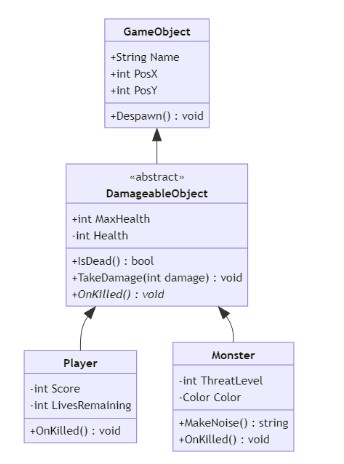
**SAFRIZAL RAHMAN\_19\_SIB\_2G**

[**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.

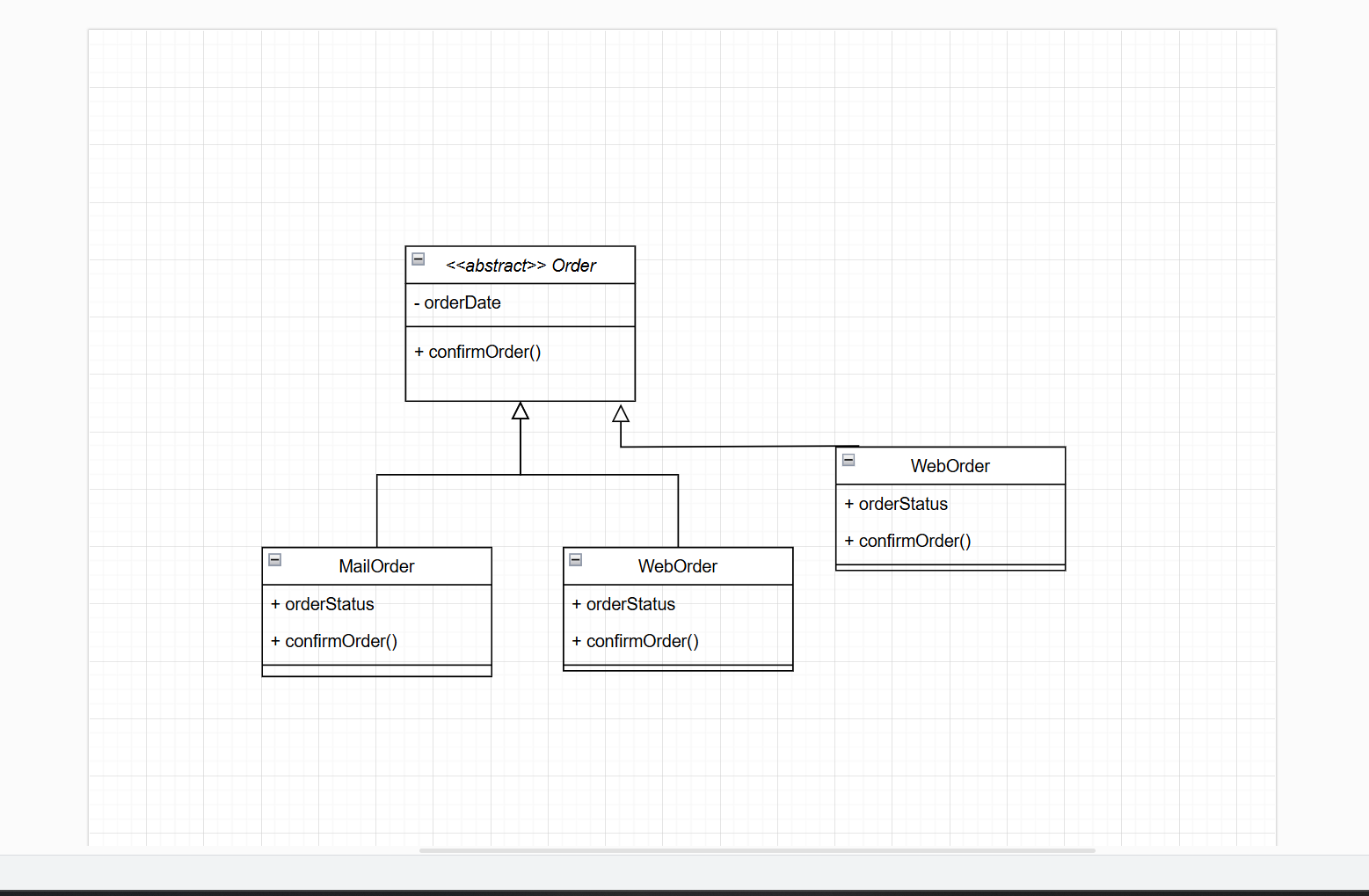
- Color: A property that may be used for visual differentiation or thematic elements within the game.

- 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.

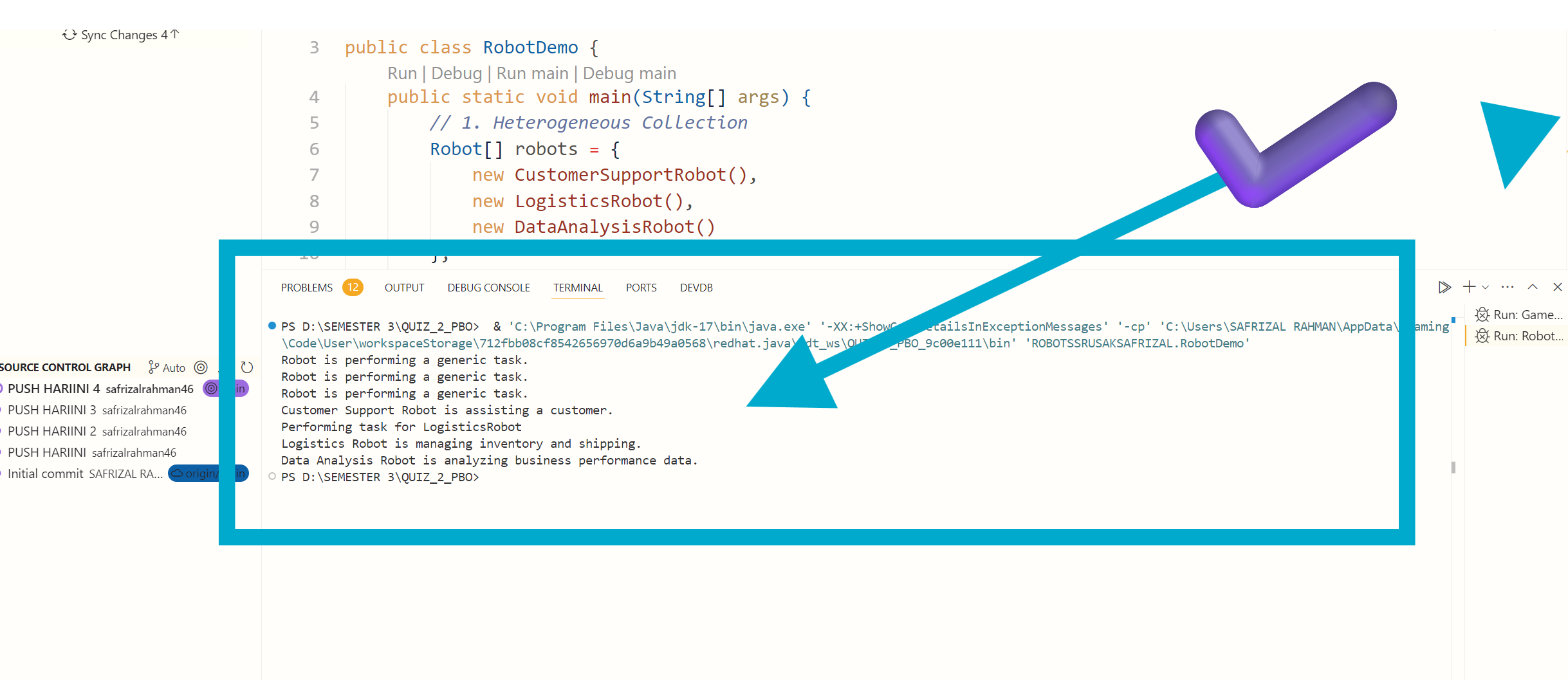
- Methods:

- 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.



3.

1. Base Class (Robot):

- Class Definition:

java

class Robot {

public void executeTask() {

System.out.println("Robot is performing a generic task.");

}

}

- The Robot class is the base class (or superclass) with a method executeTask() which prints a generic task message.

2. Subclasses (CustomerSupportRobot, LogisticsRobot, DataAnalysisRobot):

- These classes extend the Robot class, inheriting its properties and methods.

- CustomerSupportRobot:

java

class CustomerSupportRobot extends Robot {

public void assistCustomer() {

System.out.println("Customer Support Robot is assisting a customer.");

}

}

- It has an additional method assistCustomer() which prints a customer assistance message.

- LogisticsRobot:

java

class LogisticsRobot extends Robot {

public void manageInventory() {

System.out.println("Logistics Robot is managing inventory and shipping.");

}

}

- It has an additional method manageInventory() which prints an inventory management message.

- DataAnalysisRobot:

java

class DataAnalysisRobot extends Robot {

public void analyzeData() {

System.out.println("Data Analysis Robot is analyzing business performance data.");

}

}

- It has an additional method analyzeData() which prints a data analysis message.

3. Heterogeneous Collection:

- Definition:

java

Robot[] robots = {

new CustomerSupportRobot(),

new LogisticsRobot(),

new DataAnalysisRobot()

};

for (Robot robot : robots) {

robot.executeTask(); // Polymorphic method

}

- A heterogeneous collection is an array that holds objects of different subclasses of the same base class (Robot).

- The loop iterates over the array, calling the executeTask() method on each object, demonstrating polymorphism by treating each subclass as its superclass.

4. Object Casting:

- Definition:

java

CustomerSupportRobot csRobot = (CustomerSupportRobot) robots[0];

csRobot.assistCustomer();

- Object casting is converting an object of a superclass to an object of a subclass.

- This example casts the first element of the robots array to CustomerSupportRobot and calls its assistCustomer() method.

5. Polymorphic Argument:

- Definition:

java

performRobotTask(new LogisticsRobot());

public static void performRobotTask(Robot robot) {

System.out.println("Performing task for " + robot.getClass().getSimpleName());

if (robot instanceof LogisticsRobot) {

((LogisticsRobot) robot).manageInventory();

}

}

- A method (performRobotTask) is designed to accept arguments of the superclass type (Robot), allowing it to handle different subclass objects polymorphically.

- It checks if the object is an instance of LogisticsRobot and calls the specific method manageInventory().

6. InstanceOf:

- Definition:

java

for (Robot robot : robots) {

if (robot instanceof DataAnalysisRobot) {

((DataAnalysisRobot) robot).analyzeData();

}

}

- The instanceof operator checks if an object is an instance of a specific class or subclass.

- This loop checks if each robot is an instance of DataAnalysisRobot and casts it to call analyzeData().

Detailed Explanation of Concepts:

1. OOP (Object-Oriented Programming): The provided code uses OOP principles by defining classes (Robot and its subclasses) and creating objects to perform tasks.

2. Class and Object: The Robot class and its subclasses represent different types of robots. Instances of these classes (objects) are created and manipulated in the RobotDemo class.

3. Encapsulation: The classes encapsulate the properties and methods related to different types of robots. Methods like executeTask(), assistCustomer(), manageInventory(), and analyzeData() are specific to each class.

4. Relasi Kelas (Class Relationships):

- Inheritance: The subclasses (CustomerSupportRobot, LogisticsRobot, DataAnalysisRobot) inherit from the Robot base class.

- Composition: The RobotDemo class uses a composition relationship by including instances of Robot and its subclasses.

5. Inheritance: The subclasses inherit the executeTask() method from the Robot superclass and add their specific methods (assistCustomer(), manageInventory(), analyzeData()).

6. Overloading & Overriding:

- Overloading: Not explicitly shown in the provided code.

- Overriding: Not explicitly shown in the provided code since each subclass adds its own methods instead of overriding the executeTask() method.

7. Interface: Not explicitly shown in the provided code. Interfaces could be used to define common behaviors for the robots, like a TaskExecutable interface with a method executeTask().

8. Polymorphism:

- Heterogeneous Collection: Storing different subclass objects in the robots array.

- Object Casting: Casting Robot objects to specific subclasses.

- Polymorphic Argument: The performRobotTask() method accepts any Robot object.

- InstanceOf: Using instanceof to check the actual type of the object before casting.