**NEO Project**

**Operation and Development Instructions:**

**These instructions are written to explain what the NEO project is, give a high level description of how it works, and discuss how it can be further developed. If after reading you have further questions about the project, you can email the creator of this document at** [**landon.r.shumway@gmail.com**](mailto:landon.r.shumway@gmail.com)**.**

**NEO is an attempt to simulate how people learn from experience with objects in the real world and how we can replicate that understanding within machines. The main purpose of the project was to develop an intelligent agent that could perform actions through interpreting natural language commands using a relational database of English words. Other projects may stem from this original concept but NEO’s “memory” database is designed around this particular goal.**

**As of this writing, NEO can interact with and learn about objects, learn about locations in its environment, and search for certain specified objects when given natural language commands. This is a small subset of tasks that hopefully the project will be able to build on in the future. In this document first the overall architecture of NEO will be explained. Next its “memory” design will be further dived into, and finally an explanation on how to add further functions for NEO in the future.**

**NEO’s DESIGN**

**At first, NEO’s architecture may seem somewhat confusing given that it is designed using Agent Oriented Programming principles (see appendix A for more about AOP). On the contrary, NEO was designed in this way because it is actually more intuitive to how our bodies function as a whole. We have our hands, eyes, brain, just to name a few parts, all performing unique tasks yet still connected and can communicate with one another. This section will explain these different body parts of NEO and their main purpose. As of this writing, NEO2D is our current project so this explanation will be based on the code in that project.**

**In NEO2D’s design, the Bot class in the bot.py file represents NEO’s physical body in the environment. Any movement or picking up of objects in the environment is done through the bot.**

**The Neo class in the neo.py file serves as the main hub of the body, similar to the brain. The Neo class calls the shots and has access to all the other body part classes. If you want to call a function for another body part to perform, do it through this class. It also contains the different behavioral states of Neo that can be used to influence how it behaves in certain situations.**

**The following classes are all located in the neo\_body folder**

**The Ears class in the ears.py file is responsible for interpreting sound into speech. It uses the speech\_recognition as well as the pocket\_sphinx libraries to take in speech through the microphone and translate it into text.**

**The Eyes class in the eyes.py file is responsible for processing visual information about objects and for object detection. For now, it simply reads in the text Meta data about object colors. It uses raycast objects (see raycast.py) which collide with other objects. They notify NEO when an object is hit.**

**The Hands class in the hands.py file processes weight information. It also controls the hand functions of the Bot class for carrying objects.**

**The Legs class in the legs.py file is responsible for calling the moving functions of the Bot.**

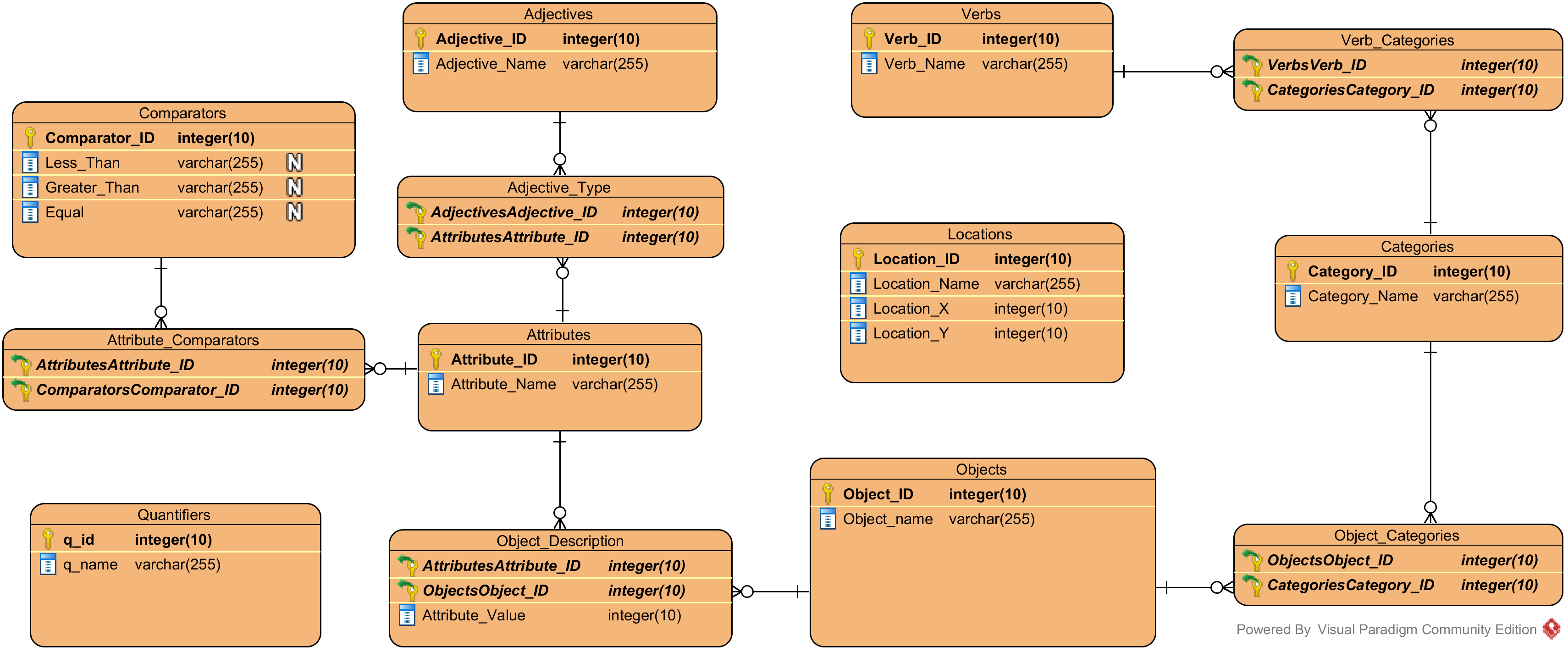
**The Memory class in the memory.py file is the main class in charge of storing information into NEO’s database. It creates the tables the first time the project is run and can be used to retrieve info when needed.**

**The Mouth class in the mouth.py file is used to convert text to speech. We have been using the pyttxs3 library but this has been having issues with most students’ laptops so it may be better to find an alternative text-to-speech library.**

**The Wernicke\_Area class in the Wernicke\_area.py file is responsible for translating natural language into SQL code to run in the database (the name of the class comes from the part of the brain responsible for speech recognition). For future development, new verbal commands should be programmed in this class. This will be explained further in the section “Performing natural language commands”.**

**NEO’s Memory**

**As of this writing, NEO’s experience with the environment is stored in a SQLITE relational database. The database file is stored internally in the project under the filename “neo\_test.db”. This section describes the different tables in the DB and their main purpose.**

****

**ADJECTIVES – the adjectives tables holds descriptive words of the English language (red, heavy, soft etc.). It is linked to the attributes table through the ADJECTIVE\_TYPE table as explained below.**

**ATTRIBUTES –the attributes table holds words that define different aspects of objects (color, weight, texture, etc.). These are linked to adjectives in order to define what type of adjective we want NEO to search for in order to find the correct information.**

**(ex. If we want to know which objects are red, NEO will need to search the ‘color’ aspect of each object because the adjective ‘red’ is a type of color.)**

**CATEGORIES – this table holds categorical words (food, weapon, money). This table is important for categorizing similar objects together (ex. apple and banana are both types of the category fruit). An object can have many categories and a category can belong to many objects, so we link the objects table and the category table together with an OBJECT\_CATEGORIES table.**

**COMPARATORS– This table holds comparative words (heavier, lighter, harder, softer, etc.). It holds a column for “greater than” words (heavier, harder) and another column for “less than” words (lighter, softer). It will be used to allow NEO to compare objects together based on a particular attribute. (ex. which objects are heavier than an orange? Heavier is linked to the attribute weight so NEO will search its objects table for objects whose weight value is greater than that of an orange.) This table is linked to the attributes table through the ATTRIBUTE\_COMPARATOR table.**

**LOCATIONS – this table holds information about locations in NEO’s environment. It holds the location name and an x and y coordinate for the location. This allows NEO to travel to different locations in the environment by forming a path from its current location to the coordinates of the saved location. In the future we hope to implement actual dimensions for each location so NEO can always determine what the name of its current location is.**

**OBJECTS – the objects table holds the names of the objects that NEO has interacted with. It is connected to the ATTRIBUTES and CATEGORIES tables through the linking tables OBJECT\_DESCRIPTION and OBJECT\_CATEGORIES, respectively. This allows an object to have many values for the same attribute (an ‘Australian Shepard’ object can have multiple colors) and belong to multiple categories (the ‘Australian Shepard’ is a type of dog and animal). These linking tables allow us to freely add new attributes in the future without needing to change the database framework.**

**QUANTIFIERS – This table holds quantity words such as ‘all’, ‘some’, ‘few’, etc. These can be used to specify the quantity of objects needed for certain tasks. (ex: Eat ‘all’ of the food)**

**VERBS – this table holds the command words that neo is able to perform. It is linked to the categories table through the VERB\_CATEGORIES table to determine which verbs can be used with which objects (ex: The verb ‘swim’ could be linked to the category ‘liquid’. If a user says “swim in this water” NEO checks the categories of water and determines it is a liquid, so it can swim in it).**

**Performing Natural Language Commands**

**This section describes how to take a natural language command and translate it into an executable command using the database framework. Consider if you were to write a command for NEO to pick up an object in a programming language, the function header might look something like this.**

**Function PICK\_UP (OBJECT object\_name)**

**Where PICK\_UP is the name of the function and object\_name is which object you want NEO to pick up. Now consider you want NEO to pick up the first object it sees with a particular attribute. You might overload the function like this.**

**Function PICK\_UP(OBJECT object\_name, ADJECTIVE adjective\_name)**

**Where adjective is the attribute value you are looking for. How can we replicate this function header using natural language? Consider the following sentence:**

***Pick up the green apple***

**This sentence contains all of the information we need to run our function using the specified parameters, OBJECT(apple) and ADJECTIVE(green). We also know which function we need to run through the words ‘pick up’. How do we know which words are the function name and which are the parameter values? The database framework: by accessing the words in the VERBS, ADJECTIVES, and OBJECTS tables, we can determine which parts of the sentence are which and what the user is asking for when they say “(pick up –verb) the (green – adjective) (apple – object)” We iterate through the sentence word for word and find matches in the tables, then return the function that is to be executed along with the necessary variables.**

**What is so powerful about using the relational database approach is as long as the proper information is given in a sentence, we can parse it into an executable command and use that command on a wide variety of objects and in a wide variety of situations. We have already seen this in NEO’s ability to group objects together based on their attributes (color, weight). Obviously there must be certain logic checks in place to make sure the object can be used for a specific task (ex. you can’t eat rocks or swim in the sky). But this can be done by categorizing objects and then labeling verbs so they can only be used with certain categories of objects (you can only eat objects labeled as food and can only swim in liquids).**

**So if you are interested in creating new functions for NEO to perform in its environment, first create a function header for the action. The function name becomes your verb, and your parameters will determine which tables you will need to search in order to find the necessary information to run the function. Then when translating the natural language sentence, simply parse the sentence word for word, searching each table in the DB for the word, and find matches for your function name and for each parameter.**

**This method should be used in cases where NEO is performing a physical action in its environment, in the case that you want to query him about relationships between objects, the process is a little different, consider these sentences:**

**Which objects are red?**

**What is the color of an apple?**

**In these cases, the words ‘which’ and ‘what’ should be considered as special types of verbs, it is a word that signifies the user wants NEO to list objects or attributes of an object. Inquiry words require us to determine what attribute is being asked for and which set of objects we are searching through for this attribute. The ‘objects’ word signifies that we are to simply search the entire table of objects, but what about a sentence written as:**

**Which fruit are yellow?**

**Now our set of objects is limited to those that have been categorized as fruit.**

**To determine which set of objects we are working with and which attribute we want to find we search through the CATEGORIES table and the ATTRIBUTES table and look for matching words in the sentence.**

**These examples should serves as a starting point for building on NEO’s current set of abilities. If you are not familiar with relational databases or primary/foreign key relationships, I highly recommend studying these topics and developing a firm understanding of the first three normal forms of databases before attempting to make any changes to the DB framework. If you understand basic SELECT statements for querying a DB you should be able to experiment with translating natural language into an executable command using SQL queries to parse the sentence.**

**Appendix A – Agent Oriented Programming**

Agent Oriented Programming (AOP) is a design pattern similar to Object Oriented Programming. It seeks to model software development after how humans perform complex tasks in groups. When we need information from one another, we simply 'ask' each other for the data. Similarly, any agent class is able to ask another agent for any needed info (see the example on the next page for how this works). AOP does this through reflection, which allows agents to ask each other for specific variables at runtime.

In object oriented programming, any class object is able to perform functions that affect the system. In the real world, humans are the only entities that perform meaningful functions, so why not create a software model that are patterned after the real world? In AOP, only agent classes perform functions.

With AOP design, there are three types of classes which all subclasses derive from: Agent, Environment, and Object. Agent subclasses are able to communicate with other agents and perform functions. Environments hold info about the system and maintain state when changes are made by agents. Objects simply hold info and are used by the agents as needed, just as humans use objects in the real world.

The most important elements about AOP design are:

**Agents cannot have their variables changed by any other class.**

**Agent variables are available to any other agent in the system.**

**Agent classes are static, there should only be one instance of any particular agent in the system.**

**Agents pass information between one another through the ask method, which in a nutshell is a fancy getter that allows any agent to get the variable of another agent in the system.**

To look at a basic contrived example of how information is passed using AOP, let’s take a real world example. Say we have two people working on a math project. They need to find the perimeter and area of a rectangle. They decide to split the project up between the two of them so one is in charge of finding the perimeter and the other works on finding the area. The first person finds the length and width of the rectangle and performs the calculation. Then the second person asks, “What is the length and width of the rectangle?” to which the first person gives them the needed information so the second person can perform the calculation to find the area of the same rectangle.

To replicate this in AOP, we could have two agent classes: rectangle\_perimeter\_finder and rectangle\_area\_finder. Neither of these two agents have a dependency in code with the other (they are in completely separate files with no import statements for each other). The first agent is working with a particular rectangle and stores variables for the rectangle width and height. Now the second agent wants to find the area of the exact same rectangle. So it asks the first for the information like so:

**Rectangle\_length = self.ask(“rectangle\_perimeter\_finder”,“rectangle\_length”)**

**Rectangle\_width = self.ask(“rectangle\_perimeter\_finder”, “rectangle\_width”)**

You can basically read what is happening here like an English sentence: ask the rectangle perimeter finder for the rectangle length. The function returns the variable value and stores it in the other variable, all without requiring a dependency in code. As long as you know the name of the agent and the name of the variable you want, you can get the information from the system.

That’s perhaps the most important element to understand in order to actually pass information around in an AOP system, if you want info, simply ask for it. If you would like to learn more about how the method works, you can check out the agent.py file in the project.