LINGUALINK

An Interactive Language Learning Tool

TABLE OF CONTENTS

Project Report

Introduction

The goal of the LinguaLink project is to develop a desktop application for language learners to diagram sentences in their target language and gain familiarity with grammatical rules. Specific requirements for this project include the ability for users to:

1. Add/remove words from the current project.
2. Move words from the word bank (inactive words) to the workspace (active words).
3. Move words around within the canvas.
4. Draw valid connections between words based on word part of speech.
5. Export an image of the current workspace diagram.
6. Save to and load from disk project configuration and state.

LinguaLink will allow language learners to get hands on, visual experience with their target language. It also provides them with immediate feedback. This serves to make the process of learning and practicing a languages grammar robust, interactive, and detailed.

Literature Review/Background Study

This project required research into similar elements of each supported language. For English, the research process began with researching all parts of speech:

- Noun
- Pronoun
- Verb
- Adjective
- Adverb
- Preposition
- Conjunction
- Interjection

The decision was made to also support articles like “a” from diagram readability. If a sentence is a graph (which strongly aligns with the application architecture), then there must be some rules with which to determine valid edges in this graph. This required research on the semantics of the English language. For application robustness, it made more sense to enforce a potentially incomplete list of invalid constructions, rather than a massive list of valid constructions. The logic is that it is preferable that LinguaLink fail to flag unspecified invalid constructions than to fail to allow unspecified valid constructions, as this would result in an application with surprisingly (to the user) few possible sentences. That is, unless you manually enumerated all valid combinations (a painstakingly intense process for a project of this scope). The following are the initial invalid constructions implemented:

- Noun -> Adjective
- Adjective -> Verb

- Adverb -> Noun
- Preposition -> Verb
- Article -> Article
- Preposition -> Preposition
- Conjunction -> Conjunction
- Pronoun -> Pronoun
- Noun -> Noun

Methodology

This project utilized certain principles of the Agile framework. Essentially this boils down to a heavy emphasis on code first, test later, iteratively redesign and refactor. Given the scope and timeline of this project, it made sense to get moving sooner rather than later.

Implementation Details

This project was written in the language Java, and utilizes frameworks Swing for the graphical user interface and Junit and Mockito for unit testing and mocking to support those tests. The IntelliJ IDEA was utilized to organize and develop the project. Lastly Git was used for version control and GitHub was used for storage. This project is comprised of three different layers, each implemented on their own but dependent upon the previous layer.

The LinguaLink project was implemented using a code-first, test-later approach, aligning with Agile principles to accommodate rapid iteration. Initially, the focus was on establishing the basic components and GUI layout, starting with a simple word list feature. As the project progressed, the architecture expanded, introducing more complex functionalities while ensuring the codebase remained maintainable and well-documented.

Early commits laid the groundwork with essential classes and a basic graphical user interface, which set the stage for the subsequent development of the Model-View-Controller (MVC) architecture. The Util class was created to centralize color logic, demonstrating an early commitment to clean code practices.

A pivotal moment in the project's evolution was the design shift from a directed graph to two sets representing WordBlocks and Connections. This decision was made to improve the application's compatibility with Swing's drawing functions, simplifying the implementation, and enhancing the user experience. The result was a more intuitive and visually coherent workspace for users to interact with word elements and their connections.

As the project matured, features such as connection validation and visual feedback were added, followed by drag-and-drop functionality and the ability to export the workspace as a PNG file. These features were carefully integrated to ensure they complemented the existing user interface without disrupting the user workflow.

The ability to save and load the application state was another significant enhancement, providing users with the flexibility to preserve and revisit their work. This functionality required meticulous attention to the serialization and deserialization of the application's state, ensuring data integrity across sessions.

Towards the later stages, the project underwent further refinements to improve usability, such as enhancing window sizing and adding the ability to select and delete connections directly from the workspace. This phase also saw the incorporation of unit tests for all components that could be reasonably tested, emphasizing the project's commitment to reliability and quality assurance.

Throughout the development process, adherence to Google's style guide ensured consistency and clarity in the codebase, contributing to the project's maintainability. The iterative redesign and refactoring allowed for the seamless integration of new features and adjustments based on testing results.

Testing and Evaluation

The testing of this application relied on two main strategies.

The first is unit testing of each component in isolation to both catch deviations of actual implementation from desired implementation, as well as to increase confidence in later commits. This means that every component .java file should have a pair test file in the test directory of the project. For these unit tests, the project makes use of the Junit library. In addition to this, it uses Mockito to mock objects and function calls not crucial to the evaluation of the target component in isolation, as per the goal unit testing. Some major bugs discovered through this strategy:

- Disconnect between Model and View
    - When trying to test the controller, I noticed a significant discrepancy between what I thought was happening in the model and what was happening in the view. This specifically had to do with moving words between the Work Space and the Word Bank. When an element was moved to either, it was correctly being removed in the GUI. This is what let this slip by me for so long – I was only testing by playing around with the running application. When I wrote unit tests (along side some print statements to view model state), I noticed that I had neglected to remove these elements from the model's representation of the application state. I was only removing their GUI representations in the GUI components. This didn't lead to any issues outright, but it meant that I was not getting the behavior I wanted out of the model in conjunction with the GUI. In order to fix this I had to update the logic that moved words between the Word Bank and the Work Space to also correctly update the model. This meant changing the way that the GUI components were created to rely more closely on the current model's internal state. I am sure this prevented future bugs and confusion down the road.

The second testing strategy was to interact with the application via its user interface. This allowed the testing workflows to mirror the workflows or series of actions that might be taken by a user in a production context. Some major bugs found through this strategy:

- Window Sizing Bug
    - I spent a majority of my time developing this application on an external monitor. On this monitor, the application would load fine with the desired size. When I ran it on my laptop alone, without connecting to the external monitor, I found that the application window loaded on my native display improperly. It was stretched way out to the right hand side so that a user on a standard laptop computer would not be able to see the word bank unless they resized the window. This pointed out the issue in my window sizing logic: absolute sizing. To counteract this issue, I redid the main window's logic to rely on dynamic values of screen size to set start up window dimensions and the maximum window dimensions. This is a bug I could not have found via unit testing, so it is a good thing I tried the application's user interface in a different environment.
- Dragged words not rendering
    - This issue manifested itself in an inability to drag elements from the Word Bank to the Work Space. When implementing the saving logic (that is, to save the application's state for loading later), I attempted to make use of the serialization interface built into Java. This required making all my classes implement the Serializable interface. This did certainly allow for very simple save and load logic as it was essentially built in. It also allowed me to restore the application state exactly without the need for any kind of parsing or looping through reconstructed values. What I did not realize was that this caused the data transfer class for dragging and dropping words to also serialize and reconstruct the actual Word element. This meant that the word that was added to the Work Space and the source word in the Word Bank were actually different objects with different identities. This destroyed my deletion logic, which required an identity search of the Word Bank to delete the proper word. While I could have redesigned the deletion to be based on attributes of the word, this would have resulted in a Word Bank incapable of hosting multiple of the same word. For articles like 'a' this was simply untenable. Many proper English sentences contain many of the same word and the user should be able to diagram simpler sentences. Without manual testing of the UI, I likely would not have noticed this issue.

Reflecting on my testing methods, testing with the user interface yielded more discoveries and, thus, fixes. That is not to say that unit testing did not play a crucial role in the development of this project. Unit testing provided a safety net on which to develop without fear of breaking something that had previously worked. The ability to define unit tests in step with components and verify the presence of desired behavior meant that further feature development could move at a rapid pace.

Results and Discussion

Successes

All in all, LinguaLink is a fine implementation of the Interactive Language Learning Tool project. I was able to successfully implement the user interface I laid out in the beginning.

As far as project requirements go, I almost fully implemented all of them. The Word Bank works spectacularly and is also visually appealing. The color coding for part of speech tagging is neat and clean to the eye.

Additionally, the Work Space turned out better than I expected. The repositioning of elements is responsive and smooth, and the colored word blocks look great when a diagram is created. The feature I am most happy with is the dragging and dropping. From the start I had a feeling this would be the most challenging to implement as it relied on decoupling elements from the user interface and allowing them to float freely around. It also demanded that where they were released be correctly tracked and that word blocks be created in the correct place to allow for a smooth user experience. I am quite happy with the connections and how they are drawn. The shift-click mechanism, once you are familiar with it, is quite intuitive. Additionally, connection highlighting, based on grammatical rules, creates for a solid feedback mechanism that fits well in the Work Space visually.

Finally, the save, load and export features are all, for the most part, fully implemented. Exporting to a PNG works exactly as intended. It is a familiar experience with the file explorer and you get the same colorful, interesting chart out that you created in the Work Space. The save and load work well, and, again, offer a very familiar experience. While the save file is human readable, this has the unintended consequence of defining a plain text language with which to define word diagrams. This is a cool feature I had not originally intended but arose from the decision to make and parse my own configuration file instead of relying on the built in Java serialization functionality.

Future Work

There are several changes I would make in the future or if I had had more time to complete this project. Many are features I had intended that I did not get the chance to implement, and many are features that I realized along the way would have been better than my original designs.

One example of a feature that I wanted to put in from the beginning was the ability to edit words on the fly. As it stands, once you add a word to the application, it is final. Your only option to edit it is to delete it and recreate it differently. This provides a serious bottleneck on the creation of larger projects in LinguaLink.

A feature I had to abandon along the way for lack of time had to do with saving and loading project workspaces. Loading connections posed a serious challenge given that there is a sense of identity of objects inherent to the way they were implemented. Saving connections is not as simple as word blocks as they rely not just on a hierarchy of basic parameters like position and word string, they rely on the actual identity in memory of the objects they reference. I certainly could have saved the word string and part of speech of word blocks they connected, however this would have meant that, upon reloading them, duplicate word blocks in the workspace would be indistinguishable.

Another feature that would have been appropriate is the ability to drag words from the Work Space to the Word Bank. This would promote a sense of symmetry of operations, since you are able to drag words from the Word Bank to the Work Space. It is also an feature that should be there, and I am sure real users would expect reciprocity here only to be let down by the one-directedness of the dragging and dropping.

An improvement that I believe all developers would like to make no matter the maturation and complexity of their project is to update the UI to be better styles. Nowhere is this more obvious than in the word creation widget. It relies on default styles and looks generally unfinished.

From the beginning of the project, I anticipated the ability for the user to change workspaces and end up with a diagram that enforces a whole different grammatical rule set. This proved to be unreasonable given the time constraints but seeing as LinguaLink is meant to be a language learning application, this feature is to be expected. This would mean defining many different Connection classes that all implement a single connection interface. The isValid() function of these classes would vary from language to language, enforcing language specific rules. Multiple WorkSpace classes could be created, each creating different types of connection based on their configuration language.

The final feature I would implement in the future is a tool bar in the Work Space. This would give users the ability to select a connection tool and draw syntactic relationship lines that way. The current implementation is fairly hidden from the user. While it is intuitive and quick once you get used to it, an outright tool would inspire users to explore instead of requiring them to dive into the documentation.

Conclusion

In completing the LinguaLink project, I've developed an interactive language learning tool through an Agile approach that, in most aspects, meets the requirements, and, in some respects, even exceeds them. This process included an essential shift from using a directed graph to a set-based model, enhancing the tool's performance and user experience. Throughout the project, I focused on creating a user-friendly interface and implementing practical features like drag-and-drop and state-saving functionality. I prioritized testing and adhered to coding standards to ensure quality and maintainability. Reflecting on the project, I've learned valuable lessons in adaptive planning, problem-solving, and software engineering (both in Java and in general). These experiences have equipped me with the skills necessary for future projects. As the project wraps up, it serves as a foundation for potential future work outlined above.

References and Appendices

- All user interface mocks in this project were designed and exported from Figma. Figma Inc. (2024). Figma. [Software]. Available at: https://www.figma.com/
- All diagrams were designed and exported from LucidChart. Lucid Software Inc. (2024). LucidChart. [Software]. Available at: https://www.lucidchart.com/
- Oracle. (n.d.). *Swing (Java Platform SE 7)*. Retrieved from https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html
- JetBrains. (n.d.). *IntelliJ IDEA Documentation*. Retrieved from https://www.jetbrains.com/idea/documentation/
- Software Freedom Conservancy. (n.d.). *Git*. Retrieved from https://git-scm.com/documentation

- GitHub, Inc. (n.d.). *GitHub Docs*. Retrieved from https://docs.github.com/en
- JUnit Team. (n.d.). *JUnit 5 User Guide*. Retrieved from https://junit.org/junit5/docs/current/user-guide/
- Apache Software Foundation. (n.d.). *Apache Maven Project*. Retrieved from https://maven.apache.org/guides/index.html
- Oracle. (n.d.). *Java SE Development Kit 8 Documentation*. Retrieved from https://docs.oracle.com/javase/8/docs/

User Manual

Installation

Prerequisites

Before installing the project, ensure you have the following software installed on your system:

- Java Development Kit (JDK) 11 or higher.
- Apache Maven 3.6.0 or higher.

Step 1: Obtain the Source Code

First, you need to obtain the source code of the project.

Step 2: Navigate to the Project Directory

After obtaining the source code, navigate to the project directory where the pom.xml file is located.

Step 3: Build the Project

Run the following command in your terminal or command prompt to build the project using Maven:

```
mvn clean install
```

This command cleans the target directory, compiles the source code, runs any tests, and installs the project to your local repository. If the project includes tests that you wish to skip, you can use the following command:

```
mvn clean install -DskipTests
```

Step 4: Verify the Installation

After the build process completes, you should see a BUILD SUCCESS message in the terminal or command prompt. This indicates that the project has been successfully built and installed.

The generated output, such as a JAR file, will typically be in the target directory of your project.

Step 5: Run the Application

If your project generates an executable JAR file, you can run it using the following command:

```
java -jar target/projectname-version.jar
```

Replace projectname-version.jar with the actual name of the generated JAR file.

Features

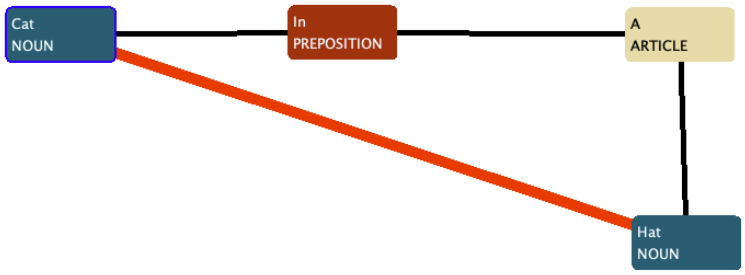
Word Bank

The Word Bank is the entry point for all words in the application. It serves as your toolbelt. You can add new words, of all different English parts of speech, to the Word Bank.

Work Space

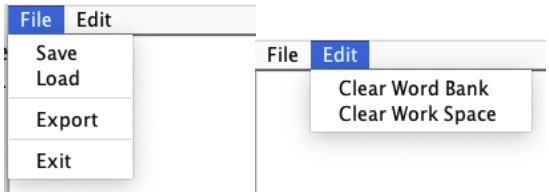
The Work Space is where you will be doing all of your sentence diagramming. Simply select a word in the Word Bank, drag it onto the Work Space, and you are ready to start diagramming. Rearrange at will!

Drawing Connections



Takes your sentence diagrams to the next level by drawing connections between various words. LinguaLink will even help you with your grammar by highlighting invalid connections in red.

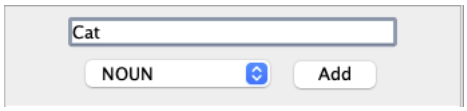
Menu Bar



The menu bar allows you to perform common tasks like saving your current LinguaLink project, loading a previous project you want to continue working on, exporting an image of your current diagram, and clearing the state of the Work Space and Word Bank.

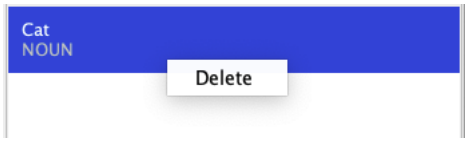
Common Tasks

Adding a word to the Word Bank



1. Type your desired word in the word creation tool.
2. Select the desired part of speech from the drop-down-menu.
3. Click add.
4. Use your new word!

Removing a word from the Word Bank



1. Right click on the word you would like to remove.
2. Click delete.

Clearing the entire Word Bank

1. Navigate to the ‘Edit’ menu at the top of the application.
2. Click ‘Clear Word Bank’

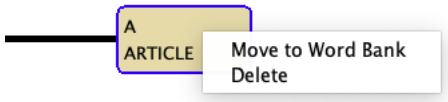
Moving a word from the Word Bank to the Work Space

To move a word from the Word Bank to the Work Space you can use one of two options:

- Double click the word you would like to move in the Word Bank.
- Drag and drop a word from the Word Bank to the Work Space.

Moving a word bank to the Word Bank from the Work Space

To move a word from the Work Space back to the Word Bank, simply double click on the word in the Work Space. Alternatively you can right click on a word in the Work Space and click 'Move to Word Bank'. Note: removing a word with active connections from the Work Space will remove these connections despite the other word still being present.



Deleting a word from the Work Space

To delete a word from the Work Space, simply right click on the word and click 'Delete' (shown above). Note: removing a word with active connections from the Work Space will remove these connections despite the other word still being present.

Drawing a connection between two words

To draw a connection from one word to another

1. Select the word you want to be the start of the connection.
2. Hold shift & click the second word you want to be the end of the connection.

Deleting a connection between two words

To delete a connection between two words you can either remove one of the words involved in the connection or

1. Select a connection. You know a connection is selected if it is highlighted blue (shown above).

2. Right click on the connection.
3. Select ‘Delete’ from the context menu.

Exporting a PNG

To export a PNG image of your current Work Space diagram

1. Click on the ‘File’ menu in the top of the application.
2. Click on ‘Export’.
3. Select a destination on your hard disk.
4. Click save.

Currently LinguaLink only supports exporting as a PNG.

Saving your current LinguaLink project

To save your project

1. Click on the ‘File’ menu in the top of the application.
2. Click on ‘Save’
3. Enter a name for your workspace file and select a location on your hard disk.
4. Click on save.

LinguaLink projects are always saved with the file extension *.llws*.

Loading an existing project

To load an existing project

1. Click on the ‘File’ menu in the top of the application.
2. Click on ‘Load’.
3. Select a valid *.llws* file from your disk.

TROUBLESHOOTING

If you encounter issues while using the Interactive Language Learning Tool, here are some common problems and their solutions to help you resolve them quickly.

1. Application doesn't start

Check Java Installation: Ensure that you have Java installed and configured properly on your system. You can verify this by running java -version in your command line or terminal.

Verify Installation Files: Make sure all required files for the application, including any external libraries, are present in the installation directory.

2. Words or connections are not being saved

Check Write Permissions: Ensure the application has the necessary permissions to write to your filesystem, especially in the directory where it's trying to save the data.

Disk Space: Verify that there is enough disk space on your drive to save the application state.

3. Words or connections not appearing in the workspace

Refresh the View: Sometimes, the view might not update in real-time due to system constraints. Try refreshing the view or restarting the application.

Data Integrity: Ensure that the data being loaded into the workspace is correctly formatted and not corrupted.

4. Cannot delete words from the WordBank

Selection Issue: Make sure you have correctly selected the word you want to delete. The application might not respond if there is no selection made.

Event Handling: If the deletion feature doesn't work as expected, there might be an issue with the event handling in the application. Consider contacting support for detailed analysis.

5. Exported images are blank or not as expected

Graphic Rendering: Ensure that the workspace is fully rendered before exporting. Quick operations might result in incomplete rendering.

File Compatibility: Check if the image file generated is compatible with the image viewers you are using and that it's not corrupted.

6. Application is slow or unresponsive

System Resources: Check if your computer meets the hardware requirements for the application and ensure that sufficient system resources are available.

Large Data Sets: If working with large data sets, the application may become slow. Consider breaking down the work into smaller manageable parts.

7. Errors during application load

Corrupted Configuration: If the application fails to load with configuration errors, try resetting the configuration to the default settings.

If these steps do not resolve your issues, please collect any error messages or logs generated by the application and contact the author for further assistance.

8. Application Window missing Word Bank on boot

Screen Size: Your screen may not be of a size tested on. Please ensure that all boundaries of the application are within the boundaries of your desktop. This varies based on operating system.

FAQs

Frequently Asked Questions (FAQ)

Q1: How do I add new words to the WordBank?

A1: To add new words to the WordBank, enter the word in the text field provided in the WordBank panel, select the appropriate part of speech from the dropdown menu, and click the "Add" button.

Q2: Can I move words from the WordBank to the WorkSpace?

A2: Yes, you can move words by double-clicking on them in the WordBank, or by dragging and dropping them into the WorkSpace.

Q3: How do I create connections between words in the WorkSpace?

A3: To create a connection, press and hold the Shift key, then click on the first word block and then the second word block to establish a connection between them.

Q4: How can I delete a word from the WordBank or WorkSpace?

A4: In the WordBank, right-click on the word you want to delete and select the delete option from the context menu. In the WorkSpace, right-click on the word block and select the delete option.

Q5: How do I save my current workspace?

A5: Go to the "File" menu and select "Save". Choose the desired location on your computer to save the workspace file.

Q6: How can I load a previously saved workspace?

A6: Go to the "File" menu and select "Load". Navigate to the location where your workspace file is saved and select it to load.

Q7: Can I export the WorkSpace as an image?

A7: Yes, you can export the WorkSpace as an image. Go to the "File" menu, select "Export", and choose where to save the image file on your computer.

Q8: What should I do if the application is not responding?

A8: If the application becomes unresponsive, try closing and reopening it. If the problem persists, check your computer’s resource usage to ensure it has enough memory and CPU available to run the application.

Q9: Are there any keyboard shortcuts to improve workflow?

A9: Currently, the main keyboard shortcuts involve using the Shift key for creating connections. More shortcuts may be available in future versions of the application.

Q10: Where can I find more help or report a bug?

A10: For more help or to report a bug, you can contact the author at his email: gnagle@ucdavis.edu.

Software Design Manual

Architecture Overview

The software architecture of the LinguaLink application is designed to facilitate interactive language learning through a modular and intuitive user interface. The system is primarily structured around three core components: the Model, the View, and the Controller, following the Model-View-Controller (MVC) design pattern. Here's a high-level overview of each component and their relationships:

Model: This component acts as the central point of data management and business logic. It handles the creation, storage, and manipulation of word objects, word blocks, and connections. The Model is responsible for maintaining the current state of the application, including the words in the `WordBank` and the `WordBlocks` in the `WorkSpace`.

View: The View component consists of various GUI elements, such as `WordBankPanel` and `WorkSpacePanel`, that present data to the user and capture user interactions. It is responsible for rendering the visual representation of the application's state managed by the Model.

Controller: Serving as the intermediary between the Model and View, the Controller processes user inputs, translates them into actions to be performed by the Model, and updates the View accordingly. It manages the flow of data to and from the Model and dictates how that data is displayed by the View.

The software's architecture also includes auxiliary components like `Util` for utility functions (e.g., exporting panels as images or managing application state persistence) and custom transferable objects like `WordTransferable` to support drag-and-drop functionality.

Overall, the architecture is designed to be extensible, allowing for easy integration of additional features such as advanced word manipulation, extended import/export capabilities, and more complex user interaction mechanisms in the future.

From an abstract perspective, the three main components of the GUI are listed below along with descriptions of user stories that they should be equipped to handle. Each of these corresponds to a concrete representation within the Model class.

Menu Bar: The menu bar resides at the top of the window. It is the users access point to high level controls. User stories include: “I would like to save my current project”, “I would like to export my current project as a PDF”, “I would like to export my current project as an image”, and “I would like to open an existing project saved to my hard drive”.

Word Bank: The word bank contains a list of words that are currently usage, but not yet placed on the canvas, within the current project. User stories include: “I would like to see what words are currently usable in my diagram”, “I would like to see what part of speech a word is”, and “I would like to add a new word for use in my diagram”.

Work Space: The work space contains current active diagrams. It also contains the language selection tool to switch to another language. Here users can place and order words, as well as draw links that form sentences. User stories include: “I would like to visually arrange certain words from my word bank”, “I would like to remove words from the work space”, “I would like to change my current project grammar to a French canvas”, and “I would like to draw a relationship between these two words”.

Design Patterns

The LinguaLink application leverages several design patterns to enhance code modularity, scalability, and maintainability. Here's an overview of the key design patterns used, their rationale, and implementation:

Singleton: Used in components like the Model, Controller, and various panels (e.g., WordBankPanel, WorkSpacePanel), the Singleton pattern ensures that only one instance of these components exists in the application at any time. This is particularly important for managing the centralized state and control logic, preventing inconsistent data states and facilitating global access to these components.

Transferable Object: The WordTransferable class implements the Transferable interface, a pattern used to encapsulate and transfer data within the application, particularly in drag-and-drop operations. This pattern allows for a flexible and decoupled way to transfer complex data types like Word objects between components.

Factory Method (implicit in some parts): While not explicitly defined as a traditional factory method pattern, the application employs factory-like methods in creating UI components or managing state transitions. These methods encapsulate the instantiation logic and promote a loosely coupled system design.

Observer: Utilized to synchronize the View with the Model, the Observer pattern enables the application to automatically update the UI components in response to changes in the application state. The Model acts as the subject, while various UI components play the role of observers, ensuring real-time updates and consistent views.

These design patterns were selected to optimize the application’s architecture for applicability to the desired product features and user stories, extensibility, testability, and ease of maintenance, supporting future enhancements and modifications efficiently.

Component Descriptions

Word

Package

LinguaLink.components.word

Description

The Word class represents a single word in the LinguaLink application. It encapsulates the string representation of the word and its grammatical classification, known as part of speech.

Attributes

WORD (String): The text representation of the word. It is final and set during object construction.

PART_OF_SPEECH (PartOfSpeech): An enumeration value representing the grammatical type of the word, such as noun, verb, etc. It is also final and set during object construction.

Constructor

Word(String word, PartOfSpeech pos): Initializes a new instance of the Word class with the specified string and part of speech.

Methods

getWord(): Returns the text of the word as a string.

getPartOfSpeech(): Returns the grammatical classification (part of speech) of the word.

Usage

The Word class is used to store information about individual words, including their textual content and grammatical classification. This class is fundamental in handling linguistic elements within the LinguaLink application, allowing for the association of specific characteristics and behaviors to each word based on its part of speech.

CONNECTION

Package

LinguaLink.components.connection

Description

The Connection class represents a linguistic relationship or linkage between two words in the LinguaLink application, specifically between two WordBlock instances. It encapsulates the source (FROM) and destination (TO) of the connection, typically representing grammatical or semantic relationships.

Attributes

FROM (WordBlock): Represents the originating word block of the connection.

TO (WordBlock): Represents the destination word block of the connection.

Constructor

Connection(WordBlock from, WordBlock to): Initializes a new instance of the Connection class, setting the source and destination word blocks.

Methods

getFrom(): Returns the source word block of the connection.

getTo(): Returns the destination word block of the connection.

isValid(): Determines if the connection between two word blocks is grammatically or semantically valid based on predefined rules.

contains(WordBlock wordBlock): Checks whether the specified WordBlock is either the source or destination in the connection.

Usage

The Connection class is used within the LinguaLink application to manage and represent the relationships between different WordBlock elements. It is essential for constructing and validating the network of word interactions, ensuring meaningful and grammatically correct connections within the workspace.

WordBlock

Package

LinguaLink.components.wordblock

Description

The WordBlock class in LinguaLink represents a visual and functional unit in the application's workspace. It encapsulates a Word object and maintains its position within the workspace, facilitating the graphical representation and interaction of words in the language learning tool.

Attributes

position (Point): The location of the WordBlock within the workspace.

WORD (Word): The Word object contained within this block, representing the linguistic element.

Constructor

WordBlock(Point p, Word w): Initializes a new WordBlock with a specified location and word. The position (p) is where the block appears in the workspace, and the word (w) is the linguistic content.

Methods

getPosition(): Returns the current position of the WordBlock in the workspace as a Point.

getWord(): Returns a copy of the Word object associated with this block, ensuring the original word data remains unaltered.

setPosition(int x, int y): Sets the position of the WordBlock within the workspace to the specified coordinates (x, y), allowing for movement and repositioning.

Usage

WordBlock objects are integral to the LinguaLink application, as they serve as the primary interface elements for displaying and interacting with words. They are used to visually organize words in the workspace, support drag-and-drop operations, and facilitate the creation and management of connections between words, thereby supporting the educational goals of the application.

WordBank

Package

LinguaLink.containers.wordbank

Description

The WordBank class in LinguaLink serves as a central repository for storing and managing Word objects. It is designed as a Singleton to ensure a single, globally accessible instance that maintains a consistent state of word elements across the application.

Attributes

wordBank (WordBank): A static reference to the Singleton instance of the WordBank.

words (ArrayList<Word>): A collection of Word objects currently stored in the WordBank.

Constructor

private WordBank(): Private constructor to prevent instantiation from outside the class, ensuring control over the lifecycle of the Singleton instance.

Methods

getInstance(): Static method that provides access to the Singleton instance of the WordBank, creating it if it doesn't already exist.

getWords(): Returns an unmodifiable list of Word objects currently in the WordBank, ensuring the integrity of the stored words.

addWord(Word word): Adds a new Word object to the WordBank, expanding the available vocabulary.

removeWord(Word word): Removes a specified Word object from the WordBank, if present.

clearWords(): Clears all Word objects from the WordBank, resetting the vocabulary repository.

Usage

The WordBank is a critical component of the LinguaLink application, acting as the main storage and management unit for words used in the learning tool. It provides controlled access to the words, supporting operations like addition, removal, and retrieval in a thread-safe manner. This ensures that all

parts of the application work with a consistent set of words, facilitating the learning and interaction processes in the language learning environment.

WorkSpace

Package

LinguaLink.containers.workspace

Description

The WorkSpace class in LinguaLink acts as the main canvas area where users can manipulate word blocks and connections. It follows the Singleton pattern to ensure a single, consistent state of the workspace throughout the application.

Attributes

workSpace (WorkSpace): A static reference to the Singleton instance of the WorkSpace.

wordBlocks (ArrayList<WordBlock>): A collection of WordBlock objects present in the workspace.

connections (ArrayList<Connection>): A collection of Connection objects representing the relationships between word blocks in the workspace.

Constructor

private WorkSpace(): Private constructor to enforce the Singleton pattern, initializing the lists for word blocks and connections.

Methods

getInstance(): Static method providing access to the Singleton instance of the WorkSpace.

clearWorkSpace(): Clears all word blocks and connections from the workspace.

getWordBlocks(): Returns an unmodifiable list of WordBlock objects in the workspace.

getConnections(): Returns an unmodifiable list of Connection objects in the workspace.

addWord(Word word, int x, int y): Adds a new WordBlock to the workspace at specified coordinates and returns its reference.

removeWordBlock(WordBlock wordBlock): Removes a specific WordBlock from the workspace and any connections associated with it.

removeConnection(Connection toDelete): Removes a specified Connection from the workspace.

addConnection(Connection toAdd): Adds a new Connection between two word blocks in the workspace.

Usage

WorkSpace serves as a dynamic area where users can interact with word blocks and visually represent their relationships. This interactive environment is pivotal for the language learning experience in LinguaLink, allowing for hands-on manipulation and visual representation of language constructs.

ComplexCellRenderer

Package

LinguaLink.guiComponents.wordBankPanel

Description

ComplexCellRenderer is a custom renderer for JList elements in the LinguaLink application, specifically designed for displaying Word objects in the WordBank. It enhances the visual representation of each word in the list, showing the word's text and part of speech (POS) with distinct formatting and background colors based on the word's POS.

Implements

ListCellRenderer<Word>: Interface defining the method required for rendering objects within a list.

Constructor

No explicit constructor. The default constructor is used.

Methods

getListCellRendererComponent(JList<? extends Word> list, Word word, int index, boolean isSelected, boolean cellHasFocus): Configures and returns a Component that displays the Word object within the list cell. The method customizes the appearance based on whether the word is selected or focused, including setting background colors and text properties.

Usage

ComplexCellRenderer is instantiated and set as the cell renderer for a JList<Word> component, typically in the WordBankPanel.

It adjusts the look of each list item based on the Word object's properties, enhancing the user interface by making the word bank visually intuitive and aligned with the part of speech color coding system.

Key Components

JPanel: Used as the main container to hold the rendered labels for each word.

JLabel: Two labels are used, one for displaying the word and the other for its part of speech.

JSplitPane: Arranges the word and POS labels vertically within the rendered cell, enhancing layout consistency.

Visual Customization

Background and text colors change dynamically based on the part of speech of the Word object, utilizing utility methods like Util.getBackgroundColor and Util.getPrimaryTextColor for color determination.

Selected and focused states affect the rendering, ensuring visual feedback for user interactions within the WordBank list.

Importance

ComplexCellRenderer significantly contributes to the usability and aesthetic appeal of the LinguaLink application by providing a clear, intuitive representation of linguistic elements, aiding users in quickly identifying and interacting with words of different parts of speech.

WordBankPanel

Package

LinguaLink.guiComponents.wordBankPanel

Description

WordBankPanel is a graphical user interface component of the LinguaLink application, designed to display and manage the Word objects stored in the application's word bank. It provides functionalities to add, view, and interact with the words, including drag-and-drop support and right-click deletion.

Extends

JPanel: Inherits from the Swing JPanel class, making it a container that can hold other components.

Constructor

WordBankPanel(): Initializes the WordBankPanel, setting up the user interface components and their behaviors.

Key Methods

initializeUI(): Sets up the UI components, including the input text field, part of speech dropdown, add button, and the list that displays the words.

createPopupMenu(): Creates a context menu that appears when right-clicking on an item in the word list, allowing users to delete the selected word.

refreshWordBank(List<Word> words): Updates the display list to reflect the current state of the word bank, typically after changes have been made.

clear(): Clears all elements from the word list display, used when the word bank is cleared.

Event Handling

Mouse interactions are handled to support actions like double-clicking a word to move it to the workspace and right-clicking to show a delete option.

The add button's action listener adds the entered word to the word bank using the controller.

Features

Drag-and-Drop Support: Enables dragging of words from the word bank to the workspace.

Context Menu: Offers additional actions like deleting a word from the word bank through a right-click menu.

Dynamic UI Updates: Reflects changes in the word bank model in real-time, ensuring the UI stays synchronized with the data.

Usage

WordBankPanel is used within the LinguaLink application's main window to interact with the word bank, providing a crucial part of the user interface for managing the language elements the user is working with.

WordTransferable

Package

LinguaLink.guiComponents.wordTransferable

Description

WordTransferable provides a means to encapsulate a Word object for transfer operations, particularly in drag-and-drop actions within the LinguaLink application. It implements the Transferable interface, enabling Word objects to be used in data transfer operations across the Swing UI components.

Implements

Transferable: Allows objects of WordTransferable to be transferred between components in the application.

Fields

WORD_FLAVOR: A static DataFlavor that uniquely represents the Word class type in transfer operations.

Constructor

WordTransferable(Word word): Initializes a new instance of WordTransferable with the specified Word object, making it ready for transfer.

Key Methods

getTransferDataFlavors(): Returns an array of DataFlavor objects indicating the supported data flavors (types) for transfer; specifically, it supports the WORD_FLAVOR.

isDataFlavorSupported(DataFlavor flavor): Checks if the specified data flavor is supported by this transferable object.

getTransferData(DataFlavor flavor): Returns the Word object if the requested data flavor matches WORD_FLAVOR; otherwise, it throws an UnsupportedFlavorException.

Usage

Used primarily in the drag-and-drop functionality of the LinguaLink application, allowing Word objects to be dragged from the word bank and dropped into the workspace.

Acts as a wrapper that encapsulates the Word object, providing a standard interface for transfer operations within the application's GUI components.

Exception Handling

Throws UnsupportedFlavorException if an attempt is made to get the Word data in a flavor other than WORD_FLAVOR.

Summary

WordTransferable class is crucial for enabling the transfer of Word objects within the LinguaLink application, specifically facilitating drag-and-drop interactions in the GUI, ensuring that the Word objects can be easily moved and utilized within different parts of the application interface.

WorkSpacePanel

Package

LinguaLink.guiComponents.workSpacePanel

Description

WorkSpacePanel is a custom Swing panel that serves as the main area in the LinguaLink application where users can interact with WordBlock and Connection objects. It provides a graphical representation of these objects and allows users to perform various actions like dragging and dropping words, creating connections between them, and modifying the workspace content

Extends

JPanel: Inherits from Swing's JPanel class to provide a container that can hold other GUI components.

Key Attributes

wordBlockShapes: A map storing the WordBlock instances along with their graphical shapes.

draggedWordBlock: The WordBlock currently being dragged by the user.

lastDragPoint: The last point recorded during a drag operation.

controller: An instance of Controller to handle business logic interactions.

model: An instance of Model to interact with the application's data.

Constructor

WorkSpacePanel(): Initializes the workspace panel, sets up mouse handling, drop target, and the graphical user interface.

Key Methods

setupMouseHandling(): Configures mouse event listeners to handle actions like dragging, clicking, and selecting within the workspace.

createWordBlockAtLocation(Word word, Point location): Creates and places a new WordBlock in the workspace based on the specified word and location.

addWordBlock(WordBlock wordBlock), addWordBlock(WordBlock wordBlock, int x, int y): Adds a WordBlock to the workspace, either at a default or specified position.

safeDeleteWordBlock(WordBlock toDelete): Safely deletes a WordBlock from the workspace, ensuring any associated connections are also removed.

createPopupMenu(), createConnectionPopupMenu(): Creates context menus for WordBlock and Connection objects, providing options like moving to word bank or deleting.

paintComponent(Graphics g): Custom paint method to draw WordBlock shapes and connections in the workspace.

Event Handling

Implements mouse event listeners to manage drag-and-drop functionality, selection, and context menu interactions, enhancing user experience and interactivity within the workspace.

Usage

The WorkSpacePanel is a crucial part of the LinguaLink UI, allowing users to visually manage words and their connections, facilitating an interactive and intuitive way to explore language concepts and relationships.

Summary

WorkSpacePanel in the LinguaLink application is a dynamic and interactive panel that visually represents words and their linguistic connections, equipped with robust event handling and user interaction mechanisms to provide a comprehensive workspace for language exploration and learning.

LOGGER

Package

LinguaLink.logger

Description

Logger is a utility class in the LinguaLink application that provides logging functionality. It is used to output informational and error messages with timestamps, helping in tracking the application's operational flow and diagnosing issues.

Key Features

Timestamps: Each log entry is prefixed with a timestamp, providing a temporal context to the log messages.

Log Levels: Supports different levels of logging, such as INFO and ERROR, to categorize the log messages based on their severity or purpose.

Methods

info(String message): Logs a message at the INFO level, indicating general informational events.

error(String message): Logs a message at the ERROR level, indicating significant problems that need attention.

log(String level, String message): The core logging method that formats and outputs the log messages to the console. This method is private and is used internally by the info and error methods.

Implementation Details

Utilizes SimpleDateFormat to format the timestamps in a readable yyyy-MM-dd HH:mm:ss format.

Standard output (System.out.println) is used to display the log messages, making it simple to redirect logs to various destinations (like console, files, etc.) if needed.

Usage

The Logger class is used throughout the LinguaLink application to log various types of information. For example, it can log the initialization of components, the execution flow of operations, and any errors or exceptions that occur.

Summary

The Logger class in LinguaLink serves as a central logging mechanism, offering a straightforward way to output time-stamped informational and error messages, thereby aiding in the monitoring and troubleshooting of the application's behavior.

App

Package

LinguaLink

Description

App is the main entry point of the LinguaLink application, serving as the root window hosting the user interface components. It integrates various elements like the word bank and workspace panels, providing an interactive environment for language learning and manipulation of linguistic elements.

Key Features

Model-View-Controller (MVC) Integration: Interacts with Model and Controller to maintain the application's state and respond to user actions.

Dynamic UI Updates: Implements ModelObserver to react to changes in the model and update the UI accordingly.

Customizable Layout: Organizes the main window layout, including menu bars, word bank, and workspace areas.

Methods

initUI(): Initializes the main window, setting its size, layout, and other properties.

setupMenuBar(): Creates the menu bar with options like file handling, saving, loading, and exiting the application.

constructWordBank(): Initializes the word bank panel where words are listed and managed.

constructWorkSpace(): Initializes the workspace panel where words can be arranged and connections made.

setupMainAndSideLayout(): Sets up the main layout with a split pane dividing the word bank and workspace.

update(): Updates the UI elements in response to changes in the application's data model.

refreshWordBank(): Refreshes the word bank display with the latest words from the model.

refreshWorkSpace(): Refreshes the workspace display with the latest word blocks and connections.

main(String[] args): The entry point of the application which initializes and displays the main window.

Implementation Details

The UI is built using Java Swing, providing a cross-platform graphical user interface.

Listeners and events are used extensively to provide interactive and responsive user experiences.

The application is designed to be modular, allowing for easy extension and modification of its components.

Usage

The App class is executed to start the LinguaLink application, providing users with a graphical interface to interact with words and their relationships in a linguistic context. It orchestrates the overall workflow, managing user inputs, data representation, and visual updates.

Summary

The App class in LinguaLink encapsulates the main application window, handling initialization and coordination of the user interface components, interactions, and data flow, embodying the core functionality of the interactive language learning tool.

CONTROLLER

Package

LinguaLink

Description

The Controller class in LinguaLink serves as a central control mechanism, facilitating interaction between the model and view components. It adheres to the Singleton design pattern to ensure there's a single point of control and coordination within the application.

Key Features

Singleton Pattern: Ensures a single instance of the controller is used throughout the application, accessed via getInstance().

Model Interaction: Facilitates operations on the model, such as adding, moving, and deleting words or connections.

UI Coordination: Acts as an intermediary to update the workspace and word bank based on user actions.

Methods

addWordBankElement(Word in): Adds a word to the model's word bank.

addConnection(Connection c): Adds a connection between word blocks in the workspace.

clearWordBank(): Clears all words from the word bank.

clearWorkSpace(): Clears all word blocks and connections from the workspace.

moveWordToWorkSpace(Word toMove, int x, int y): Moves a word to the workspace at a specified location.

moveWordToWorkSpace(Word toMove): Overloaded method to move a word to the workspace at a default location.

moveWordToWordBank(WordBlock toMove): Moves a word block from the workspace back to the word bank.

setWordBlockPosition(WordBlock toSet, int dx, int dy): Adjusts the position of a word block within the workspace.

setWordBlockPositionAbs(WordBlock toSet, int x, int y): Sets the absolute position of a word block in the workspace.

deleteWordBlock(WordBlock toDelete): Deletes a specific word block from the workspace.

deleteWord(Word toDelete): Deletes a word from the word bank.

deleteConnection(Connection toDelete): Removes a connection between two word blocks.

Implementation Details

Employs logging through Logger to facilitate debugging and operation tracing.

Exception handling in methods like moveWordToWorkSpace and moveWordToWordBank to manage errors related to nonexistent entities.

Coordinates directly with Model to reflect the state changes resulting from user actions in the UI components.

Usage

Used internally within the LinguaLink application to manage the logic and data flow between the model and the view components. It processes user interactions, manipulates the model, and updates the view accordingly.

Summary

The Controller class in LinguaLink is the backbone of the application's logic handling, orchestrating interactions between the user interface and the data model, ensuring the application's state is consistently managed and updated in response to user actions.

MODEL

Package

LinguaLink

Description

The Model class in LinguaLink encapsulates the application's core data structures and logic, managing the word bank and workspace while following the Singleton pattern to ensure centralized data management.

Key Features

Singleton Pattern: Ensures that only one instance of the Model class is active, providing a global point of access.

Observer Pattern: Implements observer functionality to notify registered observers about state changes, facilitating MVC separation.

Word and Connection Management: Manages words and connections, including addition, deletion, and movement between the word bank and workspace.

Methods

getInstance(): Provides access to the singleton instance of Model.

resetInstance(): Resets the model state, primarily for testing purposes.

addObserver(ModelObserver observer): Registers an observer to receive updates.

removeObserver(ModelObserver observer): Unregisters an observer.

notifyObservers(): Notifies all registered observers of a state change.

addWordToBank(Word word): Adds a word to the word bank.

getWordBankWords(): Retrieves all words from the word bank.

getWorkSpaceWordBlocks(): Retrieves all word blocks from the workspace.

clearWordBank(): Clears all words from the word bank.

clearWorkSpace(): Clears all word blocks and connections from the workspace.

moveWordToWorkSpace(Word toMove, int x, int y): Moves a word from the word bank to the workspace.

moveWordToWordBank(WordBlock toMove): Moves a word block from the workspace back to the word bank.

deleteWordBlock(WordBlock toDelete): Deletes a word block from the workspace.

addConnection(Connection toAdd): Adds a connection between two word blocks in the workspace.

deleteConnection(Connection toDelete): Removes a connection from the workspace.

deleteWord(Word toDelete): Removes a word from the word bank.

getActiveConnections(): Retrieves all active connections in the workspace.

loadFromFile(List<Word>, List<WordBlock>): Restores the state of the word bank and workspace from file data.

Implementation Details

Manages two core components: WordBank for storing available words and WorkSpace for active word blocks and their connections.

Usage

The Model is a central component in the LinguaLink application, directly manipulated by the Controller to reflect the user's interactions in the application's state. It maintains the integrity and consistency of the application's data.

Util

Package

LinguaLink

Description

The Util class in LinguaLink is a utility class that provides various static methods to support the functionality of the application, such as color management based on parts of speech, exporting panels to images, and saving or loading the application state.

Key Features

Color Management: Methods to get background, primary text, and secondary text colors based on the part of speech.

Image Export: Allows exporting a JPanel as a PNG image file.

Application State Management: Supports saving and loading the application state to and from files.

Methods

getBackgroundColor(PartOfSpeech pos): Determines the background color associated with a given part of speech.

getPrimaryTextColor(PartOfSpeech pos): Determines the primary text color for a given part of speech, considering readability against the background.

getSecondaryTextColor(PartOfSpeech pos): Determines the secondary text color for a given part of speech.

exportPanelAsImage(JPanel panel): Exports the contents of a specified JPanel as a PNG image.

saveApplicationState(Model model): Saves the current state of the application, including the word bank and workspace, to a user-defined file.

loadApplicationState(Model model): Loads application state from a file, updating the model with the loaded data.

Usage

Util class methods are used throughout the LinguaLink application to ensure consistent styling, facilitate user interactions such as saving workspace snapshots, and manage application state persistence.

Implementation Details

Utilizes Java Swing components for file selection dialogs and message notifications.

Employs standard Java I/O classes for reading from and writing to files.

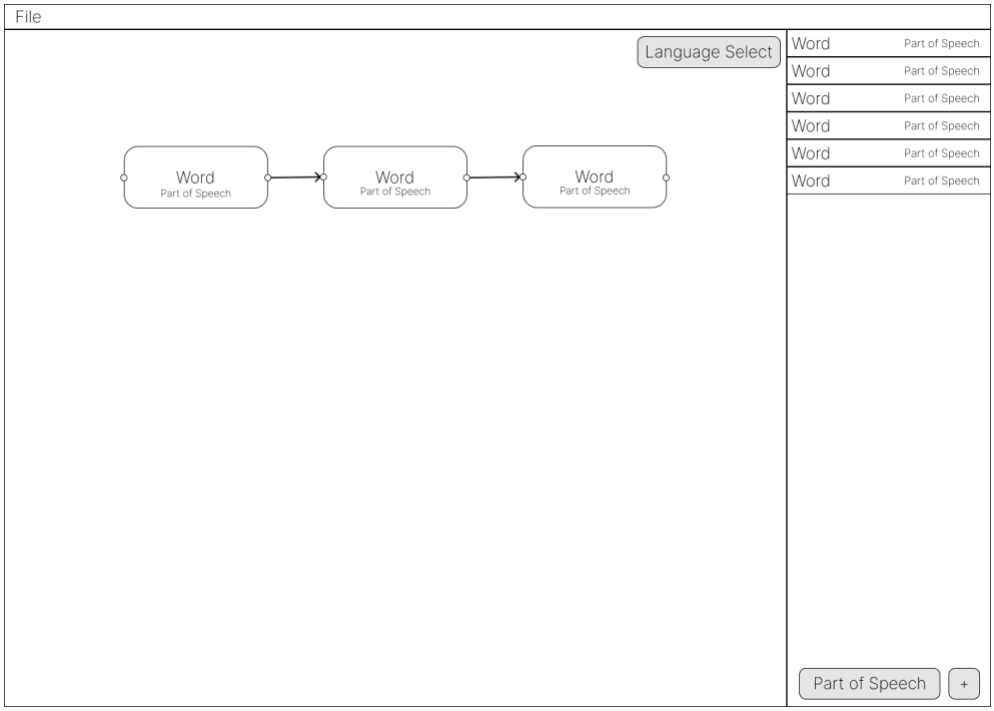
Integrates with Model, Word, and WordBlock classes for state management functionality.

This class acts as a helper, providing necessary functionalities that are not part of the core business logic but are essential for the user interface and state management of the LinguaLink application.

DIAGRAMS

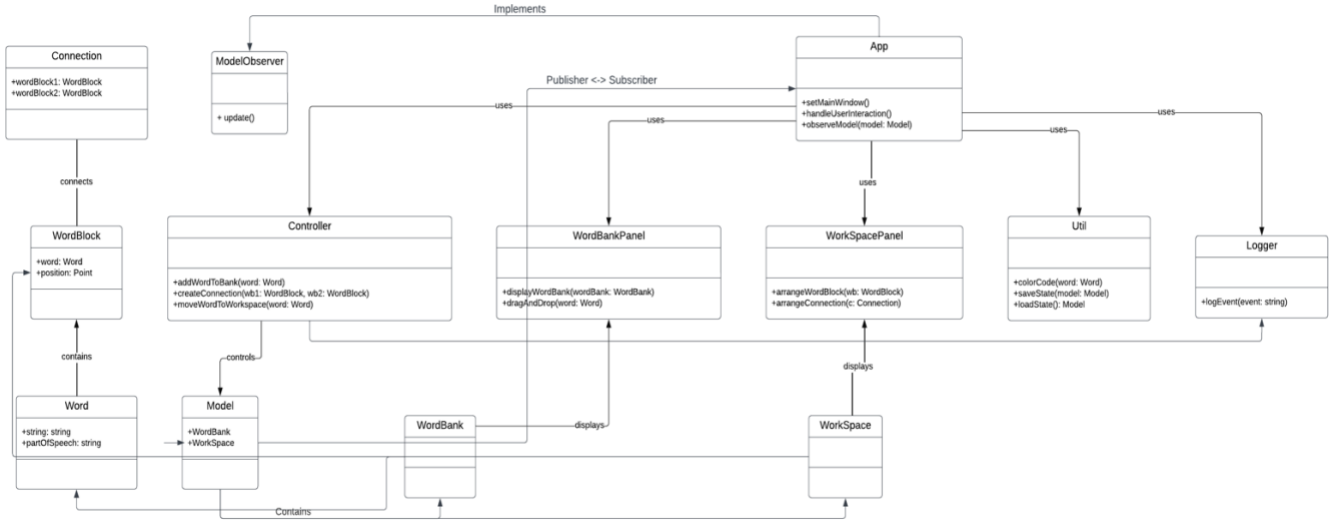
USER INTERFACE

The user interface design began with a simple wireframe of a desktop sized window. This window includes all thus far planned components.



In the top right is the menu bar. The file dropdown menu will include the option to save the current file, open a new file, and export the current file as an image or PDF. The pane on the right-hand side is the word bank. This contains a list of all words present in the current workspace and a drop down to select a part of speech next to a button to add a new instance of that part of speech to the workspace. Users will be able to drag a word out of the word bank and place it on the canvas to the left. The canvas (the left pane) is where currently active words can be placed, arranged, and linked to form sentences. The arrows between words represent a transition between the previous word and the next. The application will notify users when the link is invalid (i.e. when a verb is connected, in sequence, to another verb).

Class Diagram



The UML diagram for the LinguaLink project represents a collection of interconnected classes that form the architecture of a language learning tool. At the heart of the system is the Model class, acting as a central hub, which maintains the state of the application by holding references to WordBank and WorkSpace, where WordBank is a repository for Word objects, and WorkSpace manages WordBlock and Connection objects to represent words and their relationships spatially. Controller serves as a mediator between the Model and the user interface, implementing the logic to manipulate model states based on user actions, like adding words to the bank, creating connections in the workspace, and moving words between these areas.

The Word class encapsulates the linguistic element with attributes like the string itself and its part of speech. WordBlock combines a Word with its position, used in the workspace to visually organize words. Connection represents links between WordBlock instances, indicating linguistic relationships. The GUI is divided into main components: WordBankPanel and WorkSpacePanel, where WordBankPanel displays the Word objects stored in the WordBank, allowing for drag-and-drop interactions, and WorkSpacePanel provides a graphical interface to arrange WordBlock and Connection objects, simulating a linguistic worspace.

Util offers static utility methods for common functionalities like color coding based on parts of speech, and saving or loading the application state, enhancing usability and persistence. Logger is a utility class for logging system events and activities, crucial for monitoring and debugging. Finally, the App class stitches these components together, setting up the main application window, handling user interactions, and observing changes in the Model to refresh the UI accordingly, thus completing the ecosystem of the LinguaLink language learning tool.

Standard and Conventions

This project utilized the standards and conventions set for the in the Google style guide for Java. You can access this style guide on GitHub at https://google.github.io/styleguide/javaguide.html.

Appendices

ChatGPT Logs for Gradyn Nagle

See file: /project/GPTLogs.html for complete ChatGPT Logs.