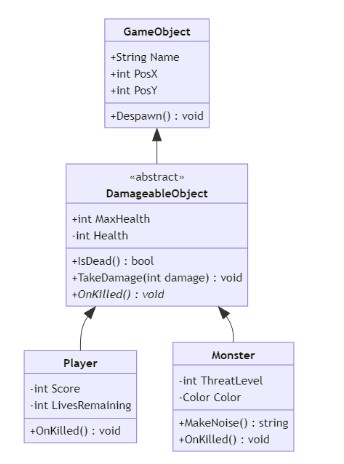
**SAFRIZAL RAHMAN\_19\_SIB\_2G**

[**https://github.com/safrizalrahman46/QUIZ\_2\_PBO**](https://github.com/safrizalrahman46/QUIZ_2_PBO) **QUIZ QUESTIONS 2**

**OBJECT-BASED PROGRAMMING PRACTICUM**

1. Identify the following Abstract method and Class usage, explain the purpose of the diagram class and create the program code to the demo to display it.



1. A client of yours is a Seller who has a lot of media to accommodate orders from customers, but this Seller has difficulty in creating Order categories, he wants every order to have an order date and there must be a confirmation method for each category which is separated into 3 classes: MailOrder, WebOrder, WhatsappOrder. There is an "order status tracking" contract on the MailOrder and WebOrder classes

Help your client by describing his diagram classes that are easy for him to understand!

1. Give an example of program code using the concept of polymorphism

(Heterogenous Collection, Object Casting, Polymorphic Arguments, InstanceOf) on 1 theme (for example, choose 1 theme: vehicle or electronic device or animal, etc... You can create any theme to apply the 4 points of polymorphism). Create interrelated java program code.

**---- Good Luck ----**

****

**1. GameObject Class**

- Purpose: This is the foundational class from which all game-related objects derive. It encapsulates common attributes and behaviors that are applicable to all game entities.

- Attributes:

- Name: A string representing the name of the game object.

- PosX: An integer or float indicating the horizontal position of the object in the game world.

- PosY: An integer or float indicating the vertical position of the object.

- Methods:

- Despawn(): This method is responsible for removing the object from the game. It might involve cleaning up resources, updating the game state, and ensuring that the object no longer interacts with other entities.

2. DamageableObject (Abstract Class)

- Purpose: This abstract class extends `GameObject` and introduces health-related properties and behaviors, making it suitable for any object that can take damage.

- Attributes:

- MaxHealth: The maximum health points that the object can have.

- Health: The current health points of the object.

- Methods:

- IsDead(): A method that checks if the current health is zero or below, indicating that the object is no longer alive.

- TakeDamage(int damage): This method reduces the object's health by a specified damage amount. If health drops to zero or below, it typically triggers death-related behavior.

- OnKilled(): An abstract method that must be implemented by subclasses. It defines what happens when the object is killed, allowing for tailored responses based on specific game mechanics.

3. Player Class

- Purpose: This class represents a player character in the game, inheriting from `DamageableObject` to incorporate health mechanics while adding player-specific attributes.

- Attributes:

- Score: An integer representing the player's accumulated score throughout gameplay.

- LivesRemaining: An integer indicating how many lives the player has left before game over.

- Methods:

- Overrides `OnKilled()`: This implementation handles player-specific behavior when the player dies, such as triggering a game-over screen, resetting scores, or providing options to restart.

4. Monster Class

- Purpose: This class represents an enemy character in the game, also inheriting from `DamageableObject`, but with its unique characteristics and behaviors.

- Attributes:

- ThreatLevel: An integer that quantifies how dangerous this monster is to players, potentially affecting AI behavior or encounter difficulty.

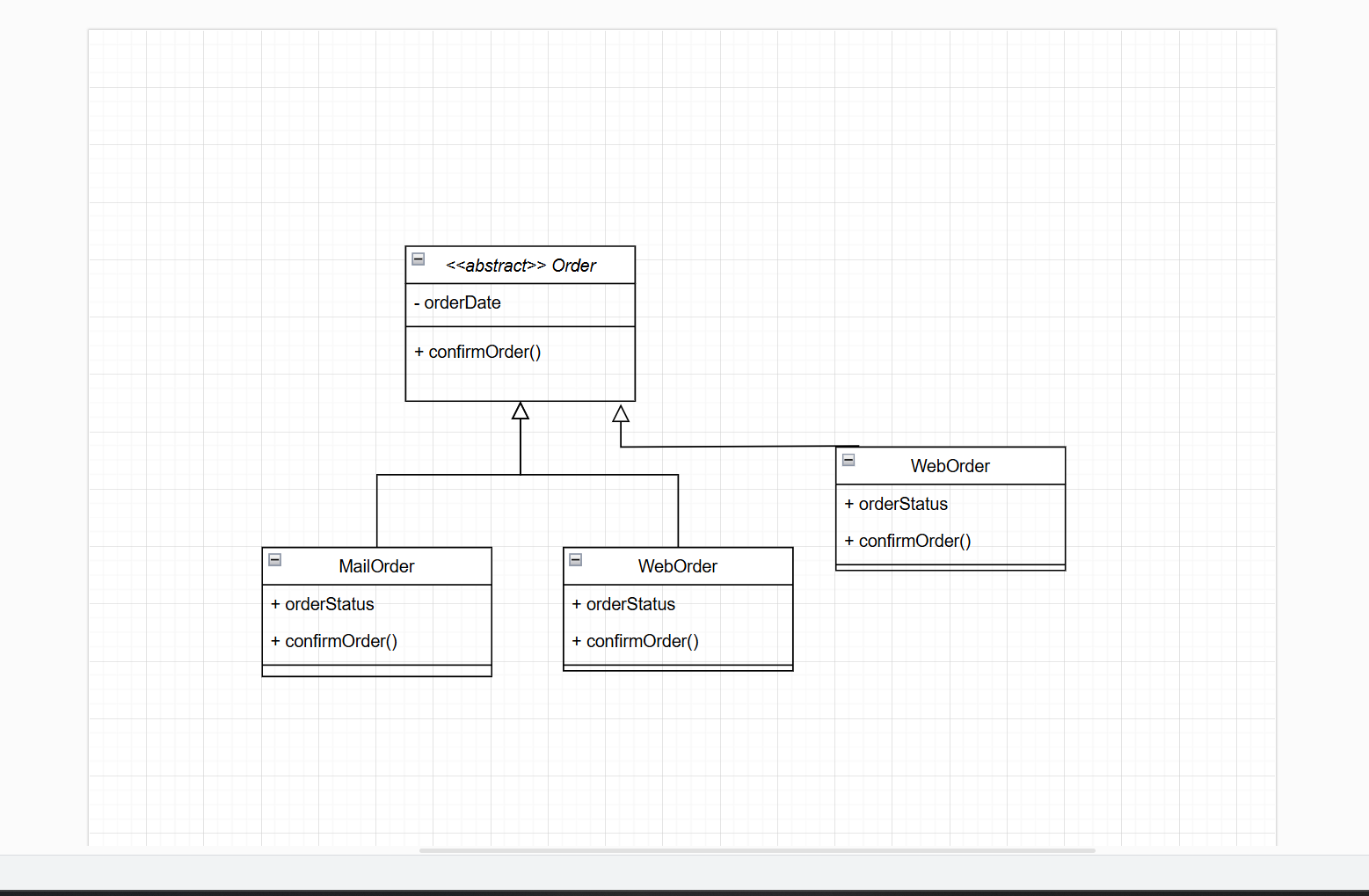
- Color: A property that may be used for visual differentiation or thematic elements within the game.

- Methods:

- MakeNoise(): A method that allows monsters to produce sounds, which could attract players or trigger specific events in-game.

- Overrides `OnKilled()`: This implementation defines what happens when a monster is killed, which could include dropping loot, spawning effects, or notifying other game systems.

This UML diagram effectively illustrates a well-organized inheritance structure within a game development context. By utilizing abstract classes and methods, it allows for flexibility and extensibility in creating diverse game objects while maintaining shared functionality. Each subclass can implement its unique behaviors while adhering to a common interface defined by their parent classes. This design promotes code reusability and easier maintenance as new features or objects are added to the game.



**2. Detailed Explanation of the UML Order Diagram**

The UML diagram you provided illustrates the class hierarchy for an ordering system that involves various types of orders. Below is a detailed explanation of each component within the diagram.

1. Order Class (Abstract Class)

- Type: Abstract class, indicated by `<<abstract>>`.

- Attributes:

- orderDate: This attribute stores the date of the order. It is essential for tracking when the order was placed and can be used for future order management.

- Methods:

- confirmOrder(): This method is responsible for confirming the order. As an abstract class, the specific implementation of this method will be defined by the subclasses inheriting from `Order`.

2. MailOrder Class

- Type: A concrete class that inherits from `Order`.

- Attributes:

- orderStatus: This attribute holds the current status of the order, such as "processing," "shipped," or "completed." It aids in tracking and managing the order's status.

- Methods:

- confirmOrder(): This class implements the `confirmOrder()` method from the `Order` class. Its specific implementation may include logic for processing orders through postal channels, including arranging shipping and notifying customers.

3. WebOrder Class

- Type: Another concrete class that also inherits from `Order`.

- Attributes:

- orderStatus: Similar to `MailOrder`, this attribute stores the order status but may have different logic or values depending on the context of online ordering.

- Methods:

- confirmOrder(): This class also implements the `confirmOrder()` method. In the context of web ordering, this implementation may involve online payment processing, sending confirmation via email, and updating status in the user interface.

4. WhatsappOrder Class

- Type: A concrete class that inherits from `Order` and is the final subclass in this hierarchy.

- Attributes:

- No additional attributes are defined in the diagram, but this class could have extra attributes if needed for the context of ordering via WhatsApp.

- Methods:

- confirmOrder(): Implements the `confirmOrder()` method in a manner suitable for orders placed through WhatsApp. This may involve interacting with WhatsApp's API to send confirmations to customers or update order status on that platform.

Relationships Between Classes

- Inheritance:

- `MailOrder`, `WebOrder`, and `WhatsappOrder` all inherit from the abstract class `Order`. This means they will all possess the attributes and methods defined in the `Order` class but can provide their specific implementations for the `confirmOrder()` method.

This UML diagram demonstrates how an ordering system can be designed using Object-Oriented Programming (OOP) principles by leveraging abstract classes to define a common contract for all types of orders. With this approach, each type of order can have its unique logic and behavior while still sharing a common foundational structure. This enhances readability, maintainability, and scalability of the system, allowing for new types of orders to be added in the future without modifying existing code.

3.