Evil HANGMAN

An Exercise in Software Engineering

Author: Bryan Wolfford

Designer: Bryan Wolfford

Developer: Bryan Wolfford

Table of Contents

Contents

Initiation

In order to illustrate my thought processes and approach to software engineering projects, I developed an enhanced version of the classic "Hangman" game, as though I will be presenting it to a client. The project's requirements were provided in a minimally-structured, Microsoft Word® document written in American English named evil-hangman.docx; the contents of which, I will analyze in this and the following chapters.

From evil-hangman.docx, I am able to identify that the player's experience must be that of the regular, single-player version of the classic hangman game, which is as follows:

1. The computer chooses a word then makes it known how many letters are in that word
2. The player guesses a letter that might be in the computer's word
    a. If the letter is in the word, the computer displays each occurrence of that letter.
    b. If the letter is not in the word, then the player loses one of their six initial "chance allowance".
3. The player continues to guess until (s)he wins by correctly guessing all the letters that comprise the word, or loses by exhausting all six of his/her chances and misses again.

For this advanced, computerized version, however, the following are additional requirements:

- There is a HCI (Human-Computer Interface) that provides feedback regarding the game state, as well as providing the means by which a player may submit a single letter guess.
- The computer must be able to keep an "active set" of words.
- The computer must be able to categorize all the words in the active set into "equivalence classes" given some "evil" heuristic
- The computer must be able to reactively choose which words to keep in that active set, so as to behave in the most evil way.
- Evil is defined as admitting to a letter's position (or non-inclusion) in the "active word" by first calculating which resulting equivalency class will include the highest amount of members.
- The computer must have a way (minimally pseudorandom behavior) for deciding which equivalency class to choose when several have the same number of members.
- A more sophisticated evil heuristic (beyond that of simply choosing the equivalence class with the most members) is welcome but not required.

After carefully reviewing that document, I am able to confidentially assert that given my current operational status, the negligible financial costs, and the receptiveness of the single shareholder, all project requirements are well within the scope of my skill-set as a software engineer. With that being the case, I can both close the initiation phase and move into the planning phase of this project.

Planning

Because I am the sole designer and developer on this project and only the finished product will be delivered to the client, I will ignore the communication stages that are essential to team projects and those which require incremental deliveries at milestones.

For the execution phase, I plan to use an iterative process where I would make several design-develop-test cycles in order to meet several milestones throughout the life of this project.

I also plan to develop "Evil Hangman" as a dynamic, multi-layered web-app on a local Apache web server using HTML, CSS, JavaScript, jQuery, AJAX, PHP, and MySQL, then integrate that web-app with my public website, bryan.WOLFFORD.com[1].

Given the negligible financial costs associated with this development strategy, I will also exclude all budgeting details from my plan.

Taking cues from a popular mobile/web design paradigm, MVC, I have chosen to develop this project in three major parts, as a distinct model, view, and controller; the details of which I elaborate on in the sections that immediately follow this section.

Furthermore, I will carefully assemble a structured list of milestones; goals that will serve as a guide for me as I design and develop the software that, along with this document, comprises the final product.

The Model

I have chosen to utilize a MySQL database to model the game environment. It will only communicate with the controller, and never communicate with the view directly. Because changes will be transparent to the view, this guarantees minimal disruption, transition time, and man hours if I decide to modify or upgrade the model to a different technology, like PostgreSQL. Also, if all modifications still conform to the interface already in use, they will be transparent to the controller as well.

Specifically, in the database I will create a table that contains all valid English words that can be used in the game, coupled with the length of the word (to reduce query-times based on word length), and a metric to determine the word's commonality (to assist in making a more evil algorithm).

I then plan to create a PHP class object that will handle the required functionality of the model, such as, getting the word list and setting game-state variables.

The View

The state of the game will be viewed by the player via an interface developed in HTML and CSS. This view of the game state will only communicate with the controller and never the model directly, thereby, reducing the amount of work required to modify or upgrade the view to a different technology or platform, like the Android mobile platform.

The Controller

The game experience will be created by the interactions controlled by software written in JavaScript and augmented with asynchronous updates made possible with jQuery and AJAX. The controller will communicate directly with the model and the view in a way that should be as transparent to the other components as possible. By transparent, I mean that if I were to change the implementation of the model, even in a radical way, no change should be required of the view, and vice versa.

Milestones

For the execution phase of this project, I plan to iteratively develop individual components of the project by aspiring to conform to the following milestones.

1. A message can be sent through the View-Controller-Model-Controller-View path.
2. The database is populated with words and the controller can send that data to the view.
3. The player can initialize a new game.
4. The view represents the "active word" and updates when a player makes a guess.
5. The controller can generate equivalency classes and choose the most "evil" one.
6. The controller updates the view when a letter is chosen to be included in the active word.
7. The controller accounts for incorrect guesses and synchronizes the view with the game state.
8. The view/controller prevents players from improper inputs (rules and security).
9. The final HCI and game flow has been designed and implemented.

Execution

I will be completing this phase by dividing the project into several engineering cycles that focus on a single milestone or goal.

There are 9 cycles in total. The following are links to navigate to the top of the description for each one:

I will be documenting my thoughts, actions and experiences as I strive to complete each of these goals, so each cycle will consist of at least one of each: design, develop and test sections, and will be formatted exactly in the following way:

Cycle 0

Milestone: The typical formatting of a cycle has been demonstrated.

Design

I will use this formatting to illustrate the details of each cycle.

Figure 0.1

This is a Figure. It can occur in any part of a cycle.

Develop

I created this example to illustrate the formatting that I will use for each cycle.

Test

I viewed this example formatting in “print preview”.

Result

Success. The example conforms to the formatting as expected.

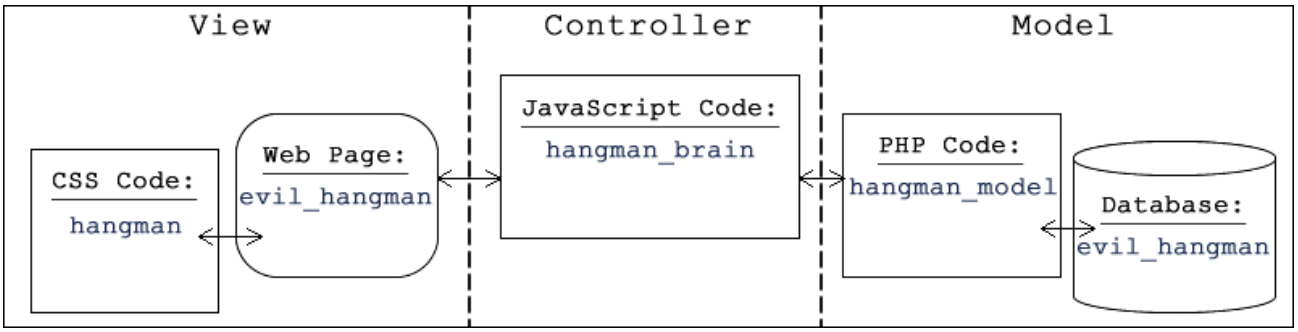
Cycle 1

Milestone: A message can be sent through the View-Controller-Model-Controller-View path.

Design

I have already decided that I will need a model, a view and a controller, so first I will create a stub for each of those parts in separate files and then create the communication channels as necessary. Figure 3.1 below concretely portrays this design concept:

Figure 3.1



Develop

To serve as the model alongside the MySQL database that I plan to create in the next cycle, I created hangman_model.php. Inside that, I also included code to connect to the database and echo back the get-variable, guess, when it exists.

To act as the controller, I created hangman_brain.js, and inside I defined hangman_brain(), a function that will act as a game object.

For the view, I created evil_hangman.php, sourced in the latest version of jQuery and the controller file, hangman_brain.js, and instantiated the hangman_brain() object. Next, I created an input textbox with the id hangman_guess_box and size and maxlength set to 1, and button for submitting guesses with id hangman_button_guess. Finally, I created hangman.css for the style sheet, and put in empty stubs, one for each id that I created in evil_hangman.php.

Test

To make sure that all the files are now connected correctly, I added one additional function to the controller, guess(), which will be called when the player submits a guess via the view. This function sends the GET variable, guess, to the model, which then returns the text in the variable as an HTTP response. The controller takes that response and injects it into a division in the view to confirm that the message has gone full route, view-controller-model-controller-view.

To also test that the CSS is being used, I changed the color of each object in the view using hangman.css.

Result

Success. The page division on the view successfully updates when the player submits a guess. Also, the colors are all different according to the style sheet.

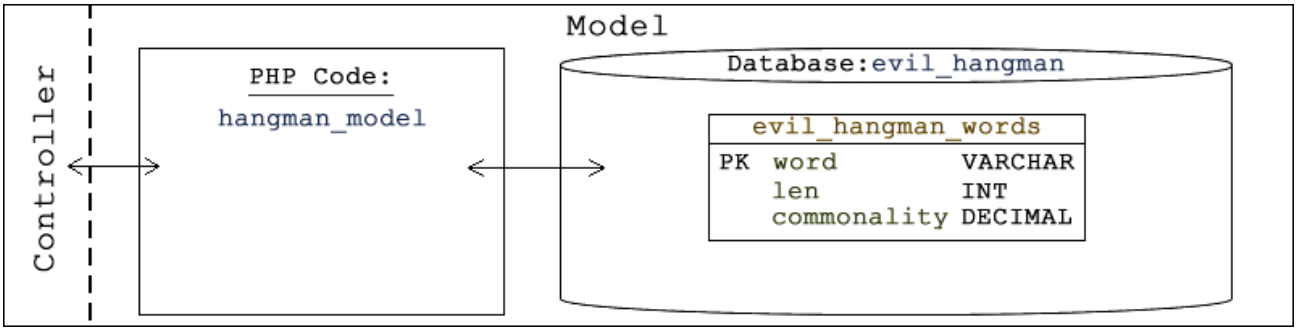
Cycle 2

Milestone: The database is populated with words and the controller can send that data to the view.

Design

I will need to create a MySQL table, evil_hangman_words, which at minimum has three columns: word as a VARCHAR and PRIMARY key, len as an INT, and commonality as a DECIMAL. Then, I will need to acquire a comprehensive list of English words to insert into evil_hangman_words. Once that is done, I must add a method in hangman_model.php to retrieve a subset of those words given their length.

Figure 3.2



Develop

First, I found a fair-use licensed word list on the SIL International website[2]. This is "a list of 109,582 English words compiled and corrected in 1991 from lists obtained from the Interociter bulletin board. This word list includes inflected forms, such as plural nouns and the -s, -ed and -ing forms of verbs."

Then, I created the MySQL tables exactly as I had designed, then used PHP to load the wordlist from the downloaded text file (excluding the contractions like I'm and we've). I also calculated the length of each word, and inserted the entire array of tuples (all commonality defaulted to 1.0) into the MySQL table.

In hangman_model.php, I created the function, get_words_of_length(), which takes in one parameter, the desired word length, then returns a JSON representation of an array of all the words in the database that have a len that matches the desired word length.

In hangman_brain.js, I created a similarly named function, get_words_of_length(), which also takes the desired word length as a parameter, but then passes that value to hangman_model.php via asynchronous HTTP GET requests and returns the JSON array that is sent by the model.

Test

The final database of words has 109,549 entries, where 16,877 of those are seven-letter words. So in evil_hangman.php, I added a call to get_words_of_length() with 7 as a parameter, then checked the length of the resulting array.

Result

Success. The length of the array was 16,877, confirming that the database is online and communicating with the view via the MVC/PHP-MySQL-JavaScript stack.

Cycle 3

Milestone: The player can initialize a new game.

Design

I need the controller to wait for the player to request to initialize the game via the view. That will allow me to easily add initialization parameters later, in the event that the client requests features like "continue previous game" and "make the hangman less evil". This will also allow me to protect all of the other controller methods and members (except guess()) by making them private, while making initialize_game() and guess() privileged functions.

To initialize the game, the controller and model must begin communication to establish and maintain a distinct "active set" of words for each and every game, but as I see it, this creates a dilemma.

According to the MVC paradigm, the controller should not model the game state, but instead rely on the model for all modeling. Theoretically then, the "active set" should be kept on my server and not with the client in the controller or view. However, the internet is mostly stateless (unless I choose to override that feature with sessions, etc), so to keep this MVC design would devour much too much bandwidth. It would send the list of 16,877 words across the network several times in both directions as the controller requests the current state, then modifies the state, then sends updates back to the model for storage.

I want to minimize the number of times I send data across the network. I have many options on how to reduce this transmission, so let me consider two of them.

I could move the controller over to the server. That would mean that I would never have to send the word list across the network, which reduces both bandwidth and ways that people can cheat the game. The problem is that I would have to break the statelessness of the internet, which is easily done by implementing PHP sessions and issuing game id numbers. However, this would mean that my server would be the controller for all of the games being played, which may or may not scale well with the popularity of the game.

Another option would be to bend the MVC requirement a little bit by saving the active word list in the controller on the client side. This would mean that I would only have to send a subset of the full word list across the network once. This passes the burden of all the game logic to the client instead of the server. The problem here is that I would need to secure the communication between the model and the controller so that clever players are not able to easily see the word list. Also, if I ever choose to reuse this model or controller with a different interface, I would have to make careful considerations based on the fact that the MVC paradigm was not used completely.

With all these considerations, I choose to implement the later method which sends the word list once and saves it in the controller.

Develop

In evil_hangman.php, I created a new button with a value 'New Game' that when pressed, calls the controller function, initialize_game().

In hangman_brain.js, I created the function, initialize_game(), which passes the initialization request to the model function, initialize_game(), and then retrieves the returned game state values.

In hangman_model.php, I created the function, initialize_game(), which retrieves a list of all words of a given length by calling the function, get_words_of_length(), then returns a JSON encoded representation of the list of words retrieved, the word length, and value for the "chance allowance".

The new controller function, initialize_game(), is an improvement on get_words_of_length(), making the later obsolete. Therefore, I deleted get_words_of_length() from hangman_brain.js while making it only a private helper function in hangman_model.php.

Test 1

To test this crucial functionality, I modified initialize_game() to log the length of the active words list in the browser console, loaded evil_hangman.php in my browser, and clicked [New Game].

Result

Partial Success. The console confirmed that a new game was generated by logging the length of the "active set"; however, that length was 168,771, which is incorrect by entire order magnitude.

Troubleshooting

By troubleshooting, I was able to discover that the words array that is sent by the model to the controller it is being interpreted as an array of characters, instead of an array of words. This is because get_words_of_length() is already encoding its return value into a JSON object. So when initialize_game() does the same, the words array is encoded twice creating this error.

It is not my intention to have the controller call get_words_of_length() ever again, so I changed the return value of that function to be a regular array, not JSON representation of an array.

Test 2

I refreshed evil_hangman.php in my browser and clicked [New Game] again.

Result

Success. The console confirms that the words list contains 16,877 members, which is as expected.

Cycle 4

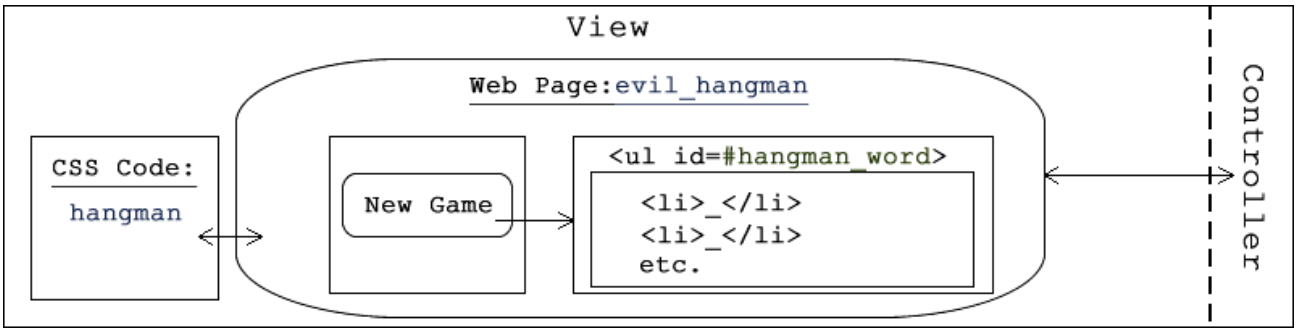
Milestone: The view represents the “active word” and updates when a player makes a guess.

Design

In the view, I wish to visualize the “active word” as an underscore for each letter, unless the computer has admitted that it exists. I still want to postpone designing the user interface until another cycle, but I also do not want any of the work that I do now to be wasted.

I believe that the best solution will be to create an unordered list in HTML to represent the word, with each list item representing a letter. This will allow me to use jQuery to update individual letters based on the UL’s id and an LI index. If I build that UL dynamically based on the word length of the game, then I will have met the fourth milestone.

Figure 3.4



Develop

In evil_hangman.php, I created an un-ordered list to represent the word that the player will try to guess and gave it the id, #hangman_word.

In hangman_brain.js, I created a new function, update_view(), which empties #hangman_word then repopulates it with as many LIs containing underscores as there are letters (given by gameState.wordLength).

Next, I modified guess(), the function that I created in Cycle 1, to now call update_view().

Test

In hangman_brain.js, I added this.counter, a temporary variable initialized as 0. Then, in guess(), I added code to increment that variable every time it is called. Next, I changed update_view() to make a list item out of the number in this.counter instead of just underscores.

Result

Success. All the list items iterate with every click of the guess button. This confirms milestone 3, so I removed the temporary counter variable from the code.

Cycle 5

Milestone: The controller can generate equivalency classes and choose the most "evil" one.

Design

This is the part of the program that makes that gives the game its "evil" reputation. My goal this cycle is to generate the equivalency classes and then select the best class with which to move forward based solely on the number of members in that class.

I will begin by generating a binary representation of a word's equivalency class, having each letter stand for binary digit. So, given the word 'hangman', a guess of 'm' will yield '0000100' or '4', while a guess of 'a' will yield '0100010' or '34'; likewise, given the word 'matches', a guess of 'm' would yield '100000' or '64'.

For algorithmic efficiency, I must avoid iterating over the entire "active set", as well as storing the words in that set, more times than are necessary. To minimalize those procedures, I can use the binary representation of each word as the key for an array of arrays, the values of which being an integer count of how many words fit that pattern. This would, in essence, be a custom implementation of hashing with linked-list collision, but instead of storing a list, I instead only store the length of the list.

Once I have the list of equivalency classes and the counts of their members, I will then identify the class with the most members and then reduce the "active set" to only include the words that belong to that selected equivalency class, thus meeting achieving this cycle's milestone.

Develop

In hangman_brain.js, I created a function, compute_hash() that takes a word and a letter as parameters and returns the binary representation in decimal notation of the positions of that letter in that word.

Next, I created a function named gen_eq_classes() that requires as a parameter the letter guessed by the player, then iterates over all the words in the game state; each time calling compute_hash() with the word and the guessed letter and incrementing by one the value in the array, gameState.eqClass[], at the key equal to the returned computed hash. This function ultimately returns an array of counts of words that are indexed by an integer representation of the class (e.g. '34' for '0100010').

Next, I created the function update_word_list(), which reduces the "active set", represented as the array gameState.words[], to include only the words that when passed to compute_hash(), match the value returned by a call to a new function, select_eq_class(). This reduction is achieved by moving the matched words to the front of the array, then trimming the remainder of the set. Finally, once gameState.words[] has been reduced, update_word_list() then empties gameState.eqClass[].

I then programmed select_eq_class() to compare all the values in gameState.eqClass[], and return the key to the highest value representing the number of members to that equivalency class, and in the case where multiple classes that have tied for most members, I select a class pseudo-randomly.

Finally, I modified guess() to first call gen_eq_classes() to generate gameState.eqClass[], then to call select_eq_class() to select the class that has been decided as the most evil. So, now that the game has committed to that class, guess() calls update_word_list() next to update gameState.words[] so that it is

reduced to only contain members of the selected equivalency class. At the very end, it also calls update_view() to bring the view in sync with the current game state.

Test

1. To first test if the hashing function is working as intended, I picked the seven letter word "hangman", and pretended that the computer only knows that one word. I did this by manually setting gameState.words[] and gameState.wordLength variables instead of calling get_words_of_length() in the model. I logged these results in the console.
2. Then, to test if gen_eq_classes() can process the entire gameState.words[] array, I removed the word limitation and reran the test.
3. To test that update_word_list() is reducing gameState.words[] correctly, I hard-coded selectedHash to always be 0 so that the program will always reduce the "active set" to only include words that do not contain the letter. I then checked size of gameState.words[] as I selected vowels and made sure that the array shrank to the size of the number in gameState.eqClass[0].
4. To ensure that select_eq_class() performs as expected, I guessed a letter via the view, then after an equivalency class is selected and gameState.words[] has been reduced, logged in the console the length of the reduced set along with all the values in gameState.eqClass[] to ensure that it is, indeed, the class with the most members.
5. Finally, to test that equally large equivalency classes are being identified and chosen pseudo-randomly, I hard-coded gameState.words[] to only include the members ['abbbbbb', 'acccccc', 'adddddd', 'eeeeeee','fffffff', 'ggggggg'] so that when I guess 'a', I would then expect to be returned a list that is always 3 in length, but alternates between the hashes 0 and 64.

Result

1. Success. With gameState.words[] limited to "hangman", gen_eq_classes('a') only updates gameState.eqClass[34] to 1, indicating that the hash was correctly computed and gen_eq_classes() is updating gameState.eqClass[] correctly.
2. Success. Without limiting gameState.words[], gen_eq_classes() returns an array with values such as 0: 10,117 and 41: 11, that all sum to 16,877. This indicates that it processes full sets correctly.
3. Success. gameState.words[] does reduce to the size given by the value of gameState.eqClass[0]. Furthermore, the words in the remaining list are absent the letter that I guessed, as expected.
4. Success. When guessing 'a', 'e', then 'i', select_eq_class() returns the hash of the highest member class, 0, 0, then 4. The 4 corresponds to an 'i' occurring in the third to last spot, which includes words like bobbing, which is the class with the most members, so the behavior correct.
5. Success, roughly half the time, hash '64' is chosen, while the other half, it is hash '0'.

Cycle 6

Milestone: The controller updates the view when a letter is chosen to be included in the active word.

Design

The most logical time for the controller to disseminate information to the view, so that it can update its representation of the "active word", would be in guess(), after the most evil equivalency class has been decided by select_eq_class().

I have already developed a function for updating the view, so if I create a new variable that stores the active word, I can simply modify this function to inject whatever letters are in the active word instead of just underscores. The best way to manage that variable will be to initialize it to all underscores when I initialize the game, then update it when a player guesses (a similar model to what I am doing to maintain the "active set").

To update that "active word" with the correct letters, I'll need to be able to decode the hash that was selected after the player makes a guess. I can do that by converting the hash into a logical vector (base10 to base2 conversion). Even though that conversion is very easy to do using the JavaScript function parseInt(), and will most likely only be used to update the view, I will make a new function for decoding the hash, just in case I choose to compute the hashes differently in the future.

Develop

In hangman_brain.js, I created a new function, update_active_word(), which requires two parameters, selectedHash and letter, and calculates what the new "active word" will be by calling a new function decode_hash() with the selectedHash and adding the letter to gameState.activeWord where there are '1's in the resulting logical vector.

Then, I modified initialize_game() to instantiate gameState.activeWord as a string of underscores the same length as value of gameState.wordLength, then call update_view().

I then created decode_hash() to take in one parameter, hash, and return a string, the base2 representation of the hash value, with the intention of using it as a logical array.

Next, I reprogrammed update_view() to rely on gameState.activeWord by iterating over the gameState.wordLength and appending new LIs to the view with each character.

Finally, I updated guess() to call update_active_word() after select_eq_class() but before update_view().

Test 1

To test if this milestone has been met, I will initiate a new game, then guess 'a', 'e', then 'i' and expect the third LI from the right to be populated with an 'i' instead of an '_'.

Result

Partial Success: Upon guessing 'a', 'e', 'i', the first LI is populated with 'i' instead of the third to last. This indicates that update_view() is interpreting the decoded hash correctly, however, gameState.activeWord is not being updated as expected.

Troubleshooting

After troubleshooting, I discovered that a decoded hash was not guaranteed to be the same length as the word because the preceding zeros were being truncated. i.e. a hash of 4 was returning '100'.

To debug this coding error, I have two options; I can modify decode_hash() to return a value that is guaranteed the be same length as gameState.activeWord, or I can modify update_view() to handle hashes of sizes less than or equal to gameState.wordLength. I believe that the best choice to ensure modularity will be to modify decode_hash().

Next, I need to decide how to guarantee the length of the decodedHash string while avoiding concatenating a string in a loop. That is because doing so results in a O(n^2) performance. After researching the issue further, I discovered this blog entry[3] from one of the developers of StackOverflow that provides insights into my concern.

My conclusion is that loops smaller than 100 (our game will typically be only 7) are unnoticeably affected by the O(n^2) behavior of string concatenation, and thus this concern falls under "micro-optimization." Even with that being the case, I should still strive to avoid doing so when it is clearly possibly.

Develop

In hangman_brain.js, I modified decode_hash() to store the decodedHash (which is potentially shorter in length than gameState.activeWord) in a variable. Then, I determine the difference between that original decodedHash and the length of gameState.activeWord and allocate an array with that many '0's. Then, I combine that array into a string, and then concatenate that string as a prefix to the original decodedHash to obtain a string that is guaranteed to be same length as gameState.activeWord.

Test

I reran Test 1.

Result

Success. Upon guessing 'a', 'e', then 'i', the third to last LI is populated with 'i'. This is because the hash '4' was selected, which is because that class has the most members.

Cycle 7

Milestone: The controller accounts for incorrect guesses and synchronizes the view with the game state.

Design

An integral part of this player's experience is the race to discover all the letters of the active word before (s)he exhausts his/her "chance allowance". I have already created a means by which the allowance may be set, retrieved and recorded in the database, so now I must begin accounting for how many incorrect guesses are made and enforcing the end of the game, given the state of that allowance.

I can accomplish this by creating a new page division in the view that will be dedicated to displaying the current number of chances still allowed to the player. In the guess() controller function, I can test for missed guesses as well as losing game states.

To engage different ending states (win, lose, or otherwise), I should create a new function that relies on a single parameter so that sending the proper signal to the player via the view, and resetting the game state variables, is as easy as possible.

Develop

In evil_hangman.php, I created a new page division in the view with the id #hangman_chances to display the current state of gameState.chances.

In hangman.css, I added an entry to #hangman_chances, and colored it red.

In hangman_brain.js, I added code to update_view() to populate #hangman_chances with information stored in gameState.chances.

Next, I modified guess() to check if the selectedHash is 0, which indicates an incorrect guess. When selectedHash is 0, gameState.chances is decremented by 1. Then, guess() checks if the value of gameState.chances is now below 0. If gameState.chances is below 0, the player has lost the game, so guess() then calls a new function, end_game() with its single parameter set to 'outOfChances'.

If the selectedHash is not 0, that indicates that the player's guess was chosen to be included in the "active word". When that is the case, guess() checks if the "active word" no longer contains underscores, which indicates a winning state, so guess() then calls end_game("win"), to display a congratulatory message.

I then created end_game() as a function that requires a single parameter, a value that indicates which signal to send to the view; 'win' for congratulations, a cheeky message for anything else. This function does that by first clearing the game state, then updating #hangman_message.

Test

1. To test that the controller is correctly keeping track of the number of guesses, I initiated a new game, then guessed 'a', 'e', then 'i' to ensure that no chance is lost for correct guesses.
2. To test for a losing state, I continued guessing until my allowance of chances went below zero.
3. To test for a winning state, I had to disable the gameState.chances decrement in guess(), because I still don't know what combination will beat this very evil hangman.

Result

1. Success. The number of guesses displayed decrements only when I made incorrect guesses.
2. Success. Upon missing another guess with a 0 chances, the game state is cleared and #hangman_message displays a cheeky message.
3. Success. When I disabled the gameState.chances decrement and finally guessed all the letters in gameState.activeWord, the congratulations message was displayed as expected

Cycle 8

Milestone: The view/controller prevents players from improper inputs (rules and security).

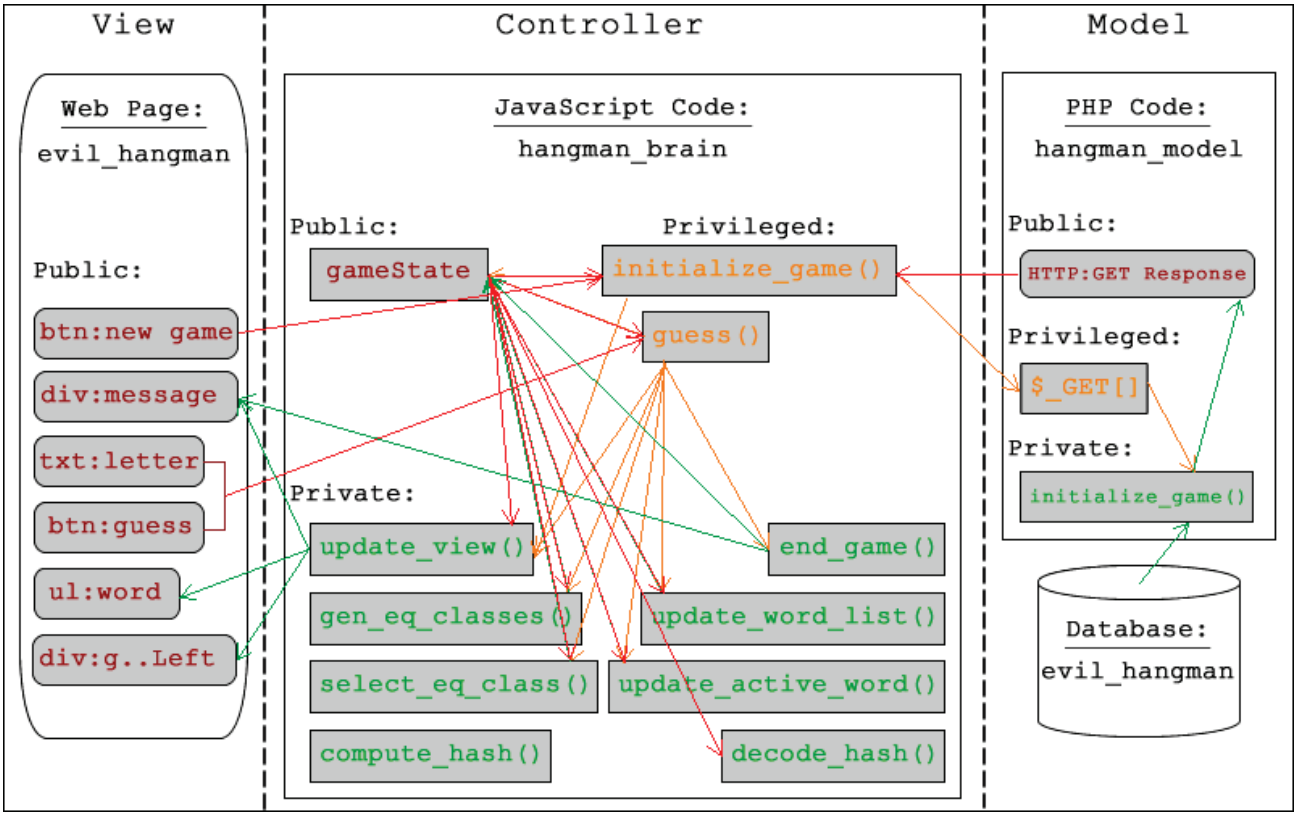
Design

The progression of this game relies on the player guessing single letters, between ‘a’ and ‘z’, one at a time. It would be a folly to rely on the player to voluntarily adhere to those rules, so the game must enforce the rules. Given the web stack on which I am developing, enforcement should occur at both the client level, to minimize bandwidth usage, and at the server level, for maximum security.

First, I must ensure that the view only submits a single digit at a time, and refuses to submit any input that is longer (or shorter) than a single character and does not fit the pattern, [a-zA-Z]. This will hopefully prevent most accidental errors from making it to the server and wasting bandwidth, but more powerful enforcement will be necessary to prohibit curious and clever players from actually hacking the game.

In the figure below, I have highlighted the connections that move between levels of authority. Of utmost concern are public vectors (red) that are sending data to private or protected methods (green and amber) because that information can be entirely fabricated. Also of concern are privileged vectors (amber) that are accessible to the public and have the privilege to access private members.

Figure 3.8a



On the client side (view and controller), I must also protect all of the controller’s members and methods from player access by privatizing everything that does not interface with the view and take extra caution with regard to how those interfaces process any incoming data.

In this example, the dangerous vectors are:

1. View: [New Game] → Controller: initialize_game()
2. View: [Guess]+letter → Controller: guess()
3. Controller: gameState → Controller: Many, etc.
4. Controller: initialize_game() → Model: $_GET[]
5. Model: $_GET[] → Model: initialize_game()
6. Model: HTTP: Get Response → Controller: initialize_game()

Vector 1

This does not directly modify any values in the controller or model, but it does provide a highly accessible means by which the player can call the controller method, initialize_game(), the dangers of which I'll consider when I analyze Vector 4. However, because that method does reset the gameState, potentially destroying a game during play, I should at minimum protect that button from accidental use when I design the UI in another cycle, by hiding it once a game has been initialized.

Vector 2

This will be the most heavily trafficked vector by a "typical" player. Because it relies on a player's inputs, I am almost guaranteed that over time, every single edge case will be tested, either accidentally or nefariously. To protect the program from potentially bad inputs, I will need to validate each player's guess for proper length and value; exactly 1 character and matching the pattern, [a-zA-Z].

Vector 3

In my current design, the gameState member is publicly accessible via the console, an obvious security deficiency which will need to be fully remedied to guarantee the integrity of the game. As famously demonstrated by Douglas Crockford[4], JavaScript provides a means in which private members can be created by utilizing closures. I am already using closures, but I have not been using the 'var' keyword to create local values, thus all values have been created at the global scope. This is a serious, but easily-corrected error, so I will add the 'var' keyword to all my controller value initializers.

Vector 4

As configured, the server is graciously providing a RESTful service as defined by Roy T. Fielding[5]. But because that service provides the model by listening for requests sent via any HTTP GET method that names the function to be executed, the potential exists that a clever player will fabricate his/her own GET request and thus be able convince the server to execute those functions outside of the game environment.

Because I am not using the actual value of variables sent with a GET request (I am only checking against a white list to call trusted variables within the hangman_model class) I have eliminated the threat of poisoned player input. However, sending a GET request to hangman_model.php with the GET variable initialize_game set will result in the JSON representation of the game state values being sent in response, including the entire word list with which the game is played.

One way that I can protect the database server from being spammed with too many requests, thus devouring all of the bandwidth and denying service to legitimate clients, is to limit how often an individual IP address can request that any function be called.

I would expect that realistic human behavior will never exceed 1 request per second, even for careless players who are guessing at random. So, to protect the server against a denial of service attack, a "DoS", I can maintain a table of IP addresses that accounts for the latest GET requests from the past 1 second. If another request comes from an IP address that is on that table, it can be denied. If this happens 5 times, I can safely assume that I am being attacked and will immediately add that IP address to a black list stored in a separate table. This will not protect the server against distributed DoS attacks, "D-DoS", but protecting a database against that threat is beyond the scope of this project.

In regards to a solution to the initial game state variables being transmitted in an unsecure mode of communication, I first need to make a policy decision. On one hand, I can leave the interface open (or even advertise it) so that other developers can reuse my model, or I can require that all clients establish a SSL connection with the server and then limit requests to only those which possess a shared key, much the same way that secured channel, e-commerce is implemented, or I could implement both.

Because the goal of this project is to make a game, I will leave the interface open and hope that all the cleverest players join me in spreading the joy of this game around the world. The worst case is that the player just wishes to cheat in the game by knowing the list of words in consideration, and the only loser in that situation is the player, and even so, only if (s)he derives no pleasure from winning via cheating.

Vector 5

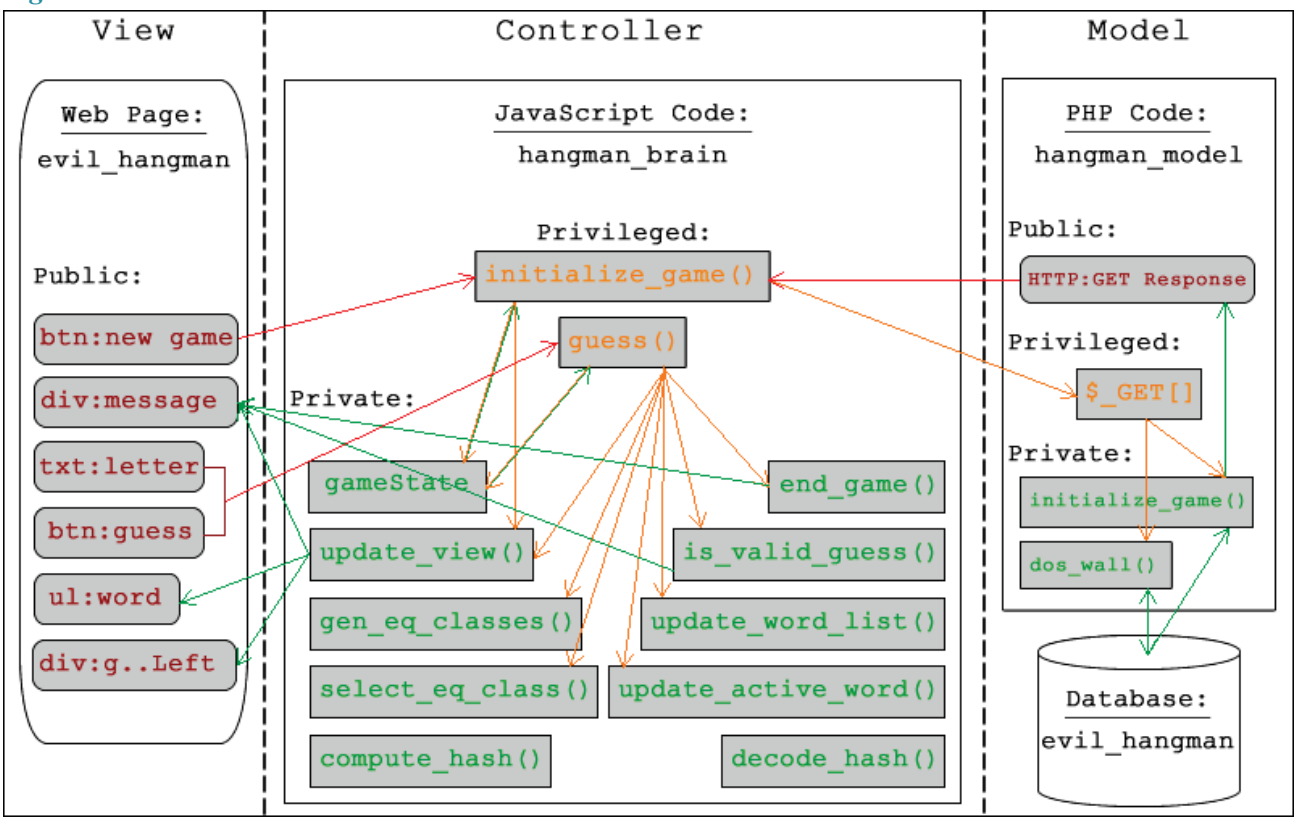
I avoid relying on values set by the GET method because it is always a security issue. What I do instead, to interface the controller with the model, is check if a certain variable is set, not what value to which it is set, then execute a predefined script that does not evaluate any player input at all.

Vector 6

Potentially, a clever player could abuse this vector to obtain information about the default game state. One way to abuse this would be to create another model for the game, spoof the JSON representation of the game state, then load "Evil Hangman" with whatever values that they wished. However, the data is not sensitive and only game initialization values are disseminated; protection from that sophisticated of an attack is beyond the scope of this project.

After securing all the dangerous vectors, the design will look like the following:

Figure 3.8b



Develop

Vector 2

In hangman_brain.js, I created a function called is_valid_guess(), which will be called by guess(). It returns a Boolean indicating the validation of the letter delivered to guess(). When negative, it will immediately terminate execution of guess(), alert the player to an improper input, then remind him/her of the rules regarding proper guesses. Only when is_valid_guess() is positive will the rest of guess() be executed.

Vector 3

In hangman_brain.js, I searched for all '=' characters in the file to locate all the variable instantiations, then added the 'var' keyword to each one.

Vector 4

In hangman_model.php, I added a call in my GET handler to a new private function, dos_wall(), which requires an IPv4 address as its only parameter and returns a Boolean that will indicate if a request will be honored or not. Then, the returned value is determined by querying the blacklist table for the provided address and the infraction count. If an address is found, and the count is at least 5, the request is denied. Next dos_wall() flushes all entries in the dos_wall table that have expired, i.e. rows that are older than 1 second. It then queries the table for the IP address, and if the address is found there, then the request is denied and the infraction is recorded (INSERT and set count to 1 if ip is not already on the blacklist,

UPDATE and increment count if it was). If the request is not denied, each successful request that is allowed through the dos_wall() is recorded in the dos_wall table.

In the MySQL database, I created the new table, evil_hangman_dos_wall, with 2 columns: ip as a VARCHAR(15), and dw_timestamp as a TIMESTAMP with a default set to CURRENT_TIMESTAMP. Then, another table, evil_hangman_blacklist, with two columns ip as a VARCHAR(15) and PRIMARY KEY and infCount as a SMALLINT.

Test

1. Because is_valid_guess() is a public method, I was able to test it directly by instantiating the hangman_brain, putting characters in #hangman_guess_box, calling is_valid_guess() via the console, then observing the value of #hangman_message. The bad cases to which I wished to be alerted are: no characters, too many characters, invalid characters. The good cases are: lower case and upper case single character guesses.
2. In the console, I executed this.window which displays all of the objects available to the user. By verifying that gameState (and other variables) is no longer present, I effectively made gameState a private member.
3. To test out the DoS protection:
    a. I added my IP address to evil_hangman_blacklist with count set to 5, changed hangman_model.php to immediately call dos_wall(), then loaded the script.
    b. Next I removed my address from evil_hangman_blacklist, and manually overrode the result of a check in the evil_hangman_dos_wall to be positive.
    c. Next, I modified hangman_model.php to call dos_wall() 100 times. Then, I changed dos_wall() so it would not remove any expired values and alert me whether a request is accepted or denied. I then loaded hangman_model.php in my browser and viewed the server log as the calls to the dos_wall() were made.
    d. Next I truncated the tables again, enabled the expired value removal, but then disabled all evil_hangman_blacklist queries, then changed the script to call dos_wall() 10,000 times.
    e. For a final test of the dos protection function, I enabled all the dos_wall() features, set all queries to echo, set the script to call dos_wall() 10 times, truncated the tables, then ran it.

Result

1. Success. No characters, too many characters, invalid characters, lower case and upper case letters are all handled individually and correctly.
2. Success. No local variables have been accidentally created in the global scope. By itself, this exercise serves to validate the importance of design and post-design analysis for many reasons
3. Success.
    a. When I tried to load dos_wall() with 5 infractions, I was denied at every request.
    b. The first call to dos_wall() resulted in an entry into the blacklist, and the next 4 passed but resulted in higher counts until 5, then all other requests were immediately denied.
    c. The first call was allowed, but the next 6 calls followed the pattern in part b.
    d. The first call was allowed, but the all other calls, except #1569 and #9782 were denied because they were caught on by the dos_wall().
    e. By the eighth call, all my requests were denied because my IP address was in evil_hangman_blacklist with an infCount of 5.

Cycle 9

Milestone: A UI and game flow has been designed and implemented.

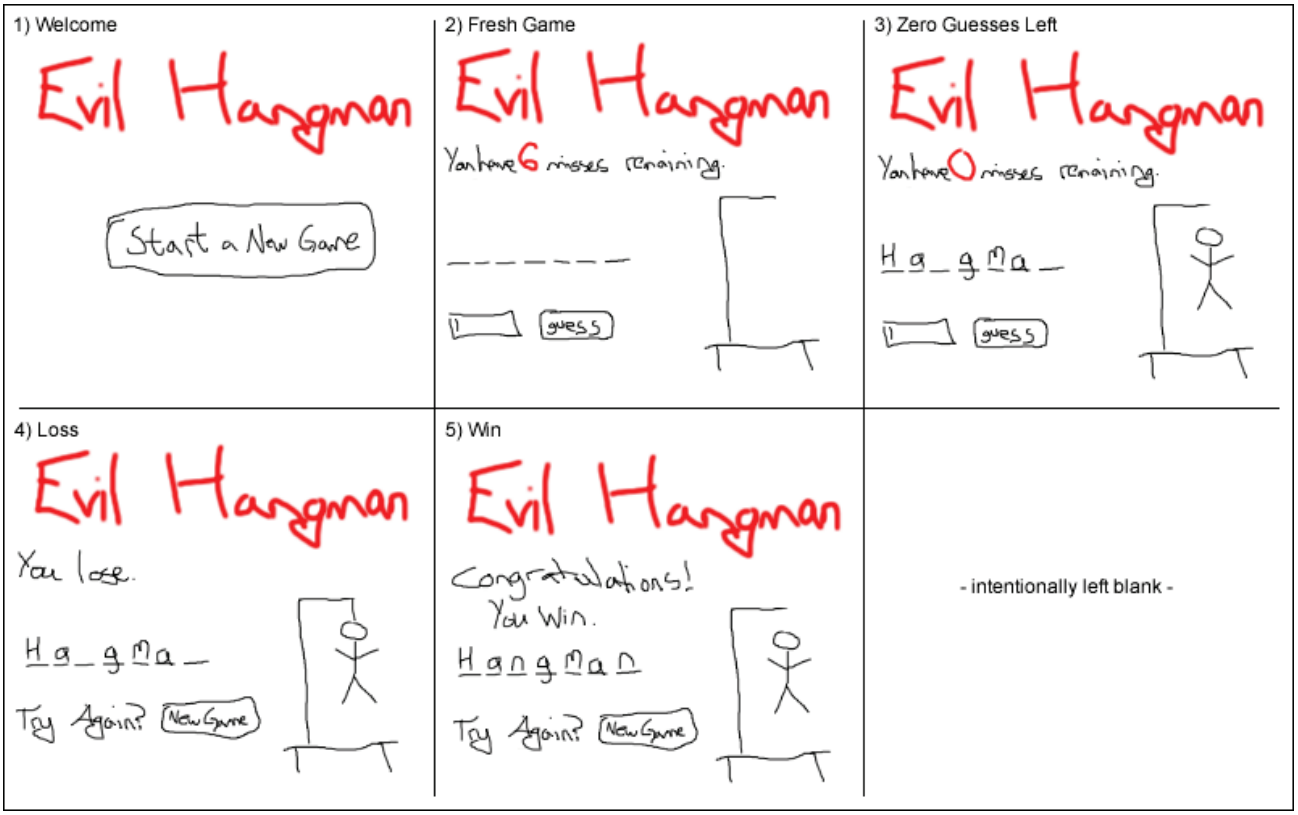
Design

In my opinion, this milestone deserves the full attention of a subset of an engineering team for the entire duration of the project, especially when the goal of the product is the player's experience. However, in order to expedite the completion of the project I am going to limit my focus to the essential details of the interface, so I expect that user feedback regarding the interface will be pushed to post-implementation.

Even with the goal of a bare minimum interface, to ensure that the Evil Hangman provides the best user experience possible, I will wire-frame/mock-up and obtain feedback on two completely separate designs. Then, having implemented the better of the designs as a view connected to the controller, I will player-test that interface and improve it based on the feedback obtained.

Figure 3.9a below is one of the two mock-ups upon which I obtained feedback:

Figure 3.9a



Feedback:

- Frame 2) Letter box may be too wide
- Frame 2) Consider putting platform on left side
- Frame 3) Misses / Guess is confusing (should say last chance)
- Frame 4) Should use more sadistic wording

- Frame 4) Make table of loser messages, choose randomly
- Frame 5) Should use more sadistic wording

Figure 3.9b below is the other of the two mock-ups upon which I obtained feedback:

Figure 3.9b

Feedback:

- Frame 1) Better word choice in button
- Frame 2) Better layout
- Frame 4) Better wording… but make even more sadistic
- Frame 5) Better wording
- Frame 5) Needs a place for in game messages (Warnings about bad input)

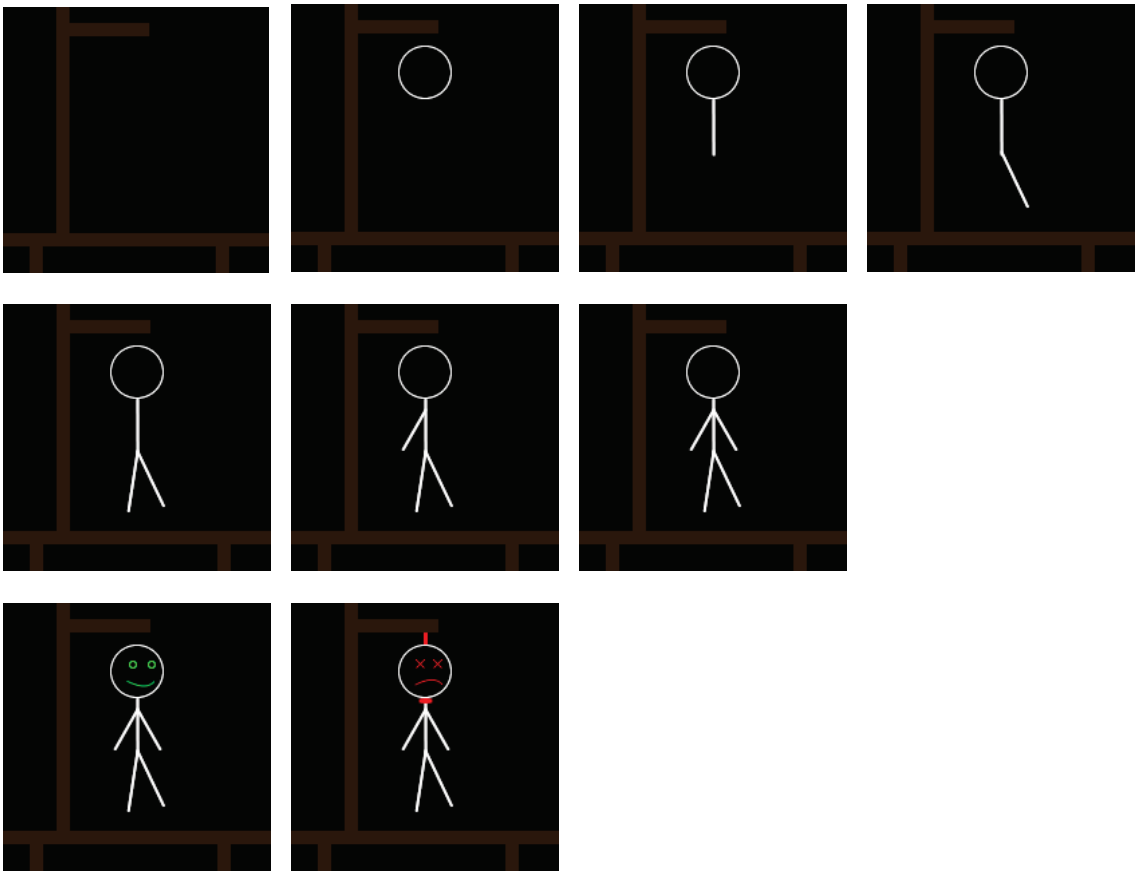
By far, the second interface was more popular for two reasons: the orientation of the hangman image in relation to the guess button and "active word" (center of the screen), and the slightly more sadistic nature of the messages used.

Based on user feedback, I have decided to proceed with mock-up 2, and will incorporate in-game messages directly beneath the line with the guess button. Also, "misses" will be replaced with "chances" to be clearer for the player.

Develop

First I created a PNG image for each hangman state. In example: 6, 5, 4, 3, 2, 1, and 0 chances, winning and losing. Then, I named them in a way that I can easily and programmatically update the view given the value of gameState.chances and the gameState.activeWord. The images are as follows:

Figure 3.9c



Next I created an image to serve as the title for the game interface as well as this document:

Figure 3.9d

Then in evil_hangman.php, I replaced the H1 tag with my new title image and created a page division with id #hangman_play to encapsulate the rest of the interface's elements. I plan to toggle the visibility of this division given the game state.

Next, I added a new image, #hangman_platform, and set its initial source to 'platform_6.png'. I plan to switch the source of this image to the other platform images depending on how many chances a player has left during the game.

Then, I added two new divisions, #hangman_message_win and #hangman_message_loss, and coded the messages and buttons that will display upon entering an end state. These divisions will be hidden initially, and only displayed during the correct end game state.

I have already created the divisions for encapsulating the remaining chances, the active word, hangman messages, the letter entry box, and guess button, so I rearranged those to be in the correct order according to the second mock-up design.

Then, I changed #hangman_chances to be a span instead, and put that span with the box and button into a new division, #hangman_controls. I will hide this division upon entering an end-game state to switch off the play controls. Next I moved the "New Game" button out of the #hangman_controls division, and into a new division, #hangman_welcome, and changed its value to "Start the Execution."

In hangman.css, I added a new stub for all the ids that I created in the previous paragraph: #hangman_play, #hangman_platform, #hangman_controls, #hangman_message_win, #hangman_message_loss, and #hangman_welcome, then set about defining their initial values. First, I set all the elements to add margin space and center themselves in relation to their parent page division. Then, I changed all the fonts to the correct sizes and weights and joined them to the serif font-family.

Because I would be hiding the entire #hangman_play division initially, I worked on stylizing all of its child elements first. I added a stub to reference the LIs that are programmatically created inside #hangman_word when the game is initialized and modified their display from block to inline. Then, I added another stub to reference the span inside of #hangman_chances, which I will use to make the number appear differently than the text in that message, and made its content large, bold and red. Next, I changed the #hangman_play to be hidden initially.

In hangman_brain.js, I then changed the code that creates the #hangman_chances text to encapsulate the number in a span and changed the word 'miss' to 'chance' per the user feedback.

Then, I modified initialize_game() to toggle the display values of #hangman_welcome, #hangman_play, #hangman_message_win, #hangman_message_loss, and #hangman_controls.

In update_view(), I added code that changes the #hangman_platform image's source to the image that corresponds with the number of chances remaining.

Then, in end_game(), I added code that hides #hangman_controls, changes #hangman_platform image's source, and toggles #hangman_message_win, and #hangman_message_loss's display to the correct values depending on the status parameter. To do that, I created a new button inside #hangman_message_win and #hangman_message_loss that works like #hangman_button_new_game, and created a stub for it in hangman.css.

Test 1

To test out the UI and game flow, I employed more test users and recorded their feedback. At one user's request, I reduced the word list to one word and allowed them to play and actually win.

Result

Partial Success. There were no bugs observed by any users, however, the users did suggest:

- Making the game easier

  -or-
- Allowing for selective difficulty
- Displaying a list of letters that have already been guessed

  -or-
- Not allowing letters that have been already guessed
- Making the buttons and text box "scary" like the rest of the interface
- Clearing the message after every guess so that errors do not carry over to successes
- Maintain record of how many times a player loses and pile up bodies in the background to depict how many men have been executed by the evil hangman
- Revealing what the final word was once player enters then end state
- Pressing Enter should fire the [guess] button
- The platform images load too slowly
- Clicking the new game button takes too long to respond

Troubleshooting

At this time, I do not plan to allow for modifying the difficulty, display a list of letters that have already been guessed or prohibit them in any way, maintain a record of previous games, or reveal what the final word was upon entering an end state. However, I do plan to:

1. Penalize players for guessing a letter that has already been guessed and selected to be in the "active word"
2. Redesigning the buttons to match the red/back/white theme of the rest of the interface
3. Clearing the message division at every guess
4. Enabling the use of the [Enter] key to fire the [guess] button
5. Increase the rate at which the platform images render

To accomplish that, I did the following:

1. In hangman_brain.js, I added code to guess() that checks if the selected hash was not 0 and the letter already exists in gameState.activeWord. That indicates that the player has already guessed that letter, so gameState.chances is decremented by 1 as if the player had guessed incorrectly.
2. In evil_hangman.php, I added the class .hangman_button to all the buttons in the view. Then, in hangman.css, I created a new stub for the class, .hangman_button, as well as its :hover status. In those stubs and in #hangman_guess_box, I set the values to make all the buttons better fit the rest of the interface, accounting for all browser types.
3. In hangman_brain.js, I modified guess() to clear #hangman_message and update_view() to empty #hangman_guess_box each time they are called.
4. In hangman_brain.js, I added code to initialize_game() that catches the "keyup" event, checks for the enter or return key, then fires guess() as if the player had clicked the [guess] button on screen.
5. First, I flattened the PNG files into GIF files. Next, I changed all instances of 'png' to 'gif' in hangman_brain.js and evil_hangman.php. Also, in evil_hangman.php, I added code to pre-fetch all the platform images for chances 6 through 0, plus the win and loss states. As a final touch, I added one LI to #hangman_word and some loading text to #hangman_message so that the player is never confused about the state of the game.

Test

1. I set gameState.words[] to only contain ‘hangman’ then guessed ‘a’, which correctly appears at zero-index 1 and 5. Then, I guessed ‘a’ again.
2. I viewed all the elements on screen at once.
3. I tried guessing ‘7’, then ‘a’.
4. I entered a letter into #hangman_guess_box then pressed my [Return] key.
5. I cleared my browser cache, and then played a full game.

Result

1. Success. When I guessed ‘a’ again, my chances decremented to 5 and the prisoner’s head was drawn in the view.
2. Success. All elements look marvelous and do not interfere with any other elements.
3. Success. The ’7’ prompted an error message and an empty guess box. Then, guessing ‘a’, made that warning message disappear.
4. Success. The letter was submitted, just as if I had clicked the [guess] button.
5. Success. Images load instantaneously and I was never confused about the state of the game.

Closure

Now that the project Execution phase is complete, I can focus on the steps required to complete and close the entire project. Remaining items include:

- Refactor the code
- Publish the Web-app
- Copy-edit and publish this document
- Package and deliver the final product

Refactor the Code

Because I chose to use an iterative execution process, I have been able to refactor many pieces of the code along the way. In Appendix 1: Suggestions, I have included details about any section of code that I feel may deserve improvement beyond the scope of this project.

The only major programming work remaining was the removal of the very revealing console.log() messages that were peppered through-out hangman_brain.js; all of which were easily eliminated by commenting out any lines of code that included the string "console.log".

Publish the Web-app

Because of the multi-layered stack that the web-app is relying on, this step actually has many parts.

Database

First, I exported the three MySQL database tables that I created for the project, evil_hangman_blacklist, evil_hangman_dos_wall, and evil_hangman_words, into a single SQL file. Then, I imported that file into my remote MySQL server. Next, I checked each table to ensure that the member count and the first 30 members were as expected; evil_hangman_blacklist and evil_hangman_dos_wall were both empty, and evil_hangman_words had all 109,549 members.

Code Files

Next, I established an FTP connection with my remote web server and uploaded my project files to the directory structure that already existed; evil_hangman.php to the pages directory, hangman_brain.js, hangman.css, and hangman_model.php to the scripts directory, and all the project images to a new sub-directory, "evil_hangman", within the images directory.

Integration

Because I have decided to integrate this project with my personal website, I now needed to integrate it with my site's interface. To do that, I was only required to create a new permalink for the project in the dropdown menus, and my interface took care of the rest.

Copy-Edit this Document

First, I read through the entire document, up to and beyond this point, looking for grammatical, semantic, and contextual reference errors, as well as ensured correct tense usage throughout related sections.

Next, I added page breaks where appropriate, while eliminating as many orphans as possible. I also removed most quotation marks and instead opted to use a color code system: files in indigo, functions and table names in gold, ids and names in green.

Then, I created the title page, footers, and table of contents for the document, and checked that all the Heading styles were properly assigned, the bookmarks were all in order and directing to the correct places, and the in-line links and the appendix links all worked.

Once everything was verified to the best of my ability, I published this document as a PDF and uploaded it with the code as a new repository at GitHub[6].

Package and Deliver the Final Product

The final product consists of:

- The documentation
  - A closing letter to the client
  - Evil Hangman.pdf (this document)
- The model
  - hangman_model.php
  - evil_hangman.sql (generative database script)
- The view
  - evil_hangman.php
  - hangman.css
  - images/evil_hangman (directory of images used by the evil_hangman.php)
- The controller
  - hangman_brain.js

This document and the model, view, and controller files are already published at GitHub[6] and bryan.WOLFFORD.com[1], but to deliver the entire project as a single package, I compressed these files into evil_hangman.zip, and then attached that compressed directory to an email containing the closing letter to the client.

Appendix 1: Suggestions

Here is a list of post-implementation suggestions that may or may not be pursued by the client after the completion of this project:

1. Currently, the controller is located on the client in order to reduce the server's processing overhead. However, it is not known if the overhead of transmitting the word list to the client, even just once per game, is more than the overhead of processing each game. More research into moving the controller to the server is highly suggested.

2. Further consideration of modification of the algorithm that generates and selects equivalency classes is suggested. After half the words in the "active set" are processed, checking whether the difference in sizes of the largest and second largest classes is greater than the number of words remaining to be processed, may identify situations where the best class can be selected early, terminating the algorithm immediately, and producing higher efficiency. However, according to preliminary calculations, this will achieve, at best, a 25% increase and, on average, a 12.5% increase in efficiency; but at worst it will incur an overhead of 25%, so more research into the topic is suggested.

3. Currently, the "active set" reduction algorithm is focused on minimizing the amount of memory required by storing only the counts of hashed words in gameState.eqClass[], but this is curtailed by the overhead of iterating over each "active set" and computing the hash of each word twice every time the player guesses a letter. This is not an enormous inefficiency given the maximum scale of the arrays; however, an alternative method would be to optimize calculation speed at the expense of memory by storing the first computed hash in a second array, then querying that array for members to the selected class instead of computing the hash again. Until this becomes a performance bottleneck or the "evilness" is to be upgraded, at a minimum, monitoring the performance of this algorithm is suggested.

4. As discovered in Cycle 5, Test Result 4, when guessing 'a', 'e', then 'i', select_eq_class() returns the hash of the highest member class, 0, 0, then 4. 4 corresponds to an 'i' occurring in the third to last spot, which includes words like "bobbing". There are not many letter combinations that will match this result, so this is an indication of how poorly this largest-class evilness algorithm performs. It is suggested that future work on this game includes utilizing the built-in commonality column in evil_hangman_words to increase the performance of "evilness."

5. It is possible to reduce the size of every gameState.eqClass[] by implementing each as an associative array, instead of numerical array, and by using hashes converted to strings instead of integers as keys. This will avoid generating gameState.eqClass[]s saturated with null values.

6. Realistic human behavior may exceed 1 request per second, so relaxing this limit imposed by dos_wall() may be advisable if too many false positives appear in the evil_hangman_blacklist.

7. Many players have complained that they quickly lose interest in playing the game after repeatedly failing to save the prisoner. So, if higher replay value becomes a goal, it is suggested that difficulty levels be implemented.

   Means by which a "less evil" experience may be provided:
   a. Allowing for custom, smaller "active word" lengths
   b. Reducing the maximum number of words that can be kept in the "active set" (which may reduce network bandwidth too)
   c. Increasing the number of chances allowed to the player
   d. Displaying a list of letters that are valid, and updating it after every guess

8. Some players suggested including a personal record of failures and successes. A system that generates and manages sessions and game ID numbers could be implemented.

9. Most players wished to see the "active word" revealed when they lost to the hangman. This could be implemented by, pseudo-randomly choosing a word from the final "active set" and updating the view to display that word upon entering the end game state.

Appendix 2: URLs

[1]: http://bryan.wolfford.com/?page=evil_hangman

[2]: http://www.sil.org/linguistics/wordlists/english/

[3]: http://www.codinghorror.com/blog/2009/01/the-sad-tragedy-of-micro-optimization-theater.html

[4]: http://javascript.crockford.com/private.html

[5]: http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

[6]: https://github.com/brychrwol/evil_hangman

Appendix 3: Figures