I. Project Description

Game Description: 2-player bomberman game with a minesweeper twist!

Gameplay: The playing field will be a grid board, with 2 players being free to roam around each square. There will be bombs spawning at random around the map, where each player can 'collect' to fill their bomb pouch. Then, they can proceed to plant a hidden bomb on whichever square that they may like. The objective of the game is to wipe out the other player by planting bombs and coercing them into stepping on the tile with the hidden bomb.

II. Tool Name

Our game will be called ‘Ok, Boomer’.

III. MoSCoW Matrix

Must have:

* Graphical User Interface (GUI)
  + Players will be able to see where they are on the screen.
  + Scoreboard: Players will be able to see the current score.
  + Player Lives: Players will be able to see the number of lives they have remaining.
  + Bomb Pouch: Players will be able to see the number of bombs they have in their pouch.
* Multiplayer support for 2 players.
  + One player will use arrow keys to move, while the other will use ‘wasd’.
* Game Logic / Engine:
  + Game logic will involve updating the state of game.
    - Calculating what to do when a player is bombed.
    - Calculating which coordinates to move the player to when they press a key on the keyboard.
    - Calculating what to do when a player picks up a bomb.
    - All bombs planted will be hidden to both players.
  + Will manage and control game logic.
  + Anything else a game engine is supposed to do.

Should have:

* Visual effects:
  + All players should be able to see where a bomb is planted for the first 2 seconds.
  + All players should have a ‘footstep’ trail following them.
  + Bomb explosion should be shown.
* More items besides bombs:
  + Shield Item: Players who are shielded do not lose a live when they step on a bomb.

Could have:

* Option to change map size
* Character / Item sprites
* More types of bombs:
  + Bombs with different blast area / effects.

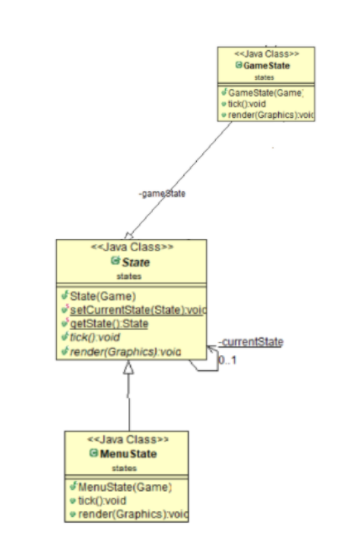
Would have:

* Game audio
* Customise character sprites / trail colours

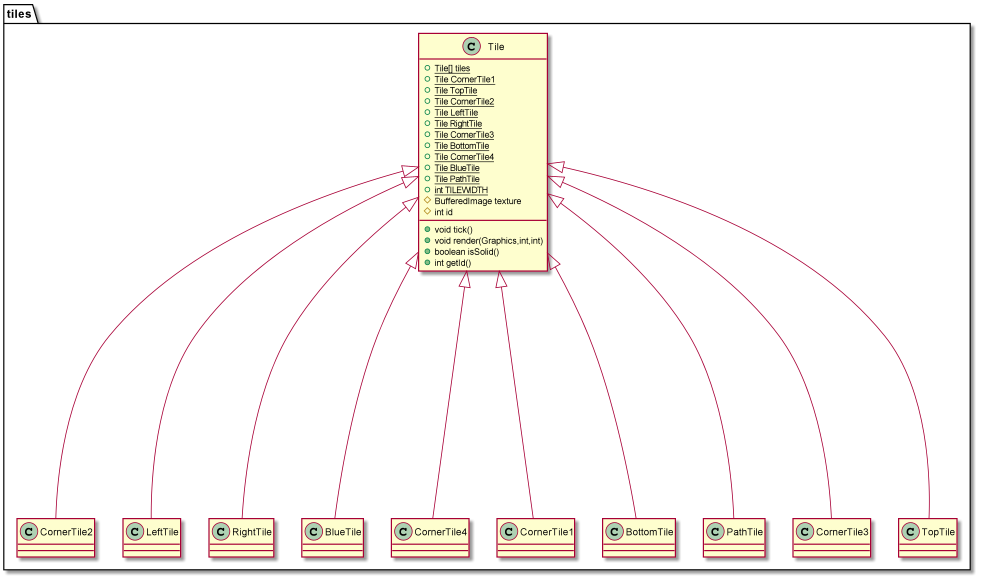
**IV. Polymorphism**

Polymorphism means “many forms” which occurs when there are many classes that are related to each other by inheritance. Inheritance means to inherit attributes and methods from another class. Polymorphism uses those methods to perform different tasks. This allows us to perform a single action in different ways.

In our project, we have a super class called State that has subclasses, the GameState and MenuState, which determines which state the game is in. These subclasses inherit methods such as tick, render and the setCurrentState method. We are planning to add one more subclass called EndState which will utilize the same methods.

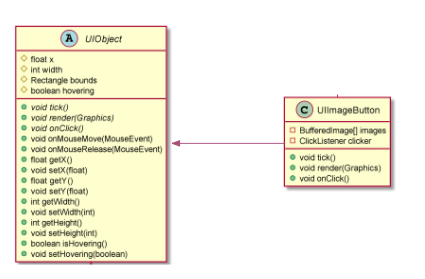


Another example of Polymorphism used in our project would be the super class Tile and subclasses: CornerTile1, CornerTile2, CornerTile3, CornerTile4, BottomTile, PathTile, BlueTile, RightTile, TopTile, LeftTile. The Tile class is responsible for storing information about each tile of the game board. It contains the static id of all the tiles, the tile dimensions, the tick method, and the render method. The subclasses extend the Tile class to obtain their unique static id and return their unique tile texture from the Assets class.



**V. Abstract Classes**

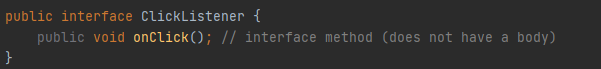
UIObject is a restricted class that cannot be used to create objects. To access it and use its abstract methods, it must be inherited by the subclass. In this case, it is inherited by UIImageButton.

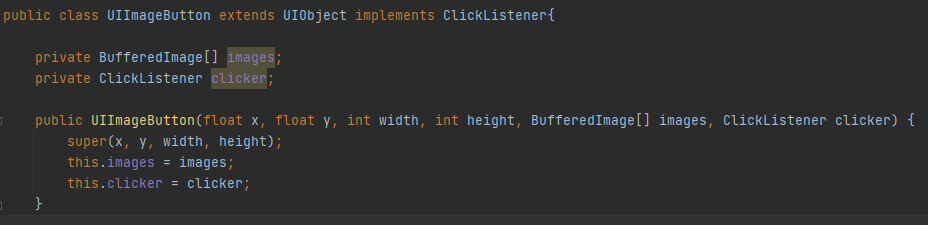


**VI. Interfaces**

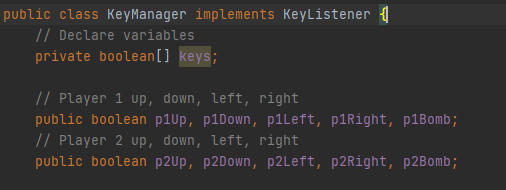
An interface is a completely “abstract class” that is used to group related methods with empty bodies. To access interface methods, the interface must be “implemented” by another class.

In this case with have a ClickListener interface class which listens for mouse clicks. It is implemented by the UIImageButton class. The UIImageButton class generates a User Interface button and implements the ClickListener class to listen for button clicks.





Additionally, the KeyManager class makes use of the KeyListener interface class to access the keyboard keystrokes. 3 classes (Display, Game, and Player) will be utilizing this class to control the game entities with keystrokes.



**VII. Exception Handling**

**Lab 5 Task 3**

We will be implementing Exception Handling in the following classes: (Listed 3 example)

1. Jukebox.java
2. Utils.java
3. Assets.java

Jukebox.java

* Jukebox.java is an interface that manages the playing of game sounds during runtime. The interface methods are called whenever a sound is to be played, and the class constructor takes in the audio file location as a string parameter. Exception handling is required as users who have access to the game files are able to modify / delete the resources required to play the sounds. If there is no error handling, the game will crash whenever the audio files are called.

Utils.java

* Utils.java is a utility class that consists of several helper functions that are called by other classes to help perform different functions. For example, World.java calls the helper functions in Utils.java to help parse the text from the world loading file into usable information for Tile.java. Exception handling is required as users may tamper with the world-loading file to obtain different world maps. Invalid input will cause the game to crash when the map is loading.

Assets.java

* The loading of images are handled in Assets.java The image will be stored in a BufferedImage object. The image file path will be parsed as a string when we load the image. Exception handling is required to ensure that the image file path is valid, and that the input file is a valid image file. If there is no error handling, the game will crash whenever the class is called.