

401I - Final Year Project

Final Report

Voice Recognition RPG

Baron Khan (bak14)

# Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Abstract</b>                                   | <b>1</b>  |
| <b>2</b> | <b>Introduction</b>                               | <b>2</b>  |
| <b>3</b> | <b>Background</b>                                 | <b>4</b>  |
| 3.1      | Voice Recognition in Games . . . . .              | 4         |
| 3.1.1    | In Verbis Virtus . . . . .                        | 4         |
| 3.1.2    | Skyrim Kinect . . . . .                           | 4         |
| 3.1.3    | Star Trek Bridge Crew . . . . .                   | 5         |
| 3.1.4    | Classic Zork on Alexa . . . . .                   | 6         |
| 3.2      | Limitations of Zork . . . . .                     | 6         |
| 3.3      | Voice Recognition Implementations . . . . .       | 7         |
| 3.3.1    | Tazti - Speech Recognition for PC Games . . . . . | 7         |
| 3.3.2    | PocketSphinx for Unreal Engine 4 . . . . .        | 8         |
| 3.3.3    | Houndify . . . . .                                | 8         |
| 3.3.4    | Watson Conversation Engine . . . . .              | 9         |
| 3.4      | Natural Language Processing . . . . .             | 11        |
| 3.4.1    | Stages of Natural Language Processing . . . . .   | 11        |
| 3.4.2    | Part-of-Speech Tagging . . . . .                  | 13        |
| 3.4.3    | Slot Filling . . . . .                            | 13        |
| 3.4.4    | Synonyms and Hypernyms . . . . .                  | 14        |
| 3.4.5    | Semantic Similarity . . . . .                     | 15        |
| 3.4.6    | Generating Sentences . . . . .                    | 16        |
| 3.5      | WordNet . . . . .                                 | 17        |
| 3.6      | Object Properties . . . . .                       | 17        |
| <b>4</b> | <b>Design</b>                                     | <b>18</b> |
| 4.1      | Target Platform . . . . .                         | 18        |
| 4.2      | Java Programming Language . . . . .               | 18        |
| 4.3      | System Overview . . . . .                         | 18        |
| 4.3.1    | Speech-to-Text . . . . .                          | 19        |
| 4.3.2    | POS Tagging . . . . .                             | 20        |
| 4.3.3    | Slot-filling Matching . . . . .                   | 20        |
| 4.3.4    | Semantic Similarity . . . . .                     | 22        |
| 4.3.5    | Context-Action Mapping . . . . .                  | 23        |
| 4.3.6    | Intent Execution . . . . .                        | 24        |
| 4.4      | System UML Class Diagram . . . . .                | 24        |
| 4.4.1    | MainActivity Class . . . . .                      | 25        |
| 4.4.2    | VoiceControl Class . . . . .                      | 26        |
| 4.4.3    | VoiceProcess Class . . . . .                      | 26        |

|          |   |           |
|----------|---|-----------|
| 4.4.4    | GlobalState Class . . . . .                       | 26        |
| 4.4.5    | ContextActionMap Class . . . . .                  | 26        |
| 4.4.6    | SemanticSimilarity Class . . . . .                | 27        |
| 4.5      | System Features . . . . .                         | 27        |
| 4.5.1    | Synonym Mapping . . . . .                         | 27        |
| 4.5.2    | Ignoring Incorrect Matches . . . . .              | 27        |
| 4.5.3    | Confirmation and Suggestions . . . . .            | 27        |
| 4.5.4    | Multiple Commands . . . . .                       | 28        |
| 4.5.5    | Multiple Targets . . . . .                        | 28        |
| 4.5.6    | Sentence Mapping . . . . .                        | 28        |
| 4.6      | Game Design Overview . . . . .                    | 29        |
| 4.6.1    | Game Flow . . . . .                               | 29        |
| 4.7      | Game Design UML Class Diagram . . . . .           | 31        |
| 4.8      | Room Generation . . . . .                         | 32        |
| <b>5</b> | <b>Implementation</b>                             | <b>33</b> |
| 5.1      | Voice Recognition Interface . . . . .             | 33        |
| 5.2      | Graphical User Interface . . . . .                | 33        |
| 5.3      | Voice Recognition System Implementation . . . . . | 35        |
| 5.3.1    | GlobalState.java . . . . .                        | 35        |
| 5.3.2    | VoiceProcess.java . . . . .                       | 35        |
| 5.3.3    | AmbiguousHandler.java . . . . .                   | 38        |
| 5.3.4    | SemanticSimilarity.java . . . . .                 | 38        |
| 5.3.5    | CustomLexicalDatabase.java . . . . .              | 40        |
| 5.3.6    | ContextActionMap.java . . . . .                   | 41        |
| 5.3.7    | Entity.java . . . . .                             | 42        |
| 5.3.8    | Action.java . . . . .                             | 43        |
| 5.3.9    | MultipleCommandProcess.java . . . . .             | 44        |
| 5.3.10   | SentenceMapper.java . . . . .                     | 44        |
| 5.3.11   | Synonym Mapping Implementation . . . . .          | 46        |
| 5.4      | Role-playing Game Implementation . . . . .        | 46        |
| 5.4.1    | GameState.java . . . . .                          | 46        |
| 5.4.2    | Battle Mode . . . . .                             | 46        |
| <b>6</b> | <b>Appendix</b>                                   | <b>48</b> |
| 6.1      | Android Application Screenshots . . . . .         | 48        |
| 6.1.1    | Battle Mode Example . . . . .                     | 48        |
| 6.1.2    | Multiple Commands Example . . . . .               | 49        |
| 6.2      | Skyrim Kinect Command List . . . . .              | 51        |
| 6.3      | Penn Treebank Tagset . . . . .                    | 53        |
| 6.4      | System UML Class Diagram . . . . .                | 54        |
| 6.5      | Game Design UML Class Diagram . . . . .           | 55        |

|                   |  |           |
|-------------------|--|-----------|
| 6.6               | Choosing the Slot-filling Grammar . . . . .            | 56        |
| 6.7               | Finding the Best Action in VoiceProcess.java . . . . . | 57        |
| 6.8               | JwiLexicalDatabase.java . . . . .                      | 59        |
| 6.9               | OpenObject.java . . . . .                              | 62        |
| 6.10              | MultipleCommandProcess.executeCommand() . . . . .      | 63        |
| 6.11              | BattleContextActionMap.java . . . . .                  | 64        |
| <b>References</b> |  | <b>65</b> |

# 1 Abstract

When adding voice commands to a video game, a developer may find themselves hard-coding text strings for phrases that a player can say, and mapping them to intents within the game. It becomes more work to include varying versions of the same intent as the developer would have to include every permutation of the phrase (for which there could be infinitely many).

This project presents a text-based role-playing game (RPG) on Android which allows the player to issue actions within the game using their voice. The game uses a new voice recognition system that aims to make it easier for the developer to add voice commands without hard-coding the phrases, and with no cloud processing required.

Various methods for improving the voice recognition system are explored and evaluated, with the aim of decreasing the developer workload. These include using semantic similarity methods, adaptive and user-controlled learning mechanisms, and various other NLP techniques. The voice recognition system is also applied to other domains such as video conferencing solutions, and cooking applications.

## 2 Introduction

In recent years, video games have explored various forms of input that diverge from the traditional control schemes of a keyboard and mouse, or an analogue stick and buttons on a controller. Touch-screen controls and motion controls have had varying degrees of success over the years, and provide new forms of interactions with games. A recent form of input used in video games has been voice controls; a user utters a sentence or phrase and this would execute an action or intent within the game.

Voice recognition and control schemes have been used in various ways in the medium, whether in tandem with traditional control schemes, or as the only form of input. Most games which feature a voice control scheme are typically programmed to work with a specific set of keywords or phrases, hard-coded by the developer. This is usually the case with games where the voice control interface is entirely optional to user, as it requires little to no input from the programmer; if the speech processing is handled by a separate library, the developer just has to map a text string (e.g. "open the door") to the function that executes the intent (e.g. the function that opens a nearby door in the game).

It becomes increasingly difficult for the developer to add more phrases that can be accepted in the game and map to the same intent (e.g. "push the door open") as they would have to hard-code each possible input that the user could possibly say: having too few phrases would mean that the user could become frustrated when their preferred way of stating an intent is not accepted by the game, while adding too many acceptable phrases would increase development time.

Natural Language Processing (NLP, also referred to as Natural Language Understanding) has been used to improve inference of what users were trying to say, and to extract meaning from the user's phrases and sentences. For instance, if the user says, "push the door open", NLP can be used to infer that the user's intent is to open a door. Using NLP, several user phrases can be mapped to the same intent without having to hard-code each possible phrase.

NLP is already used to improve voice recognition in popular personal assistants such as Siri, but these systems offload the NLP workload to the server to reduce the amount of local processing required, as NLP requires a sizeable amount of machine learning and pattern recognition [1]. Most video games do not require an internet connection to play, and therefore can be played anywhere, so it may not be possible to use these cloud services for adding voice recognition to games. In other applications such as embedded systems or remote robotics, no internet connection can be established, so the only option is to hard-code the acceptable phrases.

The goal of this project is to apply NLP to a voice control scheme used within a role-playing game (RPG), in order to reduce the amount of work required by a developer to add a flexible voice control interface, and to give more freedom of expression to the player while they play the game.

Role-playing games are a genre of video games that involve the player controlling a character in a world featuring exploration and/or battle mechanics, along with a progression system, where the character gains new abilities or items as the game progresses. These include turn-based games such as Pokemon and Final Fantasy, or more action-based games such as Dark Souls or Skyrim. In these games, the player possesses several items that they acquire during the game, and accesses them by navigating menus.

If the player has many different items, they may need to spend a lot of time searching the menus for a specific item, which can be quite tedious. It can also become quite repetitive if the user is constantly only pushing the "Attack" button over and over again with no variation in the attack used. Using voice recognition and NLP, a player could just simply say a phrase such as, "use a potion" or, "attack with axe", and the item will be used without having the player navigate several menu screens.

Another motivation for this project is to explore how voice controls can allow games to become more accessible to people with certain disabilities that prevent them from using traditional controllers or a keyboard and mouse. Very few popular games support voice recognition as a form of control, whether it is consoles, PC or mobile.

A text-based RPG will be created that will use a flexible voice control interface. It will feature a simple exploration mechanic, with the user manipulating objects and environments using voice commands, and a simple battle mechanic. These will be designed to fully demonstrate the power of the voice-control system to make the game easier to play, as opposed to using traditional mouse clicks and button presses.

Note that this project is not concerned with the real time digital processing required to convert speech to text; it is assumed that the speech from the microphone is already converted to a text string using an existing tool such as Google's Speech-to-Text API. The focus is on the integration and usability of voice control schemes.

The proposed design for the RPG and system aims to achieve the following goals:

- Allow a developer to add voice recognition to their game with little effort.
- Create a simple natural language processing system that can work offline.
- Evaluate the effectiveness of using different semantic similarity methods.

## 3 Background

### 3.1 Voice Recognition in Games

Below is a list of notable recent games that have incorporated voice control functionality as a major component of the interaction experience.

#### 3.1.1 In Verbis Virtus

In Verbis Virtus [2] is an independent 3D fantasy adventure game developed by Indomitus Games for Windows. It requires the player to solve puzzles in a 3D environment and battle enemies using magic.

When a microphone is connected, the player is able to cast spells using their voice by saying specific phrases defined by the game. For example, to cast a spell that produces a floating light source to brighten a room, a user must say, "let there be light", or if the user wishes to shoot an energy beam from their hand, they must say, "Beam of Light", and so. These phrases appear to be hard-coded into the game.

There are several limitations with this implementation of voice control. Firstly, since the phrases are hard-coded, there is no lenience in variations of the phrases, e.g. "create a light" or "Laser of Light", etc. This can seem mundane and no different to a user pressing the same button repeatedly to cast a spell.

Another limitation is that the user is still required to use a keyboard and mouse to perform other actions in the game, such as moving around, looking around, and navigating menus and message boxes. The voice control interface cannot be used to perform these actions. The voice control interface itself does not seem integral to the experience, as the user could just simply press a key that performs the spell instead of saying the same phrase over and over again.

This voice recognition interface could be improved by allowing the player to perform any action within the game using their voice, as well as allowing for more variation in what the user can say for each action.

#### 3.1.2 Skyrim Kinect

The Elders Scrolls V: Skyrim is first-person action role-playing game developed by Bethesda [3]. The Xbox 360 version of the game supports the Microsoft Kinect peripheral which allows the player to execute voice commands [4]. The user can use over 200 hundred pre-configured voice commands, and these can be used simultaneously with the traditional control scheme.



There are voice commands available that allow the user to navigate the game’s menu screens. For instance, the player would say, ”quick items” to open their inventory menu. Afterwards, the user can open menus for specific categories of items such as ”potions”, ”books”, etc. However, it doesn’t seem to be possible to actually use any of the items selected with only a voice command from the menu; this is only possible if you assign the item to a ’hotkey’ (e.g. by saying, ”assign health potion” once you have highlighted a potion item), and then saying ”equip health potion” during the game. See the Appendix 6.2 for a full list of available voice commands.



Figure 1: Voice commands that are available within the items menu in Skyrim.

This system, despite being optional to the player, helps to reduce the time spent searching through the menu screens. However, it appears that this system also hard-codes the phrases that can be said by the player with no flexibility.

### 3.1.3 Star Trek Bridge Crew

Star Trek Bridge Crew is a virtual reality game developed by Ubisoft. This game allows the player to issue voice commands to AI crew members on a spaceship [5]. In order to develop a more interactive and realistic experience, the development team used IBM Watson’s interactive speech capabilities. This API allows commands to be delivered in a variety of ways, as the speech is parsed for its meaning using Watson’s Conversation service. For example, a player could say, ”show me the ship’s status report” or they could say something drastically different such as ”damage report”

to execute the same command [6]. This gives the user more freedom in how they convey their request to the AI, giving the player a "new level of sense of presence" [7]. IBM's Watson API is described in more detail later in this section.

The voice recognition design used in this game is similar to the proposed design outlined in this report. The user's intent should be extracted from the phrase that they speak, so similar phrases should map to the same intent. Unfortunately, Watson's API is a paid cloud service; so the game requires a constant internet connection to play.

### **3.1.4 Classic Zork on Alexa**

Zork is a trilogy of classic text-based role-playing games that support a free-form style of input. The player is provided with an input scenario (e.g. "You are standing in an open field west of a white house, with a boarded front door. There is a small mailbox here."), and then types commands that they want to execute (e.g. "open mailbox" or "attack troll with sword") [8]. The exploration part of the proposed game in this report will be loosely based off Zork's exploration mechanics of interacting with objects in the surroundings.

These commands could be delivered using a speech-to-text interface to add voice control to Zork. A group of developers created a port of Zork running on Amazon's Alexa Skills Kit [9]. This allows the player to deliver commands using their voice as if they were typing the phrases into a terminal. However, this implementation still suffers from the limitations of Zork's free-form input.

## **3.2 Limitations of Zork**

The exploration mechanics of the proposed design in this report will be based on Zork's general gameplay of interacting with objects in the environment to solve puzzles, and the limitations of Zork's free-form input are discussed here.

Zork supports a number of text commands, which usually take the form of a verb-noun structure, such as "use <noun>" or "take <noun>", but it also supports some more sophisticated commands with complex structures, such as "give all but the pencil to the nymph" or "drop all except the dart gun" [10].

Zork's free-form input still has some limitations. For example, it still cannot accept all variations of specific intents. For instance, below is a list of accepted and rejected commands if the user wishes to open a mailbox in front of them:

- "open mailbox" - Accepted

- "open the small mailbox" - Accepted
- "can i open the mailbox" - Rejected
- "check the mailbox" - Rejected
- "find out what's in the mailbox" - Rejected

Below is a list of results for closing the mailbox:

- "close mailbox" - Accepted
- "shut mailbox" - Rejected
- "closing mailbox" - Rejected
- "close the red mailbox" - Rejected

Clearly the rejected text commands could be valid phrases that express an intent to open some sort of mailbox. Zork has a pre-defined list of verbs and sentence structures that it can accept, and sticks to those without much flexibility.

Since most of the supported commands take the form of a verb-noun structure, this could be used in the proposed design as the general structure of commands expected by the user.

### **3.3 Voice Recognition Implementations**

There are many tools and services available that allow developers and users to add voice recognition to video games, with some discussed below:

#### **3.3.1 Tazti - Speech Recognition for PC Games**

Tazti is a keyboard mapping tool available to players to be used with any type of PC game that uses the keyboard [11]. The user sets up a profile for each game separately by mapping speech commands to one or more keys. For instance, the user could say "fire" or "shoot fire" and this would map to the keystrokes required to cast a fire spell.

While this tool is not directly integrated into games by the developer, it has the advantage of being able to work with almost any PC game that uses a keyboard, such as role-playing games, first-person shooter games, and even platforming games. However, the voice commands are limited to being hard-coded, so there is no way of varying the speech commands slightly, even if the meaning is the same. It is also not freely available and requires a license to be purchased.

### 3.3.2 PocketSphinx for Unreal Engine 4

*Sphinx-UE4* is a speech recognition plugin for the Unreal Engine 4 game engine [12], based on the PocketSphinx library which provides offline speech-to-text [13]. The plugin allows the developer to specify keywords and the commands that they map to (e.g. "turn right", "enable sprint", "kick the ball", etc).

The plugin also supports grammar files. A grammar contains the grammatical structure that an accepted input can take, such as,  $\langle digit \rangle \langle operation \rangle \langle digit \rangle$ . This will accept input such as, "three add five", "six minus four", "two times twelve", and so on.

### 3.3.3 Houndify

Houndify is a paid cloud service developed by SoundHound that allows developers to add voice recognition and control to any application they wish [14]. The API takes as input either a text string or audio samples and returns a JSON string containing the response to the input. This response can range from anything: from home automation control, to weather updates, and more. The response would be in the form of an intent - a single word representing the action to be executed. For example, any attacking phrase would map to an intent called, "ATTACK".

As well as having built-in voice commands already for specific domains (such as all weather commands or sports update commands), Houndify also supports the creation of custom voice commands by the developer. To create a custom command, the expression to be said by the user is specified using a syntax similar to regular expressions, but instead specifies the general phrase structure (and with very different syntax), and is mapped to an intent [15].

Below is an example of a possible expression for attacking a troll:

"attack" . "troll"

Here, the player would have to say, "attack troll" to execute the intent. This can be expanded to have more variation in how the user can say this command:

("attack" | "hit") . ["the"] . "troll"

Here, the player can say phrases such as, "attack the troll" or "hit troll". The expression can be further expanded. However, the expression will eventually become long and confusing:

("attack" | "hit" | "damage") . ["the"] . ["big" | "large"] . "troll" . [("with" | "using") . ["a"] . ("sword" | "hammer" | "axe")]

Even though this expression allows for more variation in what the user can say, it still doesn't cover all the possible variations of specifying an attack, and this long expression is only for one intent; many more expressions would have to be written for other intents, and this can become tedious for the developer.

While this API would be useful for building a personal assistant similar to Apple's Siri or Amazon's Alexa, it does not seem feasible for this project due to the time required to create a flexible voice recognition scheme. This is also a paid service, so users are charged for each query they make.

### **3.3.4 Watson Conversation Engine**

IBM's Watson API features a Conversation service that allows developers to build voice recognition interfaces that understand natural language input [16]. The developer defines the training data to be used by the API (in order to train a natural language classifier) using 'intents' and 'entities' [17].

'Intents' are the goals that a user will have as they interact with the application. For example, in a voice-controlled car, an intent could be 'wipers\_on', in order to activate the windscreen wipers. This intent would be paired with example utterances in the form of text strings, such as "turn on the wipers", "switch on the windscreen wipers", and so on. The more examples given, the more accurate the trained model will be.

#return

⊕ Add a new user example...

☐ exchange

☐ I'd like to return this

☐ I don't want this anymore

☐ I need you to take this back

Figure 2: An example of an intent to return something, along with examples of phrases for the intent. Image re-used from an IBM blog post [18].

'Entities' are classes of objects that help to provide context to intents. For example, in the above example, it is not clear whether the user is referring to the wipers on the front of the car or the wipers on the back of the car (the API will assume the front by default). If the entity is included in the phrase spoken by the user, then the context of the intent becomes clear.

@returnItems

|   |       |       |                |
|---|-------|-------|----------------|
| ⊕ Add a new value                       |       |       |                |
| <input type="checkbox"/> book           | text  | tome  |                |
| <input type="checkbox"/> parrot         | bird  | macaw | Norwegian Blue |
| <input type="checkbox"/> video cassette | movie | tape  |                |

Figure 3: An example of entities for items that can be returned. Entities that have been grouped together (usually synonyms) are placed on the same row [18].

Once the classifier is trained, it will be able to use the examples of intents provided to classify whether new examples of phrases have the same meaning as those. For example, if the user says, "activate the windscreen wipers at the front of the car", the classifier will return the 'wipers\_on' intent, even though this input is very different in structure to the examples provided above. A demo is available to test this [19].

This API is very flexible and matches the first objective of this project; only a few example of possible inputs need to be provided for each intent, and then the classifier can infer whether any new input strings will match the same intent. Unfortunately, despite being a very powerful system, this still remains a paid cloud service, so users are charged per API call. It also requires a lot of processing power (which is offloaded to a cloud server) for the many machine learning algorithms being used here. However, a simpler system may possibly be implemented that uses a similar concept to the intents and entities framework, and the proposed design in this report uses a similar concept that runs locally.

According to a research paper by McCord, Murdok and Boguraev on *Deep Parsing in Watson RRR*, the engine consists of two deep parsing components: an English Slot Grammar (ESG) parser and a predicate-argument structure (PAS) builder. These allow Watson to produce parse trees of a sentence and extract pattern-based relations from it, such as question decomposition, hypothesis generation, and evidence scoring.

There is a similar API called DialogFlow [20], which works very similarly to IMB's Watson Conversation engine. However, it suffers from the same issues as above (e.g. cloud-based, pay-per-request, etc).

## 3.4 Natural Language Processing

Natural Language Processing (NLP) forms a major component of this project. This will give the developer the ability to take an arbitrary string of text and infer its meaning, and map it to an intent. Below are some explanations on NLP concepts related to this project.

### 3.4.1 Stages of Natural Language Processing

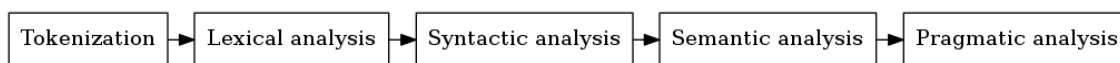


Figure 4: The general stages of processing a text string for its meaning using NLP.

NLP can be broken down into six stages [21]. The first stage is the tokenisation, where a raw text string is broken down into words (usually separated by spaces).

The second stage is the lexical analysis, where the output of the tokenisation process is improved upon by looking at words that can be taken apart even further (or rejoined if needed) to uncover more information. These include words such as: clitic contractions (e.g. "what're" would tokenise to "what" and "are", while "we're" would break down to "we" and "are"); removing end-of-line hyphens that split whole words into parts when using a justification alignment; and abbreviations (e.g. Dr., U.S.A., etc.) [22].

The third stage is syntactic parsing, where the grammatical structure of the sentence is determined. This is usually achieved by generating a syntax tree of each sentence, where each word is broken down into a terminal from a given grammar (e.g. verbs, nouns, adjectives, determiners, etc.) [23].

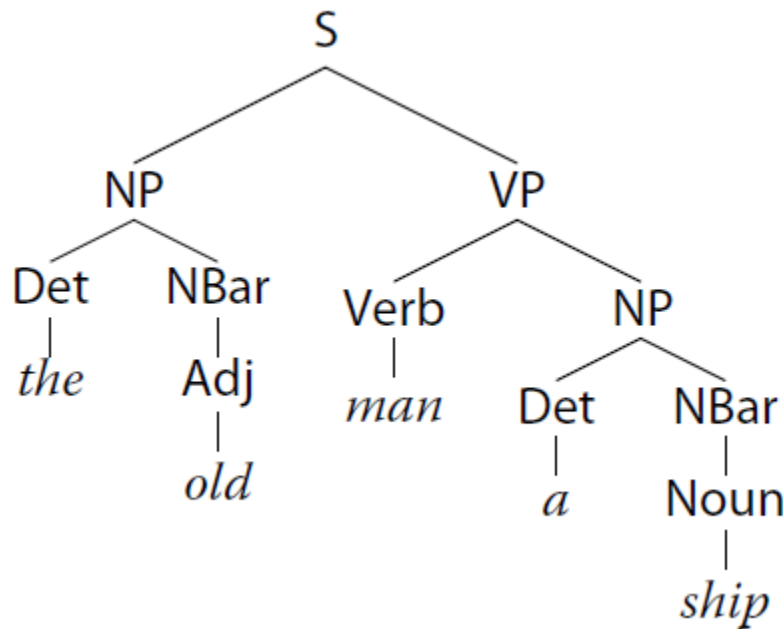


Figure 5: An example of a syntax tree for the sentence. "the old man a ship".

Diagram taken from the Handbook of Natural Language Processing [23].

In the example above for the sentence, "the old man a ship", the word, "man" could be mistaken to be a noun, but in this case, the sentence is implying that the elderly control a ship, and therefore, "man" is actually a verb here. This meaning can be more easily inferred when the sentence is broken down using a syntax tree, as above.



The final two stages are very similar and is difficult to separate them into separate stages. Both stages involve determining the meaning of the sentence. The fourth stage is often concerned with semantic analysis, whereas the last stage is more concerned with discourse analysis [22].

### 3.4.2 Part-of-Speech Tagging

A technique often used in NLP is to try to label each word with its correct part of speech, known as part-of-speech (POS) tagging. POS-tagging systems generally use a tagset containing POS tags to assign a tag to each word. The most popular tagset is the Penn Treebank tagset. This consists of 48 tags, of which 12 are for punctuation and other symbols [24]. See the Appendix 6.3 for the full Penn Treebank tagset.

There are two main challenges regarding POS tagging. The first challenge is that words can sometime be ambiguous. That is, the word can be assigned a number of possible POS tags depending on its context. For instance, the word, "can" can either be a verb ("I can..."), a noun ("a can of tuna"), or another verb with a completely different meaning to the previous ("can that noise!") [25]. The many meanings that are attributed to a word are known as its word senses.

The second challenge is words that are unknown to the tagger and cannot be tagged. A default tag is required for words which are unknown, but that could interfere with the rest of the POS-tagging process [24].

The proposed design for this project will use an open-source POS tagger in order to identify specific words in the player's input, namely verbs and nouns.

### 3.4.3 Slot Filling

According to the authors of *Speech and Language Processing* [26], three tasks need to be done to understand a user's utterance to a chatbot. The first is *domain classification*, to determine the topic that the user is talking about (e.g. weather, sport, etc), although this is unnecessary for single-domain applications such as the environment of a video game. The second task is *intent determination* to determine the goal of the user, and finally, the last task is to do *slot filling*.

Slot filling involves determining the specific details of an intent by defining a grammar that the intent must adhere to. An example for arithmetic operations would be,  $\langle DIGIT1 \rangle \langle OPERATION \rangle \langle DIGIT2 \rangle$ . The input would be parsed by a context-free grammar parsing algorithm to extract each bit of information in order to fill each slot in the grammar.

A disadvantage to slot-filling using context-free grammar parsing is that not all valid expressions for an intent would match with the slots. A solution to this would be to define multiple grammars for different structures of inputs.

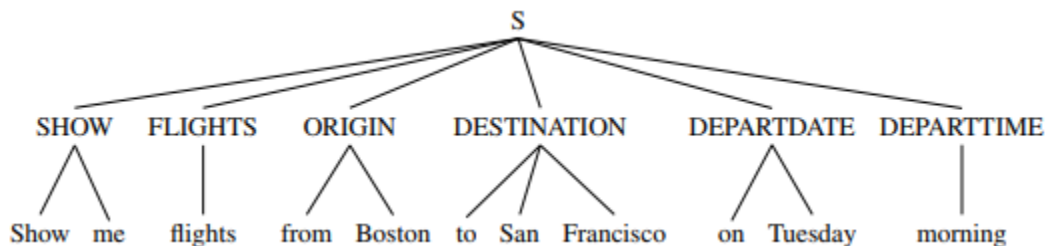


Figure 6: An example of a slot-filling grammar for flights, using slot names as the internal parse tree nodes. Diagram taken from Chapter 29 of the book, *Speech and Language Processing* [26].

### 3.4.4 Synonyms and Hypernyms

Synonyms are words or phrases which have a similar meaning to another word [27]. For example, the word "dog" has seven word senses (different meanings), and for one of the word senses with a meaning of, "a member of the genus *Canis*", the synonyms would be, "domestic dog" and "*Canis familiaris*" [28].

A hypernym, on the other hand, is a word or phrase which generalises words for which it is a hypernym of. For instance, the word, "instrument" is a hypernym of "guitar", because a guitar is a type of instrument. Similarly, the word "animal" is a hypernym of "cat" (or, at least, the sense which refers to the domestic animal, as "cat" can have several usages).

The synonyms and hypernyms of words and phrases can be represented as a hyper-tree structure, where the parent nodes are hypernyms of the child nodes, while each sibling node would represent a different word sense for a word, with each sibling containing the synonyms for that sense. Note that a hyponym is the inverse relationship between a word and its hypernyms.

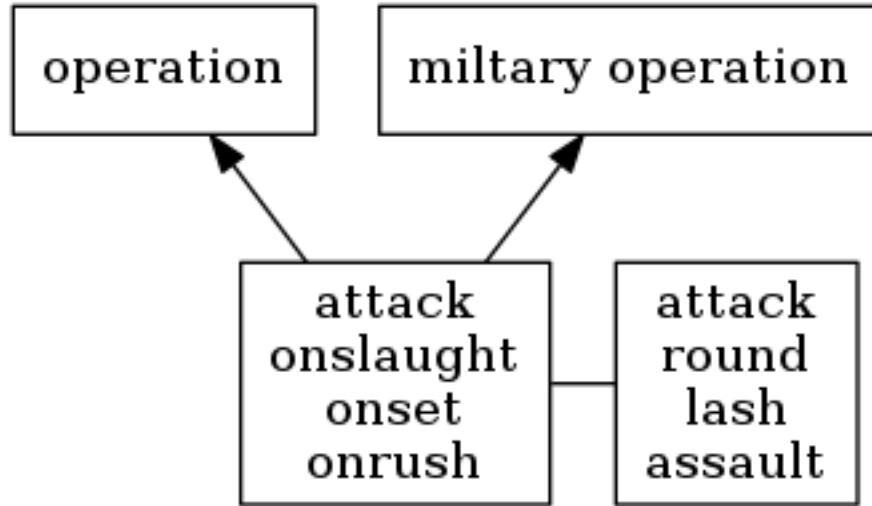


Figure 7: A partial hypernym tree showing the synonyms and hypernyms for one of the senses of the noun, "attack". The "attack" verb sense (right) is a sibling node.

Synonyms and hypernyms could be used to determine the semantic similarity between words and infer whether they match a certain intent of the user.

### 3.4.5 Semantic Similarity

There are many ways of calculating the semantic similarity between two words or phrases. Many of these methods use the synonyms and hypernym trees of the words to infer similarity, and assign a 'similarity score' to them. This score could be used to determine whether, for instance, a verb has semantic similarities to one of the intents in a game, and therefore can be mapped to that intent.

The following are some examples of measures for semantic similarity [29]. The first is the Wu and Palmer (WUP) measure, which takes the depth in the hypernym tree of the Least Common Subsummer (LCS) (that is, the most general hypernym that both words have in common) and compares it with the sum of the depths of each word. The formula for the similarity of words  $w_1$  and  $w_2$  is as follows:

$$sim_{wup} = \frac{2 * depth(LCS(w_1, w_2))}{depth(w_1) + depth(w_2)}$$

For example, the WUP score for the semantic similarity between "juice" and "water" is calculated as follows [30]:

```

1 T1 = HypernymTree( juice ) =
   [Sense #1] *ROOT* < entity < physical_entity < matter
3       < substance < food < foodstuff < juice
5 T2 = HypernymTree( water ) =
  
```

```

7      [Sense #3] *ROOT* < entity < physical_entity < matter
      < substance < food < water
9  Lowest Common Subsumer(s) = argmax(depth(subsumer(T1,T2)))
      = { subsumer(T1[1], T2[3]) } = { food }
11
12  DepthLCS = depth( food ) = 6
13  Depth1 = min(depth( {tree in T1 | tree contains LCS } )) = 8
14  Depth2 = min(depth( {tree in T2 | tree contains LCS } )) = 7
15  Score = 2 * DepthLCS / ( Depth1 + Depth2 ) = 2 * 6 / ( 8 + 7 )
      Score = 0.8

```

The WUP score of 0.8 out of 1.0 indicates that juice and water are semantically very similar [31], whereas the words "breathing" and "fire" only have a WUP score of 0.47, suggesting they are not semantically similar.

Another measure of semantic similarity is Leacock and Chodorow's method, which relies on the shortest distance between the two words in the hypernym tree, divided by the maximum depth of the whole tree; the shorter the distance, the more similar the words are [29]. There are many more different measures and it is possible to combine them to produce an overall score for semantic similarity.

### 3.4.6 Generating Sentences

One feature for the proposed design of the RPG is to be able to generate text-based descriptions of the current situation that the player is in based on objects around them. For instance, a description could be, "You are in a room lit by a candle on a table. There is a piece of broken glass on the floor and a door in front of you." One could hard-code this sentence for each room the player visits, but if there are a lot of rooms in the game, this could become time-consuming. Developers would like to generate a sentence of the current situation based on sparse pieces of information (such as the items in the room, plus their locations, in a vector representation). This is the opposite of the intent inference described above, where the meaning of a sentence/phrase is extracted; developers would like to be able to go in the opposite direction.

This is a relatively new field in NLP as it is particularly difficult. One method of sentence generation is suggested by Iyyer et al [32], where recursive autoencoders (a type of neural network) are used to generate a decomposition model of the sentence that can be used to reconstruct the sentence again. However, they found that proper nouns were not reconstructed correctly, and the model would need to be built in the first place to reconstruct the sentence.

Another group of researchers proposed a method of sentence generation which involves taking a sum of word embedding vectors and modelling it as a mixed integer programming problem [33]. However, this also requires the original sentence to be

de-constructed in the first place in order to know what the vector should contain.

One possible solution would be to only hard-code the general sentence structure of the description of rooms, and then insert the appropriate adjectives and nouns of the items, which is given as input in the form of a vector of strings.

### 3.5 WordNet

Princeton University created a large lexical database for the English language [34]. Words are grouped into synonyms (known as synsets, or *synonym sets*), and are connected to their respective hypernyms (more general words) and hyponyms (more specific words), creating the large hypernym tree mentioned previously. The database is open-source and is used extensively in this project, using various APIs that interact with the WordNet database (which will be initially downloaded by the game for offline use).

### 3.6 Object Properties

One issue to note is when a user attempts to specify an item based on a description of it (e.g. the user says, "attack the enemy with a sharp object"); players would like to be able to select an item that matches the description (e.g. a sword is sharp, so use that if the user possesses one). One approach to this is to assign to the items a set of properties such as whether it is sharp or blunt, whether it can be thrown, and so on.

A similar technique was used by the developers of Scribblenauts for the Nintendo DS. Using the company's ObjectNaut engine, the game utilises a large database of a hierarchy of objects [35]. This allows them to design sprites for different objects (such as a cyborg, robot or android), but they have the same interaction properties. However, it still required the developers to go through encyclopaedias word-by-word. Using the semantic similarity scoring mechanism, this would not need to be done, as it can be inferred whether two objects would have the same properties instead of hard-coding each one.

By assigning items different properties, the actions that the user can perform for each variation of items (e.g. sword, katana, knife, etc.) does not need to be hard-coded. Instead, there can be general objects which have certain properties, and then the actions for each general object can be coded. For example, there could be a general object with a 'sharpness' property, that would be able to, for example, cut something, and all swords and knives would be of this type of general object.

## 4 Design

### 4.1 Target Platform

According to Ovum's Mobile Games Market forecast for 2017-2022 RRR, mobile games continue to increase their market share as they grow in popularity, and in two year's time almost half of the total video game revenue generated will come from mobile games. Therefore, it makes sense to design a system for a mobile platform such as iOS or Android. It also allows for hands-free input as a future extension.

The chosen platform to develop the voice recognition system and game will be Android mobile devices, as the availability of the open-source Google Speech-to-Text API means that less effort is required to get the text input from the user's utterance.

### 4.2 Java Programming Language

The language used to develop the application is Java, which is an object-oriented programming (OOP) language. The use of an OOP language makes it easier to create objects in a video game using the inheritance hierarchy. For example, there would be an abstract class for physical objects called `PhysicalObject`, and another class such as `Glass` would be a derived class of a `PhysicalObject`, and so on.

A disadvantage to using Java is that it does not support multiple inheritance, unlike other languages with OOP constructs such as C++, due to methods such as `super()` having the same signature for all parent classes. For instance, a `Glass` class cannot inherit properties from both a `PhysicalObject` and a `BreakableObject` (i.e. they are both direct parents of the `Glass` class). Instead, this must be overcome by making the inheritance a single chain (e.g. `Glass` inherits from `BreakableObject` which inherits from `PhysicalObject`). While this can be considered a hindrance in some circumstances, it is not a major issue for the proposed design of the game.

While Java applications - and therefore Android - support the invocation of native code such as C++ using the Java Native Interface (JNI) for improved performance, this will not be used due to time constraints for the project (as performance is not a major objective for this project).

### 4.3 System Overview

The following is a brief description of the voice recognition system, including the main components and the general flow of data.

The voice recognition system takes as input the raw microphone input contain-

ing the player's utterance, and outputs the execution of a developer-defined action based on the player's utterance. Figure 8 shows a simple flow chart outlining the main stages of the voice recognition system from input to output.

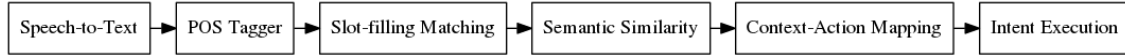


Figure 8: A flow chart showing the general stages of the voice recognition system.  
Graph generated using Graphviz.

The steps are briefly described here and explained in more detail later on.

- The audio from the microphone input is processed into a text string using a Speech-to-Text (STT) engine. This string is then split up into a list of words.
- The list of words is then used to generate a list of Part-of-Speech (POS) tags. Each word in the list is passed to the POS tagger and added to the tag list.
- The word list is parsed using a slot-filling grammar, e.g. "<ACTION> <TARGET> [with] <CONTEXT>", or "[use] <CONTEXT> [to] <ACTION> <TARGET>".
- When performing the matching, the current word being parsed is sent to a semantic similarity engine, and a similarity score is calculated.
- If the player has specified an action to be done under a certain context, then the context-action mapping is consulted to find the correct method to execute. Otherwise, the default action method is chosen.
- Finally, the in-game function is executed, and the text output is returned, with the game state updated.

#### 4.3.1 Speech-to-Text

Android features a built-in speech recognition service, via its `SpeechRecognizer` class RRR, and this is used for the speech-to-text engine. While the API's primary method of speech recognition is via remote server communication, there is an option with the Android OS settings to perform offline speech recognition instead, by downloading the language data RRR.

Using this API, the player's utterance is transformed into a text string, which is then processed by the rest of the voice recognition system. Using an existing speech-to-text API means that less time is spent trying to get the player's utterance from the microphone input.

In terms of user interface, the application features an on-screen button containing the image of microphone. When the user presses this button, a voice request is initiated and lasts until the user finishes uttering their input. If the user doesn't say anything, then the response times out and is cancelled.

### 4.3.2 POS Tagging

For the Part-Of-Speech (POS) tagging of the input, the open-source Log-linear Part-Of-Speech Tagger provided by The Stanford Natural Language Processing Group will be used RRR. It is a Java archive file so can easily be imported and used in an Android Java project. It is a flexible API that handles all the POS tagging so no time needs to be spent implementing a POS tagging system.

The software comes with two trained models. According to the software's **README** file, the first model uses a bidirectional architecture and includes training data using word shapes and distributional similarities. The second model uses a *left3words* architecture, includes words shapes, and uses the Penn Treebank tagset. According to the developers, the bidirectional model is "slightly more accurate" with a correctness score of 97.28%, but is much slower to tag with than the *left3words* model. Therefore, the *left3words* model was chosen to be used in the application. (It still features a performance correctness score of 96.97% which is not that different to the bidirectional model's performance.)

With the input string broken down into an array list of its words, each word is used as input to the POS tagger, and the word's tag (e.g. *VB*, *NN*, etc.) is added to a new list. Both lists will be used simultaneously in other processes.

### 4.3.3 Slot-filling Matching

The next stage is to extract the important information from the input. In many applications, particularly in games, voice commands are usually given in the imperative form (such as, "do this", "attack that", etc.). It is assumed that the subject of the command (that is, the entity about whom the statement is made about RRR) is always the player (or the character that the player is controlling).

Generally, commands will have an imperative verb (e.g. "attack", "grab", "look"), optionally followed by the target entity for which the imperative verb should be applied towards, as well as an optional context entity for which the imperative verb should be applied with.

As an example, take the command, "hit the rock with a spoon". Here, the word 'hit' is the imperative action and it is applied to the 'rock'. The 'hit' action is applied



using the 'spoon' as the context. The target and context help to provide a clearer intent; the user could have just said, "hit" without mentioning the target or context, and while the action to be performed is clear, it is ambiguous as to what should be hit and with what.

For this stage, slot-filling structures (SFS) are used to describe the grammar of the voice commands. The SFS grammar chosen is as follows:

#### ACTION TARGET WITH CONTEXT

Here, parts of the input will be mapped to a *block* in the grammar. The **WITH** block represents words and phrases that signify using the context to perform the action (e.g. "with...", "using...", "with the power of the...", and so on). Figure 9 shows how parts of an example command are matched to the grammar.

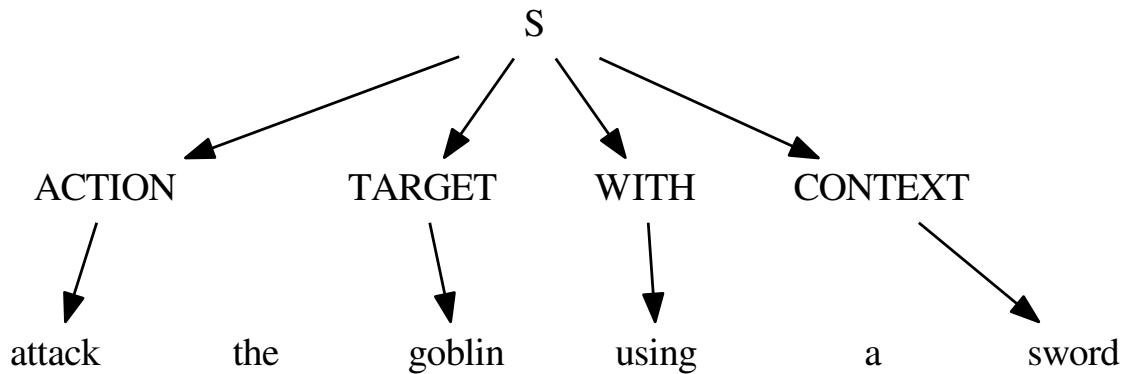


Figure 9: The slot-filling structure, {ACTION TARGET WITH CONTEXT} being applied to an example sentence, "attack the goblin using a sword". Generated using Graphviz.

Not all the slots have to be filled; the **WITH** and **CONTEXT** blocks are completely optional to fill, but they provide more information about the user's intent. Should this information be left out, the system will fall back on the defaults for each one.

While this covers a large variety of commands that the user can give, such as, "attack the troll with a sword", "pick up the knife", or "look around the room using binoculars", it does not cover commands that have a completely different structure. For example, the phrases, "use a potion to heal" and "with a key open the door" are perfectly valid commands that a user can give, but the matching blocks are in completely different order (i.e. the **CONTEXT** comes before the **ACTION**), so would not be matched with the above SFS.

An alternative SFS is applied when appropriate. The grammar is as follows:

WITH CONTEXT ACTION TARGET

Using this slot-filling grammar, the aforementioned example commands ("use a potion to heal" and "with a key open the door") would be accepted. Figure 10 shows an example of the matching.

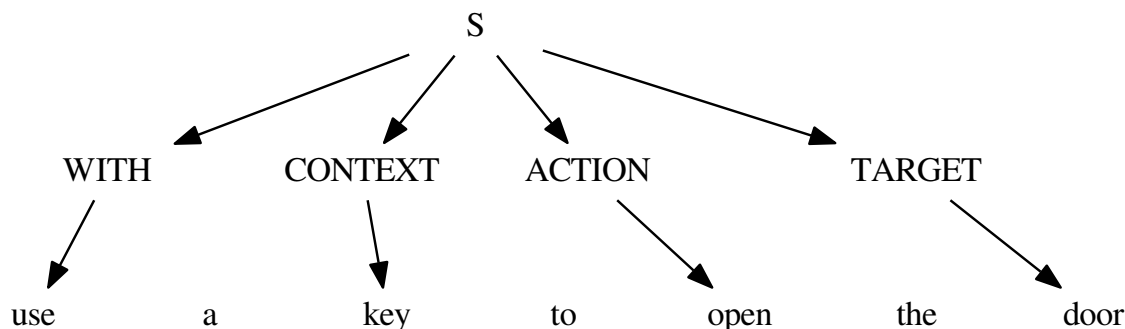


Figure 10: The slot-filling structure, {WITH CONTEXT ACTION TARGET} being applied to an example sentence, "use a key to open the door".

During the processing, the correct SFS will be chosen based on the structure of the phrase (i.e. by determining the relative positions of each piece of information).

List of possible actions, targets and contexts are created by the developer (specified as strings), and the slot-filling matching will compare the words to the possible list of words to find a match.

#### 4.3.4 Semantic Similarity

During the slot-filling matching stage, if a word does not match any of the possible list of actions/targets/contexts character-by-character, then semantic similarity methods are used to determine the whether the two words are similar. For example, if the possible list of actions are, 'attack', 'heal' or 'run', and the input is 'hit', then 'hit' is more similar to the 'attack' action, so the 'attack' action would be chosen.

As mentioned before, there are many similarity methods that have been proposed. The first method to be used is the *Wu and Palmer* method (WUP) described previously. Another method that is used is the *Lin* method (LIN), that calculates the similarity of two words based on the information content of their lowest common

subsumer (that is, their first common parent word in the WordNet hypernym tree) [29]. Information content (IC) is defined as the negative log of the probability of a concept. The probability of a concept is based on the sense frequency of its synset.

Lin used IC to define a similarity measure similar to the Wu and Palmer method, but uses IC instead of the depths of the words in the word tree instead. The formula for the similarity of words  $w_1$  and  $w_2$  is as follows:

$$sim_{lin} = \frac{2 * IC(LCS(w_1, w_2))}{IC(w_1) + IC(w_2)}$$

Another method used is the *Lesk* method. The method determines the similarity of two words based on the overlaps in their definitions RRR. For instance, the definition for certain senses of the words, 'ash' and 'coal' (to describe a burnt substance) have the words, 'combustible', 'burn' and 'solid' in their definitions. Therefore, it can be deduced that the words are semantically similar. In another paper, Banerjee and Pederson extended the Lesk method by using all the definitions of all the sense for both words in the calculation, as only using one sense definition did not give enough information RRR.

As well as using the definitions of the synonyms, hypernyms and hyponyms of the word, this method also uses the definitions of the word's meronyms and holonyms. Meronyms are words that represent part of something else RRR. For example, 'face' would be a meronym of 'person'. Holonyms are words that represent the whole of several parts, so a 'body' would be a holonym for words such as 'arm', 'leg' and 'torso' RRR. Using these definitions as well make the Lesk method more accurate, but also add more processing time for each calculation.

The WordNet database is accessed and traversed using the MIT Java WordNet Interface (JWI), which is an open-source library that interfaces with a WordNet database RRR. A full English WordNet database is downloaded from Princeton's website, and the JWI API is loaded with it.

Implementations for most of the similarity methods are provided by another open-source Java library, *WS4J* RRR. While this library provides the methods for doing the calculations (such as the WUP, LIN and LESK formulas), the methods for the navigation and analysis of the WordNet database (e.g. getting hypernyms, synsets, glosses, etc.) must be implemented from scratch (see Implementation for details).

#### 4.3.5 Context-Action Mapping

Once matches have been found from the slot-filling, they are mapped to the methods that should execute on that intent. If the player only specifies an action (and

target) with no context, then a default action should be executed (the context itself defaults to a "default" context). If the player specifies a context (e.g. "...with a sword"), then a mapping is consulted to find the correct method that should be called. The different targets are handled within the method itself.

Table 1 shows an example of what an Context-Action map would look like. Note the *null* entries that indicate that the context is not compatible with the corresponding action (for example, it does not make sense from a player to "attack with a potion"), and the intent is therefore ignored.

Table 1: An example of an developer-specified Action-Context Mapping

|                | <b>actions</b>     |                |                  |
|----------------|--------------------|----------------|------------------|
| <b>context</b> | <i>attack</i>      | <i>heal</i>    | <i>open</i>      |
| <i>default</i> | AttackDefault()    | HealDefault()  | OpenDefault()    |
| <i>weapon</i>  | AttackWithWeapon() | null           | OpenWithWeapon() |
| <i>potion</i>  | null               | HealWithPotion | null             |
| <i>key</i>     | AttackWithKey()    | null           | OpenWithKey()    |

#### 4.3.6 Intent Execution

Each non-null field in the Context-Action map contains a class that extends from an abstract class, **Action**. This abstract class contains an method called **run()**, which executes the in-game intent. This method is overridden by the derived classed (such as **AttackDefault** and **HealWithPotion**). These **run()** methods act as wrappers for the actual methods that would execute in the game (or to execute some other actions unrelated to the game, such as displaying the available commands, etc.)

## 4.4 System UML Class Diagram

Figure 11 shows a simplified UML class diagram focusing on the classes used for the voice recognition system, including the inheritance and associations of the classes. Although the **GameState** class is related to the game system, it helps to illustrate the overall system design. Only the important member fields and methods are shown for each class.

See the Appendix 6.4 for the full UML class diagram of the Android application.

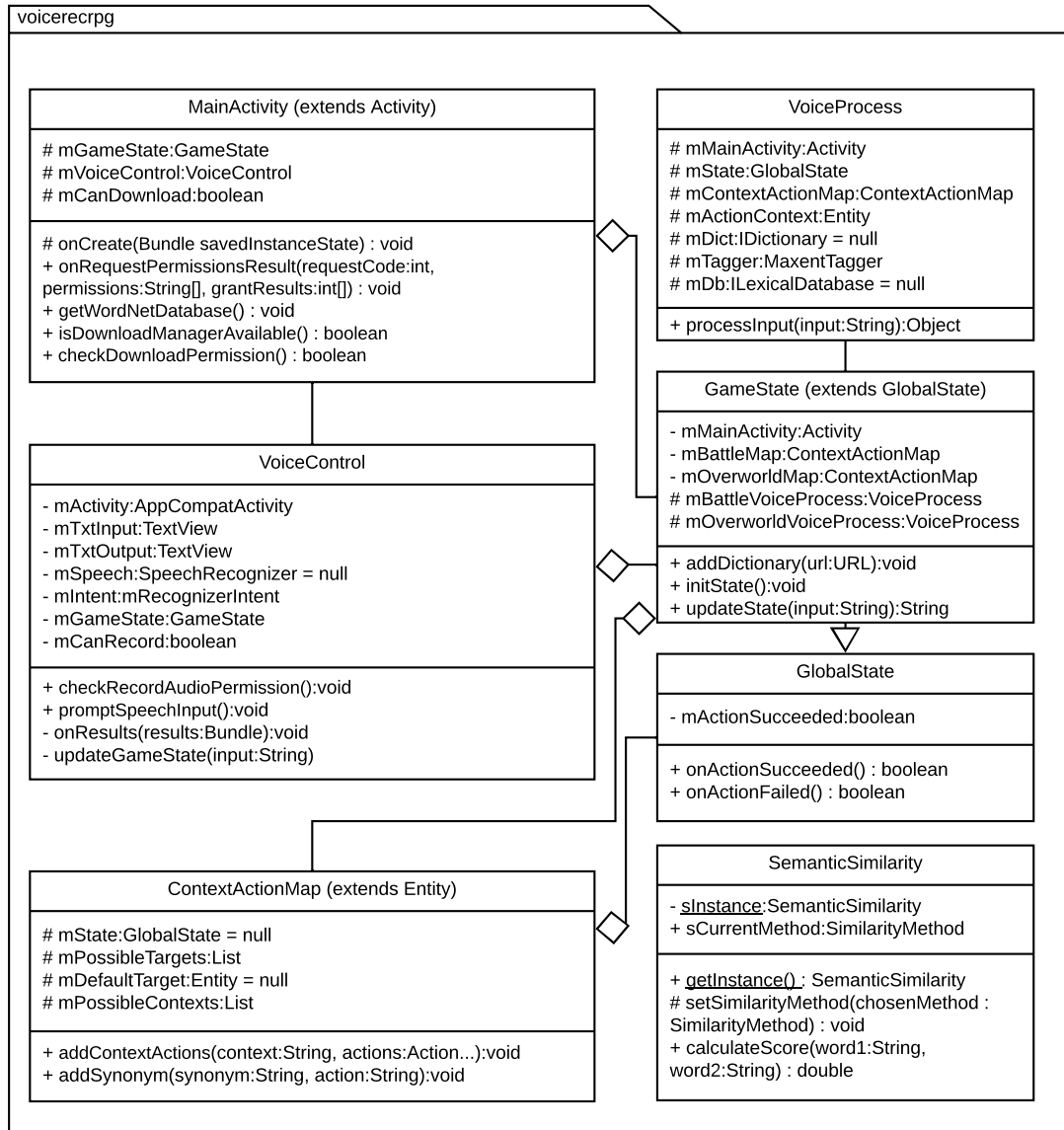


Figure 11: A simplified UML class diagram outlining the design of the voice recognition system. Diagram created using the online LucidChart tool. See the Appendix 6.5 for the full class diagram for the entire voice recognition system in the Android application.

#### 4.4.1 MainActivity Class

The **MainActivity** class is the skeleton class which contains all the systems. It acts as the interface between the I/O and the rest of the application. It contains instances of the **VoiceControl** class, which handles the I/O, and the **GameState**,

which contains the game environment.

When the application starts up, an instance of **MainActivity** is created, and searches the public directory of the phone's external storage (not necessarily an external device like an SD card, but whatever the OS has defined the external storage to be). It searches for a WordNet database that is downloaded beforehand, and if it is not found, it will download/re-download the database again. It will also copy a model used for POS tagging to the external storage from the APK archive.

#### 4.4.2 VoiceControl Class

As mentioned, the **VoiceControl** class is used to handle the input and output of the application, such as the microphone input and the text display. It contains an instance of the **SpeechRecognizer** class, which is provided by Android and uses Google's Speech-to-Text API, as well as two instances of the **TextView** class: one which is used to display the text output of the game, and the other to display the Speech-to-Text result at the top of the screen.

When the **SpeechRecognizer** instance performs a Speech-to-Text translation, the final result is forwarded to the **GameState** instance where it is processed by the voice recognition system using NLP, and used to update the state of the current game session (**GameState**).

#### 4.4.3 VoiceProcess Class

The **VoiceProcess** class is responsible for processing the string input containing the player's utterance that is received from the **VoiceControl** class. It contains the bulk of the NLP processing described later.

#### 4.4.4 GlobalState Class

The **GlobalState** class is a generic abstract class that can be derived from in order to pass around environment objects between method calls easily. For example, if a method wishes to access an instance of an **Inventory** class (to access the game's items), then a reference to the inventory can be held in a class derived from **GlobalState** (e.g. **GameState**).

#### 4.4.5 ContextActionMap Class

The **ContextActionMap** class contains the mapping of contexts to actions. It is an abstract class that is overloaded by the developer to add mappings of different ways of performing actions based on the given context (e.g. "attack with a sword" versus "attack with a knife").

#### 4.4.6 SemanticSimilarity Class

This class contains the semantic similarity engine used for calculating the semantic similarity of two words as a confidence value between 0.0 and 1.0, based on information in the WordNet database. It uses a singleton design pattern so the class does not have to be instantiated every time a calculation needs to be performed with a different method, nor does a reference to an instance need to be passed around everywhere it is used. It also allows us to use the same static instance across different Android Activity contexts (and therefore allowing the engine's parameters to be changed in a *Settings* activity).

### 4.5 System Features

In order to make the voice recognition system as as flexible as possible and able to infer the meanings of a wide variety for utterances, several mechanisms have been added to the system. These mechanisms help to reduce the amount of work required by the developer, and some are present in order to compensate for the shortcomings of the system (see the Evaluation section).

#### 4.5.1 Synonym Mapping

The developer can map synonyms to actions if the words are not considered semantically similar by the system. For instance, if the word, "regenerate" has a low semantic similarity score when compared to the "heal" action, then the developer can create a map of "regenerate"  $\rightarrow$  "heal" so the system performs the correct mapping.

#### 4.5.2 Ignoring Incorrect Matches

Semantic similarity matches can be ignored if they are incorrectly mapped by the system, as a juxtaposition to the above feature.

#### 4.5.3 Confirmation and Suggestions

If the user gives a command that has a confidence value just below the threshold, then the phrase is considered ambiguous and, using the possible candidate matches, suggestions are given to the user to confirm their intent.

For example, if the user's intent is to pick up a knife and they say, "pick up the utensil", where the similarity score between "utensil" and "knife" is just below the threshold, the system will output, "Did you mean, 'pick up the knife? (yes/no)", and waits for the user to respond. If they say no, then the system may suggest

another alternative using the next ambiguous candidate (e.g, "Did you mean, 'pick up the plate?'" , etc).

#### 4.5.4 Multiple Commands

The user has the ability to execute a chain of several commands one after the other using just one utterance. For example, if the user says, "attack the enemy and then heal using a potion", this will execute two separate commands.

#### 4.5.5 Multiple Targets

A user may want to execute a command that involves several targets. In this case, these are divided into several actions instead, with each action focused on one of the targets. For instance, if the user says, "attack the troll and the goblin", this will be split into two actions: one to attack the troll, followed by an attack on the goblin.

#### 4.5.6 Sentence Mapping

The core mechanism of the system is the slot-filling structure described above. This covers most of the forms of possible imperative commands that a user may utter to the system to execute an intent. However, one disadvantage of this system is that it will not detect phrases that do not map to the slot-filling grammar given (e.g. {ACTION TARGET WITH CONTEXT}), despite being a valid phrase. Such utterances are usually phrased as a question, such as, "what actions can i do", or , "what items are in my bag", etc. The system should still be able to detect these types of commands but prevent the developer from hard-coding each possible variation of the intents.

The proposed solution is to perform sentence matching on these phrases. The developer specifies an intent followed by several example sentences that a user would say to execute the intent. For example, if the intent is to show the player the items they possess, example sentences include: "What is in my bag?", "What items do I have?", "What are the contents of my inventory?", and so on. When parsing a new utterance from the player (e.g. "what items are in my bag"), if the phrase fails the slot-filling stage, then the similarity of the sentence is calculated compared to the example sentences.

The calculation involves using a slightly altered cosine similarity method. Based on the formula for the cosine of an angle between two vectors, the cosine similarity method measures uses the cosine of the angle as a measure of similarity, using a vector space model RRR (the vector of a sentence is represented by its word frequencies). The similarity between two vectors,  $a$  and  $b$  is:



$$similarity = \cos \theta = \frac{a \cdot b}{|a||b|}$$

Take two examples: "the apples are in the tree" and "the tree contains apples". The vectors for these sentences would be:

$$\begin{aligned} a &= \{\text{the:2, apples:1, are:1, in:1, the:2, tree:1, contains:0}\} \\ b &= \{\text{the:1, apples:1, are:0, in:0, the:1, tree:1, contains:1}\} \end{aligned}$$

The cosine similarity of these sentences is therefore:

$$\frac{a \cdot b}{|a||b|} = \frac{6}{2\sqrt{3} \cdot \sqrt{5}} = 0.77$$

The definition for cosine similarity has been altered to form a "soft cosine measure" RRR. This uses the semantic similarity between the two words (when the dot product is calculated) as a weighting: if a word only appears in one sentence and not the other, then the scores of its semantic similarity to the words in the other sentence are added to the numerator.

While the cosine similarity is not ideal for all situations (e.g. it ignores word order so may match two sentences which are not similar at all).

## 4.6 Game Design Overview

The voice recognition system will be applied to a text-based role-playing game that has several different modes of play, in order to demonstrate the flexibility of the system. The first mode is similar to the Zork-style text-adventure gameplay, mentioned previously. This involves the player interacting with objects in the environment to solve puzzles while collecting items, in order to progress. This mode is named the *Overworld* mode.

The second mode of gameplay is similar to classic turn-based RPGs such as Pokemon or Final Fantasy, where the player character battles an enemy, and the opposing sides alternate between taking turns to either attack, defend, heal, or perform some other action. This mode is called the *Battle* mode.

### 4.6.1 Game Flow

The focus of the project is not on the game itself, but rather on the power and flexibility of the voice recognition system to add voice commands to the game itself. Therefore, only a very simple game will be created with only two rooms, with each room focusing on a different game mode (*Overworld* or *Battle*).

For the first room, the player is placed in a room where they are presented with a puzzle as part of the *Overworld* mode. The room features the following description:

- "You are in a room with a locked door in front of you."
- "There is a glass table in the middle of the room."
- "There is a knife on the table."
- "A painting of a tree hangs by a string on the left wall."

The player must obtain a key hidden behind the painting by cutting it down using the knife (or any other sharp item they may possess such as a sword), and then use the key to open the door.

During the overworld gameplay, the user is capable of performing the following actions:

[look, show, grab, open, cut, break]

Example commands include, "show my inventory", "look around the room", "grab the knife", "cut the painting", and so on.

Each room in the game that features the overworld gameplay is represented by a finite state automaton. When the player performs certain actions (e.g. cutting the painting down), the state of the room will progress until the player has performed the correct actions in the correct order and is able to progress to the next room. Figure 12 show the state machine for the first room.



Figure 12: The state machine for the first room in the game.

For the second room, the player is presented with a battle with an enemy (a *troll*) as part of the *Battle* mode.

[attack, heal, show, look]

Examples include: "attack the troll" "heal with a potion", "look around", and so on.

## 4.7 Game Design UML Class Diagram

Figure 13 shows a simplified UML class diagram for the game logic. See the Appendix for the full UML class diagram of the game logic.

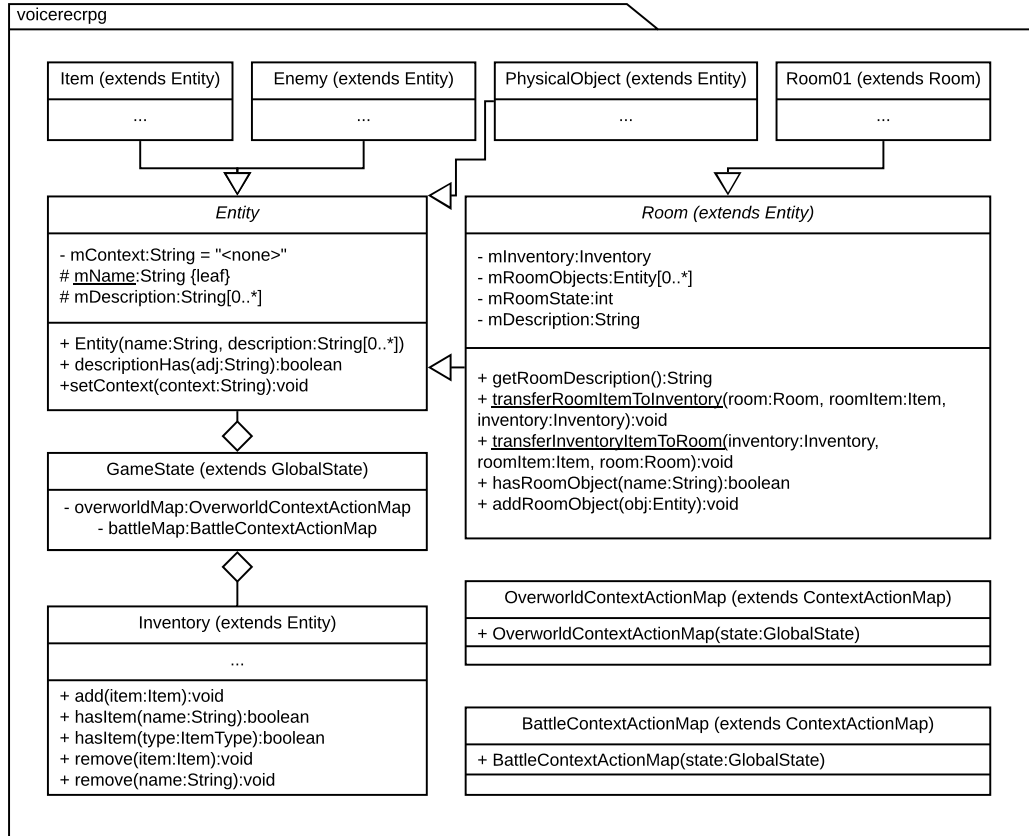


Figure 13: A simplified UML class diagram outlining the design of the gameplay mechanics. Diagram created using the online LucidChart tool. See the Appendix for the full class diagram.

The **GameState** class (shown previously in the system UML class diagram, to aid explanations) contains two different instances of classes that override the voice recognition system’s **ContextActionMap** class (which contain the voice command mappings): one for the overworld gameplay and the other for the battling gameplay.

The **Entity** abstract class represents anything that can be considered a possible target or context for the slot-filling. Each entity is given a name and a context type (as a string) that is used to determine which context it maps to (e.g. "weapon", "key", etc). See Table 1 above for examples of these contexts. Each entity also

contains a description field which is an array of words that describe the entity (e.g. a knife would be described as "sharp", "pointy", etc). These descriptions are optional but they aid the voice recognition system to help identify similar words (if the semantic similarity match fails).

Finally, `Room` is an abstract class for generating new rooms for the game, each with their own objects.

## **4.8 Room Generation**

Instead of forcing the developer to manually instantiate object within each room, a mechanism will be written that will allow a developer to generate a new room from just a description of it, using natural language processing techniques.

## 5 Implementation

This section outlines the steps taken to implement the system and the game, mostly in chronological order where appropriate. For more details on the source code, please the corresponding GitHub repository containing this project's work.<sup>1</sup>

### 5.1 Voice Recognition Interface

The first step involved creating an Android project with a skeleton I/O to take as input the microphone audio, converts the speech to text, sends it to the voice recognition system for processing (initially just an empty class that pipes the input to output), before finally displaying the text on the screen.

The interface is within an Android **Activity** class. When the activity is created, it also downloads a tar ball containing a WordNet database and unzips it on the device's local storage. While it is possible to package the database with the model, the code was originally written to download the database and, due to time constraints, could not be changed.

The **MainActivity** class contains a **VoiceControl** classes that handles the speech-to-text (STT) I/O, and implements a **RecognizerIntent**. The **VoiceControl** object contains an instance of the **SpeechRecognizer** class that has a **startListening** method which invokes an **onResults()** method when finished. This method have a **results** parameter of type **Bundle**, containing the string result of the speech recognition. The string is then processed by an instance of the **VoiceProcess** class (described below).

### 5.2 Graphical User Interface

Figure 14 shows a snapshot of the UI for the game logic with the following labels:

1. The player's input for the game as a string (i.e. it is the output of the speech-to-text API).
2. The cumulative output as the game processes each user input.
3. Pressing this button enables the microphone and starts a voice intent.
4. A timer used for debugging the time taken to process an intent by the system.
5. This drop-down list contains an option to open the settings menu.

---

<sup>1</sup><https://github.com/BaronKhan/VoiceRecognitionRPG>

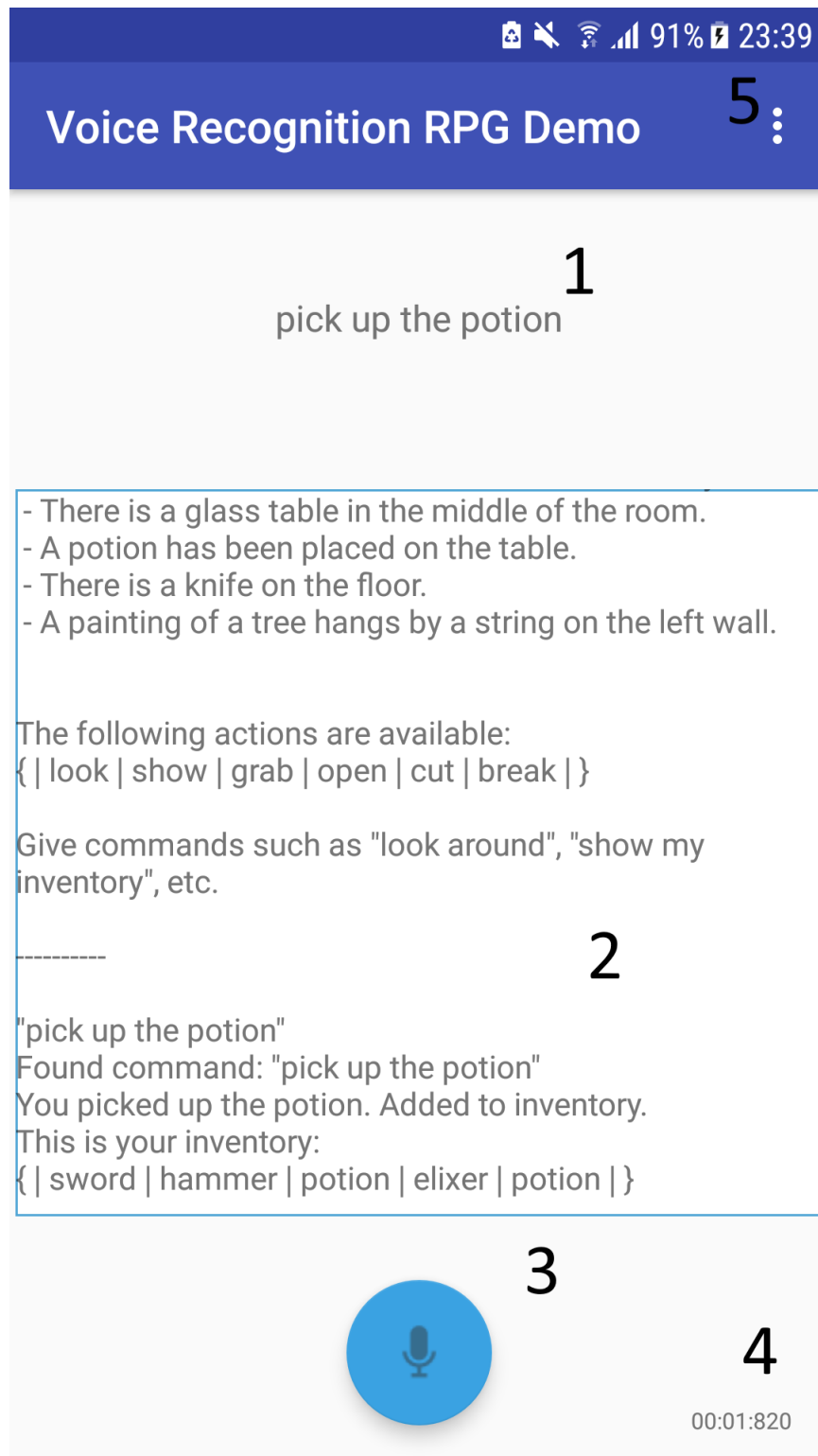


Figure 14: A snapshot of the user interface for the text-based role-playing game.

All UI elements are accessed in the `MainActivity` and are separate from the voice recognition system.

## 5.3 Voice Recognition System Implementation

### 5.3.1 GlobalState.java

The `GlobalState` class is an abstract interface that sends the string input to a `VoiceProcess` instance via the `updateState` method. Derived classes (e.g. `GameState` for a game, `KitchenState` for a cooking application, etc.) contain the application-specific logic and environment that can be accessed by the action methods. Listing 1 shows the code for the `GlobalState` class.

Listing 1: GlobalState.java

```
2 package com.khan.baron.voicerecprg.system;
3
4 //Derived classes should contain the environment objects to be used in actions
5 public abstract class GlobalState {
6     protected boolean mActionSucceeded = true;
7
8     public void actionSucceeded() { mActionSucceeded = true; }
9
10    public void actionFailed() { mActionSucceeded = false; }
11
12    public boolean getActionSucceeded() { return mActionSucceeded; }
13
14    public abstract String updateState(String input);
15 }
```

In some scenarios, a follow-up action may only take place if the user's action was successful. For instance, in a generic two-player game, the second player may only be able to make their move once the first player has made a successful move. A more relevant example is in a turn-based role-playing game, where the player is fighting an enemy. The enemy can make their move once the player has made their move.

To support this scenario, there is a Boolean field called `mActionSucceeded`, which is set within the action methods that are executed to tell the system whether the action has succeeded, and the developer can choose what happens next.

### 5.3.2 VoiceProcess.java

`VoiceProcess.java` is the main class that performs all the processing on the text string input (of the user's utterance). The main method is the `processInput` method that takes the string input and returns the response. Listing 2 show the pseudocode for the method, with most of the details ignored (as it is a relatively long method). For the full Java method, refer to the source code in the project's repository.

Listing 2: VoiceProcess.processInput() pseudocode

```

def processInput(input):
2   checkIfWordNetIsLoaded()

4   if expectingReply:
       return processPendingIntent(input)

6   //Check for action replies
8   if currentAction.wantsReply():
       return currentAction.processReply(input)

10  //Tokenise and tag the input
12  words = input.split(" ")
13  tags = getTags(words)
14  if words.size() != tags.size():
       throw Exception

16  //Check for learning phrase ("___ means ___")
18  if words.size() == 3 and words.contains("means"):
       addSynonym(words[0], words[2])

20  //Parse the input
22  actionIndex, chosenAction = getBestAction(words, tag)

24  if isValidAction(actionStr):
       chosenTarget = getBestTarget(words, tags)
26       chosenContext = getBestContext(words, tags)

28       if not isValidContext(bestContext):
           or contextActionMap.get(chosenContext).get(chosenAction) == null:
30           chosenContext = "default"

32       action = contextActionMap.get(chosenContext).get(chosenAction)
33       if action == null:
           return "Intent not understood."
34       else:
           if isAmbiguous():
               return suggestion() //Did you mean...?
36           else:
               return action.execute(state, chosenTarget)

40  else:
       //Check for another target for previous action
42       if foundAnotherTarget(words, tags):
           currentTarget = getBestTarget(words, tags)
44           return previousAction.execute(state, currentTarget)
       else if foundAnotherContext(words, tags):
           currentContext = getBestContext(words, tags)
46           return previousAction.execute(state, currentContext)
48  else:
       performSentenceMatching(input)

```

When tokenising and tagging the input create the **words** and **tags** lists, the index of a word in **words** corresponds to the index of its tag in **tags**.

On line 25 of the pseudocode, the order of choosing the best action, target and context (if they exist) is decided by which slot-filling structure (SFS) is being used. Below is the Java code for this.

1 ...



```

3  mUsingAltSFS = isUsingAltSlotFillingStructure(words, tags, actionPair.first);
4  if (mUsingAltSFS) {
5      // SFS: with/use CONTEXT ACTION TARGET
6      chosenContext = getBestContext(words, tags, true);
7      actionPair = getBestAction(words, tags);
8      currentTarget = getBestTarget(words, tags, true);
9  } else {
10     // SFS: ACTION TARGET with/using CONTEXT
11     actionPair = getBestAction(words, tags);
12     currentTarget = getBestTarget(words, tags, false);
13     chosenContext = getBestContext(words, tags, false);
14 }
15 ...

```

See the Appendix 6.6 for the Java code of the `isUsingAltSlotFillingStructure()` method.

For the `getBestAction()`, `getBestTarget()` and `getBestContext()` methods, the words in the input are checked sequentially to determine the best match. For instance, in the `getBestAction()` method, the candidate actions are chosen from the input. For each candidate action, a score out of 1.0 is calculated against each possible action defined in the `ContextActionMap`: if the words match character-by-character or the the candidate is a synonym for one of the action, then the score is set to 1.0. Otherwise, the score is set to the semantic similarity score of the two words. The `getBestTarget()` and `getBestContext()` methods work similarly.

The candidates are chosen using the POS tagging, to filter out words with no meaning such as "the", "it", etc. Words in the input are candidate actions if they are tagged as either a verb, adverb, or noun (as sometimes, the POS tagger incorrectly tags some imperative verbs as nouns). Listing 3 shows the method for finding candidate actions, which returns a list of the indices (positions) of the candidates in the `words` and `tags` lists.

Listing 3: `getCandidateActions()`

```

1 private List<Integer> getCandidateActions(List<String> tags) {
2     List<Integer> candidateActions = new ArrayList<>();
3     for (int i=0; i<tags.size(); ++i) {
4         String tag = tags.get(i).toLowerCase();
5         if (tag.charAt(0) == 'v' || tag.charAt(0) == 'n' || tag.charAt(0) == 'j') {
6             candidateActions.add(i);
7         }
8     }
9     return candidateActions;
10 }

```

Listing 4 shows the pseudocode for the `getBestAction` method. For the full Java code for this method, see Appendix 6.7.

Listing 4: `getBestAction()` pseudocode

```

def getBestAction(words, tags):
2   candidateActions = getCandidateActions(tags)
   bestScore, bestIndex, bestAction = ACTION_MIN, -1, "<none>"
4   actionList = mContextActionMap.getActions()
   for i in candidateActions:
6       word = words.get(i)
       if (wordIsSynonym(word)):
8           return new Pair(i, synonyms.get(word))
       for action in actionList:
10          if word.equals(action):
              return new Pair(i, action)
12          for action in actionList:
              score = SemanticSimilarity.getInstance().calculateScore(word, action)
14          if score > ACTION_MIN && score < ACTION_CONFIDENT:
              isAmbiguous = true
              addAmbiguousActionCandidate(word, action, score)
              if score > bestScore:
18                  bestScore = score
                  bestIndex = i
                  bestAction = action
20          return new Pair(bestIndex, bestAction)

```

On line 11 of Listing 4, the same for loop over `actionList` is split into two: the first loop checks for direct matches, and the second loop runs the semantic similarity engine. The reason for the separation is so that, if there is a direct match with any of the words, then there is no need to run the similarity engine at all, and therefore the intent is processed faster.

### 5.3.3 AmbiguousHandler.java

As shown on line 14 of Listing 4, candidate actions may be marked as ambiguous/uncertain if their confidence score is just below the `ACTION_CONFIDENT` threshold value. If the best action/target/candidate has a confidence value lower than this threshold, then all the ambiguous candidates are queried to the user using the `AmbiguousHandler` class. This class will generate a suggestion and display it to the user, starting with the ambiguous candidate which gave the highest confidence. If the user disagrees with the suggestion (i.e, they say "no" or something similar), then the next suggestion is given, until no more suggestions are available.

Figure 15 shows an example of a suggestion displayed to the user after an ambiguous intent. See the `AmbiguousHandler.java` file in the project's repository for the full code.

### 5.3.4 SemanticSimilarity.java

The semantic similarity engine is implemented as a singleton design pattern with a private constructor and a private instance of itself within the class. The main method of the class is the `calculateScore()` method which returns the semantic

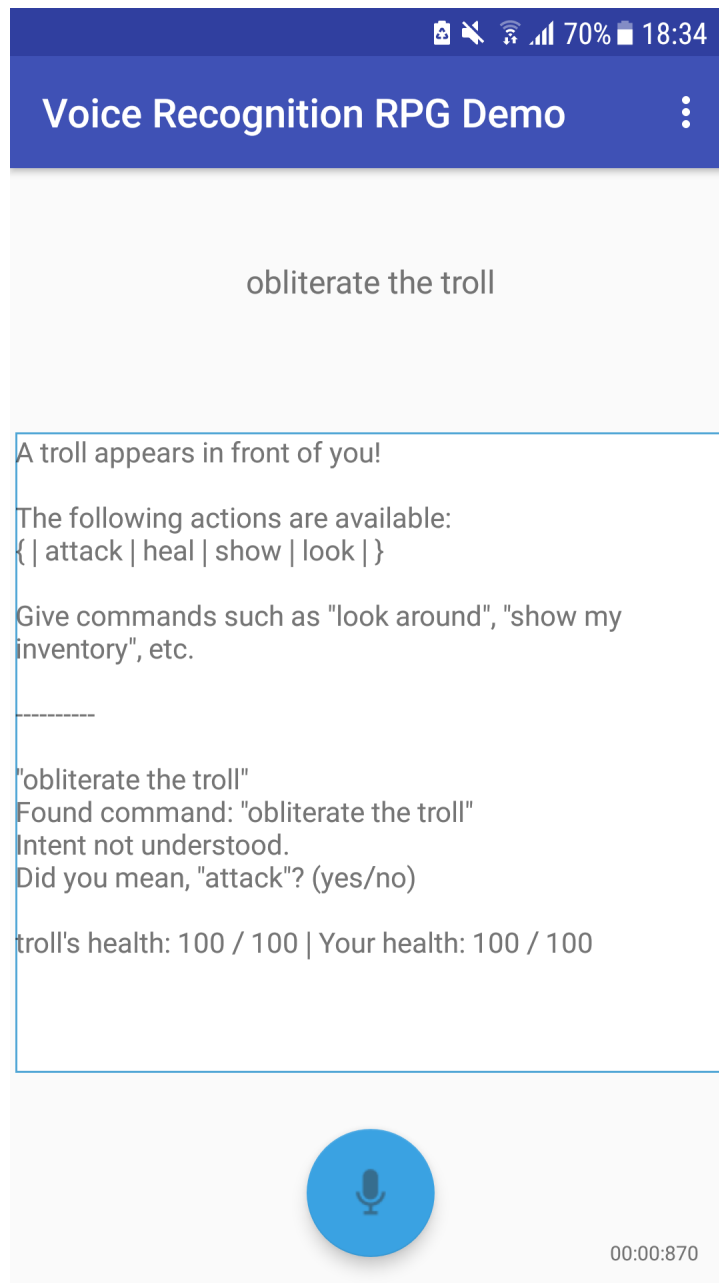


Figure 15: A snapshot of the ambiguous user intent, "obliterate the troll". Here, the word, 'obliterate' is most semantically similarity to the 'attack' action, but not enough to pass the threshold, so it is queried to the user.

similarity score for two words.

The engine uses up to two semantic similarity methods, and if two are chosen, then the score returned is the average score from both methods. As well as the Wu and Palmer (WUP), Lin and Lesk methods mentioned previously, below is a list of methods that can also be chosen:

- Fast Lesk (FASTLESK) - a faster implementation of the Lesk method, that removes definition string formatting and ignores meronym and holonym definitions of the words.
- Path-based (PATH) - a simpler version of the WUP method which uses the reciprocal of the shortest path between the words in the WordNet database as the similarity score RRR.
- Leacock and Chodorow (LCH) - another path-based measure which uses the negative log of the shortest distance between the two words RRR.
- Resnik (RES) - similar to the LIN method, but does not normalise the information content of the words' lowest common subsumer RRR.
- Cosine (COS) - similar to the Lesk method, but uses the cosine similarity of their definitions (as opposed to just overlaps in words).

Most of the above similarity method implementations are provided by the WS4J library, except the FASTLESK and COS methods, which are unique to this project and are implemented manually in the code (though the FASTLESK method is altered from WS4J's LESK method which was decompiled using Android Studio's built-in Fernflower decompiler). By default, only the WUP method is used by the engine, though other methods can be chosen in the application's settings.

### 5.3.5 CustomLexicalDatabase.java

Despite the WS4J library providing implementations of the algorithms for most of the semantic similarity methods used in this project, they require an interface to be implemented in order to access a WordNet database. Listing 5 shows the interface that needs to be implemented and used as an input to the WS4J methods.

Listing 5: ILexicalDatabase.java (from the WS4J library)

```
1 package edu.cmu.lti.lexical_db;  
3 import java.util.Collection;  
  import edu.cmu.lti.lexical_db.data.Concept;  
5  
6 public interface ILexicalDatabase {  
7     Concept getMostFrequentConcept(String word, String pos);
```

```

9      Collection<Concept> getAllConcepts(String word, String pos);
      Collection<String> getHypernyms(String synset);
      Concept findSynsetBySynset( String synset );
11     String conceptToString( String synset );
      Collection<String> getGloss( Concept synset, String linkString );
13 }

```

The WS4J library is packaged with its own implementation of this interface, called **NictWordNet** (which accesses a Japanese WordNet database similar to the Princeton WordNet) RRR. Unfortunately, due to its use of SQLite methods to access the database, the implementation is not compatible with the Android SDK, and could not be used.

A new implementation of the **ILexicalDatabase** interface was created that works with the Android SDK, called **CustomLexicalDatabase.java**. It interfaces with the Princeton WordNet database using the MIT JWI API mentioned previously. All methods were implemented using the JWI API except the **findSynsetBySynset()** and **conceptToString()** methods (these methods are not required for the above similarity methods).

Since this implementation using the JWI library may be useful for others in the future, a GitHub gist has been published online that other developers may freely use <sup>2</sup>. See the Appendix 6.8 for the full implementation.

### 5.3.6 ContextActionMap.java

The **ContextActionMap** class contains the table of mappings of contexts to intents. The developer uses this abstract class to create a derived class that defines the mapping in a 'table' format (with contexts on the left and actions along the top) in the constructor. The **setActionsList()** method specifies the possible actions, while the **addContextActions()** method describes which Java method should be invoked for each action based on the context. The entries in the 'table' are either null or inherit from the **Action** class (see Section 5.3.8).

There is also an **addDefaultContextActions()** method, which defines the actions to be called if no context is given in the input. A **null** entry means that the action is invalid for the given context (e.g. you cannot attack with a potion). In this case, the default context action is called instead.

Listing 6 shows an example of a derived class to create a table. Note that this example was only used for testing and evaluation of the system, and is not one of the tables used in the RPG demo.

---

<sup>2</sup><https://gist.github.com/BaronKhan/5679157d1a8572debc8c75af1c875c16>

Listing 6: Example of a ContextActionMap

```

1 public class GameContextActionMap extends ContextActionMap {
2     public GameContextActionMap(GlobalState state) {
3         super(state);
4         setActionList(
5             "attack", "heal", "move");
6         addDefaultContextActions(
7             new Attack(), new Heal(), new Move());
8         addContextActions("weapon", new AtkWeapon(), null, null);
9         addContextActions("potion", null, new HealPotion(), null);
10    }
11 }

```

In order to make it easier for the developer to create this 'table', a Python script was written called `generateTable.py` which takes as input a CSV file of a table, and generates a Java source file similar to Listing 6. Table 2 shows the CSV table (where cells are separated by commas) that will generate the Java source file in Listing 6 using the command,

```
python generateTable.py game-map.csv GameContextActionMap
```

See the source code in the project's repository for the full Python script.

Table 2: game-map.csv

|                |               |             |             |
|----------------|---------------|-------------|-------------|
|                | <i>attack</i> | <i>heal</i> | <i>move</i> |
| <i>default</i> | Attack        | Heal        | Move        |
| <i>weapon</i>  | AtkWeapon     |             |             |
| <i>potion</i>  |               | HealPotion  |             |

In the `ContextActionMap` class, there are two member fields that the developer must build in their application: the `mPossibleTargets` and `mPossibleContexts` lists. These are lists of objects that extend the abstract `Entity` class (described in Section 5.3.7). Only entities in these lists are used in the target/context matching (when finding the best match).

### 5.3.7 Entity.java

`Entity` is an abstract class in the system which represents anything that could be a possible target or context. Each derived class of `Entity` must set the following fields: `mContext`, which is the context of the entity (i.e. the left-hand side value in a `ContextActionMap` described in Section 5.3.6); `mName`, which is the name of the entity (e.g. "sword", "potion", etc), and, optionally, `mDescription`, which is a string array of adjectives that are used to describe the entity. `mDescription` is optional, but it assists the `processIntent` method in the `VoiceProcess` class in identifying the best matches for targets and contexts.

Once a match for a context is found, its `mContext` string value is used to index the `ContextActionMap`, along with the best action.

### 5.3.8 Action.java

The `Action` class is an abstract class that is used to wrap the methods that should be executed in the `ContextActionMap`. Listing 7 shows the source code for the class.

Listing 7: Action.java

```
1 public abstract class Action {
    private static Entity sCurrentContext = null;
3     public abstract String execute(GlobalState state, Entity currentTarget);
    protected boolean mWantsReply = false;
5     protected Entity mCurrentTarget;

7     public boolean wantsReply() { return mWantsReply; }
    public void setWantsReply(boolean wantsReply) { mWantsReply = wantsReply; }

9     public Object processReply(GlobalState state, String input) {
11         return "Intent not understood.";
    }

13     protected static Entity getCurrentContext() { return sCurrentContext; }

15     protected static void setCurrentContext(Entity currentContext) {
17         sCurrentContext = currentContext;
    }
19 }
```

The `execute()` method is overridden by the derived class and executed when the action is chosen in the `VoiceProcess` class (see line 39 of the pseudocode in Listing 2). The method takes as input the `GlobalState` object (so the method can update its state if needed) and the current target (if it exists).

Listing 8: HealItem.java

```
1 public class HealItem extends Action {
    public String execute(GlobalState state, Entity currentTarget) {
3         if (state instanceof GameState) {
            GameState gameState = (GameState)state;
5             String itemName = Action.getCurrentContext().getName();
            if (gameState.getInventory().hasItem(itemName)) {
7                 gameState.incPlayerHealth(100);
                //Remove healing item from inventory
9                 gameState.getInventory().remove(itemName);
                state.actionSucceeded();
11                return "You healed with a " + itemName;
            } else {
13                state.actionFailed();
                return "You don't have a " + itemName;
15            }
        }
17        state.actionFailed();
        return "Error: unknown GlobalState loaded.";
19    }
}
```

Some methods may wish to ask the user for confirmation before executing (e.g. "Are you sure you want to...?"). In this case, the `execute()` method would set the `mWantsReply` member field to `true`. On the next user intent, the input is processed by the `processReply()` method (i.e. whether the user said "yes"/"no", etc). See

the Appendix 6.9 for an example of an Action class (`OpenObject`) which overrides the `processReply()` method and asks the player if they want to progress through a door.

### 5.3.9 MultipleCommandProcess.java

Optionally, the developer can enable the ability to execute multiple commands with a single utterance. For instance, the player can say, "attack the troll with a sword and then heal with a potion"; this would execute the attacking and healing as separate intents. The can wrap their `VoiceProcess` class using a `MultipleProcessCommand` class.

The `splitInput()` method splits the user's input using the words, "and" and "then" as delimiters. It stores each result in a queue, and then processes each intent using the `executeCommand()` method, which takes the queue as input. If an intent is interrupt and more user input is required before continuing (e.g. if one of the intents is ambiguous, or an action requires a reply), then the remaining intents are stored in a partial queue, and then processed once the previous intent is resolved. See the Appendix 6.1.2 for an example of the aforementioned scenario. Also see the Appendix 6.10 for the Java code of the `executeCommand()` method.

The following is an example of using an instance of the `MultipleCommandProcess` class for the *Overworld* gameplay.

```
String overworldOutput = "";
2 Queue<String> commandQueue = mOverworldVoiceProcess.splitInput(input);
while (!commandQueue.isEmpty()) {
4     overworldOutput += mOverworldVoiceProcess.executeCommand(commandQueue)
        + ((commandQueue.isEmpty()) ? "" : "\n\n---\n\n");
6 }
return overworldOutput;
```

### 5.3.10 SentenceMapper.java

The `SentenceMapper` is used to match sentences provided by the developer which do not fit the default slot-filling structures (e.g. question-based intents). Each instance of `ContextActionMap` holds a reference to an instance of a `SentenceMapper`. Invoking the `addSentenceMatch()` in the `ContextActionMap` class will add a map of several example utterances to an action and target. The following code block shows an example in the `BattleContextActionMap` class for showing the inventory.

```
1 addSentenceMatch(new ShowDefault(), "inventory",
    "what is in my inventory",
3     "what are the contents of my bag",
    "what items do i have"
5 );
```



The sentence mapper is run at the very end of the `processInput` method of the `VoiceControl` class, if no slot-filling matching failed. A disadvantage to this is that the example sentences should not be able to be matched to the slot-filling grammar.

The `checkSentenceMatch()` method of the `SentenceMapper` class goes through each group of sentences, and calculates the average similarity of each group of sentences to the user's utterance, and matches with the highest value (if it is greater than a threshold of 0.5).

The similarity of each sentence with the utterance is based on the cosine similarity of the sentences (refer to Section 4.5.6). First, a vector of each sentence is created (which is just a map of the words to the word count) before the formula for cosine similarity is applied. Listing 9 shows the code for this method.

Listing 9: `SentenceMapper.calculateCosScore()`

```

1 private double calculateCosScore(List<String> words1, String sentence) {
    Map<String, Integer> vector1 = getVector(words1);
3     Map<String, Integer> vector2 = getVector(sentence);

5     //Get common keys
    double numerator = 0.0;
7     for (Map.Entry<String, Integer> entry1 : vector1.entrySet()) {
        for (Map.Entry<String, Integer> entry2 : vector2.entrySet()) {
9             if (entry1.getKey().equals(entry2.getKey())) {
                numerator += entry1.getValue() * entry2.getValue();
11            } else {
                numerator += 0.25*min(SemanticSimilarity.getInstance().
                    calculateScore(
13                        entry1.getKey(), entry2.getKey()) *
                            (entry1.getValue() * entry2.getValue()), 1.0);
15            }
        }
17    }

19    double sum1 = 0.0;
    for (Map.Entry<String, Integer> entry1 : vector1.entrySet()) {
21        sum1 += pow(entry1.getValue(), 2);
    }

23    double sum2 = 0.0;
    for (Map.Entry<String, Integer> entry1 : vector2.entrySet()) {
25        sum2 += pow(entry1.getValue(), 2);
27    }

29    double denominator = sqrt(sum1) + sqrt(sum2);

31    if (denominator <= 0.0) { return 0.0; }
    else { return numerator / denominator; }
33 }

```

### 5.3.11 Synonym Mapping Implementation

Synonyms are used to aid the `VoiceProcess` system to match more words that the semantic similarity engine does not match automatically (due to a low similarity score). For example, the word, 'obliterate' has a low similarity score to 'attack' (below the threshold) even though they are very similar. Therefore, either the player or developer can add this mapping such that 'obliterate' maps to 'attack'.

The following are the possible ways of adding a synonym mapping:

- Add the synonym in the code (e.g. `addSynonym("obliterate", "attack")`).
- The synonym is added when the same ambiguous intent is spoken twice (e.g. when the system asks the user, "Did you mean, 'attack'?" twice, and the user replies with "yes" twice).
- The player defines the mapping as an intent (i.e. they say, "obliterate means attack"). See line 17 of the pseudocode in Listing 2 on page 36.

## 5.4 Role-playing Game Implementation

Simple rooms were created to demonstrate the system working with a typical text-based role-playing game. The first room of the game focuses on the *Overworld* gameplay, and has a door which leads to a battle with an enemy (transitioning to the *Battle* mode).

### 5.4.1 GameState.java

`GameState` is derived from `GlobalState` and acts as the 'environment' for the game; it contains all the game's objects and handles the current state.

Since there are two distinct modes of gameplay (the *Battle* gameplay and the *Overworld* gameplay), there is a `ContextActionMap` for each one in the `GameState` class. It also keeps track of various data such as the current game mode; the current room/enemy; and the player's health.

The `GameState` class also has an instance of the `Inventory` class. This is a list of items currently in the player's possession. The player can transfer items from their inventory to the room and vice versa.

### 5.4.2 Battle Mode

In the *Battle* mode, the player is able to attack the enemy with a weapon in their inventory. If no weapon is specified in the intent (as the context), then a random

weapon is selected from the inventory, and if no weapon is available, the player attacks the enemy with their bare hands. The player can also heal using any healing items they have (e.g. potions, elixers, etc), as well as look at their inventory or the available actions. Figure 16 shows the 'table' of action mappings for the *Battle* mode. See the Appendix 6.11 for the full `BattleContextActionMap` class (including the helper features such as synonym maps and sentence matches).

```

setActionList(Arrays.asList(      "attack",      "heal",      "show",      "look"));
addDefaultContextActions(        new AttackDefault(), new HealDefault(), new ShowDefault(), new LookDefault());
addContextActions("weapon",      new AttackWeapon(), null, null, null);
addContextActions("weapon-sharp", new AttackWeaponSharp(), null, null, null);
addContextActions("weapon-blunt", new AttackWeaponBlunt(), null, null, null);
addContextActions("healing-item", null, new HealItem(), null, null);

```

Figure 16: A snapshot of the context-action 'table' in `BattleContextActionMap.java`.

Weapons can be classified as either "sharp" or "blunt" based on whether they are instantiated with a description containing one of those words - each type of weapon as has different multiplier for the damage caused to the enemy.

Enemies are entities with their own health that the player fights. The enemy in the demo is a troll that can attack the enemy once the player has performed a successful action.

See the Appendix 6.1 for screenshots of examples of actions performed during the *Battle* mode

## **6 Appendix**

### **6.1 Android Application Screenshots**

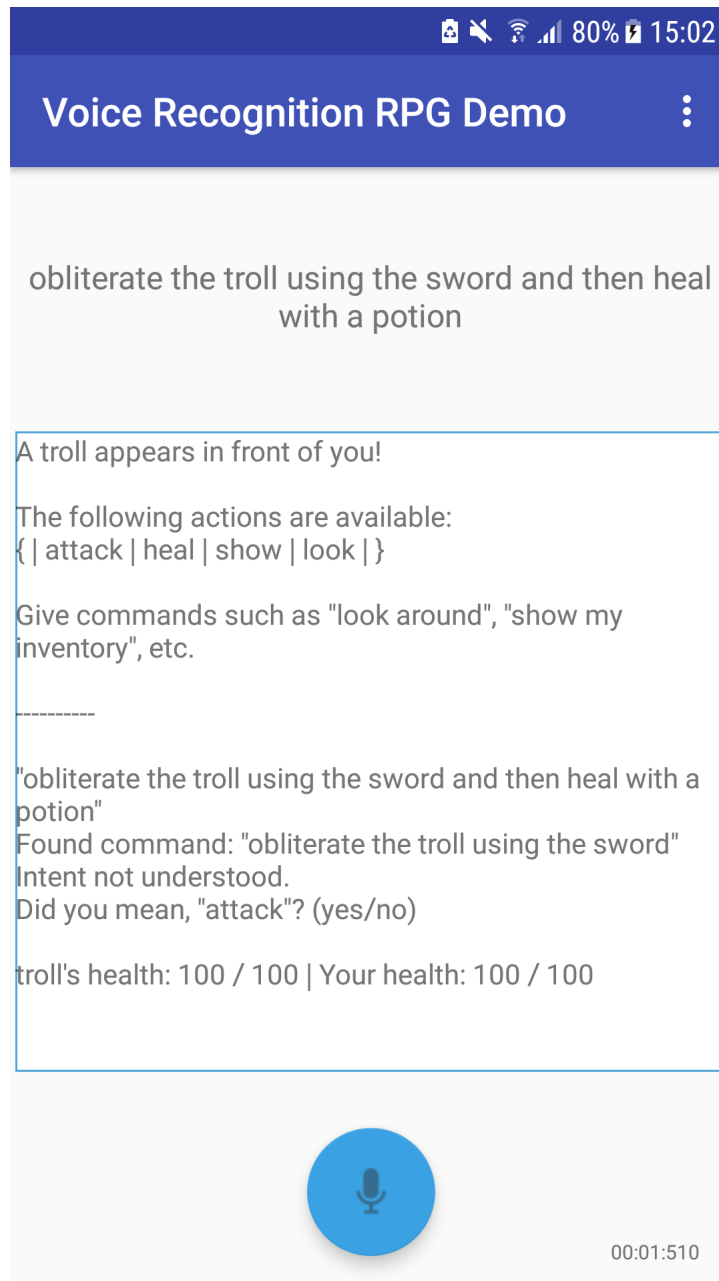
This section contains snapshots of the Android application during different scenarios.

#### **6.1.1 Battle Mode Example**

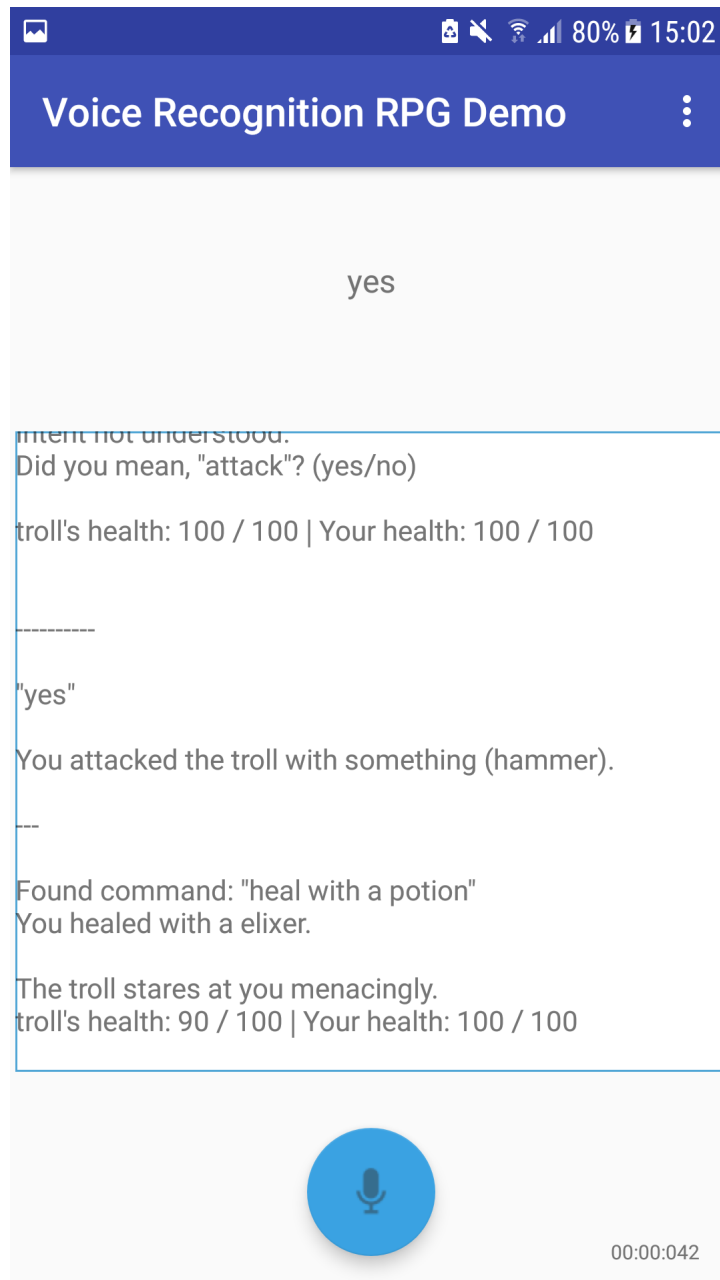
Below are screenshots of actions performed as the player battles a troll enemy.

### 6.1.2 Multiple Commands Example

The following is an example of a multiple-command intent being interrupted and then continued. In the first snapshot, the the system asks the user to resolve the first intent ("obliterate the troll using the sword"):



Once the user resolves the first intent, it executes it followed by the second intent ("heal with a potion"):



## 6.2 Skyrim Kinect Command List

Below is a list of all 200 commands that can be used in the video game, The Elder Scrolls V: Skyrim, when a Microsoft Kinect peripheral is connected [4].



FAVORITES MENU & HOTKEY EQUIPPING

ASSIGN <hotkey command> sets the selected item to the spoken hotkey. <hotkey command> can be any of the following:

• HEALTH POTION

• MAGICKA POTION

• STAMINA POTION

• POISON

• SWORD

• MACE

• AXE

• DAGGER

• BOW

• SHIELD

• DUAL WIELD LEFT

• DUAL WIELD RIGHT

• SOUL TRAP

• GREATSWORD

• BATTLEAXE

• WARHAMMER

• FIRE SPELL

• FROST SPELL

• LIGHTNING SPELL

• WARD SPELL

• RITUAL SPELL

• BOUND WEAPON

• SUMMON SPELL

• ARMOR SPELL

• CALM SPELL

• FRENZY SPELL

• HEALING SPELL

• LIGHT

Only usable in the Favorites menu.

EQUIP  
<hotkey command>

Equip item in the default hand

EQUIP LEFT  
<hotkey command>

Equip item in the Left hand, if possible.

EQUIP RIGHT  
<hotkey command>

Equip item in the Right hand, if possible.

EQUIP DUAL  
<hotkey command>

Equip the item in both hands, if possible.

EQUIP SWORD AND SHIELD/  
EQUIP MACE AND SHIELD/  
EQUIP AXE AND SHIELD/  
EQUIP DAGGER AND SHIELD

Equip the item assigned to the Sword / Mace /  
Axe / Dagger hotkey in the right hand, and the  
item assigned to the Shield hotkey in the left hand.

EQUIP DUAL WEAPONS

Equip the item assigned to the DualWieldLeft  
hotkey in the left hand, and the item assigned to  
the DualWieldRight hotkey in the left hand.

Only usable during main gameplay.

FOLLOWER COMMANDS

ALLY <action command>

Tells the ally to activate the thing you're looking at, as if you entered Command Mode and selected it.

FOLLOW / FOLLOW ME

Aly follows you

WAIT / WAIT HERE

Aly stays where they are

TRADE / ITEMS

Opens the Teammate Menu

OPEN

Aly opens the container/door you're looking at

STAND

Aly stands at the spot you're looking at

RETRIEVE

Aly picks up the item you're looking at

ATTACK

Aly attacks the NPC you're looking at

INTERACT / USE

Does a generic interact command with the ally.  
Same as if you entered Command mode and  
pressed **Q**. What the ally does is dependent on  
what you're looking at.

ITEMS MENU

After opening ITEMS, the following commands open the corresponding categories:

• FAVORITES

• ALL

• WEAPONS

• APPAREL / ARMOR

• POTIONS

• SCROLLS

• FOOD

• INGREDIENTS

• BOOKS

• KEYS

• MISCELLANEOUS

After opening any of the above sub-menu categories, you can use the following to sort your items.

SORT BY NAME

Sorts the item list by name, increasing.  
If the list is already sorted by name increasing,  
sorts it decreasingly.

SORT BY WEIGHT

Sorts the item list by weight, decreasing.  
If the list is already sorted by weight  
decreasingly, sorts it increasingly.

SORT BY VALUE

Sorts the item list by value, decreasing.  
If the list is already sorted by value decreasingly,  
sorts it increasingly.

CLOSE MENU

Closes the menu

MAGIC MENU

After opening MAGIC, the following opens the menu to the corresponding category:

FAVORITES  
ALL

ALTERATION  
ILLUSION  
DESTRUCTION

CONJURATION  
RESTORATION  
SHOUTS

POWERS  
ACTIVE EFFECTS /  
EFFECTS

CHARACTER MENU

After opening CHARACTER, the following opens the menu to the corresponding category:

ITEMS

MAGIC

SKILLS

MAP

MAP MENU

After opening MAP, the following centers the camera on the corresponding locations:

WINDHELM  
FALKREATH  
DAWNSTAR

MORTHAL  
SOLITUDE  
MARKARTH

RIFTEN  
WHITERUN  
WINTERHOLD

PLAYER / WHERE AM I  
Centers camera on you.

WAYPOINT  
Centers camera on marker set by you,  
if applicable.

QUEST MARKER / QUEST  
Centers camera on active quest target. Saying it  
multiple times will cycle through targets.

SKILLS MENU

After opening SKILLS, use the following commands to center the camera on that skill's perk tree:

ONE HANDED

HEAVY ARMOR

ALCHEMY

ILLUSION

TWO HANDED

LIGHT ARMOR

SPEECH

RESTORATION

ARCHERY

PICKPOCKET

ALTERATION

ENCHANTING

BLOCK

LOCKPICKING

CONJURATION

SMITHING

SNEAK

DESTRUCTION

52



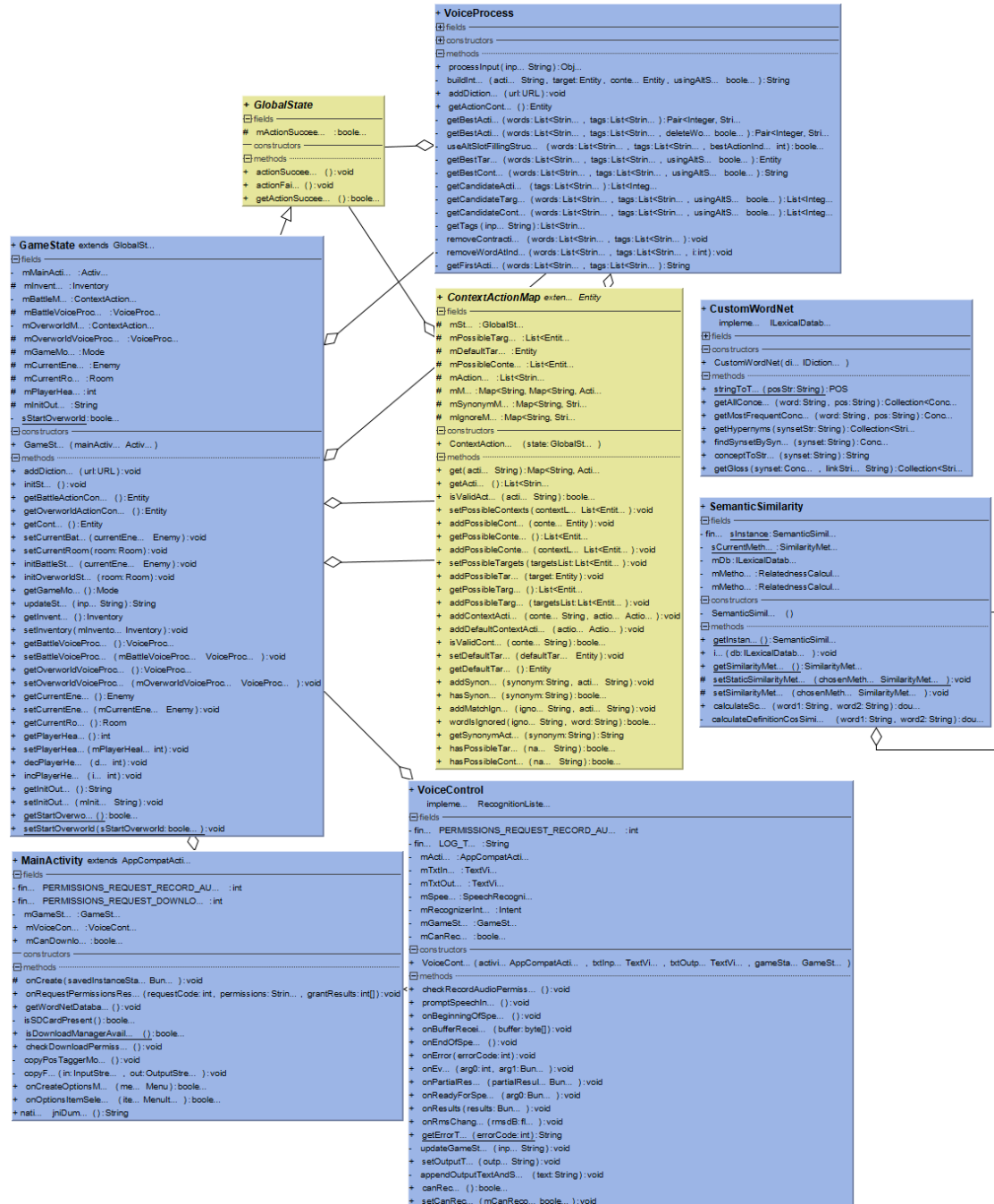
### 6.3 Penn Treebank Tagset

Below is a table of the 32 tags from the Penn Treebank tagset, used for POS tagging. The 12 remaining tags for punctuation and other symbols have been left out [36].

|       |  |      |                                       |
|-------|--|------|---------------------------------------|
| CC    | Coordinating conjunction                 | TO   | to                                    |
| CD    | Cardinal number                          | UH   | Interjection                          |
| DT    | Determiner                               | VB   | Verb, base form                       |
| EX    | Existential there                        | VBD  | Verb, past tense                      |
| FW    | Foreign word                             | VBG  | Verb, gerund or present participle    |
| IN    | Preposition or subordinating conjunction | VBN  | Verb, past participle                 |
| PRP\$ | Possessive pronoun                       | NNS  | Noun, plural                          |
| RB    | Adverb                                   | NNP  | Proper noun, singular                 |
| RBR   | Adverb, comparative                      | NNPS | Proper noun, plural                   |
| RBS   | Adverb, superlative                      | PDT  | Predeterminer                         |
| RP    | Particle                                 | POS  | Possessive ending                     |
| SYM   | Symbol                                   | PRP  | Personal pronoun                      |
| JJ    | Adjective                                | VBP  | Verb, non-3rd person singular present |
| JJR   | Adjective, comparative                   | VBZ  | Verb, 3rd person singular present     |
| JJS   | Adjective, superlative                   | WDT  | Wh-determiner                         |
| LS    | List item marker                         | WP   | Wh-pronoun                            |
| MD    | Modal                                    | WP\$ | Possessive wh-pronoun                 |
| NN    | Noun, singular or mass                   | WRB  | Wh-adverb                             |

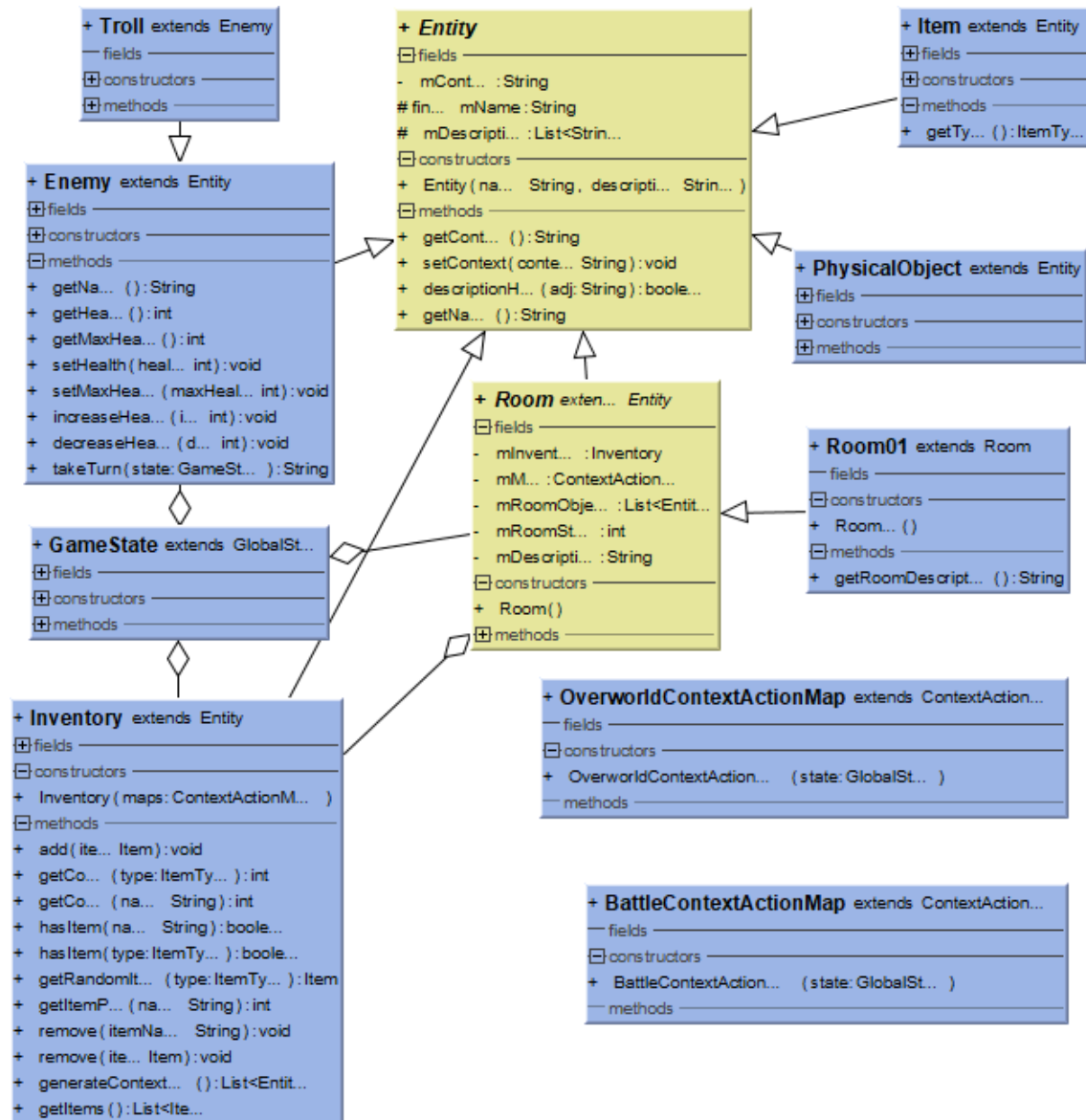
## 6.4 System UML Class Diagram

Below is the UML class diagram for the voice recognition system used in the Android app. This was automatically generated using the third-party *simpleUML* plugin for Android Studio.



## 6.5 Game Design UML Class Diagram

Below is the UML class diagram for the game mechanics - both the *Overworld* and *Battle* game modes. Generated using the third-party *simpleUML* Android plugin.



## 6.6 Choosing the Slot-filling Grammar

The code belows shows the method for choosing which slot-filling grammar to use.

Listing 10: useAltSlotFillingStructure()

```
1 private boolean useAltSlotFillingStructure(  
2     List<String> words, List<String> tags, int bestActionIndex)  
3 {  
4     List<String> keywords = Arrays.asList("use", "using", "with", "utilise");  
5     for (String keyword : keywords) {  
6         if (words.contains(keyword)) {  
7             int keywordIndex = words.indexOf(keyword);  
8             if (keywordIndex < bestActionIndex) { //If WITH CONTEXT ACTION TARGET  
9                 removeWordAtIndex(words, tags, keywordIndex);  
10                return true;  
11            }  
12        }  
13    }  
14    return false;  
15 }
```

## 6.7 Finding the Best Action in VoiceProcess.java

Listing 11: MainActivity.getBestAction()

```
1 private Pair<Integer, String> getBestAction(  
    List<String> words, List<String> tags, boolean deleteWord)  
3 {  
    mAmbiguousHandler.initAmbiguousActionCandidates();  
5    List<Integer> candidateActions = getCandidateActions(tags);  
    double bestScore = ACTION_MIN;  
7    int bestIndex = -1;  
    String bestAction = "<none>";  
9    List<String> actionsList = mContextActionMap.getActions();  
    for (int i: candidateActions) {  
11       String word = words.get(i);  
        //ignore with/use words  
13       if (!(word.equals("use") || word.equals("with") || word.equals("using") ||  
            words.contains("utilise"))) {  
15           if (mContextActionMap.hasSynonym(word)) {  
               if (deleteWord) { removeWordAtIndex(words, tags, i); }  
17           List<String> synonyms = mContextActionMap.getSynonymMapping(word);  
               if (synonyms.size() > 1){  
19               //Ambiguous synoymys - ask user about each one  
               for (String action : synonyms) {  
21                   mAmbiguousHandler.setIsAmbiguous(true);  
                   mAmbiguousHandler.addAmbiguousActionCandidate(  
23                       new Triple<>(word, action, 1.0), bestScore);  
               }  
25           }  
           return new Pair<>(i, synonyms.get(0));  
27       }  
       for (String action : actionsList) {  
29           if (mContextActionMap.wordIsIgnored(word, action)) {  
               continue;  
31           }  
           if (word.equals(action)) {  
33               if (deleteWord) { removeWordAtIndex(words, tags, i); }  
               return new Pair<>(i, action);  
35           }  
       }  
37       for (String action : actionsList) {  
           if (mContextActionMap.wordIsIgnored(word, action) ||  
39               mContextActionMap.hasPossibleTarget(word) ||  
               mContextActionMap.hasPossibleContext(word))  
41           { continue; }  
           double score = SemanticSimilarity.getInstance().calculateScore(  
               action, word);  
43           if (score > ACTION_MIN && score < ACTION_CONFIDENT) {  
               mAmbiguousHandler.addAmbiguousActionCandidate(  
45                   new Triple<>(word, action, score), bestScore);  
           }  
47           if (score > bestScore) {  
               bestScore = score;  
49               bestIndex = i;  
               bestAction = action;  
51           }  
       }  
53     }  
55 }  
  
57 if (bestIndex > -1) {  
    if (bestScore > ACTION_MIN && bestScore < ACTION_CONFIDENT) {  
        mAmbiguousHandler.setIsAmbiguous(true);  
    }  
}
```

```
59     }  
    else { mAmbiguousHandler.clearAmbiguousActionCandidates(); }  
61    //Remove chosen action from list inputs  
    if (deleteWord) { removeWordAtIndex(words, tags, bestIndex); }  
63 } else { mAmbiguousHandler.clearAmbiguousActionCandidates(); }  
  
65    return new Pair<>(bestIndex, bestAction);  
}
```

## 6.8 JwiLexicalDatabase.java

An implementation of ILexicalDatabase for the WS4J API, using MIT's JWI library.

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import edu.cmu.lti.jawjaw.pobj.Link;
import edu.cmu.lti.lexical_db.ILexicalDatabase;
import edu.cmu.lti.lexical_db.data.Concept;
import edu.cmu.lti.ws4j.util.PorterStemmer;
import edu.cmu.lti.ws4j.util.WS4JConfiguration;
import edu.mit.jwi.IDictionary;
import edu.mit.jwi.item.IIndexWord;
import edu.mit.jwi.item.ISynset;
import edu.mit.jwi.item.ISynsetID;
import edu.mit.jwi.item.IWord;
import edu.mit.jwi.item.IWordID;
import edu.mit.jwi.item.POS;
import edu.mit.jwi.item.Pointer;
import edu.mit.jwi.item.SynsetID;

/**
 * Created by Baron on 20/01/2018.
 */

// Alternative to the NictWordNet class, which is included in the WS4J library.
// https://github.com/Sciss/ws4j/blob/master/src/main/java/edu/cmu/lti/lexical_db/
// NictWordNet.java
// NictWordNet does not work because it uses some sqlite methods that do not work
// in Android.
// This is a custom class that attempts to implement the same thing using MIT's JWI
// library.
public class JwiLexicalDatabase implements ILexicalDatabase {
    public IDictionary mDict;
    public Map<String, ISynset> synsetMap = new HashMap<>();
    private static PorterStemmer stemmer;

    public static POS stringToTag(String posStr) {
        if (posStr.equals("n")) { return POS.NOUN; }
        else if (posStr.equals("v")) { return POS.VERB; }
        else if (posStr.equals("r")) { return POS.ADVERB; }
        else if (posStr.equals("a")) { return POS.ADJECTIVE; }
        else { return POS.NOUN; }
    }

    public JwiLexicalDatabase(IDictionary dict) {
        mDict = dict;
    }

    public Collection<Concept> getAllConcepts(String word, String pos) {
        List<Concept> synsetStrings = new ArrayList<>();
        IIndexWord idxWord = mDict.getIndexWord(word, stringToTag(pos));
        if (idxWord != null) {
            List<IWordID> wordIDs = idxWord.getWordIDs();
            for (IWordID id : wordIDs) {
                String synsetStr = id.getSynsetID().toString().replace("SID-", "").
                    toLowerCase();
            }
        }
    }
}
```

```

56         synsetStrings.add(new Concept(synsetStr, edu.cmu.lti.jawjaw.pobj.
            POS.valueOf(pos)));
            IWord w = mDict.getWord(id);
58         synsetMap.put(synsetStr, w.getSynset());
        }
60     }
    return synsetStrings;
62 }

64 public Concept getMostFrequentConcept(String word, String pos) {
    Collection<Concept> concepts = getAllConcepts(word, pos);
66     if (concepts.size() > 0) { return concepts.iterator().next(); }
    return null;
68 }

70 public Collection<String> getHypernyms(String synsetStr) {
    List<String> synsetStrings = new ArrayList<>();
72     ISynset synset = synsetMap.get(synsetStr);
    if (synset != null) {
74         List<ISynsetID> hypernymsList = synset.getRelatedSynsets(Pointer.
            HYPERNYM);

76         for (ISynsetID id : hypernymsList) {
            String hypernymStr = id.toString().replace("SID-", "").toLowerCase
                ();
78             //Need to add this synset to the map
            List<IWord> words = mDict.getSynset(id).getWords();
80             for (IWord w : words) {
                synsetMap.put(hypernymStr, w.getSynset());
82             }
            synsetStrings.add(hypernymStr);
84         }
    }
    return synsetStrings;
86 }

88 public Concept findSynsetBySynset( String synset ) {
    return null;
90 }

92 // To offset.
94 public String conceptToString( String synset ) {
    return null;
96 }

98 // Note: most of this method is similar to the NictWordNet one, but using JWI
    instead
    // https://github.com/Sciss/ws4j/blob/master/src/main/java/edu/cmu/lti/
    lexical_db/NictWordNet.java
100 public Collection<String> getGloss( Concept synset, String linkString ) {
    String synsetStr = synset.getSynset();
102
    char posTag = '*';
104     Pattern p = Pattern.compile("-[a-zA-Z]");
    Matcher m = p.matcher(synsetStr.toLowerCase());
106     if (m.find()) { posTag = m.group(0).charAt(1); }

108     if (posTag == '*') { return new ArrayList<>(); }

110     int synsetOffset = Integer.parseInt(synsetStr.replaceAll("-[a-zA-Z]", ""));
    SynsetID synsetID = new SynsetID(synsetOffset, POS.getPartOfSpeech(posTag))
        ;
112     ISynset synsetJWI = mDict.getSynset(synsetID);

```



```

114 //Build up pointer to synsets (link = pointer)
115 List<ISynsetID> linkedSynsets = new ArrayList<>();
116 Link link = null;
117 try {
118     link = Link.valueOf(linkString);
119     if (link.equals(Link.mero)) {
120         linkedSynsets.addAll(synsetJWI.getRelatedSynsets(Pointer.
121             MERONYM_MEMBER));
122         linkedSynsets.addAll(synsetJWI.getRelatedSynsets(Pointer.
123             MERONYM_SUBSTANCE));
124         linkedSynsets.addAll(synsetJWI.getRelatedSynsets(Pointer.
125             MERONYM_PART));
126     } else if (link.equals(Link.holo)) {
127         linkedSynsets.addAll(synsetJWI.getRelatedSynsets(Pointer.
128             HOLONYM_MEMBER));
129         linkedSynsets.addAll(synsetJWI.getRelatedSynsets(Pointer.
130             HOLONYM_SUBSTANCE));
131         linkedSynsets.addAll(synsetJWI.getRelatedSynsets(Pointer.
132             HOLONYM_PART));
133     } else if (link.equals(Link.syns)) {
134         linkedSynsets.add(synsetJWI.getID());
135     } else {
136         linkedSynsets.addAll(synsetJWI.getRelatedSynsets());
137     }
138 } catch (IllegalArgumentException e) { linkedSynsets.add(synsetJWI.getID())
139     ; }

140 List<String> glosses = new ArrayList<>();
141 for (ISynsetID linkedSynsetID : linkedSynsets) {
142     String gloss = null;
143     ISynset linkedSynsetJWI = mDict.getSynset(linkedSynsetID);
144     if (Link.syns.equals(link)) {
145         gloss = synset.getName();
146     } else {
147         gloss = linkedSynsetJWI.getGloss();
148     }

149     if (gloss == null) { continue; }

150     gloss = gloss.replaceAll("[.;:?!(){}\\\"'${}%<>]", " ");
151     gloss = gloss.replaceAll("&", " and ");
152     gloss = gloss.replaceAll("-", " ");
153     gloss = gloss.replaceAll("[ ]+", " ");
154     gloss = gloss.replaceAll("(?<!\w)", " ");
155     gloss = gloss.replaceAll("'(?!\w)", " ");
156     gloss = gloss.replaceAll("--", " ");
157     gloss = gloss.toLowerCase();

158     if ( WS4JConfiguration.getInstance().useStem() ) {
159         gloss = stemmer.stemSentence( gloss );
160     }

161     glosses.add( gloss );
162 }

163 return glosses;
164 }

165 private List<String> clone( List<String> original ) {
166     return new ArrayList<String>( original );
167 }
168 }

```

## 6.9 OpenObject.java

```
2 public class OpenObject extends Action {
3     public String execute(GlobalState state, Entity currentTarget) {
4         if (state instanceof GameState) {
5             GameState gameState = (GameState) state;
6             if (gameState.getCurrentRoom().hasRoomObject(currentTarget.getName()))
7             {
8                 if (currentTarget instanceof Door) {
9                     state.actionSucceeded();
10                    mWantsReply = true;
11                    mCurrentTarget = currentTarget;
12                    return "Are you sure you want to leave this room? (yes/no)";
13                } else if (currentTarget instanceof GlassTable) {
14                    state.actionSucceeded();
15                    return "You opened the drawer of the glass table but nothing is
16                        inside. You"
17                        + " closed it again.";
18                } else {
19                    state.actionFailed();
20                    return "You cannot open the "+currentTarget.getName()+".";
21                }
22            } else {
23                state.actionFailed();
24                return "You cannot open the "+currentTarget.getName()+".";
25            }
26        }
27        state.actionFailed();
28        return "You can't do that right now";
29    }
30
31    @Override
32    public Object processReply(GlobalState state, String input) {
33        if (mCurrentTarget != null) {
34            if (VoiceProcess.replyIsYes(input) && mCurrentTarget instanceof Door) {
35                return ((Door) mCurrentTarget).onOpened((GameState) state);
36            }
37            return "You did not open the " + mCurrentTarget.getName()+".";
38        }
39        return "Intent not understand.";
40    }
41 }
```

## 6.10 MultipleCommandProcess.executeCommand()

Listing 12: MultipleCommandProcess.executeCommand()

```
1 public Object executeCommand(Queue<String> queue) {
2     if (queue == null || queue.size() <= 0) { return null; }
3
4     String command = queue.peek();
5     if (command != null) {
6         if (mVoiceProcess.isExpectingReply()) {
7             queue.clear(); //ignore the rest of the input
8             String result = "";
9             result += mVoiceProcess.processInput(command); //process
10                confirmation
11             if (!mVoiceProcess.isExpectingReply()) {
12                 String nextCommand = mPartialQueue.peek();
13                 if (nextCommand != null) {
14                     mPartialQueue.remove();
15                     result += "\n\n--\n\n" + "Found command: \"" + nextCommand + "
16                        \"\n"
17                     + mVoiceProcess.processInput(nextCommand);
18                     mVoiceProcess.setExpectingMoreInput(mPartialQueue.size() > 0);
19                 }
20             }
21             return result;
22         } else {
23             queue.remove();
24             Object result = "Found command: \"" + command + "\"\n\n"
25                + mVoiceProcess.processInput(command);
26             if (mVoiceProcess.isExpectingReply()) {
27                 mPartialQueue = new LinkedList<>(queue);
28                 queue.clear();
29             }
30             mVoiceProcess.setExpectingMoreInput(queue.size() > 0
31                || (mPartialQueue != null && mPartialQueue.size() > 0));
32             return result;
33         }
34     } else { return null; }
35 }
```

## 6.11 BattleContextActionMap.java

Listing 13: BattleContextActionMap

```
2 public class BattleContextActionMap extends ContextActionMap {
3     public BattleContextActionMap(GlobalState state) {
4         super(state);
5         setActionList(Arrays.asList("attack", "heal",
6                                     "show", "look"));
7         addDefaultContextActions(new AttackDefault(), new
8                                 HealDefault(), new ShowDefault(), new LookDefault());
9         addContextActions("weapon", new AttackWeapon(), null,
10                          null, null);
11         addContextActions("weapon-sharp", new AttackWeaponSharp(), null,
12                          null, null);
13         addContextActions("weapon-blunt", new AttackWeaponBlunt(), null,
14                          null, null);
15         addContextActions("healing-item", null, new
16                          HealItem(), null, null);
17
18         addSynonym("punch", "attack");
19         addSynonym("assault", "attack");
20         addSynonym("observe", "look");
21         addSynonym("blade", "knife");
22
23         addMatchIgnore("jump", "look");
24         addMatchIgnore("jump", "attack");
25         addMatchIgnore("bag", "attack");
26
27         addSentenceMatch(new ShowDefault(), "inventory",
28                          "what is in my inventory",
29                          "what are the contents of my bag",
30                          "what items do i have");
31
32         addSentenceMatch(new ShowDefault(), "actions",
33                          "what can i do",
34                          "what are my actions",
35                          "what are the commands",
36                          "what action can i do",
37                          "what are my options");
38     }
39 }
```

## References

- [1] Marco Tabini, “Inside Siri’s brain: The challenges of extending Apple’s virtual assistant,” 2013, April 8. [Online]. Available: <https://www.macworld.com/article/2033073/inside-siris-brain-the-challenges-of-extending-apples-virtual-assistant.html>
- [2] Indomitus Games, “In Verbis Virtus.” [Online]. Available: <http://www.indomitusgames.com/in-verbis-virtus>
- [3] Bethesda Softworks LLC, “The Elders Scrolls Official Site,” 2018, January 1. [Online]. Available: <https://elderscrolls.bethesda.net/en/skyrim>
- [4] NowGamer, “Skyrim Kinect: Full List Of 200 Voice Commands,” 2012, May 1. [Online]. Available: <https://www.nowgamer.com/skyrim-kinect-full-list-of-200-voice-commands/>
- [5] Ubisoft, “Star Trek Bridge Crew,” 2017. [Online]. Available: <https://www.ubisoft.com/en-gb/game/star-trek-bridge-crew/>
- [6] Chris Watters, “Star Trek: Bridge Crew Integrates IBM Watson For Voice Commands,” 2017, June 22. [Online]. Available: <https://news.ubisoft.com/article/star-trek-bridge-crew-integrates-ibm-watson-voice-commands>
- [7] developerWorks, “Star Trek: Bridge Crew now live with IBM Watson Speech Technology,” 2017, June 22. [Online]. Available: <https://developer.ibm.com/tv/dwnnewsblast-watson-star-trek/>
- [8] Rob Lammle, “Eaten by a Grue: A Brief History of Zork,” June 2014. [Online]. Available: <http://mentalfloss.com/article/29885/eaten-grue-brief-history-zork>
- [9] Jacob Foster and Matt Thompson and Myles Loffler, “Classic Zork on Alexa,” Dec 2016. [Online]. Available: <https://www.hackster.io/devops-dungeoneers/classic-zork-494ff1>
- [10] Dave Lebling and Marc Blank, *Zork Trilogy Instruction Manual*. InfoCom, 1984. [Online]. Available: <http://infodoc.plover.net/manuals/zork1.pdf>
- [11] VoiceTechGroup, “tazti — Speech Recognition for PC Games,” 2018, January 1. [Online]. Available: <https://www.tazti.com/speech-recognition-software-for-pc-games.html>
- [12] Epic Wiki, “Speech Recognition Plugin,” 2018, March 17. [Online]. Available: [https://wiki.unrealengine.com/Speech\\_Recognition\\_Plugin](https://wiki.unrealengine.com/Speech_Recognition_Plugin)

- [13] CMU Sphinx, “PocketSphinx,” 2015. [Online]. Available: <https://github.com/cmusphinx/pocketsphinx>
- [14] SoundHound, “Houndify,” 2017. [Online]. Available: <https://www.houndify.com/>
- [15] —, “The ClientMatch JSON Format,” 2017. [Online]. Available: <https://docs.houndify.com/reference/ClientMatch>
- [16] IBM, “Conversation - IBM Cloud,” 2017, December 12. [Online]. Available: <https://console.bluemix.net/catalog/services/conversation>
- [17] “IBM Cloud Docs — Conversation — About,” 2018, January 5. [Online]. Available: <https://console.bluemix.net/docs/services/conversation/index.html>
- [18] Zach Walchuk, “Build a chatbot in ten minutes with Watson,” Dec 2016. [Online]. Available: <https://www.ibm.com/blogs/watson/2016/12/build-chatbot/>
- [19] IBM, “Conversation Demo,” 2017, December 18. [Online]. Available: <https://conversation-demo.ng.bluemix.net/>
- [20] “Dialogflow.” [Online]. Available: <https://dialogflow.com/>
- [21] Nitin Indurkha and Fred Damerau, *Classical Approaches to Natural Language Processing*, 2 ed., ser. Handbook of Natural Language Processing. United States: Chapman and Hall, 2010, pp. 4–6.
- [22] Craig Trim, “The Art of Tokenization,” Jan 2013. [Online]. Available: <https://ibm.com/developerworks/community/blogs/nlp/entry/tokenization>
- [23] Nitin Indurkha and Fred Damerau, *Classical Approaches to Natural Language Processing*, 2 ed., ser. Handbook of Natural Language Processing. United States: Chapman and Hall, 2010, p. 61.
- [24] —, *Part-of-Speech Tagging*, 2 ed., ser. Handbook of Natural Language Processing. United States: Chapman and Hall, 2010, pp. 205–206.
- [25] Dictionary.com2018, “The Dictionary of American Slang,” 2018, January 18. [Online]. Available: <http://www.dictionary.com/browse/can>
- [26] D. Jurasky and J. Martin, *Dialog Systems and Chatbots*, ser. Speech and Language Processing, 2017. [Online]. Available: <https://web.stanford.edu/jurafsky/slp3/29.pdf>
- [27] WikiDiff, “Synonym vs Hypernym - What’s the Difference?” [Online]. Available: <https://wikidiff.com/synonym/hypernym>

- [28] Princeton University, “Dog — WordNet Search - 3.1,” 2018, January 18. [Online]. Available: <http://wordnetweb.princeton.edu/>
- [29] Bridget McInnes and Ted Pedersen, “Evaluating measures of semantic similarity and relatedness to disambiguate terms in biomedical text,” *Journal of Biomedical Informatics*, 2013, Sep 2013. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1532046413001238>
- [30] Sagar Gole, “Words similarity/relatedness using WuPalmer Algorithm - Sagar Gole’s Blog,” Jun 2015. [Online]. Available: <http://blog.thedigitalgroup.com/sagarg/2015/06/10/words-similarityrelatedness-using-wupalmer-algorithm/>
- [31] Hideki Shima, “Juice, Water — WS4J Demo,” 2018, January 18. [Online]. Available: <http://ws4jdemo.appspot.com>
- [32] Mohit Iyyer and Jordan Boyd-Graber and Hal Daume III, “Generating Sentences from Semantic Vector Space Representations,” 2014.
- [33] L. White and R. Togneri and W. Liu and M. Bennamoun, “Modelling Sentence Generation from Sum of Word Embedding Vectors as a Mixed Integer Programming Problem,” in *2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*, 2016, pp. 770–777, ID: 1.
- [34] Princeton University, “About WordNet,” 2013, January 9 2013. [Online]. Available: <http://wordnet.princeton.edu>
- [35] Mark Bozon, “World Debut: Scribblenauts,” 2008, December 5. [Online]. Available: <http://uk.ign.com/articles/2008/12/05/world-debut-scribblenauts>
- [36] University of Pennsylvania, “Penn Treebank P.O.S. Tags,” 2018. [Online]. Available: [https://ling.upenn.edu/courses/Fall\\_2003/ling001/penn\\_treebank\\_pos.html](https://ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html)