**Software Functional Specification**

Revision – 0.1

Last printed

TEAM GOLD

**Duel Reality**

**Approval Block**

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

**Abstract (Tom)**

The following is a software design specification document for the Duel Reality personal computer game. The document is meant to convey important software design details via class diagrams, member function descriptions, and other such means. Topics such as the use of coding standards and testing methodologies employed are also discussed. This document is intended to address the design detail of Duel Reality alone, and any high level system or functional requirements for the software are beyond the scope of this document.

**Revision History (Tom)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Revision** | **Date** | **Revised By** | **Comments** |
| 1.0 | 5/13/2010 | Team Gold | Initial Version |
|  |  |  |  |

**Table of Contents (Ye)**

[1. Introduction (Obi) 8](#_Toc261543632)

[1.1. Document Purpose (Tom) 8](#_Toc261543633)

[1.2. Product Scope (Josh) 8](#_Toc261543634)

[1.3. Terminology (Obi) 9](#_Toc261543635)

[1.4. Acronyms (Obi) 9](#_Toc261543636)

[2. Overall Description (Josh) 9](#_Toc261543637)

[2.1. Module Description 11](#_Toc261543638)

[2.1.1. UI (Obi) 11](#_Toc261543639)

[2.1.2. Graphics (Tom) 11](#_Toc261543640)

[2.1.3. Game Mechanics & Artificial Intelligence 11](#_Toc261543641)

[2.1.4. Database (Ye) 12](#_Toc261543642)

[2.2. Design Constraints (Josh) 12](#_Toc261543643)

[2.3. Assumptions and Dependencies (Josh) 12](#_Toc261543644)

[2.4. Design Environment and Tools (Ye) 12](#_Toc261543645)

[3. Software Design (Josh) 12](#_Toc261543646)

[3.1. Basic Classes (Josh) 14](#_Toc261543647)

[3.2. User Interface Classes(Obi) 16](#_Toc261543648)

[3.2.1. Description 16](#_Toc261543649)

[3.3. Game Graphics Classes (Tom) 20](#_Toc261543650)

[3.4. Game Mechanics and Artificial Intelligence (Josh) 21](#_Toc261543651)

[3.5. Database (Ye) 23](#_Toc261543652)

[3.5.1. Description (Ye) 23](#_Toc261543653)

[3.5.2. Database class (Ye) 24](#_Toc261543654)

[3.5.3. Test window class (Ye) 25](#_Toc261543655)

[3.5.4. SQLITE in the Qt platform (Ye) 25](#_Toc261543656)

[4. Class Member Functions 26](#_Toc261543657)

[4.1. UI Class Member Functions 26](#_Toc261543658)

[4.2. Graphics Member Functions (Tom) 26](#_Toc261543659)

[4.2.1. void initializeGL() 26](#_Toc261543660)

[4.2.2. void paintGL() 26](#_Toc261543661)

[4.2.3. void mousePressEvent(QMouseEvent \*event) 27](#_Toc261543662)

[4.2.4. void timerEvent(QTimerEvent \*event) 27](#_Toc261543663)

[4.2.5. void resizeGL(int width, int height) 27](#_Toc261543664)

[4.2.6. void updateUnit(Unit unit) 27](#_Toc261543665)

[4.2.7. void drawGridBox(int vLoc, int hLoc) 27](#_Toc261543666)

[4.2.8. bool isGridBoxSelected(int vLoc, int hLoc) 27](#_Toc261543667)

[4.2.9. void initGrid() 27](#_Toc261543668)

[4.2.10. void drawHeaderInfo() 28](#_Toc261543669)

[4.2.11. void drawGrid() 28](#_Toc261543670)

[4.2.12. void drawUnits() 28](#_Toc261543671)

[4.2.13. void drawBackground() 28](#_Toc261543672)

[4.2.14. void updateTitleScreen() 28](#_Toc261543673)

[4.2.15. void drawEffects() 28](#_Toc261543674)

[4.2.16. void drawAttack() 28](#_Toc261543675)

[4.2.17. void LoadContent(Database db) 28](#_Toc261543676)

[4.2.18. void unitTest\_GenerateContent() 28](#_Toc261543677)

[4.2.19. void unitTest\_AddUnits() 28](#_Toc261543678)

[4.2.20. void moveUnit(int vLocPrev, int hLocPrev, int vLocNext, int hLocNext) 29](#_Toc261543679)

[4.2.21. void hitUnit(int vLoc, int hLoc, int damage, int vAttackerLoc, int hAttackerLoc) 29](#_Toc261543680)

[4.2.22. killUnit(int vLoc, int hLoc, int vAttackerLoc, int hAttackerLoc) 29](#_Toc261543681)

[4.3. Game Mechanics Member Function 29](#_Toc261543682)

[4.3.1. Mechanics Member Functions 29](#_Toc261543683)

[4.4. Database Member Functions (Ye) 32](#_Toc261543684)

[4.4.1. Database Member Functions (Ye) 32](#_Toc261543685)

[4.4.2. Test Window Member Functions (Ye) 35](#_Toc261543686)

[5. TESTING 35](#_Toc261543687)

[5.1. Unit Testing (Obi) 35](#_Toc261543688)

[5.1.1. User interface 35](#_Toc261543689)

[Game Mechanics Modeling 36](#_Toc261543690)

[5.1.2. Game Graphics 37](#_Toc261543691)

[5.1.3. Game Mechanics and Aritificial intelligence 37](#_Toc261543692)

[5.1.4. Database (Ye) 38](#_Toc261543693)

[5.2. System Testing 38](#_Toc261543694)

[6. References (Ye) 38](#_Toc261543695)

**List of Figures (Ye)**

[Figure 2‑1 Program Diagram 10](#_Toc261543696)

[Figure 3‑1 Class Diagram 13](#_Toc261543697)

[Figure 3‑2 Player Class Diagram 14](#_Toc261543698)

[Figure 3‑3 User Class Diagram 14](#_Toc261543699)

[Figure 3‑4 AI Class Diagram 15](#_Toc261543700)

[Figure 3‑5 Unit Class Diagram 15](#_Toc261543701)

[Figure 3‑6 GridBox Class Diagram 16](#_Toc261543702)

[Figure 3‑7 Map Class Diagram 16](#_Toc261543703)

[Figure 3‑8: QMainWindow showing the menu bar, tool bar, status bar and widgets 17](#_Toc261543704)

[Figure 3‑9: QMainWindow Class Structure 19](#_Toc261543705)

[Figure 3‑10: Game Graphics Class Structure 21](#_Toc261543706)

[Figure 3‑11 Mechanics Class Diagram 22](#_Toc261543707)

[Figure 3‑12 Interaction between database class and UI module, global classes (Ye) 25](#_Toc261543708)

# Introduction (Obi)

This document describes the functions of the Duel Reality modules, in accordance with its Architectural Specification 6.2.

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 specify any and all important design details for the development and maintenance 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 (Obi)

Sprite A two-dimensional pre-rendered figure

Unit An individual infantry

Widget An interactive feature pertaining to user interface graphics

## Acronyms (Obi)

AI Artificial Intelligence

AP Action Point

XP Experience Point

GUI Graphical User Interface

UI User Interface

TBS Turn-Based Strategy

# Overall Description (Josh)

Duel Reality is a game comprised of four main parts, or modules. Those modules are shown in relation below in Figure(1) and are called: Visualization, User Interaction, Game Mechanics & AI, and Database. 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 the User Interface takes input from the user and uses that to update the game conditions, which are then output back to the screen and speakers from the Visualization part of the User Interface. The Interactivity portion of the User Interface is responsible for handling player commands like “move here” and “attack this target” and for passing that information to the Game Mechanics module. It is also responsible for acting as the User’s access point for database functions such as the log-in of new and returning players, individual 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, applies the game rules to actions requested by the Player. calculates the internal math between objects, returning the results to the Visualization module to be displayed on the screen. The game mechanics section also handles the Artificial Intelligence gives the player a playable opponent.

The Database section is responsible for retaining the data associated with the players, game settings, and the units. The database has functions which save and load the data to the Game Mechanics and User Interface sections when needed for processing.



Figure ‑ Program Diagram

## Module Description

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

## UI (Obi)

The UI module provides initial interaction with the user at the start of the game. The interaction includes, but is not limited to, loading a game, creating a new game and selecting a team for the battle.

The UI is expected to enable the user fetch the data necessary configuring a battle. The UI is also expected to enable the user store define data, to be retrieved during an active battle.

During an active battle, the user will also interact with the application via the graphics module. Access to the UI’s action items will be limited during an active game battle, in order to make the game more enjoyable. Therefore, the UI’s interface will implement two modes: config mode and battle mode. When in config mode, the user will have the ability to start a new game but will not have access to an “attack” button, for example. In reverse, when in battle mode, the user will have access to an “attack” but will not be able to start a new game.

## Graphics (Tom)

The game graphics module makes visible to the user a region of the main user interface which allows for the displaying of the game graphics while allowing a certain degree of user interaction.

The game graphics module is intended to access relevant battle and user data from external modules and display them appropriately. The game graphics module also manages some game mechanics such as the action points and instantaneous state of every unit in play. The game graphics module also makes available to external modules such as the UI and game mechanics modules a set of functions which control the state of the graphics.

## Game Mechanics & Artificial Intelligence

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, what those entail, and when they are allowed.

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 providing the player a suitable opponent.

## 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 (Josh)

The overall Software class diagram closely mirrors the conceptual module development. The subsequent software classes are shown below in Figure 2. The GLWidget class is where the OpenGL information is drawn on the screen. The window for the Open GL information is housed within the MainWindow. The MainWindow Class is where the user chooses unit action as well as game functions such as Load Game, New Game, and SaveGame. The NewGameWizard and LoadGameWindow allow the Player to make selections about the upcoming game and load that data into the Database.

The Database class handles all of the game data, saving it, and loading it when being processed by thte other classes. The most active class in maniupulating the data is the Mechanics class. This class defines the rules and attributes of actions such as movement and attack. The Mechanics class also defines the rules by which the Artificial Intelligence acts. This class also interfaces with the GLWidget to determine the active units and targets, and the MainWindow Class to determine game state and game over conditions.

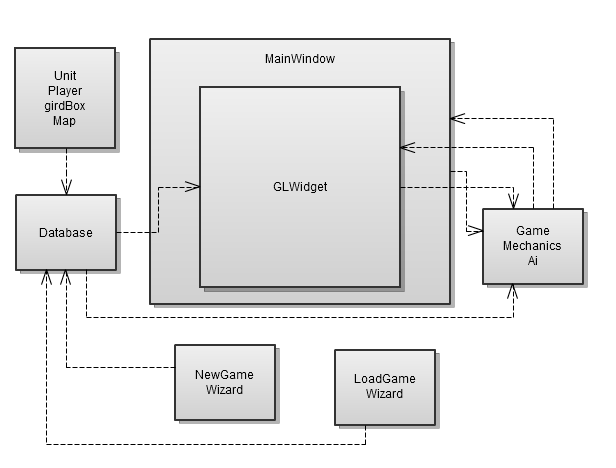


Figure ‑ Class Diagram

## Basic Classes (Josh)

* + 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.

The class diagram for the player class is shown below in Figure XX

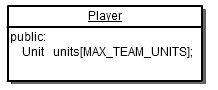


Figure ‑ Player Class Diagram

3.1.1.1 User Class

The User Class is a child of the Player Class and is associated with a human player. This class contains the Player name, accumulated Experience points, whether the Player is active, and the last campaign level played. The class Diagram is shown Below in Figure ??

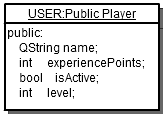


Figure ‑ User Class Diagram

3.1.2.2. The AI class

The AI Class is also a Child of the User Class meant to act as a stand-in for the Player when assigning AI units to a team. The class diagram for the AI class is below in Figure???

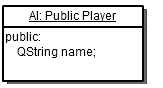


Figure ‑ AI Class Diagram

* + 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 rate, and amount of XP the unit is worth, current unit status, sprite image. The Class diagram for the Unit class is below in Figure ??

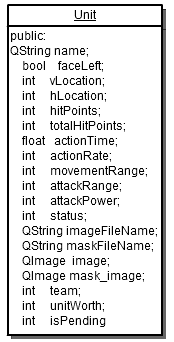


Figure ‑ Unit Class Diagram

* + 1. **GridBox Class**

This class is used to define one square on the game board. Using this class each grid on the board has an associated Unit whose attributes are changed to match whatever unit the data says is occupying that space, as well as if the grid is currently selected by the user. It also has the attributes that Open GL uses to draw the grid and contents. The class diagram for the GridBox class is below in Figure ??

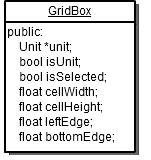


Figure ‑ GridBox Class Diagram

* + 1. **Map Class**

This class defines the overall game board. It consists of a 2-D array of GridBox objects to a desired height and width depending on the background image played. It also contains the image of the background, and the audio file associated with the level. The class diagram for the Map class is below in Figure ??

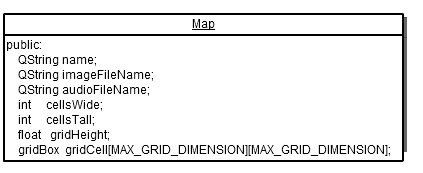


Figure ‑ Map Class Diagram

## User Interface Classes(Obi)

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

### Description

#### MainWindow Class

The User Interface has a main class called the MainWindow. This is the application framework that provides a layout for the user interaction. The MainWindow clas is derived from the Qt-native QMainWindow class. The graphics module will be the central widget for the application. Figure 3‑8 is a graphical illustration of the mainwindow.

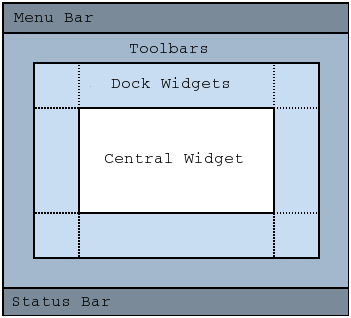


Figure ‑: MainWindow showing the menu bar, tool bar, status bar and widgets

Although the UI implements other Qt-native classes, only the MainWindow object is accessible to the Database and Game mechanics modules. Figure 3‑9 shows the structure of the MainWindow class.

|  |
| --- |
| MainWindow |
| class MainWindow : public QMainWindow  {  Q\_OBJECT  public:  MainWindow();  protected:  void closeEvent(QCloseEvent \*event);  signals:  void signalGameCfgComplete(); // Indicates new game or load game dialog is complete  void isBattleMode(bool); // True if game is in battle mode; false otherwise  void isGameCfgMode(bool); // True if game is in config mode; false otherwise  public slots:  void onBattleStart(); // Disables game config user interactions  void onBattleEnd(); // Disables battle interactions  void onUserTurn(); // Enables user turn interactions  void onMove();  void onAttack();  void onPlayerLost();  private slots:  void newGame(); // Calls New Game dialogs  void about(); // Calls About Game dialogs  void loadGame(); // Calls Load Game dialogs  void exitGame(); // Exits game  void saveGame(); // Calls Save Game dialog  void endTurnClicked(); // Disables user turn interactions  void playHoveredSound();  void playClickedSound();  private:  void createActions(); // Creates actions  void createMenus(); // Creates menus  void createToolBars(); // Creates toolbars  void createStatusBar(); // Creates status bars  // Modules from other team members  QWidget \*centralWidget;  // Action sounds  QSound \*soundBattleStart; // Played at start of battle  // Action items  QAction \*actionNewGame; // Triggered to start new game  QAction \*actionLoadGame; // Triggered to load new game  QAction \*actionSaveGame; // Triggered to save game  QAction \*actionExitGame; // Triggered to exit game  QAction \*turnIndicator; // Indicates player turn (not implemented as an action)  QAction \*actionAttack; // Triggered to attack opponent  QAction \*actionMove; // Triggered to move unit  QAction \*actionAbout; // Triggered to open About dialog  QAction \*actionEndTurn; // Triggered to end user turn  // Menu items  QMenu \*menuGame; // Game menu  QMenu \*menuAction; // Action menu  QMenu \*menuHelp; // Help menu  // Toolbar items  QToolBar \*gameToolBar; // Game toolbar  QToolBar \*actionToolBar; // Action toolbar    Database db;  }; |

Figure ‑: MainWindow Class Structure

## Game Graphics Classes (Tom)

The game graphics module consists of one central class which stores and manipulates the graphics widget itself. This class features several public and protected data classes, variables, and methods which facilitate the required functionality. Figure 3‑10 below illustrates the members and methods which comprise the game graphics class.

|  |
| --- |
| **GLWidget** |
| public:  // Various public methods.  GLWidget();  void LoadContent(Database db);  void unitTest\_GenerateContent();  void unitTest\_AddUnits();  void moveUnit(int vLocPrev, int hLocPrev, int vLocNext, int hLocNext);  void hitUnit(int vLoc, int hLoc, int damage, int vAttackerLoc, int hAttackerLoc);  void killUnit(int vLoc, int hLoc, int vAttackerLoc, int hAttackerLoc);  // Battle state;  bool isBattle;  Map battleMap;  Unit unit[MAX\_MAP\_UNITS];  QString backgroundList[7];  bool isPending;  // Background music.  Phonon::MediaObject \*music;  QString musicTrack;  public slots:  void playBackgroundTrack();  private slots:  void backgroundTrackFinished();  protected:  // Protected member functions.  void initializeGL();  void paintGL();  void mousePressEvent(QMouseEvent \*event);  void timerEvent(QTimerEvent \*event);  void resizeGL(int width, int height);  void updateUnit(Unit unit);  void drawGridBox(int cellFromBottom, int cellFromLeft);  bool isGridBoxSelected(int cellFromBottom, int cellFromLeft);  void initGrid();  void drawHeaderInfo();  void drawGrid();  void drawUnits();  void drawBackground();  // Title screen management.  void updateTitleScreen();  int titleIndex;  // GLWidget content information.  GLfloat statusWidth, fullWidth, fullHeight, cellWidth, cellHeight;  QImage bkImage, glBkImage;  point mouseClick;    // Effect management.  bool isEffect;  int effectType;  int move\_vLocPrev, move\_hLocPrev, move\_vLocNext, move\_hLocNext;  int attack\_vLoc, attack\_hLoc;  void drawEffects();  int iEventCounter;  void drawAttack();  QString moveMask; |

Figure ‑: Game Graphics Class Structure

## Game Mechanics and Artificial Intelligence (Josh)

The Game Mechanics module has several different and distinct classes that handle the manipulation of game data. The class diagram is shown below in Figure 3. The communication between this module and the other modules come form the Signals and Slots which pass signals to the Main Window, and the individual member functions, a lot of which access information in the GLWidget class such as Unit health or location at a given coordinate.

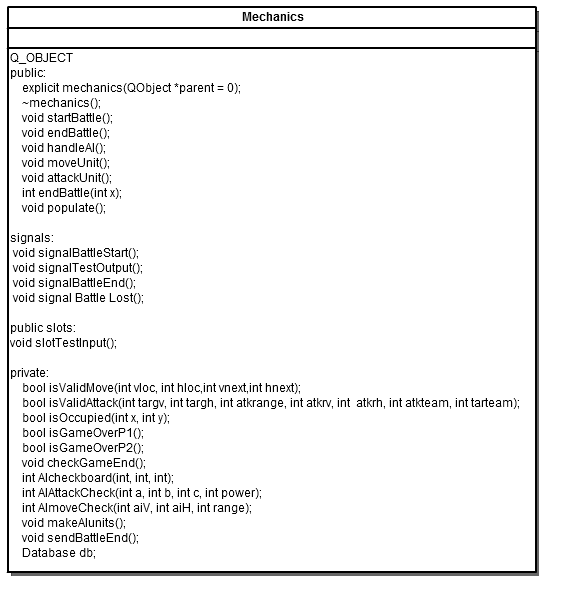


Figure ‑ Mechanics Class Diagram

* + 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 unit initial placement, unit movement, unit attacking, the direction that the unit is facing, validating functions to check if actions are allowed at that time, turns between the players, and handling game victory and defeat.

* + 1. **Artificial Intelligence**

The artificial intelligence for this game is located in the Mechanics class. consists of functions that simulate a human opponent for the player. The memebr functions are responsible for evaluating board position of the various units and and running the decision matrix which determines the AI’s next move, absed on a rating for the associated action. For instance, the AI will attack as soon as it is in range of an enemy, and will focus on chasing that enemy if it is lower than 50% health. The AI class calls the existing functions to perform moving and attacking functions.

## 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 user 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. Project 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.

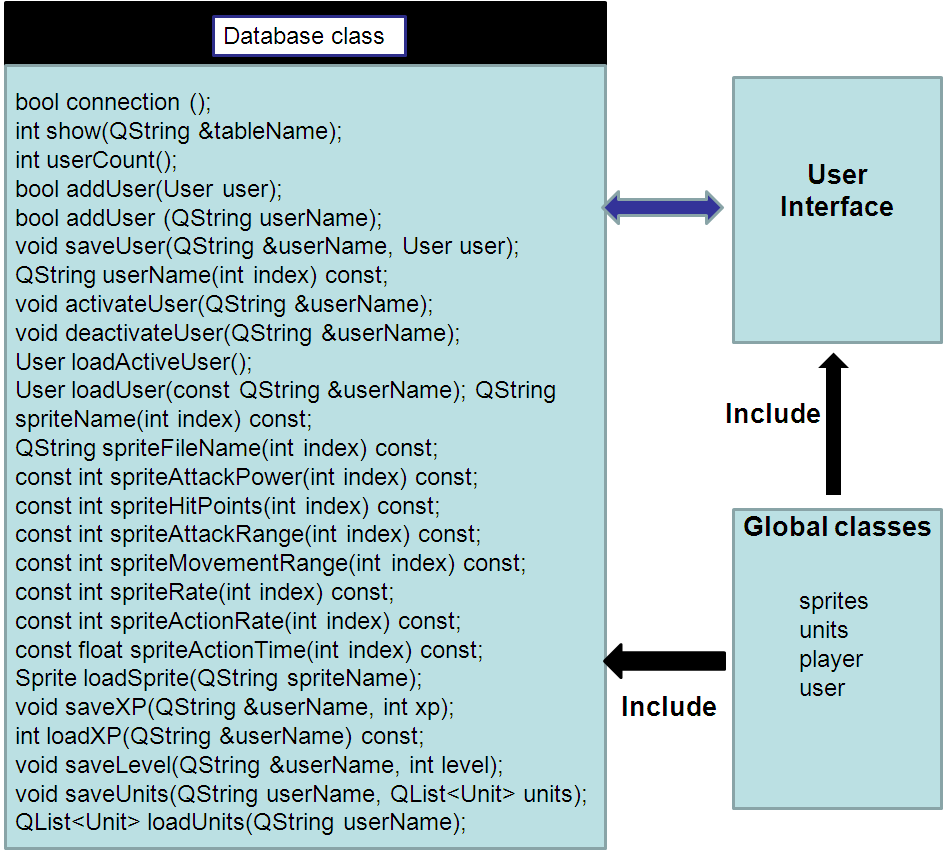


Figure ‑ Interaction between database class and UI module, global classes (Ye)

### 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 [5].

SQLite is an in-process library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. The code for SQLite is in the public domain and is thus free for use for any purpose, commercial or private. SQLite is currently found in more applications than we can count, including several high-profile projects [6].

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 [7].

# 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 (Obi)

The following is a list of MainWindow member functions:

### void closeEvent(QCloseEvent \*event)

This implements the QCloseEvent class which describes the parameters for closing an event. For the MainWindow, closeEvent is called when the user clicks on the close button on the top-right corner of the window, or when the user chooses “exit” from the menu.

### void signalGameConfigComplete()

This function is called after the user completes the dialogs for loading or saving a new game. It is within this function that the Game Mechanics is triggered to prepare the application for a battle scene.

### void onBattleStart()

This function is called when the application starts a battle scene. When in a battle scene, the function disables user accesses that are only permissible in game configuration mode. It also enables user accesses that are only permissible during battle mode.

### void onBattleEnd()

This function is called when the application returns from a battle scene to a game configuration setting. During game configuration, the function disables user accesses that are only permissible in battle mode. It also enables user accesses that are only permissible during game configuration mode.

### void onMove()

The onMove() function is called during battle when a user clicks on the “move” button. This function calls a member function in the Game Mechanics module to initiate a reaction to the user’s move request.

### void onAttack()

The onAttack() function is called during battle when a user clicks on the “attack” button. This function calls a member function in the Game Mechanics module to initiate a reaction to the user’s attack request.

### void onPlayerLost()

This function is called at the end of a battle if the user lost the battle. It displays a message to the user and offers the option to either start a new game or exit the progam.

### void NewGame()

This function is available when the application is in game configuration mode. When the user selects “New Game” from the menu options or from the tool bar, the NewGame() function creates the New Game dialogs required for new game configuration.

## Graphics Member Functions (Tom)

The following is a list of the functions which comprise the “glWidget” class responsible for the game graphics. The purpose of each function is given as well as the various inputs, outputs and return values (where they exist).

### void initializeGL()

This function initializes the OpenGL graphics widget by setting the various colors, shading, and blending options.

### void paintGL()

This is the primary painting function for the OpenGL widget. All of the miscellaneous drawing functions are called from here. This function is called at the specified update rate (10 Hz by default).

### void mousePressEvent(QMouseEvent \*event)

This function is triggered whenever a click even occurs on the OpenGL widget itself. If a battle is in progress, the function captures the mouse coordinates at the time of the event and makes them available to the rest of the class.

* **Inputs**: the event which includes the coordinates of the mouse

### void timerEvent(QTimerEvent \*event)

This function repaints the OpenGL widget whenever the timer triggers.

* **Inputs**: the event itself (unused)

### void resizeGL(int width, int height)

This function triggers whenever the OpenGL widget is resized. It modifies the dimensions of individual graphics components and re-initializes the drawing state.

* **Inputs**: new width and height of the graphics widget

### void updateUnit(Unit unit)

This is the function which draws individual units. It draws the transparent sprite along with the associated health and action bars.

* **Inputs**: an instance of the unit requesting to be drawn

### void drawGridBox(int vLoc, int hLoc)

This function draws an individual grid cell.

* **Inputs**: Coordinates of the grid cell to be drawn

### bool isGridBoxSelected(int vLoc, int hLoc)

This function determines whether or not a grid cell is selected based on the mouse coordinates.

* **Inputs**: Grid cell in question
* **Return Value**: Whether or not the cell in question is selected

### void initGrid()

This function initializes all of the current battle map grid cells with correct (initial) values in order to avoid fault conditions caused by uninitialized data.

### void drawHeaderInfo()

This function draws the header region for any pending or manually selected units.

### void drawGrid()

This function loops through all of the existed grid boxes and calls a function to draw them individually.

### void drawUnits()

This function loops through all of the units in play, updates appropriate pointers, and calls a function which draws them individually.

### void drawBackground()

This function draws the static background. This is generally the first item drawn because it is designed to be behind all other objects.

### void updateTitleScreen()

This function toggles through various battle map backgrounds and displays a "Duel Reality" logo across the top.

### void drawEffects()

This function features a large switch statement which handles the appropriate special effect in progress.

### void drawAttack()

This function handles and draws the special effect for the attack function.

### void LoadContent(Database db)

This function loads the battle map content passed from the user interface (with some hard coded).

### void unitTest\_GenerateContent()

This function loads the battle map content passed from the user interface (with some hard coded).

### void unitTest\_AddUnits()

This function creates and initializes eight “dummy” units for unit test purposes.

### void moveUnit(int vLocPrev, int hLocPrev, int vLocNext, int hLocNext)

This function handles the "move unit" event triggered by the UI and managed by the mechanics modules.

* **Inputs**: previous and new coordinates of the unit in question

### void hitUnit(int vLoc, int hLoc, int damage, int vAttackerLoc, int hAttackerLoc)

This function handles the "hit unit" event triggered by the UI and managed by the mechanics modules.

* **Inputs:** Attacker and victim coordinates as well as the damage to be done

### killUnit(int vLoc, int hLoc, int vAttackerLoc, int hAttackerLoc)

This function handles the "kill unit" event triggered by the UI and managed by the mechanics modules.

* **Inputs:** Attacker and victim coordinates

## Game Mechanics Member Function

### Mechanics Member Functions

* + - 1. *makeAIunits()*

This function creates an Ai User, assigns AI units, and makes those units available to the game for drawing

* + - 1. *handleAI()*

This function is called when an AI unit is called on to perform an action. It looks for the AI unit which is supposed to act, evaluate its position and the position of all units around it and then executes either a move or an attack based on its logic.

* + - 1. *AIAttackCheck()*

This function evaluates the spaces on the board so that the AI unit can attack an enemy unit. If a nearby unit is occupied by the Player, and the attack move is valid, the AI unit attacks. Based on relative position of the attacker and target the AI unit may turn right or left.

* + - 1. *AImoveCheck()*

This function evaluates the spaces on the board so that the AI unit can determine where to move next. It takes note of where all the Player and Ai units are, and the health of the Player units and based on the wieght asociated with those conditions and the distance to the relative targets, choses which direction to head. Based on the direction of travel, the unit may turn left or right.

* + - 1. *Move()*

This function is the basic movement action. It is called by the Player triggering the move button from the main window. It finds the Player unit’s position, the position of the selected destination, calculates whether it is a valid move, and if so, calls the visualization mdule to update the unit’s position. This shows up on screen as a unit moving. Based on the direction of travel, the unit may turn left or right.

* + - 1. *isValidMove()*

This function is called by both the Player and AI movement functions to determine if a selected move is valid. The criteria of a valid move involve the range of the unit moving, whether the selected space is occupied, and whether the space is in a straight line from the unit.

* + - 1. *attackUnit()*

This function is responsible for execuing an attack on another unit. It is called by the player selecting the attack button on the main window. It finds the player’s location, the location of the selected target, tests whether it is a valid attack, and calls the visualization module to execute either a hit or kill based on how close the target is to running out of health. *Based on the direction of travel, the unit may turn left or right.*

* + - 1. *isValidattack()*

This function determines if a selected attack is valid. It takes into consideration the distance between the units, the attacker’s range, the teams of the units, and whether the selected location is occupied. It returns either true or false depending on the conditions.

* + - 1. *checkEndGame()*

This function is called after every time a unit is killed to check the game board to determine if either Player 1 or AI units are all dead. If so it loads the correct screen depending on if the Player won or lost.

* + - 1. *isOccupied()*

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

* + - 1. *isGameOverP1()*

This function is responsible for looking through the spaces on the board and returning whether there are any of Player 1’s units alive.

* + - 1. *isGameOverP2()*

This function is responsible for looking through the spaces on the board and returning whether there are any of Player 2’s units alive.

* + - 1. *EndBattle();*

This Function calls the isGameOverP1 function, if that returns true, calls the isGameOverP2 function. If either returns false, calls the appropriate function to signal MainWindow which end state dialog to run.

* + - 1. *sendPlayerLost()*

This function, when called, emits the signal called signalPlayerLost, which is connected to the onPlayerLost slot in the MainWindow class.

* + - 1. *sendBattleEnd()*

This function, when called, emits the signal called signalBattleEnd(), which is connected to the onBattleEnd slot in the MainWindow class. This signals that Player 1 has won the game.

* + - 1. *populate()*

This function creates a two-dimensional array which is loaded with random numbers which will be used to initially position Player, AI, and any obstacle pieces on the board. The function is designed to start the Player 1 on the left side of the board, the AI on the right, and the obstacles anywhere on the board. It automatically adjusts to the dimensions of the board loaded from the database.

* + - 1. *slotTestinput()*

This funciton is called when the slot slotTestInput is called by the signal signalGameCfgComplete() is called from the MainWindow class. This signal tells the mechanics class that the game configuration is complete and to set up the game.

* + - 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.

## Database Member Functions (Ye)

### Database Member Functions (Ye)

* + - 1. *bool 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. *int show(QString &tableName)*

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

* + - 1. *int userCount()*

This function returns total number of users data in the database player table.

* + - 1. *bool addUser(User user)*

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

* + - 1. *bool addUser(QString userName)*

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

* + - 1. *void saveUser(QString &userName, User user)*

This function adds a user data with the passed string name and user class into the database user table.

* + - 1. *QString userName(int index) const*

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

* + - 1. *void activateUser(QString &userName)*

This function activates a user with the passed string name. The corresponding value in the database will be set true.

* + - 1. *void deactivateUser(QString &userName)*

This function deactivates a user with the passed string name. The corresponding value in the database will be set false.

* + - 1. *user loadActiveUser()*

This function scans the user table and loads active user data into a user object.

* + - 1. *user loadUser(const QString &userName)*

This function loads user data with the passed string user name into a user object.

* + - 1. *QString spriteName(int index) const*

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. *QString spriteFileName(int index) const*

This function returns sprite picture file name with the passed number index argument in the database sprites table.

* + - 1. *const int spriteAttackPower(int index) const*

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

* + - 1. *const int spriteHitPoints(int index) const*

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

* + - 1. *const int spriteAttackRange(int index) const*

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

* + - 1. *const int spriteMovementRange(int index) const*

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

* + - 1. *const int spriteRate(int index) const*

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

* + - 1. *const int spriteActionRate(int index) const*

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

* + - 1. *const float spriteActionTime(int index) const*

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

* + - 1. *sprite loadSprite(QString spriteName)*

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

* + - 1. *void saveXP(QString &userName, int xp)*

This function saves the current user experience points with the passed string user name and experience points value arguments into the database user table.

* + - 1. *int loadXP(QString &userName) const*

This function loads the user experience points with the passed string user name argument.

* + - 1. *void saveLevel(QString &userName, int level)*

This function saves the current user level with the passed string user name and level value arguments into the database user table.

* + - 1. *void saveUnits(QString userName, QList<Unit> units)*

This function saves array of unit data of the passed string user name and unit objects array arguments into the user table.

* + - 1. *QList<Unit> loadUnits(QString userName)*

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

### 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.

# TESTING

## Unit Testing (Obi)

### User interface

Unit testing will be performed to demonstrate the communications between the UI and the Game Mechanics; and between the UI and the database.

#### Database Modeling

The UI's interaction with the database involves fetching and updating user data. The database model for the unit test will be a class with static member variables and public methods that will be identical to the database in the final system. By having static member variables, the model will mimic the database as long as the program is running. This will satisfy the black box concept, in that the actual implementation of the database is not transparent to the UI. The static member variables will be:

##### User List

This will be a QList class that stores the User class data. At program start, the model will generate and add test users to this list. The test users will represent user data from previously saved games. After a new game is configured, new user data will be passed to the database model, to be added to this list.

##### Sprite List

This will be QList class that stores the sprite data. At program start, the model will generate and add sprites to the list. The sprite list will represent sprite selections used by a player to recruit units for a game.

Map List

This will be a QList class used for the storage of map data. At program start, the model will generate and add maps to the list. The maps will represent maps selections used by the player to select a battle map during game configurations.

### Game Mechanics Modeling

After a user completes a new game or load game configuration, the UI calls a method in the Game Mechanics to prepare the application for a battle. The Game Mechanics then loads the user data that was previously created by the UI and stored in the database. The Game Mechanics then notifies the UI when the application is ready to load a battle. This interaction between the UI and the Game Mechanics will be demonstrated using signals and slots. Signals and slots are a feature native to Qt and they are used for interactions between objects. Signals act as a trigger mechanism which slots are methods that react to signals. The signals and slots that will be implemented are:

##### signalBattleStart(), slotOnBattleStart()

After the user completes a new game or load game configuration, the UI will emit its signalBattleStart(), which will then trigger the slotOnBattleStart() in the Game Mechanics class.

##### signalBattleEnd(), slotOnBattleEnd()

After the battle is over, the Game Mechanics will emit its signalBattleEnd(), which will then trigger the slotOnBattleEnd() in the UI class.

For the purpose of unit testing, the slotBattleStart() method in the Game Mechanics will simply emit its signalBattleEnd, thereby providing a signal look back the the UI

### Game Graphics

The game graphics module is unit tested through the use of two unit test functions which were designed to create generic content without the need for a database connection. Testing of the graphics module consisted of modifying the contents of those two functions and running the code in order to verify the results. User interaction was tested without the use of additional classes as it was not necessary. The two generated functions are listed below.

* void unitTest\_GenerateContent();
* void unitTest\_AddUnits();

### Game Mechanics and Aritificial intelligence

The Game mechanics were initially coded and tested in a console version, adding complexity as it was verified to work to a satisfactory degree. Efforts focused on unit assignment, positioning, changing position in a 2-dimension array of objects, and attacking.

Much of the code for the mechanics class had to be rewritten for the system testing. Once the basic actions of move and attack were successfully tested and how they interacted with the unit objects’ attributes, it was easy to scale complexity.

The Artificial Intelligence was written and tested as an automatic call of the move and attack functions without user input. Trial and error were used to grow the AI from simply moving between squares to chosing direction and attacking the Player in accordance to its own goal-driven behavior.

### 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>.