Assignment 2 ReflectionEvaluation

by Liam Samuel Roberts

FILE	25304_LIAM_SAMUEL_ROBERTS_ASSIGNMENT_2_REFLECTIONEVALUATION_312866_1973404489.PDF (916.76K)		
TIME SUBMITTED	11-DEC-2020 03:35PM (UTC+0000)	WORD COUNT	2098
SUBMISSION ID	139275203	CHARACTER COUNT	9968

Assignment 2: Evaluation and Reflection

ACCA7000, ACCA7001, ACCA7002

Liam Samuel Roberts

December 11, 2020

1 Introduction

For this report I will be reviewing the processes myself and my team implemented to tackle scenario one of the brief which was creating a text based console game. The purpose of the exercise was to simulate an agile working environment as the team undertook a 5 day sprint. The other members of the team were Rhys Beckett, Owen Skilton and Nathan Sathanandan.

---

2 Agile Process

The team followed an implementation of agile that we were exposed to in the Driver and Vehicle Licensing Agency (DVLA). The team unanimously decided to elect Rhys as scrum master. From there it was decided that we would hold a daily stand-up at 10am every day.

The DVLA use service designers to help craft user stories and so for this process to work the team behaved as service designers whilst deriving user stories from the brief. The user stories were then evaluated using the traditional planning poker method. After evaluation the team looked at what could be considered an minimum lovable product (MLP). However, it become clear the team did not have the capacity to complete the MLP in the time allotted. Therefore, the team decided to create a feature driven release structure that could be achieved with the estimated capacity. This can be seen in Fig. 1. After deciding on the capacity of the team, the sprint backlog was populated with the sprint being completion of the user stories required for release one. Completing release one would allow the player to move around the different rooms of the game.

The team set about populating a Kanban board to track the progress made as we tackled release one. In addition to the standard 3 columns a column was added for blocked tasks as well as tasks awaiting review and integration. This an be seen in fig. 2. The addition of a kanban board allowed us to tag who had worked on what. As seen by the tags in fig. 2 the team prioritised the process and collaboration over independent working and rushed progress which is inline with the agile way. The workload was assigned amongst the team as is usually the standard for a scrum team.

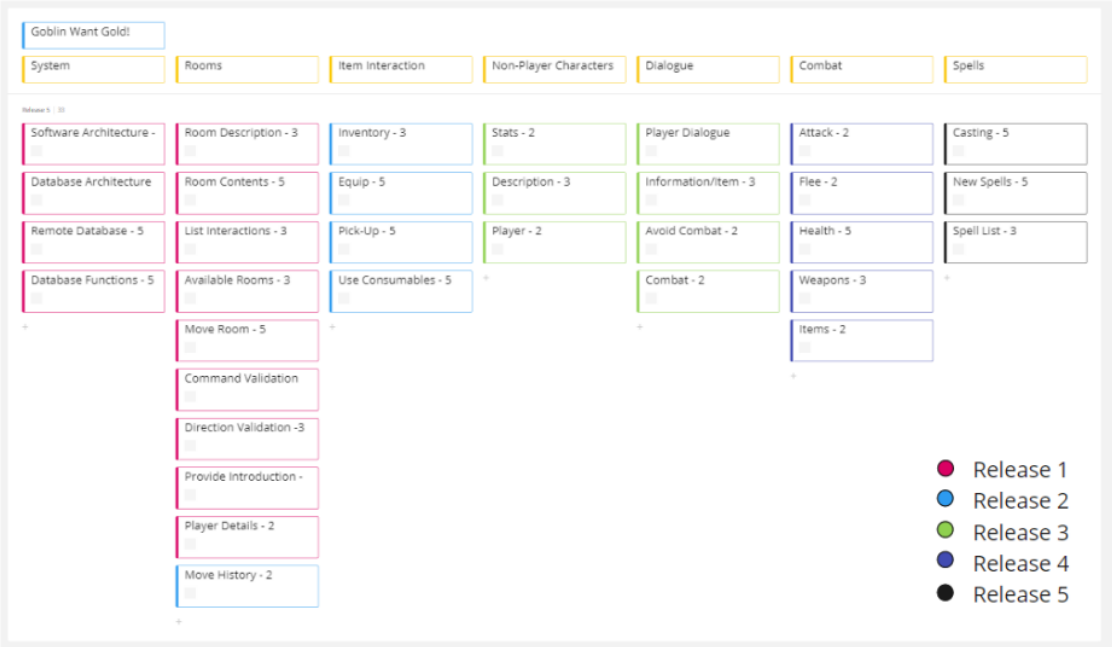

Figure 1: Release Organised User Story Board

2.1 User Stories

It was decided that I would be primarily responsible for the user stories which dealt with the actions the player could take and validating player inputs. This means I worked on the List Interactions, Command Validation and Direction Validation user stories which can be seen in fig. 3. Each of the user stories has a story point value, sub-tasks and an acceptance criteria.

To fulfil the List Interactions user story code needed to be written that would assess the type of data stored in room content to find out whether there were doors, characters or items in the room. For example, if a character was found in the room the code would need to print the "talk" and "fight" commands. List Interactions would also need to list valid contexts the user could provide. For example if there was a door to the north then the user should be informed that "move north" is a valid input. To complete this requirement the code would need to delve deeper into the room content to find the orientation of the different doors. After having gathered all the necessary data the program would need to print the data to the terminal.

Command Validation would require a function to allow the user to input a command. The user input would then need to be assessed against different synonyms for the valid game commands as users may specify "go" instead of "move." If a user input does not match any of the synonyms the user needs to be relayed this through the terminal. If the command is valid the program needs to use this to determine it's next function.

Direction Validation follows a similar pattern to the Command Validation process. The first task for this user story is to create a function that could split a user input like "go north" into a command argument of "go" and a context argument of "north." The context argument would need to be matched to different synonyms for the possible directions and then converted into a format that matched our database. For example, "forward" would change to "north" which subsequently changes the input to "N" to match the database. If the user input does not match any of the synonyms the user is told via the terminal. After this conversion process is successful the input is compared to each of the doors in the room to find the valid door. If

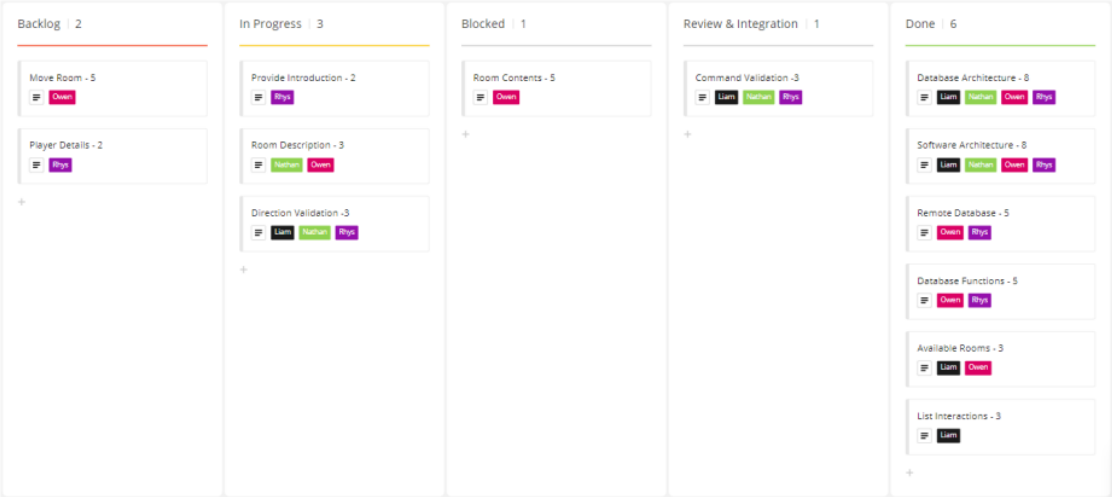

Figure 2: Kanban Board

no door is possible a message like "No Door in that direction" need to be printed to console. If a door is successfully found the doors orientation is fed to the move room functions.

2.1.1 Definition of Done

For a user story to be marked as complete it would have to meet the teams Definition of Done (DOD). The teams DOD was that a story had fully met it's acceptance criteria, code was reviewed by other member to asses that it met the coding standards, that we'd decided on, and each of the functions had been unit tested as much as possible. User Stories awaiting review were kept in the review and integration column of the Kanban board.

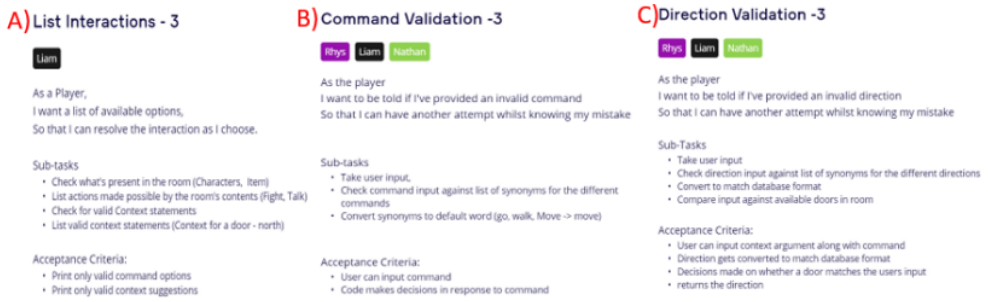

Figure 3: List Interactions(A), Command Validation(B) and Direction Validation(C) User Stories

2.2 GitHub

GitHub was used as version control software to allow members of the team to create branches for the user story they were working on. We used the pull request tool as a form of continuous integration to ensure that each time code was being merged to the master at least one member of the team had to check the code prior to the merge being permitted. GitHub has a reviewing

function to aid this process. The reviewing function shows both the code in the old master and the new code side by side, highlighting changes in red and green, drawing the reviewers attention to the necessary areas.

---

3 Database Design and Implementation

The team decided to do the Database design, implementation and population together as a group session. We managed to directly design a third normal form database diagram which we then created using PostgreSQL and hosted on Heroku. The visio drawing of the database is shown in fig. 4.

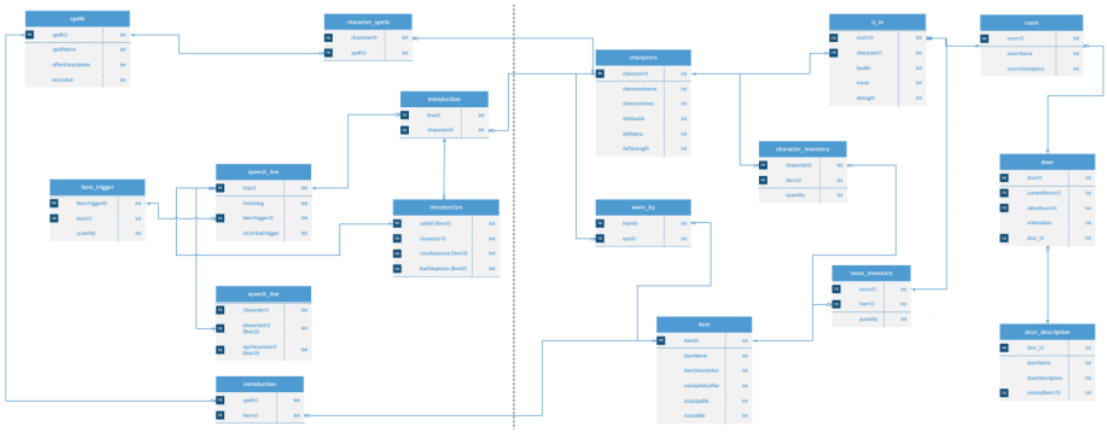

Figure 4: Visio Diagram of the Database Structure

The first step to designing the database diagram shown in fig. 4, was to consider entities needed for a functioning game. We decided there would be 12 entities in total. An example of these entities are "characters", "item" and "room." We then identified what data each required noting any overlaps to avoid data duplication. We then ensured that each non-key data was dependent on the primary key, leading to the creation of additional tables like "is in" and "room inventory." Each of these tables possessed foreign keys which relate the primary keys of the entity tables together. Finally to ensure third normal form the design was checked to ensure data duplication and transitive functional dependencies were eliminated.

My role in implementing the database was to set up the tables that involved character or spells, such as the "spell tome" or "characters table." This involved adding the necessary constraints, columns and data types using the CREATE SQL statement. I helped populate the table by using INSERT SQL statements to add in test data for the weapon and spell related tables.

We decided as a team that we would write functions in java to interact with the database to standardise the list of functions and to minimise the amount of different SQL statements contained within our code. I was responsible for unit testing the function that would use the SQL SELECT statement to search for an integer value in any of the tables contained in our database and return all of the data stored for that entry.

The tests I wrote focused on the item table and the character table as these were the only tables to have integer values. I added in test data sets into each table so that values that changes during game balancing would not effect the unit tests as the test data would remain

```java
@Test
void selectIntegerSuccess() throws Exception
{
    Connection conn = sqlFunctions.createConnection();

    // item test
    String[][] item_expected = {{"IT0000", "Test Name", "Test Desc", "99" ,"false", "false"}};
    String[][] item_results = sqlFunctions.selectStatement(conn, table: "item", column: "combat_modifier", filter: 99);
    item_results[0][1] = item_results[0][1].trim();
    item_results[0][2] = item_results[0][2].trim();
    assertArrayEquals(item_expected, item_results);

    // character test
    String[][] character_expected = {{"CH0000", "Test Name", "Test Desc", "0", "0", "0"}};
    String[][] character_results = sqlFunctions.selectStatement(conn, table: "character", column: "def_health", filter: 0)
    character_results[0][1] = character_results[0][1].trim();
    character_results[0][2] = character_results[0][2].trim();
    assertArrayEquals(character_expected, character_results);

    sqlFunctions.closeConnection(conn);
}
```

Figure 5: Units Tests for SELECT Statement Returning Integer

constant through out all development. The code for these unit tests is shown in fig. 5. Naturally the data in the database has defined lengths meaning that any strings returned would contain vast amounts of empty space. To account for this whilst unit testing the .trim() function was required to ensure the actual return matched the expected return.

---

4 Software Design Architecture and Testing

4.1 Design Architecture

Prior to begging the programming aspect of the work, the team broke the game down into the functions that we believed would be required based of our early user story refinements. This allowed for a clear process to be followed by each member of the team and meant that every member knew what their function was being passed and what their function needed to return as they were developing it. The initial version of our software architecture is shown in fig. 6. Each function is shown with the data type it returns as well as the data it should be fed. Each function name is descriptive of what that function aims to achieve.

One of design goals was to minimise the amount of work each function had to undertake before linking back to the main. However it was determined that "playerInput" and "getCurrentRoom" would act as nodes containing many different sub-functions that would relate to input validation, in the case of "playerInput" for example. Over the course of development the design architecture evolved to represent this as shown in fig. 7. Taking this approach to design helped

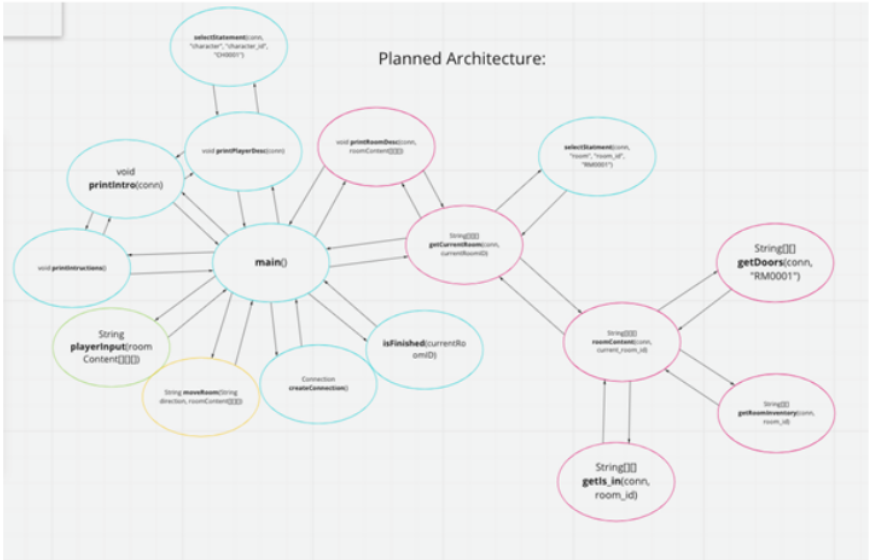

Figure 6: Release 1 Initial Design Architecture

Figure 7: Release 1 Final Design Architecture

```java
@Test
void commandValidation() throws Exception
{
    //Test success
    Boolean expectedCommandValidation = true ;
    assertEquals(expectedCommandValidation,interactions.commandValidation( input: "move"));
    assertEquals(expectedCommandValidation,interactions.commandValidation( input: "walk"));
    assertEquals(expectedCommandValidation,interactions.commandValidation( input: "go"));
    assertEquals(expectedCommandValidation,interactions.commandValidation( input: "open"));

    //test case is irrelevant
    assertEquals(expectedCommandValidation,interactions.commandValidation( input: "Move"));
    assertEquals(expectedCommandValidation,interactions.commandValidation( input: "MOve"));
    assertEquals(expectedCommandValidation,interactions.commandValidation( input: "mOVe"));
    assertEquals(expectedCommandValidation,interactions.commandValidation( input: "movE"));

    //test false conditions
    expectedCommandValidation = false ;
    assertEquals(expectedCommandValidation,interactions.commandValidation( input: "3210"));
    assertEquals(expectedCommandValidation,interactions.commandValidation( input: "0123"));
    assertEquals(expectedCommandValidation,interactions.commandValidation( input: " "));
    assertEquals(expectedCommandValidation,interactions.commandValidation( input: "%^&*("));
    assertEquals(expectedCommandValidation,interactions.commandValidation( input: "Banana"));
}
```

Figure 8: Command Validation Unit Tests

4.2 Testing

As stated previously I worked on the Command Validation user story. This involved writing the functions relevant to that user story and the subsequent unit testing. The Command Validation function is fed a user input which then compared against an array of array containing the different commands and various synonyms that match them. The functions returns a Boolean of true when a match is found or a false if it has finished comparing with the synonyms and has not found a match. The unit tests written for this function can be seen in fig. 8. The unit tests check that both true and false conditions are satisfied as well as ensuring that the comparison is case-insensitive.

---

5 Reflection

Despite the game not having many of the features specified in the brief I am happy with the outcome of the simulated sprint. We followed the agile process we'd been exposed to and produced what I believe to be good quality code after having placed significant emphasis on design of both the database and software architecture. The code produced has been thoroughly unit tested so I feel confident that our code is working code which is key for an agile team. I believe we made efficient use of the time given and estimated our teams capacity incredibly well.

Given a few more days I believe that the team could have achieved a fully functional game with well tested code. This confidence stems from the agile process implemented. We managed to ensure a reasonable working pace whilst being able to collaborate very well together as a team and the communication in the team was always excellent.

Upon reflection I realise that the estimation of stories was good however, in future I would give more consideration to the testing aspect of each story. This is something I believe would be naturally in work environments where there are software testers and developers in the team. Maybe in reality each story would be estimated a level above what the team originally estimated.

5.1 Contribution

During the sprint I was responsible for the functions in the green bubbles in fig. 7. In total through out the project I contributed to 9 story points as the lead and 24 story points as part of a group or paired programming sessions. This means I played a part in 33 Story points.

Assignment 2 ReflectionEvaluation

GRADEMARK REPORT

FINAL GRADE

61 /100

GENERAL COMMENTS

Instructor

Software design evaluation (30): 18

Database evaluation and implementation (30): 16

Coding and Testing (30): 21

Reflection (10): 6

Some curious choices of diagram tools used for the database and software design. It would have been preferable to follow existing conventions for diagrams. Your diagram choice has made it difficult to interpret the database structure particularly.

Although the majority of this report is reflective, it is a little too informal in its spelling, punctuation and grammar in places - be careful to proof read your work.

Your opening sections provided the necessary level of detail, however towards the end of the report the work became briefer and of lower quality. It seems like you may have run out of time.

A fairly good implementation overall - the code is clean, though would benefit from a few comments in places.

The database design is a little over-complicated and you haven't adequately discussed the need for such a complex set of joins - there's limited evidence of their testing or necessity.

Some good test functions generated, significant detail in granular testing of your functions.

A good code layout overall, separating functions into separate "classes".

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

Comment 1

no comment

PAGE 6

PAGE 7

PAGE 8