I. Status Summary

Names: Erica Lowrey , Thomas Deaner, Emma Goodwill
Title: Dino Doomsday: a 2D platformer game
Work Done:
Currently, we have created the “pieces” of this game. They will be customized with art and level design will be completed using these basic pieces soon. In short, the work that we have already done constitutes every “puzzle piece” needed to assemble the game into a fun platformer. Most of the work left to do is the assembly of these pieces.

- Thomas:
  I created the entire input system (Class Diagram in Section II) using OOAD principles. The player's character can be in three different states: Grounded, Aerial, and Crouched; and in each state the player can perform a unique set of actions, such as Walk, Jump, and so on. Some of these actions will be "unlocked" by the player through gameplay, and I have created the input system so that these "unlockable" actions are created dynamically in real time. Before an action is "unlocked" the player will not be able to perform that action, and pressing its designated key on the keyboard will not trigger anything. Now that I have set up the input system, I can now start working on specific player actions, states, and movement to make the game "feel good" to the player.
- Erica:
  I've been working on assets for the game. I've finished the title screen, an idle animation, and several sketches for other animations in the game. I've started concepts for colors and styles for the levels, so when I make the tiles I have a reference to work with to keep everything consistent. The title screen has a 16:9 ratio and any other background assets will hold that so things scale together, and I've made everything into separate layers to help with any future modifications we might have to make.
- Emma:
  I have created a couple of basic obstacles including the meteorite, other dinosaurs, and uneven ground. The meteorite and other dinosaur obstacles are game objects with a 2D collision circle/box that uses asynchronous programming to determine when they intersect with the player object (which is a rigid body also with a 2D collision box). At the point of intersection, the meteorite object is deleted and the other dinosaur’s health is reduced. I have also made the asynchronous function in the player class such that when the meteorite/other dinosaur and the player collide, the player’s health is reduced by one. I have created the uneven ground obstacle experience that shakes the ground upon which the player stands. The player is a rigidbody, so this shaking can cause the player to slide given the effects of gravity. I have also created a main menu system with two buttons as of now: a play game button and a create character button. The play game button causes the

scenes to change from the main menu to the sample level scene. The customize character button causes the scenes to change from the main menu to the customization scene where the user can select their dinosaur's primary color, secondary color, and a hat. I have also worked on the mechanism to save this game state to a json file. The player input for creating character customization is saved to this file.

Changes or Issues Encountered:

- It no longer makes much sense to store the character's high score, given we're operating on a level system rather than a time-based system; therefore, the only persistent data will be character customization data.
- In Project 5, we believed it made sense to create a State for each specific action the player could make such as GroundedWalkingState, AerialWalkingState, etc., but during Project 6 we decided that we should separate States and actions as PlayerActions, so that we would have States such as Grounded and Aerial and PlayerActions such as Walk or Jump. Instead of the player's input determining what State the player's character should be in, we decided that the player's input should determine what PlayerAction should be taken by the player's character, depending its current State. Ultimately, by separating States and PlayerActions, we should have a codebase that is open to extension, not modification.

Patterns:

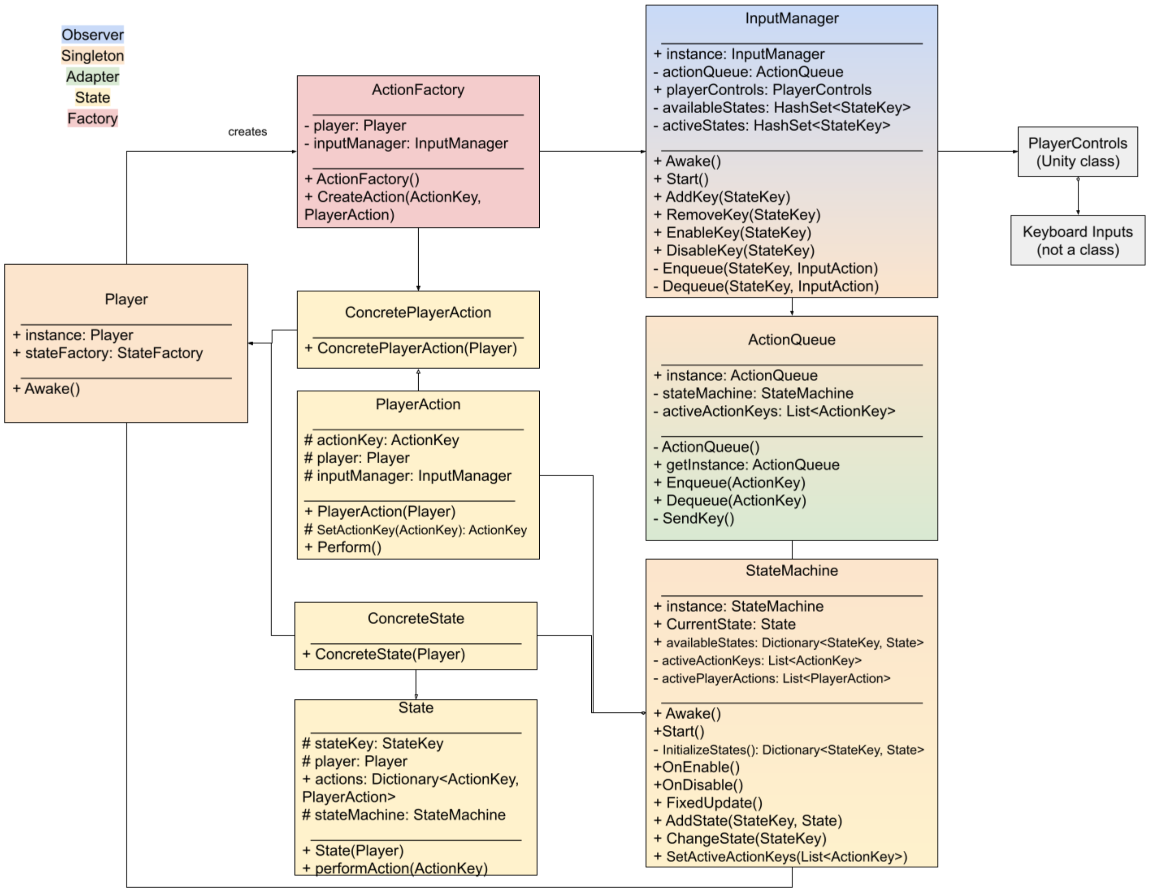
Input System: Singleton, Observer, State, Factory, Adapter

- Singleton:
  - We use this pattern for classes that we only want a single instance of.
  - For instance, we only want one InputManager class to observe player inputs on the keyboard as we don't want duplicate responses to keyboard inputs.
- Observer:
  - The InputManager class works as a subscriber to various keyboard inputs. When a player presses a key on the keyboard the InputManager is notified, and the class sends an ActionKey to the ActionQueue and then the StateManager to have a PlayerAction performed.
- State:
  - We have a StateMachine that changes the player's current State depending on whether or not the player is grounded.
  - The PlayerActions the player can invoke changes depending on the player's current State.
- Factory:

    - We use the factory class to create new PlayerActions and State classes (not yet implemented) in real time as the player "unlocks" new PlayerActions and States.
- Adapter:
    - When the player presses a key, the PlayerControls class sends data to the InputManager, which is then sent to the ActionQueue, which is finally sent to the StateMachine class. The ActionQueue class works as an Adapter: the data we get from the PlayerControls class, a class created by the Unity engine, sends redundant data that the StateMachine doesn't need, so we have implemented the ActionQueue to work as a filter for the data sent from the InputManager to the StateMachine.

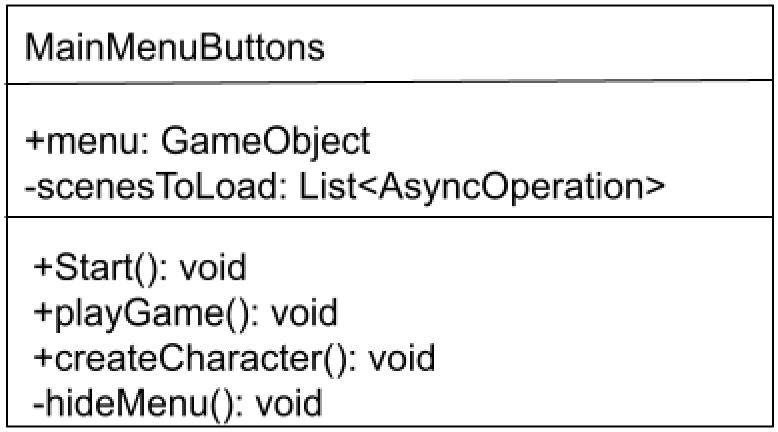
II. Class Diagrams

Input System:

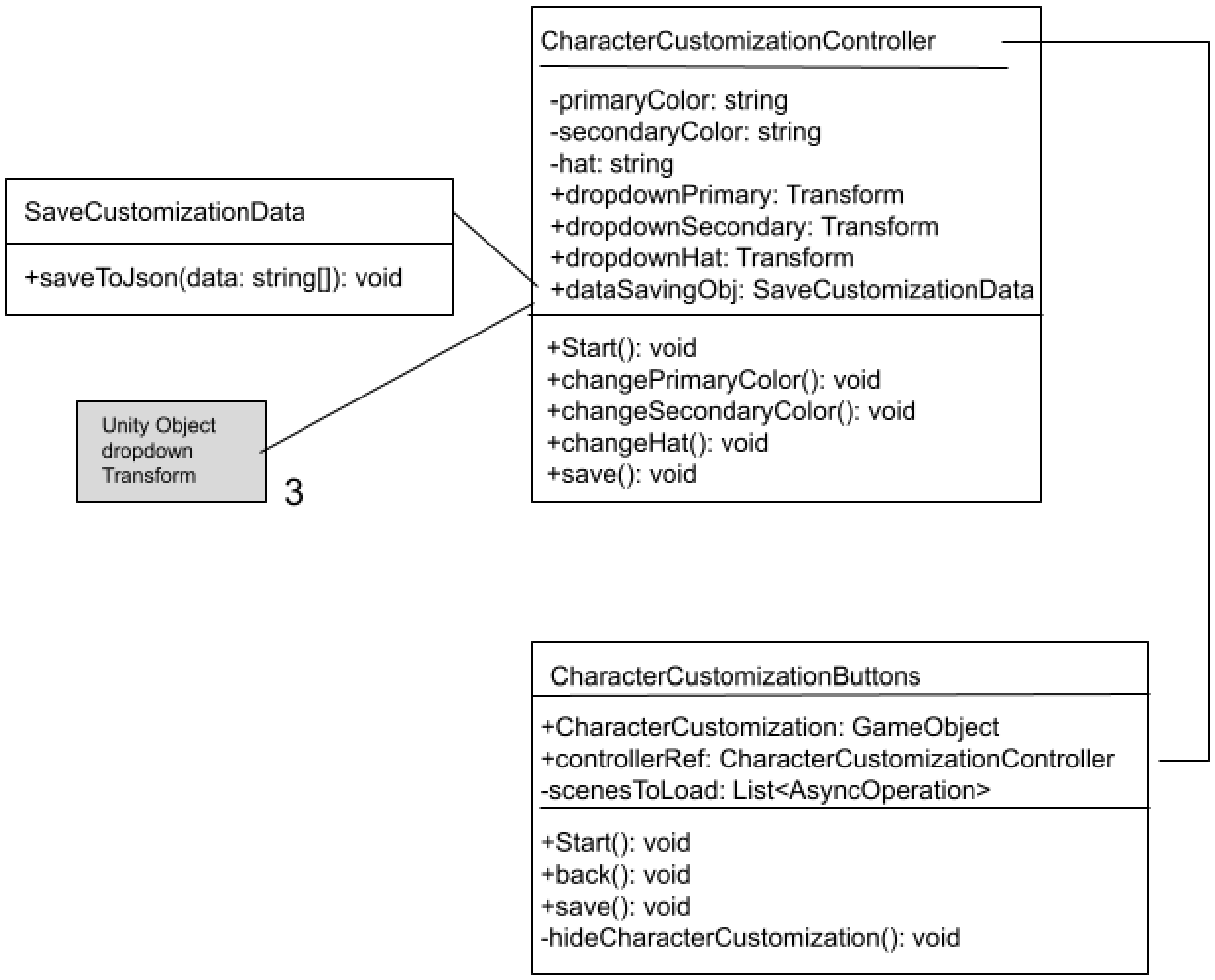


Observer
Singleton
Adapter
State
Factory
creates
ActionFactory
- player: Player
- inputManager: InputManager
+ ActionFactory()
+ CreateAction(ActionKey, PlayerAction)
InputManager
+ instance: InputManager
- actionQueue: ActionQueue
+ playerControls: PlayerControls
- availableStates: HashSet<StateKey>
- activeStates: HashSet<StateKey>
+ Awake()
+ Start()
+ AddKey(StateKey)
+ RemoveKey(StateKey)
+ EnableKey(StateKey)
+ DisableKey(StateKey)
- Enqueue(StateKey, InputAction)
- Dequeue(StateKey, InputAction)
PlayerControls (Unity class)
Keyboard Inputs (not a class)
Player
+ instance: Player
+ stateFactory: StateFactory
+ Awake()
ConcretePlayerAction
+ ConcretePlayerAction(Player)
PlayerAction
# actionKey: ActionKey
# player: Player
# inputManager: InputManager
+ PlayerAction(Player)
# SetActionKey(ActionKey): ActionKey
+ Perform()
ActionQueue
+ instance: ActionQueue
- stateMachine: StateMachine
- activeActionKeys: List<ActionKey>
- ActionQueue()
+ getInstance: ActionQueue
+ Enqueue(ActionKey)
+ Dequeue(ActionKey)
- SendKey()
ConcreteState
+ ConcreteState(Player)
State
# stateKey: StateKey
# player: Player
+ actions: Dictionary<ActionKey, PlayerAction>
# stateMachine: StateMachine
+ State(Player)
+ performAction(ActionKey)
StateMachine
+ instance: StateMachine
+ CurrentState: State
+ availableStates: Dictionary<StateKey, State>
- activeActionKeys: List<ActionKey>
- activePlayerActions: List<PlayerAction>
+ Awake()
+Start()
- InitializeStates(): Dictionary<StateKey, State>
+OnEnable()
+OnDisable()
+ FixedUpdate()
+ AddState(StateKey, State)
+ ChangeState(StateKey)
+ SetActiveActionKeys(List<ActionKey>)

Main Menu:

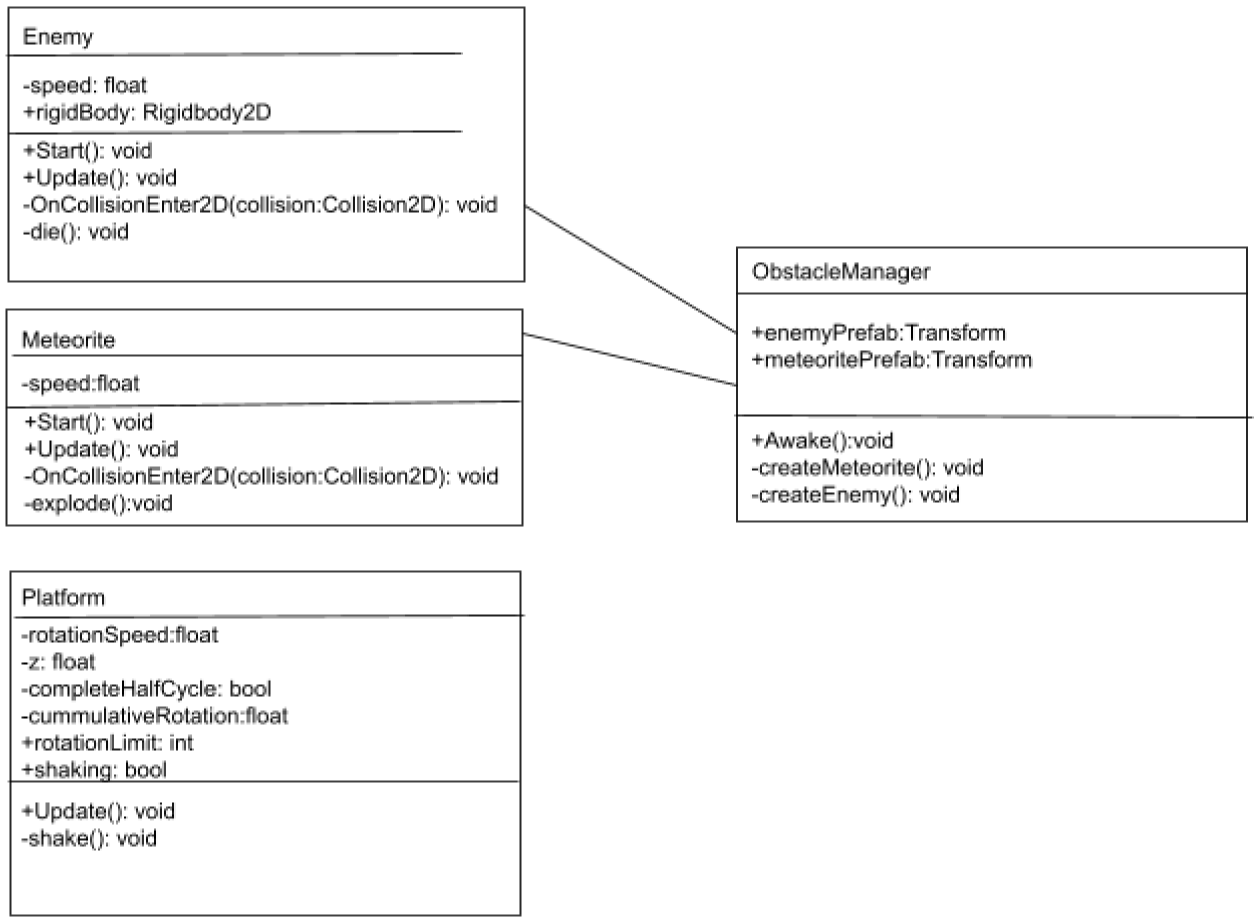


Character Customization:



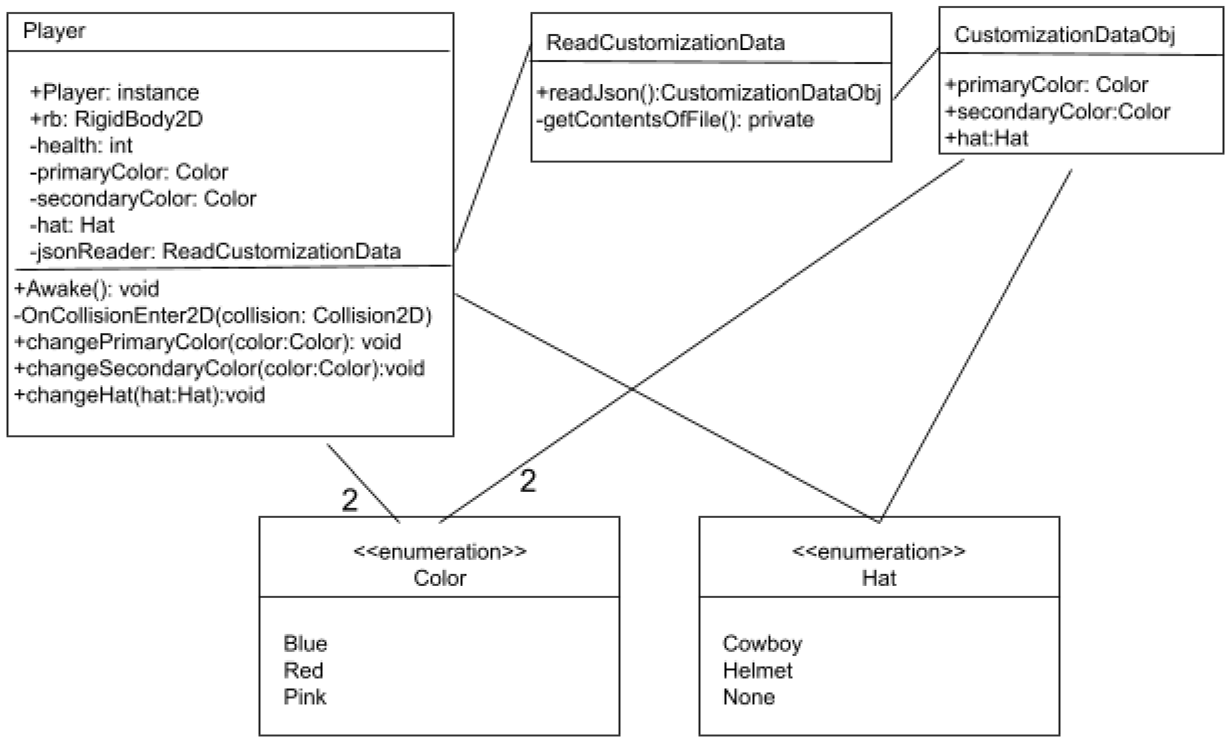
CharacterCustomizationController
-primaryColor: string
-secondaryColor: string
-hat: string
+dropdownPrimary: Transform
+dropdownSecondary: Transform
+dropdownHat: Transform
+dataSavingObj: SaveCustomizationData
+Start(): void
+changePrimaryColor(): void
+changeSecondaryColor(): void
+changeHat(): void
+save(): void
SaveCustomizationData
+saveToJson(data: string[]): void
Unity Object dropdown Transform
3
CharacterCustomizationButtons
+CharacterCustomization: GameObject
+controllerRef: CharacterCustomizationController
-scenesToLoad: List<AsyncOperation>
+Start(): void
+back(): void
+save(): void
-hideCharacterCustomization(): void

<u>Obstacle Objects:</u>



<u>Player Reading Customization Data Interaction:</u>

Referenced to determine how to mark enumerations in UML class diagrams:
https://www.softwareideas.net/class-diagram-enum



III. Plan for Next Iteration

The work that still needs to be done is predominantly level design and the incorporation of the art. Most of the UI for this work has been completed; however, the object's appearances need to change. We also want to show the level the user is currently on and their current health. The health information is already being tracked; however, we need to display it to the user. We will also make ten versions of the gameplay scenes that are increasing in difficulty. The obstacle basics for this gameplay is already complete, so all we will have to do is design each level and place the obstacles accordingly. We also would like to add cutscenes for story and a scene for when the user wins or dies. Finally, we will work on the movement of the player's character and make sure all of the movements "feel good" to the player.