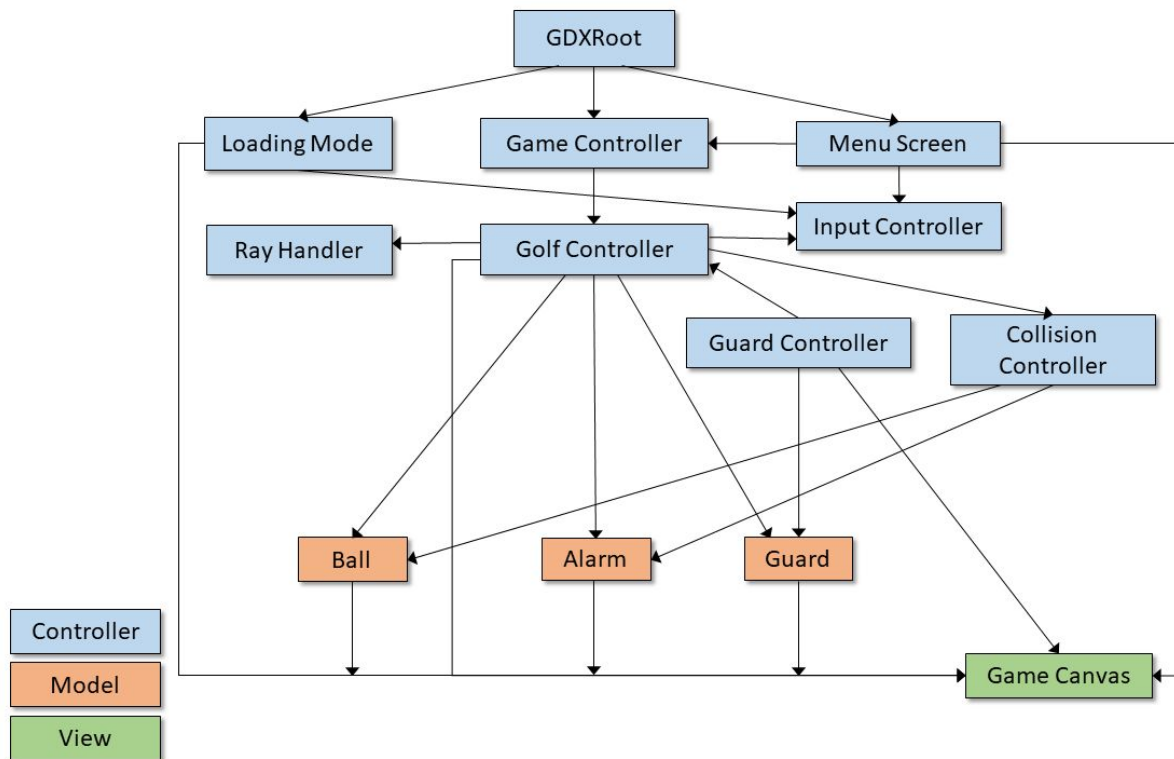


Architecture Specification

Parole in One

Waypoint. Isabel Selin, Yuxiang Yu, Courtney Manbeck, Lucien Eckert, Betsy Vasquez
Valerio, Kevin Klaben, Tony Qin, Barry Wang

Dependency Diagram



Class-Responsibility Collaboration Divisions

Controller

GDXRoot

Description: This class is the root class of the game. It will initialize the game controller, loading screen, and menu screen. It will handle the transition between these 3 screens.

Justification: This class is the root of the game.

Responsibilities	Collaborators
Initialize controllers	LoadingMode, MenuScreen, GameController
Listen for screen transition cues	LoadingMode, MenuScreen, GameController
Exit game	-

MenuScreen



Description: This class presents the main menu screen for choosing levels and settings

Justification: This controller is needed to handle most interactions when not actively playing a level.

Responsibilities	Collaborators
Load menu screen assets	AssetManager
Draw menu screen	GameCanvas
Get information about button presses	InputController
Push level number currently selected	GameController
Push exit code	ScreenListener

LoadingMode

Description: This class handles display and interaction during and immediately after asset loading.

Justification: This class makes time for and ensures assets to pre-load.

Responsibilities	Collaborators
Load loading-screen assets	AssetManager
Get progress of loading remaining game assets	AssetManager
Display progress of loading remaining game assets	GameCanvas
Get information about button presses	InputController
Push exit code	ScreenListener

GameController

Description: The game controller will create new instances of golf controllers for each level as the levels are requested. It'll also keep track of states related to gameplay such as levels completed and scores of each level. The game controller is also used to parse JSON files.



Justification: This controller allows us to detect whether the level data inputted is corrupt. This controller is also required to keep track of game information that persists across levels.

Responsibilities	Collaborators
Parse level data from JSON file	-
Instantiate levels using level data	GolfController
Listen for pause and exit level requests	GolfController
Push exit code to ScreenListener	ScreenListener

GolfController

Description: Each golf controller controls one level in the game. This controller contains information about its level, such as the position of game elements.

Justification: This controller is needed to run the main game loop, and allows for game elements to interact with each other.

Responsibilities	Collaborators
Load game assets	AssetManager
Assign textures to models	All model classes
Create game objects and add them to the world	 
Create and attach light elements to game assets	RayHandler
Get information about shot power and angle selected by players, and set ball movement based on that	InputController, BallModel
Push information about contacts between elements	CollisionController
Draw backgrounds, walls, and HUDs	GameCanvas
Draw Box2d lights effects	RayHandler

GuardController

Description: This is the AI controller that will change guard states and give guards instructions to move. It also handles ball detection from each guard.

Justification: A controller is necessary to handle state changes and pathfinding of guards.

Responsibilities	Collaborators
Get information about game world to decide on guard state	GolfController
Get information about game object locations to plan guard movement	GolfController
Set guard movement	GuardModel
Draw guard texture based on the state of the guard, using textures stored in the GuardModel. Also provides GameCanvas with location to draw.	GameCanvas

CollisionController

Description: This is the controller that will determine the correct behavior of each collision that occurs.

Justification: A collision controller is needed since we have slightly complex interactions (such as alarm activation and deactivation)

Responsibilities	Collaborators
Set victory or loss field in BallModel if it touches the goal or a guard respectively	BallModel
Set alarm to activated or deactivated if it touches the ball or a guard respectively	AlarmModel
Change the ball velocity if it touches a wall	BallModel

InputController

Description: This is the controller that reads player input and converts these inputs into usable states for other classes.

Justification: This class abstracts out information about user input to one location and stores all relevant user input for each controller to get information from. This class also converts inputs from different devices into a standardized format.




Responsibilities	Collaborators
Read user input from mouse/trackpad	-

Model

AlarmModel

Description: This model handles the drawing of alarms and keeps track of the state of an alarm, mainly whether or not it is activated.

Justification: This is a model for the alarm environmental object.

Responsibilities	Collaborators
Draw alarm texture based on whether the alarm is activated or deactivated. Also provides GameCanvas with location to draw.	 GameCanvas

BallModel

Description: This model handles the drawing of the ball and keeps track of the ball's physical properties, including its body, shape, and fixture.

Justification: This is a model for the player-controlled ball object.

Responsibilities	Collaborators
Get information about force acting on ball to move ball	-
Draw ball texture based on whether the ball is moving. Also provides GameCanvas with location to draw.	GameCanvas

GuardModel

Description: This model handles the drawing of guards and keeps track of their properties. This includes their patrol paths, starting locations, and sight radius and distance.

Justification: This is a model for the guard enemy.

Responsibilities	Collaborators
Get information about guard movement to move guard	-




View

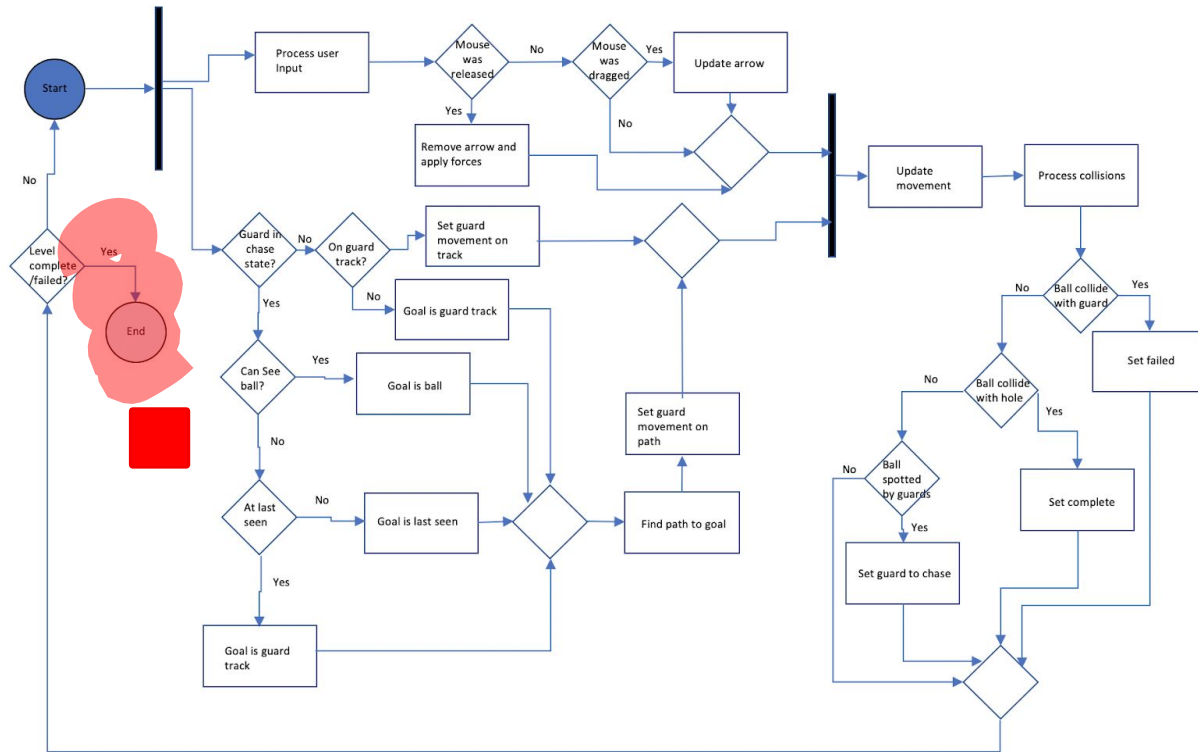
GameCanvas

Description: The game canvas draws objects onto the screen using their positions as well as any other transformations that may be applied to each object.

Justification: Game canvas must exist to facilitate the drawing of objects to the screen.

Responsibilities	Collaborators
Get location and text  information and draw them on the screen	-

Activity Diagram



Data Representation Model



Saved Game File

Overview

The game will store saved data in a JSON file. This JSON will contain the state of the game (LoadingMode, MenuScreen, GameController) and which levels the player has completed. The keys of the JSON will be “gamestate” and “completed_levels”.

Example

```
{ "gamestate": "MenuScreen", "completed_levels": [1, 3, 5] }
```


Level File

Overview

Each level will be stored in JSON file format. The levels will each contain the following information: level number, the ball starting and goal positions, placement of walls, position and patrol paths of guards, and position of alarms, boxes, buttons, and doors.

level: This section of the JSON will be an integer denoting the number of the level the JSON is representing.

ball_start: This section of the JSON will be a list of length 2 of screen coordinate locations of where the ball should be when the level is initialized. The first element in the list is the x coordinate and the second element in the list is the y coordinate.

ball_goal: This section of the JSON will be a list of length 2 of screen coordinate locations of where the ball should be to reach the goal state. The first element in the list is the x coordinate and the second element in the list is the y coordinate.

walls: This section of the JSON will be a list of arrays of screen coordinate locations of where walls should be placed. In each wall array the even indexes are the x coordinates and the odd indexes are y coordinates. If the coordinate in index i is an x coordinate it goes with the y coordinate in index $i + 1$. If the coordinate in index i is a y coordinate it goes with the x coordinate in index $i - 1$.

guard_positions: This section of the JSON will be a list of lists of length 2 of screen coordinate locations of where guards should be placed. The first element in the list is the x coordinate and the second element in the list is the y coordinate.

guard_patrols: This section of the JSON will be a list of arrays of screen coordinate locations. The length of this list will be the same as the length of the list in the guard_positions section. `guard_patrols[i]` corresponds to the patrol path of the guard in `guard_positions[i]`. In each guard_patrol array the even indexes are the x coordinates and the odd indexes are y coordinates. If the coordinate in index i is an x coordinate it goes with the y coordinate in index $i + 1$. If the coordinate in index i is a y coordinate it goes with the x coordinate in index $i - 1$.

alarms: This section of the JSON will be a list of lists of length 2 of screen coordinate locations of where alarms should be placed. The first element in the list is the x coordinate and the second element in the list is the y coordinate.

boxes: This section of the JSON will be a list of lists of length 2 of screen coordinate locations of where boxes should be placed. The first element in the list is the x coordinate and the second element in the list is the y coordinate.

buttons: This section of the JSON will be a list of lists of length 2 of screen coordinate locations of where buttons should be placed. The first element in the list is the x coordinate and the second element in the list is the y coordinate.

doors: This section of the JSON will be a list of arrays of screen coordinate locations. The length of this list will be the same as the length of the list in the buttons section. `doors[i]` corresponds to the doors associated with the button in `buttons[i]`. In each doors array the even indexes are the x coordinates and the odd indexes are y coordinates. If the coordinate in index `i` is an x coordinate it goes with the y coordinate in index `i + 1`. If the coordinate in index `i` is a y coordinate it goes with the x coordinate in index `i - 1`.

Example

```
{ "level": 1,
  "ball_start": [13.15625, 2.5],
  "ball_goal": [12.5, 16.0],
  "walls": [
    [0.0, 0.0, 32.0, 0.0, 32.0, 18.0, 0.0, 18.0, 0.0, 17.0, 31.0,
     17.0, 31.0, 1.0, 1.0, 1.0, 1.0, 17.0, 0.0, 17.0],
    [0.0, 0.0, 32.0, 0.0, 32.0, 18.0, 0.0, 18.0, 0.0, 17.0, 31.0,
     17.0, 31.0, 1.0, 1.0, 1.0, 1.0, 17.0, 0.0, 17.0],
    [0.0, 0.0, 32.0, 0.0, 32.0, 18.0, 0.0, 18.0, 0.0, 17.0, 31.0,
     17.0, 31.0, 1.0, 1.0, 1.0, 1.0, 17.0, 0.0, 17.0]
  ],
```

```
"guard_positions": [  
  [12.0,15.0]  
],  
"guard_patrols": [  
  [12, 14.5]  
],  
"alarms": [  
  [8.78125,5.5625]  
],  
"boxes": [  
  [8.78125,5.5625]  
],  
"buttons": [  
  [10.1875,10.09375],  
  [7.6875,13.65625]  
],  
"doors": [  
  [10.1875, 10.09375, 7.4375, 13.8125, 3.3125, 13.40625, 3.28125,  
  8.9375, 7.34375, 7.3125],  
  [7.6875, 13.65625, 3.375, 13.40625, 3.1875, 8.53125, 7.71875,  
  7.21875, 9.96875, 9.78125]  
]  
}
```

3152: Assignment 9 (Architecture Document)

Assessor: Walker White

SA: Strongly Agree

A: Agree

D: Disagree

SD: Strongly Disagree

	SD	D	A	SA
The dependency diagram is an easy-to-follow summary of the architecture.				
• The dependency directions are clear from the diagram.				✓
• The information flow can be discerned from the diagram.				✓
• The diagram does not have too many overlapping edges.				✓
• The diagram does not have any cycles or improperly coupled classes.				✓
The CRC tables adequately describe the class structure .				
• The class choices are sensible and properly organized.				✓
• The architecture can support serialized levels.			✓	
• The collaborators appear correct for each responsibility.			✓	
• There are no obvious missing responsibilities.	✓			
• There are no improper responsibilities (e.g. controller code in models).			✓	
• The responsibility descriptions are active and appropriate.			✓	
The activity diagram clearly describes the flow of a single frame .				
• The activities are described at the appropriate resolution.			✓	
• The activities are consistent with the CRC tables.				✓
• The diagram uses parallelism when order is not necessary.				✓
• The diagram follows proper UML conventions.				✓
The serialization format supports external tool support .				
• The file format appears to store the necessary information.				✓
• The high-level attributes are all clearly identified and described.				✓
• The format has an illustrative, but brief, example.				✓
The document is clear and well-written .				
• The document contains no spelling errors.				✓
• The document contains no grammatical errors.				✓
• The sections are properly organized.				✓
• The document adheres to the course writing guidelines.			✓	