Ian Foertsch

Joseph Corbett

Lily Nguyen

Steven Olson

# Software Engineering Deliverable II: Testing and Validation

Introduction

Due to our delay in finishing the class which contains the actual gameplay logic, we were unable to perform much integration testing. Consequently, this report primarily contains unit testing of our individual components, and any tests performed the components using mock objects to simulate component interaction.

## Utility Objects

The utility objects were a series of java classes which our group developed in order to streamline the component communication process. Rather than having methods exchanging long lists of parameters, the GameState and Action classes were developed to serve as containers which store all the information required to represent the current state of gameplay or to represent an action the human or AI players wish to perform. Due to the fact that these object are primarily containers and perform no operations on their data, their development and testing were quite simple.

One issue with the utility objects our group discovered during development was that our original specification called for representing card piles and card names with strings, this introduced the possibility of errors due to misspellings and case-sensitive comparisons. In order to eliminate these bugs, a series of enumerated data types describing card names, game types, and action names.

## Database Connector

### Database Design and Standardization

The Database was tested using a series of unit tests using the Junit framework. The debugging (as well as development process) was complicated due to the presence of the MySQL database, which had to be set up and configured correctly for the Java Database Connector class to produce expected output. This was especially problematic while exchanging software versions between development machines, where MySQL needed to be installed and configured (with a certain user and password set up on each with appropriate database privileges to allow the Java connector to initialize and make changes to the database.

### Exception Handling

The Database Connector class handles exceptions at a low level, which allowed stable performance but prevents communication between the connector and objects calling the connector’s methods. For example, unit testing on the .initialize() method passed, even though exceptions were thrown because exceptions were handled within the connector’s methods instead of being passed up the chain to the calling object. Plans for further revisions to the Connector class include making changes to eliminate low-level exception handling, and instead throw exceptions to objects calling the method in order to communicate unexpected behavior.

### Save and Load Game Functionality

Due to the incomplete GameObject class, which represents all the game logic specific to the Rat-a-tat-Cat game, the save and load game functionality has been left incomplete for the moment. A saved\_games database table has been included in the database initialization method as a placeholder, which provides the option to save and load a game using a save name and player name as a composite primary key, however all methods accessing this table currently have void return values.

### Unit Testing

A series of unit tests were developed for each database connector method, the results of which are included here. Due to the fact that the database connector is in some ways a one-way interface, meaning that it is easy to insert or add data, there are no methods at present to delete or subtract data, unit testing has to be done not with hard coded values, but rather making changes in the database and assessing those changes. For example, in order to test updating the scores associated with a particular user in the database, a preliminary set of scores were obtained by querying the database, a series of update score statements were executed, followed by obtaining a new set of scores and asserting that the two sets differed by a differential parameter.

The save game functionality, while incomplete at the moment, was tested to ensure that the table was receiving and reflecting updates correctly. The only component capable of being tested was the capacity to add a row containing the player name and the save name, which constitute a composite primary key. This was tested by adding a new saved game to the table, querying the database to obtain a list of saved game names corresponding to the test user, and asserting that the list contained a string equivalent to the original save name.

An additional issue with performing unit testing on a database which has persistence and storage beyond the execution of a particular testing run is that the changes made to the database during unit testing are permanent and stored in the database. Beyond the issue of having a saved games table which contains values like “TestUser: IGNORE\_THIS\_SAVED\_GAME\_FOR\_TESTING”, this prevents the unit testing framework from detecting changes to the database. In particular, when making changes the saved games table, because the player name and the save name constitute a composite key, any operations inserting duplicates into the table will be ignored. This means that the unit test may pass due to the fact that a tuple is found in the database which contains the variable in question, however no changes were actually performed to the database, and behavior which has become non-functional is labelled as functional. Correcting this behavior requires writing new methods for the database connector to drop rows from the saved game table, drop users from the profile table, and subtract values from a player’s scores in order to undo changes made during unit testing and ensure the integrity of tests.

## DatabaseConnector-GameWrapper Integration

Due to the fact that the GameWrapper is primarily a package class in which are aggregated sub objects, the process of integrating the DatabaseConnector class into the GameWrapper went quite smoothly due to the fact that there was relatively little which could go wrong. In particular, the GameWrapper exists to pass method calls from the GUI to the sub-objects, so implementing and testing the interface was simple, because the GameWrapper does not actually perform any on the data it receives as parameters.

However, an issue with the project architecture was discovered while integrating these two components. In particular, the project specification required a “login” method for the GameWrapper, which accepts a string as an input parameter describing the current player. Due to the fact that the database connector class is composed primarily of static methods which allows it to avoid storing any data itself, serving only as a connector to the database and translator of variables into SQL statements, storing a current player variable inside the database connector, or in the database itself would have been inappropriate.

The remedy to this problem involves creating an additional field in the game wrapper which stores the string describing the current player name. This string is then inserted into the update player profile method calls, which are made at the end of a game in order to update the database to reflect changes in the number of games played and similar statistics contained in the profile table. Lastly, storing this information in the game wrapper rather than in the GUI allows this operation to take place automatically at the end of a game or round without having to exchanges information from the GUI to the game wrapper.

Changes Required:

1. Methods to drop saved game names and profiles, subtract scores from user profiles in order to prevent changes to the database made in unit testing from persisting into the actual gameplay.
2. Alteration of the exception throwing system to allow exceptions to be thrown from the database connector to the object calling the method, (either the testing framework or the gamewrapper in the deployed game).