

Spring 2022

Design And Implementation of An Automatic Word Generator For Word Matching Interactives

Evan Miles Gertis

Follow this and additional works at: <https://digitalcommons.georgiasouthern.edu/etd>



Part of the [Software Engineering Commons](#)

Recommended Citation

Gertis, Evan Miles, "Design And Implementation of An Automatic Word Generator For Word Matching Interactives" (2022). *Electronic Theses and Dissertations*. 2406.
<https://digitalcommons.georgiasouthern.edu/etd/2406>

This thesis (open access) is brought to you for free and open access by the Graduate Studies, Jack N. Averitt College of at Digital Commons@Georgia Southern. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of Digital Commons@Georgia Southern. For more information, please contact digitalcommons@georgiasouthern.edu.

DESIGN AND IMPLEMENTATION OF AN AUTOMATIC GENERATOR FOR WORD
MATCHING INTERACTIVES

by

EVAN M. GERTIS

(Under the Direction of Daniel Liang)

ABSTRACT

An Automatic Word Match Generator is a software tool that can be used to generate word-matching interactives automatically. The purpose of a word-matching interactive is to provide students with the mechanism to learn new vocabulary and improve their reading comprehension skills. This thesis will present the design and implementation of an Automatic Word Match Generator, as well as the research and algorithms used in the program.

INDEX WORDS: Automatic programming, Computer science education, Online learning, programming synthesis, Word matching

DESIGN AND IMPLEMENTATION OF AN AUTOMATIC GENERATOR FOR WORD
MATCHING INTERACTIVES

by

EVAN M. GERTIS

B.S., University of North Carolina at Chapel Hill, 2017

A Thesis Submitted to the Graduate Faculty of Georgia Southern University in Partial
Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

©2022

EVAN M. GERTIS

All Rights Reserved

DESIGN AND IMPLEMENTATION OF AN AUTOMATIC GENERATOR FOR WORD
MATCHING INTERACTIVES

by

EVAN M. GERTIS

Major Professor: Daniel Liang
Committee: Ryan Florin
Andrew Allen

Electronic Version Approved:
May 2022

DEDICATION

I dedicate my work to my professors at Georgia Southern University.

ACKNOWLEDGMENTS

I would like to sincerely thank Dr. Daniel Liang for his mentorship and support throughout this process, Dr. Florin, for his willingness to provide feedback on my work and Dr. Allen, for motivating me with positive reinforcement.

TABLE OF CONTENTS

	Page
ACKNOWLEDGMENTS	3
LIST OF TABLES	6
LIST OF FIGURES	7
CHAPTER	
1 INTRODUCTION	10
1.1 Problem And Motivation	10
1.2 Definition Of Terms	15
1.3 Review Of Literature	17
1.4 DESCRIPTION OF REMAINING CHAPTERS	30
2 METHODOLOGY	31
2.1 Requirements Specification	31
2.2 Design	32
2.3 Implementation	37
2.4 Drag, Drop, And Match Algorithm	44
2.4.1 Algorithm: Drag, Drop, and Match	45
2.5 Testing	46
3 RESEARCH COMPONENTS	51
3.1 Survey Of Automatic Programming	51
3.2 A Generic Model For Generating A Web Page	52
3.3 Applying The Generator Model To Other Problems	53

	5
4 CONCLUSION	57
5 FUTURE WORK	59
REFERENCES	64
APPENDICES	68
Appendix A: User’s Manual	68
Appendix B: Maintenance Manual	72
Appendix C: Design Documents	74
Appendix D: Source Code	75
Appendix E: Test Suite	113

LIST OF TABLES

	Page
1.1 Two-Sample t-test Assuming Equal Variances, Spring 2015 Exam Average (N=79) And Fall 2015 Exam Average (n=79) And Fall 2015 Exam Average (n=41) Cooney (2015)	25

LIST OF FIGURES

	Page
1.1 School-level Online Platforms (Kansal et al., 2021)	12
1.2 Automatic Word Match Generator UI Before Dragging Boxes	13
1.3 Automatic Word Match Generator UI After Dragging Boxes	13
1.4 Congratulations Dialog Box	13
1.5 Original Word Matching Interactive Built With Static HTML AND Javascript	14
1.6 The Word Match Generator Generates a Word Matching Interactive . . .	15
1.7 Correlation Between Average Revel Grade And Final Course Grade, Fall 2015(n=41) Cooney (2015)	22
1.8 Correlation Between Average Revel Grade And Average Quiz Grade, Fall 2015 (n=41) Cooney (2015)	23
1.9 Relationship Between Average Revel Score And Average Quiz Letter Grades, Fall 2015 (n=41) Cooney (2015)	24
1.10 Relationship Between Average Revel Grade And Average Exam, Lab, And Final Course Grades, Spring 2016 (n=100) Cooney (2016)	26
1.11 Comparison of Average Exam Scores And Average Final Course Grade, Before Implementation of Revel, Spring 2015 (n=79) And After Implementation of Revel, Fall 2015 (n=41) Cooney (2015)	27
1.12 Learning Curve: Fast Forward Middle and High School (Agocs et. al, 2006)	29
2.1 Automatic Word Match Generator Custom multi-tiered Application Diagram Before And After.	34
2.2 Sending a Word-Matching Interactive To The Server	36

2.3	XmlHttpRequest Request Body Shown In Chrome Developer Tools . . .	36
2.4	Id Returned After Saving a Word-Matching Interactive	36
2.5	Flow Diagram For Generating html	38
2.6	XmlHttpRequest From The Word Match Generator Client To Server . . .	38
2.7	Technical Diagram For The Automatic Word Match Generator	39
2.8	Logging The Output From The Javascript Console	40
2.9	Initial Attempt At Developing Rendering	42
2.10	Word Match Controller	43
2.11	Word Match	43
2.12	View	43
2.13	Word Match Service	44
2.14	Initial Screen For The Automatic Word Match Generator	45
2.15	Key Terms And Description Inputs	47
2.16	Generated Html Code	48
2.17	Clicking Post Button	49
2.18	Generated Word-Matching Exercise	50
3.1	A Generic Model For Generating a Web Page.	52
3.2	Explanation For Line 4 of Compute Area With Console Input.	54
3.3	Explanation For Line 6 of Compute Area With Console Input.	55
5.1	Adding a Question	60
5.2	Before Flipping a Card	61

5.3	After Flipping a Card	62
A.1	Word Match Generator Without Input Data	69
A.2	Word Match Generator With Input Data	70
A.3	Word Match Generator After Generating HTML	70
A.4	Word Match Generator Before Dragging Boxes	71

CHAPTER 1

INTRODUCTION

This chapter will cover the problem and motivation, definition of key terms, and supporting literature. It will also cover research history, significance of the field, and the specific research problem that we aim to address.

1.1 PROBLEM AND MOTIVATION

Students tend to face difficulties when they are required to learn new terminology. For example, understanding how to differentiate between the terms “hardware” and “software” can be challenging for beginner computer science students. In a typical introductory programming course, students first learn the key terms and definitions of the subject material. In general, hardware is the physical aspect of the computer that can be seen, and software is the invisible instructions that control the hardware and make it work. As stated by Liang (2020), the hardware of a computer consists of a control processing unit (CPU), cache, memory, hard disk, floppy disk, monitor, printer, and communication devices.

From a technical perspective, one way to address the learning difficulties that students face is to provide them with tools that make their learning experience more enjoyable. Students face difficulties concentrating on the task at hand. It is also challenging to follow a consistent pattern while studying online. Currently, e-learning is enhancing the knowledge of students, academic staff, and other professionals via the internet (Adams et. al, 2018). Many higher education universities are providing online courses for their students (Shahzad et. al, 2021). Amongst the multitude of challenges that schools have faced in the transition to online learning accessibility has been challenging for students with disabilities and/or their parents or caregivers with disabilities (Badge et. al, 2008). The *word-matching interactives* created by the Automatic Word Match Generator addresses these challenges by providing students with access to online study material.

Vocabulary development is an essential part of the learning process. As stated by Young (2005), teaching content-area science vocabulary through a variety of inquiry methods and engaged word-meaning concept strategies allows learners to make their own intellectual connections while gaining an understanding and confidence in the language of the science content. Arifah and Kusumarasyati (2013) showed that in order for students to be successful, they should be able to understand the differences between specific terminology. Our tool, Automatic Word Match Generator, can be used to help students develop their academic vernacular through *word-matching interactives*. Given a set of key terms and corresponding definitions, students can use *word-matching interactives* to learn the meanings of specific key terms used in computer science.

A significant paradigm shift has occurred in the education system post-COVID-19 outbreak (Kansal et. al., 2021). Many in-person classes now feature a remote component. The graph shown in Figure 1.1 shows that Udemy usage increased during the pandemic around March 2020. It is possible that this growth can be attributed to the forced isolation brought on by COVID-19. The findings from Shahzad et al. (2021) showed that the consequences of the pandemic were unstoppable and uncontrollable for higher education industries, much like the rest of the world. Azzi-Huck and Shmis (2020) indicated that most higher education systems are now operating through e-learning platforms. At least 120 countries stopped face-to-face learning and approximately one billion students' education was affected worldwide with COVID-19. The stabilization of the curve in Figure 1.1 could indicate that the world has adopted online learning platforms like Udemy, Whitehat Junior, Vendatu, Byjus, Khan Academy, Sqayam, Edx, and Unaacademy.

The effects of the COVID-19 pandemic created a need for online resources that can help students learn remotely. The pandemic forced the universities to close face-to-face education and send students home. This resulted in universities introducing courses through online portals (Azzi-Huck and Shmis 2020). In the absence of effective mitigation pro-

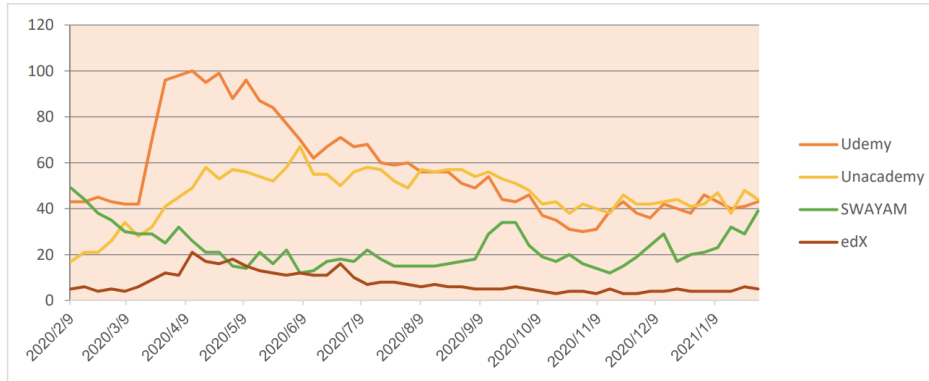


Figure 1.1: School-level Online Platforms (Kansal et al., 2021)

grams, for example, distance-learning programs will have many detrimental impacts on children and youth (Azzi-Huck and Shmis 2020). The *word-matching interactives* created by the Automatic Word Match Generator provides students with an opportunity to improve their reading comprehension from any location where there is a stable internet. Previous researchers developed successful methods for utilizing word-matching games in the classroom. However, few have developed a system that enables students to learn vocabulary from anywhere in the world.

Figure 1.2 shows an example of a *word-matching interactive*. Another live example can be viewed from https://liveexample.pearsoncmg.com/wordmatch/Section1_2.html.

Figure 1.3 shows the result after the user drags the key terms to match their descriptions. A congratulations dialog (see Figure 1.4) is displayed when all of the key terms are matched to their descriptions.

We have developed more than 60 *word-matching interactives*. The following ebooks Liang (2018), Liang (2020), Liang (2021) have embedded *interactives*. These *interactives* have received good reviews as shown in Cooney (2015) and Cooney (2016). They help students learn and grasp key terms. Previously, each of the *word-matching interactives* were programmed manually. The effort required to develop *word-matching interactives*

HTML lists

Unordered lists Ordered lists Nesting lists Description lists

An unordered list starts with the ul tag. Each list item starts with the li tag.

starts with the ol tag. Each list item starts with the li tag.

lists within lists.

A description list is a list of terms, with a description of each term.


Show Answer 

Figure 1.2: Automatic Word Match Generator UI Before Dragging Boxes

HTML lists

Unordered lists An unordered list starts with the ul tag. Each list item starts with the li tag.

Ordered lists starts with the ol tag. Each list item starts with the li tag.

Nesting lists lists within lists.

Description lists A description list is a list of terms, with a description of each term.

Congratulations
You did it!
OK

Show Answer Reset 

Figure 1.3: Automatic Word Match Generator UI After Dragging Boxes

Congratulations

You did it!

OK

Figure 1.4: Congratulations Dialog Box

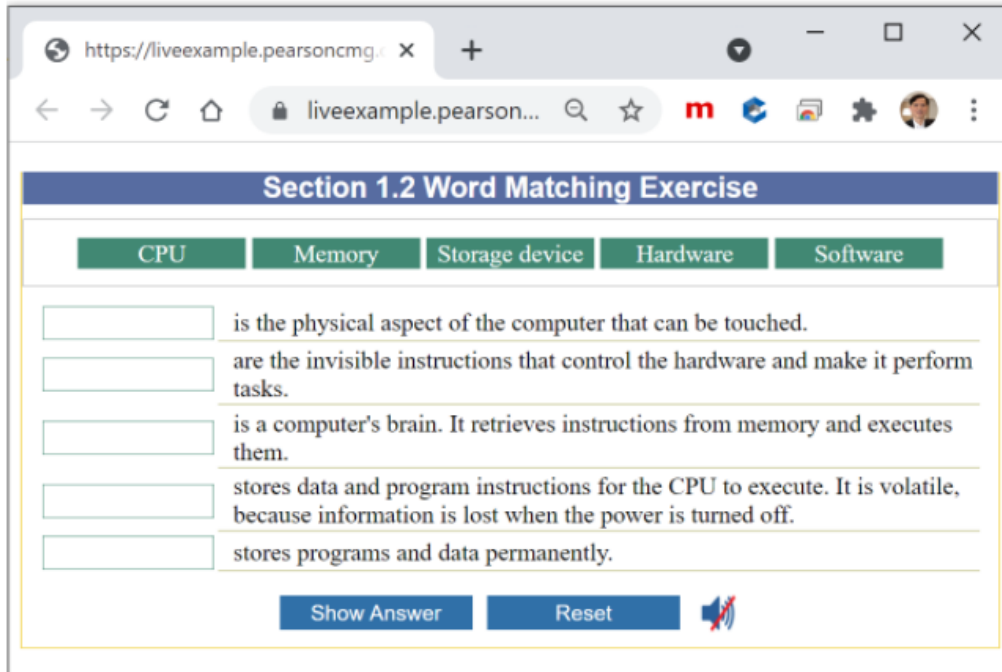


Figure 1.5: Original Word Matching Interactive Built With Static HTML AND Javascript

involved advanced programming skills and a significant time investment. The Automatic Word Match Generator empowers instructors with the ability to create *word-matching interactives* through a simple web-based GUI as shown in Figure A.1. Instructors can use the Automatic Word Match Generator to enter key terms and their descriptions. Then, they can automatically generate the HTML code for a *word-matching interactive*, thus, saving precious time and resources. Our system can be utilized by instructors from any academic discipline to create exercises that will help students learn the appropriate language corresponding to their academic subject.

As mentioned previously, we created each *word-matching interactive* manually. This was time-consuming and inefficient. We created an Automatic Word Match Generator to automatically generate *word-matching interactives*. It uses a generic model to automatically generate web pages for similar problems. In this thesis, we present the design and implementation of the Automatic Word Match Generator. In Chapter 2 and Chapter 3, we

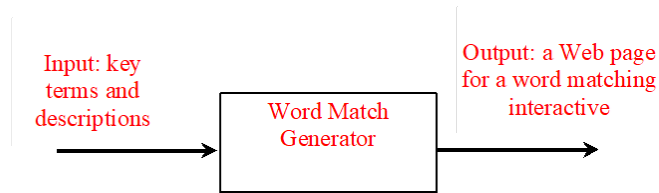


Figure 1.6: The Word Match Generator Generates a Word Matching Interactive

discuss how automatic programming can be used to generate web pages.

1.2 DEFINITION OF TERMS

1. **Automatic Programming:** the study of generative programming in the sense that the source code is generated automatically.
2. **Generative Programming:** the application of using code for a new function or software.
3. **Word-matching Interactive:** a word-matching exercise that can be embedded within an ebook.
4. **Word Matching Games:** exercises that involve matching key terms with their descriptions.
5. **Word Match Generator:** a web-based system that takes key terms and their descriptions to generate a *word-matching interactive*.
6. **Key Term:** a word or entity that can be defined for a specific topic.
7. **Description:** a statement that describes a key term.
8. **Web Page:** a hypertext document provided by a website and displayed to a user in a web browser.

9. **Javascript:** a scripting language that allows you to implement complex features on web pages.
10. **JSP** (Java Server Pages): a Java standard technology that enables you to write dynamic, data-driven pages for your Java web applications.
11. **Java:** a programming language and computing platform first released by Sun Microsystems in 1995.
12. **Spring Boot:** open source Java-based framework used to create a micro Service.
13. **View Resolver:** is a Spring Boot specific component. Once Model and View receive the data, Dispatch Servlet will transfer it to the view resolver to get the actual page view. The View Resolver provides a mapping between view models and actual views.
14. **Model:** a component of an MVC architecture that is responsible for managing the data of the application.
15. **View:** a component of an MVC architecture that is used to return a user interface output to the user in response to the user request.
16. **Controller:** a component of an MVC architecture that is responsible for controlling the way that a user interacts with an MVC application.
17. **Web Service:** used for enabling an application to invoke a method of another application.
18. **HTML:** the standard markup language for creating Web pages.
19. **Spring Boot:** open source, microservice-based Java web framework that creates a fully production-ready environment that is completely configurable using its prebuilt code within its codebase.

20. **API:** a set of programming code that enables data transmission between one software product and another.
21. **XMLHttpRequest:** an object used to interact with servers. Allows the retrieval data from a URL without having to do a full page refresh.
22. **Dispatch Servlet:** is the front controller in Spring web applications. It calls a method when a browser requests the page and combines the results with the matching jsp file to make an html document.
23. **REST:** refers to a specific set of rules that dictate how data is transmitted from a client to a server.
24. **POST Request:** used to send data to a server to create/update a resource via HTTP request.
25. **GET Request:** used to request data from a specified resource via HTTP request.
26. **XML:** a markup language and file format for storing, transmitting, and reconstructing arbitrary data.
27. **DOM (document object model):** the data representation of the objects that comprise the structure and content of a document on the web.
28. **GUI (graphical user interface):** is a system of interactive visual components for computer software.
29. **Request:** a communicative message that is transmitted between the client to a server.

1.3 REVIEW OF LITERATURE

A variety of concepts from biology to computer science can be taught through word-matching games. Specifically, subjects that use a hierarchical structure for learning vocab-

ulary. The instruction of these subjects can be enhanced with *word-matching interactives*. Forma Curran introduced the concept of word-matching games in 1994 when she showed that vocabulary was an essential part of effective communication (Arifah and Kusumarasdyati 2013). Further, Masri and Najjar proved that word-matching games were effective in two ways:

1. Games engaged students in a pleasurable manner, thus supporting them and helping them memorize new words.
2. Gains in conceptual knowledge were reported for interactive courses, regardless of whether the course was high school, college, or university level.

The average post-test score in the experimental group was higher than the average post-test score in the control group (Masri and Najjar 2014). Their studies revealed post-test scores of 80.40 for their experimental group and 77.20 for the controlled group. The measured p-value of the post-test was less than the significance value of 0.05. Therefore, the null hypothesis was rejected and the alternative hypothesis was accepted.

In a comparison of 14 classes using traditional methods with 48 classes using interactive engagement the performance of interactive engagement, and traditional lecture methods in introductory physics courses were measured (Hake 1998). Hake (1998) demonstrated that interactive classroom activities were shown to have a positive engagement effect on the 6,500 students studied. However, the relationship between cause and effect cannot be completely isolated in this non-equivalent group design.

Masri and Najjar (2014) designed an experiment that involved matching pairs of words, cards, or pictures. In their study, students had to find a partner with the appropriate card or picture. For example, students shuffled 20 word cards, 10 word matches, in random order. Then, each student was tasked with looking for a matching pair of words or pictures within a certain time until each card had the right pair (Masri and Najjar 2014). The

reported significance value from the experiment was 0.023, which was less than the p-value of 0.05. An additional study, *The Effect of Using Word Matching Games on Primary Stage Students Achievement in English Language Vocabulary in Jordan*, showed that word-matching games did not have an impact as far as gender was concerned, but they concluded that there were drastic differences in the post-test between the control and experimental groups (Masri and Najjar 2014).

The study consisted of 76 males and 82 females and reported post-test scores of 81.82 for males and 82.34 for females. The control group reported post-test scores of 76.64 for males and 77.42 for females. Essentially, the treatment had the same effect on male and female students. However, the experimental group managed to substantially improve their English vocabulary. This indicates that word-matching games have a positive effect on students' achievement in learning English vocabulary. They applied an analytical method for measuring the effectiveness of word-matching games. Few studies have actually measured the impact of games on student learning. However, this study revealed that there was a statistically significant difference between the experimental and control groups which consisted of 158 students.

Ria Dhatun and Nikmah Husein (2010) identified that vocabulary is closely related to the four language skills of reading, writing, listening, and speaking. They employed a version of *make a match*, which encouraged students to practice their vocabulary by dealing cards to each other to explain the meaning of the words. The difference between the usage of *make a match* by Ria Dhatun and Nikmah Husein (2010) differs from Forma Curran in that students used pictures instead of words in their word-matching games. In a typical *make a match* game, students are split into two groups. Then, they are paired with partners and they attempt to match words with corresponding pictures. Arifah and Kusumarasyati (2013) defined *make a match* as one of the cooperative learning techniques that is used with pairs. The disadvantages of *make a match* are that it requires guidance from teachers,

involves time restrictions, and involves organizing groups of students. However, the advantages of *make a match* are that it encourages them to cooperate, helps them avoid boredom by encouraging daily participation, which tends to lead to a more interesting classroom dynamic (Arifah and Kusumarasyati 2013).

Students who participated in *make a match* received a mean score of 18.67 in their pre-test and 25.30 in their post-test. The p-value associated with *make a match* technique was less than 0.05 (Ria Dhatun and Nikmah Husein 2010). Ria Dhatun and Nikmah Husein (2010) showed that there was a statistically significant difference between pre-test and post-test groups after they participated in *make a match*. Their experiments involved a technique where students were split into two groups, A and B. Each group received topic cards. After playing the game, students typically ended up having more discussions with teachers, which lead to an overall improvement in their vocabulary. Since the null hypothesis for this study was rejected, this supports the claim that games like *make a match* are effective at helping students improve their communication skills.

We believe that language is an essential component of learning a new topic. As stated in the introduction, games create a fun environment for students and help keep them engaged. They provide them with an outlet, especially for students who struggle to learn new vocabulary. Learning activities such as games create a fun atmosphere and keep students engaged. Ria Dhatun and Nikmah Husein (2010) showed that games that involve matching pairs of words, cards, or pictures can be used to teach a variety of academic subjects. The statistical significance of their work supports the hypothesis that word-matching games have a measurable effect on improving student vocabulary. The disadvantage of their methods is that they required the guidance of teachers.

The role of success from games in student development cannot be understated. Games bring relaxation and help students learn new words. However, they require a lot of effort on behalf of the instructor. Masri and Najjar (2014) designed a game to investigate the effect

of using word-matching games as a strategy to aid in the achievement of primary stage male and female students learning English as a foreign language. Their work has been statistically proven to help students improve their English vocabulary. In their experiments, students used pairs of cards and words to match colors, shapes, numbers, and word definitions. In each case, the experimental group subjects managed to significantly improve their English vocabulary; meanwhile, the control group did not. They showed that games promoted knowledge transfer. This is most likely due to the fact that they require student participation and active involvement with the material.

Over the last decade, books for teachers and students have focused on ways of organizing, practicing, and processing new vocabulary to help make it accessible and memorable for students. Yang and Dai (2011) showed that memorization is a major problem for students. The *word-matching interactives* created by the Automatic Word Match Generator follows a similar pattern as prescribed by Masri and Najjar (2014). The *word-matching interactives* generated by the Automatic Word Match Generator employs word-matching strategies to help students improve their vocabulary and reading comprehension. The difference between our research and that of Ria Dhatun and Nikmah Husein (2010) and Masri and Najjar (2014) is that the *word-matching interactives* generated by Automatic Word Match Generator do not require in-person participation. They can be accessed from anywhere in the world where there is a stable internet connection. *Word-matching interactives* serve a specific purpose. They help students develop their vocabulary for a particular academic topic. The success of *interactives* can be seen in Figure. 1.7, Figure. 1.9, Figure. 1.8, Figure 1.10, and Figure 1.1. Cooney (2016) reported that 83 percent of students strongly agree or agree that their understanding of the course material increased as a result of using Revel *interactives*. Cooney (2015) showed that 80 percent of students agree or strongly agree that they learned more using Revel *interactives* than they would have from a traditional printed textbook. Our intuitive user interface can be used by any instructor

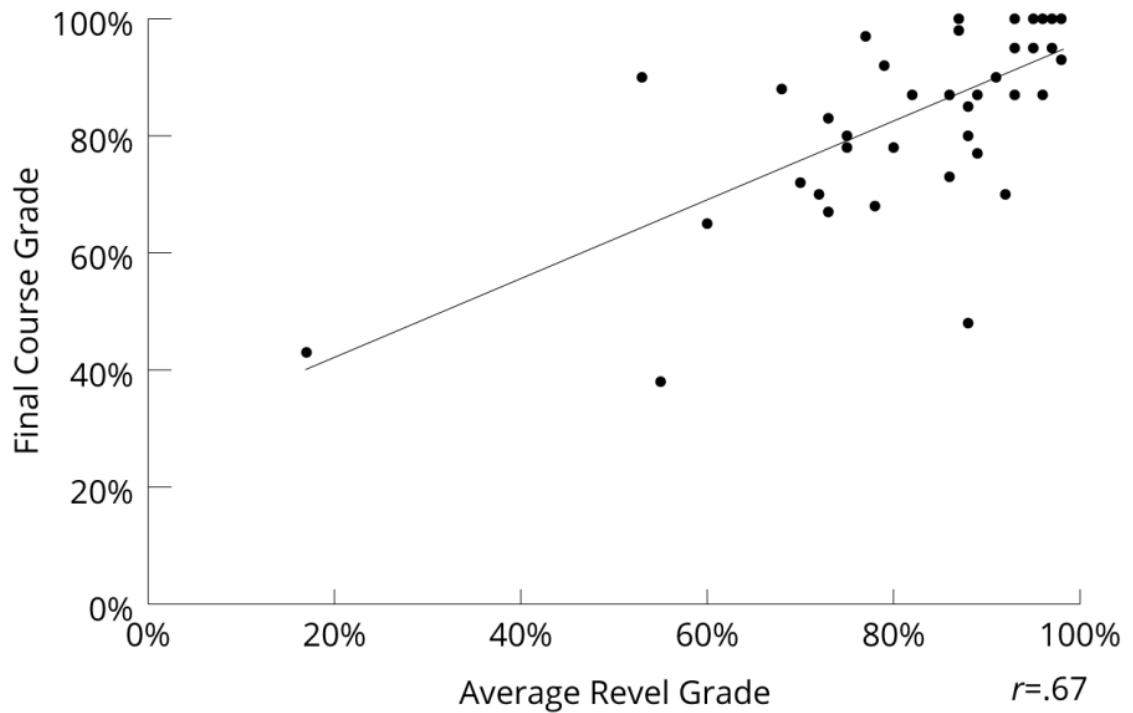


Figure 1.7: Correlation Between Average Revel Grade And Final Course Grade, Fall 2015(n=41) Cooney (2015)

to generate *word-matching interactives* based on a set of key terms and descriptions. By default, two entries for key terms and descriptions are displayed in the Automatic Word Match Generator as shown in Figure A.1. An instructor can click the Add More button to display more entries for creating additional key terms and descriptions.

Computer software products that focus on developing cognitive skills and provide an optimal learning environment have been proven by university-based research studies to help improve memory, attention, processing, and sequencing skills, which are critical for success (Agocs et. al, 2006). FAST FORWARD products used progress tracker reports to study how phonemic awareness and the acoustic properties of speech impacted the rapid development of language and reading skills.

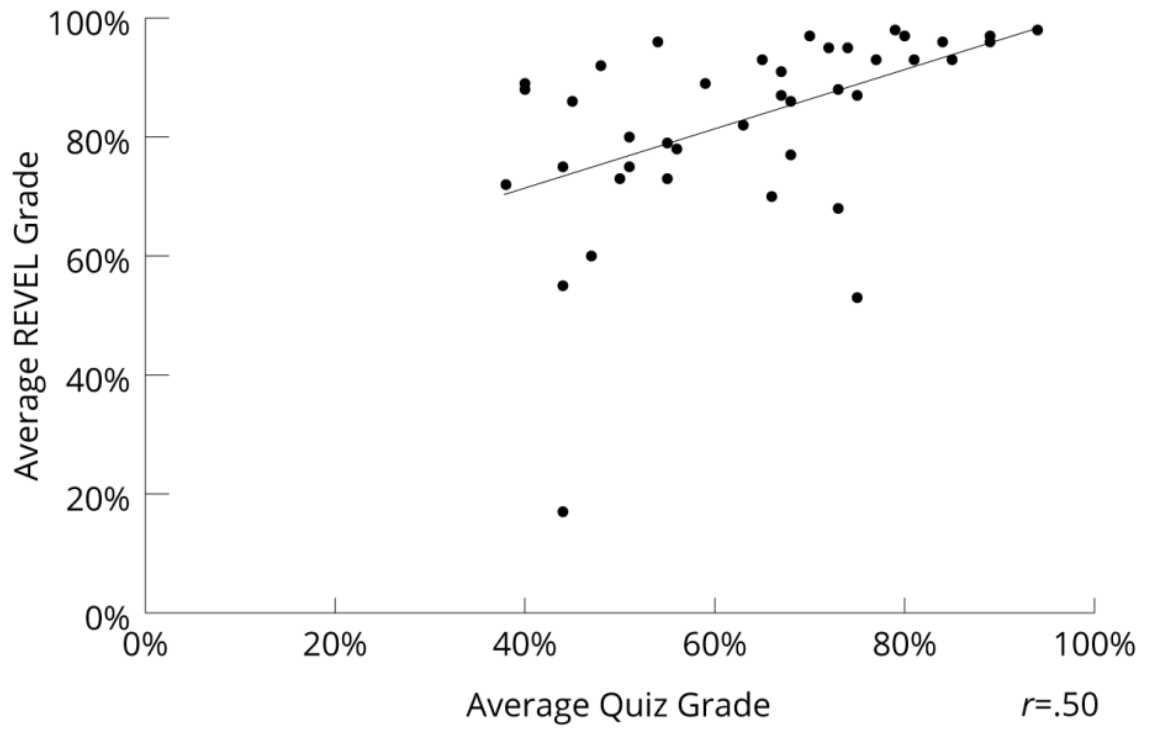


Figure 1.8: Correlation Between Average Revel Grade And Average Quiz Grade, Fall 2015
(n=41) Cooney (2015)

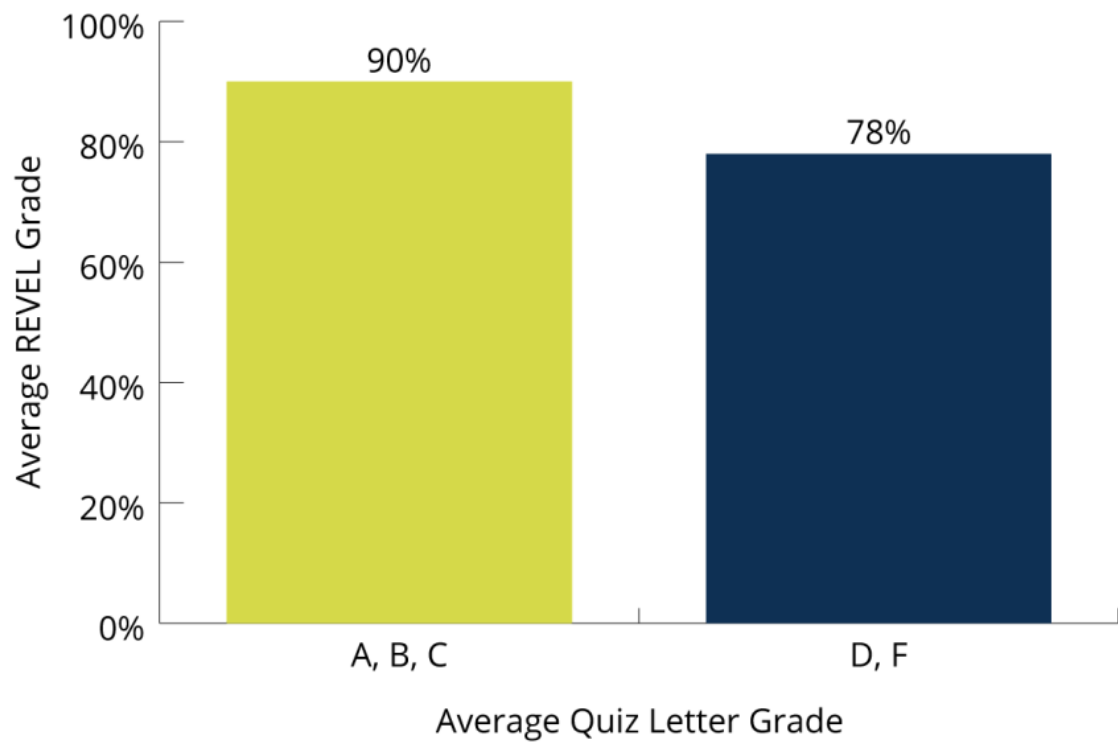


Figure 1.9: Relationship Between Average Revel Score And Average Quiz Letter Grades, Fall 2015 (n=41) Cooney (2015)

	Spring 2015 Exam Average	Fall 2015 Exam Average
Mean	59%	70%
Variance	7%	6%
Standard Deviation	27%	24%
Observations	79	41
Pooled Variance	7%	
df	118	118
t Stat	-2.14	
P(T<=t) one-tail	0.02	
t Critical one-tail	1.66	Significant
P(T<=t) two-tail	0.03	
t Critical two-tail	1.98	Significant

Table 1.1: Two-Sample t-test Assuming Equal Variances, Spring 2015 Exam Average (N=79) And Fall 2015 Exam Average (n=79) And Fall 2015 Exam Average (n=41)) Cooney (2015)

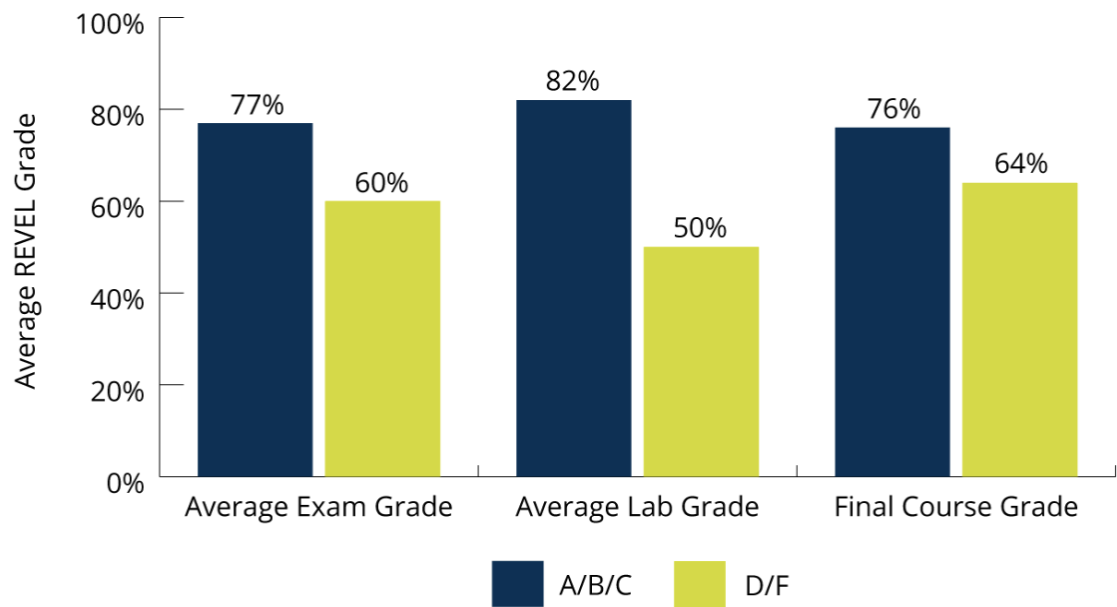


Figure 1.10: Relationship Between Average Revel Grade And Average Exam, Lab, And Final Course Grades, Spring 2016 (n=100) Cooney (2016)

The games used in FAST FORWARD products vary in nature. In *Matches and Bug Out!/Laser Match*, students chose a square on a grid and hear a sound or a word. The goal was to find each square's match and clear the grid (Agocs et. al, 2006). Students who participated in *Bear Bags and Bear Bags: More Lunch* developed an understanding of alphabetic principles (phonics) by helping "Momma Bear" sort words (on pieces of toast) into phoneme-based categories (Agocs et. al, 2006). In *Quail Mail*, a squirrel mail carrier pulls words out of a mailbag and participants sort them into different categories by clicking on the appropriate mailbox (Agocs et. al, 2006). Students develop their understanding of words meanings, auditory recognition of phonemes, and sound processing by clicking on pictures that match words that they heard (Agocs et. al, 2006). In *Ant Antics*, participants improved their vocabulary by picking one of the four *alternatives* (cards that display pictures) (Agocs et. al, 2006). As students played *Canine Crew*, they developed

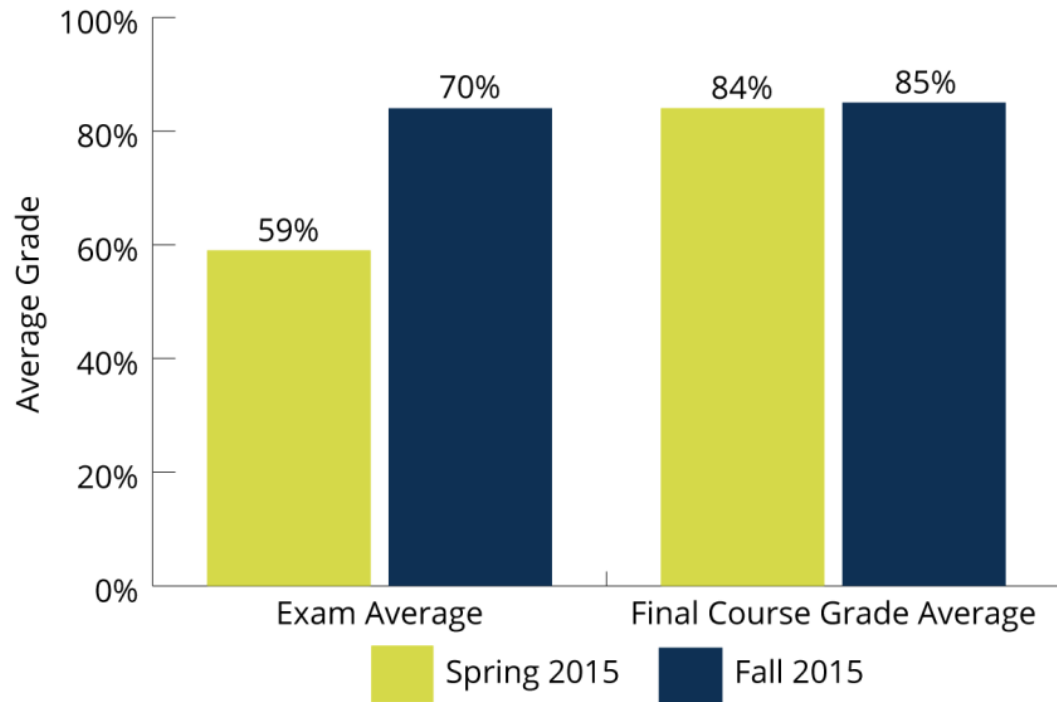


Figure 1.11: Comparison of Average Exam Scores And Average Final Course Grade, Before Implementation of Revel, Spring 2015 (n=79) And After Implementation of Revel, Fall 2015 (n=41) Cooney (2015)

their vocabulary, decoding, and automatic word recognition by matching pairs of words together on the basis of a criteria in a grid. Finally, in *Twisted Pictures*, students built their sentence comprehension by developing syntax, working memory, logical reasoning, and vocabulary by selecting sentences that more accurately described a picture based on a set of *alternatives* (Agocs et. al, 2006).

Students who used FAST FORWARD products noticed a significant improvement in their reading comprehension skills. Their improvement was measured by the Brigance Comprehensive inventory of Basic Skills as shown in Figure 1.12. The results showed that students gained three and one-half years in reading grade level. Their average letter-word identification improved by 14 months and their average gain on the passage comprehension subtest of the Woodcock-Johnson III Tests of Achievement was two years.

FAST FORWARD products took a previously successful concept, computer software that focuses on developing learning and cognitive skills, and improved upon it. They provided computer-based products that combined an optimal learning environment with a focus on early reading and cognitive skills (Agocs et. al, 2006). They added seven word-matching games to their software. Their contribution led to an overall improvement of reading comprehension skills of the participants. They noted that the longer the students used the product, the more their skill level improved.

Fundamentally, word-matching games help students learn the relationships between words and their meanings. An Automatic Word Match Generator provides instructors with the capability to create *word-matching interactives* which are analogous to *Canine Crew*, *Matches and Bug Out!/Laser Match*, *Bear Bags and Bear Bags: More Lunch*, *Quail Mail*, *Cards*, *Ant Antics*, *Canine Crew*, and *Twisted Pictures*. The *word-matching interactives* generated by the Automatic Word Match Generator get students to match key terms to their descriptions through a drag and drop action.

Word matching games increase student participation by providing them with an oppor-

Learning Curve: Fast ForWord Middle & High School

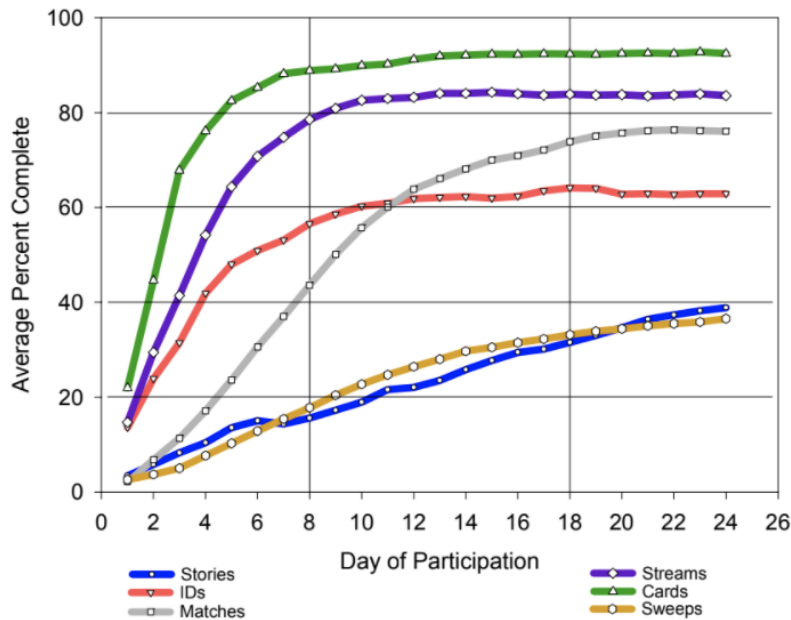


Figure 1.12: Learning Curve: Fast Forward Middle and High School (Agocs et. al, 2006)

tunity to get away from traditional learning activities. In games like *Canine Crew*, students were forced to change the way they viewed the relationships between words and meanings. The daily practice of matching different pairs of words and meanings has been shown to lead to an overall improvement in reading comprehension skills (Agocs et. al, 2006).

The effects of student participation in games used by FAST FORWARD products can be seen in Figure 1.12. This graph shows that as students practice their vocabulary with word-matching games their ability to correctly match terms with their meanings increases. This supports the claim that word-matching games help students improve their vocabulary and reading comprehension skill levels.

1.4 DESCRIPTION OF REMAINING CHAPTERS

The purpose of the methodology section is to describe the requirements specification, design, implementation, and testing of the Automatic Word Match Generator. In the requirements specification section, we will discuss the problems our research aims to address. In the design section, we will cover the reasoning behind why we selected specific components used in the software implementation design of an Automatic Word Match Generator.

Then we will discuss some of the obstacles that we encountered while planning and implementing our software. We will then describe the process of creating an Automatic Word Match Generator in the implementation section. This will be accomplished by walking through a specific set of development phases. In each of these phases, we will explain how we implemented our software. Then we will provide the reader with an example of how *word-matching interactives* are used in the testing section of the methodology. In the research components section, we will provide a brief overview of automatic programming. Then, we will cover how we used a generic model to generate a web page and how our pattern can be applied to other problems.

The purpose of the automatic programming section is to provide the reader with a basic historical context of its relationship to our research. We will cover specific examples of how previous researchers used it. In the discussion on using a generating model for creating a web page we will learn what a generator is. Then, we will describe how we applied the generator model to our problem. Finally, we will look at how we can apply the generator model to other problems. In the conclusion section, we will describe how our tool saves instructors' time and resources while providing students with easy accessibility to learning. Then, we will provide a link to a live demonstration of our tool. In our discussion of future work, we will cover the next steps for adding additional features.

CHAPTER 2

METHODOLOGY

2.1 REQUIREMENTS SPECIFICATION

The specifications for the Automatic Word Match Generator consisted of the following criteria:

1. Instructors shall be able to automatically create interactive word-matching games as shown in Figure 1.5.
2. Students shall be able to access the games online.

We have addressed the desired specifications by providing instructors with a tool that will help them consolidate the content from their course material into *word-matching interactives*. Instructors will be able to use the interface shown in Figure A.2 to create *word-matching interactives*. Students can access these *word-matching interactives* by visiting the URL associated with *word-matching interactive*. An example of retrieving a *word-match interactive* by its ID is shown in Figure 1.3. The generated *interactives* meet the needs of the students by providing them with online study material which can be used to help them improve their vocabulary by matching key terms with their descriptions. The Automatic Word Match Generator meets the requirements listed above by providing students with online study material and enabling instructors with the ability to create *interactives* automatically.

The input to the Automatic Word Match Generator program is the instructor input for key terms and descriptions. The output from the program is a web page that displays a *word-matching interactive*. From the users perspective the Automatic Word Match Generator can be used to automatically generate *word-matching interactives* that provide students with the intent to learn new vocabulary and improve their reading comprehension skills.

From the perspective of a developer, the specifications for the Automatic Word Match Generator were:

1. Enable the storage and retrieval of *word-matching interactives*.
2. Capture the input data necessary to create *word-matching interactives*
3. Provide a web interface for displaying *word-matching interactives*
4. Develop a drag and drop algorithm that facilitates an interactive learning experience using the derived data from the output.

The specifications described at the beginning of this section were designed with the intention of meeting the requirements mentioned above. Using trial and error, the development of each specification was achieved. The input to output validation was tested by generating example *word-matching interactives* based on a quiz. The output was manually tested by dragging and dropping key terms to their descriptions.

The algorithm used in the development of the Automatic Word Match Generator is a drag and drop algorithm. The list of major functions, as well as the inter-relationships used in the program are described in the implementation section.

2.2 DESIGN

The design of the Automatic Word Match Generator was based on a custom multi-tiered design pattern. The technology stack that we used to build our application consisted of HTML, CSS, Javascript, Java Server Pages and Spring Boot (a popular micro-service framework). Like most web applications, the Automatic Word Match Generator client submits an HTTP request to a server which processes the request and returns a response. In our case, an XMLHttpRequest object was designed to send a request body consisting of base64 encoded HTML code to a server via HTTP POST method. When the server

accepts the data, the HTML code is saved as a *word-matching interactive* on the server. The value in utilizing an XMLHttpRequest object is that it is asynchronous. In other words, instructors are able to create *word-matching interactives* without having to refresh the page. The interaction for this behavior is shown in Figure 2.6.

To understand the custom multi-tiered application that we have designed, we will describe how it can be used through an example. When an instructor wants to create a new *word-matching interactive*, they have to visit the instructor graphical user interface (GUI) as shown in Figure A.2. The request to visit this page is processed by a dispatch servlet. The dispatch servlet will study the class of the request and send it to the relevant controller class, in this case WordMatchController. The controller will then return the view *wordmatch.jsp* which will present the data to model and view. The source code for the WordMatchController and *wordmatch.jsp* are shown in Appendix C. The process for this interaction is shown in Figure 2.1. The steps for this process are listed below:

1. The dispatch servlet processes the request from the browser.
2. The mapping handler then routes the request to the correct controller class.
3. The WordMatchController then processes the request, which then calls the appropriate method from the WordMatchService.
4. The model data and view name (*wordmatch.jsp*) are returned.

We have designed the Automatic Word Match Generator to be a self-contained application so that it can be deployed to any web server that supports a Java virtual machine and I/O operations. All of the I/O (input/output) file operations take place on a single server, as shown in Figure 2.7. The purpose of the class WordMatchController.java is to process incoming requests from the client. The methods within the controller were arranged to the call methods from WordMatchService.java which were designed to save *word-matching*

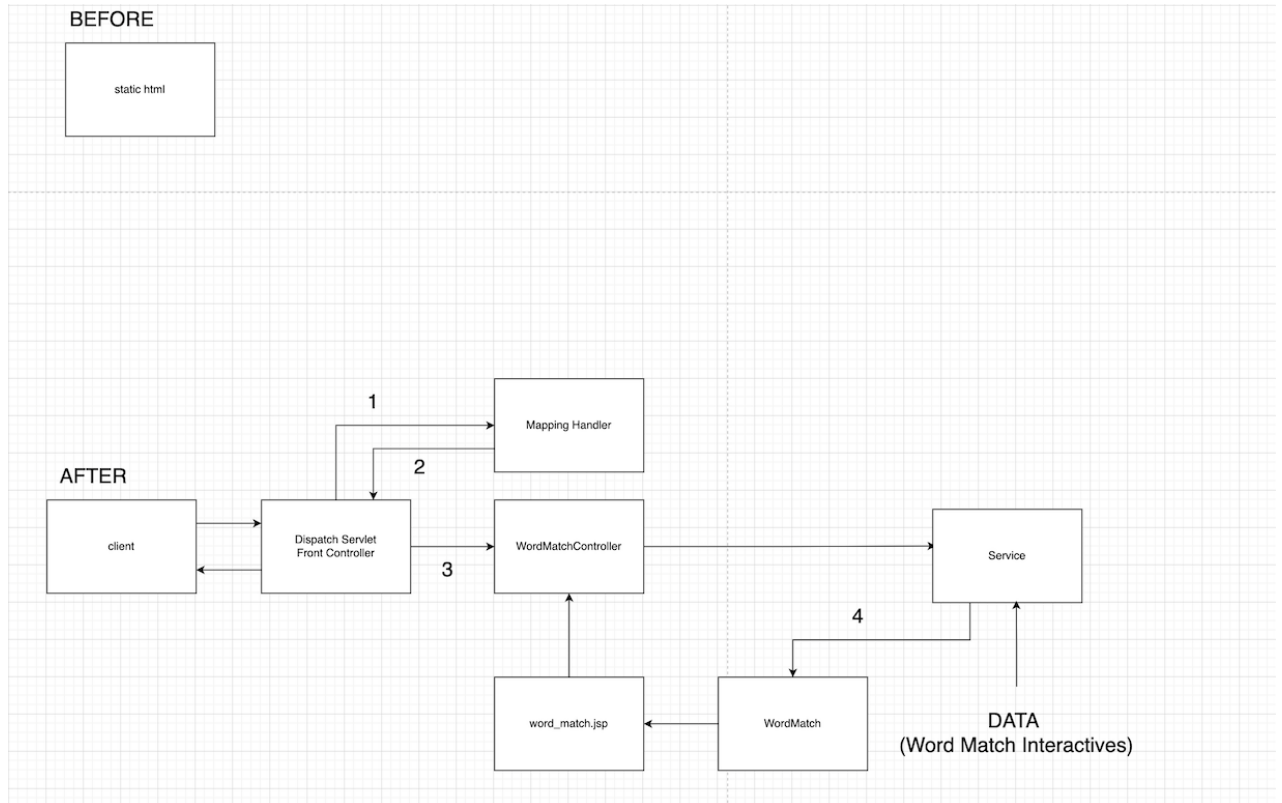


Figure 2.1: Automatic Word Match Generator Custom multi-tiered Application Diagram Before And After.

interactives on the server. The client to server interaction is captured in Figure 2.6. Subsequent methods in `WordMatchController` were designed to return the newly created *word-match.jsp* file. The methods used in the `WordMatchController` facilitate the following operations:

1. Retrieve *word-matching interactives* from the server.
2. Save *word-matching interactives* on the server.

As we can see in Appendix C in `WordMatchService.java`, the methods `saveWordMatchJSP` and `getWordMatch` handle the controller interactions listed above. In the case of `saveWordMatchJSP`, this method was used to save the content sent by the HTTP POST method. The purpose behind this method is to save a *word-matching interactive* on the server.

The model `View.java` was designed to facilitate the sending and receiving of data related to displaying *word-matching interactives*. The view, *wordmatch.jsp*, was tailored to follow the original structure of the first *word-matching interactive* introduced at the beginning of the project as shown in Figure 1.2. In Figure 2.3, we can see how the request is sent to the server by using the chrome developer tools panel to the right. The specific request body used in the request is shown in Figure 2.3.

Finally, we can see the returned ID associated with the *word-matching interactive* in the response shown in Figure 2.4. Following best practices for developing web applications, we have ensured that proper response codes are returned. In this case, we return a 201 CREATED response code when an object has been saved to the server this is shown in Figure 2.2.

One of the first obstacles in designing the Automatic Word Match Generator was to develop a method for generating the static HTML code. The HTML generating function that we landed upon leverages a Javascript method. The source code for this method is shown

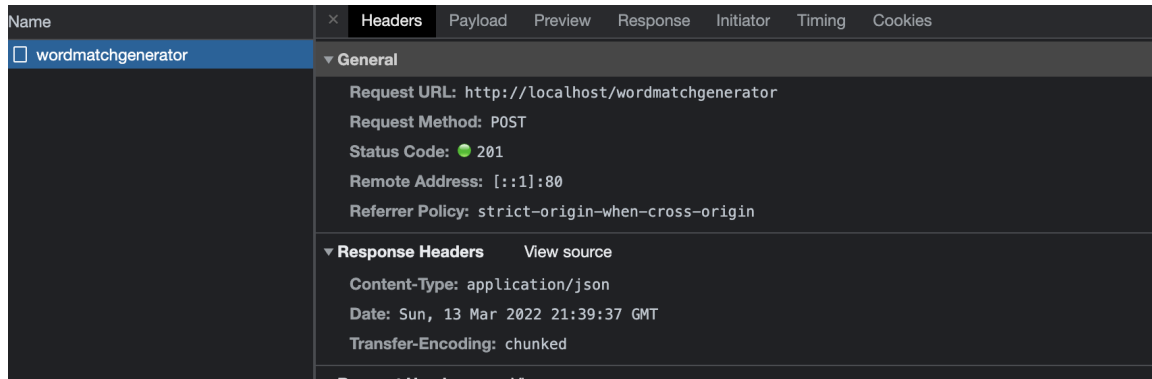


Figure 2.2: Sending a Word-Matching Interactive To The Server

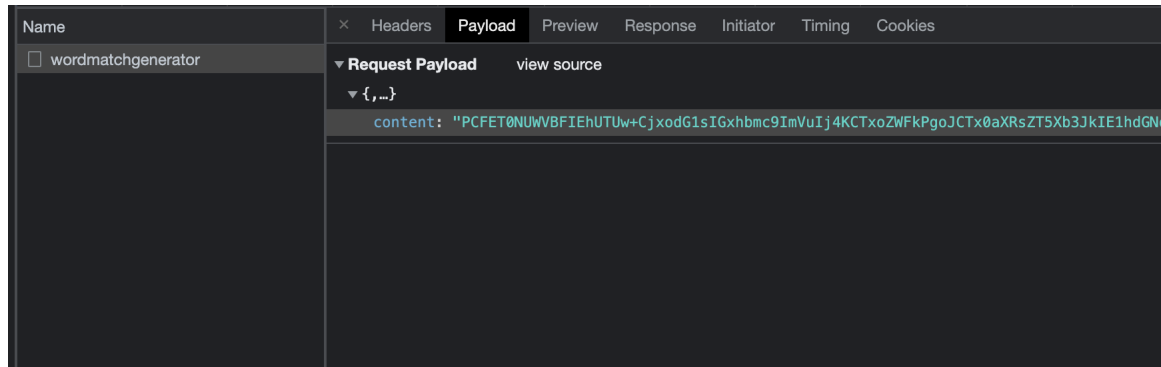


Figure 2.3: XmlHttpRequest Request Body Shown In Chrome Developer Tools

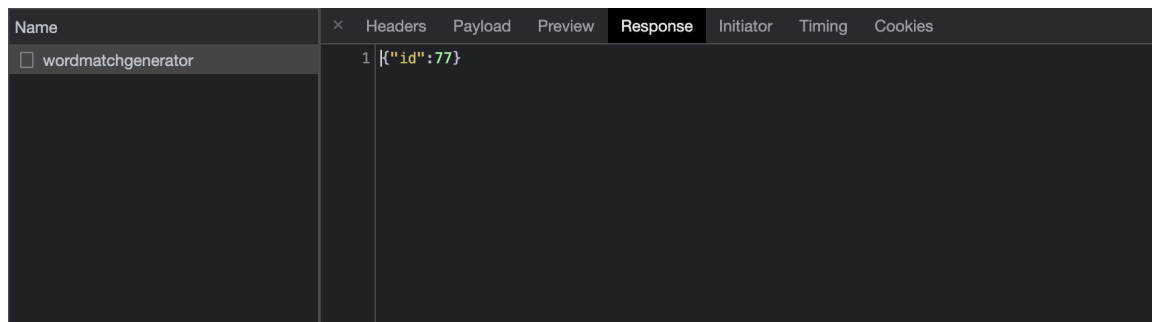


Figure 2.4: Id Returned After Saving a Word-Matching Interactive

in Appendix C under *generate_html*. The primary function of this method is to retrieve content from HTML inputs, key terms, and descriptions, then concatenate them into a string. The key terms and descriptions are randomly shuffled before they are concatenated. Then they are processed into a string, which represents a *word-matching interactive*. After this process is completed, the entire HTML string is then displayed in a textarea box as shown in Figure 2.16. We used the resulting HTML string shown in the textarea box to design the *render_html* method which was used to render a *word-matching interactive*. Finally, we used the *render_html* method to trouble shoot the rendered *word-matching interactive* HTML string. Once the final output matched the original *word-matching interactive* shown in Figure 1.5, we designed the *save_content* method. The *save_content* method was used to send the generated HTML string from the textarea to the server through an XMLHttpRequest, as mentioned previously.

2.3 IMPLEMENTATION

The process of implementing our software began with the planning phase. During the planning phase, we created an initial set of objectives to achieve before implementing the project. The goal of our project was to use the base structure of the original *word-matching interactive* as shown in Figure 1.5. The initial goal was to create an HTML parser that could recreate the same HTML page for different inputs. The example shown in Figure 1.5 demonstrated the feasibility of completing the project. After analyzing the HTML structure, we defined a consistent output, i.e., a (*word-matching interactive*), based on a predetermined set of input criteria, i.e., (key terms and descriptions).

An overview of the steps used to develop the Automatic Word Match Generator are shown below:

1. Selected languages for developing the program.

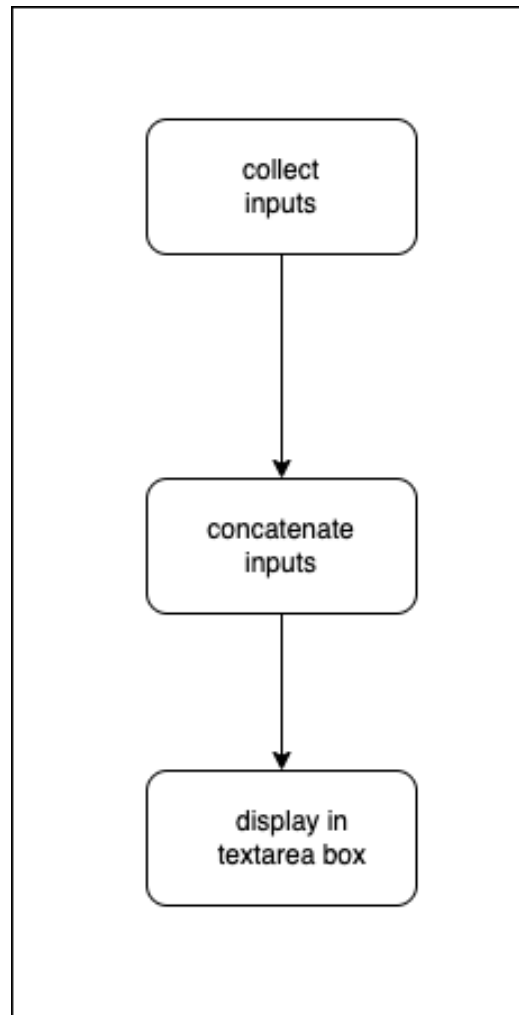


Figure 2.5: Flow Diagram For Generating html

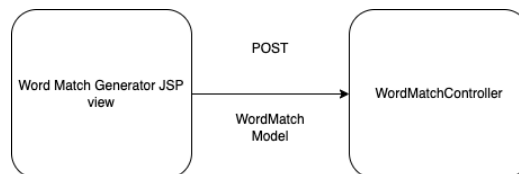


Figure 2.6: XMLHttpRequest From The Word Match Generator Client To Server

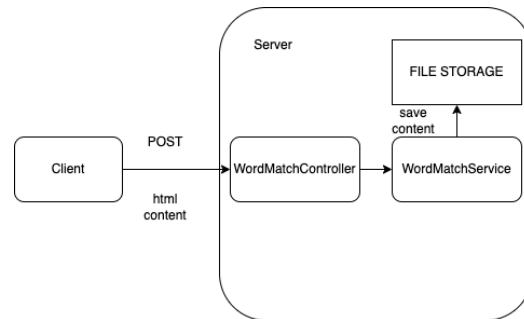


Figure 2.7: Technical Diagram For The Automatic Word Match Generator

2. Identified an expected output for the program.
3. Created a proof of concept that collected the data needed to generate the derived data to create a *word-matching interactive*.
4. Developed a user interface for collecting the input data.
5. Implemented an HTML generating function to collect the input data and generate the derived data.
6. Implemented an HTML rendering function to display the expected output with the derived data.
7. Developed a system for displaying the output to the user.

The major data structures used in the development of the Automatic Word Match Generator were two arrays: one for capturing the key terms, and another for capturing the descriptions.

Before implementing our project, a survey of possible solutions was conducted. In this phase, various programming languages were evaluated. Initially, Java and Python were selected as potential candidates for programming languages. Given that Python is typically used for scripting we decided to use Java for two reasons: it supports a set of frameworks for implementing web applications and we had more experience using it. Since the original

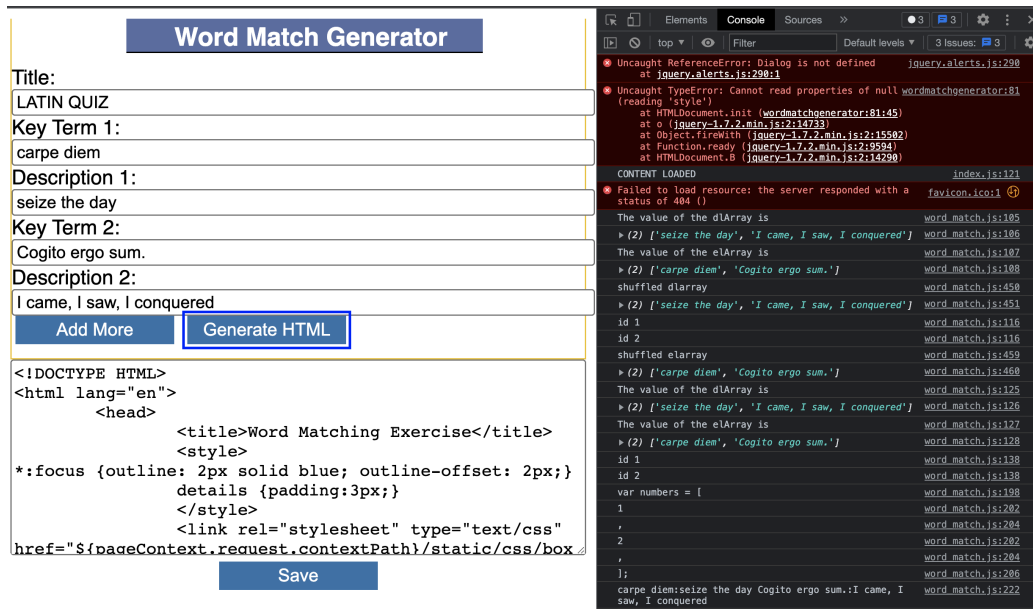


Figure 2.8: Logging The Output From The Javascript Console

word-matching interactive only consisted of Javascript, CSS, and HTML. We decided to use a popular web application framework, Spring Boot. This framework would allow us to create an application that could be used to generate, serve, and store *word-matching interactives* as static HTML pages.

To accomplish the goal of generating static HTML pages, we used the Document Object Model (DOM) to manipulate the value attributes of the key terms and descriptions associated with the instructor GUI as shown in Figure 2.14. The direct manipulation of these objects provided our program with access to the value attributes associated with each of the inputs. Therefore, enabling our application to extract specific input values and concatenate them into a string of HTML which represents a *word-matching interactive*.

The implementation of the Automatic Word Match Generator consisted of four phases. Phase one was a two-step process. In step one, we used Javascript to get the values of the HTML input elements. Then, we used chrome developer tools to log the input values to the Javascript console as shown in Figure 2.8.

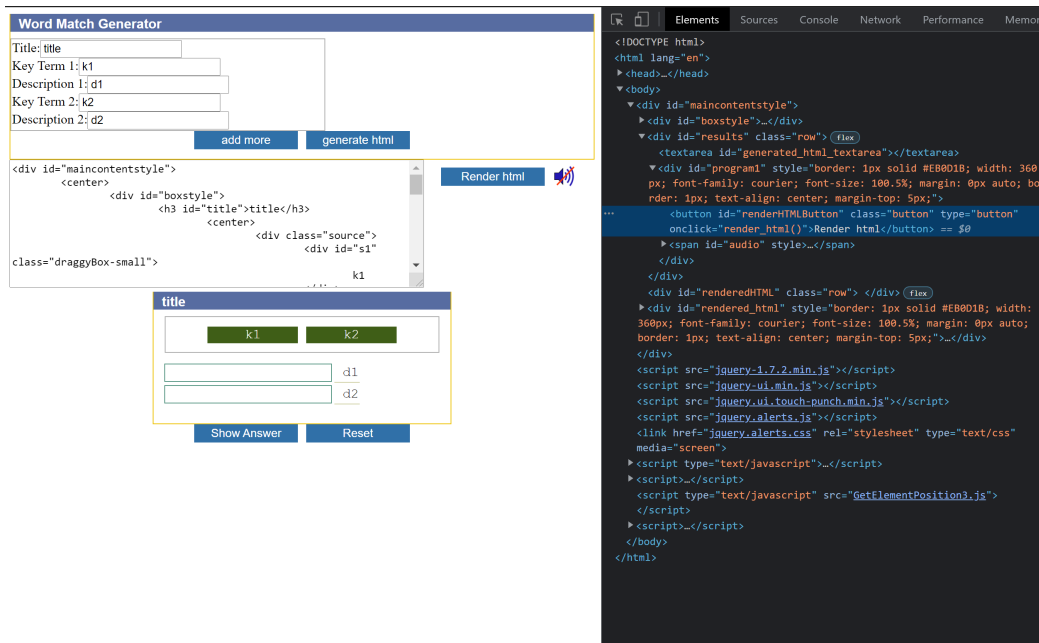
In step two, we recreated the HTML string which represented a *word-matching interactive* as a string. Then, we logged the HTML string to the javascript console as shown in Figure 2.8. In phase two, we combined steps one and two from phase one into a function that concatenates two strings into a single string. One of the strings contained the values from the value attributes from the input elements. The other string contained the HTML string which represented the *word-matching interactive*. The combination of these two strings represented the HTML for a *word-matching interactive* as a single Javascript string.

Phase three consisted of displaying the concatenated string from phase two in a textarea element below the instructor GUI as shown in Figure 2.14. Since it was difficult to read the Javascript string from Javascript console, a systematic approach of copying and pasting the output from the textarea was repeated until the desired output matched the exact HTML structure for the original *word-matching interactive* as shown in Figure 1.5.

In phase four, one of the obstacles that we faced was rendering the generated *word-matching interactive* from the textarea box. The initial attempt at developing a rendering section is shown in Figure 2.9.

At first, we attempted to render the *word-matching interactive* in a separate tab using the `new_window.document.write` method. In this scenario, a user clicks the Render HTML button and a new tab should appear with the rendered HTML. This test should have allowed us to drag and drop the key terms onto their descriptions. However, the actual result was that when Render HTML was clicked, a new tab appeared with the rendered HTML, but the drag and drop functionality did not work. This was because the Javascript libraries used by the application were not being loaded into the page properly. Therefore, a new plan was devised to render the *word-matching interactive* in the same page as the instructor GUI as shown in Figure 2.14. This was a step forward in the right direction, but we still needed to find some way to show the rendered HTML in a separate tab.

To overcome this obstacle, we decided to implement a micro-service based web ap-



plication that leveraged a custom multi-tiered design pattern. This decision was made so that our application could display a *word-matching interactive* in a separate view by its ID. This decision solved the problem that we faced with loading the Javascript libraries. It provided our application with the ability to dynamically load Javascript libraries. The custom classes used in the implementation of our micro-service are shown in Figures 2.10, 2.11, 2.12, 2.13.

The classes shown above in Figures 2.10, 2.11, 2.12, 2.13 are custom classes used in a custom multi-tiered web application. The primary purpose of the controller class shown in Figure 2.10 is to provide URL routes for our application. The specific routes used in this class are shown in Appendix C. The purpose of the classes shown in Figure 2.11 and Figure 2.12 is to provide models, which can be used to transport data throughout the service. The purpose of the View class shown in Figure 2.12 is to keep track of the *word-matching interactives* saved on the server. The WordMatch class shown in Figure 2.11 is used as a model to transport the *word-matching interactive*. The model exists entirely on the server.

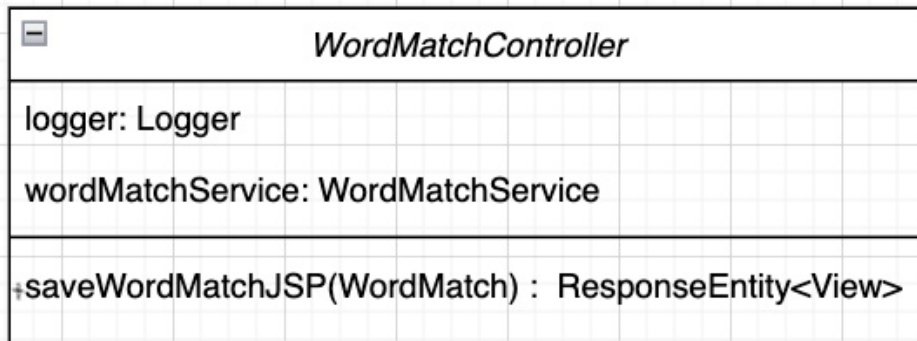


Figure 2.10: Word Match Controller

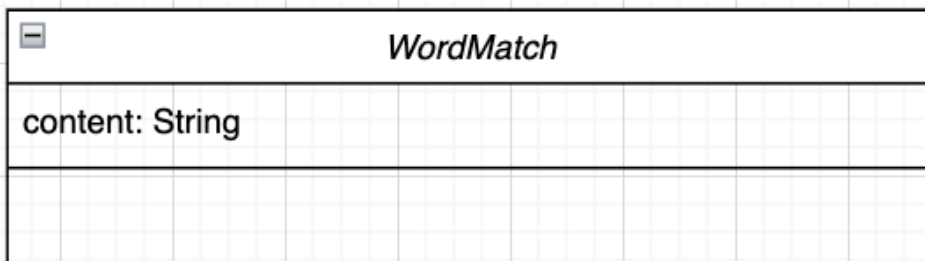


Figure 2.11: Word Match

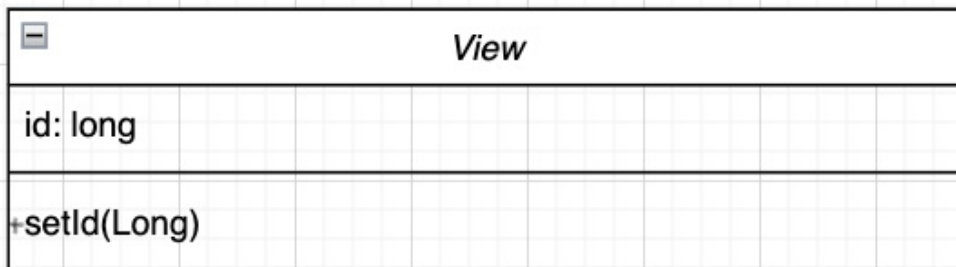


Figure 2.12: View

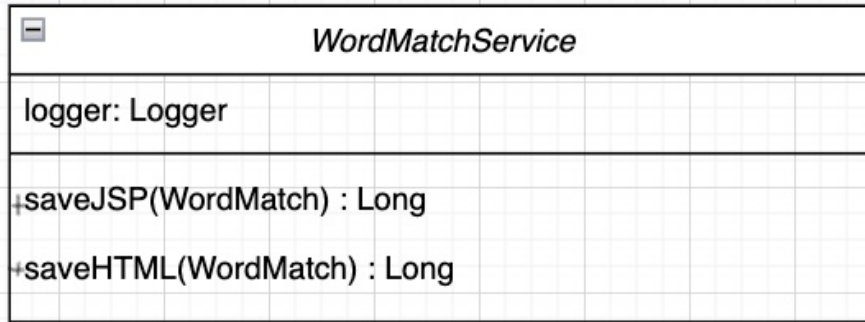


Figure 2.13: Word Match Service

The `WordMatchService` class shown in Figure 2.13 provides two methods

1. `saveJSP`
2. `saveHTML`

The method `saveJSP` was used to save a *word-matching interactive* as a `jsp` (Java Server Pages) file on the server and `saveHTML` was used to convert a `jsp` (Java Server Pages) file to a static HTML file.

2.4 DRAG, DROP, AND MATCH ALGORITHM

One of the key features of the Automatic Word Match Generator is the Drag, Drop, and Match algorithm. In this section, we present this algorithm.

The code generated from the Automatic Word Match Generator is an interactive HTML page that enables a user to be able to drag a key term from the key term pool to a matching description. If there is no match, the key term is sent back to the key term pool. If there is a match, the key term stays in the matched row. The matched row is set with a new color.

The key terms and descriptions are saved in the separate arrays and they are randomly shuffled. A map was used to map the key with its corresponding description. The algorithm works with Javascript events and it is described as follows:

The screenshot shows a web browser window with the title 'Word Match Generator' and the URL 'livelab.georgias...'. The page content includes a title field, two key term fields (Key Term 1 and Key Term 2), and two description fields (Description 1 and Description 2). At the bottom, there are two buttons: 'Add More' and 'Generate HTML'.

Figure 2.14: Initial Screen For The Automatic Word Match Generator

2.4.1 ALGORITHM: DRAG, DROP, AND MATCH

1. successCount is set to 0.
2. Obtain the index of the dragged key term as sourceIndex.
3. Obtain the index of the row where the key term was dropped as targetIndex.
4. if (sourceIndex != targetIndex) return the key term back to the key term pool.
5. Disable the key term object to be undraggable.
6. Set a new color for the matched row object.
7. Test if all key terms are matched. If so, display a Congratulations dialog.
8. Increase the successCount by 1.

The UI has a Reset button, clicking the Reset button also sets successCount to 0. This is an example of event-driven programming. The algorithm is actually triggered when the user drops a key term on a row. Note that there are two parts in this project: one is the generator program, and the other is the generated program. This algorithm was used in the generated program.

2.5 TESTING

An example of the Automatic Word Match Generator can be seen at <http://livelab.georgiasouthern.edu/wordmatchgenerator> as shown in Figure 2.10.

The functionality of our application can be described in a simple sequence of steps. In step one, an instructor can simply enter a title, Key Term 1, Description for Key Term 1, Key Term 2, and Description for Key Term 2. In step 2, he or she can click the Add More button to create more entries for key terms and their descriptions. For example, an instructor can enter the following inputs and descriptions shown in Figure 2.15. An Automatic Word Match Generator then displays the HTML code as shown in Figure 2.18. An instructor can click the Post button to send the generated HTML code for the *word-matching interactive* to the server as shown in Figure 2.17.

After clicking the Post button to post the *word-matching interactive* to the server, the server then saves the generated HTML file for the *word-matching interactive*. It then creates a URL for the exercise. After the generated HTML file is posted, a View button is displayed, as shown in Figure 2.17. An instructor can access any of the exercises by clicking the View button. The View button serves two purposes: first, it renders the HTML code for the exercise; second, it shows the URL for the exercise on the server. Clicking the View button displays the exercise using the URL, as shown in Figure 2.17. The instructor can give this URL to the student. An example of the created *word-matching interactive* is shown in Figure 2.18.

Word Match Generator

Title:
Section 1.2 Word Matching Exercise

Key Term 1:
CPU

Description 1:
is a computer's brain. It retrieves instructions from memory and executes them.

Key Term 2:
Memory

Description 2:
stores data and program instructions for the CPU to execute. It is volatile, because information is lost when the power is turned off.

Key Term 3 :
Storage Device

Description 3 :
stores programs and data permanently.

Key Term 4 :
Hardware

Description 4 :
is the physical aspect of the computer that can be touched.

Key Term 5 :
Software

Description 5 :
are the invisible instructions that control the hardware and make it perform tasks

Add More Generate HTML

Figure 2.15: Key Terms And Description Inputs

Word Match Generator

Title:
Section 1.2 Word Matching Exercise

Key Term 1:
CPU
Description 1:
is a computer's brain. It retrieves instructions from memory and executes them.

Key Term 2:
Memory
Description 2:
stores data and program instructions for the CPU to execute. It is volatile, because information is lost when the power is turned off.

Key Term 3 :
Storage Device
Description 3 :
stores programs and data permanently.

Key Term 4 :
Hardware
Description 4 :
is the physical aspect of the computer that can be touched.

Key Term 5 :
Software
Description 5 :
are the invisible instructions that control the hardware and make it perform tasks.

[Add More](#) [Generate HTML](#)

```
<!DOCTYPE HTML>
<html lang="en">
  <head>
    <title>Word Matching Exercise</title>
    <style>
*:focus {outline: 2px solid blue; outline-offset: 2px;}
details {padding:3px;}
</style>
<link rel="stylesheet" type="text/css"
href="{nameContext.request.contextPath}/static/css/boxes.css" />
  
```

[Post](#)

Figure 2.16: Generated Html Code

Word Match Generator

Title:
Section 1.2 Word Matching Exercise

Key Term 1:
CPU

Description 1:
is a computer's brain. It retrieves instructions from memory and executes them.

Key Term 2:
Memory

Description 2:
stores data and program instructions for the CPU to execute. It is volatile, because information is lost when the power is turned c

Key Term 3 :
Storage device

Description 3 :
stores programs and data permanently.

Key Term 4 :
Hardware

Description 4 :
is the physical aspect of the computer that can be touched.

Key Term 5 :
Software

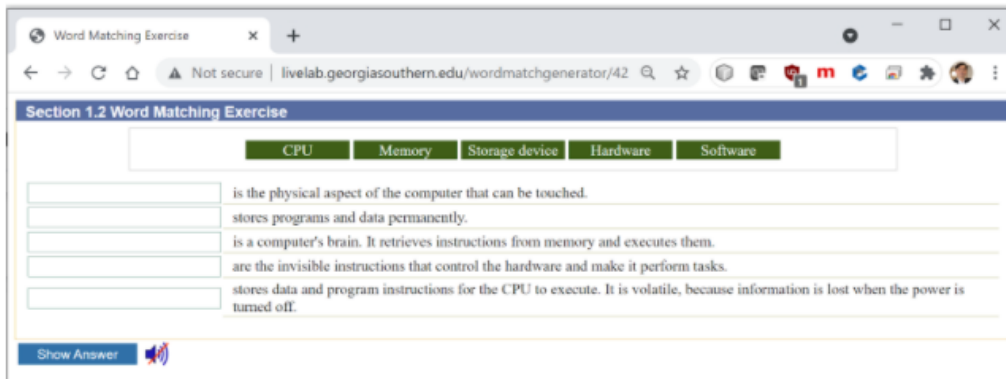
Description 5 :
are the invisible instructions that control the hardware and make it perform tasks.

[Add More](#) [Generate HTML](#)

```
<!DOCTYPE HTML>
<html lang="en">
  <head>
    <title>Word Matching Exercise</title>
    <style>
      *:focus {outline: 2px solid blue; outline-offset: 2px;}
      details {padding:3px;}
    </style>
    <link rel="stylesheet" type="text/css"
      href="{nameContext.request.contextPath}/static/css/boxes.css" />
  </head>
</html>
```

[Post](#) [View](#)

Figure 2.17: Clicking Post Button



The screenshot shows a web browser window with the title "Word Matching Exercise". The address bar displays "Not secure | livelab.georgiasouthern.edu/wordmatchgenerator/42". The page content includes a header "Section 1.2 Word Matching Exercise" and a list of terms: CPU, Memory, Storage device, Hardware, and Software. Below the terms are five matching questions, each with a blank input field for the answer.

Section 1.2 Word Matching Exercise

CPU Memory Storage device Hardware Software

is the physical aspect of the computer that can be touched.

stores programs and data permanently.

is a computer's brain. It retrieves instructions from memory and executes them.

are the invisible instructions that control the hardware and make it perform tasks.

stores data and program instructions for the CPU to execute. It is volatile, because information is lost when the power is turned off.


Show Answer 

Figure 2.18: Generated Word-Matching Exercise

CHAPTER 3

RESEARCH COMPONENTS

3.1 SURVEY OF AUTOMATIC PROGRAMMING

Automatic programming is used to write a program that generates another program based on certain specifications. For example, a compiler is an automatic program that takes a source code and generates an executable. In a broad sense, automatic programming can be classified into two types:

1. **Generative programming:** the application of reusing code for a new function or software.
2. **Code generation:** a mechanism to produce the executable form of a program.

The Automatic Word Match Generator is an example of code generation. It takes key terms and their descriptions as an input and generates an HTML source code.

The research of automatic programming started in the 1970s. The initial goal was to provide a specification and let the computer automatically generate a program that met the specification. Unfortunately, the task of automatically generating a program is harder than expected. Formal specifications were proposed to give precise requirements in a mathematical structure (Balzer 1985), (Jazayeri 1976), (Ngolah and Wang 2004), (Olsson 1995), (Whalen and Heimdahl 1999), (YooN 1990). Experimental systems were developed that take the requirements written in formal specification and generate a program automatically. However, these systems are not used in industry, because there is a wide gap between the high-level specification and target implementation (Palshikar 2001).

In recent years, domain-specific automatic programming systems have been developed. A system called “Wrex” was created to automatically generate Python code for analyzing data (Balzer 1985). A system called “Falx” was created to automatically gener-

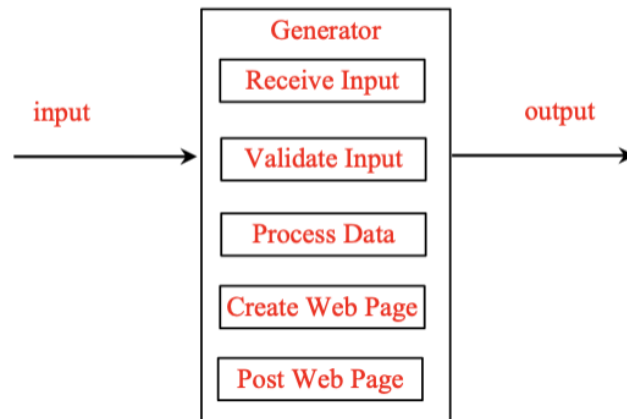


Figure 3.1: A Generic Model For Generating a Web Page.

ate R programming code for visualizing data (Wang et al., 2021). A system called “Scythe” was created to generate certain types of SQL statements (Wang et al., 2021).

Inspired by the current development in the domain-specific automatic programming systems, we developed an Automatic Word Match Generator that automatically generates a *word-matching interactive*. We hope that the knowledge obtained in the Automatic Word Match Generator can be expanded and applied into generating web pages for other types of problems.

3.2 A GENERIC MODEL FOR GENERATING A WEB PAGE

To apply and extend the Automatic Word Match Generator to other types of problems in the same domain such as code animation, we propose a generic model for generating a web page as shown in Figure 3.1

The generator is a web-based program. It contains the five components: Receive Input, Validate Input, Process Data, Create Web Page, and Post Web Page. In the case of the Automatic Word Match Generator, the input is entered from text fields and stored

in arrays. The validation might be simply to check if key terms or their descriptions are empty. In Process Data, data is randomly shuffled and the key terms are mapped to their descriptions. Create Web Page uses HTML, CSS and javascript to create a *word-matching interactive*. Post Web Page automatically posts the generated Web page to a web server so that the page can be viewed on the internet.

The problems in the same domain are similar in the sense that the generator receives the input and generates a web page. However, the implementation of the generators will be different because the problems are not the same. For example, the input may be in different forms. So, the input receiver will be different and validation of the input will also be different. Processing data needs to be customized to tailor to the specific problem. Create Web page needs to be written to generate the page that meets the specification of the problem. Post Web Page will be the same for the problems that fall into this model.

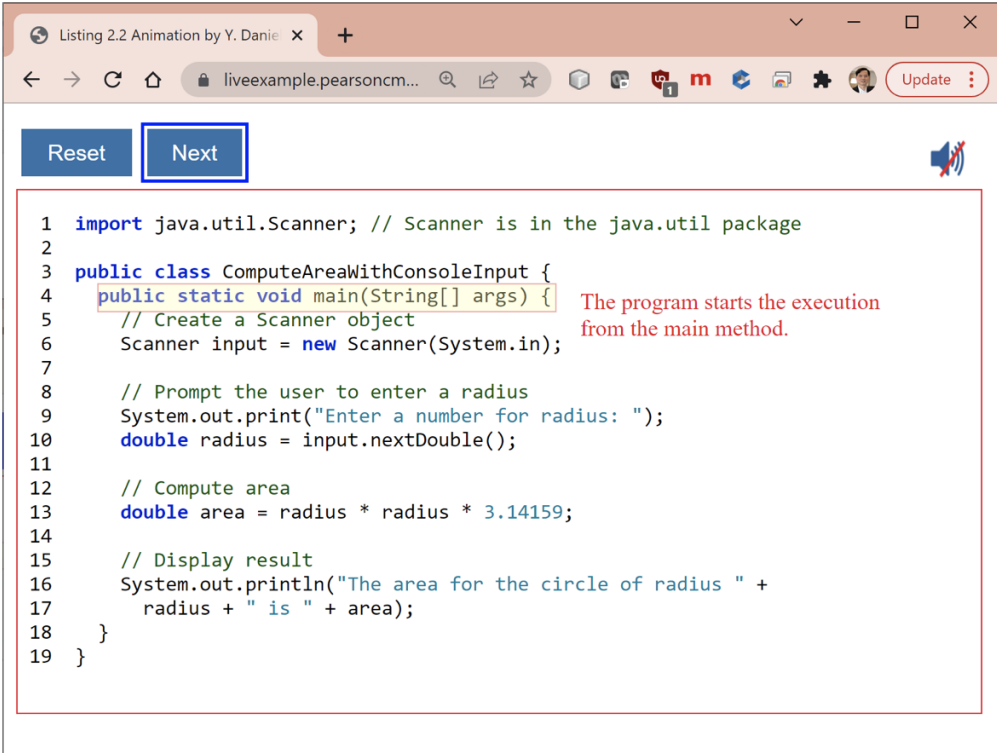
3.3 APPLYING THE GENERATOR MODEL TO OTHER PROBLEMS

Let us look at a similar problem and see how we apply our generator model to generate a web page for this example. Consider the Code Animation problem. In this example, the goal is to process source code and generate an HTML file for animating the source code. We will call this HTML file a Code Animation.

Figure 3.2 gives an example of a Code Animation. Code Animation simulates the execution of a program with step-by-step explanation of the code. Figure 3.2 shows the next step of 3.2. This example of code animation can be accessed from <https://liveexample.pearsoncmg.com/codeanimation/ComputeAreaWithConsoleInput.html>

Our objective is to develop a Code Animation Generator that takes any Java program code and generates an HTML file that animates the execution of the code.

For our generic model of the generator, we use a textarea for code input as shown in Figure 2.17. The generator receives the input and validates whether the program is correct

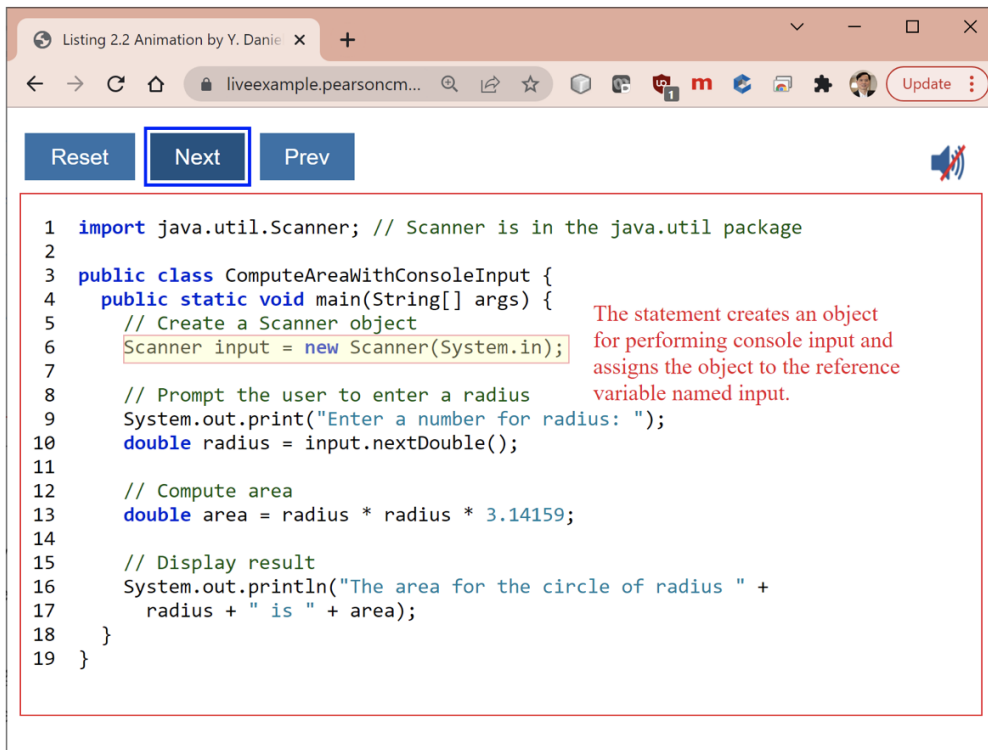


The screenshot shows a web browser window with the address bar containing "liveexample.pearsoncm...". The page has a "Reset" button and a "Next" button. A red box highlights the following Java code:

```
1 import java.util.Scanner; // Scanner is in the java.util package
2
3 public class ComputeAreaWithConsoleInput {
4     public static void main(String[] args) {
5         // Create a Scanner object
6         Scanner input = new Scanner(System.in);
7
8         // Prompt the user to enter a radius
9         System.out.print("Enter a number for radius: ");
10        double radius = input.nextDouble();
11
12        // Compute area
13        double area = radius * radius * 3.14159;
14
15        // Display result
16        System.out.println("The area for the circle of radius " +
17            radius + " is " + area);
18    }
19 }
```

A red annotation next to line 4 states: "The program starts the execution from the main method."

Figure 3.2: Explanation For Line 4 of Compute Area With Console Input.



The screenshot shows a web browser window with the address bar containing "liveexample.pearsoncm...". The page has three navigation buttons: "Reset", "Next" (highlighted with a blue border), and "Prev". A red box highlights a section of Java code. Line 6 of the code is highlighted in yellow and has a red annotation next to it. The code is as follows:

```
1 import java.util.Scanner; // Scanner is in the java.util package
2
3 public class ComputeAreaWithConsoleInput {
4     public static void main(String[] args) {
5         // Create a Scanner object
6         Scanner input = new Scanner(System.in);
7
8         // Prompt the user to enter a radius
9         System.out.print("Enter a number for radius: ");
10        double radius = input.nextDouble();
11
12        // Compute area
13        double area = radius * radius * 3.14159;
14
15        // Display result
16        System.out.println("The area for the circle of radius " +
17            radius + " is " + area);
18    }
19 }
```

The red annotation next to line 6 reads: "The statement creates an object for performing console input and assigns the object to the reference variable named input."

Figure 3.3: Explanation For Line 6 of Compute Area With Console Input.

in syntax. This can be done by invoking a Java compiler. If the program has compile errors, the generator will display the compile error and ask the user to fix the error and resubmit. In the Process Data module, we will build a set of code structural patterns with a template and have the Code Animation Generator to search for code pattern and use the template to generate the animation. We will develop a code analyzer to identify the code structural pattern and search a template for the pattern. We will also build a database of statement patterns with explanations for the statements. In the Generate Web Page module, we will create a page with the buttons “Reset,” “Next,” and “Previous” associate each statement in the code with an appropriate explanation. The explanation will be obtained from the search of the similar statement in the database. The Post Web module will post the page to the website. The Code Animation Generator is not implemented. The implementation of the Code Animation Generator is not part of this thesis. We implemented the Automatic Word Match Generator. It is our hope that the Automatic Word Match Generator will serve as the base for implementing the future web page generating projects such as the Code Animation Generator. As with many software research projects, creating new types of software is a big part of the problem. The devil is in the details. We have a complete implementation of the Automatic Word Match Generator. Although there are other programs that are capable of generating word-matching games, our Automatic Word Match Generator is amongst a few programs that generate simple *word-matching interactives*. The simplicity of our *interactives* distinguish them from word-matching games commonly found on the internet. The Automatic Word Match Generator is freely available. It does not require a subscription, nor does it feature distracting ads.

CHAPTER 4

CONCLUSION

We proposed a generic model for automatically generating web pages. The Automatic Word Match Generator is a demonstration of a concrete implementation for this generic model. We believe that many other web page generation projects can be implemented using similar approaches. Our Automatic Word Match Generator project serves as a stepping stone in the field of automatic programming for generating web pages.

An Automatic Word Match Generator removes the pain that instructors typically face when they have to create word-matching games. Before implementing the Automatic Word Match Generator, all of the *word-matching interactives* had to be developed manually. The process of manually creating exercises was a large waste of time for instructors. To create each exercise, an individual file with specific content had to be created, updated, and maintained. This is where automatic programming comes into play. We employed an automatic programming strategy to develop an Automatic Word Match Generator to remedy the difficulties associated with this process. The automated process of creating *word-matching interactives* saves instructors valuable time and effort that could be spent on more important tasks.

Our tool provides instructors with the ability to create *word-matching interactives* without having to write any code. The first iteration of our tool required instructors to at least copy and paste their code from the textarea onto the server. The manual effort required to copy and paste resulted in poor adoption, so we added a Post button to save the generated HTML code. Once the content is saved onto the server, a URL is created for the instructor to access the exercise directly without any extra work. In retrospect, we should have created this tool earlier to save hundreds of hours of writing *word-matching interactives* manually.

Our tool addresses the need for computer science instructors to be able to teach new vocabulary from anywhere in the world. Its primary purpose serves to help students im-

prove their computer science vocabulary through *word-matching interactives*. The contribution from our research is a web-based tool that can automatically generate a *word-matching interactives*. Now, instructors can enter their terms and descriptions to create fun *word-matching interactives* which can be shared with students by sending them a URL. The tool is freely available from <http://livelab.georgiasouthern.edu/wordmatchgenerator>. It enables the instructors to guide the students in need, leading to a better learning experience. The effectiveness of our application is supported by the research shown in (Arifah and Kusumarasyati 2013), (Dewi 2014), (Manik and Christiani 2016), (Masri and Najjar 2014), Ria Dhatun and (Nikmah Husein 2010).

CHAPTER 5

FUTURE WORK

At present, the generated exercises are not associated with a user. We plan to let instructors create accounts so they can create and store exercises in a database. An instructor will be able to view all created exercises and delete them as well. With a user account, the keys and their descriptions for each exercise will be saved in the database and regenerated. The instructor will not need to re-enter the keys terms and descriptions if new functionality or a new user interface is added to the generated HTML file.

We have iterated on the generic programming model that we used for the Automatic Word Match Generator to create another teaching tool called Automatic Word Flip Generator. This tool is still in the early stages of development. It will provide students an opportunity to improve their reading comprehension skills by flipping cards that display a question and answer. An example of the interface for a *word-flip interactive* is shown in Figure 5.1.

Once a question and answer is loaded into the system, a category is generated. Each of these categories contains a set of questions and answers. An instructor can create a category and give the URL to a student to practice their vocabulary. An example exercise is shown in Figure 5.2 and Figure 5.3.

Another direction of the future work is to create multiple *word-matching interactives* at once. This idea was proposed by an instructor. If an instructor wishes to create an XML file that stores information for multiple exercises, as long as that file specifies the title, key terms, and their descriptions, the Automatic Word Match Generator will take the information from the XML file and automatically generate a *word-matching interactive* for each exercise specified in the XML file.

In Section 3.3, we proposed to apply the generic model for web page generation to the

Add Question

Category

Question
Answer

Save

Figure 5.1: Adding a Question

Add Question

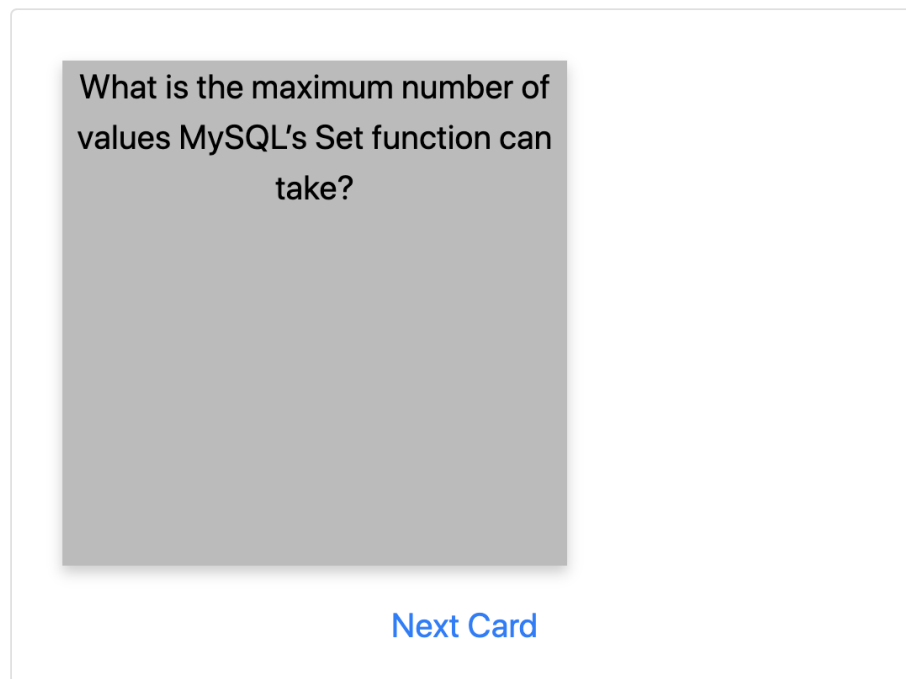


Figure 5.2: Before Flipping a Card

Add Question

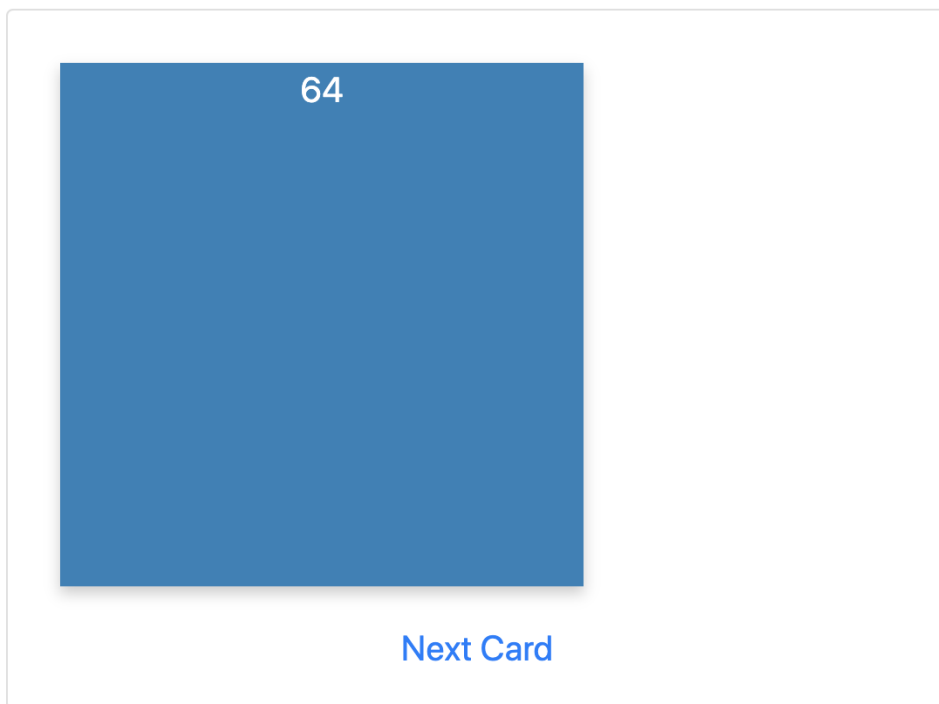


Figure 5.3: After Flipping a Card

Code Animation problem. In the future, we will work on the implementation of the Code Animation Generator.

- Adams, D., Sumintono, B., Mohamed, A., & Noor, N. S. M. (2018). E-learning readiness among students of diverse backgrounds in a leading malaysian higher education institution. *Malaysian Journal of Learning and Instruction*, *15*(2), 227–256.
- Agocs, M. M., Burns, M. S., De Ley, L. E., Miller, S. L., & Calhoun, B. M. (2006). Fast forward language. *Treatment of language disorders in children*, 471–508.
- Arcuri, A., & Yao, X. (2014). Co-evolutionary automatic programming for software development. *Information Sciences*, *259*, 412–432.
- Arifah, M., & Kusumarasyati. (2013). The effectiveness of make a match technique for teaching writing descriptive text to the seventh graders of smpn 1 karang binangun lamongan. *UNESA*, *1*(1), 1–8.
- Azzi-Huck, K., & Shmis, T. (2020). Managing the impact of covid-19 on education systems around the world: How countries are preparing, coping, and planning for recovery. *World Bank Blogs*, *18*.
- Badge, J. L., Dawson, E., Cann, A. J., & Scott, J. (2008). Assessing the accessibility of online learning. *Innovations in Education and Teaching International*, *45*(2), 103–113.
- Balzer, R. (1985a). A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering*, (11), 1257–1268.
- Balzer, R. (1985b). A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering*, (11), 1257–1268.
- Cooney, C. (Fall 2015). Revel educator study assesses quiz, exam, and final course grades at central michigan university. <http://www.pearsoned.com/results/revel-educator-study-assesses-quiz-exam-final-course-grades-central-michigan-university>.

- Cooney, C. (Spring 2016). Revel™ educator study observes homework and exam grades at university of louisiana. <http://www.pearsoned.com/results/revel-educator-study-observes-homework-exam-grades-university-louisiana/>.
- Dewi, M. (2014). The impact of the application of a match technique towards students' vocabulary mastery. *The Second International Conference on Education and Language (2nd ICEL) 2014 Bandar Lampung University (UBL), Indonesia ISSN 2303-1417*.
- Drosos, I., Barik, T., Guo, P. J., DeLine, R., & Gulwani, S. (2020). Wrex: A unified programming-by-example interaction for synthesizing readable code for data scientists. *Proceedings of the 2020 CHI conference on human factors in computing systems*, 1–12.
- Hake, R. R. (1998). Interactive-engagement versus traditional methods: A six-thousand-student survey of mechanics test data for introductory physics courses. *American journal of Physics*, 66(1), 64–74.
- Jazayeri, M. (1976). Formal specification and automatic programming. *Proceedings of the 2nd international conference on Software engineering*, 293–296.
- Kansal, A. K., Gautam, J., Chintalapudi, N., Jain, S., & Battineni, G. (2021). Google trend analysis and paradigm shift of online education platforms during the covid-19 pandemic. *Infectious Disease Reports*, 13(2), 418–428.
- Liang, Y. D. (2018). Revel™ for introduction to python programming and data structures 2e.
- Liang, Y. D. (2020). Revel™ for introduction to java programming and data structures 12e.
- Liang, Y. D. (2021). Revel™ for introduction to c++ programming and data structures 5e.
- Luxton-Riley Andrew, W. B., & Tyne, C. (2018). Intelligent tutoring systems for programming education: A systematic review. *ACE Annual Convention & Exhibition*, 10(3), 11.

- Manik, S., & Christiani, M. (2016). Teaching vocabulary using matching word on computer assisted language learning. *International Journal of English Language Teaching*, 4(7), 1–26.
- Masri, A. A., & Najjar, M. A. (2014). The effectiveness of using word games on primary stage students achievement in english language vocabulary in jordan. *American International Journal of Contemporary Research*, 4(9), 22.
- Ngolah, C. F., & Wang, Y. (2004). Exploring java code generation based on formal specifications in rtpa. *Canadian Conference on Electrical and Computer Engineering 2004 (IEEE Cat. No. 04CH37513)*, 3, 1533–1536.
- Olsson, R. (1995). Inductive functional programming using incremental program transformation. *Artificial intelligence*, 74(1), 55–81.
- Palshikar, G. K. (2001). Applying formal specifications to real-world software development. *IEEE Software*, 18(6), 89–97.
- Ria Dhatun Nikmah, B. G., & Husein, R. (2010). The effectiveness of make a match technique in teaching vocabulary. *ACM Transactions on Computing Education*, 10(3), 22.
- Sarah Cohen, W. N., & Sagic, Y. (2007). Deciding equivalances among conjunctive aggregate queries. 54(2). <https://doi.org/10.1145/1219092.1219093>
- Shahzad, A., Hassan, R., Aremu, A. Y., Hussain, A., & Lodhi, R. N. (2021). Effects of covid-19 in e-learning on higher education institution students: The group comparison between male and female. *Quality & quantity*, 55(3), 805–826.
- Tallal, P., Merzenich, M. M., Miller, S., & Jenkins, W. (1998). Language learning impairments: Integrating basic science, technology, and remediation. *Experimental Brain Research*, 123(1), 210–219.

- Wang, C., Feng, Y., Bodik, R., Dillig, I., Cheung, A., & Ko, A. J. (2021). Falx: Synthesis-powered visualization authoring. *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, 1–15.
- Whalen, M. W., & Heimdahl, M. P. E. (1999). An approach to automatic code generation for safety-critical systems. *14th IEEE International Conference on Automated Software Engineering*, 315–318.
- Yang, W., & Dai, W. (2011). Rote memorization of vocabulary and vocabulary development. *English Language Teaching*, 4(4), 61–64.
- YooN, S. S. (1990). A translator description language tdl for specification languages. *Journal of information processing*, 3(3).
- Young, E. (2005). The language of science, the language of students: Bridging the gap with engaged learning vocabulary strategies. *Science Activities*, 42(2), 12–17.

APPENDIX A: USER'S MANUAL

The first step will be installing git. Please visit <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git> for instructions on how to install the software on the your machine.

Then you will need to install maven. Please visit <https://maven.apache.org> for instructions on how to install maven.

Word Match Generator can be downloaded and installed by executing the following steps:

1. visit the github page for the source code at <https://github.com/EvanGertis/Selected-Topics/tree/master>. Then run:
2. `git clone https://github.com/EvanGertis/Selected-Topics.git`
3. `cd Selected-Topics`

This will pull down the source code for the application.

4. To clean and build maven project, use:

```
mvn clean install
```

5. To create and run the Spring boot application, run the following code:

```
mvn spring-boot:run
```


This will start the web server.

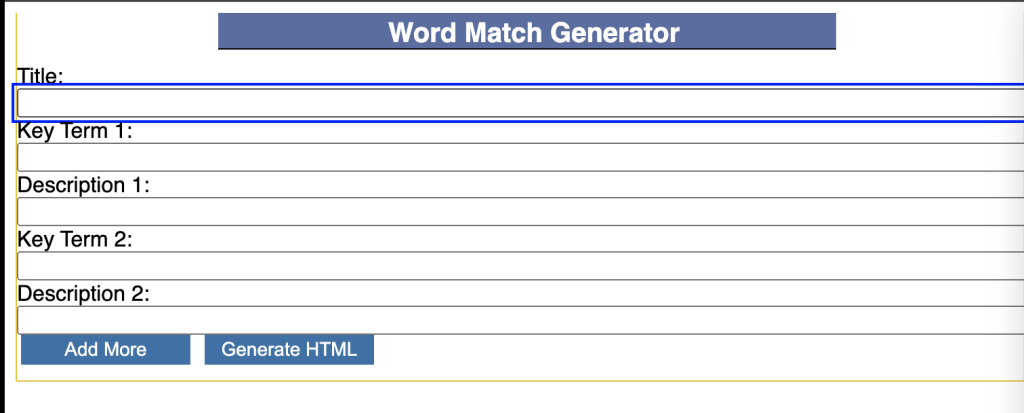
6. open the browser to

`http://localhost/wordmatchgenerator`

The purpose of this section is to describe how to use Word Match Generator. Start the application by running the command:

```
mvn spring-boot:run
```

Go to the wordmatchgenerator endpoint.



The screenshot shows a web application titled "Word Match Generator". The interface includes several input fields: "Title:", "Key Term 1:", "Description 1:", "Key Term 2:", and "Description 2:". At the bottom, there are two buttons: "Add More" and "Generate HTML".

Figure A.1: Word Match Generator Without Input Data

Then create a set of key terms and definitions.

Then click "generate HTML" button.

Then click the View button as shown below:

Then share this url with the student.

Word Match Generator	
Title:	Title
Key Term 1:	Infinite Loop
Description 1:	is a loop that runs forever due to an error in the code.
Key Term 2:	off-by-one
Description 2:	is an error in the program that causes the loop body to be executed one more or less time.
<input type="button" value="Add More"/> <input type="button" value="Generate HTML"/>	

Figure A.2: Word Match Generator With Input Data

Title:	Title
Key Term 1:	Infinite loop
Description 1:	a loop that runs forever due to an error in the code.
Key Term 2:	off-by one
Description 2:	an error in the program that causes the loop body to be executed one more or less
<input type="button" value="Add More"/> <input type="button" value="Generate HTML"/>	
<pre> DOCTYPE HTML> html lang="en"> <head> <title>Word Matching Exercise</title> <style> focus {outline: 2px solid blue; outline-offset: 2px;} details {padding:3px;} </style> <link rel="stylesheet" type="text/css" ref="\$ {pageContext.request.contextPath}/static/css/boxes.css" </pre>	
<input type="button" value="Save"/> <input type="button" value="view"/>	

Figure A.3: Word Match Generator After Generating HTML

Title

Infinite Loop off-by-one

is a loop that runs forever due to an error in the code.

is an error in the program that causes the loop body to be executed one more or less time.


Show Answer 

Figure A.4: Word Match Generator Before Dragging Boxes

APPENDIX B: MAINTENANCE MANUAL

The modules used in the Automatic Word Match Generator are listed below:

1. **reset()**: Used to initialize the drag and drop boxes used in the `init()` method.
2. **init()**: used to initialize the variables needed to run the word matching interactive drag and drop algorithm.
3. **show_answer()**: Used to display the correct answer for a word matching interactive.
4. **generate_html()**: Processes the HTML inputs from the instructor UI into a javascript string.
5. **save_content()**: Sends the request body to the server via XMLHttpRequest to create a word matching interactive. Retrieves the response from the server to display the view button that is linked to the id associated with the created word matching interactive.
6. **add_more()**: Used to create more input boxes in the instructor UI.
7. **render_html()**: Used to originally display the processed HTML from the javascript string.
8. **drag()**: Provides a dragging functionality in a word matching interactive.
9. **drop()**: Updates the word matching interactive after a user drags the key term to a description.
10. **handDrop()**: Used to verify whether or not the correct key term matches the description.
11. **speak()**: Provides audio narration of the program.
12. **add_logging()**: Sets up application logging for javascript.

13. **clickMessage()**: Provides logging for javascript.
14. **saveWordMatchJSP()**: Processes the POST request that contains the request body for generating a word matching interactive.
15. **getWordMatch()**: Returns the view associated with a word matching interactive.
16. **getWordMatch()**: Returns the view associated with a word matching interactive.
17. **saveJSP()**: Generates a word matching interactive jsp file. Uses the java.io library to create files, the java.util library for decoding the base64 encoded HTML string, log4j library to provide application logging, and the java.util library to interact with file streams.
18. **JSPtoHTML()**: converts the generated jsp file within the saveJSP method to a static HTML file.

APPENDIX C: DESIGN DOCUMENTS

The specifications for a Word Match Generator consisted of the following criteria:

1. Instructors must be able to automatically create interactive word matching games as shown in Figure 1.5.
2. Students should be able to access the games online.

From the perspective of the developer the specifications for the Automatic Word Match Generator are the following:

1. Enable the storage and retrieval of *word-matching interactives*.
2. Capture the input data necessary to create *word-matching interactives*
3. Provide a web interface for displaying *word-matching interactives*
4. Develop a drag and drop algorithm that facilitates an interactive learning experience using the derived data from the output.

APPENDIX D: SOURCE CODE

```
package com.company.app;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}

package com.frugalis.Spring.Boot.Resources;

import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.
    ResourceHandlerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class ResourceConfigs implements WebMvcConfigurer
{
    private static final String[] CLASSPATH_RESOURCE_LOCATIONS =
    {
        "classpath:/htmlFiles/",
        "classpath:/static/",
    }
}
```

```
        "classpath:/static/images"
    };

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry)
    {
        registry.addResourceHandler("/**")
            .addResourceLocations(CLASSPATH_RESOURCE_LOCATIONS)
            .setCachePeriod(3000);
    }
}

package com.company.app.service;

/*
 * WordMatchService.java
 * Author: Evan Gertis
 */
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.company.app.model.WordMatch;
import java.io.File;
import java.io.IOException; // Import the IOException class to handle
    errors
import java.io.FileWriter; // Import the FileWriter class
import java.io.IOException; // Import the IOException class to handle
```



```
errors

import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.stream.Stream;
import java.nio.file.Files;
import java.util.Base64;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;

@Service

public class WordMatchService {

    private static final Logger logger = LogManager.getLogger(
        WordMatchService.class);

    @Autowired
    WordMatchService(){
    }

    public Long saveJSP(WordMatch wordMatch){
        logger.info(wordMatch);
        Long numberOfFiles = (long) 0;
        try {
            File file = new File("./src/main/webapp/view/
                word_match0.jsp");
            logger.info("Decoding String");
            String cleanedHTML = wordMatch.toString().replace("

```

```
        WordMatch(content=",".replace(",",""));
logger.info(cleanedHTML);
byte[] decodedBytes = Base64.getDecoder().decode(
    cleanedHTML.getBytes());
String html = new String(decodedBytes, "UTF-8");
logger.info(html);
if (file.createNewFile()) {
    System.out.println("File created: " + file.
        getName());
    try {
        FileWriter myWriter = new FileWriter
            ("./src/main/webapp/view/
            word_match0.jsp");
        myWriter.write(html);
        myWriter.close();
    } catch (IOException e) {
        System.out.println("An error occurred
            .");
        e.printStackTrace();
    }
} else {
    String fileName = file.getName().toString();
    String index = fileName.substring(fileName.
        indexOf("h") + 1);
    index = index.substring(0, index.indexOf("."))
        ;
    Integer parsedInt = Integer.parseInt(index);
    System.out.println(parsedInt);
}
```

```
Stream<Path> files = Files.list(Paths.get("./
    src/main/webapp/view/"));
numberOfFiles = files.map(Path.class::cast)
    .filter(path -> path.
        getFileName().
        toString().
        startsWith("
            word_match"))
    .count();

fileName = fileName.replace(index,
    numberOfFiles.toString());
System.out.println(numberOfFiles);
System.out.println("fileName should have been
    printed by now");
file = new File(fileName);
String JSPfileName = "./src/main/webapp/view
    /"+file;
FileWriter myWriter = new FileWriter(
    JSPfileName);
myWriter.write(html);
myWriter.close();

// Write JSP file to HTML
// File path is passed as parameter
File jspFile = new File(JSPfileName);

// Note: Double backquote is to avoid compiler
// interpret words
// like \test as \t (ie. as a escape sequence)
```

```
// Creating an object of BufferedReader class
BufferedReader br
= new BufferedReader(new FileReader(jspFile));

// Declaring a string variable
String st;

// Condition holds true till
// there is character in a string
String htmlFileName = JSPfileName.replace("jsp
    ", "html");
File htmlFile = new File(htmlFileName);
String content = "Writing To File";
if (!htmlFile.exists()) {
    htmlFile.createNewFile();
}
try {
    FileWriter fw = new FileWriter(htmlFile
        .getAbsolutePath());
    BufferedWriter bw = new BufferedWriter(
        fw);
    while ((st = br.readLine()) != null) {
        System.out.println(st);
        bw.write(st);
    }
    bw.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

```

        }
        System.out.println("Done");
        // try
        // {
        //     JSPtoHTML(JSPfileName);
        // }
        // catch (IOException e) {
        //     System.out.println("An error occurred
        //         .");
        //     e.printStackTrace();
        // }
    }
} catch (IOException e) {
    System.out.println("An error occurred.");
    e.printStackTrace();
}
return numberOfFiles;
}

public void JSPtoHTML(String fileNameForJSP) throws Exception {
    // File path is passed as parameter
    File file = new File(fileNameForJSP);

    // Note: Double backquote is to avoid compiler
    // interpret words
    // like \test as \t (ie. as a escape sequence)

    // Creating an object of BufferedReader class

```

```
BufferedReader br
= new BufferedReader(new FileReader(file));

// Declaring a string variable
String st;
// Condition holds true till
// there is character in a string
String htmlFileName = fileNameForJSP.replace("jsp","html");
File htmlFile = new File(htmlFileName);
String content = "Writing To File";
if (!htmlFile.exists()) {
    htmlFile.createNewFile();
}
try {
    FileWriter fw = new FileWriter(htmlFile.
        getAbsolutePath());
    BufferedWriter bw = new BufferedWriter(fw);
    while ((st = br.readLine()) != null) {
        System.out.println(st);
        bw.write(st);
    }
    bw.close();
} catch (IOException e) {
    e.printStackTrace();
}
System.out.println("Done");
}
```

```
public Long saveHTML(WordMatch wordMatch){
    logger.info(wordMatch);
    Long numberOfFiles = (long) 0;
    try {
        File file = new File("./src/main/webapp/view/
            word_match0.html");
        logger.info("Decoding String");
        String cleanedHTML = wordMatch.toString().replace("
            WordMatch(content=", "").replace(", ", "");
        logger.info(cleanedHTML);
        byte[] decodedBytes = Base64.getDecoder().decode(
            cleanedHTML.getBytes());
        String html = new String(decodedBytes, "UTF-8");
        logger.info(html);
        if (file.createNewFile()) {
            System.out.println("File created: " + file.
                getName());
            try {
                FileWriter myWriter = new FileWriter
                    ("./src/main/webapp/view/
                        word_match0.html");
                myWriter.write(html);
                myWriter.close();
            } catch (IOException e) {
                System.out.println("An error occurred
                    .");
                e.printStackTrace();
            }
        }
    }
}
```

```
} else {  
    String fileName = file.getName().toString();  
    String index = fileName.substring(fileName.  
        indexOf("h") + 1);  
    index = index.substring(0, index.indexOf("."))  
        ;  
    Integer parsedInt = Integer.parseInt(index);  
    System.out.println(parsedInt);  
    Stream<Path> files = Files.list(Paths.get("./  
        src/main/webapp/view/"));  
    numberOfFiles = files.map(Path.class::cast)  
        .filter(path -> path.  
            getFileName().  
            toString().  
            startsWith("word_match"))  
        .count();  
    fileName = fileName.replace(index,  
        numberOfFiles.toString());  
    System.out.println(numberOfFiles);  
    System.out.println("fileName should have been  
        printed by now");  
    file = new File(fileName);  
    FileWriter myWriter = new FileWriter("./src/  
        main/webapp/view/"+file);  
    myWriter.write(html);  
    myWriter.close();  
}
```



```
        } catch (IOException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
        return numberOfFiles;
    }
}

package com.company.app.model;

/*
 * WordMatch.java
 * Author: Evan Gertis
 */

import lombok.Data;

@Data
public class View {

    public Long id;

    public void setId(Long Id) {
        this.id = Id;
    }
}

package com.company.app.model;
```

```
/*
 * WordMatch.java
 * Author: Evan Gertis
 */

import lombok.Data;

@Data
public class WordMatch {

    public String content;
}

package com.company.app.controller;

//WordMatchController.java
//Author: Evan Gertis 10/11/2021

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.PathVariable;
```

```
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;

import com.company.app.model.View;
import com.company.app.model.WordMatch;
import com.company.app.service.WordMatchService;

@Controller
public class WordMatchController {

    private static final Logger logger = LogManager.getLogger(
        WordMatchController.class);

    private final WordMatchService wordMatchService;

    @Autowired
    public WordMatchController(WordMatchService wordMatchService) {

        logger.info("visiting word match");

        this.wordMatchService = wordMatchService;
    }

    @PostMapping("/wordmatchgenerator")
    public ResponseEntity<View> saveWordMatchJSP(@RequestBody WordMatch
        wordMatch) {

        logger.info("Processing word match from client");

        logger.info(wordMatch);

        Long Id = wordMatchService.saveJSP(wordMatch);

        View view = new View();

        logger.info("New view created with id {}",Id);
    }
}
```

```

logger.info("View object before {}",view);
view.setId(Id);
logger.info("View object after {}",view);
        return new ResponseEntity<View>(view, HttpStatus.CREATED);
}

@PostMapping("/wordmatchgeneratorXML")
public ResponseEntity<HttpStatus> saveWordMatchXML(@RequestBody
        WordMatch wordMatch) {
        logger.info("Processing word match from client");
        logger.info(wordMatch);
        // Long Id = wordMatchService.saveHTML(wordMatch);
// View view = new View();
// logger.info("New view created with id {}",Id);
// logger.info("View object before {}",view);
// view.setId(Id);
// logger.info("View object after {}",view);
        return new ResponseEntity<HttpStatus>(HttpStatus.OK);
}

@RequestMapping("/wordmatchgenerator")
public String getWordMatch(Model model) {
        return "word_match";
}

@RequestMapping("/wordmatchgenerator/{id}")
public String getWordMatch(@PathVariable String id ,Model model) {
        return "word_match"+id;
}

```

```

    }

}

<%@ page contentType="text/html;charset=UTF-8" language="java" %>

<!DOCTYPE HTML>
<html lang="en">
  <head>
    <title>Word Match Generator</title>
    <style>
      *:focus {outline: 2px solid blue; outline-offset: 2px;}
      details {padding:3px;}
    </style>
    <link rel="stylesheet" type="text/css" href="${pageContext.request.
      contextPath}/static/css/boxes.css" />
    <link rel="stylesheet" type="text/css" href="${pageContext.request.
      contextPath}/static/css/style.css" />
    <script type="text/javascript" src="${pageContext.request.contextPath}/
      static/js/event1.js"></script>

<!-- Global Site Tag (gtag.js) - Google Analytics -->
<script async src="https://www.googletagmanager.com/gtag/js?id=UA
  -89940905-27"></script>
<script>
  window.dataLayer = window.dataLayer || [];
  function gtag(){dataLayer.push(arguments)};
  gtag('js', new Date());

```

```
    gtag('config', 'UA-89940905-27');
</script>

<!-- <script type="text/javascript" src="../../logging.js"></script> -->
</head>

<body>
  <div id="maincontentstyle">
    <div id="boxstyle">
      <h3 id="title">Word Match Generator</h3>
      <div>
        <div id="inputs">
          <div id="inputBoxes">
            <title>Input:</title>
            <div>
              Title: <input id="title_input" type="text">
            </div>

            <div>
              Key Term 1: <input id="e11" type="text" value="">
            </div>
            <div>
              Description 1: <input id="d11" type="text" value="">
            </div>
            <div>
              Key Term 2: <input id="e12" type="text" value="">
            </div>
          </div>
        </div>
      </div>
    </div>
  </div>
</body>
```

```

        <div>
            Description 2: <input id="dl2" type="text" value="">
        </div>
    </div>
    <span style="padding: 3px">
        <button id ="add_more" class="button" type="button"
            onClick="add_more()">Add More</button>
    </span>
    <span style="padding: 3px">
        <button id ="one" class="button" type="button" onClick="
            generate_html()">Generate Html</button>
    </span>
    </div>
</div>
</div>
</div>
<div id="results" class="row">
</div>
<div id="renderedHTML" class="row">
</div>
</div>

```

```

<script src="{pageContext.request.contextPath}/static/js/jquery-1.7.2.min.
    js"></script>

```

```

<script src="{pageContext.request.contextPath}/static/js/jquery-ui.min.js
    "></script>

```

```
<script src="{pageContext.request.contextPath}/static/js/jquery.ui.touch-
  punch.min.js"></script><script src="{pageContext.request.contextPath}/
  static/js/jquery.alerts.js"></script><link href="{pageContext.request.
  contextPath}/static/js/jquery.alerts.css" rel="stylesheet" type="text/
  css" media="screen" />
```

```
<script type="text/javascript">
$(init);
$( window ).unload(function() {
  removeStorage.removeItem("someVarKey1");
});
function init() {
  document.getElementById('resetButton').style.display = 'none';
document.getElementById("resetButton").style.visibility = "hidden";
if (false && sessionStorage.getItem("someVarKey1")) // No focus for the
  first time
  $("#one").focus();
  var numbers = [3, 4, 5, 1, 2];
  initialize(numbers);
}
</script>
```

```
<script type="text/javascript" src="{pageContext.request.contextPath}/
  static/js/word_match.js"></script>
<script type="text/javascript" src="{pageContext.request.contextPath}/
  static/js/GetElementPosition3.js"></script>
```



```
<script>

audioOn = false;

$(function() {
    $('.menulink').click(function(){
        if (audioOn) {
            $("#bg").attr('src',"${pageContext.request.contextPath}/static/images/
                audioOff.png");
            audioOn = false;
        }
        else {
            $("#bg").attr('src',"${pageContext.request.contextPath}/static/images/
                audioOn.png");
            audioOn = true; speak(" ");
        }
        return false;
    });
});

</script>

</body>

</html>

<script src="//cdnjs.cloudflare.com/ajax/libs/highlight.js/10.7.1/highlight
    .min.js"></script>

// window.load = main()
```

```
// function main(){
    // initially html is not generated.
    var htmlGenerated = false;
    // number of inputs start out as 2.
    var numberOfInputs = 2;
    // initially no additional inputs have been added.
    var addMore = false;
    // initialize the answer.
    var answer = '';

    // saved variables
    var saved = false
    var saved_id = 0

    function reset() {
        // reset the htmGenerated to false.
        htmlGenerated = false;
        numberOfInputs = 2;
        var someVarName = true;
        sessionStorage.setItem("someVarKey1", someVarName);
        window.location.reload();
    }

    function populate_numbers_array(footer, dlArray){
        console.log("populating numbers")
        dlArray.forEach(i => {
            footer += i.replace ( /^[^d.]/g, '' );
            footer += ',';
        });
    }
}
```

```
        console.log("adding "+i+" to the numbers array")
    })
    return footer
}

function show_answer() {
    jAlert(answer, 'Correct Match');
}

function generate_html() {

    // retrieve the keys and descriptions. Then load them into their
    // respective arrays.

    const e_inputs = document.querySelectorAll("[id^='el']");
    const d_inputs = document.querySelectorAll("[id^='dl']");
    let elArray = [];
    let dlArray = [];

    const title = document.getElementById('title_input').value;
    e_inputs.forEach( i => { if(i.value) elArray.push(i.value) });
    d_inputs.forEach( i => { if(i.value) dlArray.push(i.value) });

    //has the html already been generated?
    if(!htmlGenerated){

        //fetch the results box
        results = document.getElementById("results");

        //create textarea
```

```

textarea = document.createElement("textarea");
textarea.setAttribute("id", "generated_html_textarea");

// initialize blank html
header = '<!DOCTYPE HTML>\n<html lang=\n"en"\n>\n<head>\n\t\t<title>
Word Matching Exercise</title>\n\t\t<style>\n*:focus {outline: 2
px solid blue; outline-offset: 2px;} \n\t\t<details {padding:3px
;} \n\t\t</style>\n\t\t<link rel=\n"stylesheet" type=\n"text/css"
href=\n"${pageContext.request.contextPath}/static/css/boxes.css
" />\n\t\t<script type=\n"text/javascript" src=\n"${pageContext.
request.contextPath}/static/js/event1.js"\n>';
header += "<link rel=\n"stylesheet" type=\n"text/css" href=\n"${
pageContext.request.contextPath}/static/css/style.css" />"
header += '</\n'
header += 'script>\n'
header += '<script async src=\n"https://www.googletagmanager.com/gtag
/js?id=UA-89940905-27"\n>'
// header += '</\n'
// header += 'script>\n<script>\n\t window.dataLayer = window.
dataLayer || [];\n\t function gtag(){dataLayer.push(arguments)
};\n\t gtag(\n"js", new Date());\n\t gtag(\n"config", \n"UA
-89940905-27");\n'
header += '</\n'
header += 'script>\n'
header += '<script src=\n"${pageContext.request.contextPath}/static/js
/jquery-1.7.2.min.js"\n>'
header += '</\n'
header += 'script>\n'

```

```

header += '<script src="\${pageContext.request.contextPath}/static/js
    /jquery-ui.min.js">'
header += '</'
header += 'script>\n'
header += '<script src="\${pageContext.request.contextPath}/static/js
    /jquery.ui.touch-punch.min.js">'
header += '</'
header += 'script>\n'
header += '<script src="\${pageContext.request.contextPath}/static/js
    /event1.js">'
header += '</'
header += 'script>\n'
header += '<script src="\${pageContext.request.contextPath}/static/js
    /jquery.alerts.js">'
header += '</'
header += 'script>\n'
header += '<link href="\${pageContext.request.contextPath}/static/js/
    jquery.alerts.css" rel="stylesheet" type="text/css" media="
    screen">'
header += '<script type="\text/javascript\" src="\${pageContext.
    request.contextPath}/static/js/logging.js\">'
header += '</'
header += 'script>\n</head>\n\t\t<body>';
let html = '';
html += header;
html += '<div id=\'maincontentstyle\'>\n'
html += '\t<center>\n'
html += '\t\t<div id=\'boxstyle\'>\n'

```



```

html += '\t\t\t\t\t</center>\n'

//create description inputs
html += '\t\t\t\t\t<table id=\'tablestyle\'>\n'
for (let i = numberOfInputs; i < dlArray.length+numberOfInputs; i++)
{
    html +='\t\t\t\t\t\t\t<tr>\n'
    html += '\t\t\t\t\t\t\t<td id=\'row';
    id  = i-numberOfInputs+1;//dlArray[i-numberOfInputs].replace (
        /^[^d.]/g, '' );
    console.log("id "+id);
    html += id;
    html +='\>\n';
    html += '\t\t\t\t\t\t\t\t\t\t\t<div id=\'t';
    html += id;
    html +='\>' class=\'ltarget ui-droppable\'>'
    html +='\</div>\n'
    html +='\t\t\t\t\t\t\t\t\t\t\t</td >\n'
    html +='\t\t\t\t\t\t\t\t\t\t\t<td id=\'d'
    html += id
    html += '\>\n'
    html += '\t\t\t\t\t\t\t\t\t\t\t';
    html += dlArray[i-numberOfInputs];
    html += '\n';
    html +='\t\t\t\t\t\t\t\t\t\t\t</td >\n'
    html +='\t\t\t\t\t\t\t\t\t\t\t</tr>\n';
}
html += '\t\t\t\t\t\t\t</table>\n';

```

```

html += '\t\t\t\t</center>\n'
html += '\t\t</div>\n'
html += '\t</center>\n'
html += '</div>'
html += '<span style="padding: 3px"> <button id ="one" class="
        button" type="button" onClick="show_answer'
html += '()'
html += ''
html += ">"
html += 'Show Answer'
html += '</'
html += 'button> <button id = "resetButton" class="button" type
        ="button" onClick="reset'
html += '()'
html += ''
html += '>'
html += 'Reset'
html += '</'
html += 'button>'
html += '</span>'
html += '<span id="audio" style="">'
html += '<a href="" title="Turns Text-to-Speech Output On or Off"
        class="menulink" style="text-decoration: none;">'
html += ''
html += '</a>'
html += '</span>'

```



```

footer = '\n\t\t</body>\n</html>\n';
footer += ''
footer += '<script type="text/javascript">'
footer += '$(init);'
footer += '$( window ).unload(function() {'
footer += 'removeStorage.removeItem("someVarKey1");'
footer += '});'
footer += 'function reset() {'
footer += ' var someVarName = true;'
footer += 'sessionStorage.setItem("someVarKey1", someVarName);'
footer += 'window.location.reload();'
footer += '}'
footer += 'function init() {'
footer += '    document.getElementById(\'resetButton\').style.
        display = \'none\';'
footer += 'document.getElementById("resetButton").style.visibility =
        "hidden";'
footer += 'if (false && sessionStorage.getItem("someVarKey1"))'
footer += '$("#one").focus();'
console.log('var numbers = [');
footer += 'var numbers = ['
for (let i = numberOfInputs; i < dlArray.length+numberOfInputs; i++)
    {
        footer += dlArray[i-numberOfInputs].replace ( /[^\d.]/g, '' );
        console.log(dlArray[i-numberOfInputs].replace ( /[^\d.]/g, '' ))
        footer += ',';
        console.log(',');
    }

```

```
console.log('];')
footer += '];'
footer += 'initialize(numbers);'
footer += '}'
footer += '</script>'
footer += ' <script>'
footer += ' answer = '
footer += '\n'
answer = '';
for (let i = numberOfInputs; i < dlArray.length+numberOfInputs; i++)
    {
        answer += elArray[i-numberOfInputs];
        answer += ':';
        answer += dlArray[i-numberOfInputs];
        answer += ' '
    }
footer += answer
console.log(answer)
footer += '\n'
footer += ';'
// footer += '\n'
// footer += ' Iteration: is one time execution of the loop body.'
// footer += '\n'
// footer += 'Loop Continuation Condition: is a Boolean expression
        that controls the execution of the loop.'
// footer += '\n'
// footer += 'Infinite Loop: is a loop that runs forever due to an
        error in the code.'
```

```

// footer += '\n'
// footer += 'Off-by-one: is an error in the program that causes the
    loop body to be executed one more or less time.'"
footer += ' function show_answer() {'
footer += '     jAlert(answer, \'Correct Match\');'
footer += ' }'
footer += '</script>'
footer += ' '
footer += '<script type="text/javascript" src="{pageContext.request
    .contextPath}/static/js/GetElementPosition3.js"></script>'
footer += ' <script>'
footer += '     $(function(){'
footer += '     if (\'speechSynthesis\' in window) {'
footer += '         speechSynthesis.onvoiceschanged = function() {'
footer += '             var $voicelist = $('#voices\');'
footer += ' '
footer += '             if($voicelist.find(\'option\').length == 0) {'
footer += '                 speechSynthesis.getVoices().forEach(function(voice,
                    index) {'
footer += '                     var $option = $('\'<option>\')'
footer += '                         .val(index)'
footer += '                         .html(voice.name + (voice.default ? \' (default)
                    \' : \'\'));'
footer += '                     $voicelist.append($option);'
footer += '                 }'
footer += '             }'
footer += ' '
footer += '             $voicelist.form_select();'
footer += '         }'

```

```

footer += '  }'
footer += ' } '
footer += '});  '
footer += 'audioOn = false;'
footer += '$(function() {'
footer += '$(\'.menulink\').click(function(){'
footer += ' if (audioOn) {'
footer += '   $("#bg").attr(\'src\', "${pageContext.request.
        contextPath}/static/images/audioOff.png"); '
footer += '   audioOn = false;'
footer += ' }'
footer += ' else {'
footer += '   $("#bg").attr(\'src\', "${pageContext.request.
        contextPath}/static/images/audioOn.png");'
footer += '   audioOn = true; speak(" ");'
footer += ' }'
footer += ' return false;'
footer += '});'
footer += '});'
footer += ' </script> '
html += footer;

// html generation is done.
htmlGenerated = true;
textarea.value = html;
results.replaceChildren(textarea);

// Generate reset, show answer, , and render html buttons

```

```

controls = document.createElement("div");
controls.setAttribute("id","program1");
controls.setAttribute("style","border: 1px solid #EB0D1B; width: 450
    px; font-family: courier; font-size: 100.5%; margin: 0px auto;
    border: 1px; text-align: center; margin-top: 5px;");
controls.innerHTML += '<button id = "renderHTMLButton" class="button
    " type="button" onClick="render_html()">Render html</button>\n';
controls.innerHTML += '<button id = "submit" class="button" type="
    button" onClick="saveContent()"> Save </button>\n';
controls.innerHTML += '<button id=\\"view_button\\" class=\\"button\\"
    style=\\" display: none;\\"><a href=\\"${window.location.href}/${
    saved_id}\\"> view</a> </button>\n';
if(document.getElementById("renderHTMLButton"))
    results.parentNode.replaceChild(controls);
results.parentNode.appendChild(controls);
}
}

```

```

function saveContent(){
    console.log("calling save content");
    var html_content = document.getElementById("generated_html_textarea
        ");
    var b64_string = btoa(html_content.value)
    console.log(b64_string)
    var xhr = new XMLHttpRequest();
    xhr.open("POST", "/wordmatchgenerator", true);
    xhr.setRequestHeader("Content-Type", "application/json;charset=UTF

```

```

-8");
xhr.onreadystatechange = function()
{
    if(xhr.readyState == 4 && xhr.status == 201) {
        console.log(xhr.status)
        console.log("content saved");
        saved = true;
        view_button = document.getElementById("view_button");
        view_button.style.display = "inline";
        console.log('JSON.parse(xhr.response).id ' + JSON.parse(xhr.
            response).id)
        saved_id = JSON.parse(xhr.response).id
        console.log('saved_id ' +saved_id)
        view_button.children[0].href = '${window.location.href}/${
            saved_id}'
    }
    else{
        console.log(xhr.status)
        console.log(xhr.response)
        console.log("content was not save successfully");
    }
}
console.log('{"content":\'
    +b64_string+\'\'});
xhr.send(JSON.stringify({content: b64_string}));
}

function add_more() {

```

```
// we've added more inputs.
addMore = true;

// set html generated to false, because new inputs have been added.
htmlGenerated = false;

// increment the number of inputs.
numberOfInputs++;

//fetch the input boxes.
inputs = document.getElementById("inputBoxes");

// create newline
br = document.createElement("br");

//create a new row for a key term.
row = document.createElement("div");

// set the key term text.
row.innerHTML = "Key Term ";
row.innerHTML +=numberOfInputs;
row.innerHTML += " :";

// create the input for the key.
key = document.createElement("input");
key.setAttribute("id","e1"+numberOfInputs);

//add the key to the row.
```

```
row.appendChild(key);
row.after(br);

//create a row for the new description.
row2 = document.createElement("div");

// set the description text.
row2.innerHTML = "Description "
row2.innerHTML+=numberOfInputs;
row2.innerHTML+=" :";
row2.after(br);

// create the description input
description = document.createElement("input");
description.setAttribute("id","dl"+numberOfInputs);

// add the description to the row.
row2.appendChild(description);

// add the rows for the key and the description to the inputBoxes.
inputs.appendChild(row);
inputs.appendChild(row2);
}

function render_html(){

textarea = document.getElementById("generated_html_textarea");
// Set the generate html to the value from the textarea.
```



```

generated_html = textarea.value;
console.log(generated_html);
// Create a new tab.
var new_window = window.open('');
maincontentstyle = document.getElementById("maincontentstyle");
if(document.getElementById("rendered_html"))
    document.getElementById("rendered_html").remove();

rendered_html = document.createElement("div");
rendered_html.setAttribute("id","rendered_html");
rendered_html.setAttribute("style","border: 1px solid #EB0D1B; width:
    450px; font-family: courier; font-size: 100.5%; margin: 0px auto;
    border: 1px; text-align: center; margin-top: 5px;");
rendered_html.innerHTML += generated_html;
results = document.getElementById("results");

if(document.getElementById("rendered_html"))
    results.parentNode.appendChild(rendered_html);

// Append the rendered html to the results tab
results.parentNode.appendChild(rendered_html);
header = '<!DOCTYPE HTML>\n<html lang="en">\n\t<head>\n\t\t<title>
    Word Matching Exercise</title>\n\t\t<style>\n*:focus {outline: 2px
    solid blue; outline-offset: 2px;} \n\t\t<details {padding:3px;} \n\t\t
</style>\n\t\t<link rel="stylesheet" type="text/css" href="
    static/css/boxes.css" />\n\t\t<script type="text/javascript" src
    ="static/js/event1.js">';
header += '</>'

```

```

header += 'script>\n'
header += '<script async src=\"https://www.googletagmanager.com/gtag/js
    ?id=UA-89940905-27\">'
header += '</'
header += 'script>\n'
// header += '<script>\n\t window.dataLayer = window.dataLayer || [];\n
    \t function gtag(){dataLayer.push(arguments)};\t gtag(\"js\", new
    Date());\t gtag(\"config\", \"UA-89940905-27\");\n'
// header += '</'
// header += 'script>\n'
header += '</head>\n\t\t<body>';
new_tab_html = header;
new_tab_html += rendered_html.innerHTML;
footer = '\n\t\t</body>\n</html>\n';
footer += '<script type=\"text/javascript\" src=\"static/js/
    GetElementPosition3.js\">'
footer += '</'
footer += 'script>'
footer += '<script type=\"text/javascript\" src=\"static/js/word_match.
    js\">'
footer += '<script src=\"static/js/jquery-1.7.2.min.js\">'
footer += '</'
footer += 'script>\n'
footer += '<script src=\"static/js/jquery-ui.min.js\">'
footer += '</'
footer += 'script>\n'
footer += '<script src=\"static/js/jquery.ui.touch-punch.min.js\">'
footer += '</'

```

```

footer += 'script>\n'
footer += '<script src="static/js/event1.js">'
footer += '</'
footer += 'script>\n'
footer += '<script src="static/js/jquery.alerts.js">'
footer += '</'
footer += 'script>\n'
footer += '<link href="static/js/jquery.alerts.css" rel="stylesheet"
      type="text/css" media="screen">'
footer += '<script type="text/javascript" src="static/js/logging.js
      ">'
footer += '</'
footer += 'script>\n'
new_tab_html += footer;
console.log(new_tab_html);
new_window.document.write(new_tab_html);
}

rand = Math.random();
function shuffleDescriptions(a){
  for(let j,i=a.length;i>1;){
    j=Math.floor(rand*i--);
    if (i!=j) [a[i],a[j]]=a[j],a[i]]
  }
  console.log("shuffled dlarray")
  console.log(a)
  return a
}

```

```
function shuffleKeys(a){
  for(let j,i=a.length;i>1;){
    j=Math.floor(rand*i--);
    if (i!=j) [a[i],a[j]]=a[j],a[i]]
  }
  console.log("shuffled elarray")
  console.log(a)
  return a
}
// }
```

APPENDIX E: TEST SUITE

We can ensure that the Automatic Word Match Generator meets the specifications listed in the requirements specification by completing the following tests:

1. Visit the instructor UI.
2. Enter the following title: Section 1.2 Word Matching Exercise.
3. Enter the following key terms and descriptions
 - (a) CPU: is a computers brain. It retrieves instructions from memory and executes then.
 - (b) Memory: stored data and program instructions for the CPU to execute. It is volatile, because information is lost when the power is turned off.
 - (c) storage device: stores programs and data permanently.
 - (d) hardware: is the physical aspect of the computer that can be touched.
 - (e) software: are the invisible instructions that control hardware and make it perform tasks.
 - (f) Click generate HTML.
 - (g) Click Post.
 - (h) Click View.
 - (i) Drag CPU to is a computers brain. It retrieves instructions from memory and executes then.
 - (j) Memory to stored data and program instructions for the CPU to execute. It is volatile, because information is lost when the power is turned off.
 - (k) Storage device to stores programs and data permanently.

(l) Hardware to is the physical aspect of the computer that can be touched.

(m) Software to are the invisible instructions that control hardware and make it perform tasks.

1. Repeat steps 1-6 from Test 1.
2. Drag CPU to stores program and data permanently.
3. Ensure that CPU dragging box returns to original position.

1. Repeat steps 1-6 from Test 1.
2. Toggle the audio box.
3. Repeat step 7 from Test 1.
4. Ensure that audio functionality works.

1. Repeat steps 1-6 from Test 1.
2. Repeat step 7 from Test 1 except for the last key term.
3. Click the reset button.
4. Ensure that the key terms goes back to their original positions.