SOFT8027 Cloud Software Engineering - Assignment 1
David O'Regan R00156607
Kaylar's Vengeance High Level Overview:

The first thing I did when preparing for this project was to plan out the classes and tables I needed. I wrote the schema.sql file with three tables; players, equipment and players_equipment to represent the joins between the players and the equipment tables. I then wrote the data.sql file that inserted the various values into the tables. Rather than creating pieces of Equipment in the application class, I hard coded a list of equipment into the table, and the player then buys and sells from that list. The embedded H2 database takes the information from these files each time the application is run, and populates the objects that are created in the main Application class with the player's information and that of the equipment list.

I created classes for all the required entities involved in the project. I wrote classes for the Player, a generic Equipment class, and a PlayerEquipment class that represents the players_equipment table. Each of these classes has the generic getters and setter methods, and the Player class has an ArrayList that is populated with the list of Equipment that player currently has.

In the repository folder there is a repository interface for each class that holds all the methods for that repository, and a Jdbc class that implements those methods. The JdbcEquipmentRepository for example implements the get() method that uses a String variable called *sql* to "SELECT * FROM equipment WHERE id = ?", uses the EquipmentRowMapper to create an object and populates that object with every row of Equipment that can be found in the equipment table. The add() method does not create a new piece of Equipment, it just takes one of the pieces of equipment already in the equipment table and adds its reference id to the players_equipment table so it will show up in the player's arsenal when the returnPlayerInfo() method is run.

When the main application class is run, the player is presented with a greeting message and a summary of the player's information such as name, username and the list of equipment they currently possess. This uses a ResultSetExtractor with an Sql SELECT statement to populate a player object with its attributes, and to create an ArrayList that is then populated with the contents of the player_equipment table. This method returnPlayerInfo() is used after each other method is run to print a current summary of the player's details after another method is run to make a change somewhere.

The Menu and Methods:

A switch statement operates the looping menu options for the game. Case 1 is the option to Buy some more equipment. This method prints out the contents of the equipment repository using the findAll() method and a loop, and takes the user's input using the Scanner object. The Scanner populates a variable *eqChoice* and uses that to identify the piece of equipment the player will be buying. If the player chooses a piece of equipment they already have, an error message prints that fact and asks the player to choose again. I used an ArrayList populated with only the ID numbers of the player's current EquipmentList, and a conditional statement that says if the number the player has chosen is in this list, then to print the error statement.

If the player has selected a piece of equipment they do not yet own, the object is added to the player's inventory. An int variable is populated with the price of the item just bought and deducted from the player's balance.

The method for sellEquipment() in Case 2 works in a similar way, it identifies the piece of equipment the player wants to sell and removes it from the player_equipment table, and then adds on the price of that piece of equipment, minus 50 Kubits for depreciation.

The upgradeEquipment() method in Case 3 also takes an int *eqChoice* from the player and uses it to find that piece of equipment using the jdbcEquipmentRepository.get() method. The sql query upgrades that equipment by adding 15 to damageInflicted and to protectionProvided, and increasing the upgradeLevel by 1.

Unfortunately, I could not work out an efficient way of implementing the game's Rounds and allowing the player to progress through each Round earning more Kubits and upgrading each Round etc. I do have an upgrade() method where the player can upgrade their own equipment, although this might not make the end result game very exciting or challenging.

The other issue I had was when printing out the details of the player's current inventory. The ID for each piece of equipment was only displaying as 1 no matter what it was. I thought this was an error with the ID field but when equipment is purchased the ID shows up correctly. I tried various ways to fix this and populating lists with the player's inventory but I ended up having to stick with what I have. The functionality of each method does work but it does make it a little difficult to see what equipment is available to buy or sell without having the IDs displaying properly.

Testing:

I used Junit testing first for this project, and placed several Junit tests in their respective folders with src/test/java. The class named KylarsVengeanceApplicationTests has most of the single Junit tests in it, all annotated with @Test. I made use of the @DirtiesContext annotation to make sure the database reset itself after each Test so each test wouldn't be foiled by the results of the previous test. Each test I have in this file passes OK.

In addition to the JUnit tests, I also set up two Mockito test classes to test the integration of my Repository and Service classes together. This method creates a mock-up of the repositories it is testing, and creates dummy data to test, without having to access the actual database itself at all.

I wrote a class for the PlayerService class and the EquipmentService class, and although most of the tests in the worked OK and passed the tests using the dummy data inserted, the tests run through the PlayerServiceImplTests had a bit of an issue I couldn't quite work out. I created two dummy players, "James" and "Gillian" and assigned them ids of 1 and 2 respectively. When the test asserted that the firstName of player 1 was James it was incorrect, but when I changed it to assert that player 1's name was Gillian it somehow ran correctly. Any test I ran to assert that player 2's name was Gillian or anything else for that matter, it found 'null'.

In the EquipmentServiceImplTests most of the tests pass but again one does not and I was unable to figure out why. I tried to test the remove function of the Equipment Service but the test failed. There must be something wrong with my remove function somewhere along the way but I couldn't find where in time.

To check the overall code coverage I downloaded the Cobertura by running the Maven command mvn cobertura:cobertura from the Mac terminal in the project folder. The generated index.html file found in the site folder then showed the full extent of the test coverage so I can see what code has not been tested. It is not usually necessary to test every line of code but good code coverage is essential to a working system. I was not able to achieve as much code coverage in the end as I would like to have but it was very ueful to check on the progress of the tests.

I was unfortunately unable to add any additional features worth including, I had ideas of putting JFrame and JWindow popups for player feedback but it wouldn't have been ready in time.