Assessment 2: Brownfield Development

Architecture

Cohort 3 Group 9

Oliver Barden

Connor Burns

Sam Goff

Milap Keshwala

Lewis Mitchell

Jeevan Singh Kang

Harry Thomas

Titas Vaidila

Class diagram

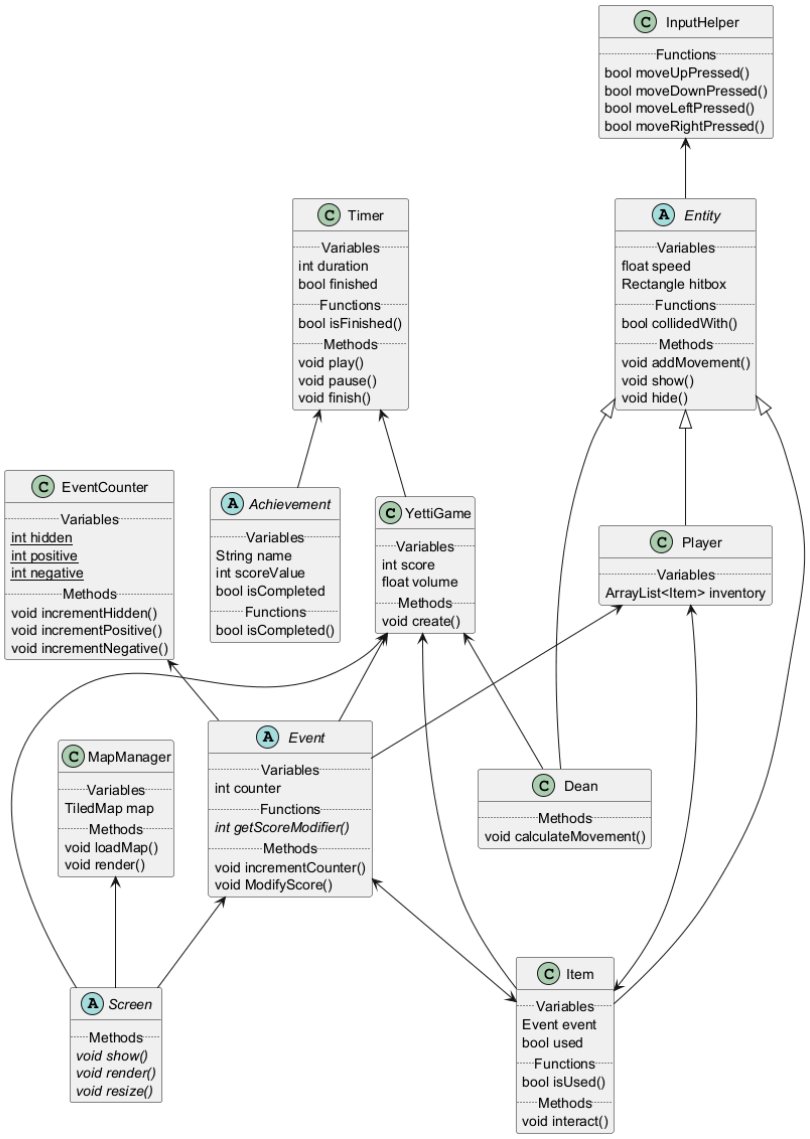


Fig 3a: the final class diagram

The above class diagram is a structural representation of the architecture of the game design. This was chosen as the game is coded using an object-oriented paradigm. A class diagram allows the relationships between the classes to be shown in a clear, brief overview of the implementation. This is a very abstract diagram, ignoring many details of implementation, allowing the main aspects of the architecture to be presented and understood with clarity. A complete class diagram of the game can be found on our website, giving a more in-depth understanding of the structure.
Note: class diagrams were created in UML with PlantUML.
Link to in-depth class diagrams: Arch - Class Diagrams

Fig 3a shows the main classes implemented. The **YettiGame** class is used as a main class to bring all the different components together. It includes variables for the entire game including the volume and score. This class extends the imported libGDX “Game” class, having direct access to the state of the game through the resume() and pause() methods.

The **Timer** class then directly calls these within its own pause() and play() methods to separate all time related dependencies into a single class, satisfying the FR_PAUSING and UR_PAUSE. Additional getter and setter methods to e.g. change the time or check whether the counter has reached zero have been created. Initially the “duration” is instantiated to 5 minutes within the **GameScreen** class and cannot be increased in any way. This satisfies UR_TIME and displaying the time remaining with the **GameScreen** class fulfills UR_UI.

The **MapManager** class contains all meta data about the game's map. Most of this data is using the “tiled” feature of the imported LibGDX library. This greatly simplifies the map creation process by enabling the map to be created using an external software, configuring all necessary components there, and finally exporting a “map.tmx” file. The loadmap() method uses the data inside to set the map appearance as well as collision data. A isRectInvalid() check uses this collision data and an input rect to determine if they overlap, this function is used by the **GameScreen** class to check whether entity movement is valid and adheres to FR_BOUNDARIES. Finally to help fulfill FR_MAP_CREATION, createMapRenderer() assigns the data to a “renderer” variable to be rendered to the screen.

Many classes extending from the abstract class **Event**, shown in Fig 3a, are instantiated indirectly within the **Item** class, and are placed throughout the playable map area. Events are necessary in the game in order to accomplish UR_EVENTS. The **Event** class contains the method modifyScore(), enabling the score to be altered, satisfying FR_SCORING and UR_SCORE. Each event that extends the **Event** class updates the abstract getScoreModifier() function to return a value, enabling events to positively, negatively, or neutrally affect the score. Additionally, within each **Event** class’s activate() method, negative and positive effects can be given such as changing the player’s speed, thus many event classes are linked to entity classes. By bestowing these effects, the FR_POSITIVE_EVENTS, FR_NEGATIVE_EVENTS, FR_HIDDEN_EVENTS requirements are fulfilled. The events created are appropriate for the game setting ensuring they are immersive and fit in fulfilling FR_EVENT_STYLE and UR_STYLE.

WinEvent - When the player has completed all actions to beat the game, the win event occurs. This calculates the score the player has accumulated through the time and any events that have happened throughout the game. Then depending on this, the **WinScreen** or **NameInputScreen** is then presented with the score displayed. This event fulfills FR_GAME_END, FR_SCORING, and UR_WIN_LOSE.

WallEvent - Preset with the tiled imported file, the player is prevented from colliding with the walls of the game. If this happens, the player movement will be prevented in the direction of the wall, and there is a chance that the player will “Stubb their toe” losing score in the process. This adds to UR_SCORE and completes FR_BOUNDARIES.

An **EventCounter** class maintains a count of each event type, being updated every time an **Event** occurs. There are 3 counters: positive, negative, and hidden, for each of the event types with each one having a setter, getter, and increment method in addition to a resetCounters() function to reset them all. This counter is displayed to the **GameScreen**, fulfilling UR_UI and FR_EVENT_TRACKING.

An abstract **Achievement** class contains a challenge the player can complete once a criteria is met, determined by each individual achievement that extends this class. The completion of each achievement is displayed within **AchievementsScreen**, informing the user, fulfilling FR_ACHIEVEMENT_DISPLAY, of the specific achievement acquired. Classes such as

Timer, **Event**, and **YettiGame** influence the outcome of many events, e.g. completing the game in a certain time, obtaining a score, etc. Within the main menu, the achievement completion list can be found, completing FR_MAIN_MENU. All logic and checking whether the user has completed said achievement is handled within the **AchievementManager** class, as well as saving the data to file so progression is saved between sessions. This being locally saved adheres to NFR_DATA.

The core visual display of the gameplay is handled within the **GameScreen** class. It renders all of the graphics e.g. entity sprites, background titled map, contributing towards UR_MAP, FR_MAP_DISPLAY and centralises a gameplay display, making debugging and version control much easier. As well, item instantiation is handled within this class, with some in a fixed position such as the “timetable” item and “homework/backpack” event triggers. These are set up by creating a new Item() to an **Entity** array, and spawning random items with the method spawnRandomItems(). The set position items and events help the gameplay flow smoother with them being appropriately placed fulfilling NTF_USABILITY, FR_START, and UR_MAP. Parallelically, randomly placed events make the game feel different every time the user loads in, emphasising a more natural gameplay loop, adequate for the NFR_IMMERSIVE and UR_STYLE requirements. Complimenting this, all sounds are handled for the game such as an item pickup or event completion satisfying FR_SOUNDS.

Additionally, the game contains other screen classes, each controlling individual components independent of one another. The relationship between screen classes can be seen with figure 1 within “Design Evidence (Class Diagrams)” on the website.

MenuScreen - The main menu that the user is initially faced with. The simple UI makes it clear what each button does: **Play** loads into the game, **Settings** opens the settings menu, **Leaderboard** displays the leaderboard, **Achievements** opens the achievement screen, **Credits** shows the credits, and **Quit** closes the game. The minimalist design fulfills NFR_USABILITY by ensuring the start of the experience is straightforward, additionally the screen itself satisfies FR_MAIN_MENU.

SettingsScreen - Contains 4 features: a volume slider to adjust the audio levels, changes the **YettiGame** volume value when changed; a button to reset the game’s leaderboard. This clears the leaderboard array within the **LeaderboardManager** class via the resetScores() method. All of this satisfies FR_SETTINGS_UI.

LeaderboardScreen - Displays the top 5 scores that the local game has seen. Once the player has successfully escaped the game, if the score is in the 5 best then they are presented with the **NameInputScreen**. Here the player’s score is presented as well as an option to input their name to be recorded. This fulfills UR_LEADERBOARD, enabling a user’s run to be added to a local leaderboard, ensuring data is not public, mandatory for NFR_DATA. If however the score is not high enough, a **WinScreen** is presented instead, simply displaying the player’s score and a button to the **MenuScreen**.

LoseScreen - In the case of a loss i.e. the player gets caught by the **Dean** or runs out of time, they are presented with the **LoseScreen**. Text is displayed showing that they have lost, as well as the score they would have achieved and a button to the **MenuScreen**. This satisfies UR_WIN_LOSE and FR_GAME_END, enabling the player to lose and clearly display this information.

CreditsScreen - All external resources / assets in which we have used to construct our project have been compiled within this display, with respective links to each of them accessible. By doing so, it allows users to see what components the game is made from and provides credit to the resources used, completing CR_INTELLECTUAL_PROPERTY and

CR_LICENSING.

Input helper - A class made to centrally control input commands that the player gives via keyboard. It contains anyOfTheseKeysPressed(), a function that returns true if any key in the input key array is pressed, and false if not. Additionally, there is a check for if any of the cardinal direction buttons are pressed, i.e WASD or Up, Down, Left, Right, each direction having a function, for simplifying the movement process. This satisfies UR_MOVEMENT and FR_CONTROLS.

All interactable objects within the game stem from the abstract **Entity** class. This extends the **Sprite** class from the **GDX** library, which contains a foundational link to be implemented. This **Entity** class contains information that would be expected from an interactable object in the game e.g. speed, hitbox, visibility, and movement. A lot of methods are present in this class such as: disable() to disable the entity; collidedWith() to check if it has collided with a specified entity; addMovement() to change their movement. Due to this being the core of all **Entities**, the class's attributes are very broad and varied, however we chose this to satisfy and cover all use cases.

Item is a class extended from the **Entity** class which covers inanimate objects subject to movement throughout the game's playthrough. Examples include **Bob**, **Long Boi**, and **TimeTable**. Some are linked to events such as the latter **TimeTable**, and many are linked to achievements such as collecting 3 **Long Boi** items. The class has an **Event** variable as well as a boolean "Used" to check if the player has interacted with it yet, conveyed through an interact() method.

Additionally, there are three other classes that extend **Entity**. **SocialSec** is an npc that wanders the bottom left of the map. If the player gets "caught" by them, the remaining time is reduced by a set amount. Their movement is random yet confined within the room / corridor adjacent. **Dean** is an entity that spawns at a set time towards the end of the game. It constantly moves towards the player, however is slightly slower enabling them to escape. If the **Dean** catches up, the **LoseScreen** is triggered. This is one of two ways the user can lose, contributing to UR_WIN_LOSE. Finally the **Player** is the only **Entity** that the user controls, and is the focus of the game. The character sprite is in clear view the entire game, and only the **Player** entity can interact with other entities and events.

Process of designing class architecture
Link to previous class diagrams: Yetti - Architecture

To initially come up with ideas on the architecture, we focused on what classes would be needed. Many of the classes remained in the final version of the class diagram, however we decided to remove some to make the overall structure simpler. For example, we had a Maze and Tile class which were removed, as we thought it may be best to store the maze array in the **Game** class. After having thought about how the array would work, we realised that integers could be used to represent the different tiles, and the graphics of the tiles could be represented by the **Sprite** class. Thus, the Tile class felt redundant.

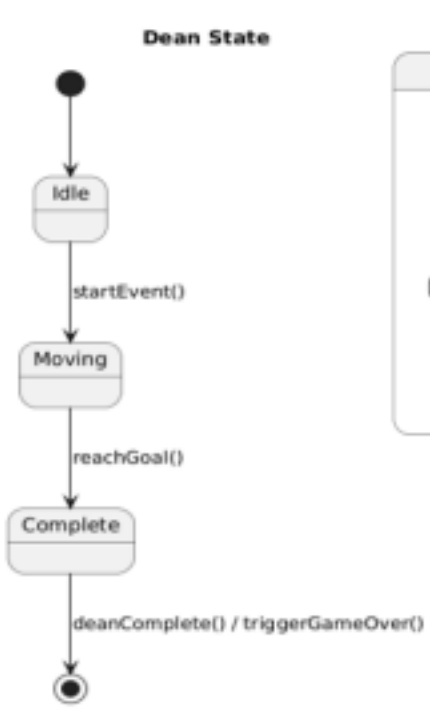
We considered the **Sprite** class and thought instead of having all the different assets (such as the Dean and Player) containing their own methods affecting the visual sprite, it would be best to put all that information in the **Sprite** class.
Another idea we had initially was having **Event** as an abstract class, and creating three classes for positive, negative and hidden events which inherited from **Event**. However, upon reflection, we realised that there wouldn't be much difference between the PositiveEvent,

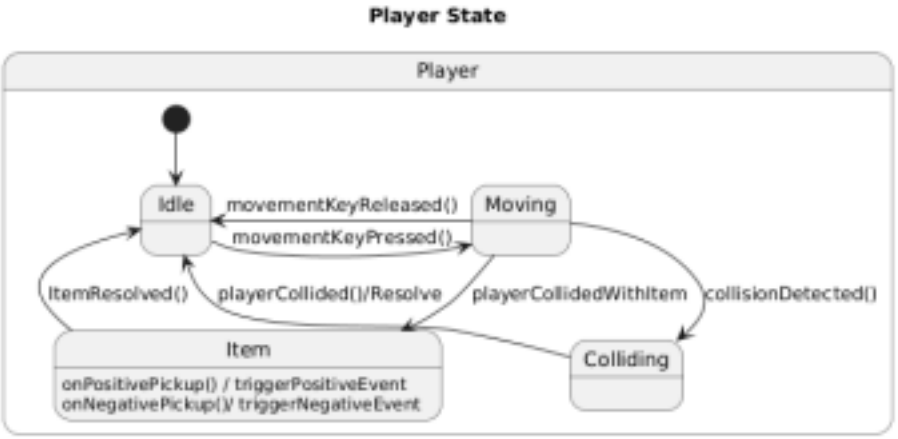
NegativeEvent and HiddenEvent classes, therefore we removed these classes, and made **Event** a regular class, and included a method to increase the score, and another to decrease the score.

State diagrams

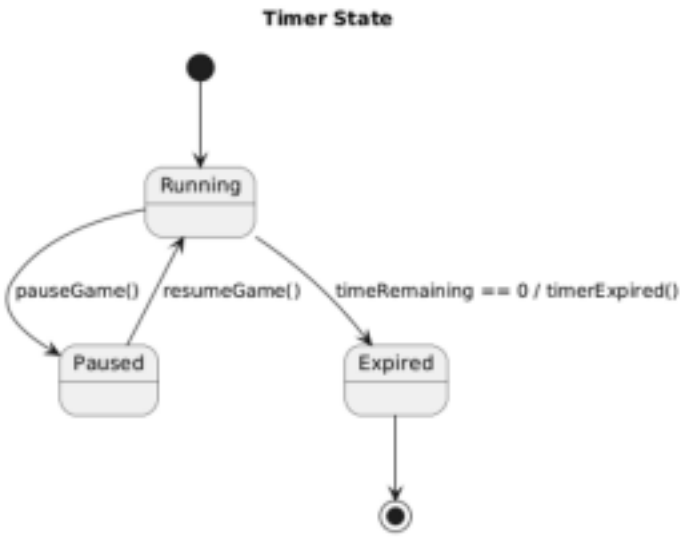
Given the event driven nature of our class architecture a state diagram of the game was created to reflect this. This was done as the overall game is determined by discrete events that trigger state transitions, as opposed to executing in a linear fashion. Meaning this system aligns more so with event driven systems as opposed to other types or architecture. These diagrams were created to show the behaviour of the game itself and to show how the different classes interact with one another such as the **Player**, **Dean** and **Score** substates. This representation makes it clear how the classes operate independently while still contributing to the overall system behaviour, the diagrams are not just a visual depiction of the game logic but also work as an architectural model that shows how the event driven system supports modularity and responsiveness.

Each substate within the overall **Playing** state acts as both an event producer and an event user. This can be seen in the **Player** substate with the transitions between **Idle, Moving** and **Colliding** are all events that are triggered by discrete events such as **movementKeyPressed()**, **movementKeyReleased()** and **collisionDetected()**. These events determine the players behaviour and influence other parts of the system. The player state also encompasses the **items** used within the game which are one of the ways to manipulate the **Score.**

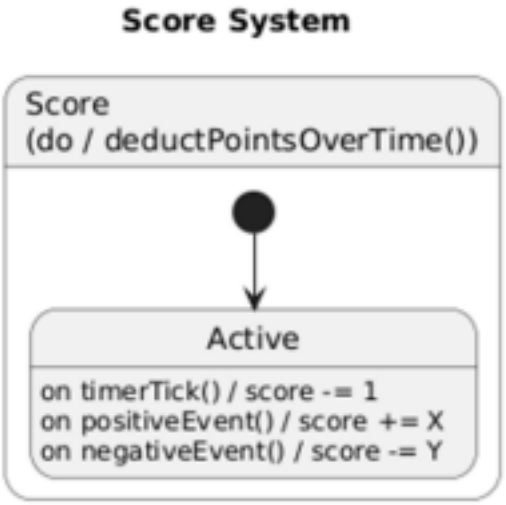




The **Enemy**/**Dean** shows the behaviour and the events caused by the main enemy of the game which begins to follow the **Player** after an event occurs. This substate responds to **startEvent()** and **reachGoal()** triggers moving through **idle**,**moving** and **Complete** states triggering a **gameOver()** when complete

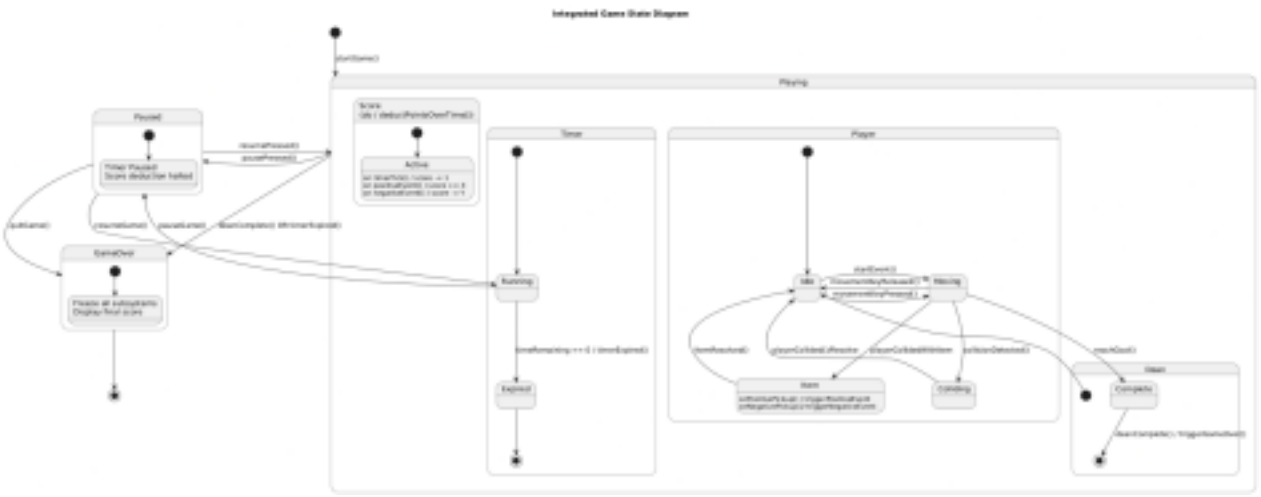


The timer substate reacts to **pauseGame**, **resumeGame** and the timer reaching 0 transitioning between **Running, Paused** and **Expired** when the **player** interacts with the pause menu buttons this substate satisfies the FR_PAUSEING and the UR_PAUSE requirements which is also shown in the larger state diagram as its own substate as well as satisfying NFR_GAME_TIME which ensures the game must last only five minutes with the expired state.

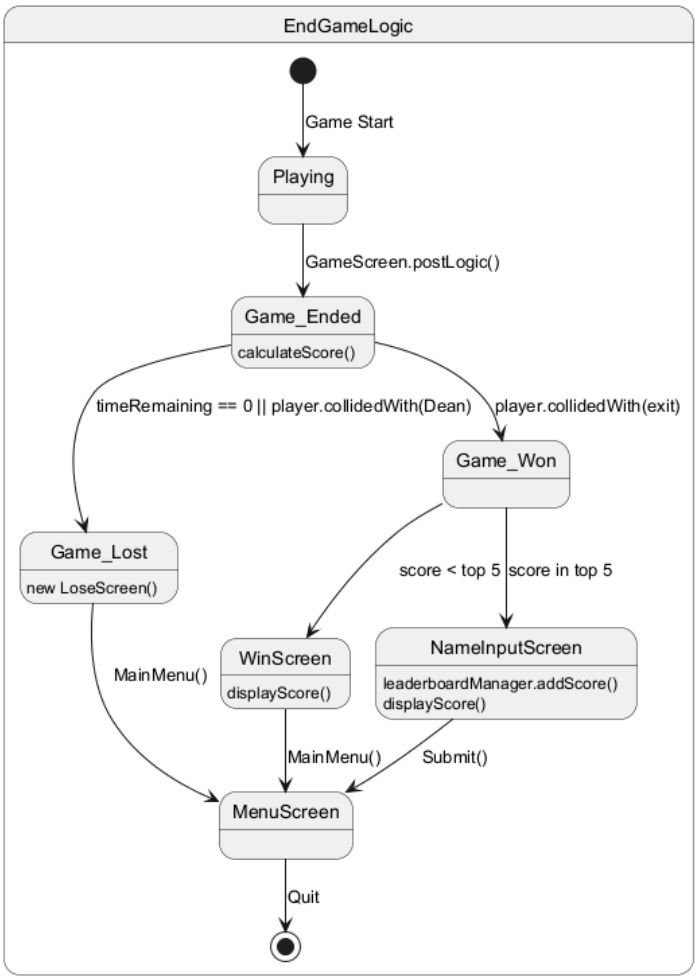


The **Score** substate updates its internal state in response to events such as **timerTick()** or the triggering of positive and negative events. This is one of the few states that is not self contained as it relies primarily on inputs from the **Timer** and the **Player**. The Timer affects score every tick making a constant change every second/tick whereas the inputs from Player require interactions with the environment this connected design highlights the nature of the event driven architecture allowing these subsystems to work together to create the score subsystem.

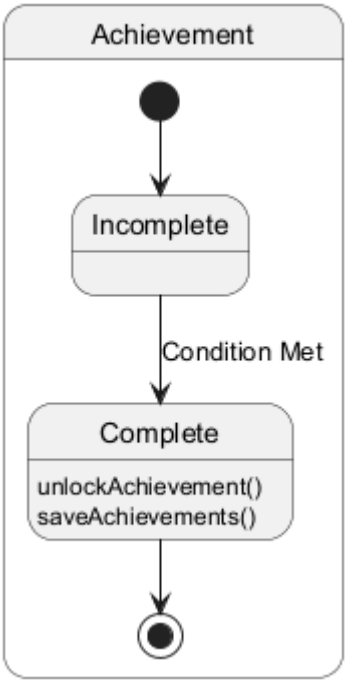
These subsystems all produce events that are used by the parent playing state in order to trigger higher level transitions such as **playing**, **paused**/**playing**, **gameOver**, which shows event propagation across multiple levels.



This state diagram shows the architecture supports a modular design making it easier to improve, change or rearrange certain aspects of the game as each substate handles its own logic separately while still being able to affect and be affected by other substates within the system. With the overall diagram you can see that the states meet the FR_WIN_SCREEN and FR_LOSS_SCREEN requirements which are triggered by the game over substrate outside the current playing states.

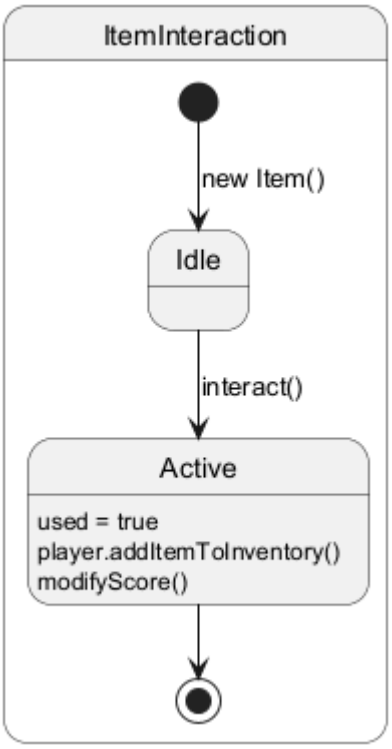


The **EndGameLogic** substrate depicts how the sequence ending the game is handled. It displays the conditions required for each of the three end screens: **LoseScreen**, **WinScreen**, and **NameInputScreen**, clearly displaying where they divulge from one another. This meets the requirements FR_GAME_END, FR_SCORING, UR_LEADERBOARD.



Achievement substrate displays how UR_ACHIEVEMENTS are met. Due to achievements varying significantly, the specific requirement for completing each has been abstracted to “condition met”. A full list of achievements and their conditions can be found below.

Name	Condition
No.1 Long Boi Fan	Collect all Long Boi in one game
Caffeine Addict	Drink all cups of coffee in one game
Bob says “I love you”	Encounter all Bobs and escape
Clumsy	Trip over all floorboards and escape
Very careful	Escape without tripping over any floorboards
Expelled	Get caught by the Dean
Close encounter	Encounter the Dean and escape
Escape Artist	Escape without stubbing your toe



An **Item** substrate reacts to the **Player entity** interacting with an instantiated item within the game. This is handled within the **GameScreen** class’s logic() method where collisions are checked. Due to items being the major source of event activations, multiple event classes are dependent on these interactions so ensuring the flow of item interaction is consistent is crucial. Below are all the events implemented within the game and their dependencies.

Event Name	Item Description	Cause	Effect
BackpackTrigger	Hidden	Moving through the corridor near the exit	Informs the player they have to pick their backpack
BackpackPickup	A backpack sprite	Collides with backpack	Allows the player to exit and plays a sound to signify pickup
Bob	A bob sprite	Collides with bob	Freezes the player and plays “sneeze” audio
Door	A door sprite	Collides with door after picking up check-in code	Opens the door with “opening door” sound
HiddenDeductPoints	A water puddle sprite	Collides with puddle	Shows the puddle, decreases score, plays “slip” sound
HiddenSpeedDebuff	A floorboard sprite	Collides with floorboard	Shows floorboard and slows the player, decreases score, plays “creak” sound
Homework	A book sprite	Collides with homework	If the player collects homework with more than 3 minutes remaining they gain score, play “paper” audio, else informs that “homework is late”
IncreasePoints	A long boi sprite	Collides with long boi	Plays “quack” sound, increases score
Key	A check-in code sprite	Collides with check-in code	Plays “paper” audio, gains check-in code
Speedboost	A coffee sprite	Collides with coffee	Plays “sips” audio, player gains speed, increases score
Wall	Hidden	Collides with a wall	Plays “stub” audio, decreases score
Win	A exit sprite	Collides with exit	Activates end screen, calculates score