RAMones Co.

# Disgruntled Avians Project

**(DAP)**

Prepared by:

Stuart Sessions

Bryan Moreno

Dan Phuong

Software Design Document

Release Date September 6, 2022

## Executive Summary

The Disgruntled Avians Project is an interactive game modeled after the popular Angry Birds™ game on mobile app stores. Due to DAP's minimalistic interpretation of physics, DAP will not have demanding hardware requirements, thus allowing it  to run on a variety of desktop hardware platforms. Coding this in Java will further expand this cross-platform availability. Future functionality will include an expansion into the mobile market, micro-transactions, player vs. player game modes, and some of these updates will focus specifically on the physics modeling of DAP.

This document provides technical information regarding the software design of DAP.

## Document Versioning

| Date | Owner | Comment |
|---|---|---|
| 09/06/22 | Dan Phuong | Draft Documentation Creation |
| 09/07/22 | Stuart Sessions | GameEngine and GameSave System Designs |
| 09/07/22 | Dan Phuong | GUI and Feature Matrix |
| 09/12/22 | Stuart Sessions | Updated System Design |
| 09/12/22 | Dan Phuong | Updated System Design |

## Project Description

The Disgruntled Avian Project (DAP) is meant to be an interactive video game similar to other projectile motion games such as Angry Birds. RAMones are looking to build the framework for a fun, expandable game that can in the future be sold to bigger gaming companies.

DAP will not require internet connectivity, and will run locally on the computer hardware that it is installed in. DAP will not have steep hardware requirements  and will be able to run on a variety of operating systems (ex. Windows or Mac).

We plan to bring a vibrant look and feel to our game application. DAP aims to be easy and intuitive to play. This approachable game design will make it easy to expand into different demographics and have wide appeal with ample range for merchandising opportunities.

Interacting with the DAP GUI should be reminiscent of its more popular counterpart Angry Birds™, as a similar way of interacting, the classic click-and-drag game, will be used in this application. Ideally, in interest of ease of use and simplicity, DAP will be able to run in a similar fashion to a Desktop application, with a clickable file that launches the game, ready for the player to dive in to the exciting and tantalizing world of the Disgruntled Avians project. Finally, and most importantly, our development team at The RAMones are excited for all people ages 4-65[1] to *Dare to Dap*™ with us.

[1] Target Demographic

## Features

The feature matrix enumerates the features requested for the project and the discussion section provides details regarding the intent of the feature. The ids will be used for traceability. Features that all stakeholders have agreed can be removed should strike-through the feature id and have a comment added to discuss the feature being dropped.

Priority Codes:

H - High, a must have feature for the product to be viable and must be present for launch

M - Medium, a strongly desirable feature but product could launch without

L - Low, a feature that could be dropped if needed

### Features Matrix

| ID | Pri | Feature Name | Comment | BRD ID |
|---|---|---|---|---|
| P.1 | H | Language | Implemented in JAVA | S.1, S.2 |
| P.2 | M | Bundling | Bundled as a JAR file | S.2 |
| I.1 | H | GUI | | U.X.1, U.X.2, E.6 |
| C.1 | H | Game Loop | | E.2, E.14 |
| C.2 | H | Save File | | E.1, E.6 |
| C.3 | H | Level/Map | | E.3 |
| C.4 | H | Physics Engine | | E.5 |
| C.5 | H | Level Objects | | E.3, E.4, E.7,E.12 |
| C.6 | H | Avian Launcher | | E.9, E.10 |
| T.1 | H | Error Handling | | U.X.3 |

## Feature Discussion

### P.1 - Language

DAP must be a multi-platform and standalone application that can run. For this reason, JAVA will be the language of choice due to its JVM, which compiles directly to ByteCode, making the application independent from the operating system the application is running on. Furthermore, JAVA's GUI behavior is typically consistent across different platforms.

### P.2 - Bundling

JAVA's JAR feature allows for bundling of executable code. This feature is also able to run on multi platforms and and many operating systems, enabling us to distribute the program without having to port into different operating systems.

### I.1 - GUI

JAVA's Swing GUI framework will be implemented since GUI implementation will be consistent throughout differing platforms and because it has numerous features that other frameworks don't (e.g. AWT).

### C.1 - Game Loop

The basic game loop for our application follows this:

1. GUI displays level
2. Player launches Avian
3. Physics Engine executes the launch
4. Process all collisions
5. Process all resulting velocity and direction from collided objects
6. Execute previous processes
7. Return to 1

### C.2 - Save File

A save file captures the current game state at the time to a file on the user's computer. This should prompt the user to choose the location to save the file in. This file can then be used to load in the game state to resume playing. When saving and loading a file, the user will enter their player name in order to save and write to the file.

## C.3 - Levels

This game consists of level GUI's that the player will interact with to play the game. Each level is a GUI that contains an Avian and objects to hit. The player will move their mouse onto the Avian, click and drag, and then release to launch the Avian. This will then simulate the Avian's trajectory and the collisions it has with the objects it has collided with.

## C.4 - Physics Engine

The core of this game's logic is contained within the Physics Engine. This engine will hold the logic for how objects interact with one another and their resulting behavior after the interaction. Because of the game's heavy dependence on collisions, it's imperative that the Physics Engine handle and run the interactions correctly.

## C.5 - Level Objects

There are three objects to keep track of: the Avian, the Block objects, and the Objective. The avian is what the player will be interacting with to launch. The block objects are the obstacles within the GUI that the Avian will interact with. The Objective is the object that the user has to strike with the Avian.

## C.6 - Avian Launcher

The Avian Launcher is what the user will use to actually launch the Avian. Logically, this works by having the user move their mouse over the Avian, clicking and dragging, then releasing the launch the Avian.

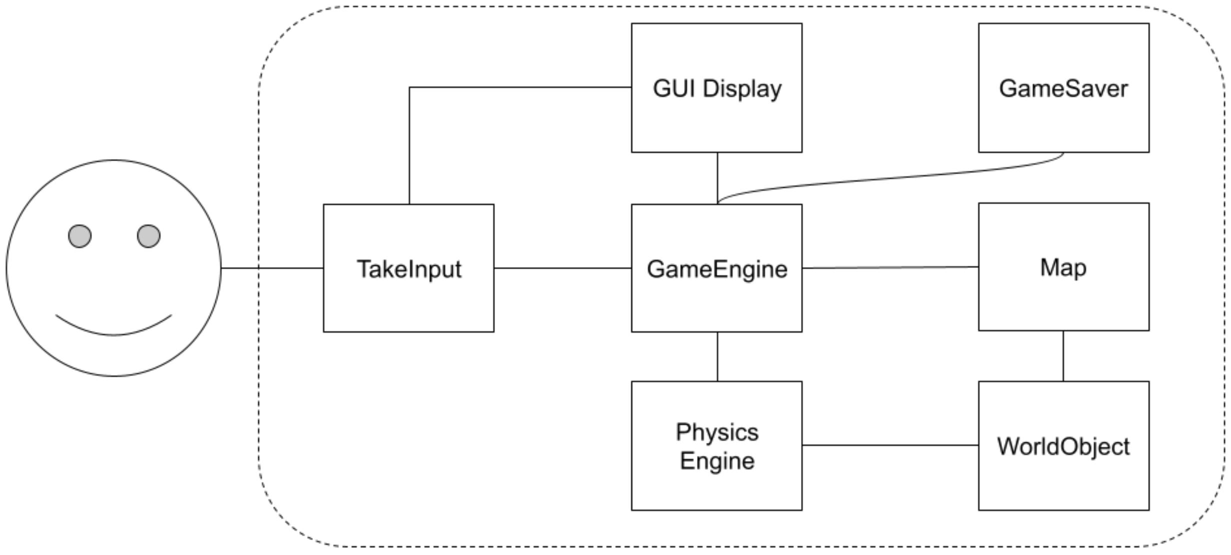
## T.1 - Error Handling

For technical bugs that occur while the application is running, Pop-up dialogues will be presented to the user to notify them of the bug. This will then give them the option to redirect prior to where the error occurred (such as entering the wrong player name for a game save) or giving them the option to generate a more detailed report of the error that has occurred.

# System Design

## Architecture Overview

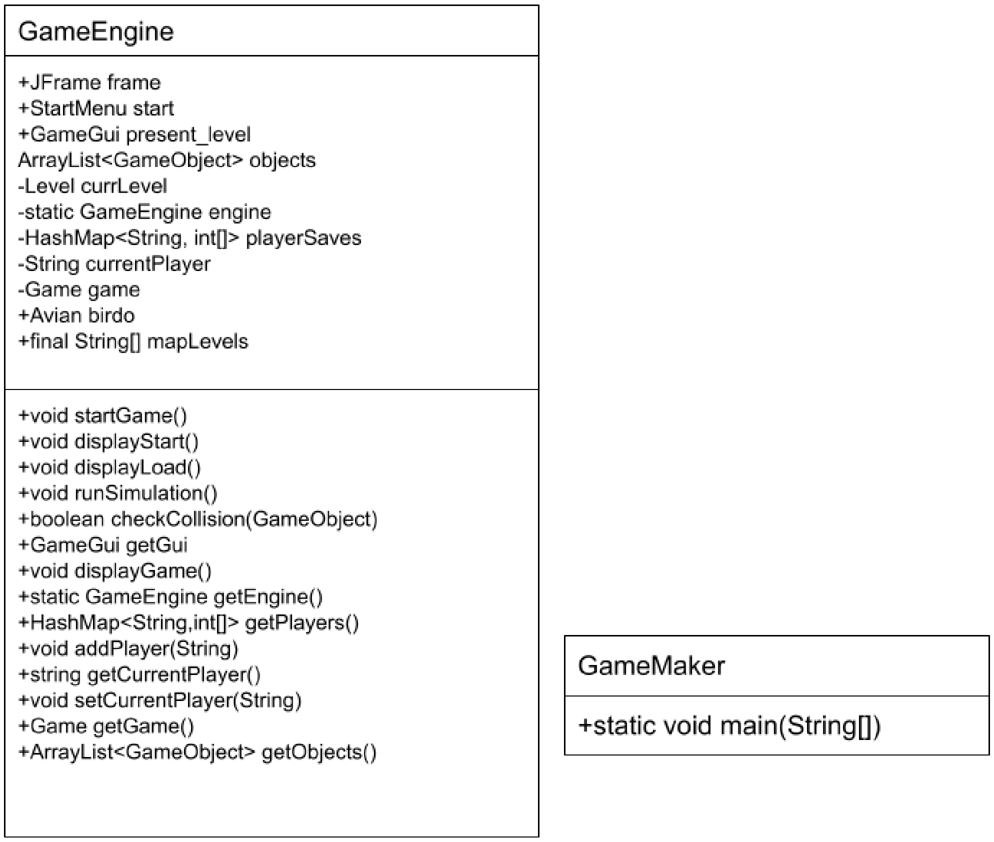
### High-level System Architecture



## Detail Design

### GameEngine

Contains the information to start and run the game, and contains the engine that stores relevant game data to send to the GUI. The main controller for the application, and main checkpoint in the game loop.


GameEngine
+JFrame frame
+StartMenu start
+GameGui present_level
ArrayList<GameObject> objects
-Level currLevel
-static GameEngine engine
-HashMap<String, int[]> playerSaves
-String currentPlayer
-Game game
+Avian birdo
+final String[] mapLevels
+void startGame()
+void displayStart()
+void displayLoad()
+void runSimulation()
+boolean checkCollision(GameObject)
+GameGui getGui
+void displayGame()
+static GameEngine getEngine()
+HashMap<String,int[]> getPlayers()
+void addPlayer(String)
+string getCurrentPlayer()
+void setCurrentPlayer(String)
+Game getGame()
+ArrayList<GameObject> getObjects()
GameMaker
+static void main(String[])

Class Diagram

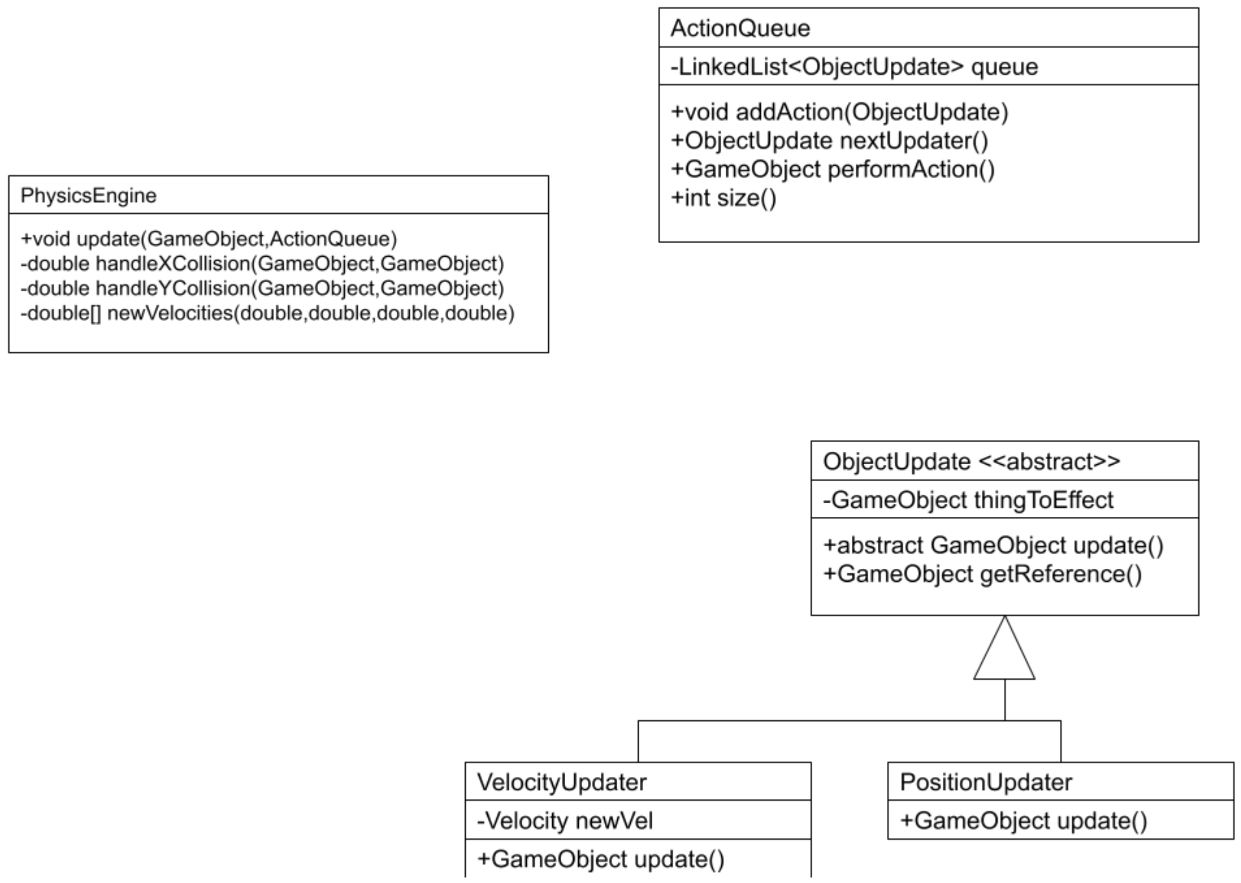
### GameEngine

The GameEngine class acts as the central database for the program, containing methods that run the game as well as holding references to important variables. This Singleton class controls the flow of the program by initializing the game in startGame() and interacting with the GUI to take input, modify objects, and then update the display.

### GameMaker

The GameMaker starts the execution of the program, creating the GameEngine object to run the bulk of the program, and populating it with Map data.

## Physics Engine


ActionQueue
-LinkedList<ObjectUpdate> queue
+void addAction(ObjectUpdate)
+ObjectUpdate nextUpdater()
+GameObject performAction()
+int size()
PhysicsEngine
+void update(GameObject,ActionQueue)
-double handleXCollision(GameObject,GameObject)
-double handleYCollision(GameObject,GameObject)
-double[] newVelocities(double,double,double,double)
ObjectUpdate <<abstract>>
-GameObject thingToEffect
+abstract GameObject update()
+GameObject getReference()
VelocityUpdater
-Velocity newVel
+GameObject update()
PositionUpdater
+GameObject update()

## PhysicsEngine

Physics Engine will handle the logic for the collision and resulting velocities after the collision. This is designed around an observer pattern and a command pattern that handles actions and requests to be parsed throughout the GameEngine

## ActionQueue

ActionQueue queues the actions to be handled later. This is apart of the Command pattern utilized in the Physics Engine.

## ObjectUpdater

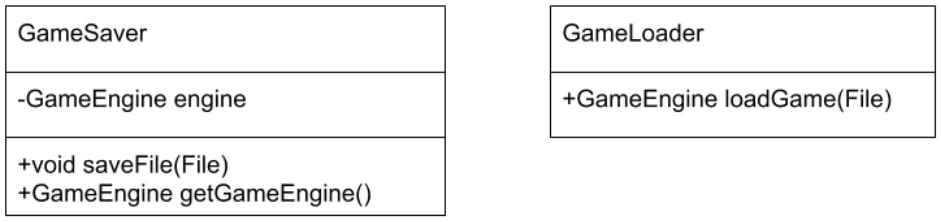
ObjectUpdater takes in an Object to be updated. This is an Abstract Class.

### VelocityUpdater

This updates an object’s velocity through the ObjectUpdater.

### PositionUpdater

This updates an object’s position through the ObjectUpdater.



Class Diagram

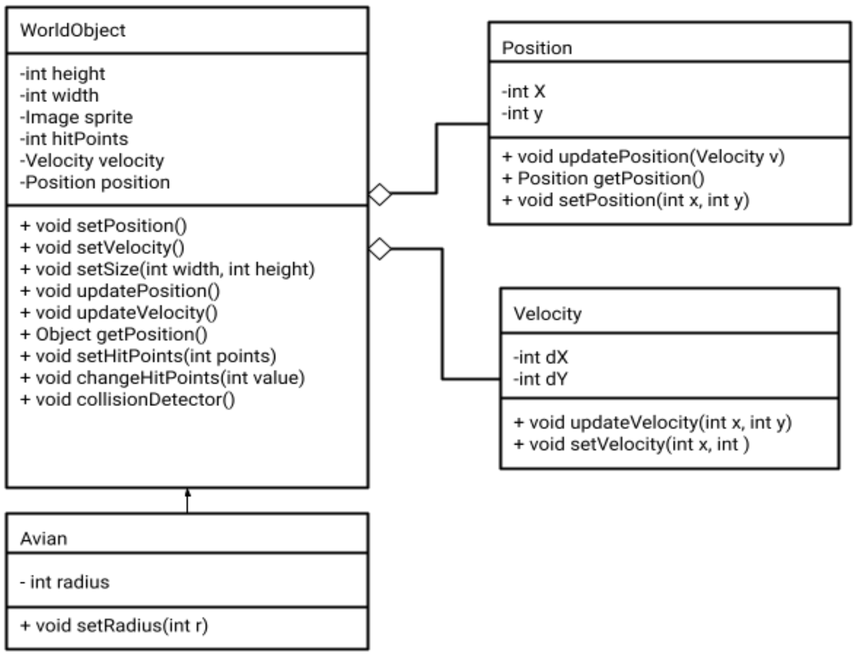
### GameSaver

Takes the current game engine and writes its information onto a File that can be accessed and reloaded by the GameLoader.

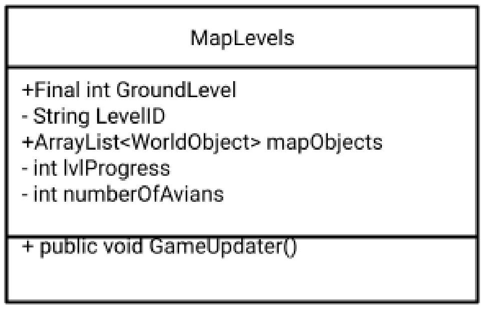
### GameLoader

The loadGame(File) method searches for a save file that matches the parameter, and if one exists, contains the logic to restore the GameEngine to the state of the read file.

## GameObjects and Map



Class Diagram



## GameObject

This is the class responsible for making all of the objects that the player and the physics engine can interact with. Various methods set and get fields like Width, Height, Velocity, Position, and the Sprite, or Image representation. The function setHitPoints() and changeHitPoints alter the Object’s representation of health. The function collisionDetecter will be able to detect if the object is experiencing a collision, and move accordingly.

## Position

This class will represent objects in the game’s center position, or frame of reference. It can be modified with updatePosition() and setPositiontains two().

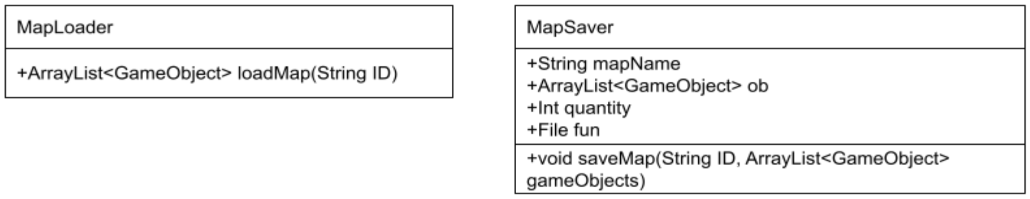
## Velocity

This class represents a change in position. It is meant to directly interface with a WorldObject's position. UpdateVelocity() and setVelocity() are used to modify and set velocity values.

## Avian

An object that extends the WorldObject class. This class has a value for radius since it is planned for it to be a circle.

## Map


MapLoader
+ArrayList<GameObject> loadMap(String ID)
MapSaver
+String mapName
+ArrayList<GameObject> ob
+Int quantity
+File fun
+void saveMap(String ID, ArrayList<GameObject> gameObjects)

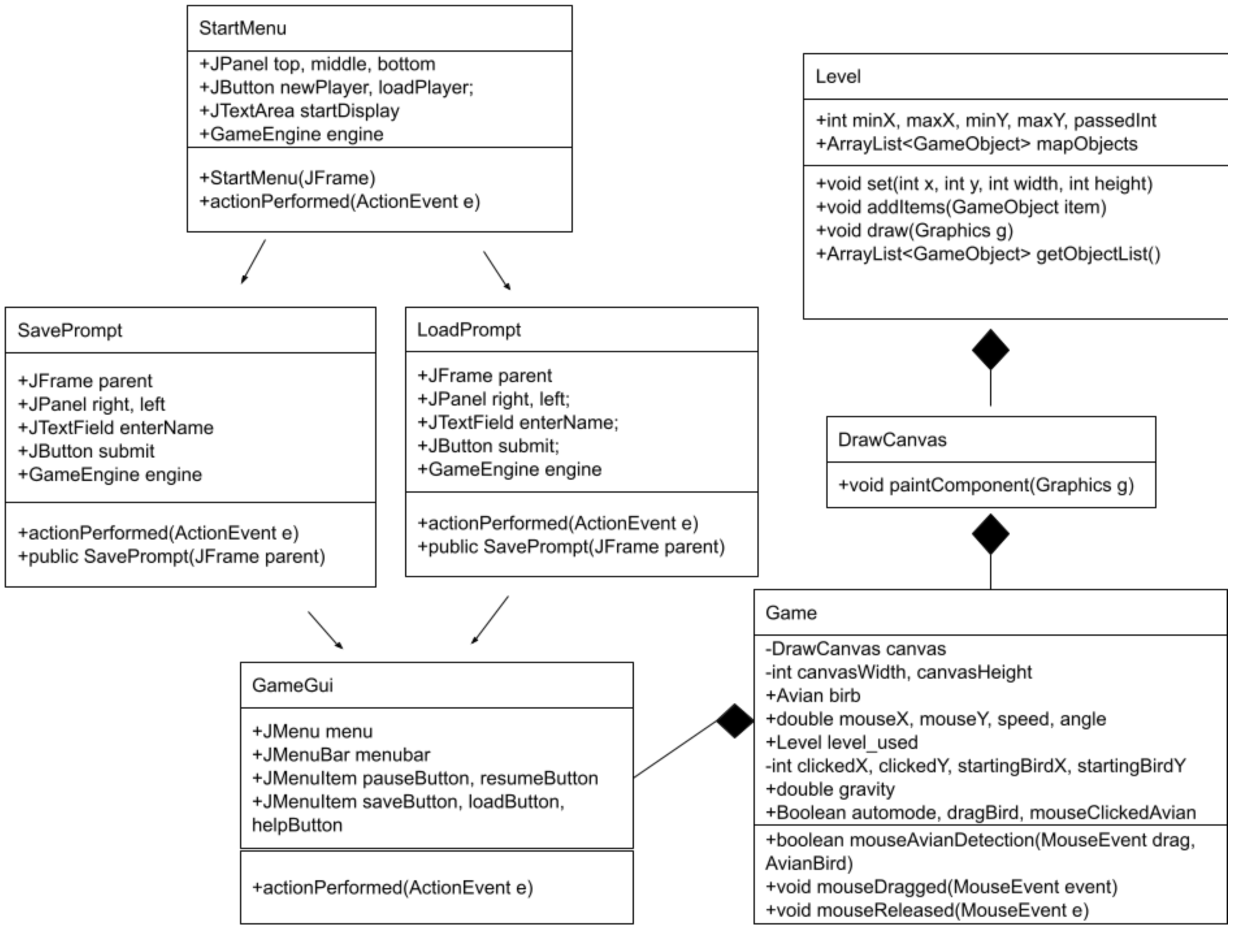
### MapLoader

MapLoader loads in a map file utilizing its file name as a string. This is used in Level in the gui.

### MapSaver

MapSaver is utilized to save a map to the application. SaveMap should take in an ID and ArrayList of objects

## GUI


StartMenu
+JPanel top, middle, bottom
+JButton newPlayer, loadPlayer;
+JTextArea startDisplay
+GameEngine engine
+StartMenu(JFrame)
+actionPerformed(ActionEvent e)
SavePrompt
+JFrame parent
+JPanel right, left
+JTextField enterName
+JButton submit
+GameEngine engine
+actionPerformed(ActionEvent e)
+public SavePrompt(JFrame parent)
LoadPrompt
+JFrame parent
+JPanel right, left;
+JTextField enterName;
+JButton submit;
+GameEngine engine
+actionPerformed(ActionEvent e)
+public SavePrompt(JFrame parent)
Level
+int minX, maxX, minY, maxY, passedInt
+ArrayList<GameObject> mapObjects
+void set(int x, int y, int width, int height)
+void addItems(GameObject item)
+void draw(Graphics g)
+ArrayList<GameObject> getObjectList()
DrawCanvas
+void paintComponent(Graphics g)
GameGui
+JMenu menu
+JMenuBar menubar
+JMenuItem pauseButton, resumeButton
+JMenuItem saveButton, loadButton, helpButton
+actionPerformed(ActionEvent e)
Game
-DrawCanvas canvas
-int canvasWidth, canvasHeight
+Avian birb
+double mouseX, mouseY, speed, angle
+Level level_used
-int clickedX, clickedY, startingBirdX, startingBirdY
+double gravity
+Boolean automode, dragBird, mouseClickedAvian
+boolean mouseAvianDetection(MouseEvent drag, AvianBird)
+void mouseDragged(MouseEvent event)
+void mouseReleased(MouseEvent e)

### StartMenuGUI

StartMenu will contain a GUI that displays after the application is loaded. The GUI will contain two buttons. This menu will prompt the user to select one of the two, depending if the user is a returning or new player. newPlayer will prompt the user to write their name to a save file and loadPlayer will resume the user’s most recently saved game state. ActionPerformed will determine what action was utilized and the resulting logic for it.

The StartMenu is apart of a long line in a Chain of Responsibility state pattern that is composed of all other GUIs in the package

### SavePrompt

SavePrompt will contain a button and a textfield for the player to write their name into. Once their name is entered, they will press submit to create a new save file. This should not take in a blank name for a valid input.

### LoadPrompt

LoadPrompt will contain a button and a textfield for the player to write their name into. Once their name is entered, they will press submit to load a new save file. This should not progress any further if the player’s save file is invalid.

### GameGUI

GameGUI will display the game and level after submit is pressed on SavePrompt and LoadPrompt. Game will contain the logic for the Avian launch. Level will be loaded in from the mapLoader.

# Data Format

Formatting for data that DAP will use is still unknown since this project is in its very early stages of development. The data that DAP would load in is whether or not a map or level is cleared, and a load/save file name that has been specified by the player. For the moment, current data read in will be as a text file.

```
File Format Ex.

[Players] |George Tito Cesar|

[Progress]

George Cleared Cleared Unattempted ;

Tito Cleared Cleared Failed ;

Cesar Failed Unattempted Unattempted ;

[EndProgress]
```

```
File Format Ex.

[Map Name]:
'Level_1'
[Objects]:order for object attributes: posX posY width height mass
velocityX velocityY HP spritePath ColorR ColorG ColorB
Avian 250 350 50 50 100.0 0.0 0.0 200 none 169 2 45 ;
Block 1200 500 50 50 15.0 0.0 0.0 100 none 165 168 172 ;
Block 1200 448 50 50 15.0 0.0 0.0 100 none 165 168 172 ;
Block 1200 396 50 50 15.0 0.0 0.0 100 none 165 168 172 ;
Block 1252 448 50 50 15.0 0.0 0.0 100 none 165 168 172 ;
Block 1252 500 50 50 15.0 0.0 0.0 100 none 165 168 172 ;
Block 1252 396 50 50 15.0 0.0 0.0 100 none 165 168 172 ;
Target 1225 344 50 50 15.0 0.0 0.0 100 none 180 203 54 ;
```