**Battleship [A desktop game]**

**Project Report**

***Developed by:***

***Kunjal Panchal [Roll No. 562, Seat and Examination No. 604029]***

***Viral Odedara [Roll No. 404, Seat and Examination No. 604027]***

***Under the guidance of:***

***Dr. Anjali G. Jivani***

***Department of Computer Science & Engineering***

***[Professor for Database Management System]***

***Dr. Mamta C. Padole***

***Department of Computer Science & Engineering***

***[Professor for Java and Graphical User Interface concepts]***

**April/May, 2018**

****

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING,**

**MAHARAJA SAYAJIRAO UNIVERSITY, VADODARA – 1,**

**GUJARAT, INDIA.**

**TABLE OF CONTENTS:**

|  |  |  |
| --- | --- | --- |
| No. | Topic | Page |
| 1 | Introduction | **3** |
| 2 | Software Specification | **5** |
| 3 | Project Plan | **7** |
| 4 | Technologies | **10** |
| 5 | Design Strategies | **10** |
| 6 | Diagrams | **12** |
| 7 | Data Dictionary | **16** |
| 8 | Screenshots | **17** |
| 9 | Source Code | **22** |
| 10 | Conclusion | **28** |

1. **Introduction**

**1.1 Definition:**

Battleship is a well-known player-versus-player and/or player-versus-computer game which is a classic first step for learning game development with JAVA/C++ and front-end GUI.

**Battleship** (also **Battleships** or **Sea Battle**) is a guessing game for two players. It is played on ruled grids (paper or board) on which the players' fleets of ships (including battleships) are marked. The locations of the fleet are concealed from the other player. Players alternate turns calling "shots" at the other player's ships, and the objective of the game is to destroy the opposing player's fleet.

**1.2 Terminology and Game Objectives:**

The game is played on two grids, one for each player. The grids are typically square – usually 10×10 (in our project, 9×9) – and the individual squares in the grid are identified by letter and number. On one grid the player arranges ships and shots by the opponent on that grid are recorded. On the other grid, vice versa.

Before play begins, each player secretly arranges their ships on their primary grid. Each ship occupies a number of consecutive squares on the grid, arranged either horizontally or vertically (diagonal arrangement is not possible and in contrary to the rules). The number of squares for each ship is determined by the type of the ship. The ships cannot overlap (i.e., only one ship can occupy any given square in the grid). The types and numbers of ships allowed are the same for each player. These may vary depending on the rules.

For the project in spotlight:

|  |  |  |
| --- | --- | --- |
| **#** | **Class of ship** | **Size** |
| 1 | Carrier | 5 |
| 2 | Battleship | 4 |
| 3 | Cruiser | 3 |
| 4 | Yacht | 3 |
| 5 | Smallboat | 2 |

After the ships have been positioned, the game proceeds in a series of rounds. In each round, each player takes a turn to announce a target square in the opponent's grid which is to be shot at. The opponent will be announced whether the square is occupied by a ship, and if it is a "miss", the opponent player’s primary grid will be marked by a distinct colour (white/ dull grey); if a "hit" it will be marked on the primary grid with a red color. Player will be able to see the status of his own grid in a small view area and on the prime area of the screen will be the opponent’s board which is to be attacked.

When all of the squares of a ship have been hit, the ship is sunk. If all of a player's ships have been sunk, the game is over and their opponent wins.

**1.3 Deliverables:**

The first iteration will be a primitive account creating (gamer ID) and playing against computer game. The next iterations will add ability to play against real players through online connectivity and search for a specific friend to play with.

Another basic thing which may not be available in first iteration is the graphics. The game will be flamboyant and more animated once the basic structure is built.

1. **Software Specification**

**2.1 Product Perspective:**

The purpose of this product is to add one more installation in the pool of basic board games implemented through basic coding techniques. A player can compete with another player online or can play against computer in the offline mode. The player career progress will be saved in server-side database every time the round is finished.

This is a classic turn-based game implements through threading where each player gets to make a move on opponent’s board and the turn will be decided from the outcome of the current move.

**2.2 User Characteristics:**

The players or users we are targeting may not be well-acquainted with the basics of computer workings. The game has made sure that the GUI is pretty intuitive with few tips and guidelines here and there to help player lead through the game.

Besides, the game provides a detailed instruction page which will fill the gaps for an user who is first time playing this type of game.

**2.3 Specific Requirements:**

- The user computer must have JRE 1.8 installed as the game is made with JDK 1.8.

- The database as of now is on local server but when it will be online, the user must have a working internet connection in order to log in and play the game. Offline rounds will not make an entry to the player career.

The whole internal working will be abstracted from the actual users of the product and it is of no importance for them to understand the specific requirements of the programs executing inside their devices.

- The game uses Swing Components, so the different operating systems will generate the same look and feel.

- The game is in non-resizable windowed mode and ideal for 1280 x 720 resolutions. Abnormal screen maybe displayed if the screen size is smaller than preferred size.

**2.4 Performance Requirements:**

- The game implements multithreading for main gameplay and soundtracks. Therefore, the peak performance will be acquired if the game is played on multiprocessor environment.

- The game implements basic code for game logic and queries on database, it can use 200 Mb of RAM [on a 64-bit multiprocessor].

- Each time it will load soundtracks and images whenever called. The audio stream acquisition time may depend on how long the tracks are and the speed of buffering.

**2.5 Design Constraints:**

- The Battleship game is made purely in JAVA. Therefore, we haven’t implemented graphically extraordinary effects and animations.

- For player vs player environment, the networking part is yet to be added as we are still learning and yet to get proper knowledge about how things work when online on a network.

- Algorithm for Computer [opponent player] is totally randomized and is on “easy mode” with no strategy given to it. The computer will randomly generate any co-ordinate and check for its validity, if valid the move can be made.

1. **Project Plan**

The game is divided in the following modules, each providing an abstraction and an interface for outside code or other modules’ interfaces. Each module can be subdivided into the GUI part, event handling part and the core logic [the functionality it need to perform]. Here we will show the modules through a higher level of abstraction to give a glimpse about each modules’ inner working and the purpose.

**NOTE**: The screenshots provided in topic 8 and diagrams provided in topic 6 can give some idea of how the modules will work and interconnect with each other.

**3.1 Main Menu Screen:**

The main menu has tabs:

* Play [only activated when player logs in the game]
* Leader-boards
* Settings [has the following tabs to customize your game]

- Change Account

- Delete Account

- Instructions

- Change Theme

- Previous Matches

- Volume Control

- Credits

- Sign up [a new player has to sign up first]

- Log In

- Exit

It is the first screen player will confront when opening the instance of a game. This has for the most part, GUI and Event Handling mechanisms. Every action will call or create methods and objects of some other class.

**3.2 Board Creation Screen:**

Once the play button is activated, the player clicks on the play button and navigate to the board creation screen. It will create a new object which does all the stuff related to a new board.

The board is basically a panel which has a 9 x 9 grid layout and each cell has a button, which represents one atomic block of a sea which can contain one block of a ship if player chooses to put ship’s one part on the area.

Player can choose not to play a round just by clicking on “back”, the panel will be reseted.

Unless the player puts all 5 ships on the board, s/he won’t be able to move to the main game screen.

All 5 ships has a different colour code to know which ship is where. Image graphics won’t be shown for the ships on sea panel.

**3.3 Play Screen:**

- Play screen will have 2 board panels: one which the player made by putting whole fleet on his/her area of sea, another which is computer generated which the player will attack, and try to eliminate opponent’s whole fleet.

- Player will not be able to see which block contained a ship, unless s/he attacks that block, it can either be water or one of 5 ships.

- Besides 2 board panels, the screen will have a timer which records the time passed playing this round.

- The player will be able to see own fleet status; how much each ship has been damaged by the opponent.

- First one to destroy whole fleet will the winner and the screen changes to “Game Over” screen.

- Player can only interact with the opponent’s board, and only his/her turn. If any player hits water in its turn, the turn will pass to the other player, if a ship has been hit, one more turn will be awarded to the current player.

**3.4 Game Over Screen:**

- Once the game is finished, all stats about the round will be inserted into database and displayed on the screen.

- The next screen will be again the “Main Menu Screen” from where you can start a new round.

**3.5 Error Dialog Boxes:**

- The errors include connection to database and all other DDL, DML, DRL related queries which can occur if database is not connected or the query statements or inserted values are wrong and don’t match the predefined constraints.

- Error Dialog Boxes are also used to show InterruptedException or IllegalStateException in threading.

**3.6 Warning Dialog Boxes:**

Exiting the current round prematurely or exiting the game will first show this dialog box to confirm the actions.

**3.7 Sound Manager:**

Manages all audio streams which are independent threads.

- The button click sounds.

- The background soundtrack.

- The sounds of bomb hitting ships or water.

**3.8 Database Manager:**

As the name suggests, this class will manage all the database interactions.

* Adds new player at the time of sign up.
* Checks validity of the player at the time of log in.
* Generates the updated leader-boards every time the leader-board tab is clicked.
* Updates player and match data every time a round has terminated.

**3.9 Event Manager:**

Separated from the GUI code class, these classes manages all kinds of events that can be generated with the given GUI components.

**3.10 Multithreading:**

- Implemented for managing the turn based play.

- Soundtracks [see Sound Manager section]

1. **Technologies**

As stated in the 2nd topic, the main intent was to make a pure JAVA based software.

* The game is implemented only in java, with additional .jar files like absolute layout and database connection for mySql.

Database resides on localhost server and is written in mySql.

1. **Design Strategies**
   1. **Approach:**

The design phase consists of identifying all the objects / entities and then apply a graphical user interface, event handlers for the components the GUI consists of – which will generate events when user interacts with them or they can be self-generating through some inner trigger fired. There’s a third layer which has the code – the business logic – which will be called through the event handlers.

Here, by “layers”, we mean different classes in object oriented. Each performing the specific operations and having attributed needed for operating.

A different and independent dbHandler class exists which only concerns itself with modifying the database with the data given and retrieving the asked data.

Another independent layer will be the SoundManager which provides threaded soundtrack for some events. It neither concerns itself with GUI or data, just a call to a specific function when some event is generated will provide the audio streams.

* 1. **Presentation Layer:**

This is our Graphical User Interface layer which consists purely of AWT or Swing components and the layouts; the bounds and constraints related to each and the initializing and packing of all the containers and components.

This is where “events are generated”, but not handled as it is the job totally for a different layer. So, this class consists of event sources and the look and feel of how a user will sees the software.

In our game, MainMenScreen, Board, GameOver are few of the GUI classes.

* 1. **Event Handling Layer:**

This layer/class implements all the “event listener interfaces” needed to handle the event generated by the GUI sources. It implements all the method declared in the listeners depending upon which listener we are talking about.

If a function for an event has to be overridden but not used then it has simply no code in it.

To not prolong a class and create a chaotic implementation, we have provided few different classes for handling events of a specific GUI class only.

eg. EventHandler, DialogEventHandler are few of the event handler classes in our game.

* 1. **Business Layer:**

This is the chief part of our game; a game can have minor GUI components and their simple event handlers but the logic behind every move and computer algorithm makes a large fraction of the game code.

Player abstract class declares few methods which must be defined by its concrete class. One method of special attention will be the createBoard of computer side function.

Here, a randomly generated board of 5 ships is returned while the real player will not define this method at all because the board will already be created.

The moves when it’s computer’s turn are also randomly generated by an algorithm.

* 1. **Database Layer:**

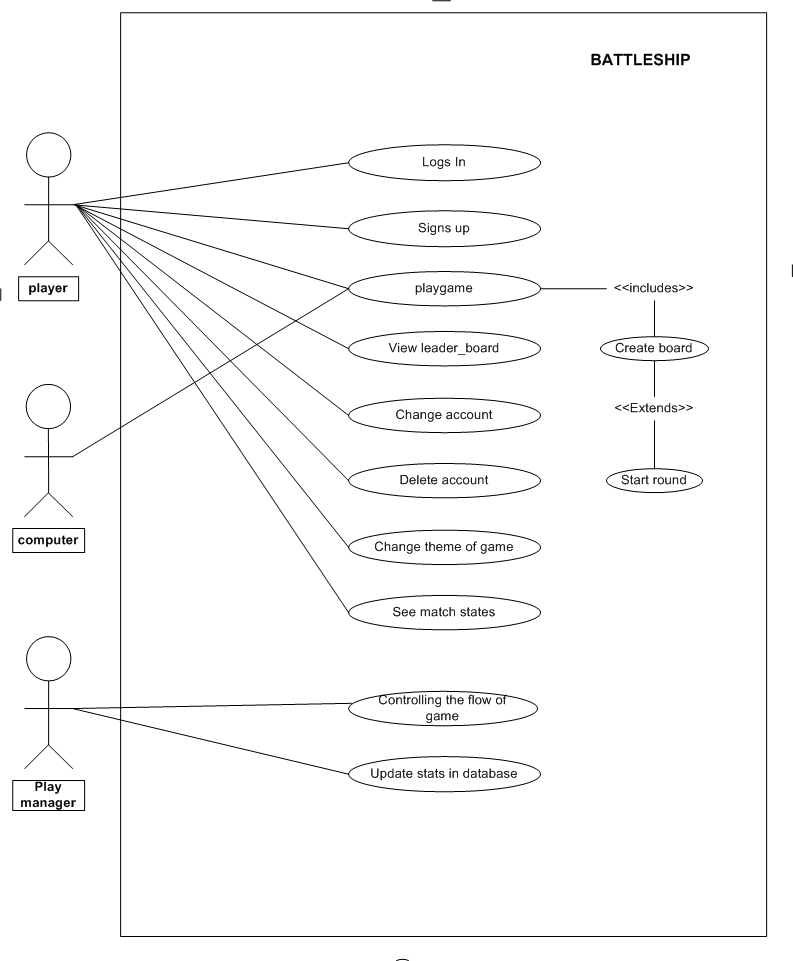
This layer interacts with business layer because the data resulted from the algorithmic calculation and operations can be saved to the database or data fetched from a database can be used to display some stats or detail about matches.

It has a separate class called “dbhandler” which provides functions for every query, insert, update and delete.

It also checks validity of the data before it can be entered into database through some different functions.

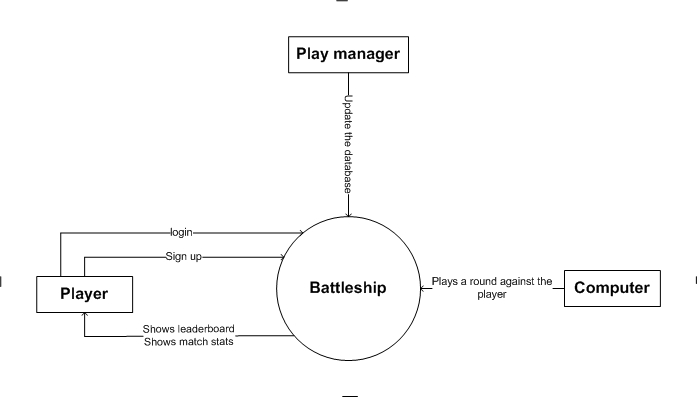
1. **Diagrams**

**Use Case Diagram:**

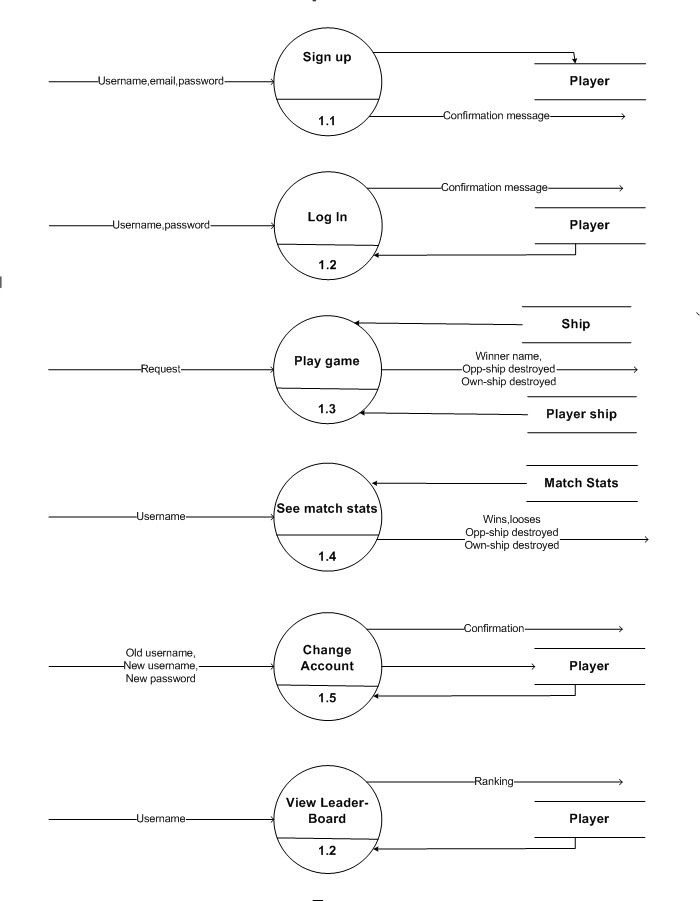


**DFD:**

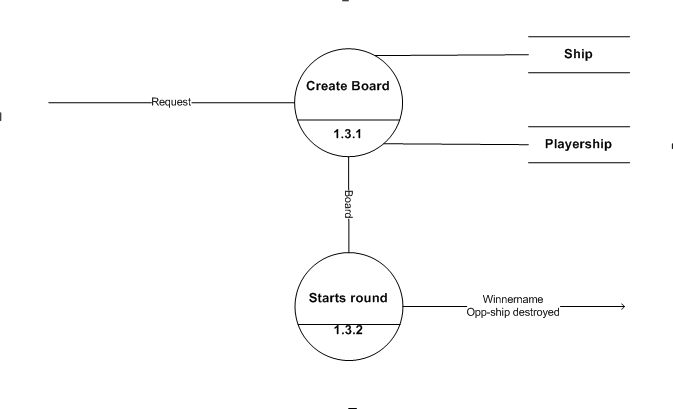
**<Context Level>**



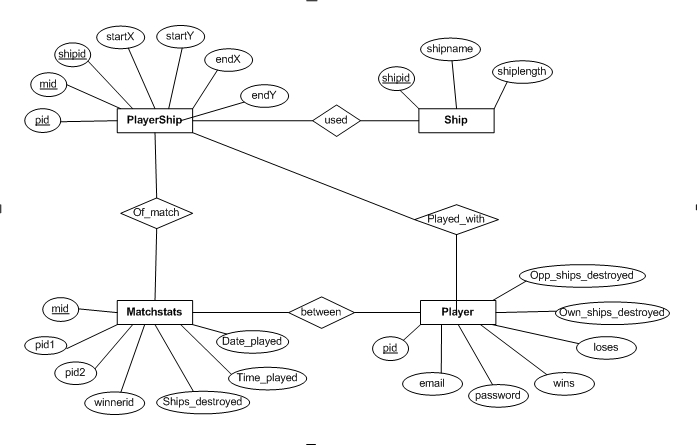
**<Level-1>**



**<Level-2>**



**E-R Diagram**



1. **Data Dictionary**
2. Ship

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Column Name | Data Type | Length | Constraint | Description |
| ship\_id | decimal | 2, 0 | Primary key | Unique ship id |
| shipName | varchar | 15 | Not null |  |
| shipLength | decimal | 1, 0 | check not negative |  |

1. PlayerShip

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Column Name | Data Type | Length | Constraint | Description |
| mid | decimal | 3, 0 | Primary key | Combined along with pid and shipid |
| pid | varchar | 15 | Foreign Key | to the player table |
| shipid | decimal | 2, 0 | Foreign Key | to the ship table |
| noOfHits | decimal | 1, 0 |  | number of ship boxes destroyed out of its length |
| startX | int | 1 |  | ship starting block in X co-ord |
| startY | int | 1 |  | ship starting block in Y co-ord |
| endX | int | 1 |  | ship ending block in X co-ord |
| endY | int | 1 |  | ship ending block in Y co-ord |

1. Player

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Column Name | Data Type | Length | Constraint | Description |
| pid | varchar | 15 | Primary key | Unique player id |
| email | varchar | 254 |  | player email id |
| password | varchar | 20 |  | player password |
| wins | decimal | 3, 0 |  | number of career wins |
| loses | decimal | 3, 0 |  | number of career loses |
| own\_ships\_destroyed | decimal | 3, 0 |  | number of own ships which got destroyed in whole career |
| opp\_ships\_destroyed | decimal | 3, 0 |  | number of opponenet ships which the player destroyed in whole career |

1. MatchStats

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Column Name | Data Type | Length | Constraint | Description |
| mid | decimal | 3, 0 | Primary key | Unique match statistic id |
| pid1 | varchar | 15 | Foreign Key | to the player table |
| pid2 | varchar | 15 | Foreign Key | to the player table |
| winnerid | varchar | 15 |  | the user name of the winner |
| ships\_destroyed\_pid1 | decimal | 1, 0 |  |  |
| ships\_destroyed\_pid2 | decimal | 1, 0 |  |  |
| time\_played | varchar | 5 |  |  |
| date\_played | datetime |  |  |  |

1. **Screenshots**



Fig: Main Menu Screen



Fig: Main Menu Screen – changed Theme



Fig: Main Menu Screen – Leaderboard Panel



Fig: Main Menu Screen – Sign up Panel

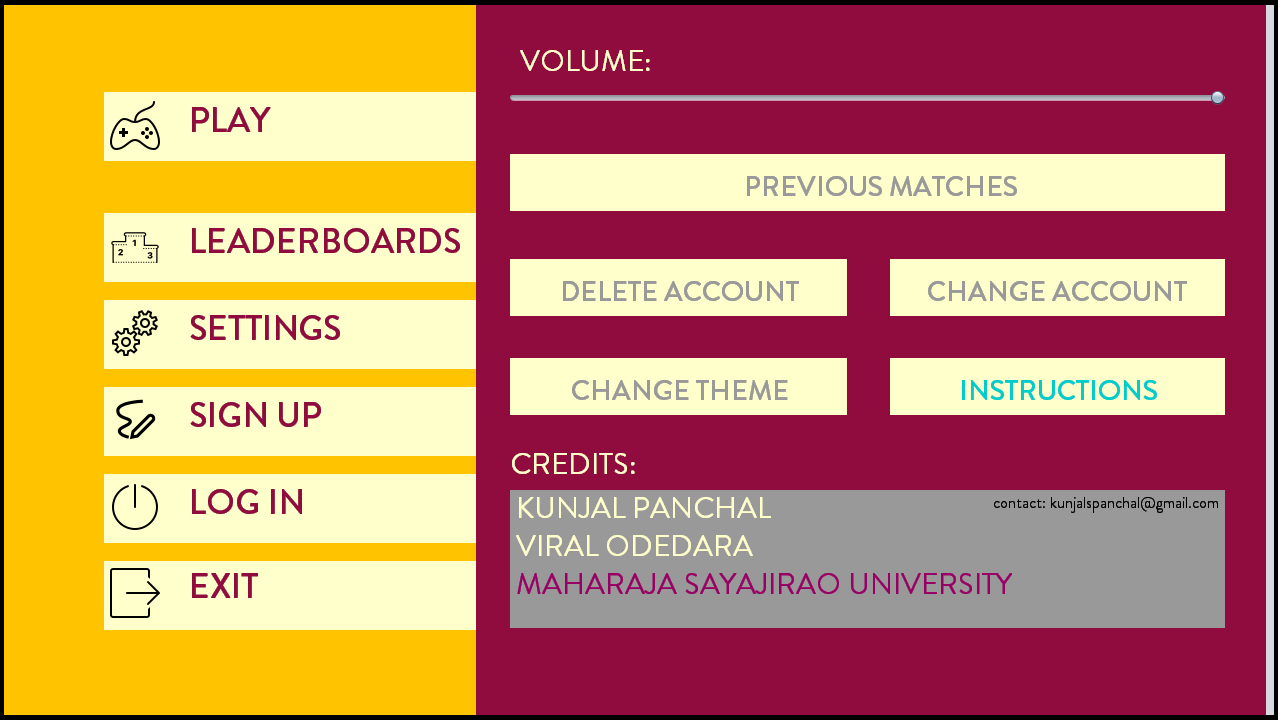


Fig: Main Menu Screen – Settings

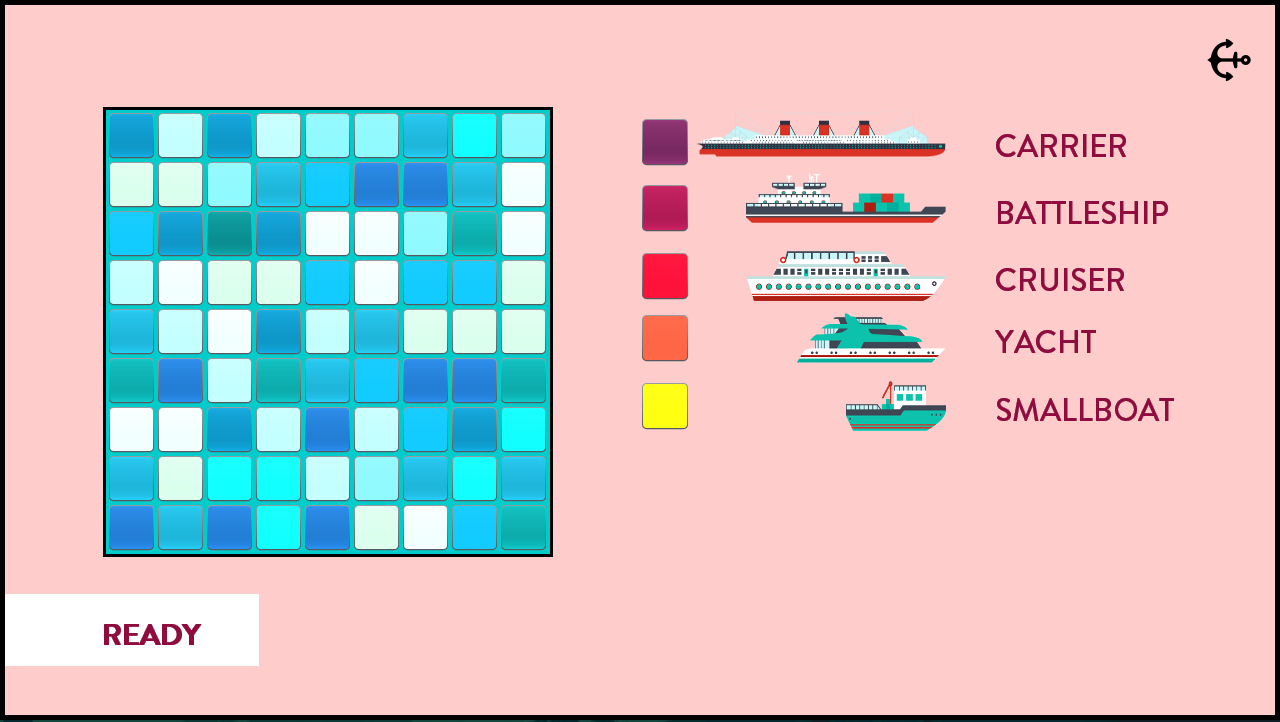


Fig: Creating the board

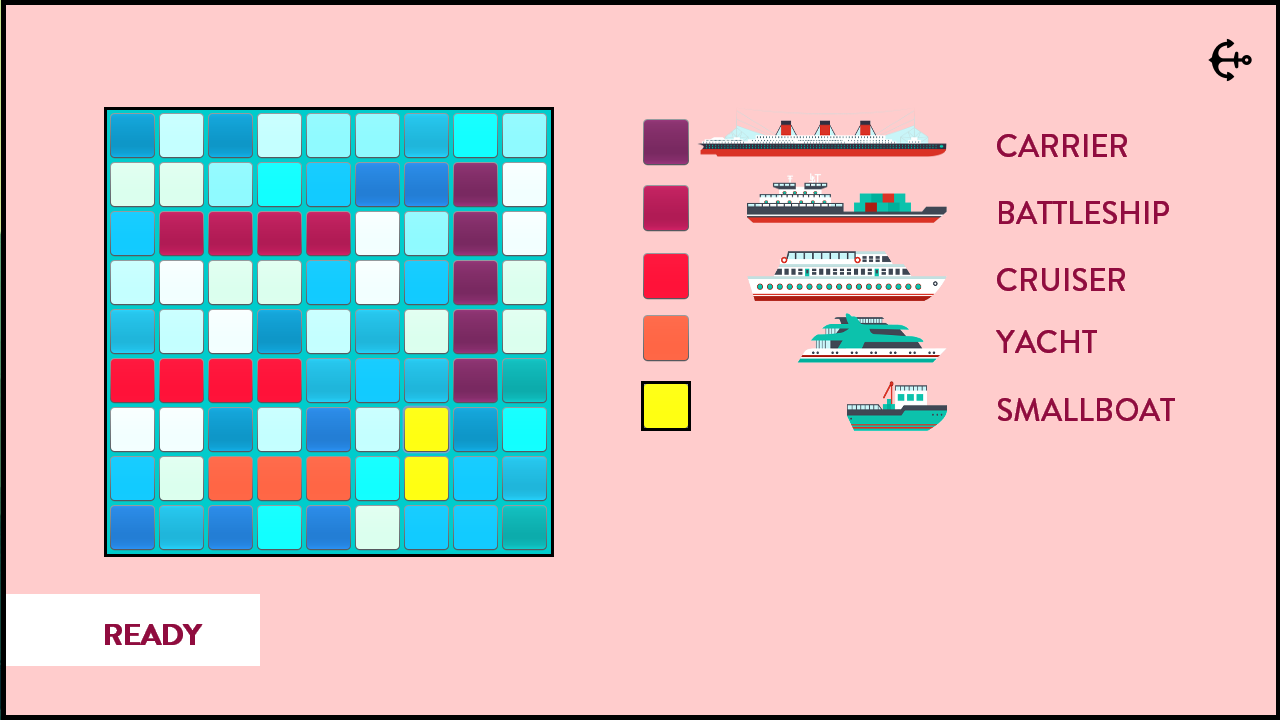


Fig: Putting the ships on board

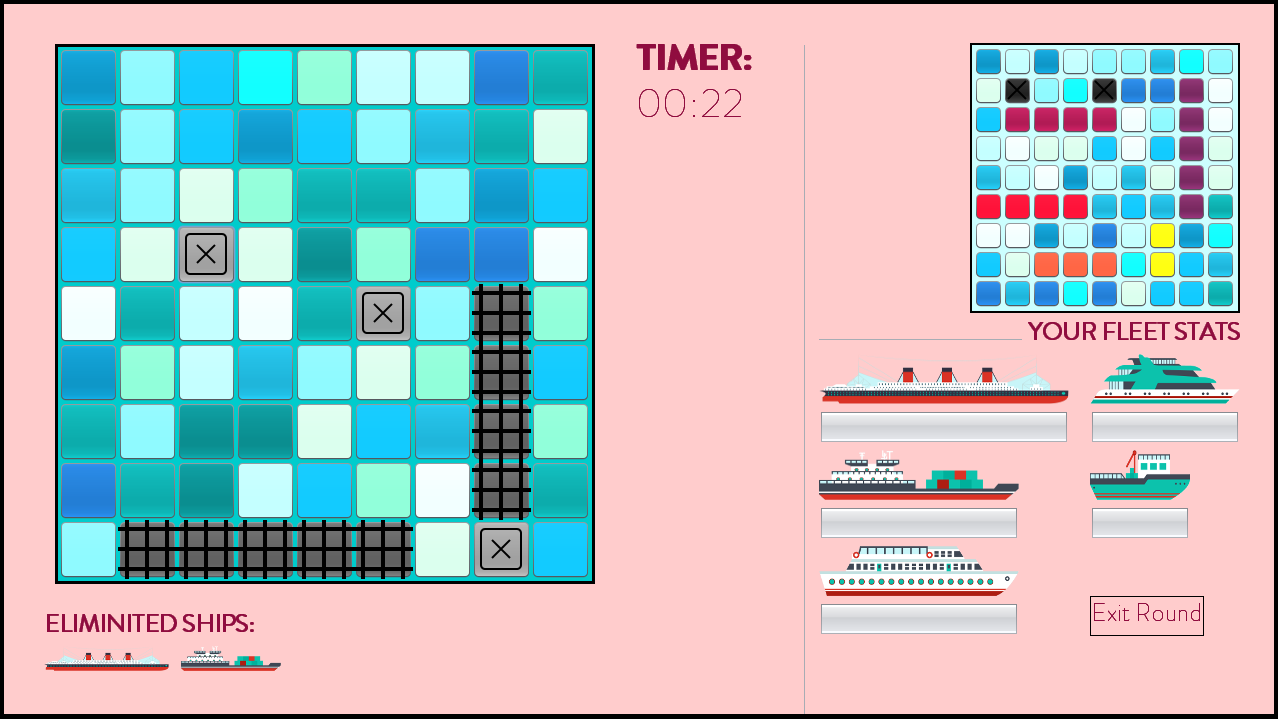


Fig: Round started

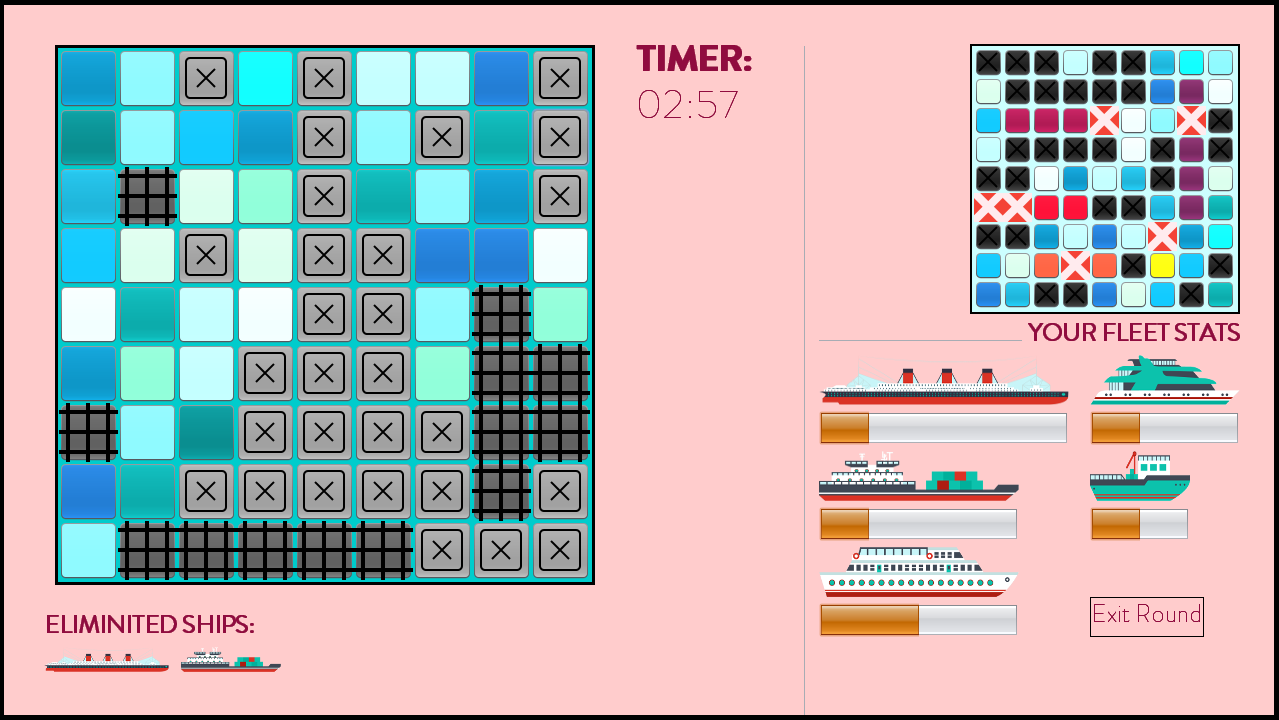


Fig: Destroying each other’s ship

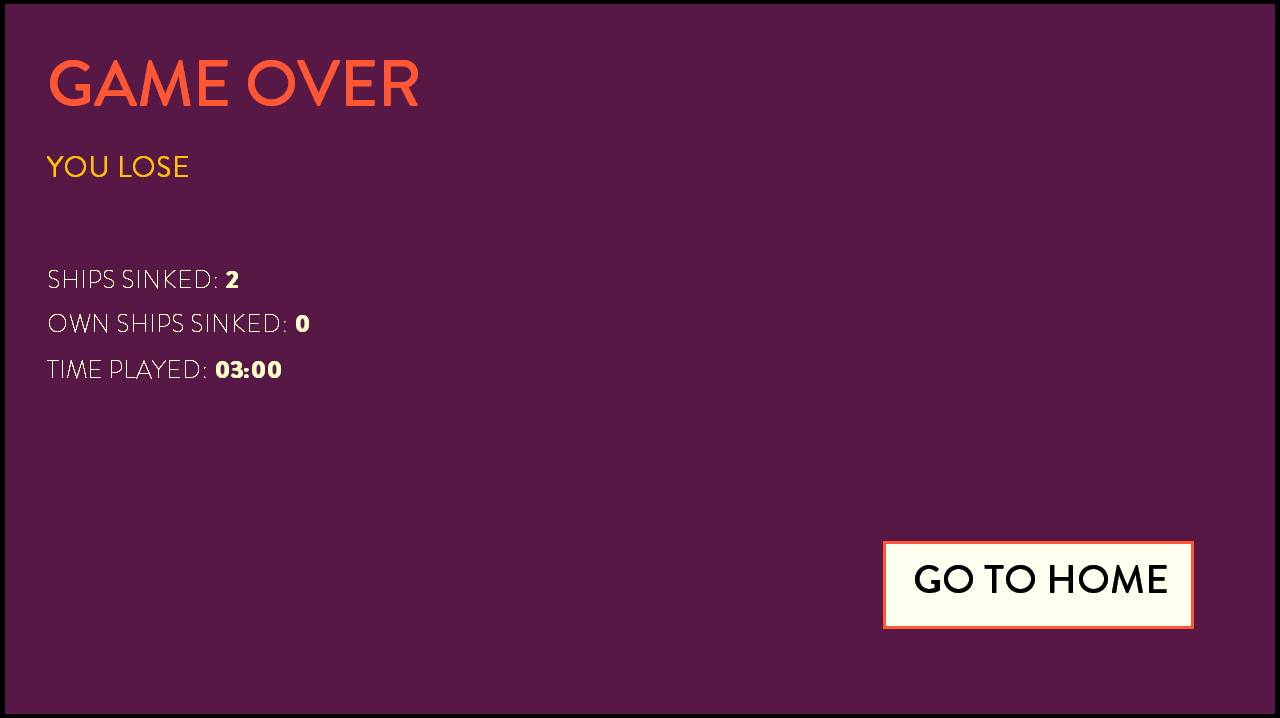


Fig: Game Over

1. **Source Code**

A snippet of game code from two classes which extends abstract Player:

1. P1Player

package battleship;

import java.awt.Color;

import javax.swing.JButton;

/\*

extends abstract Player class to define methods suitable to the real player and its input to the game

\*/

public class P1Player extends Player{

int[][] oppShipMap;

protected static boolean turn;

private int count = 0;

int shipHits[] = new int[5];

public P1Player() {

/\*oppShipMap = getBoard();

for(int row = 0; row < GRID\_SIZE; row++){

for(int col = 0; col < GRID\_SIZE; col++){

System.out.print(oppShipMap[row][col] + " ");

}

System.out.println("\n");

}\*/

turn = true;

}

@Override

protected void play(){

}

@Override

int[][] getBoard(){

return oppShipMap;

}

protected void setShipMap(){

oppShipMap = PlayManager.oppShipMap;

/\*for(int row = 0; row < GRID\_SIZE; row++){

for(int col = 0; col < GRID\_SIZE; col++){

System.out.print(oppShipMap[row][col] + " ");

}

System.out.println("\n");

}\*/

}

@Override

protected void onBombDrop(int row, int col, JButton b){

if(disabledMap[row][col]) return;

if(oppShipMap[row][col] == 5){

SoundManager.playBombSound("Bomb2.wav");

b.setIcon(new javax.swing.ImageIcon(getClass().getResource("/images/icons8-grid-100.png")));

b.setBackground(Color.DARK\_GRAY);

disabledMap[row][col] = true;

turn = true;

count++;

shipHits[0]++;

}

else if(oppShipMap[row][col] == 4){

SoundManager.playBombSound("Bomb2.wav");

b.setIcon(new javax.swing.ImageIcon(getClass().getResource("/images/icons8-grid-100.png")));

b.setBackground(Color.DARK\_GRAY);

disabledMap[row][col] = true;

turn = true;

count++;

if(shipHits[1] >=4) shipHits[2]++;

else shipHits[1]++;

}

else if(oppShipMap[row][col] == 3){

SoundManager.playBombSound("Bomb2.wav");

b.setIcon(new javax.swing.ImageIcon(getClass().getResource("/images/icons8-grid-100.png")));

b.setBackground(Color.DARK\_GRAY);

disabledMap[row][col] = true;

turn = true;

count++;

shipHits[3]++;

}

else if(oppShipMap[row][col] == 2){

SoundManager.playBombSound("Bomb2.wav");

b.setIcon(new javax.swing.ImageIcon(getClass().getResource("/images/icons8-grid-100.png")));

b.setBackground(Color.DARK\_GRAY);

disabledMap[row][col] = true;

turn = true;

count++;

shipHits[4]++;

}

else{

SoundManager.playBombSound("Bomb2.wav");

b.setIcon(new javax.swing.ImageIcon(getClass().getResource("/images/icons8-close-window-50.png")));

b.setBackground(Color.GRAY);

disabledMap[row][col] = true;

turn = false;

}

if(count == 18) {

PlayManager.gameOver = true;

PlayManager.winner = "YOU WIN!";

}

}

protected boolean getTurn(){

return turn;

}

int[] getHits(){

return shipHits;

}

}

1. P2Computer

package battleship;

import java.awt.Color;

import java.util.Arrays;

import javax.swing.JButton;

/\*

extends abstract Player class to define methods suitable to the computer and its algorithm for the game

\*/

public class P2Computer extends Player{

private int[][] shipCells = new int[GRID\_SIZE][GRID\_SIZE];

private int[] shipSize = {5, 4, 4, 3, 2};

protected static int[] shipDamaged = {0, 0, 0, 0, 0};

private P1Player p = new P1Player();

private boolean[][] disabledMap = new boolean[GRID\_SIZE][GRID\_SIZE];

private Color c;

public P2Computer() {

for(int row = 0; row < GRID\_SIZE; row++){

for(int col = 0; col < GRID\_SIZE; col++){

shipCells[row][col] = 0;

disabledMap[row][col] = false;

}

}

createBoard();

}

@Override

protected void play(){

int i = getRan();

int j = getRan();

if(disabledMap[i][j]) return;

//System.out.println("battleship.P2Computer.play()");

SoundManager.playBombSound("Bomb1.wav");

if(Arrays.asList(cyanS).contains(PlayManager.shipCells[i][j].getBackground())){

PlayManager.shipCells[i][j].setBackground(Color.BLACK);

PlayManager.shipCells[i][j].setIcon(new javax.swing.ImageIcon(getClass().getResource("/images/icons8-close-window-50.png")));

disabledMap[i][j] = true;

turn = false;

p.turn = true;

return;

}

else{

c = PlayManager.shipCells[i][j].getBackground();

if(c.getRed() == 88 && c.getGreen() == 24 && c.getBlue()== 69){

MainMenuScreen.jProgressBar2.setValue(MainMenuScreen.jProgressBar2.getValue() + 1);

if(MainMenuScreen.jProgressBar2.getValue() == 5){

Player.ownShipsDestroyed++;

}

}

else if(c.getRed() == 144 && c.getGreen() == 12 && c.getBlue()== 63){

MainMenuScreen.jProgressBar1.setValue(MainMenuScreen.jProgressBar1.getValue() + 1);

if(MainMenuScreen.jProgressBar1.getValue() == 4){

Player.ownShipsDestroyed++;

}

}

else if(c.getRed() == 255 && c.getGreen() == 0 && c.getBlue()== 42){

MainMenuScreen.jProgressBar4.setValue(MainMenuScreen.jProgressBar4.getValue() + 1);

if(MainMenuScreen.jProgressBar4.getValue() == 4){

Player.ownShipsDestroyed++;

}

}

else if(c.getRed() == 255 && c.getGreen() == 87 && c.getBlue()== 51){

MainMenuScreen.jProgressBar5.setValue(MainMenuScreen.jProgressBar5.getValue() + 1);

if(MainMenuScreen.jProgressBar5.getValue() == 3){

Player.ownShipsDestroyed++;

}

}

else if(c.getRed() == 255 && c.getGreen() == 255 && c.getBlue()== 0){

MainMenuScreen.jProgressBar3.setValue(MainMenuScreen.jProgressBar3.getValue() + 1);

if(MainMenuScreen.jProgressBar3.getValue() == 2){

Player.ownShipsDestroyed++;

}

}

PlayManager.shipCells[i][j].setBackground(Color.DARK\_GRAY);

PlayManager.shipCells[i][j].setIcon(new javax.swing.ImageIcon(getClass().getResource("/images/icons8-close-window-96.png")));

disabledMap[i][j] = true;

turn = true;

p.turn = false;

if(MainMenuScreen.jProgressBar2.getValue() == 5 &&

MainMenuScreen.jProgressBar1.getValue() == 4 &&

MainMenuScreen.jProgressBar4.getValue() == 4 &&

MainMenuScreen.jProgressBar5.getValue() == 3 &&

MainMenuScreen.jProgressBar3.getValue() == 2){

PlayManager.gameOver = true;

}

}

}

@Override

void createBoard(){

for(int i: shipSize){

putShip(i);

}

/\*for(int row = 0; row < GRID\_SIZE; row++){

for(int col = 0; col < GRID\_SIZE; col++){

System.out.print(shipCells[row][col] + " ");

}

System.out.println("\n");

}\*/

}

@Override

int[][] getBoard(){

return shipCells;

}

private int getRan(){

return (0 + (int)(Math.random() \* ((GRID\_SIZE - 1) + 1)));

}

void putShip(int size){

int k = getRan();

int l = getRan();

while(shipCells[k][l] != 0){

k = getRan();

l = getRan();

}

if(checkDown(k, l, size)){

for(int i = k; i < k + size; i++){

shipCells[i][l] = size;

}

}

else if(checkRight(k, l, size)){

for(int i = l; i < l + size; i++){

shipCells[k][i] = size;

}

}

else if(checkUp(k, l, size)){

for(int i = k; i > k - size; i--){

shipCells[i][l] = size;

}

}

else if(checkLeft(k, l, size)){

for(int i = l; i > l - size; i--){

shipCells[k][i] = size;

}

}

else{

putShip(size);

}

}

boolean checkDown(int k, int l, int size){

if(k + size >= GRID\_SIZE) return false;

for(int i = k; i < k + size; i++){

if(shipCells[i][l] != 0){

return false;

}

}

return true;

}

boolean checkUp(int k, int l, int size){

if(k - size < 0) return false;

for(int i = k; i > k - size; i--){

if(shipCells[i][l] != 0){

return false;

}

}

return true;

}

boolean checkRight(int k, int l, int size){

if(l + size >= GRID\_SIZE) return false;

for(int i = l; i < l + size; i++){

if(shipCells[k][i] != 0){

return false;

}

}

return true;

}

boolean checkLeft(int k, int l, int size){

if(l - size < 0) return false;

for(int i = l; i > l - size; i--){

if(shipCells[k][i] != 0){

return false;

}

}

return true;

}

void onBombDrop(int row, int col, JButton b){

}

protected boolean getTurn(){

return turn;

}

protected void checkIfDestroyed(int i, int j){

switch(shipCells[i][j]){

case 5:

shipDamaged[0]++;

if(shipDamaged[0] == 5) {

MainMenuScreen.jLabel51.setVisible(true);

Player.oppShipsDestroyed++;

}

break;

case 4:

if(shipDamaged[1] == 4){

shipDamaged[2]++;

if(shipDamaged[2] == 4) {

MainMenuScreen.jLabel49.setVisible(true);

Player.oppShipsDestroyed++;

}

}

else{

shipDamaged[1]++;

if(shipDamaged[1] == 4) {

MainMenuScreen.jLabel50.setVisible(true);

Player.oppShipsDestroyed++;

}

}

break;

case 3:

shipDamaged[3]++;

if(shipDamaged[3] == 3) {

MainMenuScreen.jLabel48.setVisible(true);

Player.oppShipsDestroyed++;

}

break;

case 2:

shipDamaged[4]++;

if(shipDamaged[4] == 2) {

MainMenuScreen.jLabel47.setVisible(true);

Player.oppShipsDestroyed++;

}

break;

}

}

}

1. **Conclusion**

The battleship game was a remarkable first step in learning how Java works and how we should implement the object-oriented principals in any software.

With the knowledge of Networking, we would like to enhance this project so that it can support online games against real players [pVp] and threading on networks to support both player operations in threaded form.