Advanced Application Development

Michael Little – 1103677

**King of Tokyo**

**Design Document**

Contents

1. Intro
   1. Purpose
   2. References
2. Application Description
3. Coding Standards
   1. Classes
   2. Methods
   3. Variables
   4. Braces
   5. Comments
4. High Level Design
   1. Application Flow Chart
5. User Interface Design
   1. Visual Design
   2. User Interface Elements
      1. Game Board
      2. Player Cards
      3. Dice
6. Feature Design and Implementation
   1. Application Framework
      1. Main
      2. Gameplay
      3. Player
      4. Dice
   2. Player Input
   3. Main
      1. Member variables
      2. Basic Methods
      3. UI Elements
   4. Gameplay Class
      1. Member Variables
      2. Basic Methods
      3. Method – ProcessResult()
      4. Method – Attack()
      5. Method – TakeTokyo()
   5. Player Class
      1. Member Variables
      2. Basic Methods
   6. Dice Class
      1. Member Variables
      2. Basic Methods
      3. Method – GetResult()
7. **Introduction**
   1. **Purpose**

The Purpose of this document is to provide design and implementation details of the game King of Tokyo (2015), an application based on the board game of the same name developed with Visual studios 2010 and SFML (Simple and Fast Media Library) for PC (Personal Computer). It will also show how the application will be structured and constructed in order to meet the requirements previously outlined in the King of Tokyo Specification document (Little, M. 2015).

The design will specify a structure a structure that compliments each aspect of the requirements in detail. This High level overview of the application allows the reader to visualise the application as a whole, and how the games mechanics will play out in a typical game.

* 1. **References**

King of Tokyo. 2010. [Board] Iello, Richard Garfield.

Iello 2010. King of Tokyo Instruction Leaflet

Little, M. 2015. King of Tokyo: software requirement specification. Dundee: Unpublished.

Figure 1: http://www.boardgamebeast.com/images/king-of-tokyo-board-game-21683303.jpg

1. **Application Description**

****

**Figure 1: Typical Set up for a 4 Player game**

“You are giant Monsters on a rampage. Your destructive feats earn you glory in the form of Victory points. The first to get to 20 wins the game. Or the last standing – if y’all want to take that route”(Iello 2010).

King of Tokyo is an application designed for PC, based on Richard Garfield’s King of Tokyo Board game. Players will be able to interact with the application using the mouse and take turns using a single device. The application will be completely stand alone with no external dependencies.

1. **Coding Standards**
   1. **Classes**

All classes will be written using the Google C++ naming convention, and all will have simple easy to understand names that relate to their function in the program.

Example: Player.

* 1. **Methods**

All methods written using the Google C++ naming convention, and all will be named after their function and what class(es)/variables they interact with.

Example: GetResults();

* 1. **Variables**

Local variables will be written in lowercase and follow the Google C++ naming convention. The name of each variable will easily describe their purpose to help anyone reading the code understand what’s happening.

Example: int health;

* 1. **Braces**

Braces will be indented to line up with their statement and will be given their own lines to help each Method/statement stand out and be easier to read.

Example:

Player::GetHealth()

{

return health;

}

* 1. **Comments**

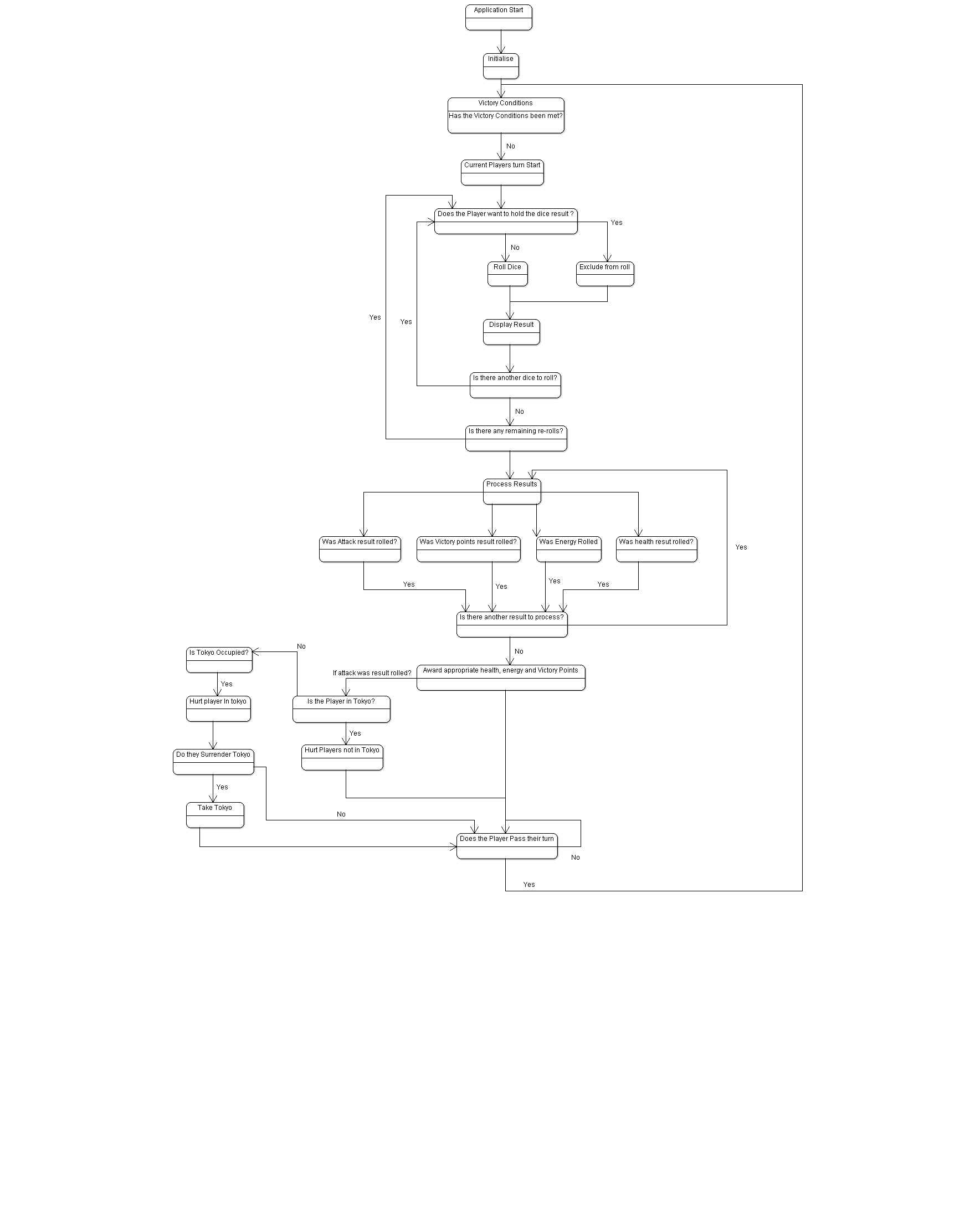
Comments will be used to help add descriptions to each class to help explain Methods and Variables and the inner processes of the project. Each class will also contain comments stating the author of the class, the date they were created and comment lines to help separate the different sections of each class.

Example:

health = 10; //Sets Player health

1. **High Level Design**
   1. **Application Flow Chart**

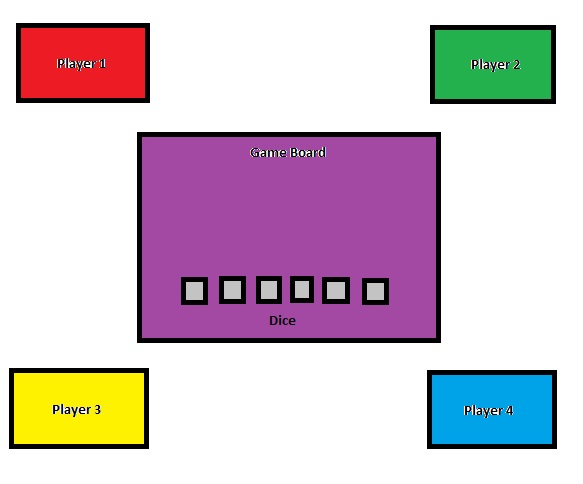
The following flow chart represents the processes and typical route the application will take during a typical turn of play.

**Figure 2: Application flow chart**

**5.0 User Interface Design**

This section is to explain the overall design of the user interface, this will cover both visual representations of the layout and the purpose of each UI element.

**5.1 Visual Design**

****

**5.2 User Interface elements**

**5.2.1 Game Board**

The game board will be used to display which players turn it is and which player (monster) is “In Tokyo” as well as the dice, the buttons roll the dice and any notifications. The board placed in the centre of the screen allows for all players to easily view and interact with it, as well as view their own stats in their own individual player screen.

**5.2.2 Player Cards**

The player cards are used to display each player’s health, victory points, energy and number. Their placed to each side of the game board so the player can easily keep track of their stats without having to look away from the game board.

**5.2.3 Dice**

The dice section is located at the bottom of the game board, this contains 6 buttons for the individual dice and a roll button. The dice buttons are used to hold the dice so that they aren’t affected by the dice roll, and clicking the dice roll button rolls the dice.

**6.0 Feature Design and Implementation**

**6.1 Application Framework**

The Application framework consists of the classes Main, Gameplay, Players and Dice. The purpose of these classes is to replicate the various features of the Board game in a way that that spaces out the various tasks and responsibilities without having one class feel too overloaded. The framework also makes use of SFML (Simple and Fast Media Library) 2.2 and its various features.

**6.1.1 Main**

The Main class is the base for which the application is created. This class is responsible for Initialising, scaling, positioning and drawing the windows which displays the various elements of game. It also handles the creation of various UI elements including Text and Buttons.

**6.1.2 Gameplay**

The Gameplay class is the mediator between the Main class and the Player and Dice classes. This class is responsible for the logic of the game, it handles the logic for the Buttons and calls the various functions in the Player and Dice classes. It also contains the ProcessResult() function which takes in the result of the dice roll and then takes the appropriate response.

**6.1.3 Player**

The Player class is responsible for holding the various information for each player, it also contains several functions that the Gameplay class uses to get, set and change the information.

**6.1.4 Dice**

The Dice class is responsible for holding the information of the dice themselves and the different functions that are called in the gameplay class. One of the main methods of this class is the RollDice() function, with generates a random number between 1 and 6.

**6.2 Player Input**

All player input is done through the mouse, thus the Interface features several buttons for the player to click on to interact with the application. These elements are created using SFML shapes and then functions in the main class detects the mouse position and whether the player clicks the mouse button on top of the buttons.

**6.3 Main**

The Main Class is the starting point for the application, it inherits from the Gameplay class and takes the information from this class and displays it in windows for the user to interact with. The Windows are created using SFML’s RenderWindow Function, which allows for windows to be easily created, scaled and drawn.

The Member Variables and functions used in the Main class are as follows:

**6.3.1 Member variables**

**Boolean**

* mouseClicked:

Used to detect when the left mouse button has been pressed, so buttons can’t be spammed.

**6.3.2 Basic Methods**

**Main loop:**

* int main(int argc, char\* argv[])

Starting point for the Application.

**Update:**

* void Update()

Used to update the variables stored in the different windows.

**6.3.3 UI Elements**

**Sf::RenderWindow**

* Board:

Window where dice and game board are viewed

* Player 1-4:

Player screens where players statistics are viewed

**Sf::RectangleShape –** Draws rectangles, used for buttons and status windows.

* DiceBtn 1-6

Used to hold the results of the 6 different dice, they also set the dice to be held if their clicked.

* RollBtn

Used the call the RollDice() function when clicked.

* EndBtn

Used to pass the turn to the next player.

* StatusBox

Game information is displayed here. (e.g. current player)

* MsgBox

Pops up when the player is asked if they want to surrender Tokyo

* YesBtn

Used to surrender Tokyo when enquired

* NoBtn

Used to refuse to surrender Tokyo when asked.

* VicScr

Pops up when a player has won the game.

**Std::ostringstream –** Strings used to draw text and display variables

* **d1 - d6**

Holds the results for each dice, drawn on top of the corresponding DiceBtn.

* **rb**

When drawn displays the remaining rolls left for the player, drawn on top of RollBtn. If no rolls remain it tells the player to click roll button again to process the results.

* **sb**

When drawn displays who’s turn it is, drawn on top of the StatusBox.

* **oc**

Drawn when a player takes Tokyo, drawn on top of StatusBox.

* **sr**

Drawn when the player is asked if they want to surrender Tokyo, Drawn on top of MsgBox.

* **yb**

Draws “Yes” on top of YesBtn.

* **nb**

Draws “No” on top of NoBtn.

* **eb**

Drawn when the player is asked if they want to pass their turn, drawn on top of EndBtn.

* **vs**

Drawn when a player Wins a game, Drawn on VicScr.

* **p1 – p6**

Displays the player’s stats when drawn, drawn in the corresponding player screens.

* **dp**

Draws “Dead” on the corresponding players screen when they die.

* **dk**

Draws the Dice key on the game board so the player can tell which each dice roll means.

**6.4 GamePlay**

The Gameplay is the Logic class of the Application, It gathers the information from the dice and player classes and sends them to be displayed in the main class. The Gameplay class is also home to the functions that allows the player to interact with application as well as the functions that lets the game be played according to the rules of the original board game.

The Member Variables and functions of the Gameplay class are as follows:

**6.4.1 Member variables**

**Integers**

* int tempAttack;

Used to temporarily store the amount of times the current player rolls the Attack result.

* int tempEnergy;

Used to temporarily store the amount of times the current player rolls the Energy result.

* int tempHealth;

Used to temporarily store the amount of times the current player rolls the Health result.

* int temp1VP;

Used to temporarily store the amount of times the current player rolls the 1 Victory Point result. Must have at least 3 results, to score the point.

* int temp2VP;

Used to temporarily store the amount of times the current player rolls the 2 Victory Point result. Must have at least 3 results, to score the points.

* int temp3VP;

Used to temporarily store the amount of times the current player rolls the 1 Victory Point result. Must have at least 3 results, to score the points.

* int currentPlayer;

Used to store the number of the current player.

* int inTokyo;

Used to store the player that is currently in Tokyo.

* int deadCount;

Used to count the number of dead players, if it reaches 3 then there is only one player remaining.

**Boolean:**

* bool endTurn;

Used to state that the player has used all of their rolls and to spawn the pass turn button (EndBtn).

* bool surrender;

Used as a trigger to display the option to surrender Tokyo.

* bool occupied;

Used to state whether or not Tokyo is occupied.

* bool processed;

Use to state whether or not the dice results have been processed.

**6.4.2 Basic Methods**

**Update:**

* void Update(sf::RenderWindow &Window);

Contains scode for detecting if a player has won or died.

**Setters:**

* void SetEndTurn(bool i);

Used to set the boolean endTurn to true or false.

* void SetCurrentPlayer(int i);

Used to set the int currentPlayer.

* void SetSurrender(bool i);

Used to set the bool surrender to true or false.

* void SetProcessed(bool i);

Used to set the Boolean processed to true or false.

**Getters:**

* int GetCurrentPlayer();

Returns the current player.

* int GetInTokyo();

Returns the number of the player in Tokyo.

* bool GetEndTurn();

Returns the value of the Boolean endTurn.

* bool GetOccupied();

Returns the value of the Boolean occupied.

* bool GetSurrender()

Returns the value of the Boolean surrender.

* bool GetProcessed();

Returns the value of the Boolean processed

**Containers:**

* std :: vector <Player> playerContainer;

A vector array for the different players.

* std :: vector <Dice> diceContainer;

A vector array for the different dice.

**6.4.3 Method – ProcessResults()**

Process Result is called after the player has used all of their re-rolls. It takes in all 6 of the results as a vector then one by one reads the result and adds one to an appropriate temporary variable. After all the results have been processed the function then checks the temporary variables and if they meet the right criteria they then call the appropriate functions.

**Code:**

void GamePlay::ProcessResults(std :: vector <Dice> diceContainer) // processes the dice results and takes the appropriate actions

{

//resets variables to 0 each time called.

tempAttack = 0;

tempEnergy = 0;

tempHealth = 0;

temp1VP = 0;

temp2VP = 0;

temp3VP = 0;

for (int i = 0; i != diceContainer.size(); i++)

{

if(diceContainer[i].GetResult() == 1)

{

//1VP

temp1VP += 1;

}

else if (diceContainer[i].GetResult() == 2)

{

//2VP

temp2VP += 1;

}

else if (diceContainer[i].GetResult() == 3)

{

//3VP

temp3VP += 1;

}

else if (diceContainer[i].GetResult() == 4)

{

//Attack

tempAttack += 1;

}

else if (diceContainer[i].GetResult() == 5)

{

//Health

tempHealth += 1;

}

else if (diceContainer[i].GetResult() == 6)

{

//Energy

tempEnergy += 1;

}

}

if(temp1VP >= 3)

{

//grant current player 1VP

playerContainer[currentPlayer].ChangeVP(1);

}

if(temp2VP >= 3)

{

//grant current player 2VP

playerContainer[currentPlayer].ChangeVP(2);

}

if(temp3VP >= 3)

{

//grant current player 3VP

playerContainer[currentPlayer].ChangeVP(3);

}

if(tempAttack > 0)

{

for(int i = 0; i < tempAttack; i++)//attack function

{

Attack();

}

}

if(tempHealth > 0)

{

if(playerContainer[currentPlayer].GetHealth() < 10)//grant player 1 health

{

for(int i = 0; i < tempHealth; i++)

{

playerContainer[currentPlayer].ChangeHealth(1);

}

}

}

if(tempEnergy > 0)

{

//grant player 1 energy

playerContainer[currentPlayer].ChangeEnergy(tempEnergy);

}

endTurn = true;

}

**6.4.4 Method – Attack()**

The Attack function is used to hurt the other players. First it detects if the player is in Tokyo, if not it then attacks the players in Tokyo, otherwise it attacks the players outside Tokyo.

**Code:**

void GamePlay::Attack()

{

if(playerContainer[currentPlayer].GetLocation() == true) //if player is in tokyo

{

for(int i = 0; i < playerContainer.size(); i++)

{

if(playerContainer[i].GetLocation() == false)

{

playerContainer[i].ChangeHealth(-1);

}

}

}

if(playerContainer[currentPlayer].GetLocation() == false) // if player isnt in tokyo

{

for(int i = 0; i < playerContainer.size(); i++)

{

if(playerContainer[i].GetLocation() == true)

{

playerContainer[i].ChangeHealth(-1);

}

}

}

}

**6.4.5. Method – TakeTokyo()**

The TakeTokyo function is used to allow players to take control of Tokyo. First it removes the current occupant from Tokyo, then it sets it so the current player is the new occupant of Tokyo.

**Code:**

void GamePlay::TakeTokyo()

{

playerContainer[inTokyo].SetLocation(false);

occupied = false;

playerContainer[GetCurrentPlayer()].SetLocation(true);

inTokyo = GetCurrentPlayer();

playerContainer[GetCurrentPlayer()].ChangeVP(1);

surrender = false;

occupied = true;

}

**6.5 Player**

The Player class is used to hold all the various information for each player, including their health, victory points and energy. The Functions of this class are all about the retrieval and manipulation of this information.

The Member variables and functions of the Player class are as follows:

**6.5.1 Member variables**

**Integers:**

* int health;

Stores the Health for the individual players.

* int vp;

Stores the Victory points for the individual players.

* int energy;

Stores the Health for the individual players.

* int playerCount;

Stores the Number of players;

* int rolls;

Stores the number of rolls the player has.

**Boolean:**

* bool intokyo;

States whether or not the player is in Tokyo.

* bool dead;

States whether or not the player is dead.

* bool won;

States whether or not the player has won.

**6.5.2 Basic Methods**

**Constructor:**

* Player();

Initialises the variables to their default values.

**Getters:**

* int GetHealth();

Returns the players health when called.

* int GetVP();

Returns the Players Victory points when called.

* int GetEnergy();

Returns the Players Energy when called.

* int GetPlayerCount();

Returns the Player Count when called.

* int GetRolls();

Returns the players remaining rolls when called.

* bool GetLocation();

Returns whether or not the player is in Tokyo.

* bool GetDead();

Returns whether or not the player is dead.

* bool GetWin();

Returns whether or not the player has won.

**Setters:**

* void SetHealth(int i);

Sets the players health to the specified value when called.

* void SetVP(int i);

Sets the players victory points to the specified value when called.

* void SetEnergy(int i);

Sets the players energy to the specified value when called.

* void SetPlayerCount(int i);

Sets the player count to the specified value when called.

* void SetRolls(int i);

Sets the amount of rolls each player has when called.

* void SetLocation(bool i);

Sets the players location to be true or false when called.

* void SetDead(bool i);

Sets the player being to dead to either true or false.

* void SetWin(bool i);

Sets if the player has won to true or flase.

**Changers:**

* void ChangeHealth(int i);

Adds the Specified value to the player’s health total.

* void ChangeVP(int i);

Adds the Specified value to the Player’s Victory points total.

* void ChangeEnergy(int i);

Adds the Specified value to the player’s energy total.

* void ChangeRolls(int i);

Adds the Specified value to the player’s roll total.

**6.6 Dice**

The Dice class is used to hold the information for the dice, as well as functions for rolling the dice, and retrieving the information.

The Member variables and functions of the Dice Class are as follows:

**6.6.1 Member variables**

**Integers:**

* int result;

Used to hold the results of the dice roll.

* int diceNum;

Used to hold the number of Dice to be rolled.

**Boolean:**

* bool hold;

Used to state whether or not the Dice is to be held when rolling.

**6.6.2 Basic Methods**

**Constructor:**

* Dice();

Initialises the variables to their default values.

**Getters:**

* int GetDiceNum();

Returns the Number of dice when called.

* int GetResult();

Returns the result of the dice roll when called.

* bool GetHold();

Returns whether or not the dice is to be held from the roll.

**Setters:**

* void SetHold(bool i);

Sets whether or not the dice is to be held to true or false when called.

* void SetResults(int i);

Sets the result of the dice to the specified value.

**6.6.3 Method – RollDice()**

Used to generate a random number between 1 and 6.

void Dice::DiceRoll()

{

result = 0;

result = (rand() % 6 ) + 1;

}