***Purple Slime's 5th Edition Character Menu Design Documentation***

**Documentation Development Team:**

Kerry Uchtorff

Alex Hodges

Kaleb Jacobs

**Overview:**

In the interest of full disclosure, this project evolved as it was created. As such, the original designs were constantly changing and growing into what this program is today. The hope here is to give a good breakdown of how the system is created for future developers.

In this document, we have broken down the system based on the top level requirements from the requirements manual.

1. Graphical Interface
2. User Decided Ability Score Assignment
3. User Driven Character Creation
4. Saving a Character
5. Deleting a Character
6. Loading a Character
7. Leveling a Character
8. Refresh Display
9. Save and Exit
10. Exit
11. User entered Data

**Total System Overview:**

This program is rather complex, with 39 individual classes and some that are only used in the UI class, so they do not have their own source code file.The architecture diagram of how the classes interrelate is in the Systems Diagrams folder called “PurpleSlimesCharacterMenuArchitectureDiagrams.jpg”. However, almost everything talks to the Character class and UI. As a result, it may be easier to see how the program is broken down in the System Control Diagram.

This program is being broken down to the individual processes that a user can trigger and how they are run. Each process will have its own section in the design documentation since it is not feasible to have them linked together here. From an overview perspective, the important thing to realize is that the entire program almost always returns to the graphical user interface and the UI class to await further instructions from the user. The two exceptions to this are the Save and Exit process and the Exit process which when finish, terminate the program. For a visual guide to the system, please see the file “System Control Diagram.jpeg” in the System Diagrams folder..

The first thing that happens to open the program is that the user clicks the .jar file. The program starts in the main class and that starts the UI. In the UI class, a Start class object called here is created and control is returned to the UI to await instructions from the user. The user can then make a choice on what they want to do.

If the user decided to choose a race, class or background, then they click on the correct dropdown. Once the choice is made, the correct variable in the UI is set to that value and the UI waits for more instruction. If the user decides to enter a name for the character, then the user types into the text field and presses the pen icon button to save the text from the text field to the correct variable in UI.

If the user presses the “Apply” button on the graphical user interface. If the variables for race, class, background and name are all filled in, then the User Driven Character Creation Process takes over. For more information about the User Driven Character Creation Process, please see Design for 3. User Driven Character Creation Process. Once that process is finished the Refresh process is called. When the refresh process is done, control is returned to the UI. If the user has not chosen a race, class, background or entered a name then the program will display a message box telling the user which of the four choices they did not enter and will return control of the UI.

If the user clicks the “Refresh” button, then the Refresh process is called. The Refresh process will display the most current information on the active character. The information is not saved when the “Refresh” button is clicked. For more information on the Refresh Process, please see Design for 8. Refresh Display for more information. If the program does not have an active character when the “Refresh” is activated, a message box will appear to tell the user to load or build a character before pressing refresh.

If the user selects “Save” from the File menu, then the Save process takes over. This process will check and take in the values that the user entered for certain fields in the User Entered Data. For more information on which fields are read in, please see Design for 11. User Entered Data. Once that data is collected, the Save process continues. When the Save Process is completed, control is returned to the UI. For more information on the save process, please see Design for 4. Saving a Character. If the program does not have an active character when the Save is selected, a message box will appear to tell the user to load or build a character before saving.

However if the user selects Save and Exit, then the process is slightly different. The Save process begins by checking and taking in the appropriate values that the user entered for certain fields in the User Entered Data. For more information on which fields are read in, please see Design for 11. User Entered Data. Once the data is collected, the active character is saved via the save process, but then the system calls System.exit() to end the program. More information about this can be found in the section Design for 9. Save and Exit.

If the user selects a new number in the level dropdown, then the User Levels Character Process is started. More information about this process can be found in Design for 7. Leveling a Character. Once the leveling process is done, the Save Process is called. This process will check and take in the values that the user entered for certain fields in the User Entered Data. For more information on which fields are read in, please see Design for 11. User Entered Data. The save process then continues to save the active character. Once the Save process is done, the Refresh process is called to display the most current information about the character on the graphical user interface. If the program does not have an active character when the level dropdown is activated, a message box will appear to tell the user to load or build a character before trying to level.

If the user selects Load from the file menu, the load character process is called. This process allows the user to load an existing saved character to the graphical user interface. More information on the load process can be found at Design for 6. Loading a Character. Once the character is loaded, the refresh process is called, which refreshes the graphical user interface to display the loaded characters information. Please see Design for 8. Refresh Display for more information.

If the user selects Delete from the file menu, the delete character process is called. This process allows the user to delete an existing characters save file and remove them from the index. More information on the delete process can be found at Design for 7. Deleting a Character. Once the delete process is done, control returns to the UI.

If the user chooses an alignment for their character, then the value that is chosen is placed in the variable align in the UI. If the character exists, then the character's alignment variable is set to the value of align. Once the alignment of the character is set, control returns to the UI. If the character does not exist, then control is just returned to the UI after the align variable is set.

If a user enters data in certain text fields, then the data has 2 methods to be saved. Some fields of data are only saved if the character is saved and the save process checks these fields. Otherwise, the data read directly into the character when a button is pushed. For more information on these two methods and the user enter fields affected, please see Design for 11. User Entered Data since the field depends on the method used.

If the user chooses to determine their ability scores, then the user clicks on the ability score buttons. There are three different ways for a user to choose how to determine their ability scores. Please see Design for 2. User Decided Ability Score Assignment Process for more information. Once the scores are determined. The values are placed in the statarray in the UI. If a character exists, then the ability scores are applied to the character and control returns to the UI. If no character exists, then the ability scores are placed in the stat array and control is returned to the UI.

The last thing the user can choose to do is select Exit from the File menu. This will call system.exit and close the program without saving any changes from the last call of the save process. The program will close.

**Design for 1. Graphical Interface:**

*Reasoning:*

To allow the user to interact with an interface as opposed to a console. This ensures that the user is able to create their character how they see fit and are able to find the information they need. It also allows for the user input that is needed to fill the necessary fields.

*Functional Description:*In order for the user to interact and view their character as it is created, a framework is required. The UI needs a multitude of empty fields to populate with information (Requirement 1.1.2.1). This is achieved by using a combination of objects from the Java Swing and Java awt libraries including JComboboxes, JTextfields, JLabels, JTextareas, JPanels, JButtons, JCheckboxes. These fields are grouped for aesthetic purposes and formatted to form a layout that is optimized for user intractability and readability. The panels hierarchy is represented by the indentation of the panels at their initialization with the leftmost panels being the primary containers. These fields are set to either be empty (if the input is text based) or they default to 0 in the case of int values (Requirement 1.1.2). This part of the program is grouped together with the other classes when the jar is created so that when it is called by main the program will launch (Requirement 1.1.1) the graphical user interface (Requirement 1.1.). At this point the program will be ready for user input.

**Design for 2. User Decided Ability Score Assignment Process:**

*Reasoning:*

In order for the user to have true control over their character choices, it was necessary to allow the user to not only roll for their ability scores, but to also determine how these rolls were made. To this end the program includes three rolling buttons which each provide a different functionality. The most important of these three buttons is the Roll Settings button(cogwheel). This button allows the user to choose the way rolls are done and how they are applied. If the user selects User Assigned they will be able to set where their rolls go. The other buttons are the roll button(Dice) which rolls initial values for the stats and prompts the user for input where needed and the reassign button(Refresh) which allows for the reassignment of rolled scores in case the user changes their mind in the process of creating their character.

*Functional Description:*

When the user clicks the cogwheel button (Requirement 2.1) the program opens a dialogue to ask the user to select their roll method (Requirement 2.1.1).

When the user clicks the button that looks like a dice (Requirement 2.2.) what occurs next depends on the selection made in the roll setting menu. If no selection is made, the default roll is that the dice will be rolled for the six ability scores(Requirement 2.2.2.1) and auto assigned (Requirement 2.2.2) in the order they are made (Requirement 2.2.2.2). If the character already has been created, the rolls modifiers are applied to ensure the values are correct (Requirement 2.2.2.3).

If the user selects the Standard Array roll option (Requirement 2.2.1) the program will load the array of values with the following numbers {15, 14, 13, 12, 10, 8} (Requirement 2.2.1.1). The program then launches a pop up window that prompts the user to assign this set of values (Requirement 2.2.1.2). After the user selects the 6 values locations and hits ok (Requirement 2.2.1.3) the program will assign the values where specified (Requirement 2.2.1.4). If the user has already applied to create their character any ability score bonuses are applied at this point as well (Requirement 2.2.1.5).

If the user selects the Rolled User Assigned roll method (Requirement 2.2.3) the program will roll six values and store them in an array (Requirement 2.2.3.1). At this point a pop up prompts the user for their assignment of the six rolls (Requirement 2.2.3.2). After the user completes their selections and hits ok (Requirement 2.2.3.3) the program will assign the new values to the requested locations (Requirement 2.2.3.4). If the character has already been generated the program also applies the needed ability score bonuses (Requirement 2.2.3.5).

When the user clicks the Reassign button which looks like a refresh symbol (Requirement 2.3). The program runs through the same steps for user assigned and standard array except it does not generate new values.

If Standard Array is the selected roll type(Requirement 2.3.1) the program will open a popup to allow the user to reassign their values (Requirement 2.3.1.1). Once the user reassigns and hits ok (Requirement 2.3.1.2), the program reassigns the new values to the ability scores (Requirement 2.3.1.3). If the character has already been created the rolls automatically have any ability bonuses applied (Requirement 2.3.1.4).

If Rolled Auto Assigned was selected (Requirement 2.3.2) the program will pop up with a message to state that this method does not allow the use of the reassign function (Requirement 2.3.2.1). This is the default setting.

If Rolled User Assigned was selected (Requirement 2.3.3), the program will prompt the user to reassign their rolls with a popup (Requirement 2.3.3.1). Once the user has reassigned their values as desired and hits ok (Requirement 2.3.3.2). The ability scores are reassigned by the program (Requirement 2.3.3.3). If the character has already been created the rolls automatically have any ability bonuses applied (Requirement 2.3.3).

**Design for 3. User Driven Character Creation Process:**

*Reasoning:*

To build a character for the game Dungeons and Dragons. This is for users who would prefer to have most of the development work automated and still allow the user to make choices as they see fit.

*Functional Description:*

This process will build a basic character. The character is not saved until a user either levels or saves the character manually, so if the user is unhappy with their character, they can hit the apply button to start with a fresh character. For a diagram of how the user drive character creation process works, please refer to the file in the System Diagrams folder called “User Driven Character Creation.jpeg”.

The process to build a character begins with the user making choices for the character. There are 4 inputs that the user must decide that are required to character creation but can be entered in any order. The user must select a race from the race drop down (Requirement 3.1), a class from the class drop down (Requirement 3.2), and a background from the background (Requirement 3.3). The user must also enter a name as well for the character (Requirement 3.4) and click on the pen icon to save it (Requirement 3.4.1). The race choice will be stored in a variable called race, the class choice will be stored in a string value called charclass and the background choice will be stored in a string variable called background. The name will be stored in a string called charname.

The user can make a couple of optional decisions. Users can roll for ability scores as outlined in Requirement 2.1 (Requirement 3.5). At this point, the user can decide to roll the character’s ability scores or it can wait until the character has been created (Requirement 3.5.2.1) If the user decides to roll the ability scores before the character is created, the ability scores will be applied as the character is built. The user can also decide to choose an alignment before the character is generated. If the user chooses to do this, the alignment is stored in a string variable called align

When the user is ready to generate a character, the user will click the “Apply” button (Requirement 3.6). The program will check if all the required choices have been made (Requirement 3.6.1). If one or more of the required choices has not been made, an error box will appear and tell the user the choice or choices that have not been made (Requirement 3.6.1.1) and the build process stops. If the user did make all the required choices, the program will begin to create the character. The first thing that the program does is clear all the fields in the graphical user interface.

The next thing done is that the Start method getRace() is called with the race variable as parameter. A Character variable is declared but not initialized. The race variable is put through a switch to determine the race of the character. Once the character’s race is determined, the constructor for that race class is called. If the race constructor will call its parent constructor. In some cases, that is another parent race class, if that is the case, then the parent constructor will call the Character constructor with no parameters (Requirement 3.6.1.2.1). If the original race constructor does not have a parent race, then it calls the Character constructor. From there, a blank Character object with default values is created. The control is returned to the calling constructor. If it is a parent constructor, then the parent race’s benefits, bonuses and features are applied to the character (Requirement 3.6.1.2.1.1). Once the parent race is done, then control returns to the original race to apply that race’s benefits bonuses and features. If the calling constructor is the original race, then that race’s benefits, bonuses and features are applied (Requirement 3.6.1.2.1.2). Once all the parents and calling races information is applied, the character object is returned to getRace() in Start. The getRace() method then initializes the character object with the new Character object and returns that to UI to be placed in the variable called player.   
 The UI then calls the method getbackground() from Start with the string background variable and the Character variable player as a parameter (Requirement 3.6.3.1). The getbackground method creates a background object with the character object that was passed to it as a parameter. This will apply any changes from the background to the active character. The background string variable will be put through a switch to find the correct background for the character. Once the correct background is determined, the appropriate method from the background is called. The method is generally named becomesBackgroundname(). This will call the appropriate method for that background from Background. Once it has been called, it will apply the correct bonuses, benefit features and equipment for that background (Requirement 3.6.3.1). When all the background information has been applied to the character, control will be returned to getbackground() in Start. The getbackground method will then be finished and return control to the UI.

The next thing the UI does is call the Start method getClass method which takes the string charclass and the Character player variables as parameters. This method will determine which class the character will be. A charclass variable is declared but not initialized. The class variable is put through a switch to determine the class of the character. Once the character’s class is determined, the constructor for that Java class is called. The Java class constructor calls the parent class of char\_class constructor. From there, a blank char\_class object with default values is created. The control is returned to the calling class constructor. If the calling constructor is the original class, then that class’s benefits, bonuses and features are applied as well as the class's calculations like hit points are applied (Requirement 3.6.2.1). Once all the parent and calling races information is applied, the char\_classobject is returned to getRace() in Start. The getRace() method then initializes the char\_class object with the new char\_classobject and returns that to UI to be placed in the variable called occupation..

After the race, class and backgrounds are applied, the character's name is added to the player via the setname method from Character that uses charname as a parameter. The next part applies the optional value for alignment and ability scores. If the user has chosen an alignment prior pushing apply, it will now be added to the player variable via the Character method setAlignment() where the new alignment is the parameter. If the user has not done this, this step is skipped due to a check to see if the variable is empty. If the user decided to roll for their ability scores before pressing the apply, the numbers that are in the statarray Array are then added to the current value in the variable placed in the correct variable for that ability score. The value in the variable before that time is the value of the bonuses from the race, class and backgrounds. If the statarray is still empty, this step will be skipped. After that any User Entered Data will be passed to the character via the correct requirement in User Entered Data Requirement 11.

Once all the data is entered, the ability new ability scores are displayed on the graphical user interface. This is pulled from the character as an integer and turned into a string to be displayed in the correct textfield. The ability mods are then calculated next and put into a String array with the use of the calcabilitymod() method from Character. This mod takes the ability score and determines the mod of that score via the getMod() method from Character. The values for all the scores are returned as a String array and the correct value is put into the correct text field on the graphical user interface.

After the ability scores are done, the skill mods are calculated with the use of the calcSkillmods() method in Character. Each skill mod has a different formula (See Requirements Document Appendix E). The calculated skill mod value will be put in the correct variable for that skill mod. Then other data fields will be filled in with the correct values from the Character player via the correct method from Character. Weapons and Armor values will be retrieved from the graphical user interface and applied to the correct variables in the active character. The values of these characters will then be redisplayed on the graphical user interface.

The next portion that the user will interact with is the choices pop ups for the character (Requirement 3.7). The first set of choices are in a method called raceChoice(). This method checks the characters race against if statements to determine if the race of the character matches that race, and if so the correct pop ups are created since each race can have a different number and types of choices (Requirement 3.7.1.1) A similar method exists for backgrounds that is called backgroundChoice, as well as one called ClassChoice for the characters class. These two methods work the exact same as raceChoice but the number and types of pop ups are appropriate for the class and backgrounds of the character. (Requirement 3.7.1.1) Since all the pop ups are similar across the methods, this will discuss how to make the pop ups in a more general discussion.

All items for user choice will be put into a pre-populated list and put into a choice pop up for the user. At this time, there is a known bug that if you click no on any pop up box, the choice is still applied. That is a bug that will be fixed in the next update. Once the user has made their choice, the value is applied to the character via the correct method from either the Character class or char\_class class. (Requirement 3.7.1.1).

A type of popup that is very common is the Choose a Language pop up (Requirement 3.7.1.1.1). This pop up box is populated from a list of possible languages that the active player does not already know that is found from the Character method possibleLang(). The possiblelanguages method takes an empty array list called possibleLang and compares it to the known languages list Languages from the character. If the character does not know a language, then that language is added to the possibleLang arraylist. Once all the if statements have been gone through, the returned arraylist given to the UI. This list is then used to pre-populate the Choose Languages choice box (Requirement 3.7.1.1.1.1 and 3.7.1.1.1.1.1.). This box is then presented to the user to make a choice. Once the user has made this choice, the new language is added to the active characters languages list via the checkandAddlang method that makes sure the language is not already contained in the characters languages feature. If the language is already in the list, the language is not added, if it is not contained, then it will be added to the list. (Requirement 3.7.1.1.1.2.) In many cases, the user may be asked to choose multiple languages. The process for the language choice box is in a for loop for how many languages the active character gets to learn. If the language choice box pops up after a choice was just made, the newly chosen language will not be in the list because the possibleLang() method is called every time a box is needed (Requirement 3.7.1.1.1.3).

Another type of popup that is very common is the Choose a Cantrip pop up (Requirement 3.7.1.1.2). This pop up box is populated from a list of possible cantrip spells that the active player does not already know that is found from the Character method possibleSpells(). This method takes a list of appropriate spells from the race or from the class as a parameter and a number that is linked to the spell level (Requirement 3.7.1.1.2.1.1). Cantrips are number 0, but if it is a race spell, it is number 10. The possiblespells method takes the passed arraylist and clones it to a new arraylist called playerallowedpossiblespells. These spells are compared to the known cantrip spell list CantripsKnown from the character via a switch that looks at the number passed to it to know which spell list to compare the spell against. If the character knows the spell, then that spell is removed from the playerallowedpossiblespells arraylist. Once all the possible spells have been gone through, the returned playerallowedpossiblespells arraylist is given to the UI. This list is then used to pre-populate the Choose Cantrip choice box (Requirement 3.7.1.1.2.1 and 3.7.1.1.1.2.1.). This box is then presented to the user to make a choice. Once the user has made this choice, the new cantrip is added to the active characters CantripsKnown spell list via the add new spell method from char\_class method that takes the active character, spell name and number as parameters. This uses the players checkandAddSpell method from Character that takes the name and level number as parameters. If the spell is already in the list, the spell is not added, if it is not contained, then it will be added to the list. (Requirement 3.7.1.121.2.) In many cases, the user may be asked to choose multiple cantrips. The process for the cantrip choice box is in a for loop for how many spells the active character gets to learn. If the cantrip choice box pops up after a choice was just made, the newly chosen spell will not be in the list because the possiblespells() method is called every time a box is needed (Requirement 3.7.1.1.2.3).

Choose Skills is a pop up is also very common (Requirement 3.7.1.1.3). This pop up box is populated from a list of possible skills that the active player does not already know that is found from the use of the char\_class method possibleskillsCharacter which takes the active character as a parameter and calls the possibleskills() Character method which takes a list of skills from the appropriate class as a parameter. (Requirement 3.7.1.1.3.1). The possibleskills method creates an empty arraylist called possibleSkill. The skills in the passed arraylist are compared by if statement to the names of different skills. If that skill name matches, the skills level is checked. If the skill level is equal to 0, then the skill is added to the possibleSkill list. Once the passed arraylist has been fully gone through, the returned possibleSkill arraylist is given to the UI. This list is then used to pre-populate the Choose Skill choice box (Requirement 3.7.1.1.3.1 and 3.7.1.1.3.2.). This box is then presented to the user to make a choice. Once the user has made this choice, the new skill name is passed as a parameter to the addskill method from Character. This passes the string parameter through a switch to identify the correct skill. Once the skill has been identified, the previous value for that skill is returned as a variable. If the previous value of the skill is less than 2, then the value for that skill is incremented. The skill name with skill in front of it is also added to the active characters Weapon Proficiency list via the checkandAddWepPro that ensures that the skill is not already in the list. Once that is done, control is returned to the UI (Requirement 3.7.1.1.3.3). In many cases, the user may be asked to choose multiple skills. The process for the skills choice box is in a for loop for how many skills the active character gets to learn. If the skills choice box pops up after a choice was just made, the newly chosen skills will not be in the list because the possibleskillsCharacter() method is called every time a box is needed (Requirement 3.7.1.1.3.4).

Every class in the Purple Slime Character Menu program allows for the user to choose the equipment for their class. (Requirement 3.7.1.1.4) This pop up box is populated from a list of possible equipment that is available to the chosen class. (Requirement 3.7.1.1.4.1). List of equipment is populated based on the class from the use of the char\_class methods to get the equipment lists that are called newequip1 through 4 where the class allows for that many choices. (Requirement 3.7.1.1.4.1.1) As opposed to many other choice boxes, the chosen equipment box will try to display all the available equipment to the user at one time by the user of multiple dropdown lists (Requirement 3.7.1.1.4.1.2). The chosen equipment will be added to the active characters equipment list by the Character method addEquip() which takes the chosen equipment name as a parameter (Requirement 3.7.1.1.4.2) This box should only appear once per class.

For every class, there is one level that will have the user choose their subclass with the Choose a Subclass choice box.(Requirement 3.7.1.1.5.1) These are mostly in the range of level 1, 2 or 3. For the generation process, we are only concerned with the level 1 classes. At this point, the char\_class list called domainoptions will be pulled from the class via the char\_class method getdomainoptions() put into a pre-populated list and presented to the user via a pop up box.(Requirement 3.7.1.1.5.1) This choice is then saved in the characters subclass variable by the setSubClass() method from Character that takes the choice as a parameter (Requirement 3.7.1.1.5.2) and places the new value in the active characters subclass variable. At this time, there is no way to undo this choice, so choose wisely.

In most class’s subclasses there are choices that can be made with the Subclass Choices pop up box (Requirement 3.7.1.1.6). In the classChoices method, sometimes there is an if statement inside the class that checks the characters subclass variable. If the subclass variable matches the subclass variable in the if statement, then a list of subclass choices will be pulled down from the correct class char\_class via the getsubclasschoices() method (Requirement 3.7.1.1.6.1.1). This information is used to pre-populate the list for the Subclass Choices pop up box (Requirement 3.7.1.1.6.1) that is presented to the user. The user’s choice is then returned and added to the correct location for that information based on various factors in the active character (Requirement 3.7.1.1.6.2) If the box appears multiple times, then the previous choice will be removed from the list of possible values. (Requirement 3.7.1.1.6.2.1).

A fairly rare pop up box is the Choose Expert Skill pop up box (Requirement 3.7.1.1.7) This pop up should only appear to classes that have the expert skill available which is Rogue and Bard (Requirement 3.7.1.1.7.1.2, Requirement 3.7.1.1.7.1.2.1 and 3.7.1.1.7.1.2.2). The Choose Expert Skill pop up will be prepopulated with skills the character already knows and is an expert in via the possibleExpertskillsCharacter method from the char\_class method with the parameter of the active character. This method passes a list of all the skills that are possible to the Character method possibleExpertskills(). The possibleExpertskillsmethod creates an empty arraylist called possibleSkill. The skills in the passed arraylist are compared by if statement to the names of different skills. If that skill name matches, the skills level is checked. If the skill level is equal to 1, then the skill is added to the possibleSkill list. Once the passed arraylist has been fully gone through, the returned possibleSkill arraylist is given to the possibleExpertskillsCharacter method that returns the list to the UI . This list is then used to pre-populate the Choose Skill choice box (Requirement 3.7.1.1.7.1 and 3.7.1.1.7.1.1). This box is then presented to the user to make a choice. Once the user has made this choice, the new skill name is passed as a parameter to the setExpert method from Character. This passes the string parameter through a switch to identify the correct skill. Once the skill has been identified, the previous value for that skill is returned as a variable. If the previous value of the skill is less than 2, then the value for that skill is incremented (Requirement 3.7.1.1.7.2.1). The skill name with skill in front of it is removed from the characters Weapon Proficiency list with the removeWepPro method (Requirement 3.7.1.1.7.2.2). The active characters Weapon Proficiency has the new skill as “Skill - skillname - Exper” added to with the checkandAddWepPro that ensures that the skill is not already in the list (Requirement 3.1.1.7.2.3). Once that is done, control is returned to the UI. In many cases, the user may be asked to choose multiple skills. The process for the Expert Skills choice box is in a for loop for how many Expert skills the active character gets to learn. If the skills choice box pops up after a choice was just made, the newly chosen skills will not be in the list because the possibleExpertskills() method is called every time a box is needed (Requirement 3.7.1.1.7.3).

By the end of the ClassChoice method choice boxes, all the values for the active character are calculated or recalculated for the character through various methods (Requirement 3.8). The program will always calculate the values for all variables that need calculated based on the correct formula for the class, subclass and ability scores as found in the Dungeons and Dragons 5th edition Player’s Handbook (Requirement 3.8.1). Once the values are calculated, the value will be placed in the correct variable, for more information, please see the Requirements Document Appendix E - Character information (Requirements 3.8.2 and Requirements 3.8.2.1). After all the choices and calculations are done, the graphical user interface is refreshed (Requirement 3.9) More information on the refresh method can be found in Design for 8. Refresh Display. (Requirement 3.9.1). After a character is built, changes can be made to certain areas. For information on adding the alignment or other information, please see Design for 11: User Entered Data. For information on adding ability scores after the character is generated, please see Design for 2. User Decided Ability Score Assignment. At this point the generate character process is done, and control is returned to the UI.

**Design for 4. Saving a Character:**

*Reasoning:*

For this program to really be useful to a player, it needed to be able to save the character the user was playing. This would allow the user to come back to the character at another time.

*Functional Description:*

To save a character sounds easy enough at first. But once you get started saving characters, and not just one character but multiple characters, it gets a bit more complex. For a diagram of how the save feature works, please refer to the file in the System Diagrams folder called “Save Method diagram.jpeg”.

The procedure to save a character starts when the user clicks Save on the File dropdown (Requirement 4.1). First the program checks to see if there is an active character in the system. (Requirement 4.1.1) This means a character has been created or loaded, and as a result the player does not equal null. If there is no active player, then the program will stop the save and a message box will inform the user that they need to build or load a character before continuing (Requirement 4.1.1.1)

The program then checks to see if a file path has been declared and if not, will ask the user for this information. (Requirement 4.1.2) If the file path has not been declared by the user, the system will open a directory chooser dialog box and will allow the user to navigate to their chosen folder (Requirement 4.1.2.1). If the user does not choose a file path and hits cancel the save is stopped. Once the file path is declared or chosen by the user, the program goes to the Start class to read in the index file with the readindex() method. The read index method looks for a file called index.txt in the declared save location. This is where a list of all the characters the program has created will be listed as a name and a filepath. If the index exists, then all the values from the index.txt file are read into an ArrayList of charindex objects called index.

The class charindex is a class that is only meant to hold the values from the index and to put them back. As a result, the charindex will take in a name and a filepath.

Once readindex() from the Start class is finished, it returns control of the procedure to the UI class. At this point, fields that do not have a button to apply them to the character are read in from their text field box and are saved to the character via mostly setter methods from the Character class. This includes things like weapons, armor, hair color, eye color and other fields.

When the program is done reading all the fields and placing their value in the character, the method saveCharacter() is called from the Start Class. Save character is where most of the save work is actually done. The readindex() method is called again to verify nothing has changed in the program’s index and then it checks a list of currently used players called savedCharacters to see if the active character is in the list. If not, the character is added to the list.

The program then goes through the list of saved characters to see if the active characters in the savedCharacters list have an old save file. If the character does, the file at that file location is deleted. Once the loop through savedCharacters is finished, a new file for our active character is created using the charactersname.txt attached to the end of the file path. The program then calls Character to run OutstringprintChar(file). File in this instance is the complete file path for the active characters save file.

Control is passed to Character who creates a file with the file path passed to it. Then every variable in the active character is passed to the FileWriter to write to a text file. When all the variables have been entered, the FileWriter finishes and control is passed back to Start. (Requirements 4.13 and 4.1.3.1)

Start then creates a charindex object for the active player. This new charindex object is compared to all the charindex objects in ArrayList index by name. If one of the names from the index matches the new charindex object, the char index at that location is removed from the index. Once the index arraylist is finished being compared, the new charindex object will be added to the index.

Next the file for the index.txt file is created and a new file created which will overwrite the old file. Then FileWriter is looped through the index and takes the name and filepath from the charindex item and writes them to the filewriter. Once this is done, the index file is closed and control is passed back to the UI. A messagebox will appear to confirm the character has been saved (Requirement 4.2) and the save procedure is done

**Design for 5. Deleting a Character:** *Reasoning:*

To remove characters the user no longer needs or wants.

*Functional Description:*

This program will delete the text file and remove the character index For a diagram of how the delete feature works, please refer to the file in the System Diagrams folder called “DeleteMethoddiagram.jpeg”.

The Delete Character procedure starts when the user selects “Delete” from the File menu on the graphical user interface (Requirement 5.1) . The program then checks to see if a file path has been declared. (Requirement 5.1.1) If the file path has not been declared by the user, the system will open a directory chooser dialog box and will allow the user to navigate to their chosen folder (Requirement 5.1.1.1). If the user does not choose a file path and hits cancel and the delete is stopped.

Once a file path has been established, the program calls getCharacterlist() from Start. The getCharacterList calls the readIndex() method. The read index method looks for a file called index.txt in the declared save location. This is where a list of all the characters the program has created will be listed as a name and a filepath. If the index exists, then all the values from the index.txt file are read into an ArrayList of charindex objects called index (Requirement 5.1.2).

The class charindex is a class that is only meant to hold the values from the index and to put them back. As a result, the charindex will take in a name and a filepath.

The readIndex method finishes and returns control to getCharacterList() which reads the index and retrieves the name from every charindex object and puts it in a new string arraylist called Charnames. The Charnames list is returned to the UI.

The UI checks to see if the list has any names in it. If the list is null, a message box will pop up and tell the user that they do not have any saved character to delete and to build a character first and then the delete process is stopped. (Requirements 5.1.2.1)

If the list does have character names, then the list is used to populate a character choice message box that allows the user to choose a character. (Requirement 5.1.3) If the user clicks no on the character choice message box, then a new message box will pop up and tell the user that they must choose a character to delete and the delete process is stopped. If the user chooses a character and clicks “Ok” the characters name is returned to the UI. (Requirement 5.1.4) The character name is then passed as a parameter to the DeleteCharacter() method in the start class.

The DeleteCharacter method will loop through the index arraylist and look for a charindex object with the same name as the name that was passed to it. Once the object has been located that has the same name value as the name parameter, the file path from the charindex object is retrieved and the file is deleted if it exists (Requirement 5.1.5). The matching charindex object is removed from the index array. Next the file for the index.txt file is created and a new file created which will overwrite the old file. Then FileWriter is looped through the index and takes the name and filepath from the remaining charindex items and writes them to the filewriter (Requirement 5.1.6). Once this is done, the index file is closed and control is passed back to the UI. A messagebox will appear to confirm the character has been deleted (Requirement 5.2) and the delete procedure is done

**Design for 6. Loading a Character:**

*Reasoning:*

To load a save character and modify it for use at a time other than when it was created. .

*Functional Description:*

This program will load a saved character from the text file. For a diagram of how the load feature works, please refer to the file in the System Diagrams folder called “Load Method diagram.jpeg”.

The Load Character procedure starts when the user selects “Load” from the File menu on the graphical user interface (Requirement 6.1) . The program then checks to see if a file path has been declared. (Requirement 6.1.1) If the file path has not been declared by the user, the system will open a directory chooser dialog box and will allow the user to navigate to their chosen folder (Requirement 6.1.1.1). If the user does not choose a file path and hits cancel and the load is stopped.

Once a file path has been established, the program calls getCharacterlist() from Start. The getCharacterList calls the readIndex() method. The read index method looks for a file called index.txt in the declared save location. This is where a list of all the characters the program has created will be listed as a name and a filepath. If the index exists, then all the values from the index.txt file are read into an ArrayList of charindex objects called index (Requirement 6.1.2).

The class charindex is a class that is only meant to hold the values from the index and to put them back. As a result, the charindex will take in a name and a filepath.

The readIndex method finishes and returns control to getCharacterList() which reads the index and retrieves the name from every charindex object and puts it in a new string arraylist called Charnames. The Charnames list is returned to the UI.

The UI checks to see if the list has any names in it. If the list is null, a message box will pop up and tell the user that they do not have any saved character to load and to build a character first and then the load process is stopped. (Requirements 6.1.2.1)

If the list does have character names, then the list is used to populate a character choice message box that allows the user to choose a character. (Requirement 6.1.3) If the user clicks no on the character choice message box, then a new message box will pop up and tell the user that they must choose a character to load and the load process is stopped. If the user chooses a character and clicks “Ok” the characters name is returned to the UI. (Requirement 6.1.4) The character name is then passed as a parameter to the getCharacterfromFile() method in the Start class.

The getCharacterfromFile method will loop through the index arraylist and look for a charindex object with the same name as the name that was passed to it. Once the object has been located that has the same name value as the name parameter, the file path from the charindex object is retrieved (Requirement 6.1.4.1).

The filepath is then passed to a method called loadfile() also in the start class. The text file is located at the file path that has passed to loadfile and the .txt document is read into the system via the Scanner class. The Scanner divides up the text file by each line. Each line is then divided by the split command on colons (:). The first part of the split is put into a variable called loadname.The second part is put into a variable called loadinfo. The loadname is put through a switch. The first line in every saved character file is the race. As a result, when loadname equals race, the value of loadinfo is passed to another method called loadRace() as a parameter.

The loadRace() race parameter is put through a switch that initializes a Character class object with the correct race constructor with the character name as a parameter. This is because all race classes and their parent classes have a constructor that does not add any of the classes information to the character. This is essentially a blank character of the correct type that is returned to the loadfile method and creates that character in loadfile (Requirement 6.1.4.2.1). From here, the next line is read in by scanner, and a loop is created of checking the loadname and filling the correct portion of the new character with the correct information using the Character class methods until there are no more lines in the text file (Requirement 6.1.4.2.1.1). This newly created character is then passed back to UI to become the active character.

Once back in the loadfile, the UI calls the Start class method of retrieveclass() which is passed the character’s class and the character itself as parameters. The method retrieveclass is another switch which matches the characters class to a list of classes. When the correct class is found, that class’s char\_class object is created using the name and the player as parameters. The player is set as the Character variable in the char\_class, otherwise no other information is created or saved. This prevents any changes to be made to the character since the character at this point should have all the necessary bonuses from creating a character. Once the char\_class object is created, the char\_class is returned to the UI. (Requirement 6.1.4.2.2). This is to allow a loaded character to level after the load is complete.

Once back in the UI, a flag that was previously false called loadflag is set to true. This prevents any changes made by the drop downs to make changes to the character. The UI at this point is refreshed (Requirement 6.1.4.2.2.1) via Requirement 8 Refresh Display and the character, race, level, background and alignment dropdowns are all set to match the character’s information via switch statements. After all of this is done, the loadflag is set back to false and the load process is done.

**Design for 7. Leveling a Character:**

*Reasoning:*

Characters in Dungeons and Dragons change as they gain experience. Their growth is determined by leveling. Levels show how mature or powerful a character is.

*Functional Description:*

This program will level an active character to another level. Do this carefully since currently there is no way to level down (This is a feature to be implemented in a future version). For a diagram of how the load feature works, please refer to the file in the System Diagrams folder called “Leveling Process diagram.jpeg”.

The user starts the leveling process by changing the value in the level drop down box.(Requirement 7.1). First the program checks to see if there is an active character in the system. (Requirement 7.1.1) This means a character has been created or loaded, and as a result the player does not equal null. If there is no active player, then the program will stop the save and a message box will inform the user that they need to build or load a character before continuing and the process returns control to the UI (Requirement 7.1.1.1) .

If an active character exists, the program checks if the new value for the level is a 1. If it is a one, then the character does need to go through the leveling process, and the process is stopped. If the level is greater than one, then the process checks to see if a file path has been declared. If the file path has not been declared, will ask the user for this information. If the file path has not been declared by the user, the system will open a directory chooser dialog box and will allow the user to navigate to their chosen folderIf the user does not choose a file path and hits cancel the character will not be saved but the leveling process will continue. If the file path is chosen, it is sent to the Start class to be stored.

Next the leveling process checks the LoadFlag variable. This is set to false by default. However, as discussed in Design for 6. Loading a Character, a true value on this flag will end the leveling process. If the Loadflag is false, the active characters class variable, char\_class value is retrieved and set to the variable Charclass.

This is where what the process does is HIGHLY specialized based on what the character’s class and level are. There is not a general set of procedures or changes to the character for all the levels. Each class has specific instructions for what happens to a character at every level based not only on their class, but also their subclass. As a result, please refer to the Dungeons and Dragons 5th Edition Player's handbook for specifics on what needs to happen when. The attempt here is to bring up a very generalized idea of what may happen in a leveling situation, but please know that all, some or none of what will be described in one level may happen at the same time from here till the character is saved in Requirement 7.1.1.2.3.5.

The overly generalized version of what happens begins with putting the value of the new level through a switch. When the new value is matched to a case statement, the player’s experience is set to the value that begins the range for that level. Next is where the Charclass variable is put through a series of if statements for only classes that have choices at that level (Requirement 7.1.1.2.2). Those choices are based on what the class has set for options and are highly variable. All classes will get an ability point pop up at 4th, 8th, 12th, 16th and 19th. However, other classes may receive extra ability points at various levels. (Requirement 7.1.1.2.2.1.1.)

Some classes have choices that apply to all members of the class. These choices will be put into a pre-populated list and put into a choice pop up for the user. At this time, there is a known bug that if you click no on any pop up box, the choice is still applied. That is a bug that will be fixed in the next update. Once the user has made their choice, the value is applied to the character via the correct method from either the Character class or char\_class class. There can be multiple choices for class and a level of this type (Requirement 7.1.1.2.2.1.4).

For every class, there is one level that will have the user choose their subclass. These are mostly in the range off level 1, 2 or 3. At that point, the char\_class list called domain options will be pulled from the class and put into a pre-populated list and presented to the user via a pop up box. This choice is then saved in the characters subclass variable by the setSubClass() method from Character that takes the choice as a parameter.(Requirement 7.1.1.2.2.1.2)

In every class, there are subclasses that will be chosen as the character levels. These subclasses may have choices of their own. In the switch in the UI class, the character’s subclass is pulled from a variable and compared in an if statement to see if the subclass matches. If the subclass matches, then the information is pulled from the char\_class class as a list and put into a pre-populated list and presented to the user to make a choice. Once the choice is made, the information is added to the character via the correct Character class or char\_class class method depending on the information. (Requirement 7.1.1.2.2.1.3)

Once all the choices are made, the code presents a break and then the switch ends. Next the Character class method Checklvl() is called (Requirement 7.1.1.2.3). The Checklvl method is a series of if statements that match the characters experience level. The low and high experience for each level is provided and if the player’s experience falls in that range and the current level is not in the pastlvls ArrayList, the character has the information inside the if statement applied to them. First thing that will always happen is that the level is set to the new level. The second thing that will always happen is that the previous level will be put in an arraylist of previous levels. Then the characters race is compared to other races that have certain things happen at certain levels. If the character's race matches, then the method inside is applied to the character, which will come from that race’s Character subclass. This information is applied to the character as needed and the race specific leveling is complete (Requirement 7.1.1.2.3.2). If the race doesn’t match, it is skipped.

After the race specific leveling is completed, the character will receive its class specific leveling information. Again this information is highly specialized, but the char\_class lvl#() method will be called. In this case, the # means the number of the level being applied. This will link the character back to its class by the fact that all class classes are children of char\_class and have their own version of lvl#(). Once inside the lvl#() method, the class specific leveling information will be applied (Requirement 7.1.1.2.3.3). After this information is applied, the char\_class method recalcabovelvl1hitpoints will be called. Again, each class has a different formula for calculating hit points based on the level adn the formula for levels 2 and above is different from the formula for level 1. As a result, please consult Dungeons and Dragons 5th Edition Player's handbook for specifics.

Once the hit points are calculated, control is handed back to the lvl# method. This then calls GUIsubclassinfo method.. The first thing it does however is fill a variable with the characters features list. This list contains a feature that is called the name of the subclass decision. This method is mostly a collection of if statements. The statements check the players level, the features list for the Subclass feature and the characters subclass against the name of the subclasses. If these 3 things are all true, then the subclass features and information is applied to the character using the appropriate methods from character and char\_class.

Once GUIsubclass is done, control is returned to lvl# method, which returns control to checklvl() in Character. The checklvl() method is also done, so it returns control the UI. The UI then saves the active character per Requirement 4 Saving a Character. Once saving the character is done, the graphical user interface is refreshed per Requirement 8 Refresh Display. Once all this is done, the leveling process is done and control is given back to the UI.

**Design for 8. Refresh Display:**

*Reasoning:*

This displays the most current information on the character. This includes recalculating all all values and recharging any values that can get low or deplete. This includes recharging hit points or spell numbers after a long rest.

*Functional Description:*

The user starts the refresh process by clicking the Refresh button.(Requirement 8.1). First the program checks to see if there is an active character in the system. (Requirement 8.1.1) This means a character has been created or loaded, and as a result the player does not equal null. If there is no active player, then the program will stop the save and a message box will inform the user that they need to build or load a character before continuing (Requirement 8.1.1.1)

If an active character does exist in the system, the UI class clears the text out of all the fields via the clearAllAreas method (Requirement 8.1.1.2.1) . Once all the fields have been clear of other text, the program recalculates the values for the character. The recalculation portion calls methods both from the Character class and the specific char\_class for that characters class. As a result, each child char\_class has different formulas and items that need calculations that are specific to that class. Please see the Dungeons and Dragons 5th edition player's handbook for more information on what needs calculated. However, these are called to the UI from the child char\_class by either the recalcClassModaboveLvl1 for characters who are level 2 or above or recalcClassmods for characters who are level 1. (Requirements 8.1.1.2.3)

Once all the calculations have been finished, all the fields are set with the most recent information from the player. These are generally the UI textbox using setText() using a call to character to get the information it needs from the correct call. An example of this would be:

uitextbox.setText(activecharacter.getStringvariable());

In some cases, if the value being called from the character is an integer and not a string. In those cases, the Integer class’s toString, to convert the value. An example of this is

Inttextbox.setText(Integer.ToString(activecharacter.getintvariable()));

The skill mods in particular have a dedicated method to turn the skill mod values into a string. This is found in Character as displayMods() with the parameter of the Characters skill mod variable. The displayMod method takes the value placed in it and turns it into a String with a plus or minus sign in front of it. Below is the pseudocode that would display a skill mod to the GUI.

public String displaymods(int skillmod) {

String Value;

if(skillmod >0 ) {

Value = "+" + Integer.toString(skillmod);

}

else

Value = Integer.toString(skillmod);

return Value;

}

The longer ArrayList values have their own method as well to help display them to the user interface. ArrayLists need to be converted to Arrays before it can be displayed on the graphical user interface. As a result, a method was created for every ArrayList Variable that needs displayed on the GUI. Below is the pseudocode for turning an ArrayList into an array for display.

public String[] StringarrayVariable() {

String[] str = new String[Variable.size()];

for (int i = 0; i < Variable.size(); i++ ) {

str[i] = Variable.get(i);

}

return str;

}

Once all the fields have called the correct information from the active character and displayed it to the graphical user interface, the refresh process is done.

**Design for 9. Save and Exit:**

*Reasoning:*

For this program to really be useful to a player, it needed to be able to save the character the user was playing. This saves the character before closing the program for the user.

*Functional Description:*

The user starts this process by clicking the “Save and Exit” button on the File Menu (Requirement 9.1). For a visual explanation of the process, please refer to the file in the System Diagrams folder called “Save and Exit Method diagram.jpeg”.

The process for saving the character is almost the exact same as in Requirement 4 (Requirement 9.1.1). The only difference is that if there is no active character, the program does display a message if there is no active character and closes immediately. Once the save has completed or there is no active character, the program closes via System.exit(0) and the graphical user interface disappears from the user's screen (Requirement 9.1.2)

**Design for 10. Exit:**

*Reasoning:*

Closes the program when the user is finished.

*Functional Description:*

The user starts this process by clicking the “Exit” button on the File Menu (Requirement 10.1). The program closes via System.exit(0) and the graphical user interface disappears from the user's screen (Requirement 10.1.1).

**Design for 11. User Entered Data:**

*Reasoning:*

To collect data that cannot be automated or that is information to be set up for automatic retrieval in a later version.

*Functional Description:*

This process has multiple parts that can initiate it; each part is fairly small and self contained. As a result, each part will be discussed separately. This is to collect data that the user will enter into various fields in order to be retained with the characters information (Requirement 11.1) Most if not all of this data is completely optional and will not be required to create a character, but in some cases, can change calculated values if entered. At this time there are a couple of fairly significant bugs in the user entered data. The first is that no colons may be used in the data otherwise, the save feature will not work correctly and the information after the colon will not be loaded. This issue will be addressed in the next patch.

The user is able to enter a hair color for the active character in the hair text box (Requirement 11.1.1.). The text from the hair box is saved to the character either during the User Driven Character Creation process if the user has entered it before hitting “Apply” (Requirement 11.1.1.1) or during the save process (Requirement 11.1.1.2) . The hair color will be redisplayed after a refresh or after a load or save (Requirement 11.1.1.2.1 and Requirement 11.1.1.2.2).

The user is able to enter an eye color for the active character in the eye text box (Requirement 11.1.2.). The text from the eye box is saved to the character either during the User Driven Character Creation process if the user has entered it before hitting “Apply” (Requirement 11.1.2.1) or during the save process (Requirement 11.1.2.2) . The eye color will be redisplayed after a refresh or after a load or save (Requirement 11.1.2.2.1 and Requirement 11.1.2.2.2).

The user is able to enter the height for the active character in the height text box (Requirement 11.1.3.). The text from the height box is saved to the character either during the User Driven Character Creation process if the user has entered it before hitting “Apply” (Requirement 11.1.3.1) or during the save process (Requirement 11.1.3.2) . The height will be redisplayed after a refresh or after a load or save (Requirement 11.1.3.2.1 and Requirement 11.1.3.2.2).

The user is able to enter the weight for the active character in the weight text box (Requirement 11.1.4.). The text from the weight box is saved to the character either during the User Driven Character Creation process if the user has entered it before hitting “Apply” (Requirement 11.1.4.1) or during the save process (Requirement 11.1.4.2) . The weight will be redisplayed after a refresh or after a load or save (Requirement 11.1.4.2.1 and Requirement 11.1.4.2.2).

The user is able to enter the age for the active character in the age box (Requirement 11.1.5.). The text from the age box checked to see if it is empty. If it is not empty, the string value is parsed from a string to an integer using the Integer.parseint() method where the string value is a parameter. This new integer value is saved to the character either during the User Driven Character Creation process if the user has entered it before hitting “Apply” (Requirement 11.1.5.1) or during the save process (Requirement 11.1.5.2) . The age will be redisplayed after a refresh or after a load or save (Requirement 11.1.5.2.1 and Requirement 11.1.5.2.2).

The user will be allowed to enter up to three weapons in the graphical user interface to be stored with the character (Requirement 11.1.6. The process is the same for all weapons, but the save locations differ slightly, so this document will be very generalized for the discussion of the process. All the information about a single weapon should be listed in the same line. Each line will be turned into a Weapon object and stored in the Weapon array called weaponsheld. (Requirement 11.1.6.1) Each weapon has four variables in the class that can be pulled from the graphical user interface when the character is saved. The weapon name will be read in from the graphical user interface box called weaponName# where the # is the line the box is on to a variable called name# using the .getText() ability on the weaponName# text field (Requirement 11.1.6.1.1). The weapon’s attack is read into the program from the weapAtk text field as a String. If the field is empty, then a 0 is placed in the field, otherwise the text in the box is parsed to a number via the Integer.parseInt() method where the text value from the field is a parameter. This numeric value is then stored in a variable in UI called attack# (Requirement 11.1.6.1.2) There is a known bug at this time that if a non-numeric value is entered in this box, it will break the program and that will be addressed in the next patch. The weapon damage will be read in from the graphical user interface box called weaponDmg# where the # is the line the box is on to a variable called damage# using the .getText() ability on the weaponDmg# text field (Requirement 11.1.6.1.3). The weapon type will be read in from the graphical user interface box called weaponType# where the # is the line the box is on to a variable called type# using the .getText() ability on the weaponType# text field (Requirement 11.1.6.1.4). Once all these values are collected, a Weapon object is created called weap# where the constructor from Weapon is called that allows for all the values to be variables. This object is then added to the active character using the character method addtoweapons which takes the Weapon object and a number to indicate the location in the array as parameters. The addtoweapons method then adds the Weapon to the weaponsheld array at the location that is associated with the number passed. (Requirement 11.1.6.2). These values can be retrieved for display when the UI is refreshed. The values are pulled from the Character method as getweaponname, getweaponattackbonus, getweapondamage or getweapontype and all methods take the location of the weapon in the array as a parameter. These values are then pulled from the weapon at the object using the correct value from the weapons class. The information is then returned to the UI and put in the appropriate text field with the setText method. The weapon attack bonus will turn the numeric value to a String before returning it to the UI. (Requirement 11.1.6.2.1 and Requirement 11.1.6.2.2).

The user will be allowed to enter one armor item into the graphical user interface. This information is entered in the armor boxes in the graphical user interface (Requirements 11.1.7.) The information about the armor is kept in an Armor object in the character called currentarmor. The Armor object has three variables in the class that can be pulled from the graphical user interface when the character is saved. The armor name will be read in from the graphical user interface box called armorName1 using the .getText() ability on the armorName1 text field (Requirement 11.1.7.1.1). The armor’s ac value is read into the program from the armorAC1 text field as a String. If the field is empty, then a 0 is placed in the field, otherwise the text in the box is parsed to a number via the Integer.parseInt() method where the text value from the field is a parameter. This numeric value is then stored in a variable in the UI called ArmorAC1value (Requirement 11.1.7.1.2) . The armor’s bonus value is read into the program from the armorMagic1 text field as a String. If the field is empty, then a 0 is placed in the field, otherwise the text in the box is parsed to a number via the Integer.parseInt() method where the text value from the field is a parameter. This numeric value is then stored in a variable in the UI called ArmorAC1bonus(Requirement 11.1.7.1.3) There is a known bug at this time that if a non-numeric value is entered in these boxes, it will break the program and that will be addressed in the next patch Once these values are retrieved from the UI, an Armor object is created with the constructor that allows the use of the variables as parameters to create the object. The Armor object is then placed in the active character as the currentarmor object using Character’s Setarmor method. This method also uses the setArmorArmorClassbonus method that takes the AC bonus from the armor object and calculates that into the active character’s ac value. (Requirement 11.1.7.1.2.1) Please see Requirements Document Appendix E for more on how to calculate the AC value. These values can be retrieved for display when the UI is refreshed. The values are pulled from the Character method as getarmorname, getarmorAC, or getarmorbonus. These values are then pulled from the currentarmor object in character. The information is then returned to the UI and put in the appropriate text field with the setText method. The getarmorAc and getarmorbonus both have to use the Integer.toString method to be turned to strings before being displayed. (Requirement 11.1.7.2.1 and Requirement 11.1.7.2.2).

The user has the ability to enter notes into the info box (Requirements 11.1.8). The user can type in any other information that they would like to keep with the character. The user can type directly into the Info box (Requirement 11.1.8.1.). When the user is done typing, the user presses the “save” button at the bottom of the save info box (Requirement 11.1.8.2). The information from the text box is read with the getText() and if it is empty, then nothing is done. If the information from info\_area is read as having data in it, then the entire box is a single string and placed in the active character with the Character’s setinfo method and the string as a parameter (Requirement 11.1.8.2.1). This information is displayed during a refresh or a load with by getting the data from the Characters getinfo method which is then handed to the setText method to place the data inside the info\_area text box (Requirement 11.1.8.2.2 and Requirement 11.1.8.2.3).

The rest of the user data from this point is developed in much the same way. The only differences are the names of the methods from Character. As a result, this will be a generalized overview that is used for Requirements 11.1.9, 11.1.10, 11.1.11, 11.1.12, 11.1.13, 11.1.14, 11.1.15, 11.1.16, 11.1.17, 11.1.18, 11.1.19, 11.1.20, 11.1.21 and 11.1.22. As such, the requirements for these sections are nearly identical, so the sub requirements are as well and will be listed as x.requirement numbers. In each of these cases, there is a text box where information about the character can be displayed. There is also a smaller text box at the bottom of the section (Requirement x.1.1).. The user can enter information about that portion of the character in that box they wish to add (Requirement x.1). Once the information is entered, the user presses the + button to add the information to the character (Requirement x.1.2). The information the user enters is put into a string in the UI which is then added to the active character’s matching variable by the correct Character method for that variable. Most of the requirements are held in a string ArrayList on the back end of the program (Requirement x.1.2.1). This textbox is then refreshed immediately with the correct values including the newest value. The next time the user refreshes the display or loads the program if they have saved before closing the program, the new value will also be displayed with the other values by calling the correct method to retrieve the information from Character and putting it into a string array. This array is then looped through until all information in the array is displayed using the append(value) method (Requirement x.1.2.2 and Requirement x.1.2.3).The user can also remove information from these fields (Requirement .2) The user enters the information they want removed in the text field at the bottom of the area the information is contained in. (Requirement x.2.1) The information must match exactly with case and spelling the information already in the box that they are trying to remove otherwise this information will not be removed because it will not match the existing information (Requirement x.2.1.1). Once the information has been entered, the user clicks the - button (x.2.2). At this point the information is taken from the bottom text box and put into a string and passed to the active character via the correct Character remove method as a parameter. The information is removed from the characters correct variable and control is passed back to the UI. the UI then clears the larger text area and redisplays the current information for that variable (x.2.2.1). This information minus the removed portion will be displayed after a refresh or a load if the user saved before closing the active character (Requirement x.2.2.2 and Requirement x.2.2.3).