**Software Design Specification**

Revision – 0.1

Last printed

**Duel Reality**

**Approval Block**

|  |  |  |
| --- | --- | --- |
| **Name** | **Responsibility** | **Initials** |
| Josh Kilgore (Team Lead) | Game Mechanics & AI |  |
| Obi Atueyi | User Interface |  |
| Tom Calloway | Graphics Window |  |
| Ye Tian | Database |  |

**Abstract (Tom)**

The following is a software functional specification document for the Duel Reality personal computer game. The document fully identifies and describes both the high and low level functionality of the software without going into the design details themselves. This document attempts to serve the needs of those looking to understand the functional requirements of the game from both the user and designer perspectives.

**Revision History (Tom)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Revision** | **Date** | **Revised By** | **Comments** |
|  |  |  |  |
|  |  |  |  |

# Introduction (Obi)

This document describes the functions of the Duel Reality modules, in accordance with its Architectural Specification [2].

Interactions between the user and the game are provided via the User Interface (UI). The UI is the main window that comprises the menu, toolbar, and status bar. It is through this interface that the user sets the desired game play options and receives error messages during battles.

The main window also contains a game view that comprises the map and player unit. This is the Game Graphics and it provides the user a visual representation of the state of the game during a battle.

The Game Mechanics provides the state of the game to the Game Graphics. In response to the user’s turns during battle, the Game Mechanics uses the game play options entered at the UI and interactions with the Artificial Intelligence (AI) to define the state of the game at any time.

The AI is the user’s opponent that adapts to the game level and the user’s units. It retrieves game and user information from the Database and uses this information to model an opponent suitable for the user’s experience level.

The Database stores information provided by the AI and UI. It also fetches information on request by the AI and UI. The Database provides permanent storage of such information for use during future program runs.

## Document Purpose (Tom)

The purpose of this document is to provide the functional specifications for the development of the Duel Reality turn based strategy role playing game.

## Product Scope (Josh)

This game is meant to be run on a solitary Windows PC. User interactions will take place with a standard mouse and keyboard. Graphics will be displayed on the screen and sound played through the standard sound output device of the computer.

## Terminology

Sprite A two-dimensional pre-rendered figure

Unit An individual infantry

Widget An interactive feature pertaining to user interface graphics

## Acronyms

AI Artificial Intelligence

AP Action Point

XP Experience Point

GUI Graphical User Interface

UI User Interface

TBS Turn-Based Strategy

# Overall Description (Josh)

The program displays graphics and text on the screen and interacts with the user by use of the mouse and keyboard. The Interactivity portion of theUser Interface takes input from the user and uses that to update the game, which is then output back to the screen from the Visualization part of the User Interface. The Interactivity portion of the User Interfaceis responsible for handling player commands like “move here” and “attack this target.” It is also responsible for such things as the log-in of players, game settings, and menu commands like “NewGame” and “Save Game.” The Visualization part of the User Interface outputs its data by drawing all of the objects which appear on the screen, effects, sound, and the visual interactions between the objects and the game board.

The Game Mechanics section handles the inputs from the User Interface and applies the game rules and does the internal math between objects, and returning the results to the User Interface to be displayed on the gameboard. The game mechanics section also handles the Artificial Intelligence which is responsible for giving the player a playable opponent.

The Database section is responsible for retaining the data associated with the game and the units and passing that data back to the Game Mechanics and User Interface sections when needed.

(Basically reposted from Arch Spec, will re-work, suggestions welcomed)



## Module Description

//BRIEF Description of Module – will be expanded in section 3.0 (what it is, what it consists of, what it does)

## UI

## Graphics

## Game Mechanics

## The Game Mechanics module comprises the rules about how the different parts of the program work together to present the player with a full experience. In this section are defined the units, their interaction with the board and the players, the basic rules of the game including movement and actions.

The AI consists of a set of decisions and actions which will mimic the actions of another human opponent for the single Player to play against. The AI will operate under the same constraints as the Player in terms of gameplay, but will not have the ability to automatically upgrade its capabilities. This AI model will be sufficient for the purposes of this game, and can scale easily in complexity.

## Database (Ye)

## The game database contains all of the tables and data records, which function like a background support to other modules. It is a collection of the names, parameters, status of all units, and the game content such as maps. The data is retrieved and overwritten in real-time game going and asynchronous backup.

## Design Constraints (Josh)

Some of the limiting factors in our design and execution of this project include the time limit of having the project due on a certain date and being limited to C++ for development language.

## Assumptions and Dependencies (Josh)

We assume that C++ object oriented programming would be sufficient to do game design. That using QT tools adds value to this process, that time constraints would be   
sufficient to make a working game, and that we were assuming a Windows   
environment for deployment.

## Design Environment and Tools (Ye)

SQLite is used to design database module. SQLiteSpy is used to easily create and operate database file gamedata.db3.

# Software Design

## User Interface Classes(Obi)

//DISCUSS CLASSES IN THIS MODULE AND HOW THEY FIT TOGETHER INTERNALLY & WHAT THEY INTERACT WITH FROM OTHER MODULES.

### Description

Although the UI module is, in general, the application display to the user, it will be separate from the game graphics. The UI enables the user to configure game options for each level of the game. The menu options are used to start, load, save or quit a game. During a battle, the user uses the tool-bars to move or perform actions on units. Any invalid moves or actions during battle are indicated on the status bar.

## Game Graphics (Tom)

//DISCUSS CLASSES IN THIS MODULE AND HOW THEY FIT TOGETHER INTERNALLY & WHAT THEY INTERACT WITH FROM OTHER MODULES.

### Description (Tom)

The main dialog of the Duel Reality PC game features a game graphics window, which represents the map and units currently in play. As such, the graphics window is the user’s primary feedback regarding the current state of the game. The graphics window will include a background image, two-dimensional sprites to represent units, and a grid representing the discrete locations of the map. The game graphics window will also feature sounds and visual effect to represent significant game events.

## Game Mechanics (Josh)

//DISCUSS CLASSES IN THIS MODULE AND HOW THEY FIT TOGETHER INTERNALLY & WHAT THEY INTERACT WITH FROM OTHER MODULES.

//CURRENTLY JUST A DRAFT BASED ON MY .h and .cpp FILES

### Description

The Game Mechanics module has several different and distinct classes that handle the back end manipulation of data.

* + 1. **Player Class**

The Player class is responsible for determining the individual player and the attributes associated with that player such as the current campaign level played, the XP gained so far, the amount of XP the player has to spend on unit upgrades.

* + 1. **Unit Class**

The Unit class represents the units which are placed in the map to do battle. The Units are each distinct in terms of class, for instance a soldier has different amount of health and action points than a wizard, as well as a different appearance. Units have several attributes including:type, facing direction, vertical location, horizontal location, team number, health points, action points, movement and action costs, an amount of XP the unit is worth, current unit status, sprite image.

* + 1. **JMap Class**

The Jmap class creates a 2-dimensional array of ints called MapGrid which correspond to places on the board map. The Jmap class has functions which assign places on the board to the units, and track the location of the units.

* + 1. **Mechanics Class**

The Mechanics class has several functions that perform the actions of the game by manipulating the attributes of the units according to the game rules and player input to produce a result. Functions included in this class include those associated with selecting an action, moving the units, making a unit attack, validating functions to check if actions are allowed at that time, turns between the players, upgrading player units, and initiating a battle.

## AI (Josh)

//DISCUSS CLASSES IN THIS MODULE AND HOW THEY FIT TOGETHER INTERNALLY & WHAT THEY INTERACT WITH FROM OTHER MODULES.

* + 1. AI Engine class

The AIEngine class has modules which handle the functioning of the AI opponent. The memebr functions of this class are responsible for evaluating board position and running the decision matrix which determines the AI’s next move. The AI class calls the existing mechanics functions to perform actions.

## Database (Ye)

//DISCUSS CLASSES IN THIS MODULE AND HOW THEY FIT TOGETHER INTERNALLY & WHAT THEY INTERACT WITH FROM OTHER MODULES.

### Description (Ye)

The database module interacts with other modules, but does not affect them. It shall provide operation function by other modules. Finally, this database module will be added into the entire project through following three steps:

* + - 1. Header file and source file

Only two files will be added into the entire project: “database.h” and “database.cpp”. Beside of defining its own member variables and functions, “database.h” needs to include all of the header definition of global AI classes such as: sprite, unit, player, and user so that such objects can be passed and returned as arguments.

* + - 1. SQLITE database file

A SQLITE database file “gamedata.db3” will be placed in the game execution folder. This file contains a sprites table including all the prototypes of the game sprite whose parameters are used to initialize the game. The file also has a players table that stores the user-created name. Every player name corresponds to one or more storage table to save all the units parameters during the game ongoing. The file is operated dynamically by the database member functions.

* + - 1. SQLITE database file

In order to ensure that the application is linked against the QtSql library, one code line

QT += sql;

must be added to the .pro file.

### Database class (Ye)

Only one database class is designed to be combined into the entire project. This class defines the basic functions to create database tables with specific format, to connect the database file to QT platform, and all other functions that are necessary for other modules calling. Actually the design and application of this class is independent, and just like a black box. Other modules are not necessary to know SQL, but just use one or two code lines to call the database class member functions and pass the arguments. The interaction between database class and UI module, global classes is shown in Figure 3.1.

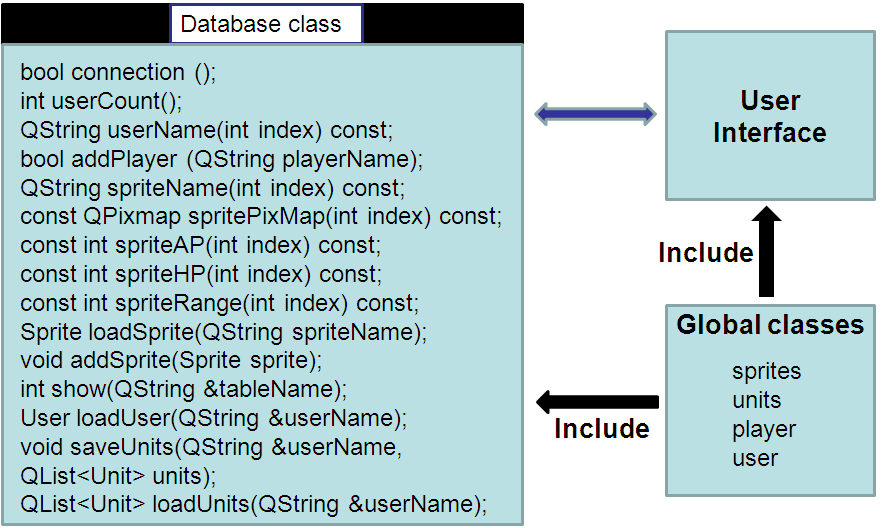


Figure 3.1 Interaction between database class and UI module, global classes

### Test window class (Ye)

Additionally, one test window class is designed to display the database table. In the main function, after some database operations are applied, test window class object calls its member function to display the operated database table. Consequently, the member functions of database class can be verified.

### SQLITE in the Qt platform (Ye)

For better compatibility, only one platform Qt is used to develop all of the modules. The QtSql provides a platform- and database-independent interface for accessing SQL database. A database connection is represented by a QSqlDatabase object. Qt uses drivers to communicate with the various database APIs. Considering SQLite database is a public domain in-process database, we use it.

Qt supplies plenty of classes for operating SQLITE database. For users who are comfortable with SQL syntax, the QSqlQuery class provides a means of directly executing arbitrary SQL statements and handling their results. For users who prefer a higher-level database interface that avoids SQL syntax, QSqlTableModel and QSqlRelationalTableModel provide suitable abstractions.

QVariant class provides good compatibility for mapping the different types of data stored in the database into the corresponding Qt types in QVariants.

# CLASS MEMBER FUNCTIONS

//EXAMPLE SHOWS DISCRIPTION OF EACH OF THE FUNCTIONS UNDER EACH CLASS ALONG WITH A STATE DIAGRAM OF HOW THE CLASS FLOWS, IF THAT’S USEFUL. SO SOMETHING ALONG THOSE LINES HERE. Don’t have to show actual code!!! just describe what it is and does, inputs/outputs wouldn’t be bad either.//

## UI Class Member Functions

## Graphics Member Functions

## Game Mechanics Member Functions

* + 1. **Player Class Member Functions**
       1. *player()*

This function creates the plater object and assignes a new player their starting attributes for XP earned, and level.

* + - 1. *get Team()*

This function asks the player which types of units they want on their team.

* + 1. **Unit Class Member Functions**
       1. *unit()*

This function creates a unit based on the type chosen by the player in the player::getTeam() function;

* + 1. **Jmap Class Member Functions**
       1. *jmap()*

This function creates a new 2-dimention int array the size of the gameboard and initializes the values to zero

* + - 1. *populate()*

This function randomly assignes units to locations on that players side of the board at the start of the battle and returns those locations to the user.

* + - 1. locationUpdate()

This function creates a 2-dimension array of the vertical and horizontal locations for each object on the board.This function is used to track where objects are at any given point in the game and to validate moves and actions.

* + 1. **Mechanics Member Functions**
       1. *mechanics()*

This function loads a set of rules for the battle taking place

* + - 1. *moveUP()*

This function moves the unit up 1 space vertically and if successful, subtracts the appropriate action points.

* + - 1. *moveDOWN()*

This function moves the unit down 1 space vertically and if successful, subtracts the appropriate action points.

* + - 1. *moveLEFT()*

This function moves the unit left 1 space horizontally and if successful, subtracts the appropriate action points.

* + - 1. *moveRIGHT()*

This function moves the unit right 1 space horizontally and if successful, subtracts the appropriate action points.

* + - 1. Move()

This function takes a unit and asks the usr in which direction to attempt to move the unit

* + - 1. isValidMove()

This function determines whether the location to which the player desires to move a given unit to is occupied. If the space is occupied, the move is invalid

* + - 1. isOccupied()

This function determines if a given space on the board is occupied by an object.

* + - 1. isValid attack()
      2. isEnemy()
      3. isGameOver()
      4. isBattleRunning()
      5. attack()
      6. BattleInitiate()
      7. BattleEnd()
      8. SwitchPlayers()
      9. Unit Upgrade()

## Database Member Functions (Ye)

* + 1. **Database Member Functions (Ye)**
       1. *connection ()*

This function establishes a connection with a SQLITE database and creates test tables if necessary. This is the first step for following operations to the database tables. It will return true if connection is successful or false if it is failure unsuccessful.

* + - 1. *userCount()*

This function returns total number of user players’ data in the database player table.

* + - 1. *userName()*

This function returns user player name corresponded to the passed number index argument in the database players table.

* + - 1. *addPlayer ()*

This function adds a new created user player name with the passed string name into the database player table. Before adding, it looks up the table to find matched player name. If the name exists, it returns false, and do not add new player. If not, it adds new player and returns true.

* + - 1. *spriteName()*

This function returns sprite name with the passed number index argument in the database sprites table which provides all of the prototype sprites of the game.

* + - 1. *spritePixMap*

This function returns sprite PixMap with the passed number index argument in the database sprites table.

* + - 1. *spriteAP()*

This function returns sprite AP value with the passed number index argument in the database sprites table.

* + - 1. *spriteHP()*

This function returns sprite HP value with the passed number index argument in the database sprites table.

* + - 1. *spriteRange()*

This function returns sprite range value with the passed number index argument in the database sprites table.

* + - 1. *loadSprite ()*

This function returns a sprite object with the passed string name argument in the database sprites table.

* + - 1. *addSprite ()*

This function adds one sprite with parameters into current user player data table.

* + - 1. *loadUser ()*

This function returns a user player object for the passed string player name argument.

* + - 1. *saveUnits ()*

This function saves array of units of the passed string player name argument into the user player data table.

* + - 1. *loadUnits ()*

This function loads array of units of the passed string player name argument from the user player data table.

* + - 1. *show ()*

This function displays the database table of the passed table name argument in a window. It is used for testing.

* + 1. **Test Window Member Functions (Ye)**
       1. *TableEditor ()*

Only one constructor function is defined in this class. It creates a test window class object with passed string table name. All of the related table data are displayed in the window. This object is initialized in the database member function show().

# TESTING

## Unit Testing

### User interface

### Game Graphics

### Game Mechanics

### AI

### Database (Ye)

The database was tested by inputting a series of coordinates and displaying the data. All of the database member functions and the test window member function were called by a database object in a main function to operate the database file and display the data. All of the functions worked well.

## System Testing

# References (Ye)

* 1. Team Gold: Josh Kilgore, Obi Atueyi, Thomas Calloway, Ye Tian, "Duel Reality: A Turn-Based Battle Strategy Game", Proposal, 02/19/2010.
  2. Team Gold: Josh Kilgore, Obi Atueyi, Thomas Calloway, Ye Tian, "Duel Reality: A Turn-Based Battle Strategy Game", Software Architecture Specification, 03/20/2010.
  3. Team Gold: Josh Kilgore, Obi Atueyi, Thomas Calloway, Ye Tian, "Duel Reality: A Turn-Based Battle Strategy Game", Software Function Specification, 04/13/2010.
  4. Joe Russell, "Coding Guidelines", Software Engineering (16.453/16.553), University of Massachusetts at Lowell.
  5. Jasmin Blanchette, Mark Summerfield, "C++ GUI Programming with Qt 4", Prentice Hall, In association with Trolltech Press, ISBN 0-13-187249-4, First printing: June 2006.
  6. <http://www.sqlite.org>.
  7. <http://www.sqlite.com>.
  8. <http://qt.nokia.com>.
  9. <http://nehe.gamedev.net>.