SOFTWARE ARCHITECTURE DOCUMENT

ECSE 321 Introduction to Software Engineering
Term Project Phase Two Deliverable

Simon Ho, Raymond Guo, Henry-Michel Cantave, Jeffrey Tichelman, Laurent Jacob, Frank Luong
HAIBOMB GROUP

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Purpose

The purpose of this document is to provide a description of the high level architecture of the game at this pre-development stage. It attempts to define the packages and classes and their interaction, as well as associated methods and attributes. It is intended to provide a blueprint as we move onto the next phase of actually building the game. Thus, it is only an initial plan and it must be noted that the details of this document are subject to adjustment in development. The rationale behind most of the specific design choices was to attempt to maximize modularity for efficient development in our relatively small group.

1.2 Scope

This Software Architecture Document presents an architectural description of the game Bomberman, to be developed by HAIBOMB GROUP (Simon Ho, Raymond Guo, Henry-Michel Cantave, Jeffrey Tichelman, Laurent Jacob, and Frank Luong) for the Introduction to Software Engineering term project. The game is being built to showcase the software design skills learned in class as well as to provide an entertaining rendition of the popular arcade game Bomberman. The rules and gameplay are mostly standard and are described in the Software Requirements Specification, which this document continues from. The following will not describe the implementation plan in detail but was created to display a high level view of the game's design.

1.3 Definitions, Acronyms & Abbreviations

Board: the screen on which the game is played; tiled field of play

Bomb: placed on gameboard by user or AI character and explodes after a time limit, destroying character's and objects within a range

Map: essentially the same as board but particularly referring to the color scheme and layout

PowerUp: item which can be acquired and deployed on game board for different effects such as speed boost, added bomb range, etc.

Tile: one square of the board's grid layout

1.4 References

The diagrams contained in this document were created using the chrome application draw.io, available in the Chrome Web Store or at www.draw.io.

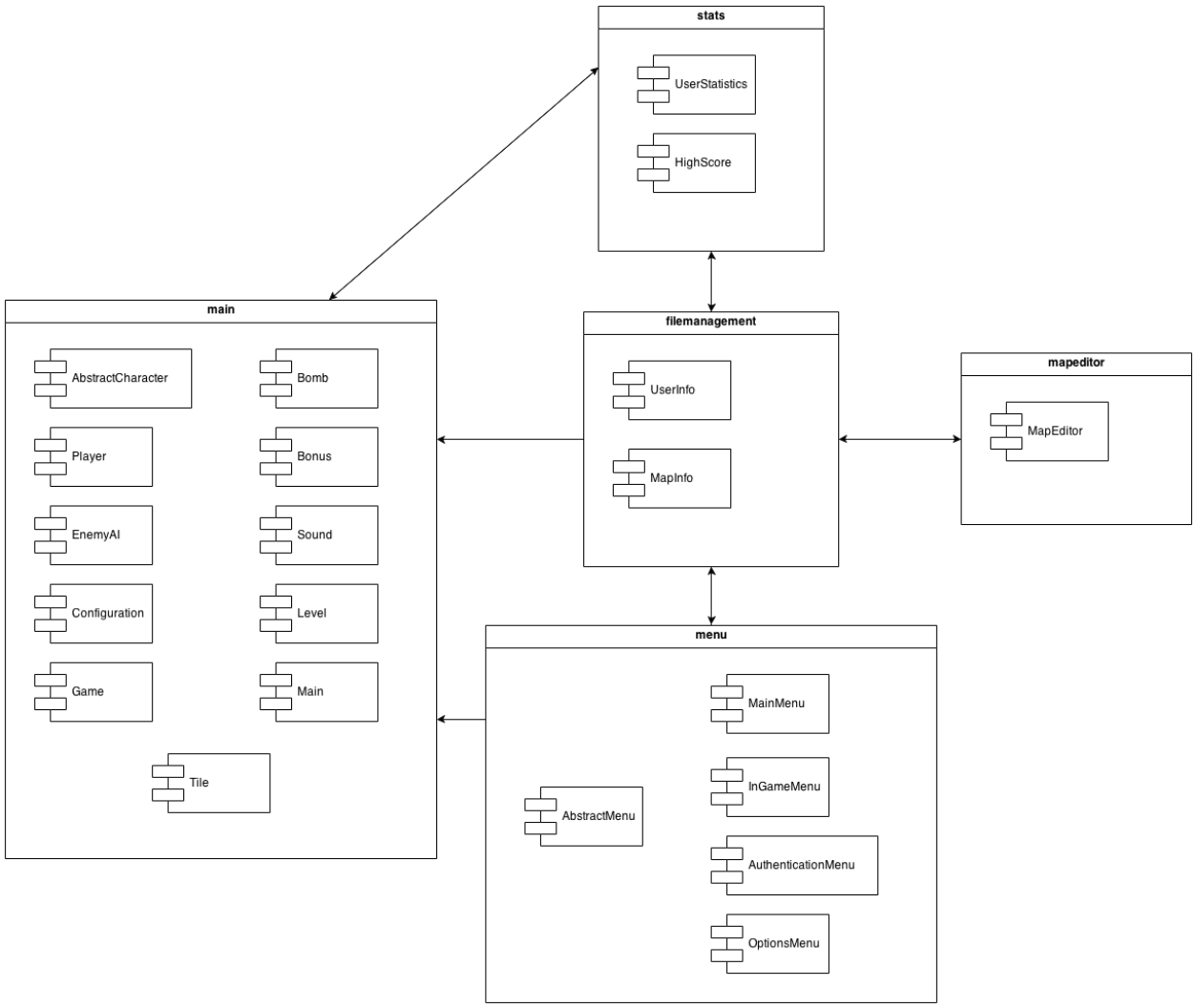
1.5 Overview

This document portrays an architectural view of our intended design for the game Bomberman. Section 2 provides views at various levels using the standard Unified Modeling Language (UML) conventions, besides multiplicity for which a legend is provided. Section 3 then describes the various classes and their purpose. As noted, the details of the design are subject to change but the high-level architecture should be maintained in development.

2. VIEWS

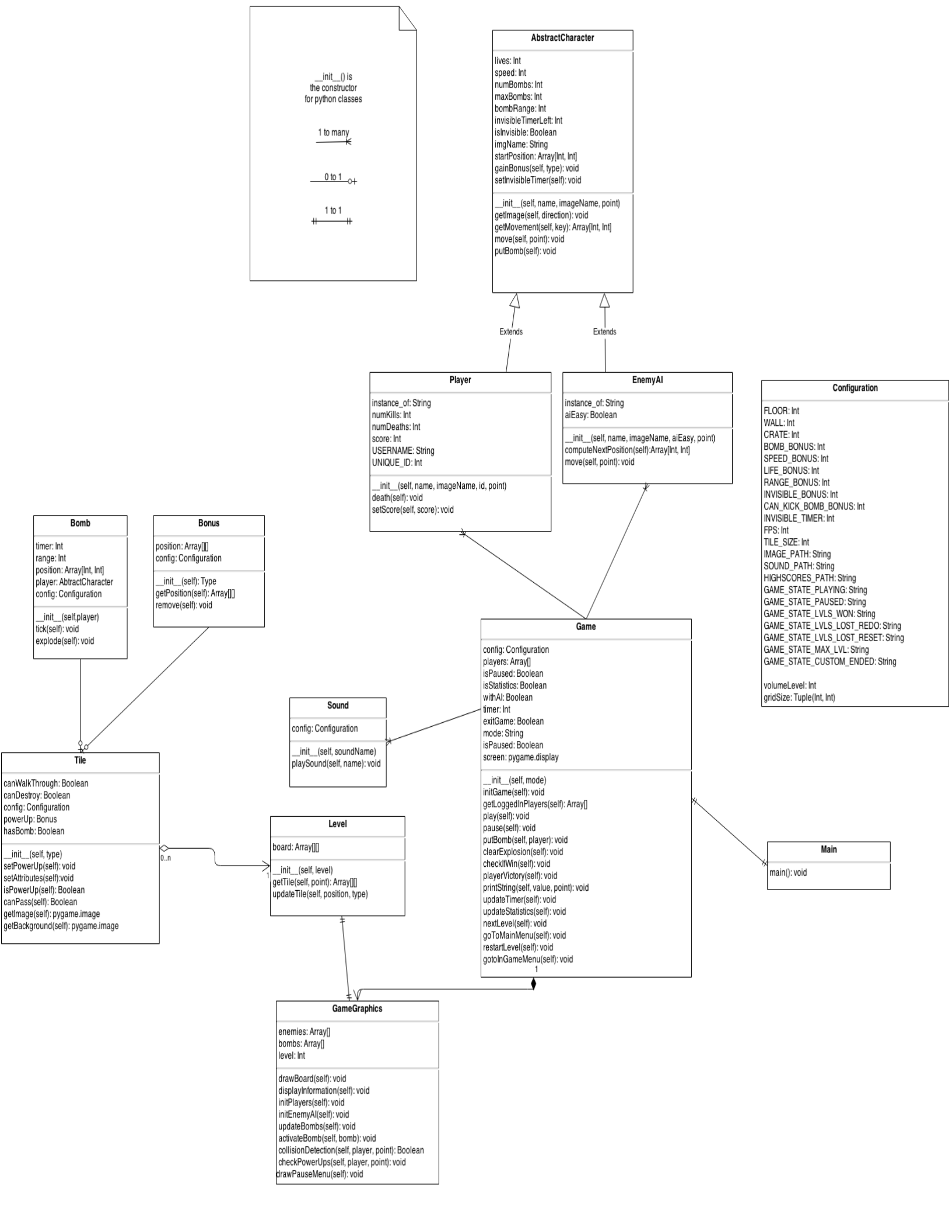
2.1 Package View

This diagram depicts the packages and their relationships:



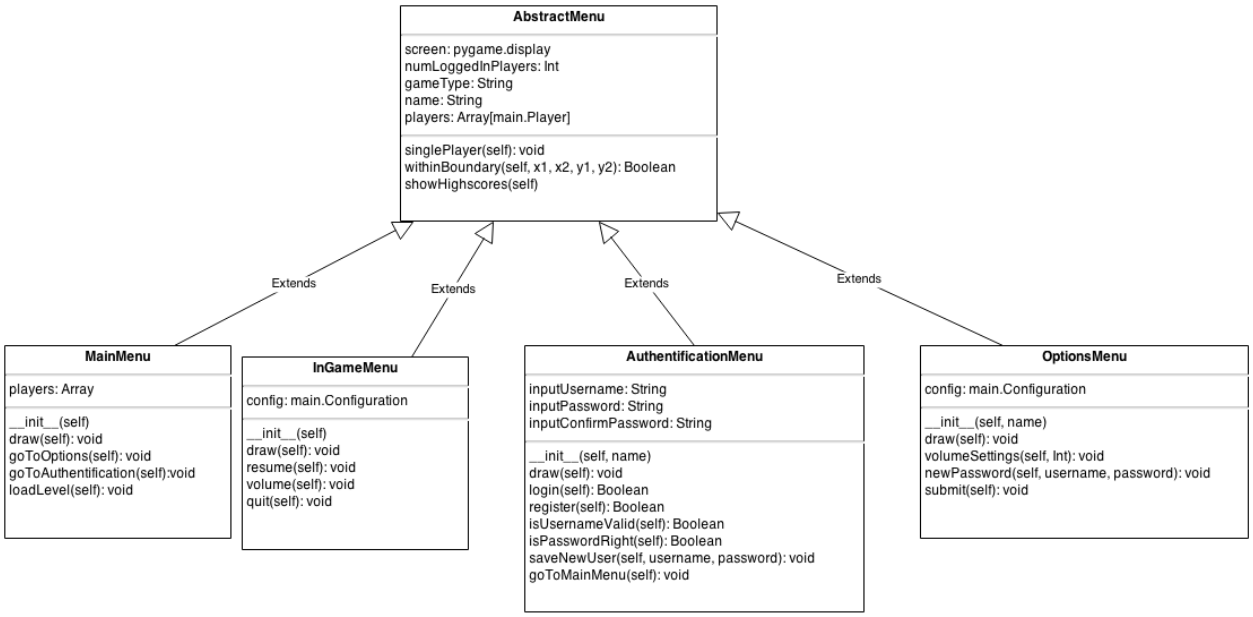
2.2 Main UML

The diagram on the following page describes the classes involved during gameplay:


__init__() is
the constructor
for python classes
1 to many
0 to 1
1 to 1
AbstractCharacter
lives: Int
speed: Int
numBombs: Int
maxBombs: Int
bombRange: Int
invisibleTimerLeft: Int
isInvisible: Boolean
imgName: String
startPosition: Array[Int, Int]
gainBonus(self, type): void
setInvisibleTimer(self): void
__init__(self, name, imageName, point)
getImage(self, direction): void
getMovement(self, key): Array[Int, Int]
move(self, point): void
putBomb(self): void
Extends
Extends
Player
instance_of: String
numKills: Int
numDeaths: Int
score: Int
USERNAME: String
UNIQUE_ID: Int
__init__(self, name, imageName, id, point)
death(self): void
setScore(self, score): void
EnemyAI
instance_of: String
aiEasy: Boolean
__init__(self, name, imageName, aiEasy, point)
computeNextPosition(self):Array[Int, Int]
move(self, point): void
Configuration
FLOOR: Int
WALL: Int
CRATE: Int
BOMB_BONUS: Int
SPEED_BONUS: Int
LIFE_BONUS: Int
RANGE_BONUS: Int
INVISIBLE_BONUS: Int
CAN_KICK_BOMB_BONUS: Int
INVISIBLE_TIMER: Int
FPS: Int
TILE_SIZE: Int
IMAGE_PATH: String
SOUND_PATH: String
HIGHSCORES_PATH: String
GAME_STATE_PLAYING: String
GAME_STATE_PAUSED: String
GAME_STATE_LVLS_WON: String
GAME_STATE_LVLS_LOST_REDO: String
GAME_STATE_LVLS_LOST_RESET: String
GAME_STATE_MAX_LVL: String
GAME_STATE_CUSTOM_ENDED: String
volumeLevel: Int
gridSize: Tuple(Int, Int)
Bomb
timer: Int
range: Int
position: Array[Int, Int]
player: AbtractCharacter
config: Configuration
__init__(self,player)
tick(self): void
explode(self): void
Bonus
position: Array[][]
config: Configuration
__init__(self): Type
getPosition(self): Array[][]
remove(self): void
Game
config: Configuration
players: Array[]
isPaused: Boolean
isStatistics: Boolean
withAI: Boolean
timer: Int
exitGame: Boolean
mode: String
isPaused: Boolean
screen: pygame.display
__init__(self, mode)
initGame(self): void
getLoggedInPlayers(self): Array[]
play(self): void
pause(self): void
putBomb(self, player): void
clearExplosion(self): void
checkIfWin(self): void
playerVictory(self): void
printString(self, value, point): void
updateTimer(self): void
updateStatistics(self): void
nextLevel(self): void
goToMainMenu(self): void
restartLevel(self): void
gotoInGameMenu(self): void
1
Sound
config: Configuration
__init__(self, soundName)
playSound(self, name): void
Tile
canWalkThrough: Boolean
canDestroy: Boolean
config: Configuration
powerUp: Bonus
hasBomb: Boolean
__init__(self, type)
setPowerUp(self): void
setAttributes(self):void
isPowerUp(self): Boolean
canPass(self): Boolean
getImage(self): pygame.image
getBackground(self): pygame.image
0..n
1
Level
board: Array[][]
__init__(self, level)
getTile(self, point): Array[][]
updateTile(self, position, type)
Main
main(): void
GameGraphics
enemies: Array[]
bombs: Array[]
level: Int
drawBoard(self): void
displayInformation(self): void
initPlayers(self): void
initEnemyAI(self): void
updateBombs(self): void
activateBomb(self, bomb): void
collisionDetection(self, player, point): Boolean
checkPowerUps(self, player, point): void
drawPauseMenu(self): void

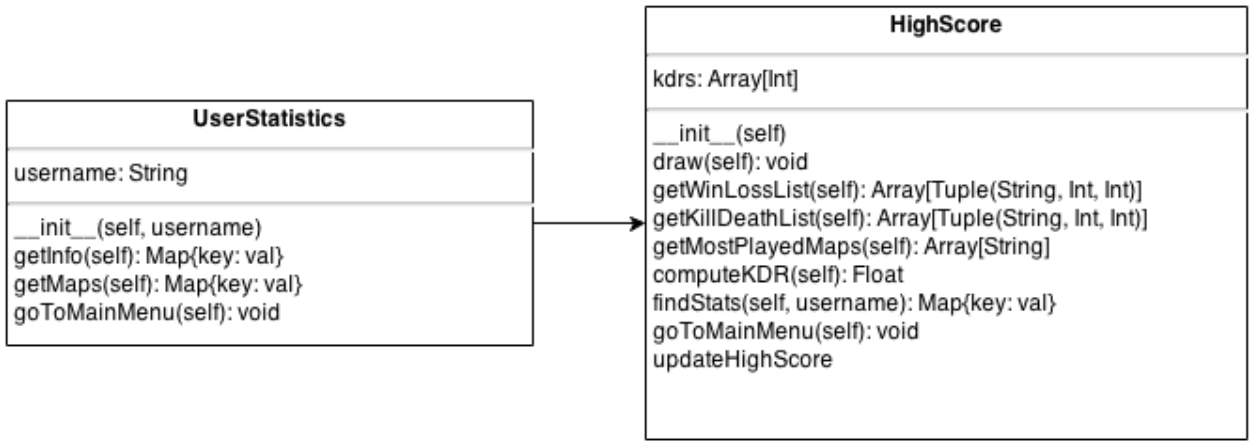
2.3 Menu UML

This diagram shows the menu's within our game and their relationship with a general, abstract menu:



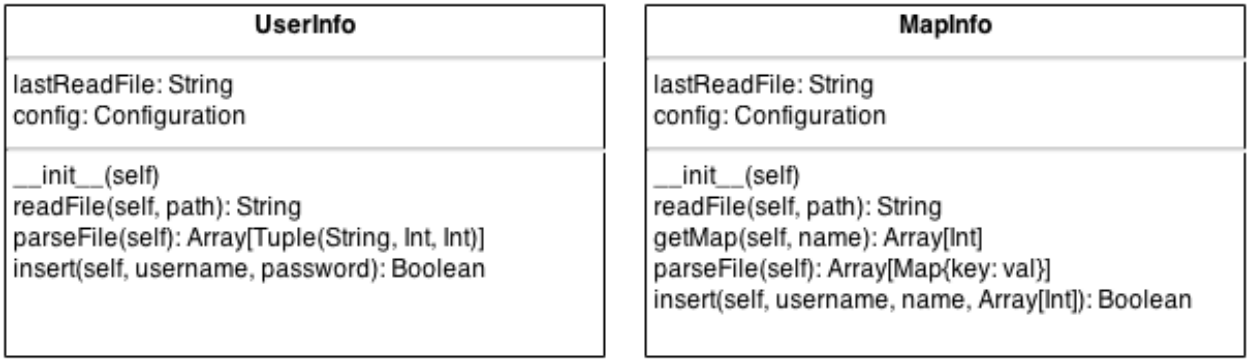
2.4 Stats UML

This diagram depicts how user's stats are tracked on the back end:



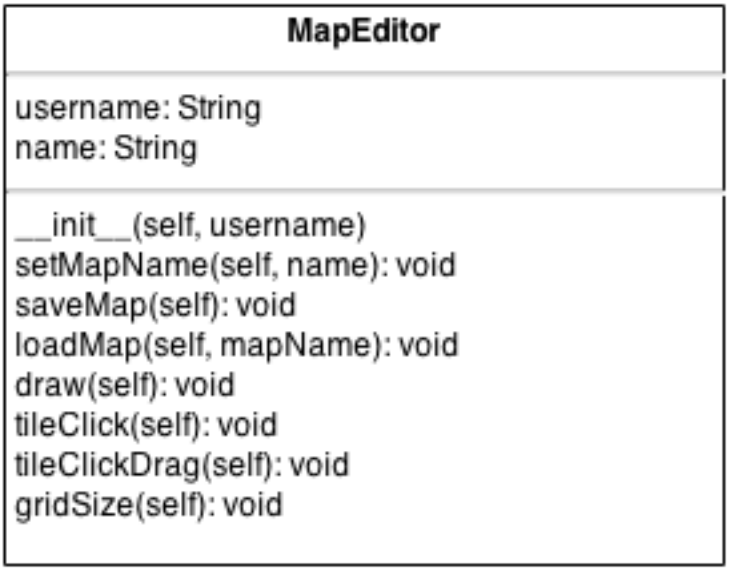
2.5 File Management UML

The below encapsulates the information associated with users and maps:



2.6 Map Editor UML

The following provides a rather general depiction of our map editor class, a feature we hope to include if time permits:



3. CLASS DESCRIPTIONS

3.1 Main Package

3.1.1 Abstract Character

This is an abstract class used to define the methods for the in-game characters to inherit from. Basic methods such as move() and putBomb() will need to be implemented for all characters.

3.1.2 Player

This class inherits from the Abstract Character class. It will override its movement methods and will pass in its own set of controls. Its field variables will contain statistics information to be stored under the player's username. This class is the representation of the real player's character.

3.1.3 Enemy AI

This will also inherit from the Abstract Character class. It will override its movement methods with a path finding algorithm and its own logic. This class does not need to store and information in statistics.

3.1.4 Configuration

This is a class used to store all constant variables in the game. It will act as a small utility class to provide unique values for bonuses, sizes, game states and obstacles. None of these fields must be changed, as they are constants (except for volumelevel and gridsize).

3.1.5 Main

This will be the class that contains the main method. It will create an instance of the Game.

3.1.6 Game

This class is the heart of the Main package. It will create and define all other classes such as Player, AI, Tile, Level, etc. It will also contain the main game loop, where every movement, rendering and game logic methods should be called in this loop.

3.1.7 Tile

This class represents a single tile in the game. It will define whether it contains a power-up, bomb or nothing. It uses the Bomb and Bonus class to identify its state.

3.1.8 Level

This class' purpose is to create a board suited for the selected map size and level. It will create an array of tiles which will represent the game map.

3.1.9 Game Graphics

This will be used to render all game components (Board, Players, Enemies, Bombs, Powerups). It will take its image files from each class and represent it graphically in the game window.

3.1.10 Bomb

This will contain attributes and methods for the bomb. It will have a timer and range variables that are modified as the player acquires powerups. The explode method will be called as soon as the timer is done.

3.1.11 Bonus

This class will be used to generate powerups across the map. Each tile may have a certain powerups which the position is defined in this class.

3.1.12 Sound

This class will be used for music and sound effects in the game. It will also be used to change in-game volume.

3.2 Menu Package

3.2.1 Abstract Menu

This is an abstract implementation of a menu, where all menus inherit from. This will act a template for all menus to use.

3.2.2 Main Menu

This menu allows navigation to the Game, to Authentication Menu and to the Options Menu. It will be the menu shown as soon as the user logs in.

3.2.3 Authentication Menu

This menu allows the users to log in/out from the game. It also contains a register form where new users can use to create accounts. It will be the first menu shown when the program starts.

3.2.4 Options Menu

This menu contains game options for the users to modify (volume, password).

3.2.5 In-Game Menu

This is the only menu that can directly be accessed while in-game. It contains methods to quit the game (and go back to main menu) and modify game options.

3.3 Stats Package

3.3.1 User Statistics

This class will interface the Main package to the file management package. It will take in all user information from the current game and send it to the file management package.

3.3.2 High Score

This class will take in values from the file management package and create a highscore list. It will sort the players in a type of ranking depending on selected attributes (KDA, Win/loss ratio, etc)

3.4 File Management Package

3.4.1 User Info

This class acts as a database manager. It will take in user statistics from the game and will append or write them to the file. Also, it can look up values under a certain username to retrieve its user statistics.

3.4.2 Map Info

This will save all custom maps created and write or append them to a file. Custom maps can be loaded by reading from the file.

3.5 Map Editor Package

3.5.1 Map Editor

This class allows the creation of custom maps for users. These maps will only be saved under Player 1's username. Saving and loading custom maps for modification will be handled in this class. The editor will then pass its map information to the file management package.