**Eliza Project Report Sid Ramachandran & Alicja Wladyslawska [210015419 &** **220019540] 08/02/2023**

**Overview**

Our task was to write a chat engine that could take input from the user, identify key words within the sentence, and then compose a human-like reply that could simulate a real interaction while staying on topic of the conversation.

To do this, we created 3 scripts – each representing a separate character, which would include rules and keywords that would help create a meaningful reply. The Eliza engine could select a suitable reply and implement parts of the user input to create a relevant and human output, almost simulating a conversation with a psychotherapist[1].

The main goal of this project was to be able to manipulate input of users from the command line, and be able to efficiently load in a text file and read the correct data from it to create a good response.

**Design**

One of the first design choices of the project was to decide the third personality of our chat engine.

The two prerequisites for this project were a psychotherapist and an eight-year-old child – two polar opposites from each other. Our main goal here was to find a distinct third personality that could have its own quirks, while still being a realistic personality to interact with.

The third personality we decided to implement was a long-term acquaintance or friend – we imagined that the setting of the conversation would be a ‘catch up’ after a period of time, which allowed us to implement casual conversation, which came from another adult. This created a whole new personality which didn’t have the ‘professional’ qualities of a psychotherapist, but also didn’t have the often times unserious interactions of a child. That way, the replies could seem a lot more casual and personal – as the conversation could be viewed as a close friend giving advice and being able to joke about.

***The design of the script files***

To efficiently read and identify information from the script files, they had to follow an order that could clearly indicate different pre and post substitution rules.

The script files are neatly sectioned by keywords, with the beginning containing welcome / quit commands and replies. There is also contained a section with all pre-substitution rules and post-substitution, allowing replies to be calculated and generated with ease.

Each keyword is sectioned off into one or more decomposition phrases that in turn can choose a different reassembly rule, often accompanied by additional similar replies so that the chat engine isn’t repeating itself constantly.

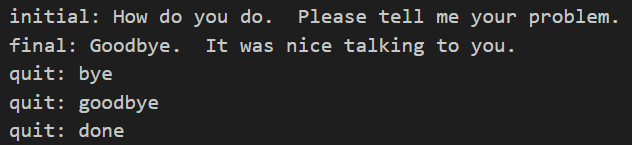


Figure 1 Initial and Quit

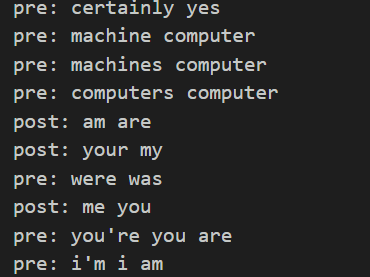


Figure 2: Pre Post

Text

Description automatically generated

Figure 3: Decomp rule and reconstruction string.

***Synonyms***: We had also decided to go further into this project and implemented detection of synonyms within our script files. Catching out synonyms from the user input is an effective way of making the conversation seem more human-like, as synonyms of keywords may not be caught, and would thus receive a different reply.

Text

Description automatically generated

Figure 4: Synonyms

In the above example, the following words in each line are all synonyms of each other. If one was caught out in the line, then the keyword would be directed to the correct output in the script.

***Memory***: A simple memory feature is also attempted. When a decomposition string has a $ then when the user inputs a key that calls that string, it will also be stored in a memory class.

***Goto feature***: To limit the number of keys, for keys that are identical like “alike” & “similar”, a simple goto feature is implemented in the script.

**Key Priority:** There is a priority number tracked for each key. When an input has several key words, the response will pertain to the key with the highest priority.

**Word substitution:** The reassembly string has the following tag %#%. To place an input word or string in there. The string is identified by the \* in the reconrule.

***Advantages of the Script Design***: The line by line aspect of the script design to map to each tag for the program to process simplified the reading and object mapping of the script’s contents. The Keys are also granularized so the input is scanned for a matching key word and the decompose rule works from there which makes the script intuitive and readable and anyone can take the script and modify with minimal training.

***Limitations of the Script Design:*** The simplicity of design creates the limitations on the number of key words that can be included in a real life scenario. In real life application, the synonyms should ideally look up a thesaurus and the key words should do semantic matching to find the closest meaning rather than just looking at exact matches.

***The design of the program***

The design has a wrapper class, Eliza that calls the ElizaProcessor class. ElizaProcessor class houses all the operations and logic for reading the script and crafting the response. I chose this model since the wrapper class provides an easy and simple interface while hiding the complex implementation details within the ElizaProcessor class. This improves readability and usability of the software. This model follows the Facade Pattern, which comes under the creational pattern, where Eliza acts as the façade that delegates the actual work to the Eliza processor object without exposing the creation logic [2]. The interface to the ElizaProcessor class is through the readScript() and runProgram() methods accessed through the newly created Eliza object.

Since ElizaProcessor has to processes several objects of interconnected ArrayLists, I created separate classes to handle the different functions and used the delegation pattern to assign task to other classes[3]. This helps with inheritance and readability of the program.

Tracking the Key to the decomposition string and to the reassemble string was a bit complex. Initially, we wrote a simple app that had a set of arrays for each Key, and had a matching array with a reply[4]. The Pre and Post were additional arrays that switched the prepostions as needed. This restricted us in building the 1 X n relationships between Key and Decompositon rule and Decomprule to Reassembly String. We did some research online and found code that were using deprecated versions of Vectors and extending the Vector class which is no longer recommended[5][6]. Then we changed the design to split them into two different lists and updated the script to support it as well. We used ArrayLists to store the object instances and the object model of each class to store the key and pointer to the next object/list. This simplified the code and made it fast and efficient.

The diagram below shows the logical data flow in the program:



Figure 5 a. Logical Data Flow Diagram

*UML Diagram*

Modelio gave an error while trying to reverse the relationships, so we have done it manually. This seems to be a known error in Modelio 5.1 [7].

Figure 1 shows the UML diagram with my design:

Diagram

Description automatically generated

**Figure 5: UML Diagram**

**Testing**

Sample test scripts for the different scenarios are given in the table below:

|  |  |  |  |
| --- | --- | --- | --- |
| Class/ Feature tested | Test case | Expected Outcome | Actual Outcome |
| Script File | Check if the program can read the script file. | If successful, no error should be displayed and Eliza should greet the user. | >> Hello.  I'm Eliza. I'm 8. Would you like to chat? |
| Key match | Ask a query with a key = toy to an 8 yr old | Relevant follow-up response | >> Do you like toys?  Toys are my best friends. Do you like toys? |
| Key priority | When two key words are used, system responds based on one with higher priority. Computer has 50 and Toy has 20. | Response is about computers not toys | >> Toys are great but do you like computers  I love computers. They have fun games. |
| Change response when script is changed | Run with therapist script and ask Do you like toys? | Response will acknowledge the like professionally. | >> I love toys  You say you love toys ? |
| Word substitutions | Ask a question with a noun in a reassmb string that has a %2%. | Response has the noun. | >> My family is big  Tell me more about your family. |
| Synonyms | User says a synonym word sister instead of family | Response pertains to family | >> My sister is tall  Who else in your family is tall ? |
| Goto phrases | Use a keyword no one and it will forward to everyone | Response is from everyone. | >> no one remembers me  Really, no one ? |
| Memory | Use an input with a My and then type some gibberish later | Response will come back to my from memory | >> My big red dog.  Your big red dog ?  >> ieuqwopeu  Let's talk more about your big red dog. |
| Different responses different scripts | Input a question with computer to a friend | Reply is about watching shows etc | >> My computer is ancient  Do you watch shows? |

Test Outputs:

A screenshot of a computer

Description automatically generated with medium confidence

Figure 6: 8 Yr old

Text

Description automatically generated with low confidence

Figure 7: 8 Yr old Matching a key word

Text

Description automatically generated

Figure 8:8 Yr old Multiple key words

Text

Description automatically generated

Figure 9: Therapist interaction for toys

Graphical user interface, text

Description automatically generated

Figure 10: Therapist Input word in output

Text

Description automatically generated

Figure 11: Therapist Synonym sister = family

A screenshot of a computer

Description automatically generated with low confidence

Figure 12: Friend goto no one to everyone

Text

Description automatically generated

Figure 13: Friend :Memory recall:

Text

Description automatically generated

Figure 14: Friend: computers – different responses

No known bugs or defects identified.

**Evaluation**

We believe our program managed to efficiently replicate an Eliza engine.

Our Eliza engine can:

* Efficiently receive an input from a user
* Apply pre-substitution rules to the text to allow a successful read from the program.
* Decompose the sentence to find the keyword and identify the context around it
* Create a relevant and human-like reassembly sentence.
* Apply post-substitution rules to have the sentence make perfect sense and be readable.

All of these texts’ manipulations combined create a tool that can maintain a conversation with a human in real-time, and deal with the inputs it gives.

We decided to expand on our project by creating a list of synonyms, making our code more robust and human-like. Synonyms allow for keywords detected in the text to have responses more suited to them, by grouping them with words that have similar meaning instead of going to a default response when the word isn’t understood by the engine.

We were also very satisfied with using a version control repository during our work. Version control allowed us to remote work on projects very efficiently, making us able to monitor progress and divide tasks between us efficiently.

The link to the version control repository can be found here: [link to repo]

**Conclusion**

We found this project to be a very interesting look into NLP and how AI is constructed. It is very useful to be familiar with the bare bones of Artificial Intelligence and to recognise how human behaviour was ‘replicated’ in the early days of computers.

It is evident that Eliza is of an older era – some replies may be difficult to understand, and conversations may often seem to lead nowhere. After looking into the Turing test and how it proceeds, I can see how participants of the experiment could easily identify between the human interaction and the machine.

Pair work is a new experience for both of us in software development, and it was very interesting getting to know version control and discussing plans for software creation together. It is a very important experience that will be useful in the workplace, where often teams of software engineers work and communicate on a single project together, helping each other out to create a final solution in the end.

If given additional time, we would love to expand on the human-like interactions of Eliza by doing semantic matching and expanding the memory capability. Either by fleshing out the scripts with more responses and keywords, or implementing even different ways of creating replies, it would be very interesting to see how Eliza could be improved to fit our modern generation of AI communication between humans.

Total word count: 1979.

**References**

[1] https://studres.cs.st-andrews.ac.uk/CS1006/Lectures/L01-eliza.pdf

# [2] Design Patterns Explained: A New Perspective on Object Oriented Design, 2nd Edition

by [Alan Shalloway](https://www.amazon.com/Alan-Shalloway/e/B001IXPWYW/ref=dp_byline_cont_book_1), [James Trott](https://www.amazon.com/s/ref=dp_byline_sr_book_2?ie=UTF8&field-author=James+Trott&text=James+Trott&sort=relevancerank&search-alias=books)

[3] https://en.wikipedia.org/wiki/Delegation\_pattern

[4] https://vaadin.com/blog/building-a-chatbot-in-java

[5] <https://github.com/codeanticode/eliza>

[6] <https://codereview.stackexchange.com/questions/252051/extending-class-vector-in-java>

[7] <https://www.modelio.org/forum/9-extensions/524-solved-javadesigner-reverse-engineering-error.html>