Course: Object Oriented Programming with C++

1. Title: Design Activity document of Open Ended Assessment

Slot Machine

2. Team members list with roll numbers

Shribhakti S V	**: 01FE22BCI048(144)**
Shruti Sutar	**: 01FE22BCI052(147)**
Sanskruti A K	**: 01FE22BCI062(156)**
Chinmay J S	**: 01FE22BCI351(165)**

3. Problem Definition (Description)

A slot machine is a gambling device that operates on a principle of randomness and chance, offering players the opportunity to win money or other rewards based on the combination of symbols displayed on its reels after a spin.Design and implement a digital slot machine simulation that replicates the experience of playing a real slot machine. The simulation should provide a graphical user interface (GUI) for interaction, simulate the random spinning of reels, determine the outcome based on predefined rules, and update the player's balance accordingly.

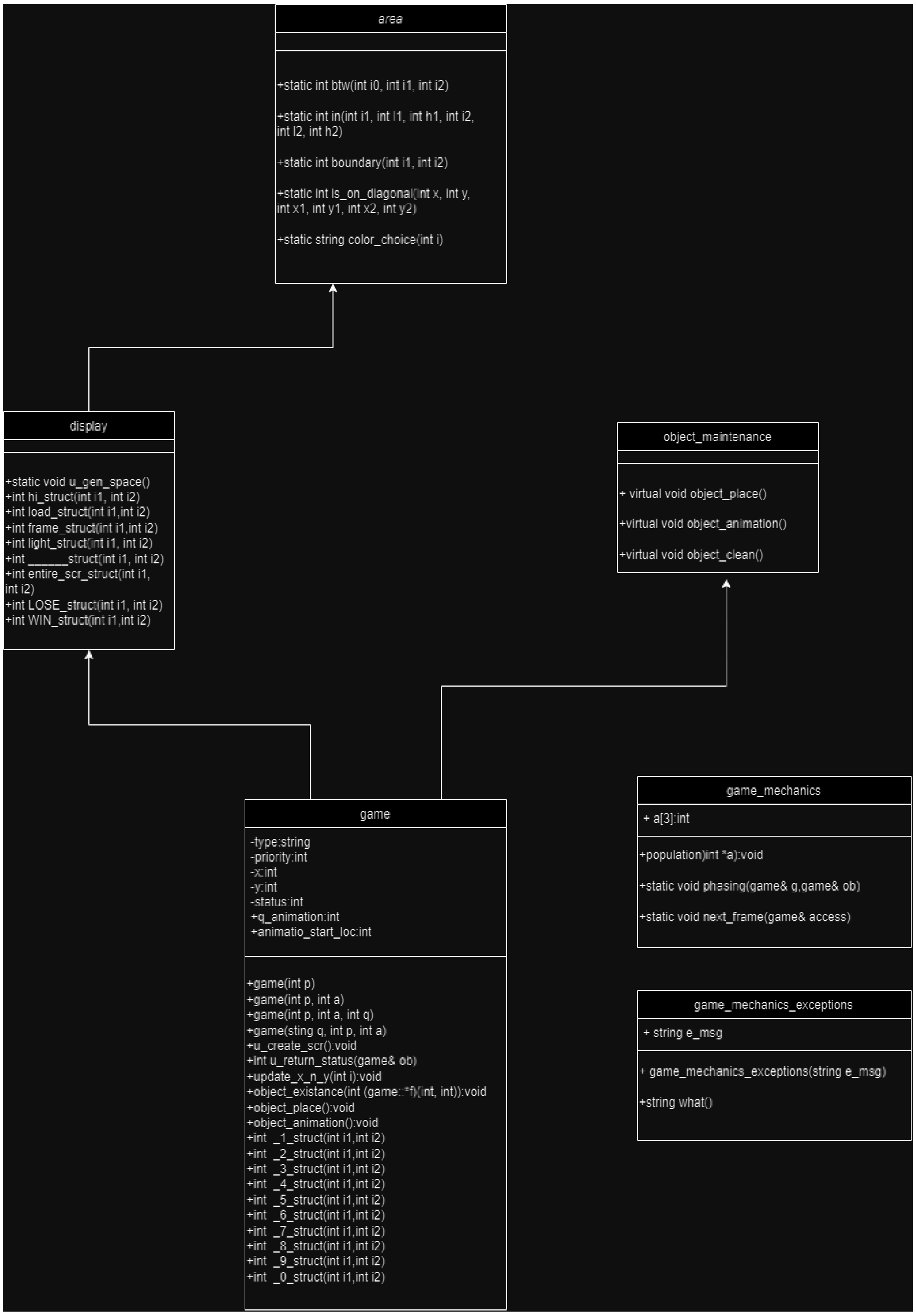
The heart of the simulation lies in its ability to mimic the random nature of a slot machine’s reels. Using complex algorithms, the simulation ensures each spin is unpredictable, providing an authentic feeling of suspense as the reels come to a gradual halt. The symbols that line up across the machine’s paylines determine the outcome, with various combinations leading to different payouts. These rules are pre-programmed into the simulation, ensuring that players have clear expectations of winning patterns.

To enhance the experience further, the simulation can include additional features such as sound effects that mimic the clinking of coins, thematic music, and animations that celebrate wins. The goal is to create an immersive experience that not only entertains but also stays true to the probabilistic nature of slot machines, all while ensuring a fair and enjoyable gaming environment.

4. List of objects identified

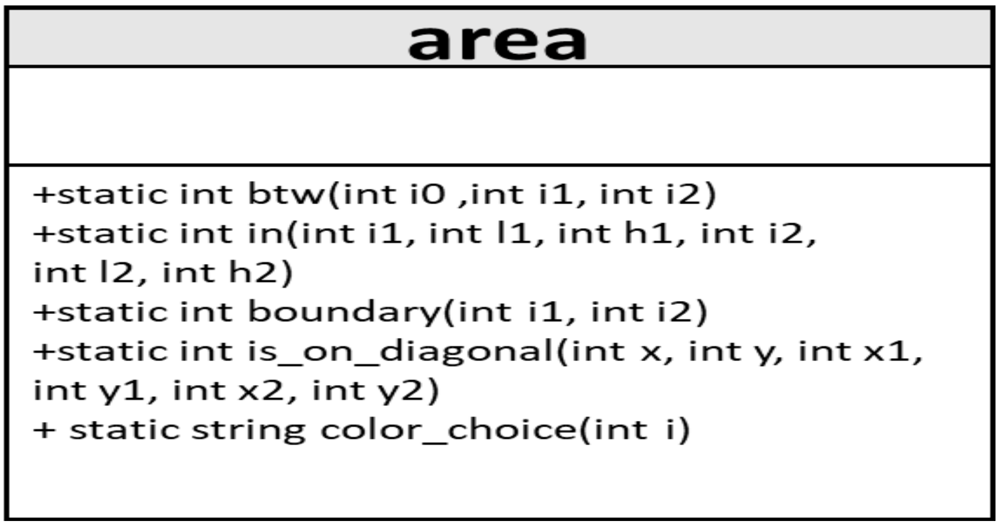
- game access
- hi
- load
- frame
- light
- blank
- num1
- num2
- num3
- num4
- num5
- num6
- num7
- num8
- num9
- num0
- endl
- endw
- loose
- win

5. Class Diagram


area
+static int btw(int i0, int i1, int i2)
+static int in(int i1, int l1, int h1, int i2, int l2, int h2)
+static int boundary(int i1, int i2)
+static int is_on_diagonal(int x, int y, int x1, int y1, int x2, int y2)
+static string color_choice(int i)
display
+static void u_gen_space()
+int hi_struct(int i1, int i2)
+int load_struct(int i1,int i2)
+int frame_struct(int i1,int i2)
+int light_struct(int i1, int i2)
+int _______struct(int i1, int i2)
+int entire_scr_struct(int i1, int i2)
+int LOSE_struct(int i1, int i2)
+int WIN_struct(int i1,int i2)
object_maintenance
+ virtual void object_place()
+virtual void object_animation()
+virtual void object_clean()
game
-type:string
-priority:int
-x:int
-y:int
-status:int
+q_animation:int
+animatio_start_loc:int
+game(int p)
+game(int p, int a)
+game(int p, int a, int q)
+game(sting q, int p, int a)
+u_create_scr():void
+int u_return_status(game& ob)
+update_x_n_y(int i):void
+object_existance(int (game::*f)(int, int)):void
+object_place():void
+object_animation():void
+int _1_struct(int i1,int i2)
+int _2_struct(int i1,int i2)
+int _3_struct(int i1,int i2)
+int _4_struct(int i1,int i2)
+int _5_struct(int i1,int i2)
+int _6_struct(int i1,int i2)
+int _7_struct(int i1,int i2)
+int _8_struct(int i1,int i2)
+int _9_struct(int i1,int i2)
+int _0_struct(int i1,int i2)
game_mechanics
+ a[3]:int
+population)int *a):void
+static void phasing(game& g,game& ob)
+static void next_frame(game& access)
game_mechanics_exceptions
+ string e_msg
+ game_mechanics_exceptions(string e_msg)
+string what()

6. Description of each class

AREA:



```
area

+static int btw(int i0 ,int i1, int i2)
+static int in(int i1, int l1, int h1, int i2,
int l2, int h2)
+static int boundary(int i1, int i2)
+static int is_on_diagonal(int x, int y, int x1,
int y1, int x2, int y2)
+ static string color_choice(int i)
```

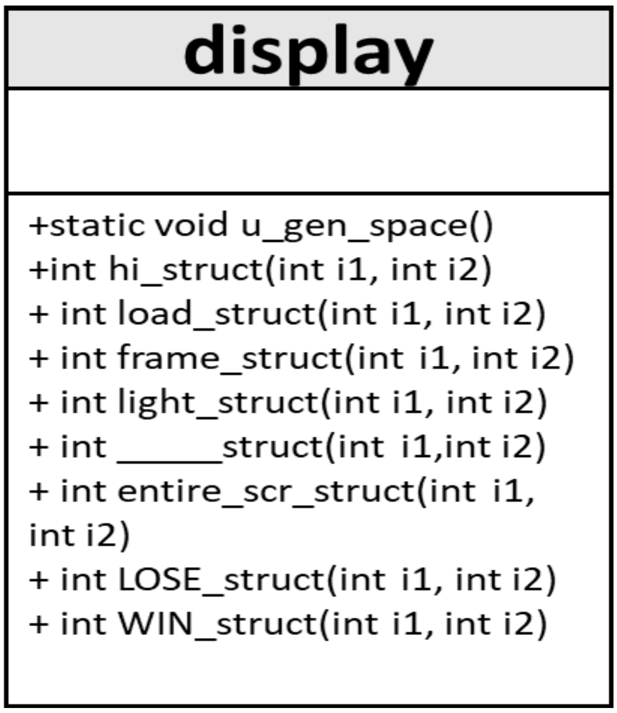
- area is a base class , which has static functions that are btw() , in() , boundary(), is_on_diagonal(), and color_choice()

The area class in the code is a collection of tools for handling geometric and visual elements in a game or application. Here's what each function does in simple terms:

- btw : Checks if a number (i0) is within a range (i1 to i2).
- in : Determines if two points (i1, i2) are within a rectangular area defined by the lower (l1, l2) and higher (h1, h2) bounds.
- boundary: Checks if a point (i1, i2) is on the boundary of a predefined rectangular area.
- is_on_diagonal: Determines if a point (x, y) lies on the diagonal line between two other points (x1, y1 and x2, y2).
- color_choice: Returns a colored block or a space character based on the input number, which is used to display different colors in the game.

In essence, this class helps manage the positions and colors of elements on the screen, which is useful for creating visual effects and interfaces in games. It's like a toolbox for drawing and placing items in a 2D space

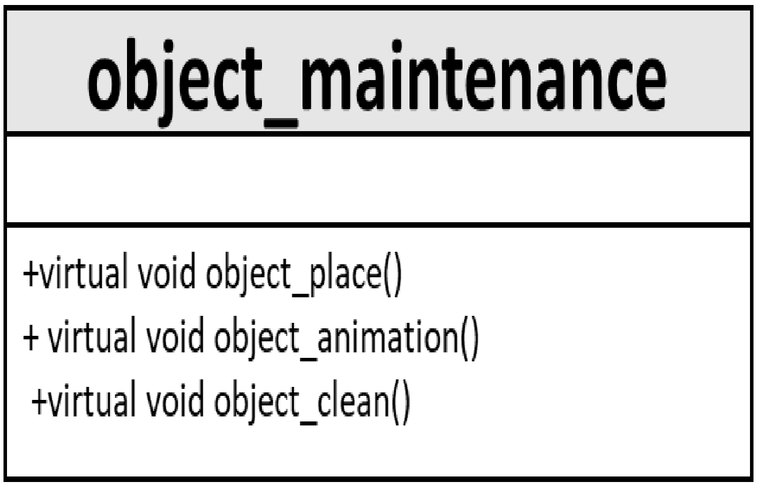
DISPLAY:



```
+static void u_gen_space()
+int hi_struct(int i1, int i2)
+ int load_struct(int i1, int i2)
+ int frame_struct(int i1, int i2)
+ int light_struct(int i1, int i2)
+ int _____struct(int i1,int i2)
+ int entire_scr_struct(int i1,
int i2)
+ int LOSE_struct(int i1, int i2)
+ int WIN_struct(int i1, int i2)
```

- display is a derived class of area
- It has static function u_gen_space and functions hi_struct() , load_struct(), frame_struct(), light_struct(), ___struct(), entire_scr_struct(), LOSE_struct() and WIN_struct()
- Inherits from area: Uses geometric functions from the area class.
- u_gen_space: Creates formatted spaces on the screen.
- u_generate_scr: Displays elements using a 2D array.
- hi_struct: Defines the “Hi” message structure.
- load_struct: Defines the “Load” message structure.
- frame_struct: Defines the “Frame” structure.
- light_struct: Defines the “Light” structure.
- _____struct: Defines a blank structure.
- entire_scr_struct: Represents the entire screen.
- LOSE_struct: Defines the “Lose” message structure.
- WIN_struct: Defines the “Win” message structure.

OBJECT_MAINTENANCE:

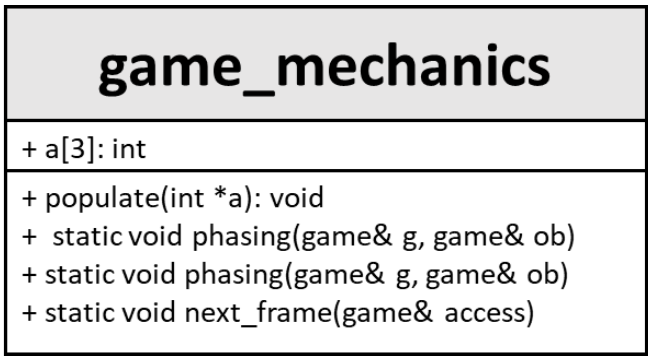


object_maintenance

+virtual void object_place()
+ virtual void object_animation()
+virtual void object_clean()

- It’s an abstract class, meaning it provides a template for other classes but cannot be instantiated on its own.
- Virtual Functions: Contains three virtual functions, which are placeholders for functionality that derived classes must provide.
- object_place: This function is meant to define how an object is placed or positioned in the game.
- object_animation: This function should handle the animation of an object, like making it move or change appearance.
- object_clean: This function is intended for cleaning up or removing an object from the screen.

GAME_MECHANICS:



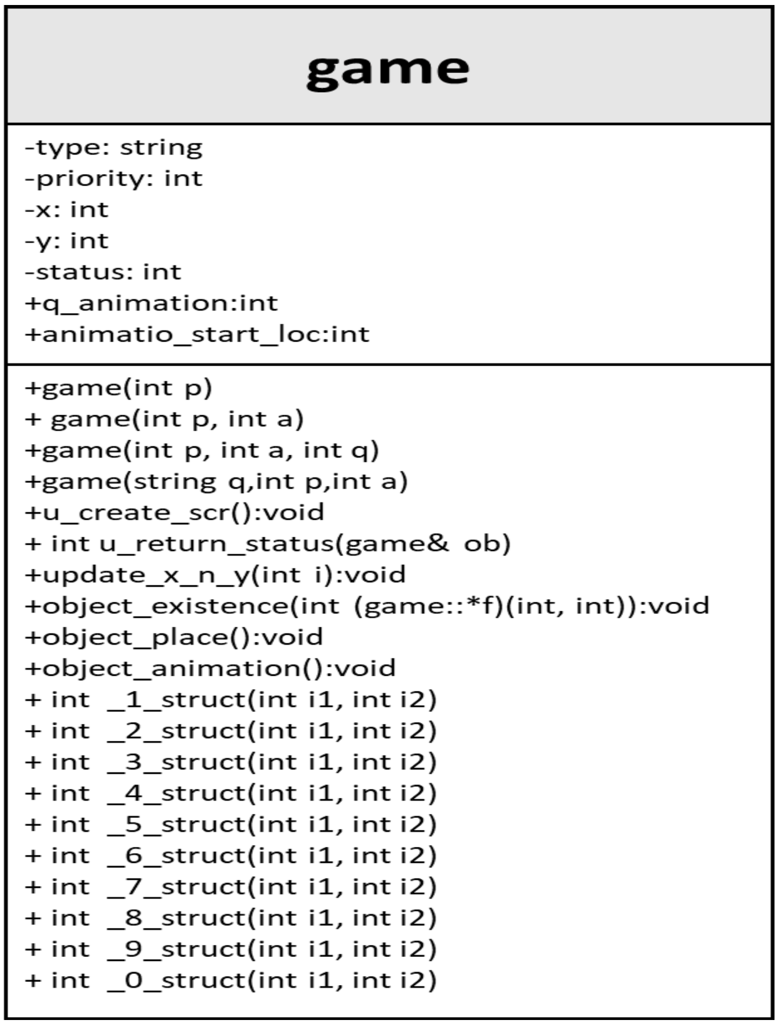
game_mechanics

+ a[3]: int

+ populate(int *a): void
+ static void phasing(game& g, game& ob)
+ static void phasing(game& g, game& ob)
+ static void next_frame(game& access)

- The game_mechanics class is responsible for the behaviour and control of game elements
- game_mechanics is a class which has static functions phasing() , next_frame() and populate()
- Array a: Holds three integers, likely used for storing game-related values like random numbers for slots.
- populate Method: Fills the array a with random numbers between 0 and 9. If the number is out of this range, it throws an exception.
- phasing Method: Manages the animation and placement of game objects. If an object has animations queued, it animates them; otherwise, it places the object directly.
- no_delay_phasing Method: Similar to phasing, but it doesn't pause between animations, making the game feel faster.
- next_frame Method: Prepares the next frame of the game, updating the screen with any changes.

GAME:



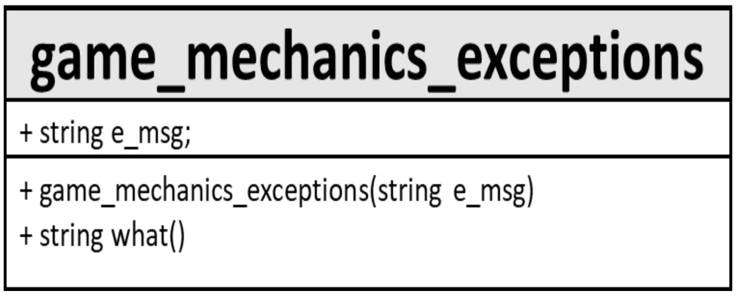
game

```
-type: string
-priority: int
-x: int
-y: int
-status: int
+q_animation:int
+animatio_start_loc:int
```

```
+game(int p)
+ game(int p, int a)
+game(int p, int a, int q)
+game(string q,int p,int a)
+u_create_scr():void
+ int u_return_status(game& ob)
+update_x_n_y(int i):void
+object_existence(int (game::*f)(int, int)):void
+object_place():void
+object_animation():void
+ int _1_struct(int i1, int i2)
+ int _2_struct(int i1, int i2)
+ int _3_struct(int i1, int i2)
+ int _4_struct(int i1, int i2)
+ int _5_struct(int i1, int i2)
+ int _6_struct(int i1, int i2)
+ int _7_struct(int i1, int i2)
+ int _8_struct(int i1, int i2)
+ int _9_struct(int i1, int i2)
+ int _0_struct(int i1, int i2)
```

- The game class is a complex part of a game's code that manages visual elements and animations
- It has functions u_create_scr(), u_return_status(), update_x_n_y(), object_existance(), object_place(), object_animation(), _1_struct(), _2_struct(), _3_struct(), _4_struct(), _5_struct(), _6_struct(), _7_struct(), _8_struct(), _9_struct() and _0_struct()
- Inherits from two classes: Inherits functionalities from both display and object_maintenance classes.
- Private Variables: Stores information like object type, priority, and position (x, y), and status.
- Public Variables: Includes variables for managing animations and a static 2D array p for the screen display.
- Constructors: Multiple constructors to initialise objects with different sets of data.
- u_create_scr: Fills the 2D array p with a default character to create a screen.
- u_return_status: Returns the status of an object.
- update_x_n_y: Updates the x and y position of an object based on an index.
- object_existence: Assigns a colour to positions in the array based on a condition.
- object_place: Places coloured objects on the screen.
- object_animation: Handles the animation of objects by moving coloured blocks across the screen.
- object_clean: Clears the screen by resetting the array a.
- Number Structures: Methods like _1_struct, _2_struct, etc., define the structure of numbers on the screen.

GAME_MECHANICS_EXCEPTION:



```
game_mechanics_exceptions
+ string e_msg;
+ game_mechanics_exceptions(string  e_msg)
+ string what()
```

- It is a class which is used for exception handling

The game_mechanics_exceptions class is like a custom alarm system for the game. It’s used to create special alerts or messages when something unexpected happens. Here’s how it works:

- Custom Error Messages: It allows the game to give specific messages about errors.
- Storing Messages: It keeps the error message ready to be shown if needed.
- Creating an Alert: When making a new alert, it saves a message that explains the problem.
- Showing the Message: It has a way to show the saved message when asked.

So, if the game runs into a problem, this class helps by saying exactly what went wrong in a way that’s easy to understand. It’s like having a helpful guide that points out mistakes so they can be fixed

7. Main Function

In the main function it starts by setting up the game environment, including random number generation for the slot outcomes. The game then creates visual elements like frames, lights, and numbers, each represented by a ‘game’ object. These objects are linked to specific game functions that define how they appear and behave on the screen.

During the game, numbers are randomly displayed in a sequence. If the same number appears three times in a row, it triggers a win condition; otherwise, the player loses. The game uses a loop to cycle through these numbers and check for a win or loss. After the loop, the game displays the result and waits for the player to acknowledge it before ending. In essence, this function is responsible for running the slot machine, handling the display of numbers, and determining the game’s outcome based on random chance. It’s a digital version of the classic casino slot machines.

8. Use of Standard Design Patterns

Factory design pattern observer pattern command pattern and singleton pattern are used for the above application factory pattern overview definition the factory pattern is a creational design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created it helps in promoting loose coupling by reducing the dependency of the client code on concrete classes usage in this context the object existence method acts as a factory method within the game mechanics class this method is responsible for creating and returning instances of different classes such as hi struct load struct and possibly other related structures the main idea is to encapsulate the object creation process and provide a common interface for creating these objects singleton pattern overview definition the singleton pattern is a creational design pattern that ensures a class has only one instance and provides a global point of access to that instance this pattern is useful when exactly one object is needed to coordinate actions across the system usage to implement the singleton pattern for the game mechanics class we need to ensure the following

1. private constructor the constructor is made private to prevent direct instantiation from outside the class
2. static instance variable a static variable is used to hold the single instance of the class
3. static instance method a public static method returns the instance of the class this method checks if the instance already exists if not it creates one observer pattern overview definition the observer pattern is a behavioural design pattern that defines a one to many relationship between objects when the state of one object the subject changes all its dependents observers are notified and updated automatically this pattern is useful for implementing distributed event handling systems usage in this scenario game mechanics can be designed as an observer that monitors the state of various game objects e g access hi load etc these game objects will act as subjects notifying game mechanics whenever their state changes command pattern overview definition the command pattern is a behavioural design pattern that encapsulates a request

as an object thereby allowing for parameterization of clients with queues requests and operations this pattern decouples the object that invokes the operation from the one that knows how to perform it the command pattern is useful for implementing undoable operations transaction management and operation queuing usage in this scenario the game mechanics class can act as the invoker and methods like phasing populate and animate can be treated as commands these methods will be encapsulated in command objects that perform specific actions on game objects receivers