Playing the Turing Game (Nag -21-266)

Final report

Submitted for the BSc in

Computer Science

April 2022

by

Callum Oliver Thomson Gray

Word count:

Abstract

The Turing game is a thought experiment created by Alan Turing to test whether a computer or Artificial intelligence (AI) is capable of thinking or exhibiting intelligent behaviour comparable to that of a human. This is done by having a human, the Questioner, ask questions to an unknown third party, the Subject, this third party is either another human or it could be an AI. The job of the Questioner is to determine whether they are talking to an AI or to another human. If the Questioner cannot correctly identify the Subject or is not certain of their identity then the Subject is said to have passed the Turing Test. The developer has investigated creating a proof-of-concept application to implement the Turing game concepts into an educational tool for use in a classroom environment.

This Paper aims to show via a working-proof-of-concept network-based application how an educational tool may be implemented that allows the students to be paired with either a human or AI and have a conversation which can be monitored remotely by the teacher.

**Keywords:**  Turing, AI, Artificial Intelligence, networking, client, server, chatbot

Contents

[Abstract i](#_Toc101032765)

[1 Introduction 3](#_Toc101032766)

[1.1 Background to the project 3](#_Toc101032767)

[1.2 Aims and objectives 3](#_Toc101032768)

[1.2.1 Objective 1 – Design and create an application that would act as the client connecting to the server application remotely 3](#_Toc101032769)

[1.2.2 Objective 2 – design and create an application to act as the server / Teacher’s application 3](#_Toc101032770)

[1.2.3 Objective 3 - design and create an application that can host a chatbot to act as the AI in the Turing game scenario 4](#_Toc101032771)

[1.3 Report Structure 4](#_Toc101032772)

[2 Literature review 5](#_Toc101032773)

[3 Requirements 6](#_Toc101032774)

[3.1 Product requirements 6](#_Toc101032775)

[3.2 Functional requirements 6](#_Toc101032776)

[3.2.1 Interfaces 6](#_Toc101032777)

[3.2.2 Functional Capabilities 6](#_Toc101032778)

[3.2.3 Data Structures/Elements 7](#_Toc101032779)

[3.2.4 Safety 7](#_Toc101032780)

[3.2.5 Reliability 7](#_Toc101032781)

[3.2.6 Security/Privacy 7](#_Toc101032782)

[3.2.7 Quality 7](#_Toc101032783)

[3.2.8 Constraints and Limitations 7](#_Toc101032784)

[3.2.9 Performance requirements 7](#_Toc101032785)

[4 Design 8](#_Toc101032786)

[4.1 Software design 8](#_Toc101032787)

[4.1.1 Client application Design 8](#_Toc101032788)

[4.2 Image design 20](#_Toc101032789)

[4.3 Design constraints 20](#_Toc101032790)

[5 Implementation and testing 21](#_Toc101032791)

[5.1 Implementation 21](#_Toc101032792)

[5.2 Testing 21](#_Toc101032793)

[6 Evaluation and discussion of results 22](#_Toc101032794)

[7 Conclusion 23](#_Toc101032795)

[References 24](#_Toc101032796)

[Appendix A – Interesting but not vital material 25](#_Toc101032797)

[Appendix B – Other things which may be useful 26](#_Toc101032798)

# Introduction

## Background to the project

The use of AI (Artificial Intelligence) In recent years has grown vastly and the chances are that most people will be using and interacting with AI without ever realising it. With this in mind teaching people about the concepts of AI is vitally important for them to understand this technology as it becomes more and more prevalent in everyday applications.

The Turing game is the name given to a thought experiment created by and named after Allan Turing. He devised this test as a means of evaluating whether an artificial intelligence (AI) could be considered conscious, or to be able to exhibit sufficient intelligence in order to compete with a human being. The Game involves a human known as the Questioner and one or more unknown third parties known as Subjects. The aim of the Questioner is to identify what the Subject is by asking questions and reviewing the responses. Based on the conversation that has occurred the Questioner now has to determine whether they are talking to a human or a computer. If the Questioner cannot say for certain or reliably identify the computer than the computer is said to have passed the Turing test.

In order to aid in teaching people about AI a network-based application could be used that allows students to connect with another random student or a chatbot. The student and partner would have a conversation at the end of which they would have to determine what they were talking to. The Teacher would be able to monitor the conversations that were happening remotely.

## Aims and objectives

### Objective 1 – Design and create an application that would act as the client connecting to the server application remotely

* Understand the concepts behind network-based applications and distributed systems
* Design a system that would allow for the sending and receiving of messages across a TCP connection asynchronously.
* Design a graphical user interface to be utilised by the application
* Implement an MVVM architecture pattern into the design for the User interface
* Implement the design and test

The first part of Objective 1 was to research and understand how a network-based application could be produced which would allow multiple end users to be connected and send messages to each other.

The second part was to take what had been learnt in the research component in order to design a framework that incorporated these principles

The third and fourth parts involved researching how to implement a graphical user interface (GUI) into the design and implementing the MVVM structure to achieve this.

The final part is to build a solution based on the design produced and test it using a simple console application that it can connect to in order simulate the connection to a server.

### Objective 2 – design and create an application to act as the server / Teacher’s application

* Design a system that would be able to allow the pairing up of two clients
* Expand the design to allow the system to receive and redirect connected clients to the available rooms
* Expand the design further in order to dynamically create rooms as required
* Implement an MVVM architecture pattern for the GUI
* Build and test the designed solution

The first part of Objective 2 is to design a concept for pairing the clients together in a “Room” and allowing messages to pass between them. The second part it to implement a design with a single room and a primary connection that could be used to receive the initial connection request and then redirect It to a room.

The third part involves expanding the solution design so that it can dynamically create and manage the “Rooms”

The fourth parts involved implementing the MVVM structure to the solution to create a GUI for the end user to work with.

The final part is to build a solution based on the design produced and test the application using the client application from Objective 1

### Objective 3 - design and create an application that can host a chatbot to act as the AI in the Turing game scenario

* Research chatbots and find some off the shelf solutions which could provide the chatbot functionality
* Design a program that can be used to wrap the chatbot and extract the functionality but allow it to connect to the server/Controller in the same manner as any other client
* Implement the design and test using the client and controller applications from the previous objectives.

## Report Structure

This Report will be structured in the following way. Section 2 Is made up of the Literature Review which discusses the background and current state of the fields with particular focus son Artificial intelligence, the inception and validity of the Turing game as the field has advanced and network-based communication. Section 3 will cover the Requirements for the project which will lead on to section 4 of the document that shall cover the design and conception of the project solution. Section 5 shall cover the Implementation and testing of the project. Section 6 is the conclusion and critical evaluation of this report. Section 6 will also aim to identify any short coming in the project and areas that can be improved upon in future if the development process was allowed a greater time period.

# Literature review

This section of the report will cover the current history and state of the domain that this project will be influenced by. This Project focuses on developing a series of applications that will allow multiple machines to connect and communicate to play the Turing Game. It is therefore important to understand What the Turing game is and the point behind it as well as the best way to achieve network communication between the applications.

The literature review is a survey of the history and state of the art in the domain of your project. It will summarize the work that has already been done in the field; this may be scientific literature, known techniques, and even previous student projects. It will provide a historical perspective on how the subject area has arrived at its current state by looking at important developments over time. If appropriate, it may examine existing software in the domain, especially in terms of the technology used and the features offered. The focus of the literature review is to summarise the existing arguments and ideas of others, identifying which are important.

A good literature review could be a project on its own, and form a very useful guide to anyone new to the particular field. It would identify the important work, authors and publications which would be a good place to begin research activity. Open questions and areas where new work is required would be discussed. Really good reviews are often published in scientific journals. Your literature review is not expected to be quite so substantial, but should still provide a comprehensive summary which will allow the reader to understand the field.

Images can be very useful here. Remember to attribute them properly to avoid accusations of plagiarism. Your literature review will naturally refer to lots of existing work, which must all be properly cited and referenced – see the ‘References’ section towards the end of this document.

Delete the red paragraphs and replace this one with your content (use the “Normal” paragraph style).

# Requirements

This project will use network sockets and an internet connection for communication between the various applications. This section of the document will cover the design decisions that were made and the rational behind those choices over potential alternatives.

## Product requirements

What will your software or hardware do? Who requires it? You might want to refer back to your aims and objectives to inform this section, and perhaps consider if they are still appropriate. UML use case diagrams are very helpful here (even for hardware).

The software produced because of this project is aimed at providing a teaching aid for teachers who wish to teach their students about AI and the Turing game. The idea behind this is that it would be being used in a computer room at a school or college with multiple machines accessing the same network to allow communication between students with a teacher monitoring the results.

The software is designed to run on a machine running a windows 10 operating system or later.

## Functional requirements

The exact content here will vary (especially if your project is hardware-based), but there are some standard items which you should consider including:

### Interfaces

### Functional Capabilities

The functional requirements of the solution are broken down into individual functional requirements for each Application that is produced as part of the solutions these are as outlined below

#### Client Application

|  |  |
| --- | --- |
| Ref | Requirement |
| TGF\_REQ\_CA\_01 | The Solution shall provide the capability to connect to the Controller Application |
| TGF\_REQ\_CA\_02 | The Solution shall provide the capability to send data to the Controller Application |
| TGF\_REQ\_CA\_03 | The Solution shall provide the capability to receive data from the Controller Application |
| TGF\_REQ\_CA\_04 | The UI shall provide the capability to display a message history |
| TGF\_REQ\_CA\_05 | The Solution shall have a clear and user-friendly User Interface |
| TGF\_REQ\_CA\_06 | The UI shall provide the capability for the user to enter text |
| TGF\_REQ\_CA\_07 | The Solution shall provide the capability for the user to submit their choice as to what they are speaking to |

#### Controller/Server Application

|  |  |
| --- | --- |
| Ref | Requirement |
| TGF\_REQ\_SA\_01 | The Solution shall provide the capability to connect to the Client Application |
| TGF\_REQ\_SA\_02 | The Solution shall provide the capability to send data to the Client Application |
| TGF\_REQ\_SA\_03 | The Solution shall provide the capability to receive data from the Client Application |
| TGF\_REQ\_SA\_04 | The Solution shall provide the capability to create virtual Rooms for the Client applications |
| TGF\_REQ\_SA\_05 | The Solution shall provide the capability to allocate Connected Clients to available rooms |
| TGF\_REQ\_SA\_06 | The Solution shall provide the capability to observe the data being shared in the rooms |
| TGF\_REQ\_SA\_07 | The Solution shall provide the capability to send data to the Chat bot Application |
| TGF\_REQ\_SA\_08 | The Solution shall provide the capability to receive data from the Chat bot Application |
| TGF\_REQ\_SA\_09 | The Solution shall have a clear and user-friendly User Interface |

#### Chatbot Wrapper Application

|  |  |
| --- | --- |
| Ref | Requirement |
| TGF\_REQ\_CB\_01 | The Solution shall provide the capability to receive text |
| TGF\_REQ\_CB\_02 | The Solution shall provide the capability to send Text |
| TGF\_REQ\_CB\_03 | The Solution shall provide the capability to answer questions |
| TGF\_REQ\_CB\_04 | The solution shall provide the capability to remember previous responses to questions within the current session |

### Data Structures/Elements

### Safety

### Reliability

### Security/Privacy

No personal data is stored as part of the solution and users are kept completely anonymous unless the user chooses to reveal any data about themselves, this data however is simply passed on to the end point and is not stored in any form upon cessation of the session.

### Quality

### Constraints and Limitations

### Performance requirements

# Design

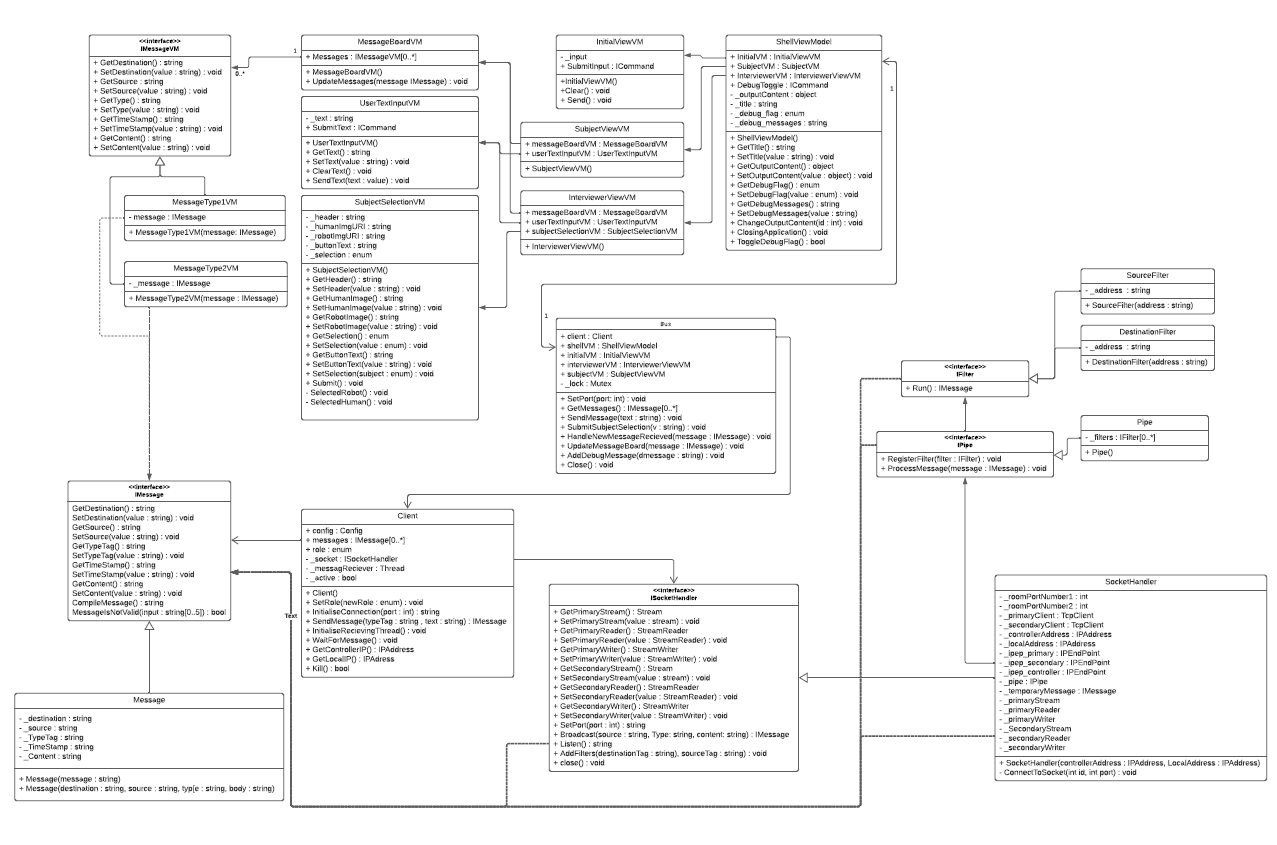
## Software design

This section of the report will cover the design of each of the three applications that make up the Turing Game proof of Concept the UML Diagrams are fairly large and have been scaled down however Full-Size Diagrams can be found in the Appendices of this document and will be referenced in their relevant section

### Client application Design

This section covers the design of the Client application, this section will include a UML (Diagram see Appendix B for the Full-size Diagram), A Breakdown of each of the Classes and their functionality as well as several sequence diagrams to demonstrate the concept of their operation.

#### Client UML Diagram



#### Client Classes

This section will cover the breakdown of the classes that make up the Client application and their functionality.

##### Bus

This is a static class that has been designed for use as a data interface to allow the View models to interact with the business logic without becoming Tightly Bound to the solution and vice Versa.

###### Variables:

**client** – This is a public variable used to store the current instance of the Client Class.

**shellVM** – This is a public variable used to store the current instance of the ShellViewModel Class.

**initialVM** – This is a public variable used to store the current instance of the IntitalViewVM Class.

**interviewerVM** – This is a public variable used to store the current instance of the InterviewerViewVM Class, it is unused when the client is set to Subject Mode.

**subjectVM** – This is a public variable used to Store The current Instance of the SubjectViewVM Class, it is unused when the Client is set to Interviewer Mode.

**\_lock** – This is a private Mutex variable that is used in order to promote thread safe data write access for the message board and other aspects of the solution with a multithreading component.

###### Functions:

**SetPort(port : int)** – This is a Function that will be used to initiate the Digital handshake between the Server and the Client, the integer that it takes as a parameter should be the User entered 4 digit session code displayed by the Controller application. It will call the Initialise Connection Function in the Client and upon return will set the mode of the Client based on the Returned Data either as an Interviewer or Subject before finally setting the Active view Model to the Corresponding Viewmodel.

**GetMessages() –** This Function is used to access the current List of messages in an abstracted fashion It returns the Current List of IMessage’s from the session.

**SendMessage(text : string) –** This Function is used to abstract the Functionality of sending the User input message to the server. It takes the user inputted message as the string parameter. It Calls the Client SendMessageFunction() passing the passed in string as a parameter.

**SubmitSubjectSelection(v : string) –** This Function is used to Send the Interviewers Selected Guess at who they are speaking to It has very similar functionality to the SendMessage Function however it filters the Message to have a specific Tag Type, It calls the Client SendMessage Function.

**HandleNewMessageRecieved(message : IMessage) –** This Function is used to abstract the Adding of a new Message to the Views message board. It Does this by taking in a IMessage derived Data Structure and calls the Update Message board function using the \_lock to control thread acces and prevent concurrency issues.

**UpdateMessageBoard(message:IMessage)** – This Function is used to update the Messageboard for the Active session it takes an IMessage Derived Data Structure as a parameter and adds it to the active view models Message board.

**AddDebugMessage(dmessage : string)** - This function is used as part of the Debug/testing and adds messages with time stamps to the Debug window of the application.

**Close() –** This Function handles the Close events and triggers the deconstructors/close operations o the child classes of the current Class.

##### Client

This Class is used to host the Back-end functionality and Business Logic of the Client application

###### Variables:

**config –** This is a public variable that is used to store Config data as a **Dictionary<string,string>**.

**messages –** This is a public ListIMessage Derived Classes.

**role –** this is an enum that is used to store the current role of the Client application (Interviewer or Subject).

**\_socket** – This is a private variable that is used to store an instance of an **ISocketHandler** derived Class.

**\_messageReciever** – This is a private variable used to store the instance of the thread that will be listening for messages.

**\_active –** This is a private Boolean that is used to control the action loops within the Client code when false all code loops should fall out and cease operation.

###### Functions:

**Client() –** This is the Constructor for the Client Class it initialises the variables and sets up the environment for the **\_messageReciever** Thread.

**SetRole(newRole : enum) –** This Function takes an enum as a parameter and is used to set the role variable of the Client Class.

**InitialiseConnection(port : int) –** This Function takes the controller port number as an int as a parameter and uses it to initiate connecting to the controller by calling the **SetPort()** function on the **\_socket** variable. It Returns a string that is received from the controller after the Handshake is complete.

**SendMessage(typeTag : string, text : string) –** This function is used to send The clients messages to the server via the **\_socket** variableit returns an **IMessage** derived class instance of the message that was sent.

**InitialiseRecievingThread()** – This Function initialises and starts the **\_messageReciever** Thread.

**WaitForMessage()** – This Function is the task that is targeted by the **\_messageReciever** Thread. It imply loops whilst checking the **\_socket** variable for a new message, when a new message is found it calls the **HandleNewMessageRecieved()** Function in the Bus Class passing in the received message

**GetControllerIP()** – This Function returns the Currently stored IP address for the Controller from the \_**config** variable.

**GetLocalIP()** – This function Finds the IP address of the Current Computer/ device that the Client application is running on.

**Kill()** – This Function is used to Terminate any active threads and break them out of any loops on application close.

##### Message

This class is used to parse and store the message data that the client both sends and receives. It inherits from the **IMessage** Interface.

###### Variables:

**\_destination –** This is a private variable used to store the destination data of the current **Message** instance

**\_source –** This is a private variable used to store the source data of the current **Message** instance

**\_typeTag –** This is a private variable used to store the type data of the current **Message** instance

**\_timeStamp –** This is a private variable used to store the time of creation for the current **Message** instance

**\_content –** This is a private variable used to store the actual contents of the current **Message** instance

###### Functions:

**Message(message : string) -**  This is a constructor for the Message Class that takes a single string with all of the message data separated by commas for a parameter. This is then split up and assigned to thew appropriate variables.

**Message(destination : string, source : string, type : string, body : string) –** This is another constructor that takes in all of the individual values as separate string and assigns them to the corresponding variables generating a new time stamp for the **\_timeStamp** variable.

##### Pipe

This class inherits from the **IPipe** interface. It is used to apply filters to messages before sending them and upon receiving them.

###### Variables:

**\_filters** – This is a private variable that is used to store a list of the various filters which are to be applied to messages.

###### Functions:

**Pipe()** – This is a constructor that instantiates the Pipe class.

##### SourceFilter

This is a class that derives from the **IFilter** Interface. It is used to check the source address value of the message and correct it if necessary.

###### Variables:

**\_address** – This is a private function that stores the address value to be applied to any messages which pass through the Filter.

###### Functions:

**SourceFilter(address : string)** – This is the constructor for the **SourceFilter** class.

##### DestinationFilter

This is a class that derives from the **IFilter** Interface. It is used to check the destination address value of the message and correct it if necessary.

###### Variables:

**\_address** – This is a private function that stores the address value to be applied to any messages which pass through the Filter.

###### Functions:

**DestinationFilter(address : string)** – This is the constructor for the **DestinationFilter** class.

##### SocketHandler

This class inherits from the **ISocketHandler** Interface. This class Handles the logic and functionality for connecting to the Controller using network Sockets. It uses two client one for writing and one for reading in order to maintain complete asynchronous communication.

###### Variables:

**\_roomPortNumber1 –** This is a private variable that stores the port number for the **\_primaryClient** to connect to.

**\_roomPortNumber2 –** This is a private variable that stores the port number for the **\_secondaryClient** to connect to.

**\_primaryClient –** This is a private variable that isused to store an instance of a **TCPClient** that will be used to send messages to the Controller.

**\_secondaryClient –** This is a private variable that isused to store an instance of a **TCPClient** that will be used to Receive messages from the Controller.

**\_controllerAddress –** This is a private variable that is used to store the IP Address of the Controller application.

**\_localAddress –** This is a private variable that is used to store the IP Address of the Current computer or device that is running the client application.

**\_ipep\_primary –** This is a private variablethat is used to store the primary IPEndpoint assigned to this client by the Controller. It is made up of the **\_controllerAddress** and the **\_roomPortNumber1**.

**\_ipep\_secondary –** This is a private variablethat is used to store the secondary IPEndpoint assigned to this client by the Controller. It is made up of the **\_controllerAddress** and the **\_roomPortNumber2**.

**\_ipep\_controller –** This is a private variable that is used to store the IPEndpoint of the Controllers Main Port. It is made up of the **\_controllerAddress** and the Controllers port number entered by the user.

**\_pipe –** This is a private variable that is used to storean instance of an **IPipe** derived Class.

**\_primaryStream –** This is a private variable used tostore the instance of the DataStream that is used by the **\_primaryClient.**

**\_primaryReader –** this is a private variable used to store an instance of a stream reader that is used to read data from the \_**primaryStream**.

**\_primaryWriter –** this is a private variable used to store an instance of a stream writer that is used to write data to the \_**primaryStream**.

**\_secondaryStream –** This is a private variable used tostore the instance of the DataStream that is used by the **\_secondaryClient.**

**\_secondaryReader –** this is a private variable used to store an instance of a stream reader that is used to read data from the \_**secondaryStream**.

**\_secondaryWriter –** this is a private variable used to store an instance of a stream writer that is used to write data to the \_**secondaryStream**.

###### Functions:

**SocketHandler(controllerAddress : IPAddress, LocalAddress : IPAddress) –** This is a Constructor for the class taking the IPaddress of the current machine and the controller as parameters.

**ConnectToSocket(id : int, port : int) –** This Function is Used to connect the primary and secondary clients to their respective ports the id parameter identifies which client variable is being connected and the port is the port number for that client to connect to.

##### ShellViewModel

This class is the View model for the main WPF window. It is used to store the view mode instances for the various nested views.

###### Variables:

**InitialVM** – This is a public variable that stores the current instance of the **InitialViewVM** Class.

**SubjectVM** – This is a public variable that stores the current instance of the **SubjectViewVM** Class.

**InterviewerVM** – This is a public variable that stores the current instance of the **InterviewerViewVM** Class.

**DebugToggle** – This is a public Variable that binds an ICommand, which is used to bind to an event in the view.

\_**outputContent –** This is a private variable that stores the instance of the current nested view.

**\_title –** This is a private variable that stores the title property for use in the view.

**\_debug\_flag** – This is a private variable used to store the visibility Enum for the debug window.

**\_debug\_messages –** This is a private variable that is used to store the debug messages that are sent to it these messages are separated by a newline character.

###### Functions:

**ShellViewModel()** – This is the Constructor for the Shell View model Class it instantiates the nested view models and adds them to the Bus class.

**GetTitle() –** This function is the getter function for use in binding the **\_title** property to a view.

**SetTitle(value : string) –** This function is the setter function it sets the **\_title** as well as updating the view.

**GetOutputContent() –** This function is the getter function for use in binding the **\_outputContent** property to a view.

**SetOutputContent(value : object) –** This function is the setter function it sets the **\_outputContent** as well as updating the view.

**GetDebugFlag() –** This function is the getter function for use in binding the **\_debug\_flag** property to a view.

**SetDebugFlag(value : enum) –** This function is the setter function it sets the **\_debug\_flag** as well as updating the view.

**GetDebugMessages() –** This function is the getter function for use in binding the **\_debug\_messsages** property to a view.

**SetDebugMessages(value : string) –** This function is the setter function it sets the **\_debug\_messages** as well as updating the view.

**ChangeOutputContent(id : int) –** This Function is used to change out the viewModel currently stored in the \_**outputContent** variable.

**ClosingApplication()** – This Function is called upon initiating the window.Close() event it calls the **kill()** function in the **Bus** Class.

**ToggleDebugFlag() –** This Function is used to Toggle whether the Debug window is visible or not.

##### InitialViewVM

This class is the View model for the Initial View presented upon start up of the Client application.

###### Variables

**\_input –** This is a private variable that is used to store user input property for use in the view.

**SubmitInput –** This is a public variable that is bound to an ICommand, which is used to bind to an event in the view.

###### Functions:

**InitialViewVM() –** This is the Constructor the **InitialViewVM** Class.

**Clear() –** This Function will clear the value stored in the \_**input** property and updates the view.

**Send() –** This Function is used to send the value of the **\_input** property to the Bus Class by calling Bus.SetPort(\_input).

##### SubjectViewVM

This class is the View model for the Subject View it handles the display logic and business logic interactions for the Subject view.

###### Variables:

**messageBoardVM -** This is a public variable that stores the instance of the **MessageBoardVM**.

**userTextInputVM -** This is a public variable that stores the instance of the **UserInputVM**.

###### Functions:

**SubjectViewVM() –** This is the constructor for the SubjectViewVM Class it is used to instantiate the instance.

##### SubjectViewVM

This class is the View model for the Subject View it handles the display logic and business logic interactions for the Subject view.

###### Variables:

**messageBoardVM -** This is a public variable that stores the instance of the **MessageBoardVM**.

**userTextInputVM -** This is a public variable that stores the instance of the **UserInputVM**.

###### Functions:

**SubjectViewVM() –** This is the constructor for the SubjectViewVM Class it is used to instantiate the instance.

##### InterviewerViewVM

This class is the View model for the Interviewer View it handles the display logic and business logic interactions for the Interviewer view.

###### Variables:

**messageBoardVM -** This is a public variable that stores the instance of the **MessageBoardVM**.

**userTextInputVM -** This is a public variable that stores the instance of the **UserInputVM**.

subjectSelectionVM – This is a public variable that stores the instance of the **SubjectSelectionVM**

###### Functions:

**InterviewerViewVM() –** This is the constructor for the InterviewerViewVM Class it is used to instantiate the instance.

##### MessageBoardVM

This class is the view model for the messageBoard view it handles the display logic and business logic interactions for the message Board.

###### Variables:

**Messages** – This is a public variable that stories a list of **IMessageVM** derived classes for use in the message board view.

###### Functions:

**MessageBoardVM()** – This is the Constructor for the **MessageboardVm** Class

**UpdateMessages(message : IMessage) –** This function is used to add new messages to the Messages List it takes a Message as a parameter and creates a new instance of an **IMessageVM** Derived class to display the message.

##### UserTextInputVM

This class is the View model for the user input region of the view it handles retrieving the user input and relaying it to the business logic as well as storing the model values for use in the view.

###### Variables:

**\_text –** This is a private variable that is used to store the store user input property for use in the view.

**SubmitText –** This is a public variable that is bound to an ICommand which is bound to a button click event in the view.

###### Functions:

**UserTextInputVM() –** This is the constructor for the **UserTextInputVM** Class.

**GetText() –** This function is the getter function for use in binding the **\_text** property to a view.

**SetText(value : string) –** This function is the setter function it sets the **\_text** property as well as updating the view.

**ClearText() –** This function clears the value of the \_**text** property and updates the view.

**SendText(text : value) –** This Function is bound to the **SubmitText** ICommand and will pass the user input value to the business logic of the application via the **Bus** Class.

##### SubjectSelectionVM

This class is the View model for the Subject selection region of the view it handles retrieving the user input and relaying it to the business logic as well as storing the model values for use in the view.

###### Variables:

**\_header –** This is a private variable used to store the header property for use in the view.

**\_humanImgURI –** This is a private variable used to store the file path of the human image for use in the view.

**\_robotImgURI –** This is a private variable used to store the file path of the robot image for use in the view.

**\_buttonText –** This is a private variable used to store the button text property for use in the view.

**\_selection –** This is a private variable used to store the Enum of the current user selection for use in the view.

**SubmitSelection –** This is a public variable that is bound to an ICommand this is used for binding click events in the view to functions in the view model.

Functions:

**SubjectSelectionVM()** – This is the constructor for the **SubjectSelectionVM** Class

**GetHeader()** **–** This function is the getter function for use in binding the **\_header** property to a view.

**SetHeader(value : string)** **–** This function is the setter function it sets the **\_header** property as well as updating the view.

**GetHumanImage()** **–** This function is the getter function for use in binding the **\_humanImgURI** property to a view.

**SetHumanImage(value : string) –** This function is the setter function it sets the **\_humanImgURI** property as well as updating the view.

**GetRobotImage()** **–** This function is the getter function for use in binding the **\_robotImgURI** property to a view.

**SetRobotImage(value : string)** **–** This function is the setter function it sets the **\_robotImgURI** property as well as updating the view.

**GetSelection() –** This function is the getter function for use in binding the **\_selection** property to a view.

**SetSelection(value : enum)** **–** This function is the setter function it sets the **\_selection** property as well as updating the view.

**GetButtonText()** **–** This function is the getter function for use in binding the **\_buttonText** property to a view.

**SetButtonText(value : string) –** This function is the setter function it sets the **\_buttonText** property as well as updating the view.

**Submit() –** This Function is Bound to the **SubmitSelection** ICommand it relays the user selection to the business logic via the **Bus** Class

**SelectedRobot() –** This function sets the \_**selection** Enum value and updates the button text to correspond to the selection made.

**SelectedHuman() –** This function sets the \_**selection** Enum value and updates the button text to correspond to the selection made.

##### MessageType1VM

This is the View Model for the messages sent from the current client it handles displaying the message data to the user.

###### Variables:

**Message –** this is a private variable that is used to storethe displayed instance of the Message class

###### Functions:

**MessageType1VM(message: IMessage) –** This is the constructor for the MessageType1VM class it takes an IMessage Derived class as a parameter and uses it to initialise the displayed values.

##### MessageType2VM

This is the View Model for the messages sent to the current client it handles displaying the message data to the user.

###### Variables:

**Message –** this is a private variable that is used to storethe displayed instance of the Message class

###### Functions:

**MessageType2VM(message: IMessage) –** This is the constructor for the MessageType2VM class it takes an IMessage Derived class as a parameter and uses it to initialise the displayed values.

#### Client Interfaces

This section will cover the breakdown of the Interfaces used in the design of the Client application and their functionality.

##### IMessageVM

This Interface is used to provide required functionality for the Message view models so that message views can be bound to them.

###### Functions:

**GetDestination() –** This function is the getter function for use in binding thedestinationproperty to a view.

**SetDestination(value : string) –** This function is the setter function it sets the destinationproperty as well as updating the view.

**GetSource() –** This function is the getter function for use in binding thesourceproperty to a view.

**SetSource(value : string) –** This function is the setter function it sets the sourceproperty as well as updating the view.

**GetType() –** This function is the getter function for use in binding thetypeproperty to a view.

**SetType(value : string) –** This function is the setter function it sets the typeproperty as well as updating the view.

**GetTimeStamp() –** This function is the getter function for use in binding thetimestampproperty to a view.

**SetTimeStamp(value : string) –** This function is the setter function it sets the Time Stampproperty as well as updating the view.

**GetContent() –** This function is the getter function for use in binding thecontentproperty to a view.

**SetContent(value : string) –** This function is the setter function it sets the contentproperty as well as updating the view.

##### IMessage

This Interface is used to provide required functionality for classes which will inherit from it

###### Functions:

**GetDestination() –** This function is the getter function it returns the destination value from the class.

**SetDestination(value : string) –** This function is the setter function it sets the destinationvalue.

**GetSource() –** This function is the getter function it returns the destination value from the class.

**SetSource(value : string) –** This function is the setter function it sets the destinationvalue.

**GetType() –** This function is the getter function it returns the destination value from the class.

**SetType(value : string) –** This function is the setter function it sets the destinationvalue.

**GetTimeStamp() –** This function is the getter function it returns the destination value from the class.

**SetTimeStamp(value : string) –** This function is the setter function it sets the destinationvalue.

**GetContent() –** This function is the getter function it returns the destination value from the class.

**SetContent(value : string) –** This function is the setter function it sets the destinationvalue.

**CompileMessage() –** This function compiles the 5 properties into aa single comma separated string and returns that value.

**MessageIsNotValid(input : string[0..5])** – this message takes a list of strings and checks if they can be made into a valid message returning a bool with the outcome.

##### ISocketHandler

This interface is used to provide the required functionality for classes that will inherit from it. In order to deal with socket communication.

###### Functions:

**GetPrimaryStream()** **–** This function is the getter function it returns the primaryStream value from the inheriting class.

**SetPrimaryStream(value : stream) –** This function is the setter function it sets the primaryStreamvalue.

**GetPrimaryReader()** **–** This function is the getter function it returns the PrimaryReader value from the inheriting class.

**SetPrimaryReader(value : StreamReader) –** This function is the setter function it sets the primaryReadervalue.

**GetPrimaryWriter()** **–** This function is the getter function it returns the PrimaryWriter value from the inheriting class.

**SetPrimaryWriter(value : StreamWriter) –** This function is the setter function it sets the primaryWritervalue.

**GetSecondaryStream()** **–** This function is the getter function it returns the secondaryStream value from the inheriting class.

**SetSecondaryStream(value : stream) –** This function is the setter function it sets the secondaryStreamvalue.

**GetSecondaryReader()** **–** This function is the getter function it returns the secondaryReader value from the inherited class.

**SetSecondaryReader(value : StreamReader) –** This function is the setter function it sets the SecondaryReadervalue.

**GetSecondaryWriter()** **–** This function is the getter function it returns the secondaryWriter value from the class.

**SetSecondaryWriter(value : StreamWriter) –** This function is the setter function it sets the secondaryWritervalue.

**SetPort(port : int) –** This function initiates the communication between client and controller initiating the Handshake, It does this by attempting to connect to the Controllers primary Port once done it will receive a message with two new port number in the body this is the primary and secondary port number of the room the client has been placed in The socket handler will now connect the primary and secondary sockets to the new port numbers and the handshake is complete.

**Broadcast(source : string, Type: string, content: string) –** This function is used to send a message from the client to the controller it does this using the primary writer to write a message to the primary dataStream.

**Listen()** – This function is used to Listen for new messagesby calling on thesecondaryReader to read data from the secondaryDataStream.

**AddFilters(destinationTag : string, sourceTag : string) –** This function adds filters to the Pipe class that is owned by the instance of the inherited class.

**Close() –** This function terminates any active threads or awaiting tasks such as connection attempts in order to release the resources on application close.

##### IFilter

This Interface is used to provide the required functionality to all filter classes that inherit from it.

###### Functions:

Run() – This Function runs the specified actions that the filter has been designed to apply and returning the modified IMessage derived Class.

##### IPipe

This filter allows for multiple pipes to inherit the same functionality in order to abstract and reproduce the functionality

###### Functions:

**RegisterFilter(filter : IFilter) –** This function takes an IFilter derived class as a parameter and adds it to the list of IFilter classes stored in the inheriting class.

**ProcessMessage(message : IMessage) –** This function loops through each IFilter in the list of IFilter Classes and calls the run command on each one returning the modified IMessage derived class as a result.

## Image design

If your project involves building hardware, give full details about the process here. Include diagrams as appropriate Use them strategically to illustrate points in your text. Remember that ‘a picture is worth a thousand words’ (we don’t apply this rule literally) but pictures on their own don’t explain everything.

If your project requires electronics and/or mechanical design, don’t forget to include that. Photos, CAD drawings, electronic schematics and other diagrams are welcome.

Delete the red paragraph and replace this one with your content (use the “Normal” paragraph style).

## Design constraints

You might include this in the next section if you prefer. Consider the limitations on how you are able to conduct your project. Relate the bounds (time and resources are obvious ones) which have an impact.

Delete the red paragraph and replace this one with your content (use the “Normal” paragraph style).

# Implementation and testing

Delete the red paragraphs and replace this one with your content (use the “Normal” paragraph style).

## Implementation

In this section you will describe what you did, and why you made the important decisions affecting your actions. It’s not a diary – don’t write a blow-by-blow account of every little thing that happened. Be selective and report those choices and techniques which made a difference. Make sure you discuss what options you considered. Explain how the criteria and methodology you used to select amongst different options (which tools are most appropriate, for example).

It may help to imagine that you are reading this project in the future, trying to replicate the work without making the same mistakes along the way. What would you need to know to make your job easier, and what is unimportant or obvious? Explain how you implemented the design in the previous chapter.

This is the place in which you would explain any novel or especially complex algorithms, data structures or systems you have used.

Make it clear what you have done, and what is pre-existing. For example, if you are using third party software libraries, describe how you have used them, and how they have benefited your project rather than simply what they do. If you have built on a framework, make it clear how you have developed new functionality.

Delete the red paragraphs and replace this one with your content (use the “Normal” paragraph style).

## Testing

If you are developing software or hardware, you must test it. This section should explain how your work will be (or has been) tested.

You should have a test plan at the very least (full details of it and its results if required can go in an appendix). Ideally, you will have automated tests for any software you build. You will also define user acceptance tests, or something similar which can be used to determine whether your output meets the requirements stated earlier. Explain how and when the tests should be conducted.

Delete the red paragraphs and replace this one with your content (use the “Normal” paragraph style).

# Evaluation and discussion of results

This section evaluates the *software (or other artefact)* you have developed. You should compare it with the original specification and see how well it satisfies the requirements. You may wish to refer back to your aims and objectives at this point. You should report the results of user testing and a summary of feedback if that has been collected.

If you have done experiments, then the results of these should be reported and discussed here.

If you have involved people in doing user evaluations, that information should be include here.

Delete the red paragraphs and replace this one with your content (use the “Normal” paragraph style).

# Conclusion

In this section you should evaluate the *project* as a whole, and draw conclusions from the work you have done. Ask yourself what the project has achieved – what is its contribution? Has it met its initial aims and objectives? If not, why? How does the work you have done enhance the field in general? What has been learned from the project? If you have a well defined research question, has it been answered? What do the results mean?

You should also use this section to reflect on the *process* by which you undertook the project. Was your methodology appropriate (and did you stick to it)? Was your time planning good? Did you complete the primary and secondary objectives, and if not then why? What have you learned from the process? What would you do better/differently if you had more time?

Sometimes, it’s appropriate to include a subsection on ‘Further work’, making suggestions of how to proceed and what could be done to enhance the project in future.

Delete the red paragraphs and replace this one with your content (use the “Normal” paragraph style).

References

References must be formatted in the correct manner. For this assignment you must use the University of Hull’ approved variant of the Harvard referencing style (Fallin 2019), fully described at https://libguides.hull.ac.uk/referencing/harvard. Note that the details of the expected format vary depending on the type of document being referenced. Make sure you are familiar with them. If you use reference management software such as Zotero, EndNote or RefWorks, then you should be able to export a table of references in the correct format, which will save you work.

Every reference should have at least one citation in the text. Most will probably be in the ‘Background’ or ‘Literature review’ sections.

Remember that there is a difference between references and a bibliography. You will certainly need references, but a bibliography is optional.

There is much more information and guidance about referencing on the library’s website at https://libguides.hull.ac.uk/referencing/home

Some examples, illustrating different types of source:

Bahraini, M.S., Bozorg, M., Rad, A.B., (2018). SLAM in dynamic environments via ML-RANSAC. *Mechatronics* 49, 105–118.

Fallin, L., (2019)*. LibGuides: Referencing your work: Harvard Hull.* Available online: http://libguides.hull.ac.uk/referencing/harvard (accessed 10/10/2019).

Janis, I., (1972). Victims of Groupthink: A psychological study of foreign-policy decisions and fiascoes. Houghton Mifflin, Boston.

Office For Students (2018) *. Securing student success: Regulation framework for higher education in England*. Available online: https://www.officeforstudents.org.uk/media/1406/ofs2018\_01.pdf (accessed 10/10/2019)

Schmuck, P., Chli, M., (2019). CCM-SLAM: Robust and efficient centralized collaborative monocular simultaneous localization and mapping for robotic teams. *Journal of Field Robotics* 36, 763–781.

Delete the red paragraphs and replace this one with your content (use the “Normal” paragraph style).

Appendix A – Interesting but not vital material

Appendices are used to include information which may be of interest but is not necessary for the reader. You do not have to include appendices if there is no need for them.

You might, for example, want to include some details of a particular piece of software (an API, perhaps) or hardware which your project uses. This might be something that a reader might wish to consult, but you wouldn’t want to include in the main body of the report. You could also put raw data from experiments in an appendix, or perhaps survey results. It should still be information of relevance, but nothing that everyone would be expected to read.

If you wish to refer to elements of your PID, you could include them in appendices.

Delete the red paragraphs and replace this one with your content (use the “Normal” paragraph style).

Appendix B – Other things which may be useful

You can have more than one appendix, or none at all. Give them meaningful names and titles (not the ones given here), so that you can refer to them in the text, and so that they appear in the table of contents.

Delete the red paragraphs and replace this one with your content (use the “Normal” paragraph style).