CodeBash Design Manual
CSCI 205 - Fall 2022 - team05

Introduction – *A general technical overview of CodeBash*

Our system, CodeBash, walks the user through 4 different scenes. First, an Introduction Scene (Fig. 1) describing the reasons why the user might be interested in playing the game. Second, we have a Welcome Scene (Fig. 2) that prompts the user to choose settings for game play and then start the game. Settings include timers (15, 30, 45, or 60 seconds) color mode (dark, light, or purple), and language (english, java, or python) Thirdly, we have the Game Play Scene (Fig. 3) that displays a timer and text that the user types, along with an option to quit. When the timer gets to zero or the user presses quit, they are brought to a Results Scene (Fig. 4) that displays their results, words per minute, accuracy, and errors. This scene also asks the user to play again or end the game. When "PLAY AGAIN" is clicked, the user is brought back to the Welcome Scene (Fig. 2). From there, the user can pick different settings or just continue playing.

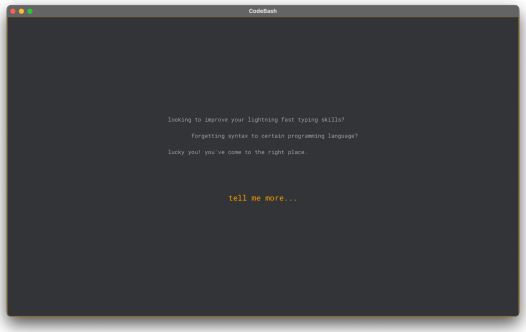

Fig. 1 - The Introduction Scene

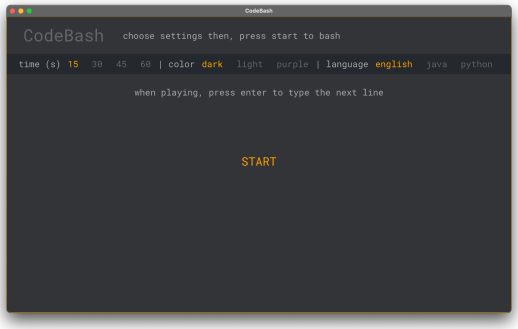

Fig. 2 - The Welcome Scene

Fig. 3 - The Game Play Scene

Fig. 4 - The Results Scene

More technically, our CodeBashMain file handles instantiating these scenes and handling instances where the scene changes by buttons or by the end of a timer. Though we would have preferred having all of the event handling in respective controllers, we found that having these specific event handlers in the main method proved to be the most simple implementation.

We designed 2 of our 4 user interfaces, the Welcome Scene and the Game Play Scene, using a Model View Controller design pattern. Otherwise, we contained our Introduction Scene and Results Scene in one class each. This approach allowed us to easily communicate and distribute responsibilities between classes. We used the JavaFX library to display these scenes and add functionality to our buttons. For user input, we handled clicks of buttons and key presses. We used 3 css files (CodeBashDark.css, CodeBashLight.css, and CodeBashPurple.css) in order to style our displays and different color modes. To generate our text for the java and python language mode, we used the text files java.txt and python.txt, which contain short lines of code for the user to practice typing.

User Stories - *Descriptions and their inspirations*

For our user stories, we created three people to help us envision how our system would benefit others, all with different skills and backgrounds. The first person we had was Clara Walker. Clara Walker was a college student who was majoring in computer science. She has some experience with computer science from high school, but is still considered to be a beginner/intermediate in her computer science classes. It is also important to note that Clara is incredibly sensitive to bright lights and would rather use a system that had a darker tone. Our system is friendly to those with more sensitive eyes; the settings start out in dark mode and those who do not like the default settings are free to change it. Clara is quoted to say “I'm hoping to improve how fast I can code and my familiarity with the programming languages. I'm learning a bunch of new languages around the same time, so I'm struggling to differentiate between languages with similar syntax.” Her needs are met in the system of CodeBash, as instead of having just sentences to type, we have two other different options for the coding language: Java and Python. The more popular syntax of Java and Python are lines that can be typed, which would help Clara with getting more familiar with the coding languages and learn to differentiate between the two.

The second person is someone who has somewhat more experience than Clara Walker, and her name is Deniz Babacan. Deniz is a brand new software developer who has just graduated from her college and is currently job searching. She believes that if

she were able to type faster and more efficiently, it would benefit her greatly once she finds a job. She has once said "I am looking to maintain my proficiency in coding languages and staying up to date," making her a perfect candidate for our system. We have two different coding languages that are mainstream and well known throughout the world to help practice coding with: Java and Python. As a new software developer, she would be able to code much faster in these languages and improve her efficiency in coding. It is also important for her to keep up with these coding languages due to them being incredibly common.

Our final user is someone by the name of Elif Tunaboylu. She is a high school student who is interested in learning how to code and improve her typing speed and accuracy. She spends a lot of time in front of screens doing her assignments and her eyes get tired easily. She likes apps that have the ability to change the text size and that have bright interfaces. Unfortunately, the text size, which was initially part of our plan to be able to change, had not been finished; however, she would like how our program comes with an option to change the color of the interface to a white background. It is good that she would like to improve her typing speed as well, because our program allows for the user to test and see how fast they can type through english sentences. The program also gives back comments to the typing speed to help her improve her speed. The program will not teach Elif how to code, but it will help her understand the syntax and improve her typing skills for programming.

OOD - *A clear, detailed explanation about our object-oriented design*

When we began ideating, we searched the web for other examples of using JavaFx to create a typing game. We were curious as to how we could get a JavaFx program to handle a user typing without a text area and how to interact with text already displayed. We used Alamas Baimagambetov's JavaFX Typing Game video on YouTube (https://www.youtube.com/watch?v=1If6xa4hM1Q) to gain inspiration for our project. He showed how to compare the user's key press to a text object presented on a screen. We copied his code and implemented it to better understand this implementation. For instance, in his example, when the user typed a letter correctly, the letter disappeared as shown in Fig. 5 and Fig. 6.

Fig. 5 - Before the user types Fig. 6 - After the user types correctly

Later, we attempted to convert that code to a Model View Controller design pattern with little luck. So, using some of the tools and techniques from Baimagamebtov's video, we started from scratch with a new UML diagram and organization of our code, adding a plethora of features and functionality to this original design.

Our original UML Diagram and CRC cards are pictured in Figures 7 & 8. As shown, they intentionally had little detail because we decided what we needed as we began to implement our ideas through code. But, right away we knew we needed a main CodeBash file, a class to represent settings, a class to evaluate user input, and a class that would generate random sentences (Fig. 7). We also thought that having an enumeration class for game states would be useful in determining when the scene should change, such as when a timer ends or when the user wants to play a new game. Many of these were features we kept in our final implementation (Fig. 11).

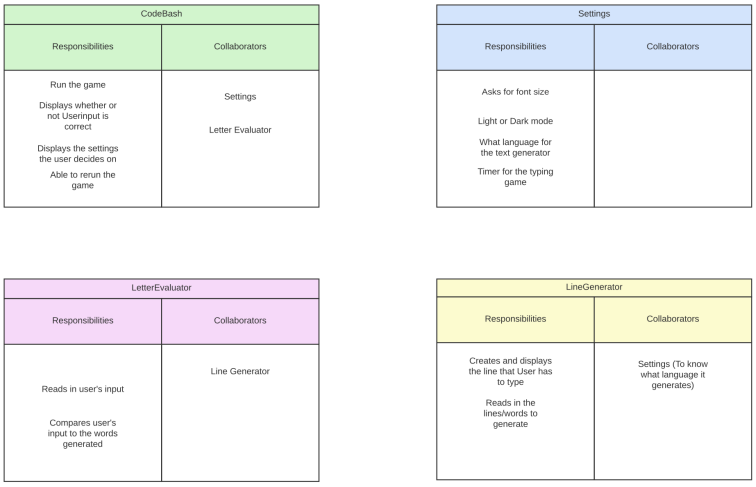

Fig. 7 - Original CRC Cards

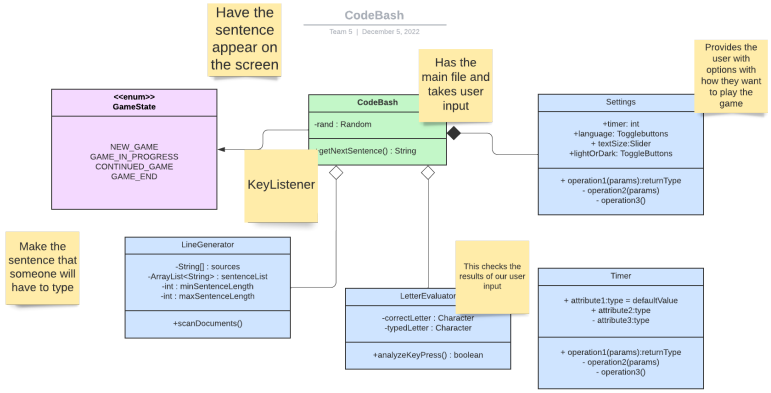

Fig. 8 - Original UML Diagram

As we continued developing CodeBash, we stuck with an MVC design pattern, because it allowed us to add more functionality and features to the game. We heavily improved our original CRC Cards (Figures 9 & 10) and UML Diagram (Fig. 11). Though our UML diagram only covers the model, we thought that was most appropriate given how complicated the IntelliJ diagram was. We created a model that handled sentence generation (LineGenerator), had multiple views that formatted the different scenes in the game (CodeBashView & CodeBashWelcomeView), and lastly a controller (CodeBashController) to handle the user input and button presses. CodeBashMain instantiated all the different view objects from the different scenes along with model objects that were needed for sentence and timer calculations.

CodeBashMain	
Responsibilities	**Collaborators**
Run the game	CodeBashModel
Displays whether or not Userinput is correct	CodeBashView
Displays the settings the user decides on	CodeBashWelcome
Able to rerun the game	CodeBashResults
	CodeBashWelcome
	CodeBashController
	CodeBashIntro

Fig. 9 - CodeBashMain CRC

CodeBashModel	
Responsibilities	**Collaborators**
Generate a new sentance	LineGenerator
Set the Default mode to English and handle generating other (coding) languages	GameState
	TypingStats
Sets the default mode to dark mode	GameMode
	ColorSate

CodeBashView	
Responsibilities	**Collaborators**
Handle the main text display for gameplay	CodeBashModel
Instatiate a model object then call a method getCurrentSentece()	
Displays sentence char by char as text objects on the screen	

CodeBashController	
Responsibilities	**Collaborators**
Handles the keys pressed on screen, turns them to appropriate colors based on accuracy	CodeBashModel
	CodeBashView
Calls for new sentence to be generated	LetterEvaluator
	ColorState

Fig. 10 - CodeBash MVC CRC Cards

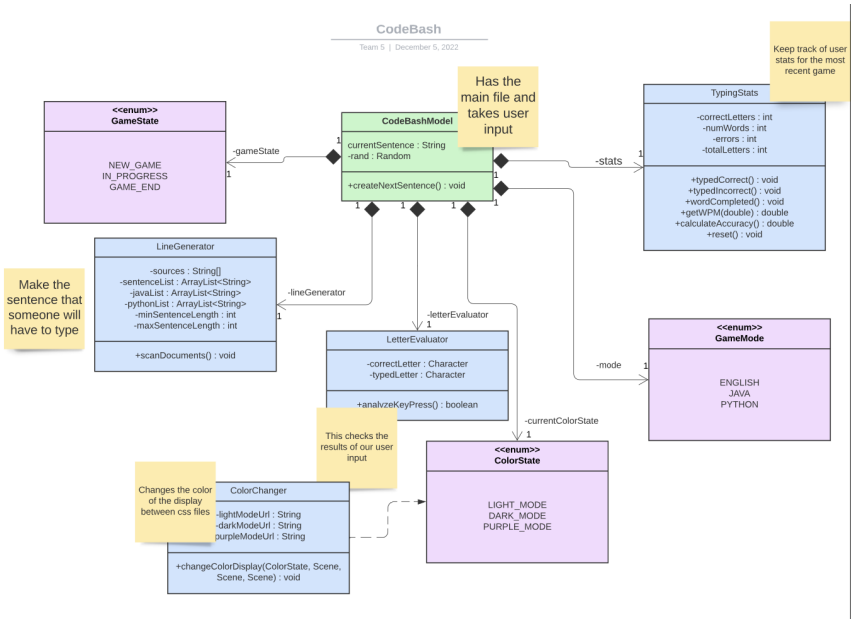

Fig. 11 - Current High-Level UML Diagram from LucidChart

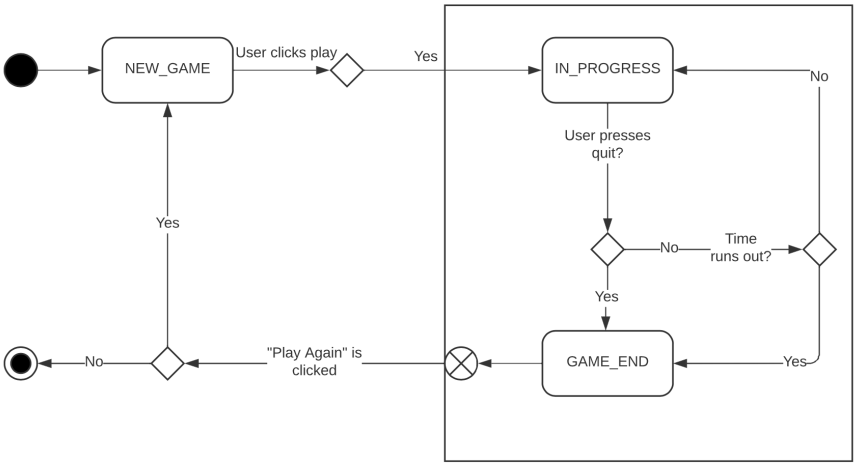

Fig. 12 - UML Diagram for our GameState Enumeration Class

To highlight the most important aspects of our design, we made a UML diagram for our Model of our CodeBash MVC (Fig. 11). The most important aspect to note is the central class highlighted in green, “CodeBashModel”. From the start, we wanted there to be one class that combined all the functionality of the others without being too convoluted itself. There are three main functional classes that branch from this: LineGenerator to create the sentences (or code snippets) that will be printed to the screen, LetterEvaluator to check each input from the user to see if it’s correct, and

TypingStats to keep track of how the user is performing in order to give them a summary at the results portion of the game.

We used the IntelliJ generated UML diagram to show our MVC approach. Since the classes were so intertwined, the whole IntelliJ UML was overly complicated and didn't make sense to include. So, instead we included Fig. 13 that shows the Main file that is run when you play the game and the MVC for the Game Play scene. Fig. 13 shows the nodes created with IntelliJ in CodeBashView, such as quitBtn (Button) or showTimer (HBox). It also shows the event handlers (initEventHandlers and initBindings) in CodeBashController and the skeleton of our game in CodeBashModel that creates important features for our game, like sentences for the user to type and setting the color mode for the whole game.

The primary enumeration that most of our game logic depended on was GameState (depicted in Fig. 12), which was planned from the start. This provided crucial information to the rest of the model, including what GUI to display (is the user playing?) and when to forcefully stop the game (based on the timer). Two additional enums were developed partway into the process, those being GameMode and ColorState. These were tasked with keeping track of the selected language and color theme, respectively. All of these parts work together to provide the underlying functionality of our base game.

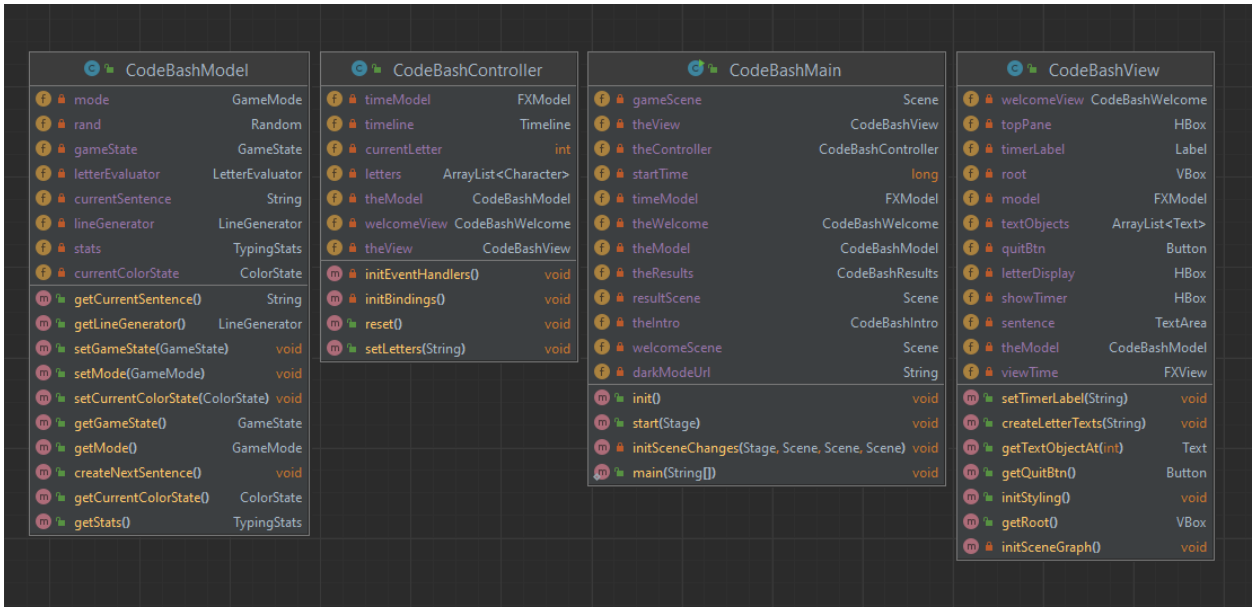

Fig. 13 - IntelliJ-Generated Cards for the Core Classes