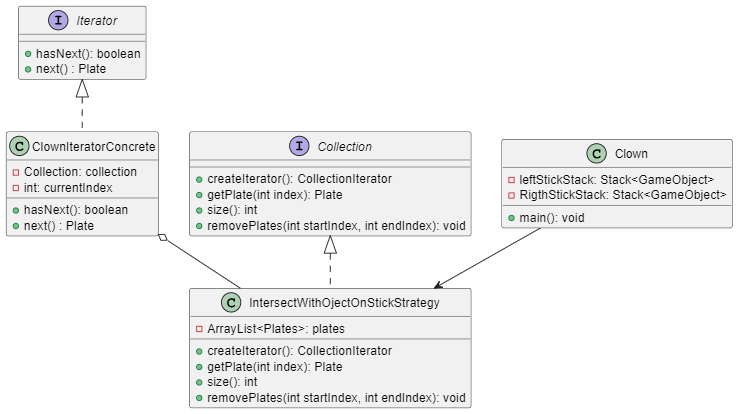
**Programming II Final Project Report**

1. **Shadi Reda Elattar – ID: 8180**
2. **Baher Ahmed Hosny – ID: 8165**
3. **Abdelrahman Mohammed Ramadan – ID: 8020**
4. **Feloubatteir Leasha Shafik – ID: 8231**

**Design Patterns Implemented:**

1. Iterator Design Pattern

The iterator pattern is employed in this code to provide a systematic way of accessing and traversing elements in a collection, specifically a Stack<GameObject>. This pattern is implemented through the **ClownIteratorConcrete class**, which conforms to the **Iterator interface (Built in)**, enabling sequential access to the stack's elements without exposing its underlying representation.

In the context of the game, the stack represents a collection of game objects, likely stacked on the clown's sticks. The ClownIteratorConcrete iterator facilitates traversing these objects in a specific order. This order is determined by the iterator's internal counter y and index i, which are manipulated during iteration.

The **hasNext()** method in the iterator checks if there are more elements to traverse by comparing the current position y with the stack's size. The **next()** method retrieves the next element in the stack, incrementing y and decrementing i. This approach allows for reverse iteration over the stack, starting from the top element and moving downwards, which is particularly useful in the game's context where the most recently added objects (on the top of the stack) are likely to be interacted with or checked first.

In the **checkObjOnStick()** method, the iterator is used to examine the top three objects on either of the clown's sticks. If these objects share certain attributes (in this case, color), a specific action is taken - the score is increased, and the objects are removed from the stack and the game. This use of the iterator pattern simplifies the process of examining and manipulating the collection of game objects, allowing for clean and understandable code while adhering to good object-oriented design principles by encapsulating the iteration logic within the **ClownIteratorConcrete** class.

1. Singleton

A screenshot of a computer

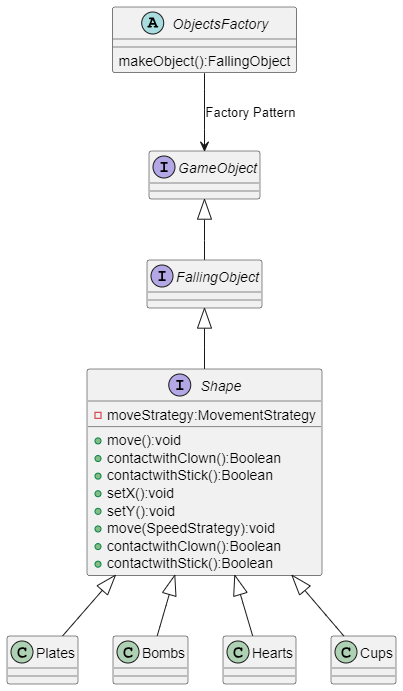
Description automatically generatedThe Clown class employs a private constructor, **Clown()**,which initializes the Clown object and sets up essential properties. This constructor is deliberately made private to prevent external classes from directly creating instances of Clown.

Within the class, there's a static variable clown of type Clown, marked as private static. This variable serves as the single instance holder for the Clown class and is initially set to null.

The crucial method enabling the Singleton behavior is **getInstance()**. It's a public static method synchronized on the Clown class. This method is responsible for controlling the instantiation of the Clown object. It checks if the clown instance is null. If it is, it creates a new instance of Clown with the provided parameters. If not, it returns the existing clown instance, ensuring only one Clown object exists in the application.

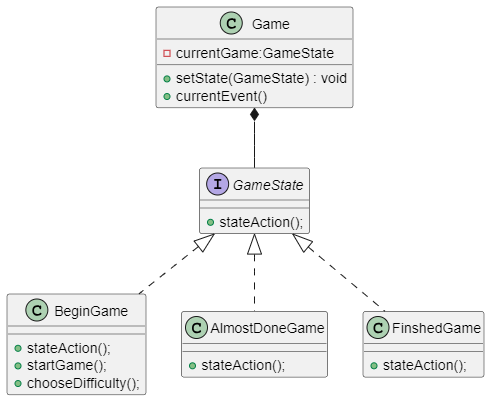
The usage of this Singleton pattern ensures that whenever a Clown instance is required, developers access it via the **getInstance** method rather than directly creating a new instance using the constructor. This prevents multiple Clown instances from being generated and ensures uniform access to the same Clown instance across the application.

1. Factory

The **ObjectFactory** class embodies the **Simple Factory pattern** by centralizing the creation of FallingObject instances. Its **createObject** method takes parameters defining the object to be created (**objectName**). Depending on the **objectName** provided, the factory method instantiates and returns a corresponding object, such as a **Plate**, **Cup**, **Bomb**, or **Heart**. This abstraction shields the calling code from the complexities of object creation and allows for a straightforward interface to generate different FallingObject types.

By employing a switch-case structure within the **createObject** method, the factory handles object creation logic based on the input string. If an invalid or unrecognized **objectName** is supplied, it defaults to printing an error message and returning null. This approach offers centralized object creation and easy extensibility by adding new object types.

1. State Pattern

The **State** design pattern organizes a system into multiple states, encapsulating the behavior and logic unique to each state within separate classes. In this scenario, there are three distinct states:

1. **Start**: This state handles the initial phase of the game, presenting options to the player like choosing difficulty, starting a new game, or loading a saved game.
2. **Almost**: This state represents a transitional phase where the game is nearing completion. It involves certain conditions and actions that occur when the game nears its ending seconds.
3. **Finish**: This state signifies the conclusion of the game. It manages actions related to game over scenarios, displaying the final score, allowing the player to restart or exit the game, etc.

Each state class (**Start, Almost, Finish**) encapsulates specific functionalities related to its respective phase of the game. The design allows for clean separation of concerns, making it easier to maintain and extend the game by modifying or adding new states without affecting the entire system.

By employing the State pattern, the game logic is compartmentalized, ensuring that the behavior of each phase is contained within its own class. This modular approach simplifies code management, improves readability, and facilitates easier modifications or enhancements to the game's different states without causing disruptions across the entire system.

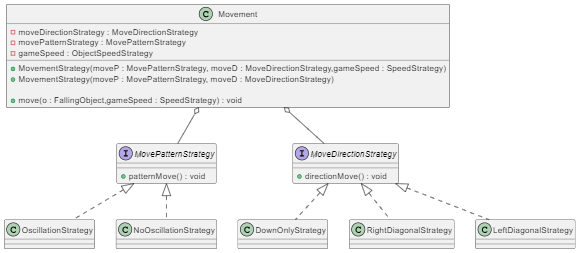
1. Strategy Pattern
2. **Difficulty Strategy**

In the **Difficulty** class, the Strategy design pattern is evident through the encapsulation of various strategies used to configure different aspects of gameplay. The class employs composition by holding references to different strategy interfaces such as **ObjectSpeedStrategy**, **ObjectsFallingStrategy**, and **Movement**, representing strategies for determining object speeds, generating falling objects, and managing movement patterns, respectively. Through the multiple constructors, the class facilitates the implementation of the **Strategy pattern** by allowing two distinct approaches to defining a game difficulty. One constructor takes a **PredefinedDifficultyStrategy**, enabling the setup of a difficulty using a predefined strategy, while the other constructor accepts custom strategies for object speed, movement, objects falling, and the rate of falling objects per second. These strategies can be interchanged dynamically during runtime, promoting flexibility and enabling the adjustment of gameplay behaviors without altering the core structure of the Difficulty class. Overall, the **Strategy pattern** in the **Difficulty** class fosters adaptability and modifiability in configuring various gameplay elements based on different strategies and parameters.



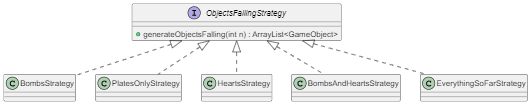
1. **Movement Strategy**

The **Movement strategy** within the system orchestrates the movement behavior of falling objects, relying on two essential sub-strategies: **MovementDirection** and **MovementPattern**. The **MovementDirection** strategy offers three distinct movement behaviors: **DownOnly**, guiding objects to fall straight down; **RightDiagonalStrategy**, prompting objects to move in a rightward and downward diagonal path; and **LeftDiagonalStrategy**, dictating objects to descend in a leftward and downward diagonal direction. Complementing these directional strategies, the **MovementPattern** strategy offers two variations: **NoOscillation**, where objects fall undisturbed in a direct vertical path, and **Oscillation**, causing objects to zigzag unpredictably between right and left while descending. This hierarchical structure of **Movement** and its dependent strategies facilitates a modular and customizable approach to managing diverse movement patterns for falling objects in the game, enabling dynamic and varied gameplay experiences.



1. **Objects Falling Strategy**

The **Objects Falling Strategy** encompasses a range of strategies, each aimed at generating specific combinations of falling objects. These strategies include **PlatesOnlyStrategy**, producing solely plates; **BombsStrategy**, which generates both bombs and plates with a 20% chance for bombs and 80% for plates; **HeartsStrategy**, exclusively yielding 10% hearts and the remaining as plates; **BombsAndHearts**, creating objects with a distribution of 10% hearts, 20% bombs, and 70% plates. Additionally, **HeartsStrategy** offers a specific 10% chance for hearts while generating the rest as plates, while **BombsAndHeartsStrategy** introduces a distribution of 10% hearts, 20% bombs, and 70% plates among the falling objects. These strategies enable diverse and customizable gameplay experiences across various difficulty settings, each offering unique combinations of falling objects to enrich the game dynamics.



1. **Intersection Handler Strategy**

The **IntersectionHandler** **Strategy** oversees the interaction between the clown's sticks and various objects within the game, employing distinct strategies for each object type. Objects like plates and cups adhere conventionally to the sticks, stacking atop one another; if three plates or cups of the same color align, they're removed, increasing the score. Bombs, upon intersection, remove a plate, reducing the score, and causing the loss of a life. Conversely, intersecting hearts grant an additional life. These strategies describe unique behaviors for each object type, enabling diverse outcomes upon intersection and enhancing the game's dynamics by rewarding strategic stacking, penalizing bomb interactions, and providing life bonuses through heart interactions.

