Ajeet Kotturu, Matthew Wang

Project Specifications

1. Structural Design

Our structural design decisions are shown in Table 1.

|  |  |
| --- | --- |
| **Data** | ***interface* => class** |
| Game Board Piece : Knight | Piece => GamePiece |
| Game Board Piece : CrossBows | Piece => GamePiece |
| Game Board Piece : LongBows | Piece => GamePiece |
| GameBoard : Piece Locations | Object => GamePiece[][] |
| GameBoard: Locations | List => ArrayList<Location> |
| PieceOne : Knight Moveset | List => ArrayList<Location> |
| PieceTwo: CrossBow Moveset | List => ArrayList<Location> |
| PieceThree : LongBow Moveset | List => ArrayList<Location> |
| PieceOne : Knight Possible Attacks | List => ArrayList<Location> |
| PieceTwo : CrossBow Possible Attacks | List => ArrayList<Location> |
| PieceThree : LongBow Possible Attacks | List => ArrayList<Location> |
| GameBoard : PlayerOne List of Pieces | List => ArrayList<GamePiece> |
| GameBoard : PlayerTwo List of Pieces | List => ArrayList<GamePiece> |
| GameGUI : Squares List | List => ArrayList<Square> |

**Table: Structural design decisions for *BattleZone***

Our project is called BattleZone, and it is an turn based medieval themed board game. It is played on a 8 squares by 8 squares board, each player having 12 pieces initially. The game ends when one player’s pieces are completely eliminated. The game features many mechanics similar to chess, but instead of having many different pieces, BattleZone features three exclusive pieces. Moreover, there are a total of four turns in one round, in which each player attacks and moves. No two pieces can occupy the same place; therefore, each piece has a determined attack range. BattleZone is a two player versus game, where both players play on one device.

The three pieces are called the following: the Knight, the CrossBow, and the LongBow. Each piece has a certain integral amount of health, and each piece has an attack damage.. Each piece also has a range in which it can execute its attack. Health is how much attack damage a piece can take before being removed. Thus, when health becomes zero or less, the piece is removed from the game. Remaining health is calculated by the following equation:

Health - Attack = Updated Health

An ArrayList is used to keep track of all the possible moves and all the possible attacks a piece can execute, one of each for every piece. The Move(Location) and Attack(GamePiece) in the GamePiece class make sure the move or attack is a valid move. Attack checks if the GamePiece parameter’s Location is located in the attacking piece ArrayList of all possible attacks. Move checks if the Location parameter is within the ArrayList of all possible moves and the Location does not already contain a piece.

The Knight is the sturdiest of all the pieces, and its role is to soak as much damage as possible. The Knight can move one space in any direction. It has a health of eight, and an attack of two. The Knight’s biggest weakest is its range, a meager range of one. The Knight can be easily taken out from a distance. If a player lets the opponent’s Knights reach his or her LongBows, the game will sway in the opponent’s favor.

The CrossBow is what is commonly referred to as the “glass cannon”. The CrossBow can move one space diagonally. It has a health of four, and an attack of three. The CrossBow has a decent range of two, but mispositioning could leave the CrossBow an easy target. CrossBows also serve as the main source of damage, so eliminating a player’s CrossBow greatly reduces his or her ability to do damage.

The LongBow deals consistent damage throughout the game. The LongBow can move one space horizontally or vertically. It has a minute health of two, and has a meager attack of one. However, the LongBow has an absurd range of five. Thus, as long as player is able to keep his or her LongBows behind the front lines, the LongBow can pack consistent damage throughout the game. LongBows will die quickly upon any attack.

The GameBoard class contains the main data structure of the game. It contains a two dimensional array that holds the object type GamePiece. The GameBoard uses the location class to help store the location of each piece. The x-value in Location is the row value in the two dimensional array, and he y-value in Location is the row value in the two dimensional array. Throughout the game, the pieces are commonly switched or eliminated from the two dimensional array. This is by far the best data structure for the game board because the size of the game board never changes, and quick access to the location of pieces is an absolute must.

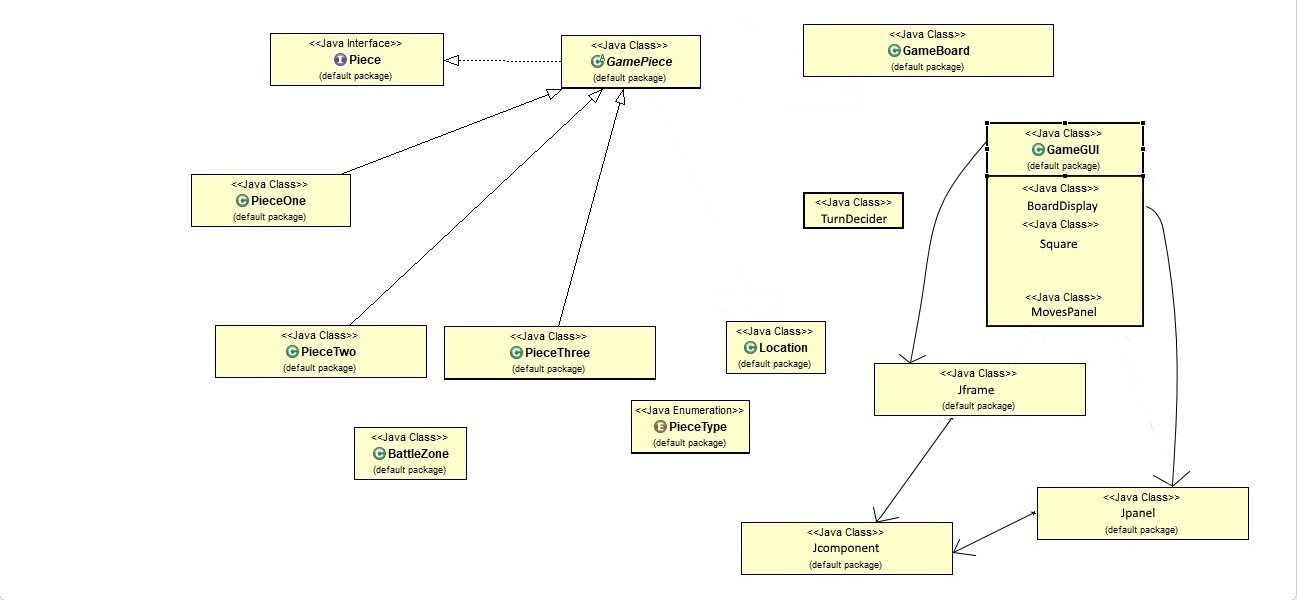
The GameBoard class also contains an ArrayList that stores all the possible Locations on the GameBoard. This is initialized in the constructor, in which every possible row and column value is added into this ArrayList in ta newly constructed Location. This Location ArrayList is used mostly for error checking and making sure the execution of a move or attack occurs within one of these Locations.

The Move(Location) and Attack(GamePiece) methods in GamePiece are the methods used for the execution of player moves. Move utilizes setGameBoard(GamePiece, Location) to make the changes in the private field GamePiece[][]. Since in Attack(GamePiece) the pieces do not actually move, setGameBoard(gamePiece, Location) is unnecessary for the this method. However, the mouseListener in the GUI has to check to see if the Location that the player is attacking contains a GamePiece. These methods combined do the error checking described above.

To differentiate which piece belongs to which player, we decided to use the two ArrayLists PlayerOne and PlayerTwo to store this data. Each piece is put into a certain player arrayList by their string play parameter. By using this data structure we can easily remove pieces that are defeated in the game. Eventually one of the players will lose all the pieces they have. With the ArrayList Data Structure, we can easily determine when the game ends by seeing if one of the ArrayLists is empty.

GameGUI contains an ArrayList that stores Square, an object that will be discussed in further detail later. Square essentially refers to one Location within the two dimensional array in GameBoard on the GUI. This ArrayList is used by BoardDisplay, the class that contains the grid layout, each grid containing a Square. This allows BoardDisplay to easily search and make changes to all Square objects on its grid layout.

1. Object-Oriented Design



The data hierarchy of BattleZone consists of one data hierarchy tree for the game pieces and another for the GUI. The BattleZone class containing the Main, the Enum PiecType, the Location class, and the GameBoard class do not have any hierarchical connections with these two trees, but they are still essential to running this BattleZone.

The following describes the structure of the hierarchy tree containing the game pieces. At the very top is the interface Piece, which contains all the methods that the pieces must have, including assessor methods like getAttack() and getHealth() as well as manipulating methods like setLocation(). Abstract class GamePiece implements as many of these methods as it can, excluding the methods that are exclusive to each piece. PieceOne, PieceTwo, and PieceThree inherit the Game Piece class, adding the piece specific methods such as getPossibleMoves() and including constructors for their own pieces. GamePiece contains two very crucial methods, Attack(GamePiece) and Move(Location) that manipulate the pieces on the board and drive the pace of the game and will be discussed in more detail below.

The Enum PieceType consists of three different values. These three values are used to identify the type of piece in each piece class. If a piece is asked to return its type, it returns ONE< TWO, or THREE depending on the piece. This enum is used in the getPieceType method in each piece class.

GameBoard is separated from this main hierarchy tree, but it contains the main data structure of the game, the two dimensional array storing GamePieces. GameBoard consists of methods that manipulate this two dimensional array, such as turnGameBoard(), which flips the gameBoard so that the top becomes the bottom and vice versa. It also includes a setGameBoard(GamePiece, Location) which allows for changes to be made directly on the two dimensional array.

The Location class takes two integers and stores the first one as the x-coordinate and the second one as the y-coordinate. It can also return the x-coordinate and the y-coordinate through getter methods. Location is used to organize the coordinates of each piece.

The Attack(GamePiece) method returns a boolean. It returns true if the attack is executed, and false if it is not. The attack method checks to see if the piece the player wants to attack is within range of the attacking piece. If the piece the player chooses meets this condition, attack will get the piece that is attacked and call the reduceHealth(int) method from its respective piece class, subtracting the attacking piece’s attack from the being attacked piece’s health.

The Move(Location) method also returns a boolean. Just like the Attack class, it returns true if the player moves, or false if it is unable to. The move method checks to see if the space that the player wants to move to is in the possible moves ArrayList. Move(Location) calls setGameBoard(GamePiece, Location) to make the move on the two dimensional array. This method in GameBoard further checks if the location the piece wants to move to is occupied. If it is, it will return false and also cause Move(Location) to return false. If not, it will execute the move on the array, set the new Location to the piece’s Location variable, and return true.

As stated earlier, GameBoard contains a two-dimensional array which hold the pieces in the game. It also contains two arrayLists each one holding pieces the player uses. To get these arrayLists, other classes can access them with the getter methods that return each array. When the game first starts the gameGUI constructors calls fillGameBoard(player, player) to fill the board with the pieces. If a piece loses all its health, the piece is removed from the associated player arrayList and the two dimensional array..

BattleZone is the main class that runs the game. BattleZone contains some test methods that print integers to represent how the two dimensional array in GameBoard looks like visually. It also contains the main method that starts the program by constructing a new GameGUI object and opening it.

The second data tree contains the GUI elements. The main GUI class, GameGUI, extends JFrame. It combines all the components of the GUI and builds an infrastructure for them. The components of the GUI, BoardDisplay, Square, and MovesPanel all describe a construct a different part of the GUI. Square expands upon BoardDisplay, initializing and providing methods to the individual blocks in BoardDisplay. BoardDisplay contains a 8 X 8 grid consisting of Square objects. MovesPanel is the side panel on the East that displays the current turn and statistics of the piece that the player hovers over.



Here is an overview of the GUI. GameGUI creates the entire GUI by putting the pieces together. Utilizing a BorderLayout() from JFrame, GameGUI is able to combine the movesPanel, which is located on the right hand side (both the blue and the green panels) and the BoardDisplay containing Square objects. GameGUI also has a Menu Bar at the top. Here are the components of the Menu Bar:

* Game
  + Reset Game (calls ResetGameBoard() in GameBoard, which creates a new two dimensional array and fills the new array with fillGameBoard())
  + Exit Game (removes the BorderDisplay and tells the system to exit)
* Skip
  + Skip Turn (tells TurnDecider(more detail later) to advance a turn)
* Tools
  + Toggle Displaying Range on Hover (turns on/off when the user hovers over a piece, it displays the attack range of the piece - on when boolean ghost is true, off if false)

It contains two private GamePiece fields, selectedPiece and movetoPiece that will be used later. It has another private GamePiece field called ghostPiece that stores the piece the user is hovering his or her mouse over. GameGUI calls fillGameBoard(“PlayerOne”, “PlayerTwo”) to set the default pieces in default locations. Thus, “PlayerOne” is also “Blue” and “PlayerTwo” is also “Red”. GameGUI also initializes a turnDecider, which is a private class in GameGUI.

The turnDecider class creates a string object that keeps track of the turns. It contains an advanceTurn() method that finds the current turn and changes the string to the next turn. TurnDecider also has a contains(string) method that allows the other classes to check which player is making the play and whether it is an attack or move turn.

Here is the turn order:

* PlayerOne “Blue” Moves
* PlayerTwo “Red” Moves
* PlayerTwo “Red” Attacks
* PlayerOne “Blue” Attacks

For balance purposes, PlayerOne moves first but PlayerTwo attacks first for compensation.

The BoardDisplay creates a GridLayout(8, 8) from JPanel. This allows the GUI to mirror the two dimensional array, both with the same dimensions. Each grid in BoardDisplay contains a Square, a total of 64. BoardDisplay’s Constructor initializes each Square and its corresponding Location on the two dimensional array in GameBoard. This allows changes in the main data structure to be reflected upon in the GUI. BoardDisplay has a drawBoard(GameBoard) method, which is used to update the BoardDisplay on the GUI. It goes through the ArrayList of Square objects and tells each one to redraw itself. This method also calls checkWinCondition(), which will display a pop up with which player wins when one of the ArrayLists containing the player’s pieces has a size of zero.

The Square method is the primary object that makes the BoardDisplay interactive. It has an mouseListener with the following responses:

Right click - If a player has selected a piece to move/attack, right click will set selectedPiece to null and cancel the play.

Left click - If a player has not selected a piece, the selectedSquare will become selectedPiece if the Square has a piece in the corresponding Game Board. Checks if selectedPiece is one of the pieces of the player making the move. If not, does nothing. If a player has already selected a piece, depending on whether TurnDecider contains move or attack, the gameboard will highlight possible moves/attack range. If the move/attack is successful and returns true, advances the turn and makes the play. If not, the play is canceled and the turn stays the same.

Mouse Enters (Hover) - If the Square has a piece and the boolean ghost is true, assigns ghostPiece to the piece if selectedPiece is still null. Updates movePanel with statistics of the piece.

Mouse Exits (Stops Hover) - Undoes everything from when the mouse is hovered over the Square.

The Square also has a drawSquare(GameBoard board) that redraws itself. It assigns its proper color and updates the image of the piece (if there is a piece). BoardDisplay calls this method for each Square in its arrayList when repainting the Board. The default color method assigns the Square a color of brown or white, in which no adjacent Square objects have the same color. There are also three highlight methods that override the the color assigned in this method.

HighlightMoves(GameBoard) gets the valid moves and highlights the Square objects with those Locations. It only occurs on a move turn and looks like this (green):



HighlightAttacks(GameBoard) gets the attack range and highlights the Square objects with a Location in this range. It only occurs on an attack turn and looks like this (orange):



GhostAttacks(Game Board) displays the attack range of ghostPiece and uses almost identical code to the previous method. Since ghostPiece is assigned by hovering your mouse, the range is displayed when you hover your mouse (grey):



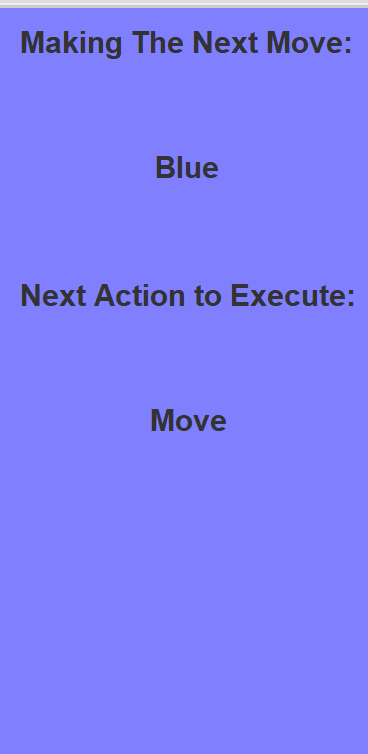
Images are assigned in the setBoard(GameBoard) method and follow the naming convention:

Filepath + piece + player + .png

This allows for the correct piece type to be assigned to the correct player.

The movesPanel class displays to the user to know what the current turn is and the statistics of a hovered piece. The JPanel that displays the current turn is named combined. Combined changes color depending on whether Blue or Red is making the move.

Example displays are shown below:



The JPanel that displays the statistics for each piece is currenthp. It changes color depending on the health of the piece. There are three stages:

Green - Healthy

Yellow/Orange - Damaged

Red - Severely Damaged

If ghost is true, currenthp also notifies the user that hovering a piece shows its attack range.

Here are some sample displays of currenthp:



Both these JPanels combined form the movesPanel that is displayed on the east side of the BorderLayout() in GameGUI.

Combining all of the classes from the two trees and all the individual classes, GameGUI is able to run the game according to the actions of the players. Each component from Location to movesPanel help run the move and attack turns of the game and make sure the players have a fair turn rotation.Translating the game mechanics onto the GUI was by far the hardest part of the project and establishing that connection was crucial to running the game.