D&D Forge

System and Unit Testing Document

12/1/15

Table of Contents

Database Testing (Unit)

User Story(s):
“As a programmer I want to have all stuff encoded into the program.”
Tasks:

- Database Race
- Database Class
- Database Background

Tested by: Andrew Miller

Testing:

I implemented the databases and worked closely with Kevin as he used them during his character page (character sheet) implementation. When he had trouble accessing something from the database we brainstormed on how we could better represent that data. One of the solutions we came to was to use array of booleans to represent certain constraints, such as which skills a class was automatically proficient in. This reworking of the databases started in sprint two and evolved when we moved into sprint 3. I reworked most of race data after better establishing a consistent pattern that the data would use in all 4 of the databases.

Our final structure for race data is as follows

```
setRaceFeatures(){
        //stats and other stuff like that, flavor text
        String race = “Race”; //race is shared with features
        int level = 0; //features are level 0, searchable.
}

setSubraceFeatures(){
        //stats and skills from subrace
```

```
        String race = “Race”;
        String subrace = “Subrace”; //searchable by subrace
    }

    setRaceAbility1(){
        //An ability that requires explanation, such as dark vision,
        String race = “Race”;
        String subrace = “Subrace”
        int level = 1; // abilities are level 1 to help with sorting through database.
    }

    …
    more setters
    …

    //loads all the racedata into the array list to be returned when called.
    public ArrayList<Data> getRd() {

    setRaceFeatures(raceFeatures); //set ability
    rd.add(ability1);//Then add them to the Arraylist

    //more setting and adding...

    return rd; //arraylist that all the race data is loaded into
    }
```

Class and Background data follows a similar pattern.

Character Saving/Loading Tests (Unit/Informal)

User Story(s):
“As a user, I want to be able to save and load characters quickly and locally on the app”
Tasks:

- Construct an algorithm using Android SharedPreferences to dynamically save data to the device memory.
- Construct an algorithm using Android SharedPreferences to dynamically load data from the device memory.
- Load all characters into a clickable card array (App RAM) on the app home screen that links the user to the corresponding Character pages.

Tested by: Alex Vincent

Testing:

This test required all portions of the data passing cycle (save to and load from memory) be implemented prior to use. As a result, the way this testing was carried out came in three parts: loading test (from memory), saving test (to memory), and efficiency initializing test (from memory) into the device's Random Access Memory.

The precondition to loading a character is that the character being loaded must have some non-null field that can be read from. In order to avoid reading from a null variable, each character object was given a default value that forced the fields to contain some item prior to being read. Depending on the variable (such as a character string stat), the default string would be "NULL". Load cases that caught the string "NULL" would be handled as a special case. Though during all unit tests, the string "NULL" was never loaded. The postcondition to loading a character ensures that the character object (inside a globally accessible array) has non-null fields corresponding to the saved values inside of it. To test this, we created an array of 20 hard-coded character values with distinct fields inside the program itself (not through app) and 20 hard-coded character values in the SharedPreference (system/device) memory. When the app loaded, we would load each character individually by clicking on their character card. For both hard coded and memory-loaded values, all characters successfully loaded into memory.

There is no precondition to the character being saved. This is because the postcondition of a load (ensuring non-null fields) guarantees that the fields being saved are either edits of existing fields or fields that have not changed since the last load (in which case the fields are overwritten by the same value). The postcondition of a character save guarantees that the changes made inside the character sheet are copied into the SharedPreferences/system memory. In order to test this, we loaded 20 hard-coded (both from the program as code and from the device as memory) character objects and individually changed and saved the editable fields. This process involved opening up a character, writing a new entry into the field, and pressing the "back" button to save that field into memory. We would then repeat the process for each possible field. At the end, we would edit all possible editable fields, save those changes, and load again. For every single unit test case, the character object was correctly updated with the changes and successfully loaded.

The last test done was an efficiency test to ensure that our saving/loading algorithm did not slow the app during heavy use. In order to test this, we created 20 characters from the application (each with their own unique fields, items, backgrounds), and observed any performance changes during runtime. At around 10-15 characters, we noticed a fractional performance hit in the form of a very slight (less than a millisecond) load delay when the app first loads from memory. However, this is still significantly less than loading from a web server (whose load times can be inconsistent with the same amount of data). Each of our loads loaded in the same measured time when running on a Nexus 5, LG G4, and Android Emulator. Overall, we found

that for large numbers of characters, the performance of the application (when it comes to saving and loading) is insignificant by a margin of a few milliseconds.

It's important to note the test was run with live user-inputs, not automated tests (such as JUnit). We did not need to run automated tests because the airtight algorithm constructed in saving/loading accounted for all cases. Also the reason our tests gave consistently positive results was due to the fact that Alex would constantly Unit test his code prior to pushing to the master branch. This ensured in the future that the project would have little to no bugs that would arise during the testing phase.

Character Questionnaires (Unit)

User Story(s):
"As a user, I want to be able to take a questionnaire to determine what class and race combination I should play, so I can quickly start making a character"
Tasks:

Tested by John Baden

- Design Questionnaire backbone
- Develop questionnaire questions
- Display suggested class/race combination after questionnaire is finished

Testing:

The questionnaire works by dynamically assigning tags in Android Studio to TextViews depending on the question.. When a user clicks one of these views, the program checks the tags associated with the view and applies the custom tag object to an arrayList. The arraylist is then parsed after the questionnaire is done and the results are displayed. Each question is hardcoded within the phone and is not passed into the app by any outside repository. Through thorough logging, I made sure that each choice in the questionnaire was being assigned the appropriate tags based on the current question, and its attributes that relate to the final end results were being tallied. After each completion of the questionnaire, I cross referenced the results that should appear with the result that do appear. I found that the results were appropriate.

Character Pages (Unit)

User Story(s):
“As a player I want to record my character’s race, class, stats, etc choices to the character sheet”
“As a programmer I want to implement the character sheet”
This includes:

1. Create items page
2. Create background page
3. Create abilities page
4. Create stats page
5. Create skills page
6. Create spells page
7. Create equipment page
8. Link pages together

Tested by: Kevin Mai

Testing:

I first designed the pages and attempted to hardcode values into the pages to test to see if they would display correctly. This included putting several strings and integers for the basic data fields and creating dummy arrays with possible expected values. Once those displayed correctly, I worked with Alex to try and pass hardcoded data from other sources to simulate the passage of saved data. Once that was done, I attempted to run the app on my own personal device and input values dynamically myself.

If I could manage to enter information from my device, then I would go back to the app and try to get the information to store locally to that page. I worked with Alex to implement saving functions and then attempted to have other pages access data that was saved from a previous page. If the app crashed for any reason, I would go back into the code and try to find the source of the bug either through the use of print statements or using the android debugger.

Once I got some of the data to pass safely between two character pages, I repeated the procedure with the rest of the pages and then attempted to link the character pages to the home screen to see if it was possible to send data from when the user first accesses their character.

From there I used a series of getter and setter methods to access data from the home screen and pass it into the character pages and attempted to have the system display the information independently and dynamically. If anything failed to show up or crash, I would once again trace through the code using the debugger and various print statements.

Once I finally managed to have the passed information working, I worked with Andrew to set up the database locally inside of each character page and attempted to access with the information that was passed from the home screen. I ensure the correct data was being passed by printing out each item that was being grabbed from the database and cross referencing it with the rule book to make sure the correct values were being withdrawn.

Lastly I did a full run through by creating a custom character from the screen and going through each page to see if the correct outputs were being displayed.