Christopher Caldwell

Professor Jesse Cecil

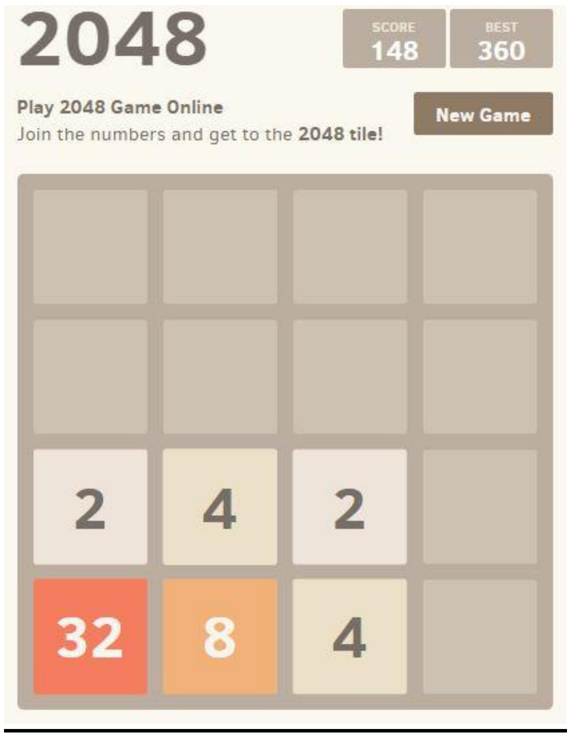
CST-338 – Software Design

10 August 2018

A 2048 Implementation Specification

Background on the 2048 game:

The game 2048 is a relatively new and simple computer game. The player is presented with a 4x4 grid that comes populated with two tiles to start with. By selecting one of the cardinal directions (up, down, left, or right) these tiles can be shifted over towards one of the four edges of the grid. When two tiles of the same value are shifted into each other, they combine to become the sum of their values. The goal of the game is to combine tiles until the 2048 tile is created, at which point the player wins. The player loses if no more moves can be made. At such a time, the entire board would be completely filled with tiles, with no horizontal or vertical adjacent tiles sharing the same value. Upon each movement of the tiles, a new tile is placed on the board. New tiles placed on the board most likely have a value of 2, but there is a small chance the tile may be a value of 4. This is all there is to the game! Another version of the game can be played in a web browser at: http://2048game.com/

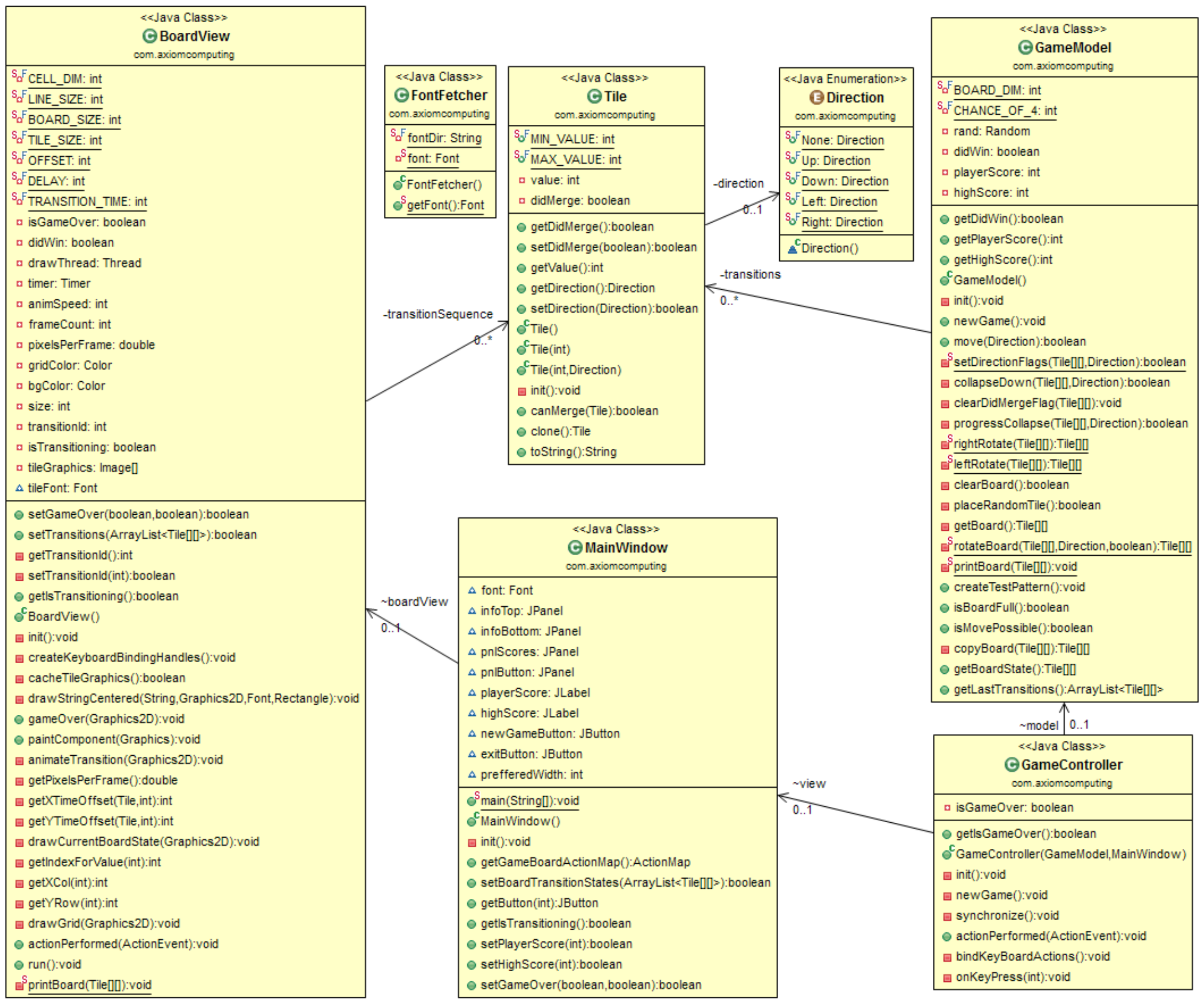


A screenshot from the web version of 2048.

Understanding the Problem:

For our implementation of 2048, we'll be working with the Java Swing Library for our GUI interface, and Java generally. Our goal is to create screen that looks and behaves much the same way as the game located at: http://2048game.com/. We'll be using the **MVC design pattern**, since it lends itself so well to GUI type applications. One of the few files we'll need to obtain is some kind of font for the game to use for drawing string text onto **JPanels**. It really doesn't matter which font is used, but we will need one. We'll be primarily coding three classes. A **GameModel** class will encapsulate the data and game logic. A **BoardView** class will be used to display the state of our game. And a **GameController** class will coordinate between the **GameModel**, the **BoardView**, and listen for events from the user. Along the way, we'll need to create a Tile class that primarily hold the tile values used in the game. First though, we'll be coding the **MainWindow** which is a **JFrame** in which our **BoardView** will reside.

<u>Here is a UML diagram of how the application classes interface with eacho ther:</u>



<u>Phase 1 The MainWindow, Misc Data Structures, and Enumerations:</u>

1. **The MainWindow.** We'll first get the **MainWindow** class up and running. The **MainWindow** extends a JFrame. The **MainWindow's** purpose in life is to host other components, such as the JPanels, JLabels, and JButtons, in addition to our **BoardView** class for viewing the state of the **GameModel**. For this class we'll need:

- *Main()* : Main will be the entry point of the program. It will also instantiate the **GameModel,** the **MainWindow**, and the **GameController**.
- *Various component accessors:* Provide public access to things like JButtons and JLabels that the **GameController** will need to have access to in order to respond to their events.
- *getGameBoardActionMap():* This public accessor will eventually return an **ActionMap** from the **BoardView** object. Our **GameController** will need accessor in order to subscribe to events that are registered in the **BoardView's InputMap**. Both the InputMap and ActionMaps are built-in attributes of the JPanel class.

After completing Phases 2-4 inclusive, you'll want to come back to the **MainWindow** class to add some functionality, such as buttons that can start a **New Game, Exit,** and some JLabels to let the user know what their **Score/HighScore** is.

2. **The Tile class.**

The **Tile** class will encapsulate the data needed to represent a game tile. It does not store a graphical representation of itself, as the **BoardView** handles that job. All variables in the Tile class are private, and should be given getters and setter methods.

- **value.** An int that stores the value of the tile, IE 2, 4, 8, 16, 32, 64 … 2028.
- **didMerge.** A Boolean that stores whether or not this tile is the result of a merge.
- **direction.** An enumeration that we will create next. The direction tells us which way the tiles are moving during a **move()** operation.

Next we describe the methods of the **Tile** class.

- **canMerge()** A public method that returns a Boolean and takes another Tile object as an argument. A tile can merge with another tile if they share the same **value** and neither tile **didMerge** during the current **move().**

- **clone()** A public method that returns a copy of the Tile object. It should create a new Tile object with the exact values as the original. This means the **Tile class** must implement the **Cloneable interface**.
- **toString()** A useful override for debugging.

3. **The Direction Enumeration.**

- **Contains 5 enum values. Up, Down, Left, Right and None.** None is used to indicate that a tile is not moving.

4. **(Optional) FontFetcher**

The **FontFetcher** class accesses the font file that we have chosen for the game. It's job is to load that font once, store it, and serve it up the next time a class calls **getFont().**

Phase 2 The BoardView class:

The **BoardView** class extends the JPane**l** class. On it, we shall paint the current state of the game by overriding the JPanel's **paintComponent()** method. The **BoardView** also implements **ActionListener** and **Runnable** both for animating transit of tiles across the board while not monopolizing the underlying executing thread.

All variables are private, except those exposed by accessors. Private variables include,

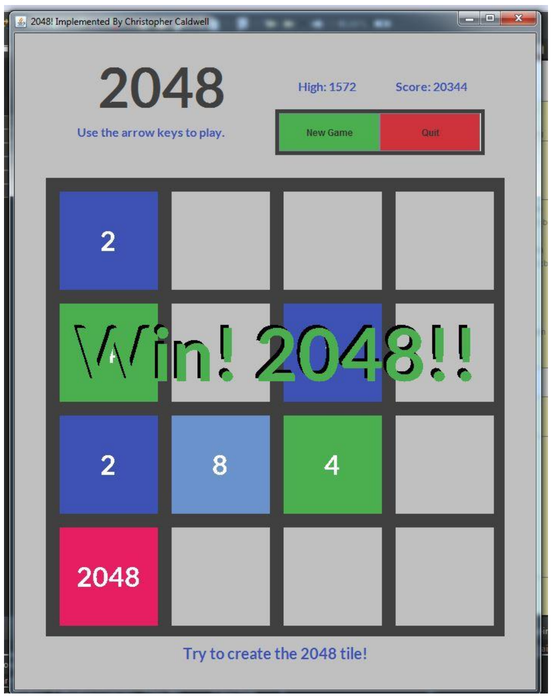
- **A Thread object**. For running the JPanel on.
- **A Timer object**. For controlling how often to retain the JPanel.
- **An int.** For counting how many frames have elapsed.
- **An int.** For tracking board transition state IDs. More on that later.

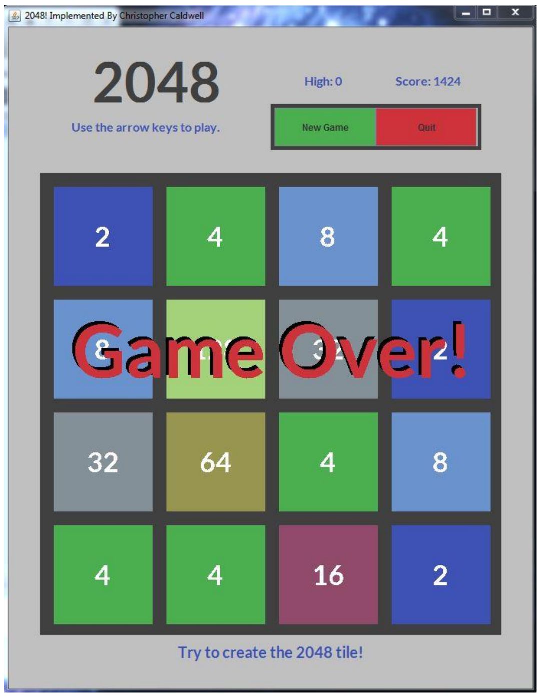
- **Three Booleans.** One for storing whether the player won/lost. One for storing if it is game over. And another for storing whether or not we are transitioning tiles across the board for animation purposes.
- We'll also need to make an ArrayList of 2D **Tile** Arrays. The **Tile** objects will be described more in the **GameModel,** but essentially they hold the value of the tiles, and an enum of **Direction** which the **BoardView** will need in order to know which direction Tiles should move during animating transitions. To be clear, our variables is declared like this: *ArrayList<Tile[][]> transitions;*
- We'll want a private array of **Image** objects. This array will store the cached game Tile images that we'll be generating.

The class itself will need the following methods. <u>**All methods are private member methods and return void unless otherwise specified.**</u>

  - cacheTileGraphics(): This method will generate all the game tiles and store them in the Images array. The tile values to be generated are 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, and finally 2048. The size of the images depends on the dimensions of the grid to be drawn in our JPanel. We just need a solid color background, and our font that we obtained earlier to write the values on our tiles.
  - **drawStringCentered()** is a multipurpose method for helping us draw the text on the center of our game tiles, as well as drawing/displaying GameOver messages to the user. The **drawStringCentered()** function should take as arguments, a **String** to draw, **a Graphics** object, a **Font,** and a **Rectangle** object, representing the target to center the text in.

- **drawGrid()** The method that we'll call to draw the 4x4 grid upon which our game shall be played. It will take as an argument a **Graphics** object.
- **drawCurrentBoardState()** This method will draw our tile Images onto the grid we drew earlier. **drawCurrentBoardState()** gets called along with **drawGrid()** each time our **paintComponent()** method is called.
- Public **paintComponent()** override this method so that when repaint() gets called from anywhere, all the components on our JPanel are redrawn. At the bottom of **paintComponent()** we **Sleep** our **Thread** object for a very short time, before calling **repaint()** again. This allows us to repaint the JPanel so fast that the tile Images appear to slide smoothly across the game grid.
- **gameOver()** We call this method on a game over condition to paint the appropriate win/lose message to the JPanel. This method takes a Graphics object as an argument.


2048! Implemented By Christopher Caldwell
2048
High: 1572
Score: 20344
Use the arrow keys to play.
New Game
Quit
2
Win! 2048!!
2
8
4
2048
Try to create the 2048 tile!


2048! Implemented By Christopher Caldwell
2048
High: 0
Score: 1424
Use the arrow keys to play.
New Game
Quit
2
4
8
4
Game Over!
32
64
4
8
4
4
16
2
Try to create the 2048 tile!

Phase 3 The GameModel class:

The **GameModel** class holds all the data and logic for our 2048 game. It will receive commands via our **GameController** class to be created in the next phase.

Variables for the **GameModel** class include:

- A Boolean called **didWin**, for tracking if the player won/lost.
- An int called **playerScore**, for tracking player score.
- An int called **highScore**, for the high score.
- And a ArrayList of 2D Tile arrays called **ArrayList<Tile[][]> transitions**. In this array we store not only the state of the board, but also each step we take in transforming the state of board after receiving input from the user. These step-by-step snapshots of how the board moves from one configuration to the next, will be helpful to our **BoardView** that doesn't know anything about our game's logic, but is still responsible for animating and displaying what happens in the **GameModel**.

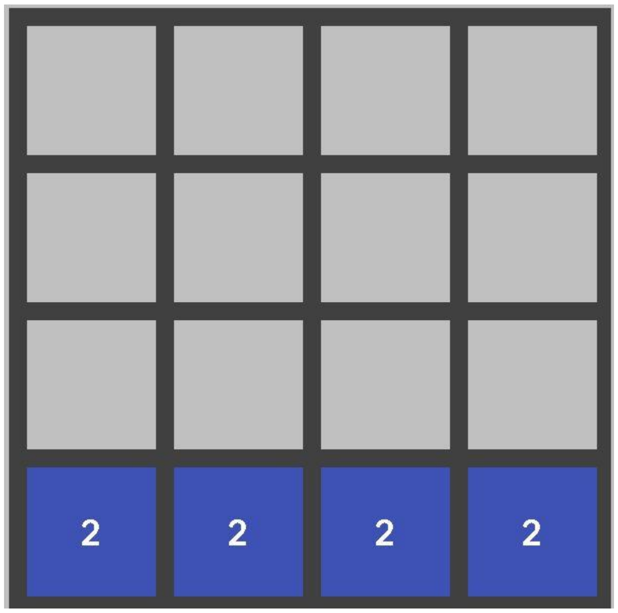
The class itself will need the following methods. **All methods are private member methods and return void unless otherwise specified.**

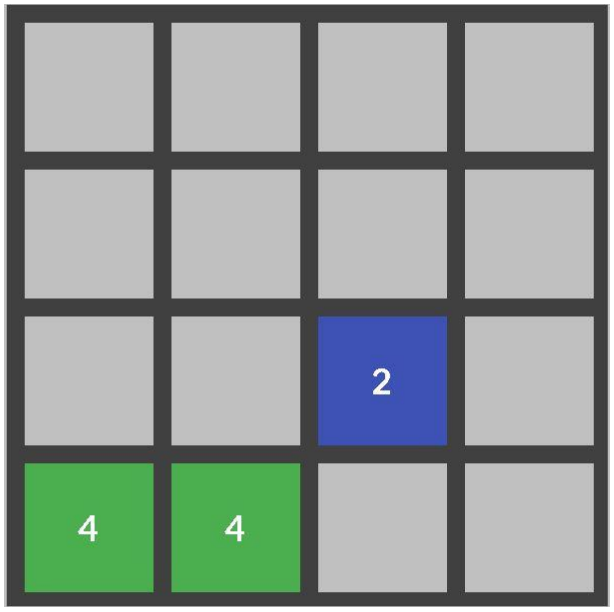
- **placeRandomTile()** This method's job is to place a tile in an open position on the board. It returns a Boolean of false if there is no room to place a tile, true otherwise. Normally we place a tile with a value of 2. However, this is a small chance of something like 1 in 20 that we place a tile with a value of 4.
- **newGame()** This is a public method that can be called to initialize the state of the game for play to begin. This function needs to reset the playing board by making sure it is clear of any tiles, before adding two tiles to the board by calling **placeRandomTile()** twice.

This method is also a good place to update any **highScore** that may have been accrued on the previous play.

- **move()** The move() method is where the game logic is applied. It is the most complex method of our application, and should probably be decomposed into several methods. It returns a Boolean of true if a move changed the state of the board, false otherwise. The **move()** method takes an argument of the **Direction** enumeration. It then applies a movement to the board in the appropriate **Direction.** At each step in the move() operation, we save a snapshot of the board to our **transitions** ArrayList. The first index of the **transitions** ArrayList will store the state we begin the transition at. The last index of the **transitions** ArrayList will store the state of the board we end up at, including the addition of the newest tile after we have called **placeRandomTile()**. Intermediate indexes in the **transitions** ArrayList will store the step-my-step changes to the board. Ultimately, the first index should be transformed into the last index. Anytime we want to reference the most current state of the board, we'll want the last index of **transitions.** In a **newGame()** there are no intermediate states to the **transitions** ArrayList. In that case, the first index is also the last index. At each step of the **move()** we also want to make sure the **Tile**s are flagged with a **Direction** to make sure that the **BoardView** can animate them appropriately. Here is the movement logic in more detail:
    - The user can request of a move Up, Down, Left or Right. The movement should push all Tiles towards the edges of the board in that direction.
    - If two tiles of the same value are pushed up against one another, then they **merge** to become the sum of their values.

- **Be careful**. A Tile once merged in a movement cannot merge again in the same movement. For example. If we have four tiles, each with a value of 2, lined up in a row and we move them in a direction that causes them to merge, we should be left with two tiles each with a value of 4, **not one tile with a value of 8**. To merge again would require another move() operation. **See the example below.**



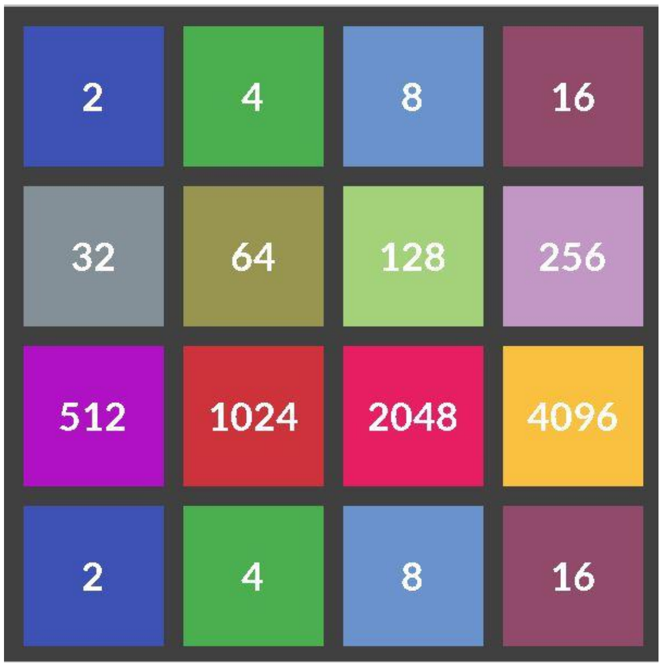


Before a Left Move. **After the Left Move.**

- To ensure that the move() method modifies the game board like you would expect, compare them to the movements on the web version at: http://2048game.com/

- **copyBoard()** The copyBoard method is a private method that takes a 2D Array of Tiles, and copies their values into another 2D array of Tiles to be returned. This method is useful for manipulating the board without changing its state, or for making snapshots of a **move()** operation.

- **getBoardState()** A public method that returns a copied snapshot of the board in the form of a 2D Tile array.
- **getBoard().** A private method that returns a reference to the actual state of the board stored in the transitions ArrayList. Changes to this 2D array of Tiles changes the state of the game board.
- **createTestPattern()** A public method for debugging the **BoardView**. This method populates the board with a tile of each possible value, helping us figure out how to render a board in the **BoardView** as shown below.



The Test Pattern.

- **printBoard()** A private method that takes a 2D Tile array and prints it to the console. Useful for debugging.
- **clearBoard()** Removes all game tiles from the main board. Useful for the **newGame()** method**.**

- **rotateBoard()** Rather than implement the same **move()** logic for each direction, it can be useful to rotate the board one or more times perform the **move(),** then rotate the board back. This method returns a deep copy of the **2D Tile array** argument rotated. It also takes a **Direction** enum, and a Boolean for whether or not we are reversing un-rotating a prior rotation.
- **rotateLeft() / rotateRight()** are methods called by **rotateBoard()** to rotate a 2D array of Tiles by reference 90 degrees to the left or right.
- **getTransitions()** This method returns a deep copy of the **transitions** ArrayList. It can be called by the **GameController** so that the controller can then pass that data into the **GameView** so that the **GameView** can animate the transition.
- **isMovePossible()** This is a public method that returns Boolean. It inspects the current board and determines whether or not a move is possible by the user. First, it checks if the board has any empty (null) locations. If it does, a move is possible and it returns true. If the board is full, it checks to see if any horizontal or vertically adjacent tiles have the same value, if yes, it returns true. Otherwise false.

<u>Phase 4 The GameController class:</u>

The job of the **GameController** is to function as a mediator between the **GameModel** and the **BoardView**. The controller will listen for button clicks, and keyboard events coming from the **BoardView** or the **MainWindow**, and the pass along the appropriate commands to the **GameModel.** As such, the **GameController** implements the **ActionListener** interface. If those commands to the **GameModel** should be displayed, then the **GameController** will set the **transition** data in the **BoardView** which will then be displayed as the **BoardView** continually repaints itself.

Variables for the **GameController** class include:

- A **GameModel**. This **GameModel** is the only instance of GameModel in the application. It is where all the game data and logic reside.
- A **MainWindow.** The **MainWindow** provides direct access to the GUI, within which resides the **BoardView**. In order to affect the **BoardView**, methods must be passed from the **MainWindow** up to the **BoardView** via intermediate getters/setters.
- A Boolean called **isGameOver.** This variable stores whether or not the game has ended. It gets set to true either when the **didWin** flag in the **GameModel** is set to true, or when the **isMovePossible()** method returns false. Once this flag is set, we invoke the **gameOver()** method in the **BoardView** which paints the appropriate win/lose message. We then ignore any user input while the **isGameOver** flag is set to true.

The **GameController** class itself will need the following methods. **<u>All methods are private member methods and return void unless otherwise specified.</u>**

- **GameController()** this public constructor takes a **GameModel** object, and a **MainWindow** object and stores them as private variables. There are no other **GameController** constructors without arguments.
- **Synchronize()** this method copies over the game data from the **GameModel** to the **BoardView v**ia intermediate setter methods within the **MainWindow** class. It should pass along the **highscore**, the current **score**, and the latest sequence of board **transition** states stored in the **GameModel.**
- **newGame()** Here we set out **isGameOver** flag to false. Tell the **GameModel** to initialize a **newGame(),** we **synchronize()** our **GameModel's** data with the view, and tell the view to **setGameOver()** flags to not **isGameOver** and not **didWin**.

- **bindKeyBoardActions()** Here we call the view's **getGameBoardActionMap()** method and subscribe **AbstractAction** methods containing an **onKeyPress()** method to each mapped keyboard press event. Within each abstract method for the key presses we create, we first check to see if the view is transitioning by calling the **getIsTransitioning()** method. If it returns true, we ignore the keypress. We do not want to accept user input while the board is animating to the next state, it would be very jarring.
- **Init()** here we subscribe to buttons in the **MainWindow** for **exit**ing and starting a **newGame()**. The buttons are received by calling the view's **getButtons()** method. We also call **bindKeyBoardActions()** here to subscribe to key presses from the **BoardView** class.
- **onKeyPress()** this method accepts an int which maps to a **keyCode.** Within this method, we first check to see if it **isGameOver**, or if the **view.getIsTransitioning().** If either of those are true, we return immediately. Otherwise, we check to see which key the user pressed. Keyboard input from the user serves as a means to tell the **GameModel** which direction to **move()** in. The **move()** method returns a Boolean for whether or not a move in the **GameModel** was successful. If it was, we call **synchronize().** We also check to see if a move in the **GameModel** is possible by checking the return flag from **isMovePossible().** If not, it's game over and a lose, and we report that to the view by calling **setGameOver()** and passing the appropriate flags. We also check the model's **didWin** flag, and **setGameOver()** and pass a true if we have a win condition.
- **actionPerformed()** Within this overridden method, we pass along the button clicked events from the **MainWindow.** For **newGame()**, and for calling **System.exit(0).**