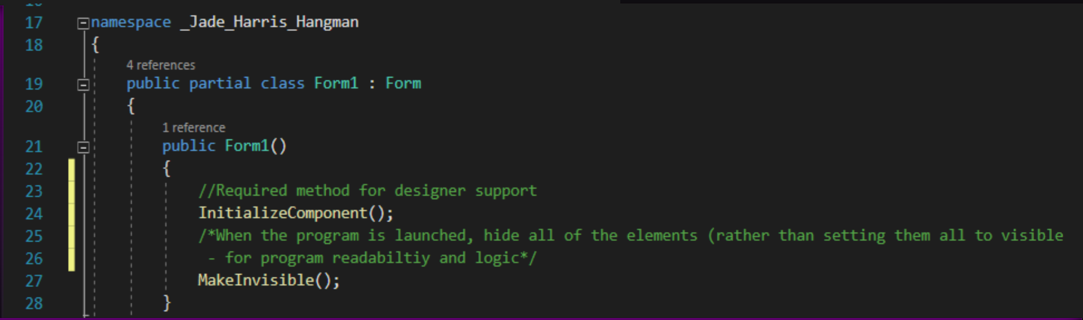
```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
/*Abbreviated from system input/output, this namespace contains classes and structures (etc.)
to perform reading and write operations on difference sources. In this case enables it the external text file to be read into the program*/
using System.IO;
/*This provides a class for controlling and accessing threads among other abilities, used in the program to pause the interface and create
a 'thinking' loading screen with Thread.Sleep()*/
using System.Threading;
```

NAMESPACES (class organisation)

Alongside the default namespaces provided with the .NET Framework, the program begins by adding System.IO and System.Threading. This allows the use of the file.ReadAllLines() and Thread.Sleep() methods. ReadAllLines opens the external file, reads all of its lines into a string array and then safely close the file again, while the thread.Sleep() is used in the program's 'loading screen' animation to add a pause

FORM1()

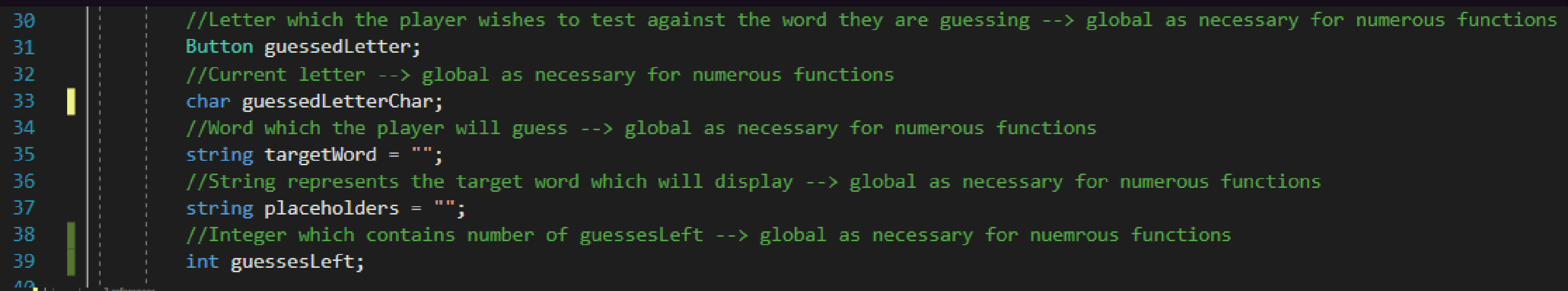
This code is provided default with the Windows Forms Application .NET framework, initializing the design components for the form. However, the MakeInvisible() function was also added as the program should launch straight into the title screen, hiding the actual gameplay elements. This function sets the visibility of the on-screen keyboard and other objects to false so that only the title screen image and button is visible.



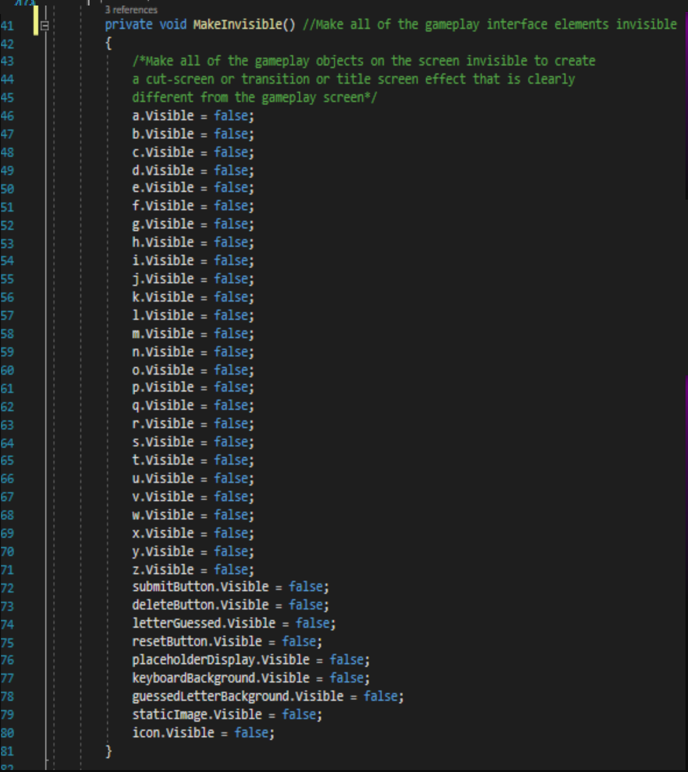
```
namespace _Jade_Harris_Hangman
{
    4 references
    public partial class Form1 : Form
    {
        1 reference
        public Form1()
        {
            //Required method for designer support
            InitializeComponent();
            /*When the program is launched, hide all of the elements (rather than setting them all to visible
             - for program readabiltiy and logic*/
            MakeInvisible();
        }
```

PUBLIC VARIABLES

These variables are deliberately not defined or initalised in any specific module as they serve as public variables. Because of the nature of the event-called functions, passing variables as parameters to functions does not work. This method allows the target word, guessed letter, placeholder string and number of guesses left to be accessed by numerous functions. For instance, the function which chooses the word and creates the placeholder string accordingly would not be able to just pass and call the placeholder string into the Letter_Click function as the function is waiting to be triggered by a click event. This is a simple solution.



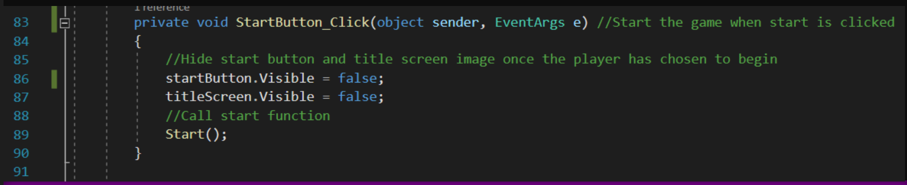
```
        //Letter which the player wishes to test against the word they are guessing --> global as necessary for numerous functions
        Button guessedLetter;
        //Current letter --> global as necessary for numerous functions
        char guessedLetterChar;
        //Word which the player will guess --> global as necessary for numerous functions
        string targetWord = "";
        //String represents the target word which will display --> global as necessary for numerous functions
        string placeholders = "";
        //Integer which contains number of guessesLeft --> global as necessary for nuemrous functions
        int guessesLeft;
```



```
        3 references
        private void MakeInvisible() //Make all of the gameplay interface elements invisible
        {
            /*Make all of the gameplay objects on the screen invisible to create
            a cut-screen or transition or title screen effect that is clearly
            different from the gameplay screen*/
            a.Visible = false;
            b.Visible = false;
            c.Visible = false;
            d.Visible = false;
            e.Visible = false;
            f.Visible = false;
            g.Visible = false;
            h.Visible = false;
            i.Visible = false;
            j.Visible = false;
            k.Visible = false;
            l.Visible = false;
            m.Visible = false;
            n.Visible = false;
            o.Visible = false;
            p.Visible = false;
            q.Visible = false;
            r.Visible = false;
            s.Visible = false;
            t.Visible = false;
            u.Visible = false;
            v.Visible = false;
            w.Visible = false;
            x.Visible = false;
            y.Visible = false;
            z.Visible = false;
            submitButton.Visible = false;
            deleteButton.Visible = false;
            letterGuessed.Visible = false;
            resetButton.Visible = false;
            placeholderDisplay.Visible = false;
            keyboardBackground.Visible = false;
            guessedLetterBackground.Visible = false;
            staticImage.Visible = false;
            icon.Visible = false;
        }
```

MAKEINVISIBLE()

This module hides the on-screen keyboard, textbox which displays the placeholders, static image and icon image invisible to create the illusion of separate interfaces of a loading screen, cut-screen and title screen.



```
        private void StartButton_Click(object sender, EventArgs e) //Start the game when start is clicked
        {
            //Hide start button and title screen image once the player has chosen to begin
            startButton.Visible = false;
            titleScreen.Visible = false;
            //Call start function
            Start();
        }
```

STARTBUTTON_CLICK()

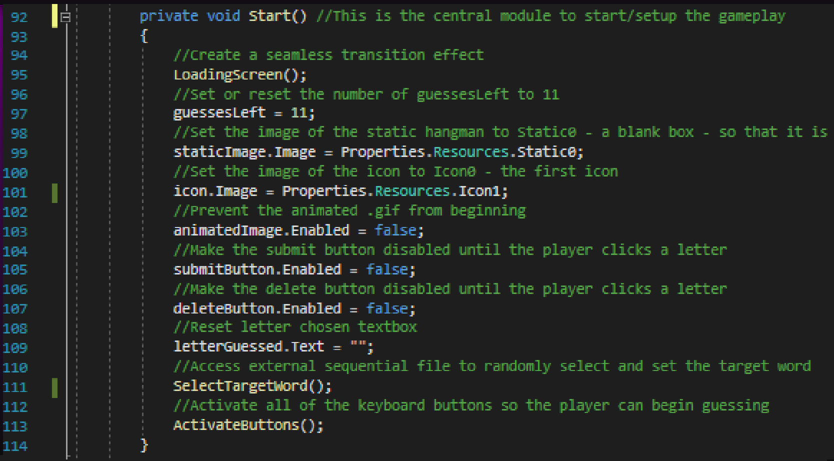
This module triggers the start events to occur when the title screen start button is clicked. By hiding the titleScreen picture box (to create the effect the user is moving from the title screen to in game) and making the start button invisible, this aesthetic addition is successfully achieved and provides a clean transition. The start button is only the used on the first time the game is launched, hence why Start() is a separate function so that it can be triggered by other events too (restart).

START()

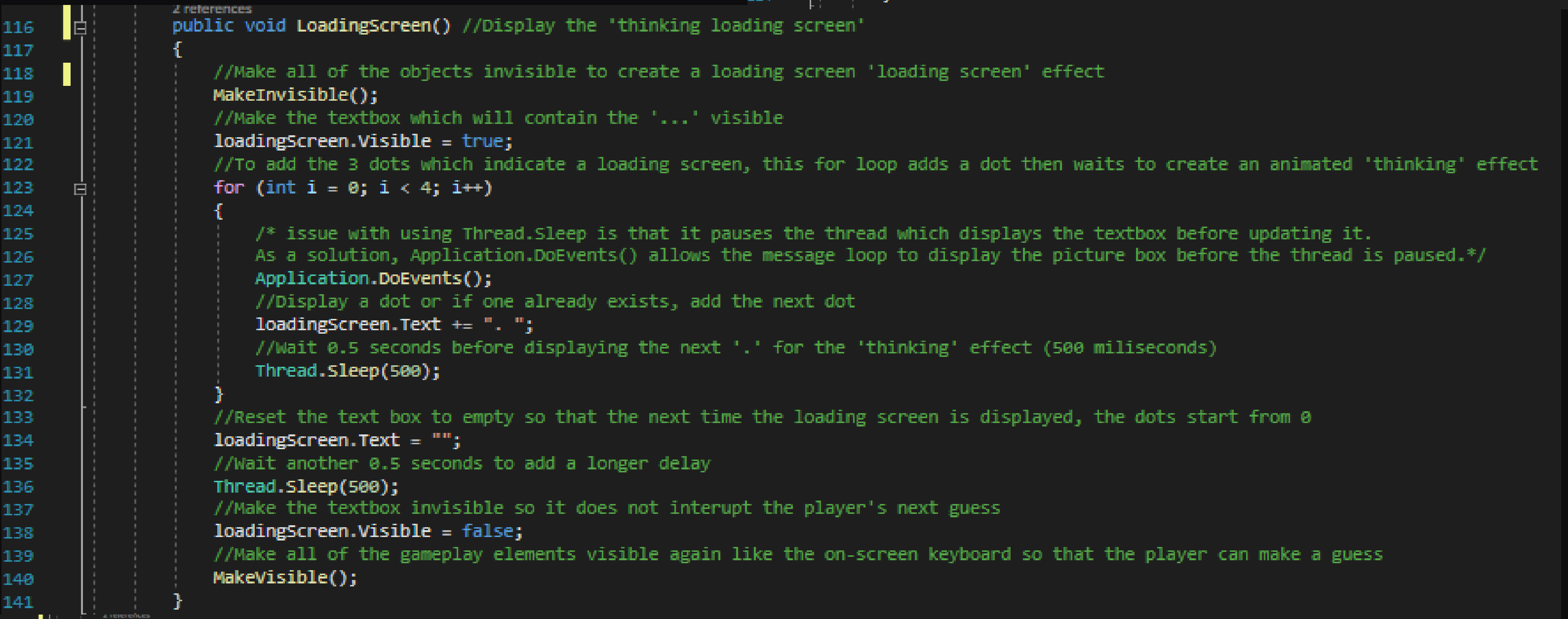
Called from either a click event from the start button when the player first launches the application, or from the reset button if they choose to respond to the message box “Would you like to restart?” with a yes – this function starts/restarts the game. The function begins by calling the LoadingScreen() function, a ‘thinking’ cut-screen designed to create a smooth transition. While guesses is defined but not initialized as a public variable, it is given a value here. This is because if the player chooses to restart, by only running this function, the number of guesses can be reset as well. The module then sets the images of the static image box to image 0 – an image of a black box to make the hangman appear invisible (particularly important when a game is reset as the number of guesses is reset too). This could have been achieved by hiding the box, but changing the image is more consistent with the switch case. The module then sets the icon image as well, a minor design flair which contributes to the overall aesthetics. Any .gif’s currently running are paused (precaution) and the submit and delete buttons for the keyboard are disabled (as now the keyboard is visible but no guess has been made yet). Rather than simply having code which prevents the user submitting or deleting an empty guess, it makes more sense for the interface to disable these buttons from use at all. The next line resets any guessed letter (or sets the textbox which displays the user’s current guess) to empty. The module then calls the separate SelectTargetWord() and ActivateButtons() on the keyboard then the user can make their first guess.

LOADINGSCREEN()

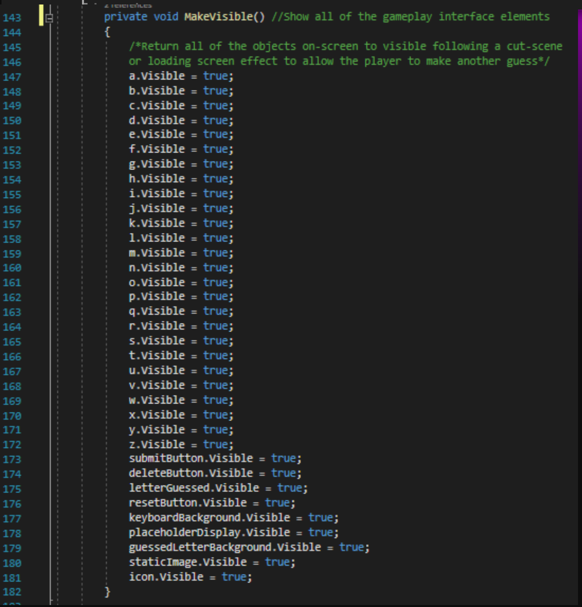
This module contributes to usability and aesthetics, which enhance the overall user experience and player engagement. This module attempts to create a loading screen and ‘thinking’ effect to add a distinct flair to the game interface. Firstly all of the main gameplay elements are made invisible then a textbox is shown which prints a “. . .” effect. This creates the illusion the computer is ‘processing’ and is achieved by using the Thread.Sleep() namespace in combination with a for loop to add another “.” to the textbox each 5 milliseconds. 5 milliseconds is an optimal amount of time as it retains player interest without becoming tedious. The Application.DoEvents() is expanded on in the later ‘TESTING’ section of this report,. In effect, it was a workaround for an issue Thread.Sleep() exposes in Windows Forms applications. Once the loading screen has ‘complete’, the textbox is set to empty and hidden prepared for the next function call, and the gameplay elements prepared for their next use.



```
        private void Start() //This is the central module to start/setup the gameplay
        {
            //Create a seamless transition effect
            LoadingScreen();
            //Set or reset the number of guessesLeft to 11
            guessesLeft = 11;
            //Set the image of the static hangman to Static0 - a blank box - so that it is
            staticImage.Image = Properties.Resources.Static0;
            //Set the image of the icon to Icon0 - the first icon
            icon.Image = Properties.Resources.Icon1;
            //Prevent the animated .gif from beginning
            animatedImage.Enabled = false;
            //Make the submit button disabled until the player clicks a letter
            submitButton.Enabled = false;
            //Make the delete button disabled until the player clicks a letter
            deleteButton.Enabled = false;
            //Reset letter chosen textbox
            letterGuessed.Text = "";
            //Access external sequential file to randomly select and set the target word
            SelectTargetWord();
            //Activate all of the keyboard buttons so the player can begin guessing
            ActivateButtons();
        }
```



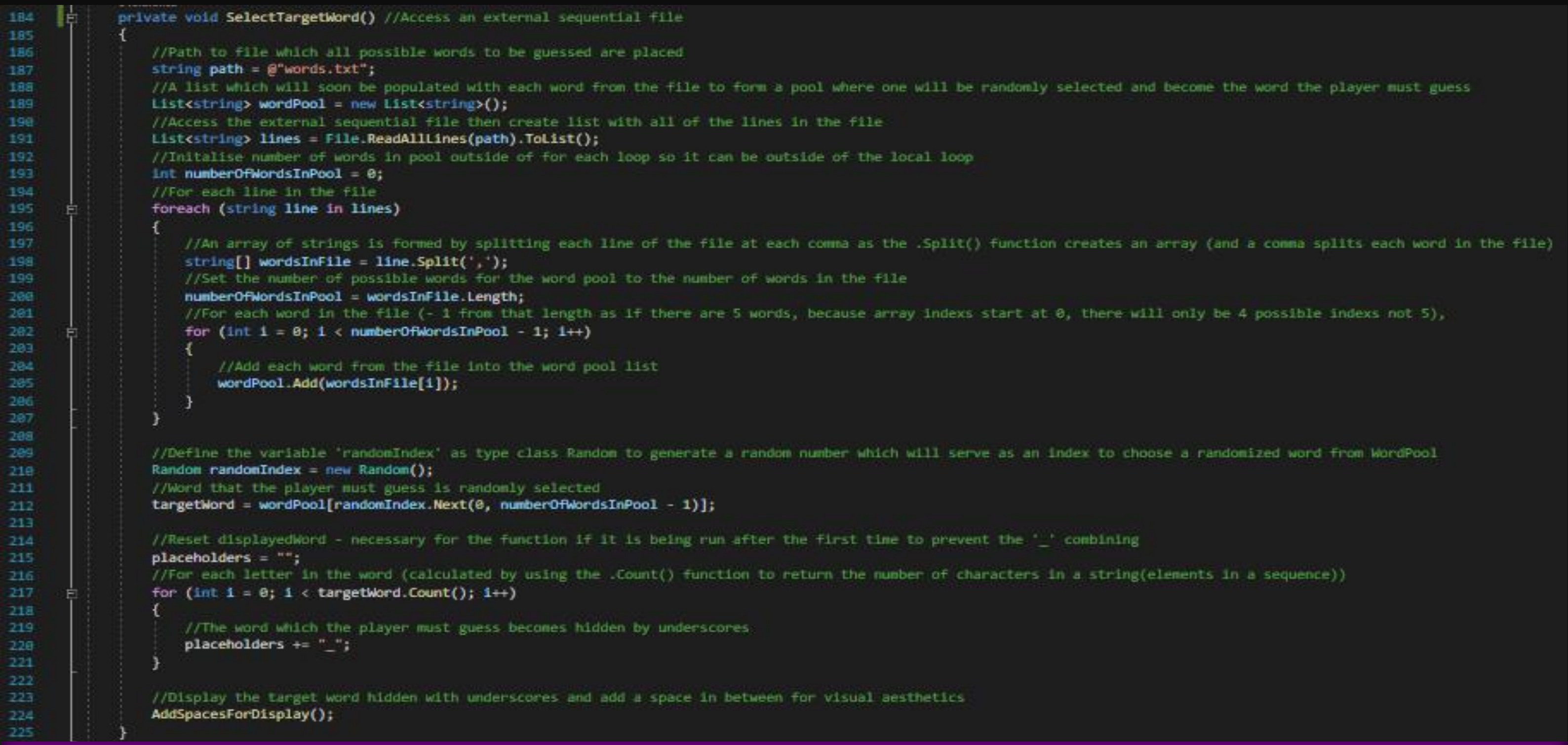
```
        2 references
        public void LoadingScreen() //Display the 'thinking loading screen'
        {
            //Make all of the objects invisible to create a loading screen 'loading screen' effect
            MakeInvisible();
            //Make the textbox which will contain the '...' visible
            loadingScreen.Visible = true;
            //To add the 3 dots which indicate a loading screen, this for loop adds a dot then waits to create an animated 'thinking' effect
            for (int i = 0; i < 4; i++)
            {
                /* issue with using Thread.Sleep is that it pauses the thread which displays the textbox before updating it.
                As a solution, Application.DoEvents() allows the message loop to display the picture box before the thread is paused.*/
                Application.DoEvents();
                //Display a dot or if one already exists, add the next dot
                loadingScreen.Text += ". ";
                //Wait 0.5 seconds before displaying the next '.' for the 'thinking' effect (500 miliseconds)
                Thread.Sleep(500);
            }
            //Reset the text box to empty so that the next time the loading screen is displayed, the dots start from 0
            loadingScreen.Text = "";
            //Wait another 0.5 seconds to add a longer delay
            Thread.Sleep(500);
            //Make the textbox invisible so it does not interupt the player's next guess
            loadingScreen.Visible = false;
            //Make all of the gameplay elements visible again like the on-screen keyboard so that the player can make a guess
            MakeVisible();
        }
```



```
        private void MakeVisible() //Show all of the gameplay interface elements
        {
            /*Return all of the objects on-screen to visible following a cut-scene
            or loading screen effect to allow the player to make another guess*/
            a.Visible = true;
            b.Visible = true;
            c.Visible = true;
            d.Visible = true;
            e.Visible = true;
            f.Visible = true;
            g.Visible = true;
            h.Visible = true;
            i.Visible = true;
            j.Visible = true;
            k.Visible = true;
            l.Visible = true;
            m.Visible = true;
            n.Visible = true;
            o.Visible = true;
            p.Visible = true;
            q.Visible = true;
            r.Visible = true;
            s.Visible = true;
            t.Visible = true;
            u.Visible = true;
            v.Visible = true;
            w.Visible = true;
            x.Visible = true;
            y.Visible = true;
            z.Visible = true;
            submitButton.Visible = true;
            deleteButton.Visible = true;
            letterGuessed.Visible = true;
            resetButton.Visible = true;
            keyboardBackground.Visible = true;
            placeholderDisplay.Visible = true;
            guessedLetterBackground.Visible = true;
            staticImage.Visible = true;
            icon.Visible = true;
        }
```

MAKEVISIBLE()

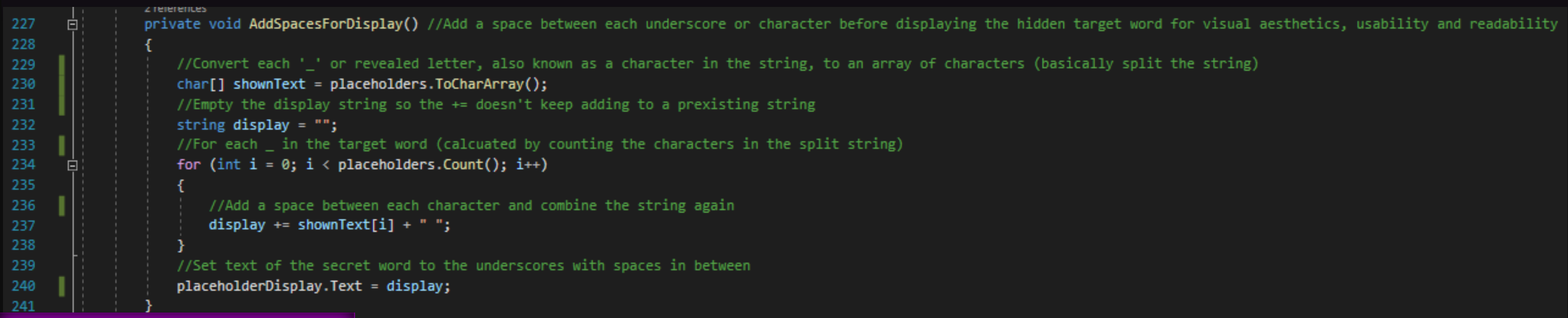
This module reverses MakeInvisible(), showing all of the gameplay elements to allow the player to guess a letter or, if it the end of the game, see if they have won or lost and what the target word was.



```
private void SelectTargetWord() //Access an external sequential file
{
    //Path to file which all possible words to be guessed are placed
    string path = @"words.txt";
    //A list which will soon be populated with each word from the file to form a pool where one will be randomly selected and become the word the player must guess
    List<string> wordPool = new List<string>();
    //Access the external sequential file then create list with all of the lines in the file
    List<string> lines = File.ReadAllLines(path).ToList();
    //Initalise number of words in pool outside of for each loop so it can be outside of the local loop
    int numberOfWordsInPool = 0;
    //For each line in the file
    foreach (string line in lines)
    {
        //An array of strings is formed by splitting each line of the file at each comma as the .Split() function creates an array (and a comma splits each word in the file)
        string[] wordsInFile = line.Split(',');
        //Set the number of possible words for the word pool to the number of words in the file
        numberOfWordsInPool = wordsInFile.Length;
        //For each word in the file (- 1 from that length as if there are 5 words, because array indexs start at 0, there will only be 4 possible indexs not 5),
        for (int i = 0; i < numberOfWordsInPool - 1; i++)
        {
            //Add each word from the file into the word pool list
            wordPool.Add(wordsInFile[i]);
        }
    }

    //Define the variable 'randomIndex' as type class Random to generate a random number which will serve as an index to choose a randomized word from WordPool
    Random randomIndex = new Random();
    //Word that the player must guess is randomly selected
    targetWord = wordPool[randomIndex.Next(0, numberOfWordsInPool - 1)];

    //Reset displayedWord - necessary for the function if it is being run after the first time to prevent the '_' combining
    placeholders = "";
    //For each letter in the word (calculated by using the .Count() function to return the number of characters in a string(elements in a sequence))
    for (int i = 0; i < targetWord.Count(); i++)
    {
        //The word which the player must guess becomes hidden by underscores
        placeholders += "_";
    }

    //Display the target word hidden with underscores and add a space in between for visual aesthetics
    AddSpacesForDisplay();
}
```

SELECTTARGETWORD()

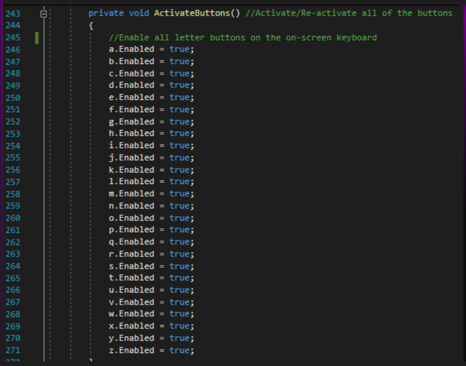
This module selects the random target word by reading the external text file and then conceals it beneath underscore placeholders. Called after the Start or Restart function is called, this segment of code serves to select a target word (replacing or setting a new word). By using the File.ReadAllLines method, the external text file (words.txt as defined by path) is opened, and placed into an array and then closed. The .ToList() function is used due to developer personal preference, as the list data structure has a simple and readable .Add() function. The number of the words in the pool is also defined outside of the for loop for readability. Each line in the words file is then separated at the comma and added to an array as separate elements. This array is read into a for loop which goes from 0 to the number of words in the word pool – 1 (this is necessary as by default for loops start at 0 so this prevents the loop attempting to add an empty array element to wordsInFile. Each word in the array is then added to the WordPool list. A random number is generated using the random class which is then used as the index to select a random word from the pool of words. This is how the random word requirement is achieved. The global variable target word is then set to this word. In the most readable way, to create the placeholders that conceal the word, a for loop is simply used which goes from 0 to the number of letters in the word (-1 as the for loop starts at 0) . This adds an underscore to the global placeholders variable. For aesthetics, a separate function which adds spaces in between these underscores is called. While here the 'placeholders' variable could be passed through as a parameter, it is already a global variable so this would be redundant.



```
2 references
private void AddSpacesForDisplay() //Add a space between each underscore or character before displaying the hidden target word for visual aesthetics, usability and readability
{
    //Convert each '_' or revealed letter, also known as a character in the string, to an array of characters (basically split the string)
    char[] shownText = placeholders.ToCharArray();
    //Empty the display string so the += doesn't keep adding to a prexisting string
    string display = "";
    //For each _ in the target word (calcuated by counting the characters in the split string)
    for (int i = 0; i < placeholders.Count(); i++)
    {
        //Add a space between each character and combine the string again
        display += shownText[i] + " ";
    }
    //Set text of the secret word to the underscores with spaces in between
    placeholderDisplay.Text = display;
}
```

ACTIVATEBUTTONS()

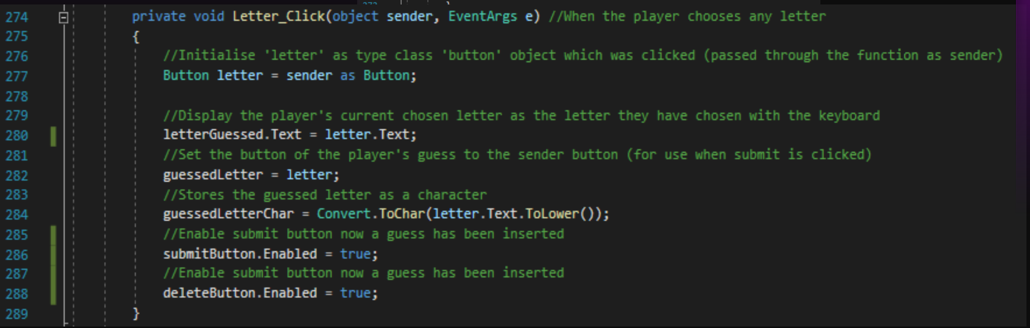
The final subroutine called from the start button, this module simply activates the on-screen keyboard so that the player can make their first guess.



```
private void ActivateButtons() //Activate/Re-activate all of the buttons
{
    //Enable all letter buttons on the on-screen keyboard
    a.Enabled = true;
    b.Enabled = true;
    c.Enabled = true;
    d.Enabled = true;
    e.Enabled = true;
    f.Enabled = true;
    g.Enabled = true;
    h.Enabled = true;
    i.Enabled = true;
    j.Enabled = true;
    k.Enabled = true;
    l.Enabled = true;
    m.Enabled = true;
    n.Enabled = true;
    o.Enabled = true;
    p.Enabled = true;
    q.Enabled = true;
    r.Enabled = true;
    s.Enabled = true;
    t.Enabled = true;
    u.Enabled = true;
    v.Enabled = true;
    w.Enabled = true;
    x.Enabled = true;
    y.Enabled = true;
    z.Enabled = true;
```

ADDSPACESFORDISPLAY()

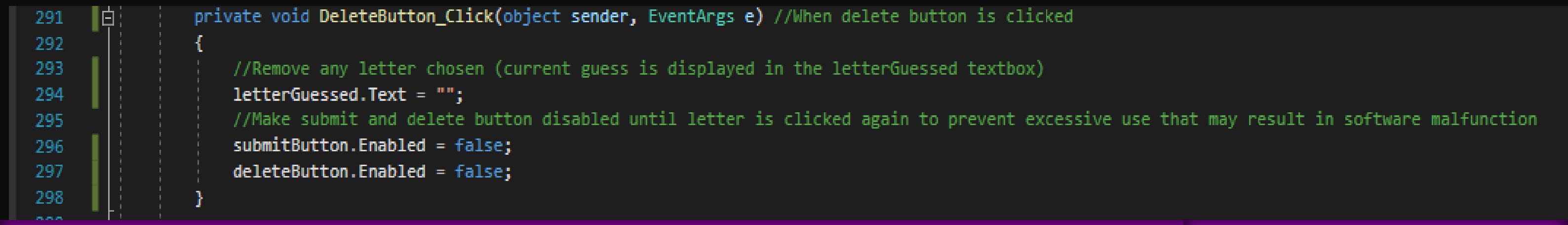
This module adds a space between the underscores and/or any revealed letters by splitting the string into an array of characters then adding the character itself and a space and combining the string again. The placeholderDisplay textbox then displays this more readable progress indicator as it now has spaces in between.



```
private void Letter_Click(object sender, EventArgs e) //When the player chooses any letter
{
    //Initialise 'letter' as type class 'button' object which was clicked (passed through the function as sender)
    Button letter = sender as Button;

    //Display the player's current chosen letter as the letter they have chosen with the keyboard
    letterGuessed.Text = letter.Text;
    //Set the button of the player's guess to the sender button (for use when submit is clicked)
    guessedLetter = letter;
    //Stores the guessed letter as a character
    guessedLetterChar = Convert.ToChar(letter.Text.ToLower());
    //Enable submit button now a guess has been inserted
    submitButton.Enabled = true;
    //Enable submit button now a guess has been inserted
    deleteButton.Enabled = true;
}
```

LETTER_CLICK()

Whenever a letter is selected, this module enters the selected letter into the guessed letter text box to indicate to the user that it is the letter that will be guessed once they hit submit. This dynamically stores the current guessed letter for use if the player hits submit. Once a letter has been submitted, the submit and delete buttons also become enabled for use (as a guess has now been made)



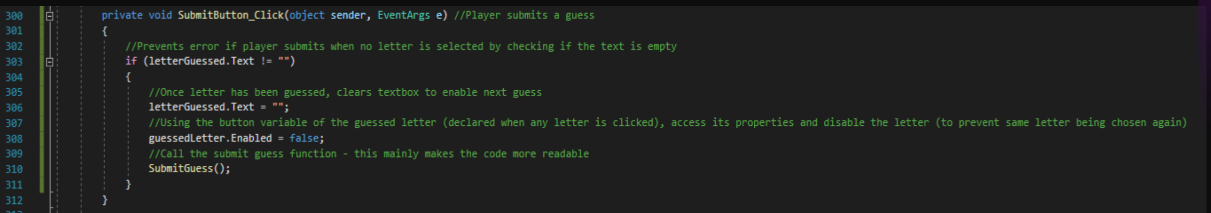
```
private void DeleteButton_Click(object sender, EventArgs e) //When delete button is clicked
{
    //Remove any letter chosen (current guess is displayed in the letterGuessed textbox)
    letterGuessed.Text = "";
    //Make submit and delete button disabled until letter is clicked again to prevent excessive use that may result in software malfunction
    submitButton.Enabled = false;
    deleteButton.Enabled = false;
}
```

DELETEBUTTON()

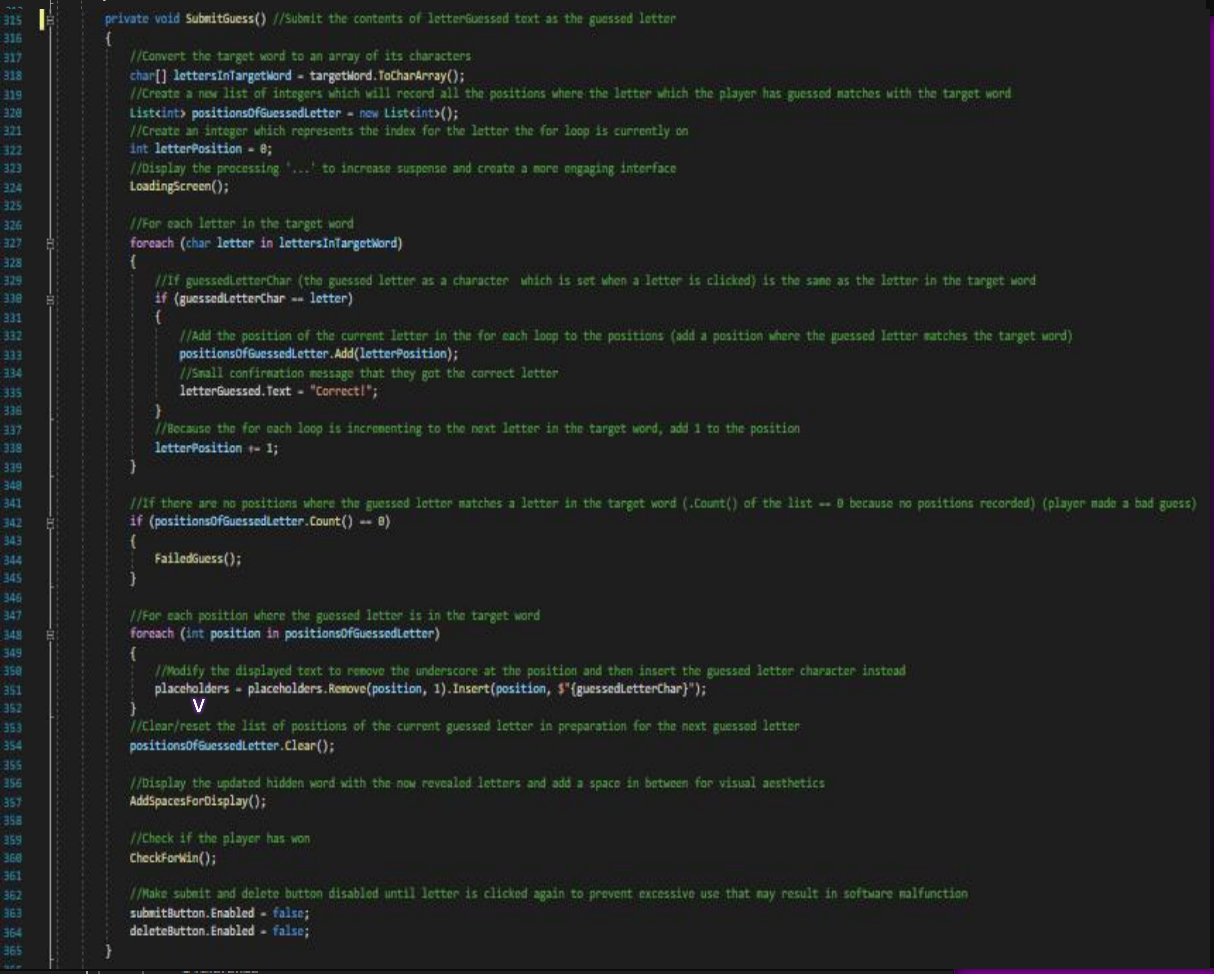
Intended for those who require learning aid to assist them in visualizing their guess before submitting, a delete button to clear the textbox with the current guess is provided by setting the text to empty. This also disables the submit and delete button for usability as there is now nothing to submit.

SUBMITBUTTON()

Additional safety-guard to prevent user from entering an empty guess (for instance if their machine is slower functioning and does not disable the submit button in time). This also sets the letter that the user has guessed to disabled so that it cannot be used again, and also so the player can keep track of which letters they have already selected.



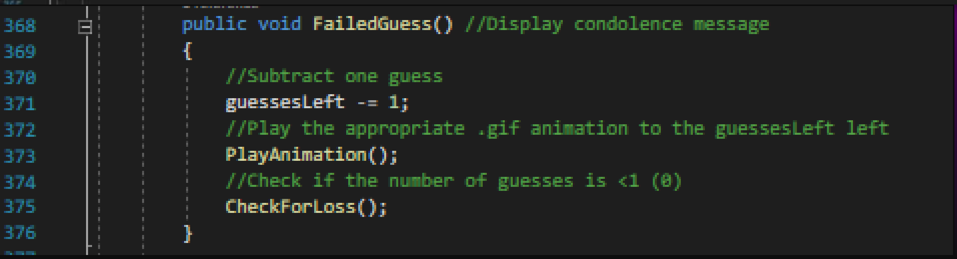
```
private void SubmitButton_Click(object sender, EventArgs e) //Player submits a guess
{
    //Prevents error if player submits when no letter is selected by checking if the text is empty
    if (letterGuessed.Text != "")
    {
        //Once letter has been guessed, clears textbox to enable next guess
        letterGuessed.Text = "";
        //Using the button variable of the guessed letter (declared when any letter is clicked), access its properties and disable the letter (to prevent same letter being chosen again)
        guessedLetter.Enabled = false;
        //Call the submit guess function - this mainly makes the code more readable
        SubmitGuess();
    }
}
```



```
private void SubmitGuess() //Submit the contents of letterGuessed text as the guessed letter
{
    //Convert the target word to an array of its characters
    char[] lettersInTargetWord = targetWord.ToCharArray();
    //Create a new list of integers which will record all the positions where the letter which the player has guessed matches with the target word
    List<int> positionsOfGuessedLetter = new List<int>();
    //Create an integer which represents the index for the letter the for loop is currently on
    int letterPosition = 0;
    //Display the processing '...' to increase suspense and create a more engaging interface
    LoadingScreen();

    //For each letter in the target word
    foreach (char letter in lettersInTargetWord)
    {
        //If guessedLetterChar (the guessed letter as a character  which is set when a letter is clicked) is the same as the letter in the target word
        if (guessedLetterChar == letter)
        {
            //Add the position of the current letter in the for each loop to the positions (add a position where the guessed letter matches the target word)
            positionsOfGuessedLetter.Add(letterPosition);
            //Small confirmation message that they got the correct letter
            letterGuessed.Text = "Correct!";
        }
        //Because the for each loop is incrementing to the next letter in the target word, add 1 to the position
        letterPosition += 1;
    }

    //If there are no positions where the guessed letter matches a letter in the target word (.Count() of the list == 0 because no positions recorded) (player made a bad guess)
    if (positionsOfGuessedLetter.Count() == 0)
    {
        FailedGuess();
    }

    //For each position where the guessed letter is in the target word
    foreach (int position in positionsOfGuessedLetter)
    {
        //Modify the displayed text to remove the underscore at the position and then insert the guessed letter character instead
        placeholders = placeholders.Remove(position, 1).Insert(position, $"{guessedLetterChar}");
    }      V
    //Clear/reset the list of positions of the current guessed letter in preparation for the next guessed letter
    positionsOfGuessedLetter.Clear();

    //Display the updated hidden word with the now revealed letters and add a space in between for visual aesthetics
    AddSpacesForDisplay();

    //Check if the player has won
    CheckForWin();

    //Make submit and delete button disabled until letter is clicked again to prevent excessive use that may result in software malfunction
    submitButton.Enabled = false;
    deleteButton.Enabled = false;
}
```

SUBMITGUESS()

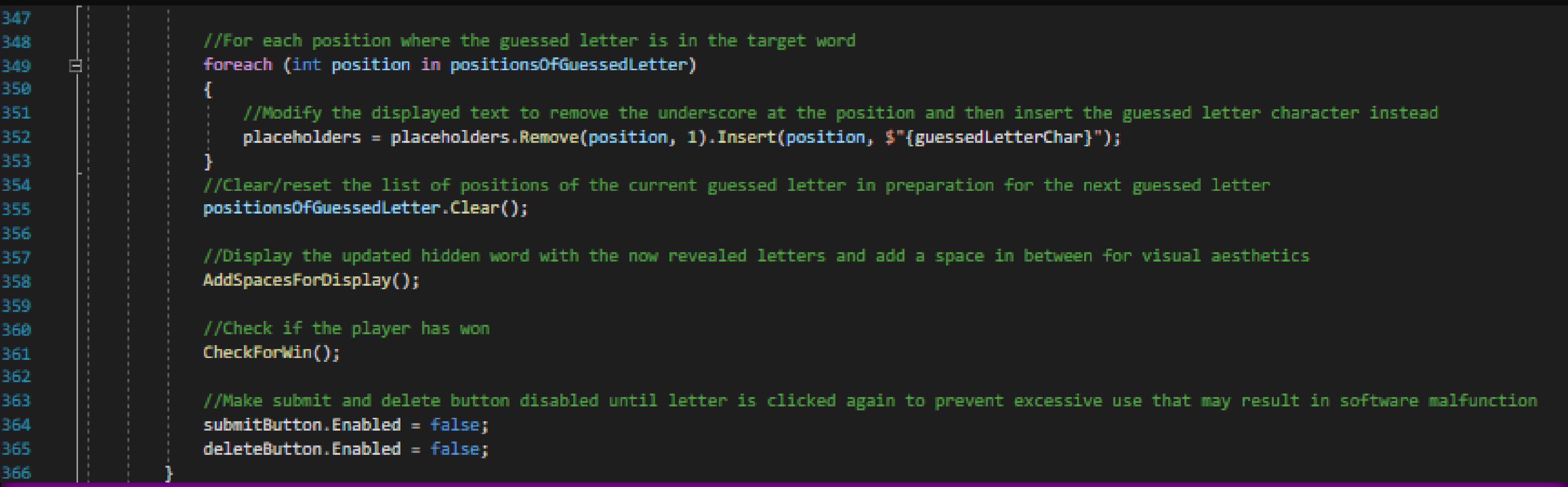
This module executes when the person enters their guess – either called from the start button click subroutine or on reset. The module should evaluate the guessed letter against all letters in the target word and if there are no matches, then it was an incorrect guess, otherwise it should reveal the letter in the correct position/s, It begins by splitting the target word into an array of its characters. A list of integers which contains the positions of the guessed letter is also created – this is necessary as there may be multiple positions. The loading screen is called for user-experience, creating an effect as if the computer is thinking about their result. Then, for each letter in the target word, it will be compared against the guessed letter. This is stored as a global character variable which is updated on LetterClick(). The loss condition is achieved by counting the length of the array, if it is empty then there have not been any matches so the player has made an incorrect guess.



```
public void FailedGuess() //Display condolence message
{
    //Subtract one guess
    guessesLeft -= 1;
    //Play the appropriate .gif animation to the guessesLeft left
    PlayAnimation();
    //Check if the number of guesses is <1 (0)
    CheckForLoss();
}
```

FAILEDGUESS()

This module is called if the guessed letter was incorrect and not contained in the target word. This subtracts one guess from the global guessesLeft variable, calls a module to play the hangman animation and then calls a subroutine to check for loss.



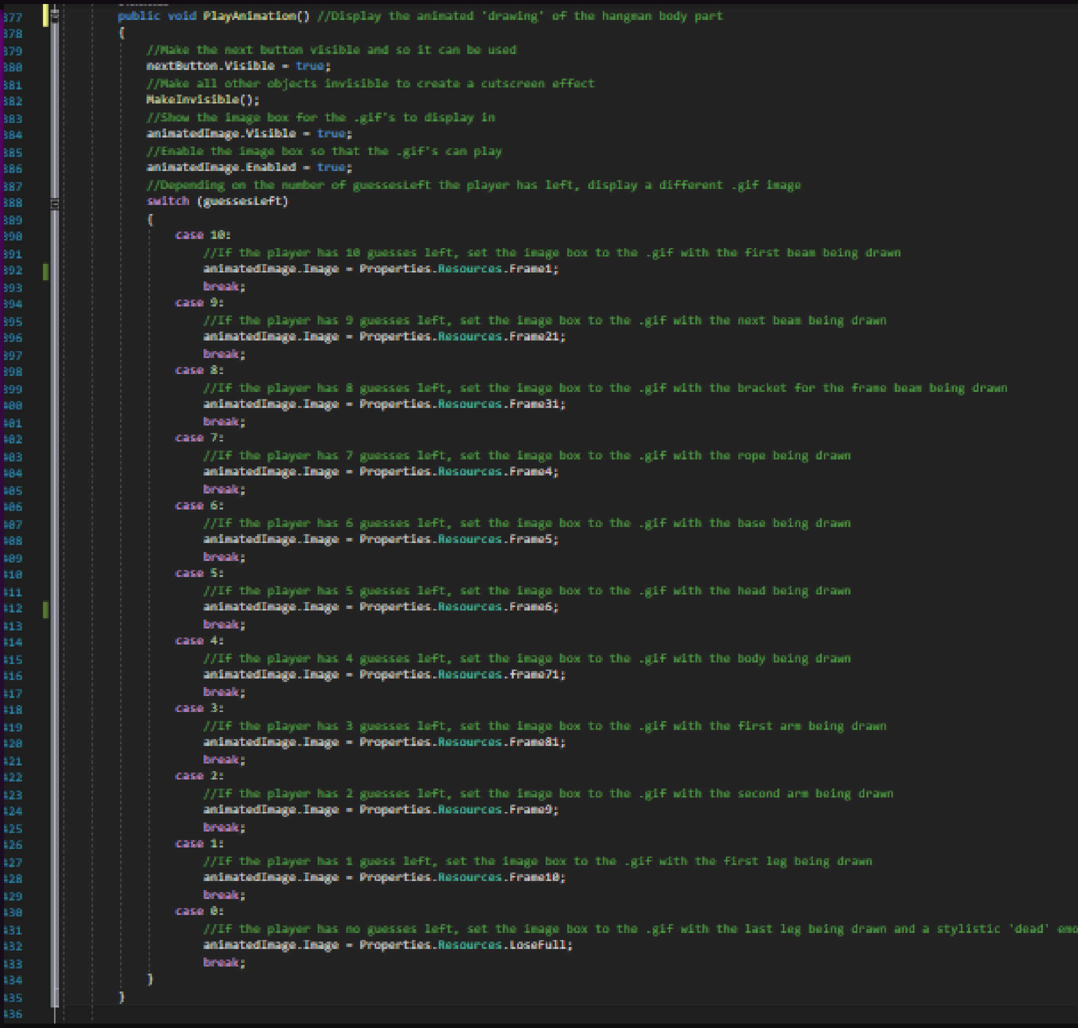
```
            //For each position where the guessed letter is in the target word
            foreach (int position in positionsOfGuessedLetter)
            {
                //Modify the displayed text to remove the underscore at the position and then insert the guessed letter character instead
                placeholders = placeholders.Remove(position, 1).Insert(position, $"{guessedLetterChar}");
            }
            //Clear/reset the list of positions of the current guessed letter in preparation for the next guessed letter
            positionsOfGuessedLetter.Clear();

            //Display the updated hidden word with the now revealed letters and add a space in between for visual aesthetics
            AddSpacesForDisplay();

            //Check if the player has won
            CheckForWin();

            //Make submit and delete button disabled until letter is clicked again to prevent excessive use that may result in software malfunction
            submitButton.Enabled = false;
            deleteButton.Enabled = false;
        }
```

SUBMITGUESS() CONTINUED…

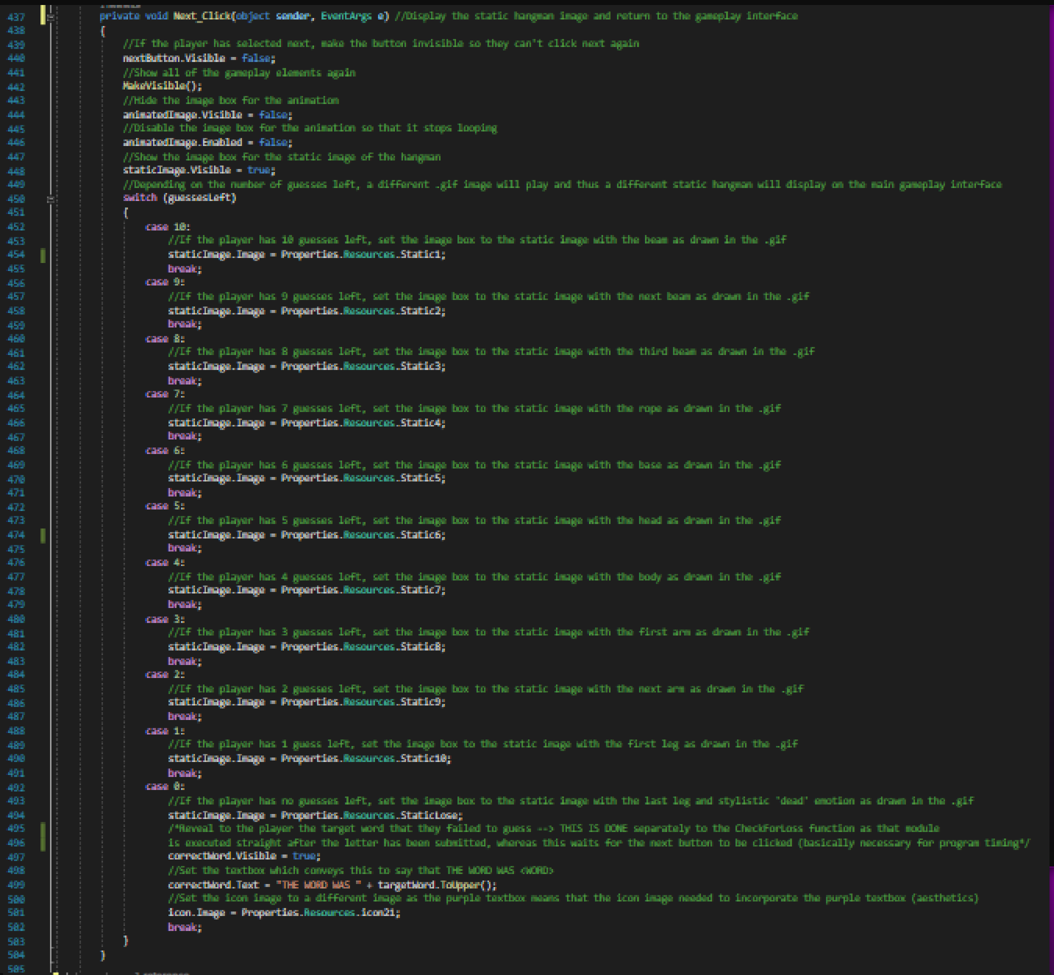
If this occurs, a separate failed guess function occurs. However, if there are positions (this will just be skipped if there aren't), the remove and insert methods are used to reveal the letter. The remove method simply returns a new string which removes 1 letter from that specific position. The insert button is then used to insert the correct letter in the position. This is achieved by using string interpolation (prefixing the string with $ then adding the variable in {}) in the Insert method's string value input. This inserts the guessed letter at the position/s – this code is specifically useful for efficient and readable code when there are multiple instances of the same letter in a word. The list of positions is then cleared for its next use, spaces are added to display the word by calling the AddSpacesForDisplay() function (and as all of the string modification occurred on the global placeholder variable) and the CheckForWin() module. For program consistency, once the player has submitted a guess the submit and delete button is disabled again as there is now no current guess (because the guess was just submitted and text box set to empty).

PLAYANIMATION()

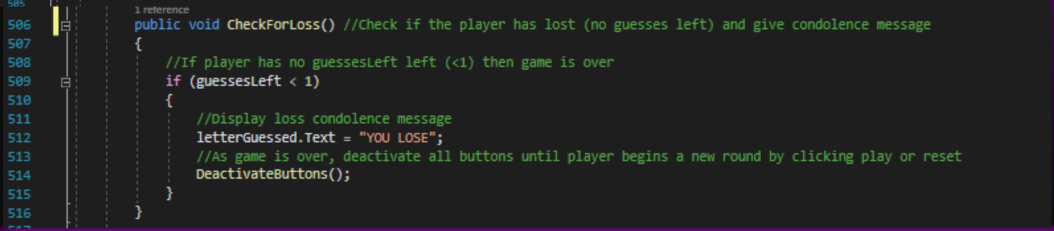
This subroutine achieves the animated drawing hangman by utilising a number of .gif assets uploaded in the program's resource folder. To create a cut-screen effect, all of the gameplay elements are set to invisible again, but a 'NEXT' button is revealed. The explanation for this addition is included in the 'TESTING' section, but this allows the user to complete the cut screen and continue. This was mainly a response to the infinitely looping .gifs (graphics interchange format), but ultimately became a better addition than originally intended. The image box which contains the .gifs then is set to visible and enabled, this allows the .gif's to play. Depending on how many guesses the player has left and therefore what stage the hangman should be, the image box is set to a different gif. These animations were designed specifically for the game and leverage Adobe Photoshop to achieve their effect. This is how the animated .gif effect was achieved however, and it makes for a unique and engaging gameplay experience.



```
        public void PlayAnimation() //Display the animated 'drawing' of the hangman body part
        {
            //Make the next button visible and so it can be used
            nextButton.Visible = true;
            //Make all other objects invisible to create a cutscreen effect
            MakeInvisible();
            //Show the image box for the .gif's to display in
            animatedImage.Visible = true;
            //Enable the image box so that the .gif's can play
            animatedImage.Enabled = true;
            //Depending on the number of guessesLeft the player has left, display a different .gif image
            switch (guessesLeft)
            {
                case 10:
                    //If the player has 10 guesses left, set the image box to the .gif with the first beam being drawn
                    animatedImage.Image = Properties.Resources.Frame1;
                    break;
                case 9:
                    //If the player has 9 guesses left, set the image box to the .gif with the next beam being drawn
                    animatedImage.Image = Properties.Resources.Frame21;
                    break;
                case 8:
                    //If the player has 8 guesses left, set the image box to the .gif with the bracket for the frame beam being drawn
                    animatedImage.Image = Properties.Resources.Frame31;
                    break;
                case 7:
                    //If the player has 7 guesses left, set the image box to the .gif with the rope being drawn
                    animatedImage.Image = Properties.Resources.Frame4;
                    break;
                case 6:
                    //If the player has 6 guesses left, set the image box to the .gif with the base being drawn
                    animatedImage.Image = Properties.Resources.Frame5;
                    break;
                case 5:
                    //If the player has 5 guesses left, set the image box to the .gif with the head being drawn
                    animatedImage.Image = Properties.Resources.Frame6;
                    break;
                case 4:
                    //If the player has 4 guesses left, set the image box to the .gif with the body being drawn
                    animatedImage.Image = Properties.Resources.frame71;
                    break;
                case 3:
                    //If the player has 3 guesses left, set the image box to the .gif with the first arm being drawn
                    animatedImage.Image = Properties.Resources.Frame81;
                    break;
                case 2:
                    //If the player has 2 guesses left, set the image box to the .gif with the second arm being drawn
                    animatedImage.Image = Properties.Resources.Frame9;
                    break;
                case 1:
                    //If the player has 1 guess left, set the image box to the .gif with the first leg being drawn
                    animatedImage.Image = Properties.Resources.Frame10;
                    break;
                case 0:
                    //If the player has no guesses left, set the image box to the .gif with the last leg being drawn and a stylistic 'dead' emo
                    animatedImage.Image = Properties.Resources.LoseFull;
                    break;
            }
        }
```



```
private void Next_Click(object sender, EventArgs e) //Display the static hangman image and return to the gameplay interface
{
    //If the player has selected next, make the button invisible so they can't click next again
    nextButton.Visible = false;
    //Show all of the gameplay elements again
    MakeVisible();
    //Hide the image box for the animation
    animatedImage.Visible = false;
    //Disable the image box for the animation so that it stops looping
    animatedImage.Enabled = false;
    //Show the image box for the static image of the hangman
    staticImage.Visible = true;
    //Depending on the number of guesses left, a different .gif image will play and thus a different static hangman will display on the main gameplay interface
    switch (guessesLeft)
    {
        case 10:
            //If the player has 10 guesses left, set the image box to the static image with the beam as drawn in the .gif
            staticImage.Image = Properties.Resources.Static1;
            break;
        case 9:
            //If the player has 9 guesses left, set the image box to the static image with the next beam as drawn in the .gif
            staticImage.Image = Properties.Resources.Static2;
            break;
        case 8:
            //If the player has 8 guesses left, set the image box to the static image with the third beam as drawn in the .gif
            staticImage.Image = Properties.Resources.Static3;
            break;
        case 7:
            //If the player has 7 guesses left, set the image box to the static image with the rope as drawn in the .gif
            staticImage.Image = Properties.Resources.Static4;
            break;
        case 6:
            //If the player has 6 guesses left, set the image box to the static image with the base as drawn in the .gif
            staticImage.Image = Properties.Resources.Static5;
            break;
        case 5:
            //If the player has 5 guesses left, set the image box to the static image with the head as drawn in the .gif
            staticImage.Image = Properties.Resources.Static6;
            break;
        case 4:
            //If the player has 4 guesses left, set the image box to the static image with the body as drawn in the .gif
            staticImage.Image = Properties.Resources.Static7;
            break;
        case 3:
            //If the player has 3 guesses left, set the image box to the static image with the first arm as drawn in the .gif
            staticImage.Image = Properties.Resources.Static8;
            break;
        case 2:
            //If the player has 2 guesses left, set the image box to the static image with the next arm as drawn in the .gif
            staticImage.Image = Properties.Resources.Static9;
            break;
        case 1:
            //If the player has 1 guess left, set the image box to the static image with the first leg as drawn in the .gif
            staticImage.Image = Properties.Resources.Static10;
            break;
        case 0:
            //If the player has no guesses left, set the image box to the static image with the last leg and stylistic 'dead' emotion as drawn in the .gif
            staticImage.Image = Properties.Resources.StaticLose;
            /*Reveal to the player the target word that they failed to guess --> THIS IS DONE separately to the CheckForLoss function as that module
            is executed straight after the letter has been submitted, whereas this waits for the next button to be clicked (basically necessary for program timing*/
            correctWord.Visible = true;
            //Set the textbox which conveys this to say that THE WORD WAS <WORD>
            correctWord.Text = "THE WORD WAS " + targetWord.ToUpper();
            //Set the icon image to a different image as the purple textbox means that the icon image needed to incorporate the purple textbox (aesthetics)
            icon.Image = Properties.Resources.icon21;
            break;
    }
}
```



```
1 reference
public void CheckForLoss() //Check if the player has lost (no guesses left) and give condolence message
{
    //If player has no guessesLeft left (<1) then game is over
    if (guessesLeft < 1)
    {
        //Display loss condolence message
        letterGuessed.Text = "YOU LOSE";
        //As game is over, deactivate all buttons until player begins a new round by clicking play or reset
        DeactivateButtons();
    }
}
```

NEXT_CLICK()

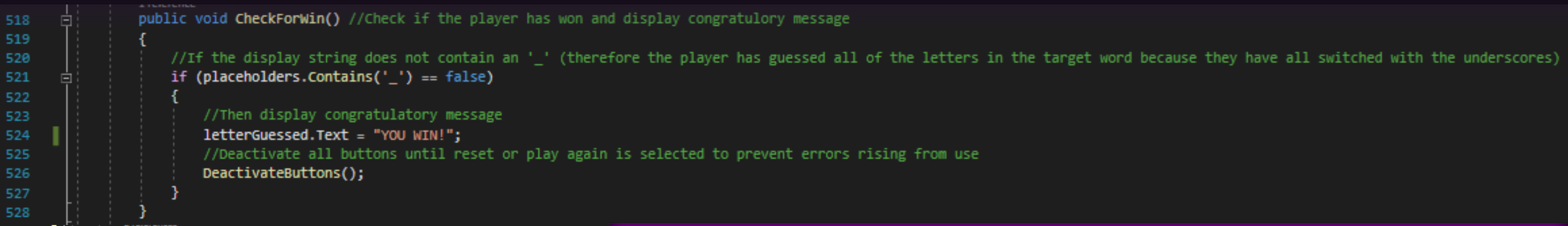
This button click event can only be activated following the PlayAnimation function, allowing this module to act as a sort of 'skip' or 'finish' cut-screen feature to return back to the gameplay. Once the player has watched the animated drawing adequately, once they click 'next' they will be able to make their next guess (or if they have lost, the keyboard will be disabled). Specifically, a static image of the stage of the hangman will display when they return back to the on-screen keyboard. This function achieves this effect by hiding the button once it has been pressed and making all of the gameplay objects visible again (returning the player to the 'guessing' screen). The animated image, which was previously playing, will be hidden and disabled to prevent the .gif from infinitely looping. A switch case is used depending on the number of guesses the player has left to indicate which stage the static hangman should be on. Each of these cases simply set the static image box (smaller image box) to the appropriate illustration. However, if the player has no guesses left then case 0 is executed, indicating the end of the game. This case simply sets the image to the 'dead' hangman drawn and reveals the correct word. The icon image is also required to change as the purple bar alters the interface.

CHECKFORLOSS()

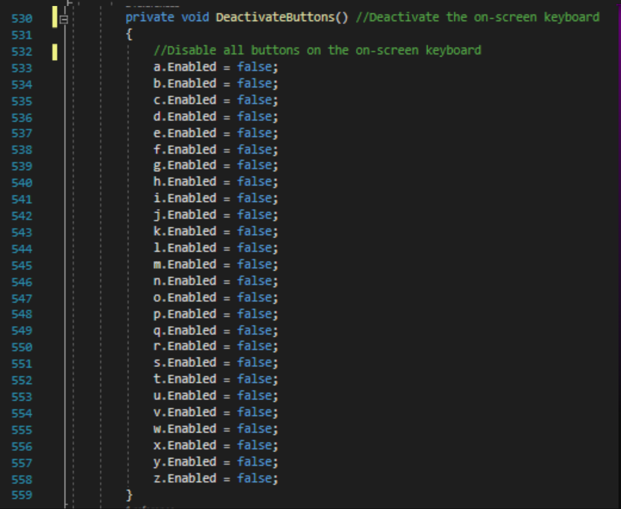
This module is called after the switch case occurs and the player has triggered the next button. This simply checks If the user has no guesses and then state YOU LOST and deactivates the on-screen keyboard. It is included separately for readability and code logic.

CHECKFORWIN()

This function is called after the play has made any guess to check if a win event has occurred. This is achieved by simply checking if the placeholder string does not contain any '_' which would mean that the player has successfully guessed all of the letters. The player is notified by receiving a congratulatory message 'YOU WIN!' in the textbox. In addition, the on-screen is disabled for interface consistency and to indicate that the game has been complete.



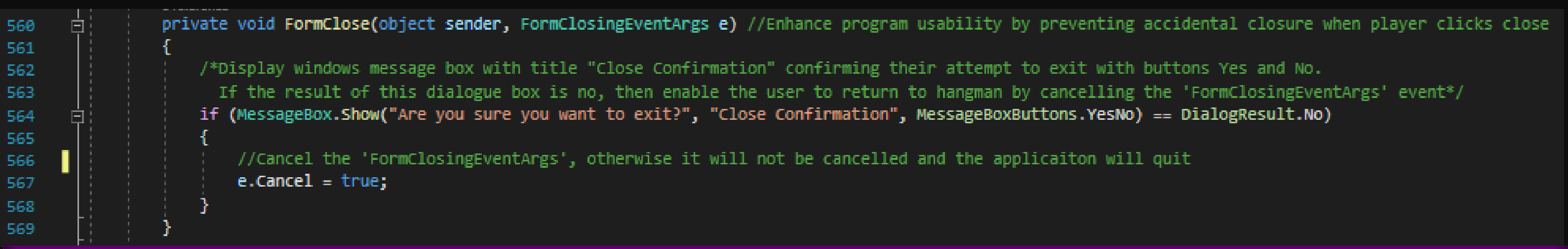
```
public void CheckForWin() //Check if the player has won and display congratulory message
{
    //If the display string does not contain an '_' (therefore the player has guessed all of the letters in the target word because they have all switched with the underscores)
    if (placeholders.Contains('_') == false)
    {
        //Then display congratulatory message
        letterGuessed.Text = "YOU WIN!";
        //Deactivate all buttons until reset or play again is selected to prevent errors rising from use
        DeactivateButtons();
    }
}
```



```
private void DeactivateButtons() //Deactivate the on-screen keyboard
{
    //Disable all buttons on the on-screen keyboard
    a.Enabled = false;
    b.Enabled = false;
    c.Enabled = false;
    d.Enabled = false;
    e.Enabled = false;
    f.Enabled = false;
    g.Enabled = false;
    h.Enabled = false;
    i.Enabled = false;
    j.Enabled = false;
    k.Enabled = false;
    l.Enabled = false;
    m.Enabled = false;
    n.Enabled = false;
    o.Enabled = false;
    p.Enabled = false;
    q.Enabled = false;
    r.Enabled = false;
    s.Enabled = false;
    t.Enabled = false;
    u.Enabled = false;
    v.Enabled = false;
    w.Enabled = false;
    x.Enabled = false;
    y.Enabled = false;
    z.Enabled = false;
}
```

DEACTIVATEBUTTONS()

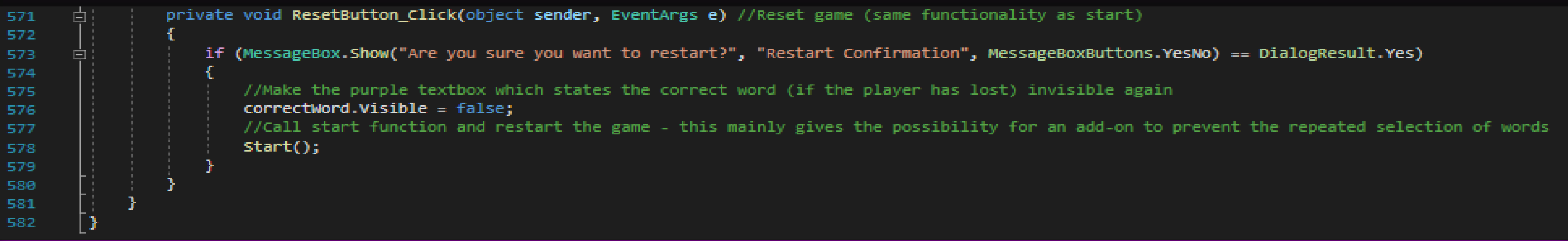
Designed to run when the player has one or lost, this module disables all of the buttons on the on-screen keyboard to emphasise that the game is over. This aesthetic feature also contributes to the consistency of gameplay as every time a letter is chosen, that button is disabled. This means it makes sense for the entire keyboard to be disabled when no more letters can be chosen.



```
        private void FormClose(object sender, FormClosingEventArgs e) //Enhance program usability by preventing accidental closure when player clicks close
        {
            /*Display windows message box with title "Close Confirmation" confirming their attempt to exit with buttons Yes and No.
              If the result of this dialogue box is no, then enable the user to return to hangman by cancelling the 'FormClosingEventArgs' event*/
            if (MessageBox.Show("Are you sure you want to exit?", "Close Confirmation", MessageBoxButtons.YesNo) == DialogResult.No)
            {
                //Cancel the 'FormClosingEventArgs', otherwise it will not be cancelled and the applicaiton will quit
                e.Cancel = true;
            }
        }
```

FORMCLOSE()

This event protects the exit button from being used accidentally and the program shutting down. While it wasn't specific in the program requirements, it makes the program interface more coherent as the restart button has a dialogue box pop-up so it only makes sense that closing the application is secured in a similar way. A dialogue box pops up and if the user selects yes, then the software will not be interception and close but if the user choses no, the close event will be cancelled and the application remain running.



```
        private void ResetButton_Click(object sender, EventArgs e) //Reset game (same functionality as start)
        {
            if (MessageBox.Show("Are you sure you want to restart?", "Restart Confirmation", MessageBoxButtons.YesNo) == DialogResult.Yes)
            {
                //Make the purple textbox which states the correct word (if the player has lost) invisible again
                correctWord.Visible = false;
                //Call start function and restart the game - this mainly gives the possibility for an add-on to prevent the repeated selection of words
                Start();
            }
        }
    }
}
```

RESTARTBUTTON_CLICK()

The final module of the program is the restart button as required in the project outline. This is a simple and sleek solution to the problem and continues the overall program aesthetic. When the restart button is clicked, to protect accidental use a dialogue box similar to the FormClose() module appears which asks "Are you sure you want to restart?". If the user selects Yes, then as a precaution the correct word which is appears if the player has lost disappears, and then the start function is called again. The game then restarts and application has successfully achieved all of the requirements detailed.

Implementation Errors

Implementation issues are runtime, syntax and logic errors which occur during development of the application code. During the programming stage, several errors were encountered particularly involving the visuals and interface which the algorithms could not sufficiently consider. **While the client stated reproduction of the errors to provide screenshots in this section was not necessary, the encountered errors are described below:**

Numerous logical errors occurred due to values which were accidentally set incorrectly in implementation. For instance, the display picture on incorrect guess behaved unexpectedly at first as the number of attempts was accidentally set to 13 instead of 11, creating a logic error. However, by returning to the code, this obvious error was noticed and adjusted appropriately to trigger at 11, resolving the issue. A similar issue occurred in the FailedGuess() function where it was accidentally set to -= 2, which caused the switch case to trigger incorrectly. Though this issue was more complicated than an incorrect value, another logic error occurred in the .Remove().Insert() methods where the letter would not be replaced. By engaging in extensive research into the two methods, it was revealed that the first value in the parameter of .Remove() is the starting position, then the second value counts how many is removed from that. This corrected the assumption that .Remove() removed from value 1 to value 2, changing from .Remove(position, position + 1) to .Remove(position, 1).

The most significant issue faced was likely the .gif images which unexpectedly looped. When the graphics were first implemented, this problem hadn't yet emerged as only the static hangman images had been inserted (and as the Gantt chart illustrates, placeholder images were used at first). However, during the development of the first ‘cut-screen’, the .gif infinite loop was exposed. At first, the gif was supposed to loop once then return to the gameplay screen, however, this issue meant that the player could not progress. At first solutions to this error involved altering the export settings of the .gif image file to loop ‘only once’. However, through thorough research, it was revealed that ongoing looping is an inherent feature in Windows Forms Applications. The obvious solution to this error was to implement code which simply made the gif disappear with a Thread.Sleep() after it had looped once. However, this problem was complicated by the fact that Thread.Sleep() from the System.Threading package cannot be used because the method blocks the user-interface thread. This means the gif is prevented from updating as well so to simply wait the duration of one .gif loop and then hide the picture box would cause the .gif to loop not play.

Ultimately, I discovered that the ‘enabled’ property of a picture box could pause a gif. This introduced the solution of adding a ‘next’ button that would turn ‘enabled’ to false when pressed. This meant the gif would keep looping until the player was ready to progress and prompted the implementation of the seamless cutscene idea. To further enhance the looping animation of the hangman being drawn, the .gif image file itself was adjusted to pause once the body part was drawn then add a ‘blinking’ animation before repeating. This enabled a much smoother visual effect. Although this issue posed a major complication during testing stages, it ultimately improved the user-experience past its original ideas.

Multiple other errors were faced during development. One of these was a logic error where the ‘_’ placeholders variable would continually combine in the textbox, rather than clearing. As mentioned in the ‘Ideas Generation’ and ‘Implementation’ sections, the placeholders variable is created by += ‘_’ for each letter in the target word. However, by inserting a breakpoint and following the value of the variables, it was exposed that the += meant it was continually adding to the previous string. By re-initialising the placeholders variable in the ChooseWord() function, this error was solved. Another issue was a syntax error where I had forgotten how to define a switch case. However, his was swiftly resolved by revising the syntax.

One final implementation error was faced in the code which replaced the placeholder with the letter if the player’s guess was correct. Originally, the spaces were inserted in the placeholder to create a string like: “_ _ _”, then the .Insert() and .Remove() methods were used. This would mean that .Remove(position, 2).Insert(position, $”{guessedLetter}” should have been used. However, this was not realised at first and a complicated for loop was designed – which did not work at all. To avoid this issue completely, instead the string was used with no spaces (“___”) and then the spaces were added in after seen in the AddSpacesForDisplay() function. Ultimately, this error actually led to a more readable solution.

Overall, several syntax and logic errors were encountered during the implementation of sthe code. However, runtime errors were generally minimal as the modules were guarded against these types’ errors – for instance – the algorithm structure made it impossible to submit an empty guess. The guidance provided by the algorithms and the structured approach allowed effective solutions to be reached, at times even bettering the existing code.

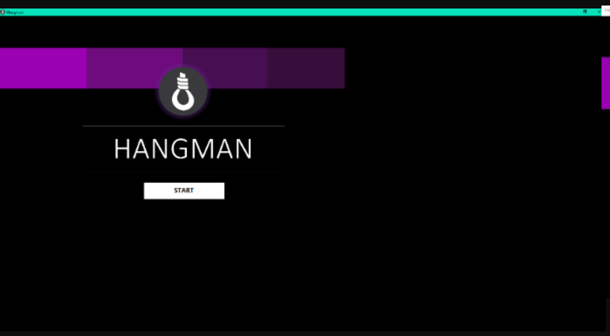
Testing of code

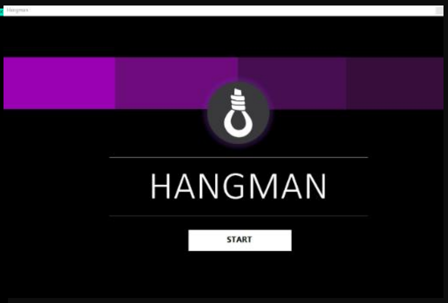
Throughout the development process of the application, continuous testing was conducted following the implementation of any new module once they appeared to be completed. Firstly, the game would be completely played through with the new module to ensure none of the modules conflicted with the added segment. This also confirmed that the module worked correctly. Once the module had been sufficiently tested for flaws, I employed the assistance of a family member to test its boundaries. As each distinct user has a unique approach to the game, this play-testing step is vital to any program. Majority of the time the module worked as expected given the well-planned interface and solution. However, if the module behaved unexpectedly or an error occurred, the code was firstly checked for any obvious mistakes. If the issue continued then a breakpoint was inserted. Three significant issues were discovered during the continuous testing of the code:

1. At first, an issue occurred during testing when the player could enter an empty guess selection. While the submit guess button would not let the player submit if an invalid guess was made, testing of the function found that it appeared in the user interface like a program error. The play-testers also agreed with this conclusion. Instead, the module was adjusted and the submit button was visually disabled when no guess was evident. While this required further refining to ensure that the button was disabled when there were actually no words, it was an accurate solution to the issue. This was solved by employing a technique where the project was duplicated and started in a completely different Windows Forms application. This reduced the pressure of designing a solution as there was a clear checkpoint to return to.

2. During the initial testing stages of the program, a major issue was discovered. Every target word up until testing by my family somehow involved words with no duplication of letters. However, during one of the first play-throughs the word ‘U M B R E L L A” was the target word. The testing exposed a critical flaw in the logic of the program as only one of the placeholders was replaced with an ‘L’. This happened because the position was originally stored in a single integer variable, so only the position of the last occurrence was stored and changed. After inserting a breakpoint to carefully follow the flow of data and understand the context of the issue, the use of a list became obvious. This list could then cycle through each position that matched, which would allow words with repeated letters to reveal the letters correctly.

3. A rather unique issue which was revealed during testing was the impact of full screen mode on the interface. From the thorough testing by the external testers, one had attempted to full screen the application. This produced a surprising result and, in effect, ‘broke’ the interface and its overall effect. Through thorough research into how to prevent Windows Form applications from being launched in full screen, the FormBorderStyle property was discovered. This was an ideal solution as it removed the full screen option all together (preventing it appearing like a program error) and created a memorable form application.





Testing

To effectively conduct testing before release, the finalised version of the modules should undergo rigorous testing with an extensive range of test data. If these functions had required a user input, this could have been tested with a desk check. However, performing a desk check is difficult as the modules are already guarded from invalid inputs. **Future improvements, the development of each module, and this justification for lack of test data is elaborated on below:**

Form1()

No test data is required to test the boundaries of this module as it only serves to call other functions. However, testing can be performed by executing the module and checking its behaviour, and it clearly works as intended.

INPUT	EXPECTED OUTPUT	OUTPUT
Application begins	Form loaded and the main gameplay elements are invisible	Form loaded and the main gameplay elements are invisible

MakeInvisible()

The use of rigorous test data is not applicable for this module as it simply makes interface elements invisible and there is no user input to test. However, testing can be performed by executing the module and checking its behaviour, and it clearly works as intended.

INPUT	EXPECTED OUTPUT	OUTPUT
Main gameplay elements are visible	Main gameplay elements are hidden	Main gameplay elements are hidden

StartButton_Click()

The use of rigorous test data is not applicable for this module as it simply reads a mouse click input. However, testing can be performed by executing the module and checking its behaviour, and it clearly works as intended.

INPUT	EXPECTED OUTPUT	OUTPUT
Start button clicked	Start function called successfully (indicated by start events executing)	Start function called successfully (indicated by start events executing)

Start()

The use of rigorous test data is not applicable for this module as it simple makes interface elements invisible and there is no input to test. However, testing can be performed by executing the module and checking its behaviour, and it clearly works as intended.

INPUT	EXPECTED OUTPUT	OUTPUT
GuessesLeft No elements displayed on the interface No guesses or gameplay active	Seamless 'loading screen' transition Set/reset guesses Prevent gif from looping in background Allow use of submit and delete button Prepare for gameplay interface	Seamless 'loading screen' transition Set/reset guesses Prevent gif from looping in background Allow use of submit and delete button Prepare for gameplay interface

LoadingScreen()

The use of rigorous test data is not applicable for this module as it simply calls other events and displays objects thus there is no input to test. However, testing can be performed by executing the module and checking its behaviour, and it clearly works as intended.

INPUT	EXPECTED OUTPUT	OUTPUT
Current program interface	Seamless loading screen transition Gameplay interface displayed for use	Seamless loading screen transition Gameplay interface displayed for use

MakeVisible()

The use of rigorous test data is not applicable for this module as it simply makes interface elements visible and there is no input to test. However, testing can be performed by executing the module and checking its behaviour, and it clearly works as intended.

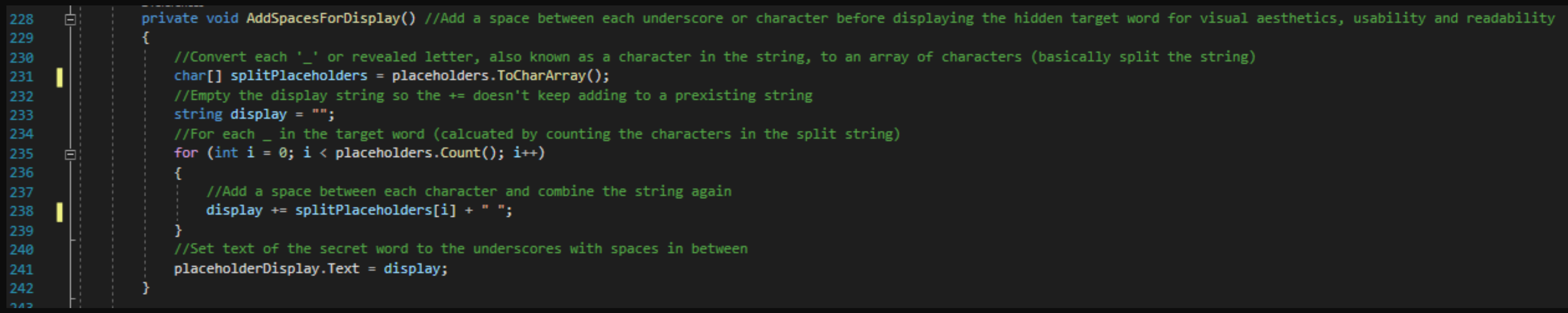
INPUT	EXPECTED OUTPUT	OUTPUT
All gameplay interface elements are invisible	All gameplay elements are displayed	All gameplay elements are displayed

SelectTargetWord()

The use of rigorous test data is not applicable for this module as it always reads in a word and there is no unexpected input to test. The main test data which would be used here was if no word was chosen, as creating the placeholders would not work. However – there will always be a word due to the structure of the program. This makes test data from a desk check of no use. Testing can however be performed by executing the module and checking its behaviour, and it clearly works as intended.

AddSpacesForDisplay()

DATA ITEM: TargetWord	EXPECTED OUTPUT	REASON FOR INCLUSION
“Bob”	“B o b”	Shorter word to test spaces are still inserted when there are more letters
“Daffodil”	“D a f f o d i l”	Longer word to test spaces are still inserted when there are more letters
“”	“”	Test if a program error occurs and no string is inputted to get spaces inserted



```
        private void AddSpacesForDisplay() //Add a space between each underscore or character before displaying the hidden target word for visual aesthetics, usability and readability
        {
            //Convert each '_' or revealed letter, also known as a character in the string, to an array of characters (basically split the string)
            char[] splitPlaceholders = placeholders.ToCharArray();
            //Empty the display string so the += doesn't keep adding to a prexisting string
            string display = "";
            //For each _ in the target word (calcuated by counting the characters in the split string)
            for (int i = 0; i < placeholders.Count(); i++)
            {
                //Add a space between each character and combine the string again
                display += splitPlaceholders[i] + " ";
            }
            //Set text of the secret word to the underscores with spaces in between
            placeholderDisplay.Text = display;
        }
```

placeholders	splitPlaceholders	Display	i	Placeholders.Count	splitPlaceholders[i]	placeholderDisplay.Text
“”	{}	“”	0	0	SKIPPED	“”

placeholders	splitPlaceholders	Display	i	Placeholders.Count()	splitPlaceholders[i]	placeholderDisplay.Text
“Daffodil”	{D,a,f,f,o,d,i,l}	~~“D ”~~	~~0~~	8	~~“D”~~	
		~~“D a “~~	~~1~~		~~“a”~~	
		~~“D a f ”~~	~~2~~		~~“f”~~	
		~~“ D a f f “~~	~~3~~		~~“f”~~	
		~~“ D a f f o “~~	4		~~“o”~~	
		~~“ D a f f o d“~~	~~5~~		~~“d”~~	
		~~“ D a f f o d i “~~	~~6~~		~~“i”~~	
		“ D a f f o d i l “	7		“i”	“D a f f o d i l “

placeholders	splitPlaceholders	Display	i	Placeholders.Count	splitPlaceholders[i]	placeholderDisplay.Text
“Bob”	{B,o,b}	~~“B ”~~	0	3	~~“B”~~	
		~~“B o ”~~	1		~~“o”~~	
		“B o b “	2		“b”	“B o b”

This rigorous data testing clearly exposes the potentials for an error if an empty word is chosen to have spaces inserted in between. However, this is prevented from ever occurring by the structure of the program as the module is never called unless placeholder has already been assigned. This was the main issue regarding thorough testing as the function is already guarded from an invalid input. This brief test data does, however, also illustrate the successful functioning of the module.

ActivateButtons()

The use of rigorous test data is not applicable for this module as it simply enables the interface elements visible and there is no input to test. However, testing can be performed by executing the module and checking its behaviour, and it clearly works as intended.

INPUT	EXPECTED OUTPUT	OUTPUT
All buttons are deactivated	All buttons on the on-screen keyboard are activated	All buttons on the on-screen keyboard are activated

Letter_Click()

The use of rigorous test data is not applicable for this function as it simply reads the letter pressed and converts it to character so there is no unexpected input to test. The fact that a letter must be clicked guards the function from the most likely error of no letter being entered (which would throw an error when converting an empty string). This means test data has little use. However, testing can be performed by executing the module and checking its behaviour, and it clearly works as intended.

INPUT	EXPECTED OUTPUT	OUTPUT
Button of letter played guessed	Text box displays the letter the player has chosen	Text box displays the letter the player has chosen

DeleteButton_Click()

The use of rigorous test data is not applicable for this module as it simply clears the textbox and there is no unexpected input to test. However, testing can be performed by executing the module and checking its behaviour, and it clearly works as intended.

INPUT	EXPECTED OUTPUT	OUTPUT
Delete button clicked	Remove any letter chosen so no guess is currently active	Remove any letter chosen so no guess is currently active

SubmitButton_Click()

The use of rigorous test data is not applicable for this module as it simply checks if there is text thus there is no unexpected input to test. However, testing can be performed by executing the module and checking its behaviour, and it clearly works as intended.

INPUT	EXPECTED OUTPUT	OUTPUT
Submit button clicked	Guess is made and processed.	Guess is made and processed.

SubmitGuess()

The use of rigorous test data is not applicable for this module as it is guarded against any potential errors. This is similar to the previous AddSpacesForDisplay() desk check, where an error would obviously be thrown if no letter was submitted to be guessed or if there was no target word. However, because of the structure of the application and how it disables the submit button if no guess is currently made, this testing is redundant. Further, the SelectTargetWord module is always called before the player can sub,it a guess, so there will always be a word to compare it against. However, testing can still be performed by executing the module and checking its behaviour, and it clearly works as intended.

INPUT	EXPECTED OUTPUT	OUTPUT
Letter player guessed Target word	If the guess was correct, reveal the letter in the placeholders string If the guess was incorrect, a hangman body part should appear and a guess left subtract Disable submit and delete button (as a guess was just submitted so no guess made yet) Loading face interface (to increase suspense)	If the guess was correct, reveal the letter in the placeholders string If the guess was incorrect, a hangman body part should appear and a guess left subtract Disable submit and delete button (as a guess was just submitted so no guess made yet) Loading face interface (to increase suspense)

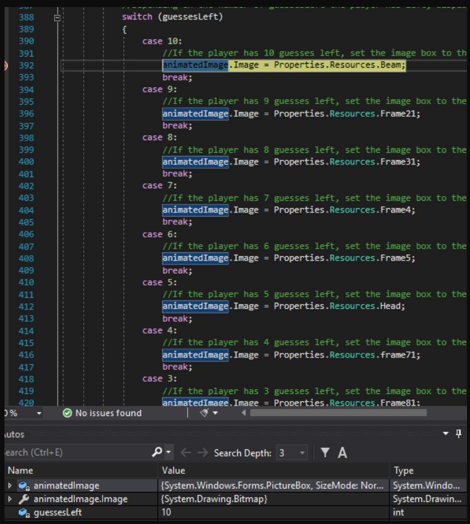
FailedGuess()

The use of rigorous test data is not applicable for this module as it simply subtracts 1 from guessesLeft then calls two functions so there is no unexpected input to test. However, testing can be performed by executing the module and checking its behaviour, and it clearly works as intended.

INPUT	EXPECTED OUTPUT	OUTPUT
Number of guesses left Loading screen interface	One less guess left Indicate a life was lost If the player has lost, display loss interface	One less guess left Indicate a life was lost If the player has lost, display loss interface

PlayAnimation()

The use of rigorous test data is not applicable for this module as it is a simple switch case which reads guessesLeft so there is no unexpected input to test. Additionally, the structure of the program means guessesLeft always has a value, for instance, if the program is reset, the number of guesses is reset too allowing the variable to continue working. However, testing can be performed by executing the module and checking its behaviour, and it clearly works as intended. Further, the graphics work successfully which indicate that the correct switch case has been entered. Inserting a break point serves to ensure that these cases work correctly.



```
switch (guessesLeft)
{
    case 10:
        //If the player has 10 guesses left, set the image box to th
        animatedImage.Image = Properties.Resources.Beam;
        break;
    case 9:
        //If the player has 9 guesses left, set the image box to the
        animatedImage.Image = Properties.Resources.Frame21;
        break;
    case 8:
        //If the player has 8 guesses left, set the image box to the
        animatedImage.Image = Properties.Resources.Frame31;
        break;
    case 7:
        //If the player has 7 guesses left, set the image box to the
        animatedImage.Image = Properties.Resources.Frame4;
        break;
    case 6:
        //If the player has 6 guesses left, set the image box to the
        animatedImage.Image = Properties.Resources.Frame5;
        break;
    case 5:
        //If the player has 5 guesses left, set the image box to the
        animatedImage.Image = Properties.Resources.Head;
        break;
    case 4:
        //If the player has 4 guesses left, set the image box to the
        animatedImage.Image = Properties.Resources.frame71;
        break;
    case 3:
        //If the player has 3 guesses left, set the image box to the
        animatedImage.Image = Properties.Resources.Frame81;
```

Name	Value	Type
animatedImage	{System.Windows.Forms.PictureBox, SizeMode: Nor...	System.Windo...
animatedImage.Image	{System.Drawing.Bitmap}	System.Drawin...
guessesLeft	10	int

INPUT	EXPECTED OUTPUT	OUTPUT
Loading screen interface	Animated 'drawing' cut-screen of the hangman body part according to number of guesses left 'Next' button for when user is ready to make their next guess	Animated 'drawing' cut-screen of the hangman body part according to number of guesses left 'Next' button for when user is ready to make their next guess

NextButton_Click()

The use of rigorous test data is not applicable for this module as the structure of the program means guessesLeft always has a value, so performing a desk check is redundant. However, testing can be performed by executing the module and checking its behaviour, and it clearly works as intended. Further, the correct graphic is displayed which indicates that the correct switch case has been entered. Inserting a break point serves to ensure that these cases work correctly.

INPUT	EXPECTED OUTPUT	OUTPUT
Next button clicked	Interface returns to gameplay interface Static image is included on the interface which changes according to number of guesses left Display loss interface (specifically once the next has been clicked)	Interface returns to gameplay interface Static image is included on the interface which changes according to number of guesses left Display loss interface (specifically once the next has been clicked)

CheckForLoss()

The use of rigorous test data is not applicable for this module as it simply checks if guessesLeft is less than 1 so there is no unexpected input to test. However, testing can be performed by executing the module and checking its behaviour, and it clearly works as intended.

INPUT	EXPECTED OUTPUT	OUTPUT
Number of guesses left	Provide condolence message Disable on-screen keyboard (as loss has occurred)	Provide condolence message Disable on-screen keyboard (as loss has occurred)

CheckForWin()

The use of rigorous test data is not applicable for this module as it simply checks if placeholders contain underscore and the only error would be if there was no underscore. The program is guarded from this error due to its structure however, as placeholders always contains a word. However, testing can be performed by executing the module and checking its behaviour, and it clearly works as intended.

INPUT	Provide congratulatory message	OUTPUT
Placeholders string	Disable on-screen keyboard (as win has occurred)	Provide congratulatory message Disable on-screen keyboard (as win has occurred)

DeactivateButtons()

The use of rigorous test data is not applicable for this module as it simply disables the interface element and there is no input to test. However, testing can be performed by executing the module and checking its behaviour, and it clearly works as intended.

INPUT	EXPECTED OUTPUT	OUTPUT
All buttons are activated	All buttons on the on-screen keyboard are deactivated	All buttons on the on-screen keyboard are deactivated

FormClose()

The use of rigorous test data is not applicable for this module as it simply displays a message box with two options so there is no unexpected input to test. However, testing can be performed by executing the module and checking its behaviour, and it clearly works as intended.

INPUT	EXPECTED OUTPUT	OUTPUT
Form closing	Protects form from being accidentally closed	Protects form from being accidentally closed

ResetButton_Click()

The use of rigorous test data is not applicable for this module as it simply displays a message box with two options so there is no unexpected input to test. However, testing can be performed by executing the module and checking its behaviour, and it clearly works as intended.

INPUT	EXPECTED OUTPUT	OUTPUT
Reset button clicked	Gameplay cannot be accidentally reset Restarts game	Gameplay cannot be accidentally reset Restarts game

Evaluation

Overall, I believe the thorough planning of the structured approach has enabled the developed product to effectively and efficiently fulfil the initial requirements. As demonstrated in the above 'Testing' section, the program successfully achieves its expected outputs. The produced solution and overall project can then be evaluated by comparing these results against the initial requirements defined in the 'identification of the problem' section. **These requirements were succinctly articulated into the following points in the 'Understanding and Defining Problem Section':**

→ Produce a working graphical user-interface game of Hangman, or alternatively Snowman, designed to run in a Windows form application

The proposed application undoubtably delivers on the project's purpose, creating a working graphical user interface game of the traditional Hangman. The software's mechanics outlined in 'functionality' is clear in the intuitive gameplay of the application, and the interface successfully serves to enhance the user-experience. Further, the Windows Form application contains distinct style and flair, creating an immersive gameplay experience. The test data serves to exemplify the functionality of the program, where performing a desk check typically has minimal use. This is because the modules are guarded to the point that any data which would even cause any errors cannot be entered.

→ Randomly select a word from a list of words residing in an external text file which the player must guess. This should be hidden.

As witnessed in the testing section, the module is guarded to most errors (providing an external file is excluded) due to the structure of the program so the program successfully accesses an external sequential file to read a list of words. This is achieved simplistically and efficiently with the File.ReadAllLines method which opens, reads the file into a string array and then closes the file, exposing minimal security threats. The system's solution to randomly selecting a word by generating a number which serves as a random index for an element in this list is clean and readable. Additionally, the use of a for loop to hide the target word beneath a separate string of underscores was an effective solution and the AddSpacesForDisplay makes it more readable. I believe this application successfully achieves this requirement in a clear and efficient way.

→ If the user guesses an incorrect letter, each body part is to appear or disappear (if Snowman was chosen)

Through the use of .gif's, an extremely distinct and memorable solution was produced to display the body parts of the hangman when an incorrect guess is made. The product of these animated images exceeded my original expectation and sufficiently fulfils this requirement. By using a simple switch case to change the contents of an image box, and by utilising the enable and disable properties of the picture box, the working of this feature is clean and refined. This clear function is evident in the PlayAnimation() test data, entering the correct switch case which SETs the image to the correct file. Combining the .gifs with the static image box created a seamless interface – allowing the player to immerse themselves in the game when a letter was guessed wrong but still be able to make a guess depending on how much of the hangman was left. The 'loading screen' also serves to enhance this intensifying effort. The test information on the PlayAnimation() module serves to support its effective solution. Overall, this requirement was exceeded, and I am extremely satisfied with the implemented solution.

→ If the user makes a correct guess, the letter/s must appear in their location in the chosen word

If the user makes a correct guess, by simply splitting the string of targetWord into an array, comparing the guessed letter against each character, and then adding any positions where they match to a list, this requirement is successfully fulfilled. As code efficiency is always an important consideration, the use of only methods .Remove() and .Insert() at each position serve to make the letter 'appear' in a clear and short way. As detailed in the 'test data' section, the module 'SubmitGuess' is guarded against errors due to the program's structure. Overall, this is a streamlined solution and the code itself remains readable– and readability is imperative for further maintenance.

→ Already used letters must be displayed on the screen

The application successfully implements this requirement in a clear and modern solution evident in the success of the test data. By incorporating the already used letters by disabling their use on the keyboard, the interface can remain decluttered and user will not get confused why a letter won't enter because it is clearly disabled. This system feature is rather unique from the original broad specification and contributes greatly to the memorability of the product produced.

→ If a player successfully solves the word, the game will provide a congratulatory message

Setting the letterGuessed textbox to 'YOU WIN!' is a clean solution which successfully conveys a congratulatory message. However, this feature has further potential for a .gif to be displayed, or the player skip to a cut-screen, but an appropriate graphic to match the aesthetic and keep the elegant interface was not achieved. Overall though, this potential is simply for aesthetics and the program requirement was successfully achieved.

→ If a player runs out of guesses and the final body part appears or disappears (if Snowman was selected) a condolence message will appear

The evaluation of the condolence message requirement is extremely similar to the congratulatory message. However, I believe that the unique static image of the hangman with a ‘dead’ emotion and the purple textbox which displays the correct word provides the level of aesthetics required for these win and lose events. In comparison to the congratulatory message, the execution of the condolence message exceeds the requirement, though both solve the original problem.

→ A button should be accessible that allows a new game to be played at any time but must be protected from accidental use

The protection for the reset button was well-executed, providing a noticeable yet clean Message Box which confirms the users choice. This is consistent if the user chooses to exit, where a message box requesting their confirmation appears. The testing section proves that the message boxes are successfully triggered. These are extremely clean and refined ways to achieve the protected restart button which work while seamlessly integrating another feature.

→ The project must follow the structured approach and its appropriate documentation in a PDR

The project successfully follows the structured approach and documents the process to great effect. This is evidenced in the produced Project Development Report.

The system performance clearly fulfills the original requirements laid out in understanding and defining with a unique and memorable flair. This evaluation is justified by the test data section which ultimately concludes that the modules are guarded against errors occurring originally. Providing the program is used appropriately and the external text file has not been adjusted, no particular security issues arise. Despite the limited portability of Windows Form applications, the engaging gameplay has the potential to be extended for further platforms with the existing codebase through use of frameworks. Further, the potential for networkability can overcome this restriction. While not included in the initial requirements, inclusivity and accessibility of the software to a broad range of backgrounds has been considered and potential opportunities identified for further celebration of diversity.

By using input data from only the mouse and the power of C# Windows Form Applications, it is undeniable that the project delivers a memorable solution. This program fulfils the requirements laid out in the ‘Understanding and Defining the Problem’ section and brings the traditional game of Hangman into the technological era.

MAINTENANCE OF THE SOFTWARE | PDR BODY

Maintenance is imperative to ensuring the longevity of the software application and is provided by the developer until no longer viable. The readability and clarity of the software solution produced considers this ongoing development and enables adjustments to be made easily. As the hangman project has a submission date deadline, ongoing support and extending features of the program does not have an established time frame. However, there is a number of instances where maintenance of the program is necessary and the appropriate methods should be taken. A number of these potentials have been already considered, for instance if the user upgrades their hardware, such as new monitor, the program interface will continue to open correctly as it has been created the size of a small screen resolution. Yet several others cannot yet be considered, such as when Microsoft releases an update to C#. Here, maintenance is necessary to ensure that the code is compatible to runtime. This can be tested by launching the software application in the updated version and resolving any complications or unexpected behaviours accordingly.

Another maintenance regard should be made to potential security issues, particularly if Microsoft discovers any security vulnerabilities with the code or elements used. For instance, if Microsoft discovers security vulnerabilities in C#, a new SDK and runtime will be released. This means that the application must be updated to use the new SDK. Similar adjustment applies to vulnerabilities in the program discovered through client response, where the system should be adjusted accordingly as well.

Ensuring the longevity of the software also includes potential upgrades and addons for the program, in particular considering the ongoing adjustments required to sustain player engagement. As mentioned in the 'Networkability' and 'Portability' sections, upgrades for this application could involve the networking and increase of platforms to create differing and unique versions of the gameplay. For example, by leveraging a web framework such as ASP.Net the program could implement networking to allow two players to play over the internet and simultaneously compete for a hidden word. The use of a web server which communicates with the players' client machines would enable the application's output to display on each monitor while obfuscating the code which could allow cheating. This topology is expanded on in the 'Networkability' section. In regards to the migration of the developed package to other hardware platforms, this could involve extending the existing codebase with the .NET developer platform Xamarin. This would extend portability to IOS and Android operating systems and with careful consideration of potential security threats, could ensure the relevance and longevity of the application. Further, the migration on to other platforms may require adjusting the interface, however, this can be seen as an opportunity to lead to more engaging and memorable gameplay ideas.

Add-on suggestions of the existing game itself could elaborate on the current cut-screen and seamless interface to enhance user-experience and engagement. The application could also be extended to add a hint button and currency system, increasing the substance of the game. Additionally, as noted in the 'Implementation' section, the solution has opportunity to implement a feature that prevents words being repeated as the target word, however, the large pool of 200 words should reduce the likelihood of this. Expanded on earlier in the report, the application provides opportunity for a hint and currency system to be implemented.

Overall, the readability and clarity of the software solution produced allows maintenance and further upgrades of the program to be completed with ease. While it is arguable that the use of a Windows Form application is outdated, the produced solution is versatile with rich potential to expand portability and networkability with the existing codebase. Consistent maintenance and upgrades of the software can ensure the ongoing longevity of the produced application, and ultimately enable the traditional game of hangman to continue to educate through the test of time.